Output-Sensitive Rendering of Detailed Animated Characters for Crowd Simulation

Alejandro Beacco$^1$, Bernhard Spanlang$^2$, Carlos Andújar$^1$, Nuria Pelechano$^1$

$^1$Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona
$^2$Dept. de Personalitat, Avaluació i Tractament Psicològic, Universitat de Barcelona, Barcelona

abacco@lsi.upc.edu, bspanlang@ub.edu, andujar@lsi.upc.edu, npelechano@lsi.upc.edu

Abstract

Rendering detailed animated characters is a major limiting factor in crowd simulation. In this paper we present a new representation for 3D animated characters which supports output-sensitive rendering. Each character is encoded through a small collection of textured boxes storing color and depth values. At runtime, each box is animated according to the rigid transformation of its associated bone. A fragment shader is used to recover the original geometry using an adapted version of relief mapping. Unlike competing output-sensitive approaches, our compact representation is able to recover high-frequency surface details and reproduces view-motion parallax. Furthermore, our approach does not require us to predefined the animation sequences nor to select a subset of discrete views. Our user study demonstrates that our approach allows for much more simulated agents with negligible visual artifacts.

1 Introduction

Real-time crowd rendering is a key ingredient in many applications, from urban planning and emergency simulation, to video games and entertainment. Crowd simulations typically require hundreds or thousands of agents, each one with its own individual behavior. Real-time rendering of detailed animated characters in such simulations is still a challenging problem in computer graphics.

Detailed characters are often represented as textured polygonal meshes which provide a high-quality representation at the expense of a high rendering cost. The animation of polygonal meshes is usually achieved through skeletal animation techniques: a set of geometric transformations are applied to the character’s skeleton, and a weighted association between the mesh vertices and the skeleton bones (skinning) defines how these transformations modify the mesh geometry. Polygonal meshes are suitable for simulations involving a relatively small number of agents, but not for large-scale crowd simulations, as the rendering cost of each animated character is roughly proportional to the complexity of its polygonal representation.

A number of techniques have been proposed to accelerate rendering of animated characters. Besides view-frustum and occlusion culling techniques, related work has focused mainly on providing level-of-detail (LOD) representations so that agents located far away from the viewpoint are rendered in a more efficient way with little or no impact on the visual quality of the resulting images. A typical approach is to store, for each animated character, a small subset of independent polygonal meshes, each one representing the character at a different level of detail. Unfortunately, most surface simplification methods are devoted to simplifying static geometry and do not work well with dynamic articulated meshes. As a consequence, the simplified versions of each character have to be created manually. Moreover, these simplified representations either retain a large number of vertices, or suffer from a substantial loss of detail, which is particularly
noticeable along character silhouettes.

Image-based precomputed impostors [7, 19, 20] provide a speed up by rendering distant characters as a textured polygon, but suffer from two major limitations: all animations cycles have to be known in advance (and thus animation blending is not supported), and resulting textures are huge (as each character must be rendered for each animation frame and view angle); otherwise characters appear pixelized.

Using separate impostors for different body parts provides a much more memory-efficient approach. Polypostors [9] subdivide each animated character into a collection of pieces, each one represented using 2D polygonal impostors. Unfortunately, the representation is view-dependent, the animation sequence still has to be known at construction time, and character decomposition is done manually.

Relief mapping has been proven to be a powerful tool to encode detailed geometry and appearance information. Most importantly, since relief maps support efficient random-access, impostors based on relief mapping are output sensitive, i.e. their rendering cost is roughly proportional to the area of their screen projection. This feature makes relief impostors especially suitable for accelerating the rendering of scenes involving a huge number of objects.

In this paper we present a new representation for animated characters (Figure 1) which uses relief impostors to represent the different body parts of the character delimited by the skeleton bones. Each character is encoded through a collection of oriented bounding boxes, each box representing the geometry influenced by a skeletal bone, along with textures projected orthogonally onto the six faces of each box, each texture storing color and depth values. During animation the bounding boxes are transformed rigidly by a vertex shader according to the transformation of the associated bone in the animated skeleton. A fragment shader efficiently recovers the details of the avatar's skin and clothing using an adapted version of relief mapping.

Unlike competing output-sensitive approaches, our compact representation has very low preprocessing requirements and does not require us to predefine the animation sequences nor to select a subset of discrete views. Our performance experiments show a significant improvement with respect to geometry rendering. We have also conducted user perception tests validating our technique for rendering agents at middle and far distances from the observer.

2 Related Work

2.1 Crowd Rendering Acceleration

Rendering a large number of highly realistic animated characters can become a major bottleneck if we render the full geometry of all
characters with animation and skinning. To achieve highly realistic populated scenes in real time several techniques have been developed. A well known solution to this problem involves applying level-of-detail for the characters depending on their distance to the camera [10]. De Heras Ciechoski et al. [5] avoid computing the deformation of a character’s mesh by storing pre-computed deformed meshes for each key-frame of animation, and then carefully sorting these meshes to take cache coherency into account.

Impostors have been suggested to avoid rendering the 3D geometry during simulation time. Aubel et al [3] described their dynamic impostors, where a multisolution virtual human was constructed to be rendered off-screen, from a view-angle and with its animated position, into a buffered texture. The texture was then mapped into a 3D polygon oriented towards the camera. This texture is only refreshed when necessary. Pre-generated impostors were first used by Tecchia et. al. [19] by rendering each character from several viewpoints and for every animation frame of a simple animation cycle. The images were stored in a single texture atlas, and each crowd agent was rendered as a single polygon with suitable texture coordinates according to the view angle and frame. Pre-generated impostors with improved shading have also been used [20]. Impostors can achieve rendering of crowds consisting of tens of thousands of agents, but require a large amount of memory and for short distances they appear pixelated.

Dobbyn et. al. [7] introduced the first hybrid system that was presented by using impostors on top of a full, geometry-based human animation system, and switching between the two representations with minimal popping artifacts. Coic et. al. [6] described a similar hybrid system but with three LODs, by introducing a volumetric layered based impostor between flat impostor and geometry to help to achieve continuity during transitions.

In order to reduce the memory requirements of impostors, while keeping a high level rendering efficiency, 2D polygonal impostors have been used [9], where an impostor is used per body part and viewing direction and then when the character is animated, dynamic programming shifts the vertices of the 2D polygon to approximate the actual rendered image as closely as possible.

Pettre et al. [13] described a three-LOD approach, combining the animation quality of dynamic meshes with the high performance offered by static meshes and impostors. A GPU acceleration crowd rendering is presented in [10], alternating the use of a single impostor per agent with pseudo-instancing of polygonal meshes.

2.2 Relief Mapping

Among image-based techniques, relief mapping [15] has proven to be useful for recovering high-frequency geometric and appearance details. Relief maps store surface details in the form of a height field. Typically the RGB channels encode a normal or a color map, while the alpha channel stores quantized depth values. The programmability of modern GPUs allows us to recover the original geometry by a simple ray-heightfield intersection algorithm executed in the fragment shader [15]. Acceleration techniques for computing the ray-heightfield intersection include, among others, linear search plus binary search refinement [15], varying sampling rates [18], precomputed distance maps [4] and cone maps [14].

A few recent techniques adopt a relief mapping approach to encode details in arbitrary 3D models with minimal supporting geometry [4, 2]. Unfortunately, these output-sensitive approaches are limited to static geometry.

Only few works attempt to animate geometry encoded as relief impostors. In [12] the animator is requested to create an animation by manually defining and moving a few control points in texture space. Radial basis functions are used to warp the original image by texture coordinate modification. The above method suffers from two major limitations: control points defining the animation are just moved in 2D, providing only image-warp animation, and it does not support standard skeletal animation.
3 Our approach

3.1 Overview

We aim at increasing the number of simulated agents in real-time crowd simulations by reducing the rendering cost of individual agents. This involves using a simple representation for animated characters supporting output-sensitive rendering, so that rendering times are roughly proportional to the number of rendered fragments, instead of depending on the complexity of the underlying surface. Therefore only characters that are very close to the observer are rendered as polygonal meshes, while the rest of the agents are rendered using our new relief impostor method.

We assume the input character conforms to the de facto standard in the video games industry and thus consists of a textured polygonal mesh (skin), a hierarchical set of bones (skeleton) and vertex weights. We assume that both the skin and the skeleton have been designed in a reference pose. The nodes of the skeleton represent joints and the edges represent the bones. Since each bone can be easily identified by its origin, we can use the term joint interchangeably. The transformations affecting joints in the hierarchy are assumed to be rigid. The vertex weights describe the amount of influence of each joint on each vertex.

In our implementation a vertex can be influenced by a maximum number of 4 bones.

Since we want to keep preprocessing and memory costs at a minimum while still supporting real-time mixing of animation sequences, we use a separate relief impostor for each animated part of the articulated character. Our representation for distant characters consists of a collection of oriented bounding boxes (OBB), one for each bone in the skeleton, along with a collection of textures projected into the OBB faces, each texture encoding color and depth values (Figure 1). The OBB will be transformed in the same way as the bones of the skeleton, giving the impression that our impostor character is animated.

Our approach differs from previous work in several aspects. First, we do not attempt to animate a single relief impostor representing a whole character, but to provide relief impostors representing an already animated character. Second, we require much less memory than competing image-based approaches which require prerendering the character for every possible animation frame for every view angle. Finally, our method provides a detailed rendering for any character, viewpoint, and animation sequence.

Our technique relies on the Halca animation library [17, 8] to draw the animated characters from which we create our impostors. Halca is a hardware accelerated library for character animation which is based on the Ca3D XML file format [1] to describe skeleton weighted meshes, animations, and materials. Our current implementation works with any animated avatar and any animation that can be exported to the Ca3D format.

3.2 Construction

The construction of our relief impostors from a given 3D character proceeds through the following steps, described in detail below:

1. Associate mesh triangles with impostors.
2. Select a suitable pose for capturing the impostors.
3. Compute the bounding boxes with the chosen pose.
4. Capture the textures of each bounding box.

We start by assigning mesh triangles with impostors, where each impostor corresponds to a joint of the articulated character. We assume that each input vertex $v_i$ is attached to joints $J_1, ..., J_n$ with weights $w = (w_1, ..., w_n)$. Now the problem is, given a triangle with vertices $v_1, v_2, v_3$, to decide which impostors the triangle will be attached to. This determines which triangles will be captured by the impostor. Since we wanted to keep preprocessing tasks at a minimum, we only tested simple, automatic solutions. One option is to distribute mesh triangles into joints, attaching each triangle to the joint with the highest influence over the triangle (measured e.g. as the sum
of the corresponding vertex weights). It turns out that this partition tends to produce visible gaps in the joint boundaries during animation, the higher the deviation with respect to the reference pose, the larger the resulting gaps. Therefore our current implementation follows the opposite approach: each triangle is attached to a bone if at least one of its vertices is influenced by the bone, regardless of the corresponding weight. Therefore some triangles (those around joints) will be attached to a variable number of impostors. Notice that the above strategy only uses vertex weights and thus is pose-independent.

The second step is to choose a suitable pose for capturing the impostors. Triangles will be captured according to the chosen pose, i.e. after mesh vertices have been blended according to the pose (we use linear blend skinning). This choice of the pose affects both the extent of the impostor’s bounding box and the captured geometry. Ideally, we should select a pose (a) minimizing the overall volume of the bounding boxes (to save memory space), and (b) representing a somewhat average pose of the animation sequence. For example, if the animation sequence shows a character walking with the arms in a rest position, it is better to capture the triangles around the shoulder with the arms in such a position rather than e.g. stretching arms out sideways. Since impostors will undergo only a rigid transformation, choosing a pose corresponding to a walking animation keyframe tends to minimize artifacts around joints. Our current implementation just picks a random pose from a walking animation sequence, rather than using the reference pose. Notice that the above choice only affects triangles influenced by multiple joints; triangles influenced by a single joint will be reconstructed in their exact position regardless of the selected pose.

Once a suitable pose has been chosen, we deform the mesh accordingly by applying linear blend skinning to the mesh vertices, i.e. the transformed vertex \( v' \) is computed as \( v' = \sum w_i M_{J_i} v \), where \( M_{J_i} \) is the rigid transformation matrix from the reference-pose of joint \( J_i \) to its actual position in the chosen posture.

The bounding box of each impostor is then computed as the axis-aligned bounding box of the (deformed) triangles attached to the impostor.

The last step is to render the deformed mesh to capture the relief maps corresponding to each one of the six faces of its bounding box. For each bounding box face, we set up an orthographic camera with its viewing direction aligned with the face’s normal vector, and then render the triangles attached to the corresponding impostor. We capture the following RGBA textures (Figure 2):

- Color map: the RGB channels encode the color, and the alpha channel encodes the minimum (front) depth value \( z_f \).
- Normal map: the RGB channels encode the normal vector, and the alpha channel encodes the maximum (back) depth value \( z_b \).

Front depth values are captured by rendering the attached triangles with the default GL_LESS depth comparison function. Likewise, back depth values are captured by clearing the depth buffer with a zero value (instead of the default unit value) and switching depth comparison to GL_GREATER. Although storing both depth values is redundant (front depth values of a face equal one minus back depth values of the opposing face), we have chosen this option to improve the locality of texture fetches during rendering.

Once a suitable pose has been chosen, we deform the mesh accordingly by applying linear blend skinning to the mesh vertices, i.e. the transformed vertex \( v' \) is computed as \( v' = \sum w_i M_{J_i} v \), where \( M_{J_i} \) is the rigid transformation matrix from the reference-pose of joint \( J_i \) to its actual position in the chosen posture.
Assuming a typical animated character for crowd simulation consists of about 40 bones, this accounts for storing \(40 \times 6 \times 2 = 480\) RGBA textures per character. This is quite reasonable, considering that competing output-sensitive approaches need to capture the character for each view angle (typically 136 discrete view directions are sampled) and for each animation frame (typically sampled at 10Hz). Using \(64 \times 64\) textures (which provides a resolution of about 1cm/texel for geometry, colors and normals), each character requires only about 7.5 MB of storage (10 MB with mipmapped textures).

Color and normal maps of each character are stored in texture arrays to avoid texture switching while rendering the instances of the same articulated character.

### 3.3 Real-time rendering

Our current prototype uses two level-of-detail representations for each character type; a textured polygonal mesh which is used for agents close to the viewpoint, and the impostor set described above for the rest of agents. We first render nearby polygonal agents (grouped by character type to minimize rendering state changes) and then the rest of the agents as impostors (again grouped by character type).

Each character is rendered through an adapted version of relief mapping over the fragments produced by the mстерization of the transformed bounding boxes. The CPU-based part of the rendering algorithm proceeds through the following steps:

1. Bind the corresponding texture arrays (color and normal maps) into different texture units, and bind also the vertex buffer object with the geometry of the bounding boxes in the pose used to capture the impostors. These steps are performed only once per character type.

2. Modify the uniform variables encoding the rigid transformation matrices \(M_1, \ldots, M_n\) of each bone corresponding to the current pose to make this information available to the shaders (this is performed once per instance).

3. Draw the bounding box associated to each bone, just to ensure that a fragment will be created for any viewing ray intersecting the underlying geometry.

The vertex shader multiplies the incoming vertices of the bounding boxes by the corresponding rigid transformation matrix so that they follow the original skeleton animation. The vertex shader also transforms the variables encoding the location and orientation of each relief map, as these will be used in the fragment shader.

The most relevant part of the rendering relies on the fragment shader, which uses the depth values stored in the A component of the color and normal maps to find the intersection \(P\) of the fragment’s viewing ray with the underlying geometry. For this particular task any ray-heightfield intersection algorithm can be adopted. Pyramidal displacement mapping [11] is particularly suitable as it guarantees finding the correct intersection on any heightfield and viewing condition. Our current prototype though is based on the relief mapping algorithm described in [15].

The fragment shader receives as input the following information:

- World space viewpoint coordinates \(E\).
- World space fragment coordinates \(C\).
- The origin \(O\) of the face, i.e. the vertex whose texture coordinates are \((0, 0)\).
- An orthonormal basis of the bounding box face, consisting of a normal vector \(n\) and two vectors \((u, v)\) aligned along the horizontal and vertical sides of the transformed face.

The viewpoint coordinates are available as a uniform variable (set only once per application frame). The fragment coordinates are encoded as a varying variable computed by interpolation of the vertices transformed by the vertex shader. The face origin and basis vectors are available through flat varying variables.

The fragment shader computes the intersection of the fragment’s viewing ray \(r = (C - E)\)
with the height field encoded by the displacement values stored in the relief map. If no intersection is found, the fragment is discarded. As in [15], we use first a linear search by sampling the ray \( r \) at regular intervals to find a ray sample inside the object, and then a binary search to find the intersection point. This allows us to retrieve the diffuse color of the fragment being processed, along with a normal vector to compute per-fragment lighting. Unlike classic relief mapping, we use two depth values \( z_f \) and \( z_b \) per texel. During the search processes, a sample along the ray with depth \( d \) is classified as interior to the object if \( z_f \leq z \leq z_b \).

4 Results
4.1 Performance tests
We first compared performance using two radical approaches: pure geometric rendering and pure impostor rendering. Polygonal meshes had between 4K and 6K triangles with 2048 \( \times \) 2048 texture atlases for color and normal values. Each impostor was represented with 53 boxes (we do not group individual finger bones in our current prototype), using 128 \( \times \) 128 array textures for color, normal and depth values. This resulted in 53 \( \times \) 6 \( \times \) 128\(^2 \) \( \times \) 8 = 41.6 MB for character type.

Figure 3 shows the results with a varying number of agents, using the same character type for all instances (a) or using ten character types (b). All times were measured on an Intel Core2 Quad Q6600 PC equipped with a GF 8800 GT, using a 1280 \( \times \) 1024 viewport.

In a typical impostor usage scenario with the (distant) agents covering a 15% of the viewport, as depicted in Figure 1(e), pure impostor rendering clearly outperforms geometry-based rendering, enabling e.g. 1,000 fully-animated agents at about 40 fps; using polygonal meshes, only about one quarter of the agents can be rendered at the same frame rate.

We also measured impostor performance on a much more stressing situation, changing the camera position such that agents covered 90% of the viewport. In this extreme case, fragment processing becomes a major bottleneck and reduces impostor performance. Therefore, performance is maximized by using polygonal meshes for very close-up agents (with a large screen projection) and relief impostors for the rest of agents. The optimal agent-to-viewpoint distance for switching from mesh rendering to impostor rendering depends on a number of factors including mesh complexity and the particular CPU/GPU configuration. We did not study the optimal switch distance from a performance point of view (this is an interesting avenue for further research); instead, we focused on the effect of the above switch distance from an image-quality point of view, as discussed in next section.

4.2 Image quality
Figure 6 shows images rendered using relief impostors. Although the images show some artifacts around joints, these are very hard to perceive in the context of a crowd simulation. We conducted a preliminary user study to evaluate our impostor-based approach in terms of
image quality. The main goal of the experiment was to evaluate whether users perceive any image quality loss when using our impostors instead of polygonal meshes, for different switch distances.

For this purpose, we rendered a crowd simulation with multiple switch distances, ranging from 0.0 (pure impostor rendering) to $\infty$ (pure geometric rendering). We produced a 25 s movie for each resulting animation, with switch distances $d \in \{0, 10 \text{ m}, 15 \text{ m}, \infty\}$, i.e. switching to impostor rendering when the viewer-to-agent distance was above $d$.

The movie with pure geometry was stacked on the top/bottom randomly. We did not force our crowd simulation engine to produce a deterministic animation for the different simulations, as this would have enabled users to compare the above/below images on a pixel-by-pixel basis; we believe that our option better represents a typical impostor usage scenario. In order to allow comparison with pure impostor geometry, the height of the camera above ground level was set so that the screen projection of each character was below $60 \times 120$ pixels.

Nine subjects aged 23-35 participated in the experiment. Users were requested to watch all the movie pairs in a random order and decide which of the two images (above/below) had better image quality, if any.

Considering all answers, in a 40% of the trials users were unable to choose the best image; in the remaining 60%, in a 29% of the trials users choose the pure geometry render and in a 31% they choose the one using impostors. We got similar percentages when considering the answers grouped by the switch distance ranges $(0, 10 - 15 \text{ m})$: the hit/fail percentages (hit means choosing the pure geometry image as the best one) where 52%/48% for $d = 0$, and 40%/60% for $d = 10 - 15 \text{ m}$. Notice for example that, when switching to impostors at $d = 10 - 15 \text{ m}$, which produces a nearly error-free image, in a 60% of the trials users choose the movie using impostors as the highest-quality one. Our best explanation for this is that, since image quality differences

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{image.png}
\caption{Image rendered with polygonal meshes (a) and our impostors (b). Image difference is shown in (c).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{image2.png}
\caption{Relief impostors are rigidly animated by transforming the vertices of the supporting bounding boxes}
\end{figure}
were very hard to notice (particularly for large switch distances, see accompanying videos), most users made a somewhat random choice. In summary, for a moderate screen projection of the individual agents, replacing polygonal geometry by impostors produces negligible visual artifacts (see Figure 5).

5 Conclusions and Future Work
We have presented a new method to accelerate the rendering of crowds by using static relief impostors on rigidly animated bounding volumes. Our method allows for real time rendering of thousands of agents. Compared to previous work where impostors were used, our method provides the advantage of being independent from both the viewing direction and the animation clips available. These two advantages offer not only important savings in terms of the memory required to store the impostors, but also that the library of animations can be increased on-the-fly without the need for capturing new impostors.

Our method works with relief impostors captured for each of the 6 faces of the bounding box associated to each bone of the skeleton. Our current prototype provides good quality rendering for those characters rendered farther away from the camera so we can combine static relief impostors with geometry rendering depending on a threshold distance. At closer distances, some artifacts appear in the relief impostor rendering, since our current solution does not take into account how the joint transformations affect the appearance of the character skin.

As future work we would like to consider the use of dynamic relief impostors, where the impostors would be dynamically animated depending on another set of textures capturing the rigging information. Rendering time could be further accelerated by calculating the intersection between the ray and the relief mapping fragment shader using a cubemapping-like projection. Cube mapping would suit nicely the geometry we are dealing with, since each body part of the character can be easily fitted with spheres and cylinders. Uploading the animation to the GPU beforehand would also reduce bandwidth requirements and would enable to perform all motion-blending computations on the GPU. Finally, we would also like to study the possibility of adapting the texture size according to the relevance of the body part, e.g., increasing resolution for the head textures w.r.t. those for the legs.

6 Acknowledgements
This research has been partially funded by the Spanish Government grant TIN2007-67982-C02-01 and the Integrated Project PRESENCIA (6th Framework FET, Contract Number 27731).

References


