UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS
MASTER IN COMPUTING

MASTER THESIS

Output-Sensitive Rendering of Detailed Animated Characters for Crowd Simulation

STUDENT: Alejandro Beacco Porres
DIRECTORS: Nuria Pelechano, Carlos Andújar

Date: June 2010

Contents

1 Introduction **1**
- 1.1 Introduction . . . 1
- 1.2 Objectives . . . 3
- 1.3 Contributions . . . 3
- 1.4 Organization . . . 5

2 State of the Art **7**
- 2.1 Character Animation and Skinning . . . 7
- 2.2 Crowd Rendering Acceleration . . . 9
- 2.3 Relief Mapping . . . 18
- 2.4 Conclusions . . . 27

3 Our Approach **31**
- 3.1 Overview . . . 31
- 3.2 Construction of the Impostors . . . 35
- 3.3 Real-Time Crowd Rendering . . . 43
- 3.4 Conclusions . . . 48

4 Experimental Results and Discussion **49**
- 4.1 Performance Tests . . . 49
- 4.2 Image Quality . . . 52
- 4.3 Discussion . . . 55

5 Conclusions and Future Work **61**
- 5.1 Conclusions . . . 61
- 5.2 Future Work . . . 62

Bibliografía **63**

A Vertex Shader **67**

B Fragment Shader **71**
- B.1 Input . . . 71
- B.2 Ray-Heightfiel Intersection Search . . . 72

B.3 Main 74

List of Figures

1.1 A crowd example 2
1.2 Overview of the presented approach 4
1.3 Crowd with about 5,000 agents, all of them rendered with our relief impostors. 4

2.1 An avatar with 4000 polygons (left) and the same reduced to 500 polygons (right) 9
2.2 5 models of the same avatar with decreasing number of polygons as they are far away. 10
2.3 An impostor 11
2.4 View direction discretization 12
2.5 Pre-generated Impostors Shadows 13
2.6 Geopostors 14
2.7 Volumetric layered based impostors rendering scheme 15
2.8 A layered impostor of a cow. 15
2.9 A polypostor animation 16
2.10 Polypostor lack of data 16
2.11 A relief texture 18
2.12 Linear search 19
2.13 Relief Mapping with one polygon 20
2.14 Relief Mapping with 6 faces of a box 21
2.15 Single depth vs. dual-depth relief textures problems 21
2.16 Dual-depth relief textures 21
2.17 Parallax Occlusion Mapping results 22
2.18 Parallax Occlusion Mapping ray intersection 23
2.19 The Safety Radius 24
2.20 Cone Step Mapping 24
2.21 ORIs (Omni-directional Relief Impostors) 25
2.22 Animated Relief Impostors control points 26

3.1 A character in a reference pose 31
3.2 Bones influence over the mesh 32
3.3 Oriented Bounding Boxes, one for each bone. 33

3.4 Collection of textures projected into the OBB faces. 33
3.5 Relief Impostors Overview 34
3.6 Construction steps of our relief impostors 35
3.7 Capturing textures problems 37
3.8 The section of a character's mesh representing a foot 37
3.9 Association of mesh triangles with impostors (Option 1) . . . 38
3.10 Association of mesh triangles with impostors (Option 2) . . . 39
3.11 Under arm zone of a mesh in a reference pose 40
3.12 A character's mesh in a walking pose (on the left) and it's under arm zone with the same pose 40
3.13 Color, normal and depth textures 42
3.14 Distance Threshold 44
3.15 Impostor parameters 47
3.16 Ray-heightfield intersection search 47

4.1 Performance results with one type of character 50
4.2 Performance results with ten types of characters 51
4.3 Crowd with about 5,000 agents, all of them rendered with our relief impostors. 51
4.4 Agents rendered with relief impostors 52
4.5 Rigidly animated relief impostors 53
4.6 Two screenshots from two of our videos used in our user study. 54
4.7 Polygonal meshes vs. our impostors 56
4.8 Different texture resolutions 57
4.9 Sampling and distance variation 58
4.10 Artifacts due to rigid animation 59

Abstract

High-quality, detailed animated characters are often represented as textured polygonal meshes. The problem with this technique is the high cost that involves rendering and animating each one of these characters. This problem has become a major limiting factor in crowd simulation. Since we want to render a huge number of characters in real-time, the purpose of this thesis is therefore to study the current existing approaches in crowd rendering to derive a novel approach.

The main limitations we have found when using impostors are (1) the big amount of memory needed to store them, which also has to be sent to the graphics card, (2) the lack of visual quality in close-up views, and (3) some visibility problems. As we wanted to overcome these limitations, and improve performance results, the found conclusions lead us to present a new representation for 3D animated characters using relief mapping, thus supporting an output-sensitive rendering.

The basic idea of our approach is to encode each character 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 in the animated skeleton. 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 effects. Furthermore, the proposed approach ensures correct visibility among different animated parts, and it does not require us to predefine the animation sequences nor to select a subset of discrete views. Finally, a user study demonstrates that our approach allows for a large number of simulated agents with negligible visual artifacts.

Chapter 1

Introduction

1.1 Introduction

In the recent years crowd simulations are becoming an important area in computer graphics. Crowd simulations typically require hundreds or thousands of agents, each one with its own individual behavior. Its rendering is a key ingredient in many applications, from urban planning and emergency simulation, to video games and entertainment. Some of these applications require not only to render realistic and detailed animated characters but also to perform well in real-time. Because of this, real-time crowd rendering 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 [26]. 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 [12, 30, 31] provide a substantial 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.

Figure 1.1: An example of a crowd simulation rendered with animated characters. Here distant agents are rendered by precomputed impostors. [12]

Using separate impostors for different body parts provides a much more memory-efficient approach. Polypostors [17] 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 [25] 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 distant objects, which are not projected in a large number of pixels. That is the reason why we could consider them as an option for rendering crowds.

1.2 Objectives

Applications visualizing crowd simulations require rendering multiple agents reacting to an environment in real-time. An important problem in this area is the trade-off between speed and realism when rendering a large number of virtual agents populating virtual environments. Therefore, the main purpose of this thesis has been to improve crowd simulation achieving a large number of realistically rendered agents in a real time crowd simulation.

In order to achieve our goal, we first studied the previous work that has been done in the field of simulation and rendering of crowds. We focused more specifically on the work based in the use of impostors applied to crowds. Impostors typically employ a minimal set of textures to replace the geometry mesh of each agen, and thus reducing the computational time required for rendering. We also studied the basis of image-based techniques for impostors, such as relief mapping and its possible acceleration algorithms.

The idea was to study the feasibility of employing relief mapping techniques combined with novel ideas to construct our own agent impostors. We wanted these impostors to be easily and automatically constructed. We also wanted our technique to be general enough so that it can be applied to any kind of character mesh, and any kind of skeletal based animation. So we have proposed a novel approach unifying those aspects and analyzed its results in terms of perception and rendering performance.

1.3 Contributions

In this work, after analyzing the state of the art, we present a new representation for animated characters (Figure 1.2) 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 (OBB), 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 by achieving a larger number of agents in crowd simulations with a higher level of realism (see Figure 1.3).

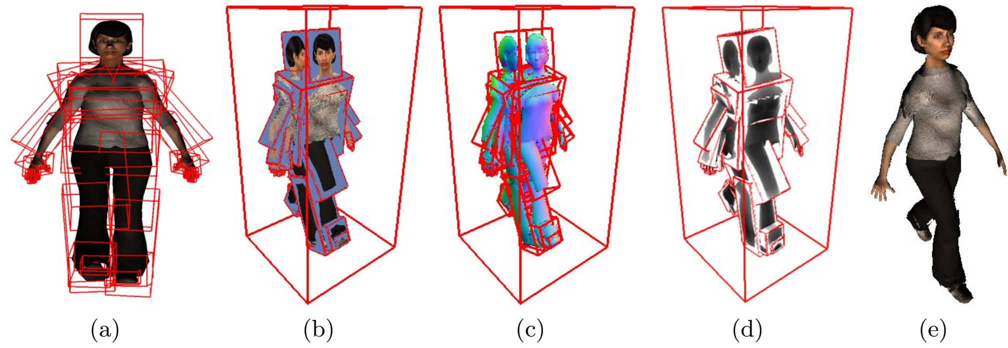


Figure 1.2: Overview of the presented approach: A bounding box is created for each articulated part of an animated character (a). Color (b), normal (c) and depth (d) information is projected onto the box faces, which are rendered through relief mapping (e).

Figure 1.3: Crowd with about 5,000 agents, all of them rendered with our relief impostors.

Finally, since we wanted our new image-based representation to offer visual results as close as possible to those obtained when rendering the full geometry of the characters, we have run a user study to validate our new approach for rendering agents at middle and far distances from the observer. This user study is to determine the extend to which a user can or cannot notice the differences between the two types of rendering.

As a result of this work, a paper has been submitted and accepted for publication in CEIG 2010 (*Congreso Español de Informática Gráfica*) [8], which will take place in September 2010 in Valencia.

1.4 Organization

The rest of the work is organized as follows. In the next chapter we discuss the state of the art on crowd rendering with impostors, relief mapping, and animation with relief impostors. In chapter 3 we make an overview of the presented solution, explain how we construct and use our relief impostors, as well as how the real time rendering and animation works. The details about the user perception tests that we have run are discussed in chapter 4 together with the results obtained. Finally, in chapter 5 we present conclusions and future work.

Chapter 2

State of the Art

In order to understand the current state of the art on crowd rendering, this chapters starts with a short summary about character animation and skinning techniques. Then we proceed with a compilation of some of the most important and most recent approaches on crowd rendering acceleration: LOD (Level Of Detail), impostors, hybrid systems and several GPU acceleration techniques. The requirements of these methods to be valid for crowd simulation are efficient rendering and animation of each agent, and a good visual quality with almost unnoticeable artifacts (if any). We therefore analyze such existing methods in terms of memory and computing cost, and seeking for any visualization problems. Finally, since relief impostors are lately being used to replace geometry, we explore and evaluate the main approaches on relief mapping, which might be a good option for crowd rendering.

2.1 Character Animation and Skinning

Before discussing the details on rendering large crowds composed of hundreds of animated agents, we should take a look on how each avatar or character representing an agent is usually animated and rendered. In this section we want to explain some general aspects of character animation which can or can not be exploited for crowd rendering. These aspects go from how a 3D character is usually represented to how this representation is modified to become animated.

In some cases we will want a physically accurate mesh deformation. For that cases there are physically based methods which, logically simulate the internal structure of the body: bones, muscles and fat tissues [5]. Those methods generally obtain a high level of realism (delivering also dynamic effects and muscle bulges), but at high computational costs. However, in the case of crowd simulations in real time, a fast method capable of animating multiple models interactively is needed.

The most extended technology for animation of 3D characters is the *skeletal animation* [20]. In this technique, most suitable for vertebrates, the character is represented by its mesh or "'skin"', and by its underlying skeleton. The skin is a 3D triangular mesh with no assumed topology or connectivity and the skeleton is a hierarchy of bones, which are carefully placed so that they fit inside the skin (both are designed in a reference pose), which will be deformed with the bones movement.

Each bone is associated with a section of the mesh. In most cases, the bone is linked to a subset of the mesh vertices. For example, all the vertices forming the left hand will be linked to the left hand bone. In several cases, a subsection of the mesh can be associated to more than one bone, by defining a weight associated to each one. Those vertices will then be deformed based on the weights of all its linked bones that are being moved. Therefore, an animation can be defined by the movement of the bones, and the associated vertices will move along the skeleton.

At some point, the weight (amount of influence) of all influencing bones must be specified. This weighting, as well as skeleton fitting, is called *rigging* and it is typically done manually. However, recently an automatic procedure has been described [7], thus simplifying the rigging process considerably.

An animation is defined by a series of *keyframes*, each one defining a different *pose* for a time t, by applying a geometric transformation to each *bone* of the *skeleton*. These geometric transformations are usually encoded in matrices, so we have one matrix per bone and per keyframe. During the animation, at a time not corresponding to any of the keyframes, the new pose is computed by interpolating the matrices of the bones between the two closest keyframes.

The standard algorithm for low-cost skinning is known by many names: *linear blend skinning, vertex blending, skeletal subspace deformation or enveloping*. It is sometimes used not only for skin deformation (as the name suggests) but also to animate other deforming elements, for example cloth, because it is considerably faster than physically based cloth simulation [11]. The basic principle is that skinning transformations are represented by the skeleton matrices, which are blended linearly in function of the applied rigging.

The direct linear combination of matrices is a troublesome way of blending transformations. This might produce artifacts in the deformed skin. Dual quaternions approaches [16] are supposed to reduce those artifacts in an elegant way, by blending quaternions.

There is an alternative straight forward technique known as *morph target animation*, where the vertices positions are stored for each frame, or for each keyframe and then interpolated between them [19, 32]. An advantage of using morph target animation over skeletal animation is that the artist

can have more control over the movements because it allows to define the individual positions of the vertices within a keyframe, rather than being constrained by skeletons. This can be useful for animating cloth, skin, and facial expressions because it can be difficult to conform those things to the bones that are required for skeletal animation.

But, its basic idea implies a large amount of memory to store animations. For that reason it is not a suitable technique for crowd rendering. Therefore we will not take it into account.

2.2 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 speed up the rendering of a geometrical scene, and to achieve highly realistic populated scenes in real time several techniques have been developed. They mainly fall into three categories: culling, geometrical level-of-detail (LOD), and image-based rendering (such as the use of impostors).

Culling algorithms are basically made to discard objects or parts of objects which are not visible. So if objects are not in the camera frustum or occluded, they are not sent to the graphics card. One drawback of occlusion culling is that it usually requires an organization of the whole geometry. When it comes to rendering crowds, where agents are in constant movement, it would be difficult to apply such technique.

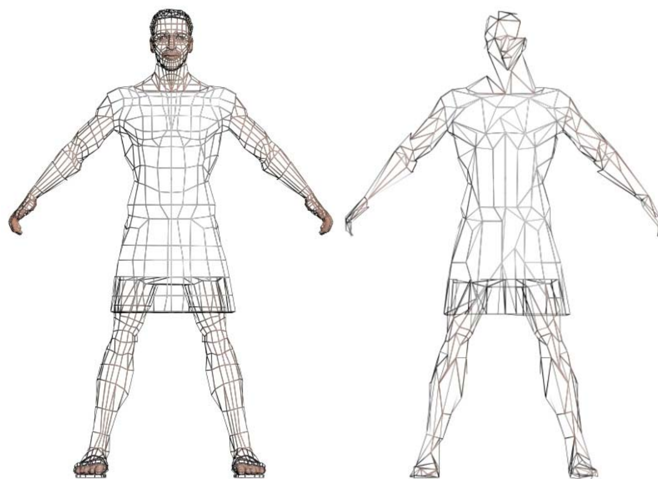


Figure 2.1: An avatar with 4000 polygons (left) and the same reduced to 500 polygons (right)

2.2.1 Level-Of-Detail (LOD)

A well known solution to the problem of accelerating crowd rendering involves applying level-of-detail (LOD) for the characters depending on their distance to the camera [26]. LOD usually consists on decreasing the complexity of the 3D object or avatar representation as it moves away from the camera (see figure 2.1). Since it is further away, the viewer should not notice any difference (see figure 2.2).

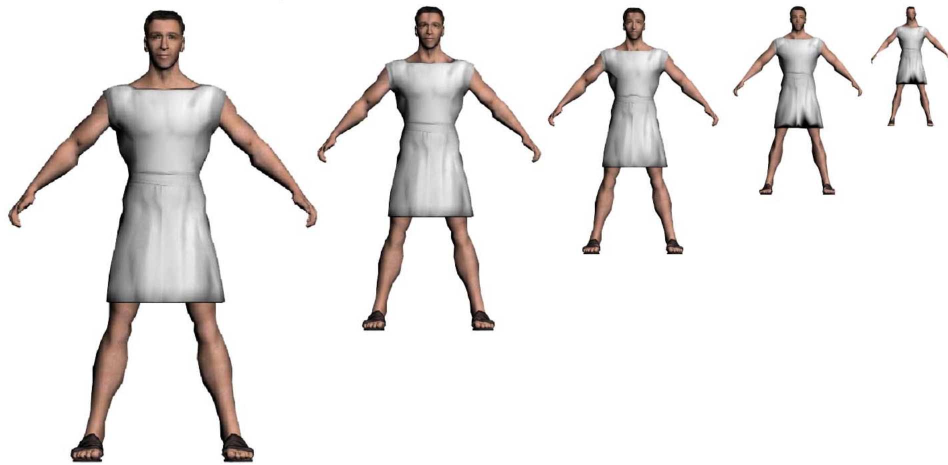


Figure 2.2: 5 models of the same avatar with decreasing number of polygons as they are far away.

The main problem of using LOD is related to the problem of multi-resolution modeling. The automatic generation of a simplified model that bear as strong a resemblance as possible to the original object has never been an easy problem, because removing too much detail can produce blocky results. Moreover the fact that we might apply LOD on animated characters, it can also introduce animation artifacts, due basically to the loss of joint vertices.

De Heras Ciechomski et al. [9] 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.

2.2.2 Impostors

One of the most known techniques suggested to avoid rendering 3D geometry, is the use of Impostors during simulation time. An Impostor is in essence something simple that has the capacity to fool the viewer.

As opposed to LOD, Impostors are not a reduced complexity version of the original geometry, but a different entity made to replace it with its

appearance. As Impostors we can find from simple billboards (3D sprites) with an image of the rendered object, to a minimal set of textured polygons retrieving surface details and parallax. Although impostors are easier to be applied with static objects, there is several work in the literature where they have been applied to crowds and animated agents.

Since Impostors are essentially images, there are two main approaches: to generate dynamically those images at runtime, or to pre-compute and store them into a texture atlas and access them when necessary.

Dynamic Impostors

The virtual human impostor used by Aubel et al [4] is a simple textured plane which rotates to face continuously the viewer. A snapshot of the virtual human is mapped onto it and re-used over several frames, so the geometry complexity is reduced to a single plane (see figure 2.3).

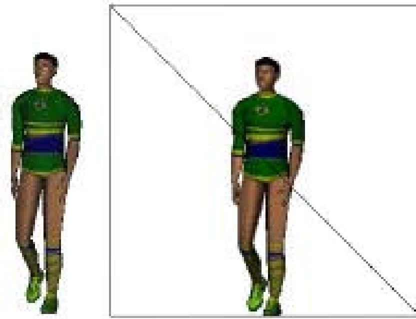


Figure 2.3: A brazilian football player and its impostor. [4]

As the humanoid moves or the camera moves, the mapped texture might need to be refreshed. To take updated snapshots they set-up an off-screen buffer to receive it and they place their multiresolution virtual human in front of the camera in the right posture. Then they render it and copy into texture memory, ready to be mapped onto the billboard.

To decide whether to refresh or not the render of the texture they proposed two fast algorithms. The first one tests distance variations between some pre-selected points in the skeleton, so they can decide if the posture has changed significantly. In a way this sub-samples the motion.

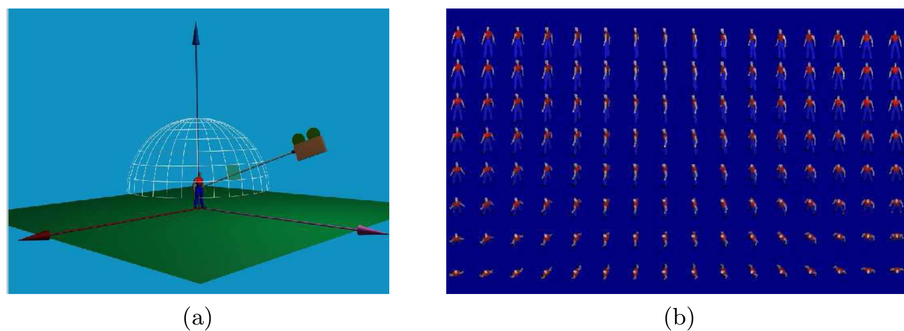
The second algorithm does not test independently the camera motion and the character's orientation because it is not important to know what factor caused the visual variation. Instead, they test the variations of the "view" matrix corresponding to the transformation under which the viewer sees the virtual human. These impostors are dynamic in the sense that they are not pre-computed, but that they change dynamically depending on the results of these two algorithms at every frame.

The off-screen buffer can be set up in a pre-process, adjusting the frustum to the character. It can also be re-used for other human meshes. Because of that, and since posing up the character would have been done with the whole geometry, the approach is not slower than rendering the 3D geometry. But even if the impostor is re-used, after a few frames it will finally be discarded.

The main limitation of these kind of approaches is that replacing the whole geometry by a textured plane might introduce visibility problems. For example, depending on what the character is doing and how he is interacting with the rest of the scene, occlusion problems could arise.

Pre-generated Impostors

Pre-generated impostors were first used by Tecchia et. al. [30] by rendering each character from several viewpoints and for every animation frame of a simple animation cycle (see Figure 2.4). 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.



(a) (b)

Figure 2.4: Discretising the view direction between the object and the view-point (a) allows to generate a texture with all the captured directions for one frame of one animation (b). The process can be repeated for every animation frame. [30]

Pre-generated impostors with improved shading have also been used in [31]. They add shadows for each agent. Since the shadow is just the projection onto the ground of the character's silhouette, they can just project the polygon of the impostor onto the ground. Then they map on the ground polygon the same image than on the impostor polygon, but appropriately darkened to reflect the shadow coverage (see Figure 2.5). This fake shadow is valid only on a ground with a parallel light source but give realistic results in the available computing time.

Pre-generated impostors can achieve rendering of crowds consisting of tens of thousands of agents. But, although image and texture compression techniques can be applied to these resulting texture atlas, they require a

(a) (b)

Figure 2.5: A scene with shadowing pre-generated impostors (a). The shadow of each agent is the projection onto the ground of the character's silhouette (b). [31]

large amount of memory. Moreover, depending on the texture resolution, at short distances they appear pixelated. Another limitation of impostors is that they do not allow interpolation or blending between two or more different animations, since they just use their pre-computed images.

2.2.3 Hybrid Systems

Hybrid approaches are the ones which combine more than one character rendering technique for different groups of agents in the same scene. It is the same concept as the LOD one, but instead of using different LODs of geometry, they use different approaches. For example, depending on the distance to the camera, an agent is rather rendered as pure 3D geometry, or rendered with an Impostor.

Geopostors

Dobbyn et. al. [12] introduced the first hybrid system, known as Geopostors, that was presented by using pre-generated impostors on top of a full, geometry-based human animation system, and switching between the two representations with minimal popping artifacts. Figure 2.6 shows how impostors are used for far agents while the ones close to the camera are rendered with full geometry.

The switching between the mesh and the impostor is based on an impostor image pixel size to impostor texel size ratio, and it is made when this ratio equals a certain threshold. Ideally this ratio should be 1:1, because aliasing starts when a texel is bigger than a pixel.

An extension of this approach was made by Pettre et al. [23], combining the animation quality of dynamic meshes with impostors and adding a third

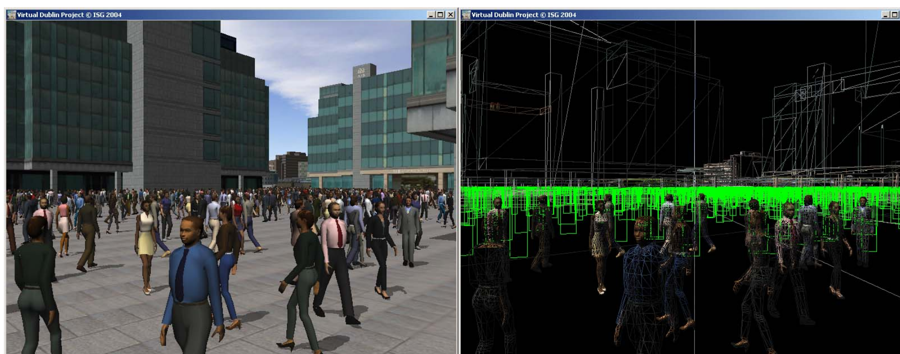


Figure 2.6: Geopostors. Far agents are rendered with impostors while closer ones are rendered with geometry. [12]

LOD using the high performance offered by static meshes, i.e. meshes where animated poses were already computed.

Volumetric layered based impostor

In geopostors, when switching and for some angles of view, the visual gap between flat impostor and geometry is too big to completely avoid popping artifacts. Coic et. al, [10] 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: Instead of one single textured polygon, an adaptive number of layers of the color texture are drawn, depending on the texel's depth. These layers fill a volume in the 3D scene instead of a single polygon.

At an intermediate LODs, they extended the single-polygon fixed-shaded impostors by their dynamically-shaded layered impostors approach (see Figure 2.7), thus enhancing the texture information with depth and normal. Depth information helps to give a 3D aspect to the impostor and the normal is used to compute a per pixel dynamic lighting.

The impostor is rendered in layers parallel to the viewpoint. For each of these layers, they select the pixels that correspond to a certain depth. After selecting the required number of layers, they divide the volume captured during the preprocessing in as many intervals as the number of layers, defining the intervals of depth for selecting pixels in the color texture. By this mean, they reconstruct the initial volume with several layers. The selection of the right pixels for each depth interval is done in a fragment shader, where lighting is also computed.

To extend the validity of the layered impostors, they can use the overlapping depth intervals. Without overlapping, cracks appear on the layered impostor as soon as the viewpoint differ a little bit from the pre-computed one. By drawing a small part of the previous and next layer, they avoid

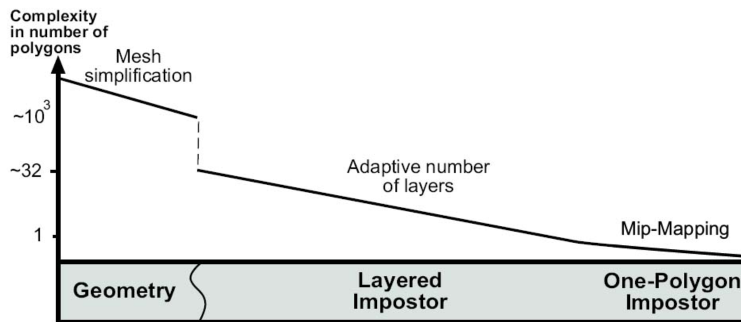


Figure 2.7: Volumetric layered based impostors rendering scheme: between geometry and the one-polygon impostor, an adaptive number of layers is used for a layered impostor. [10]

these gaps from more farther viewpoints, extending the lifetime of the layered impostor and decreasing the density of precomputed views (see Figure 2.8).

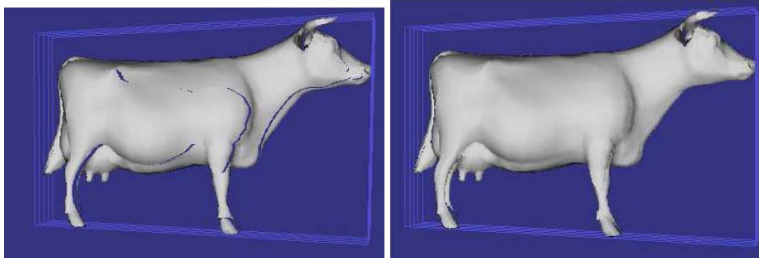


Figure 2.8: A cow rendered with 5 layers and dynamic lighting, without (left) and with overlapping (right). [10]

Although this approach improves visual quality and fills the gap between the polygonal representation and the flat impostors, they add more data (normal, depth and several layers), which grows the memory problem. Their layered impostors rendering is also slower than the one-polygon impostors rendering.

Polypostors

In order to reduce the memory requirements of pre-generated impostors, while keeping a high level rendering efficiency, 2D polygonal impostors (called polypostors) have been used [17], where an impostor is used per body part and viewing direction, thus avoiding the per frame memory consumption.

The original 3D character is cut into several body parts in order to resolve occlusion issues. The original skeletal animation is applied to the body parts. This, when composed together, gives exactly the same animation

as the originally provided. For the first frame of the animation, each body part is rendered and enclosed within textured 2D polygons, using a standard contour tracing algorithm.

For all subsequent frames, an algorithm based on dynamic programming shifts the vertices of the 2D polygons so that they approximate the actual rendered image as closely as possible (see Figure 2.9). This algorithm matches two textured polygons in an optimal way with respect to a chosen error metric.

At run-time, the deformed polygons are composited in depth order, creating the illusion of an animated 3D character. But since polypostors approximate the animation by deforming their texture, they are not as accurate as other impostors. They can only be applicable for animations that can be described as deformations of the initial key-frame. They may also produce artifacts with views where there is a lack of texture information in the first key-frame (see Figure 2.10).

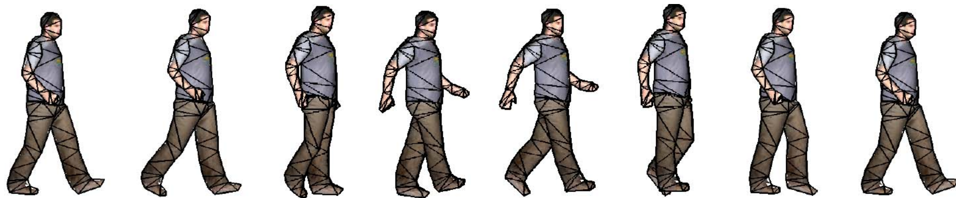


Figure 2.9: An example of a polypostor animation (overlaid with wireframe). Note that the character animation is created simply by displacing polygon vertices (stretching the texture accordingly). [17]

Figure 2.10: The Polypostor texture is generated from the fist key-frame (left) and deformed for subsequents key-frames, producing artifacts due to lack of data in areas that have become visible (right). [17]

2.2.4 Pseudo Instancing

To render nearby characters, instancing is a technique that uses certain attributes in current graphics hardware to optimize rendering of several copies of an object using a single draw call. Through instancing, graphics processor deals with per-instance geometry transformations and appearance modifications, releasing the main processor from this task. A GPU acceleration crowd rendering is presented in [21], alternating the use of a single impostor per agent with pseudo-instancing of polygonal meshes.

Even when instancing is originally designed for static objects, a similar technique may be used to render large crowds of animated characters. To achieve this goal, a pseudo-instancing technique is used, where geometry is updated on every animation frame and sent to the graphics memory to be used later for rendering nearby characters. Pseudo-instancing takes advantage on the efficiency of using persistent vertex attributes, such as color or transformations, to provide information for an entire instance.

However, this model update implies copying information into graphics memory. Hence, to maximize the outcome of this technique, several copies of the same object must be rendered in every frame. This is a problem when using animated models, since every different animation pose needs to be sent to the graphics memory. As a workaround, a few poses can be selected, and nearby characters are rendered using the closest pose to the ones selected.

2.2.5 Dynamic Caching

Recently, Lister et. al. [18] improved the efficiency of linear-blend skinning by utilizing the temporal and intracrowd coherencies that are inherent within populated scenes. They achieved it through the allocation of a small geometry cache within which transformed key-poses can be stored. This key-poses are then re-used by multi-pass rendering, between multiple agents and across multiple frames.

The cache of skinned key-posed is a maintained fixed-sized cache, from which crowd members can be reconstructed by interpolation. So, crucially, the generic poses may be shared amongst crowd members to significantly reduce the number that must be stored.

This cache size becomes also a trade off between the rendering performance and the memory usage, because it is the number of characters that have the key-poses from which to interpolate stored in the cache that will have the greatest effect on the rendering performance. Clearly, the choice of which key-poses to store is fundamental to maximizing the potential of the approach. Since this is a NP-hard problem, they present a greedy algorithm suitable for real-time applications.

2.3 Relief Mapping

During the last decades a number of image-based representations (including plain textures and depth textures) and rendering techniques (including bump mapping, parallax mapping and relief mapping) have been proposed for the efficient visualization of 3D models [15]. Textures naturally provide random-access to its elements, making these techniques output-sensitive, their rendering time being proportional to the screen projection of the model rather than to the model's complexity. Moreover, these techniques fit well in current hardware architectures and can be implemented as fragment shaders.

Among image-based techniques, relief mapping by Policarpo and Oliveira [25] has proven to be useful for recovering high-frequency geometric and appearance details.

2.3.1 Relief Maps

Relief maps store surface details in RGBA textures in the form of a height field, or, more precisely, a depth map, because the stored values represent depth measured under a reference plane. Typically the RGB channels encode a normal or a color map, while the alpha channel stores quantized depth values, scaled to the $[0, 1]$ range (see figure 2.11).

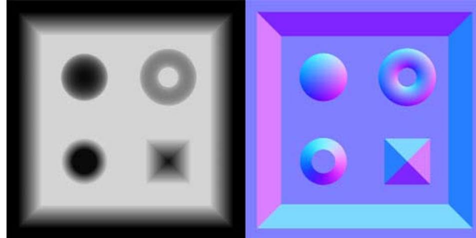


Figure 2.11: A relief texture with depth map (left) and normal map (right). On the depth map, brighter pixels represent deeper geometry. [25]

The mapping of relief details to a polygonal model is done in the conventional way, by assigning a pair of texture coordinates to each vertex of the model. During rendering, the depth map can be dynamically rescaled to achieve different effects, and correct occlusion is achieved by properly updating the depth buffer.

2.3.2 Relief Rendering

The programmability of modern GPUs allows us to recover the original geometry by a simple ray-heightfield intersection algorithm executed in the fragment shader [25]. Conceptually we can divide relief rendering in three steps:

1. For each fragment f with some texture coordinates (s, t), we transform the view direction to the tangent space of f.

2. Find the intersection P of the transformed viewing ray against the depth map. Let (k, l) be the texture coordinates of such intersection point.

3. Use the corresponding position of P, expressed in camera space, and the color and/or normal stored at (k, l) to shade f.

In order to perform the second step, a linear search is performed along the view ray to find a first point inside the surface (see figure 2.12). For each search step, a texture access is made to know the depth value at the actual point and whether the point is inside the surface or not (that is, whether the depth along the viewing ray is bigger than the stored depth). The linear search is followed by a binary one in order to refine the search and find a more accurate intersection point.

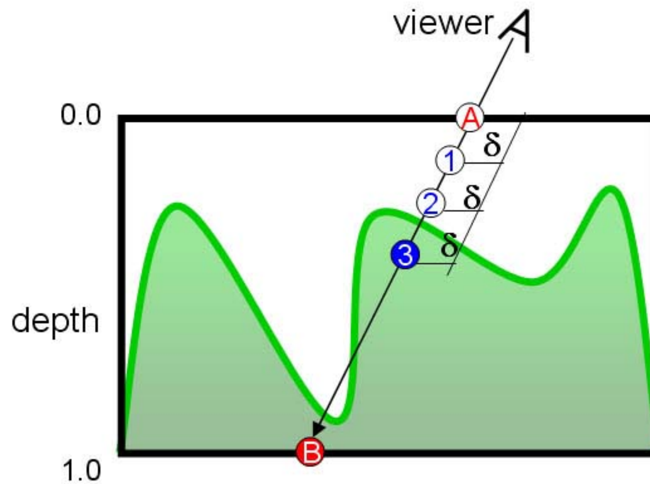


Figure 2.12: Linear search, from A to B, for the first point inside the hight-field surface. [25]

2.3.3 One vs. Several Polygons

Relief mapping can be applied to just one simple polygon (see Figure 2.13). This gives good results as long as the polygon is not seen at a too much grazing angle. In those cases, the flattening of the polygon will be noticeable.

But relief mapping can also be used by mapping multiple depth maps onto more than one polygon. For example, we can use the 6 faces of a bounding box to map the corresponding 6 relief maps (see Figure 2.14). This avoids the previous problem and preserves the parallax effect, because

Figure 2.13: Color image rendered as a conventional texture (left) and relief texture mapped on one polygon (right). [25]

all the geometry is completely inside the bounding box and thus can be retrieved with the 6 faces.

2.3.4 Multiple Depth Layers

Extending this idea, relief maps can have multiple layers allowing to store more than one depth map. Dual-depth relief textures [25], for example, can be used to produce representations for closed-surface objects using only one relief-mapped polygon. They store and combine a front and back depth layer to produce tight bounds of the represented object. This avoids the presence of artifacts, like "‘skins"‘ (see Figure 2.15) which appear with one single layer since there is no information about what lays behind the object (see Figure 2.16).

2.3.5 Ray-Heightfield Intersection Acceleration Techniques

Acceleration techniques for computing the ray-heightfield intersection include, among others, linear search plus binary search refinement [25] (which we have already commented), varying sampling rates [29], precomputed distance maps [6] and cone maps [13, 24]. Szirmay-Kalos et. al, in [28], review all this particular GPU based methods, as well as their numerous variations, providing also implementation details of the shader programs.

Parallax Occlusion Mapping

The previously seen linear and binary search helps approximating the heightfield intersection. Using bilinear texture filtering to interpolate the intersection point can result in visible stair-stepping artifacts at steep viewing

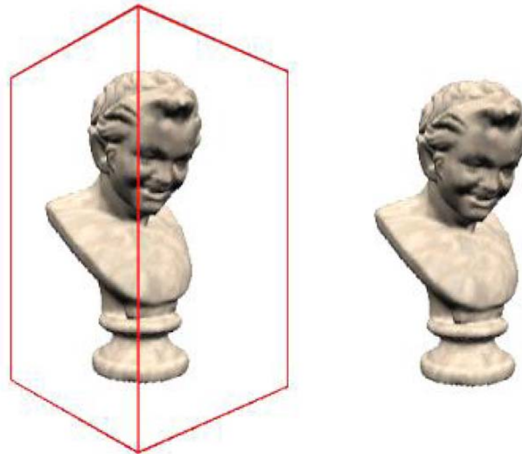


Figure 2.14: A 3D object rendered by relief texture-mapping the visible faces of a box. Object rendered showing the borders of the quads (left) and without borders (right). [25]

Figure 2.15: A single depth relief texture may produce skins (left), while dual-depth one avoid them with a tighter bound of the object (right). [25]

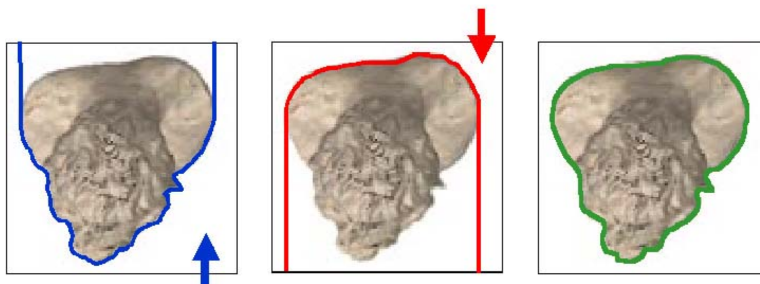


Figure 2.16: Dual-depth relief textures combine front and back depth layers and produce a better bound of the object surface. [25]

angles, since just 8 bits of precision are used (see Figure 2.17 on the left). Depth biasing toward the horizon hides these artifacts but introduces excessive feature flattening at oblique angles (see Figure 2.17 in the middle). In order to reduce these artifacts, Tatarchuk proposed a method in [29] for increased precision of the critical ray-heightfield intersection and adaptive heightfield sampling.

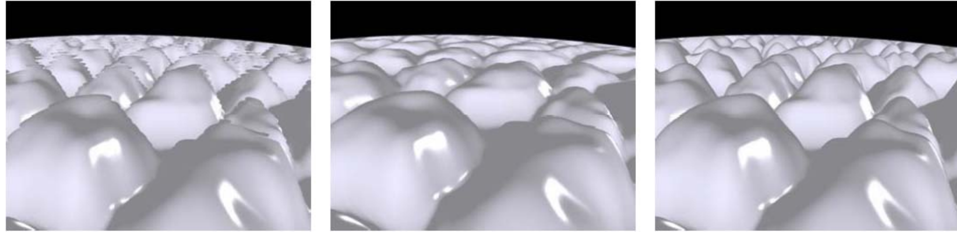


Figure 2.17: Relief mapping rendered with both linear and binary search but without depth bias applied, can produce artifacts due to sampling aliasing at grazing angles (on the left). With depth bias applied, we can still notice the flattening of surface features towards the horizon (in the middle). Parallax occlusion mapping avoids those artifacts (on the right). [29]

The idea is to sample the heightfield with a linear search and approximating the height profile as a piecewise linear curve (see Figure 2.18). This allows us to combine the 8 bit precision due to bilinear texture filtering with the full 32 bit precision for root finding during the line intersection. Figure 2.17 shows on the right the improved visual results with the lack of aliasing using this approach.

The accuracy of the technique corresponds to the sampling interval δ. So some aliasing artifacts appear, as in previous methods, if too few samples are used for a relatively high-frequency height field, though the amount will differ between the techniques. But δ can not be determined using the texture resolution. At grazing angles, the parallax amount is quite large and thus we must march along a long parallax offset vector in order to arrive at the actual displaced point. In that case, the step size is frequently much larger than a texel, and thus unrelated to the texture resolution. To solve this, they provide both directable and automatic controls, and they express the sampling rate as a linear function of the angle between the normal and the view direction ray. This assures that they increase the sampling rate along the steep viewing angles.

Precomputed distance maps

Baboud et. al proposed precomputed distance maps in [6]. They analyze the heightfield in a preprocess step and store extra information in the depth texture.

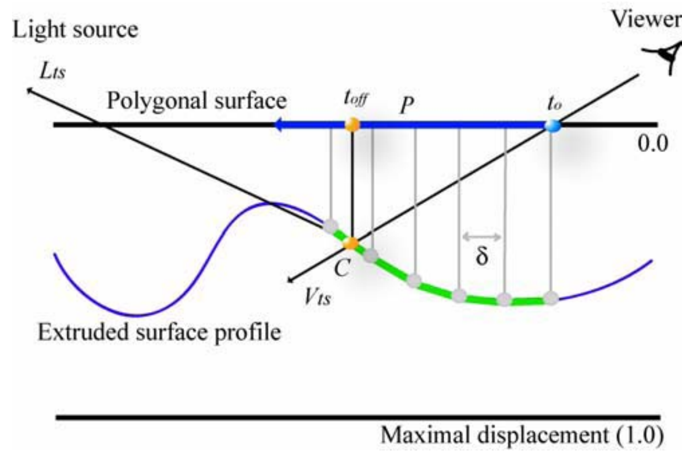


Figure 2.18: For each linear segment of the green piecewise linear curve the heighfield profile is sampled along parallax offset vector P. The intersection yields parallax-shifted texture coordinate offset t_{off}. δ is the interval step size. [29]

They define a safety radius r for each pixel with texture coordinates (x, y) and angle θ, giving a lower bound on the distance to the second intersection, if any, for unblocked rays. In other word, unblocked rays can have at most one intersection with the heightfield in the neighborhood defined by r (see Figure 2.19).

They use this safety radius to find the intersections. At a current position along the ray, above the heightfield, instead of moving a fixed amount dt they advance an amount corresponding to the safety radius r. Therefore, by property, there can be at most one intersection between the two positions. If the new one is above the heightfield, there is no intersection. Otherwise, there is exactly one and we can run a binary search to find it.

A conservative discrete 2D version of the safety radius is stored in a 2D texture. For a texel (i, j) the safety radius is now a number of pixels n such that any ray, whose projection crosses the centerlines within the texel, has at most one intersection with the heightfield within the $2n \times 2n$ square centered on (i, j).

Cone Maps

Cone step mapping [13] replaces both the linear and binary search steps with a single search based on a cone map. A cone map associates a circular cone to each texel of the depth texture. The angle of each cone is the maximum angle that would not cause the cone to intersect the height field. Therefore, at each step along the ray, we can compute its intersection with the current cone (see Figure 2.20).

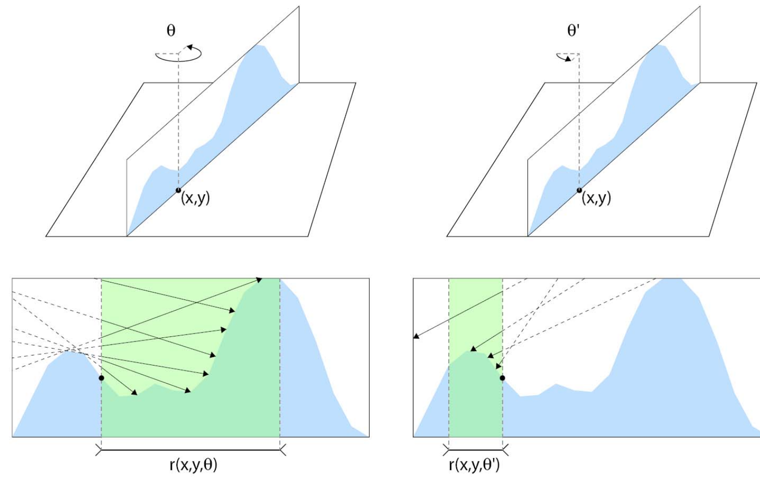


Figure 2.19: The safety radius $r(x, y, q)$ indicates a region in which rays passing above pixel (x, y) with direction q can have at most one intersection with the heightfield. [6]

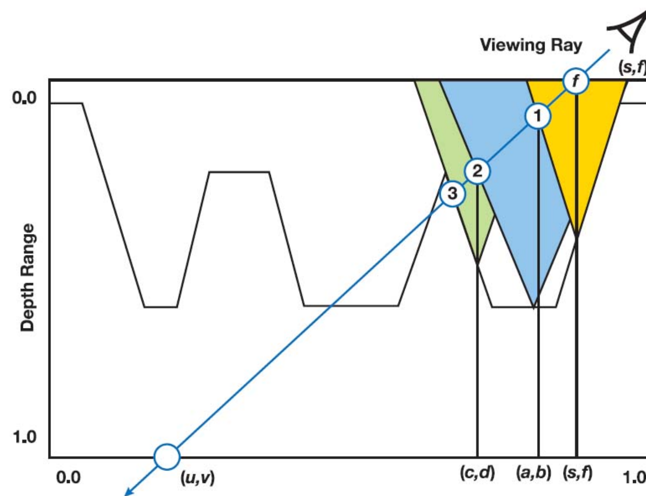


Figure 2.20: At each pass of the iteration, the ray advances to its intersection with the cone centered at the current texel. [13]

Although cone step mapping is guaranteed to get the first intersection of a ray with a height field, it may require too many steps to converge to the actual intersection. For performance reasons, however, one is often required to specify a maximum number of iterations. As a result, the ray tends to stop before the actual intersection, implying that the returned texture coordinates used to sample the normal and color maps are, in fact, incorrect.

A better and more efficient ray-height-field intersection algorithm is the relaxed cone stepping [24]. It combines the strengths of both approaches: the space-leaping properties of cone step mapping followed by the better accuracy of the binary search. Because the binary search requires one input point to be under and another point to be over the relief surface, we can relax the constraint that the cones in a cone map cannot pierce the surface. The idea is to make the radius of each cone as large as possible, observing the following constraint: As a viewing ray travels inside a cone, it cannot pierce the relief more than once.

Note that the radius used by RCS is considerably larger, making the technique converge to the intersection using a smaller number of steps. The use of wider relaxed cones eliminates the need for the linear search and, consequently, its associated artifacts. As the ray pierces the surface once, it is safe to proceed with the fast and more accurate binary search.

2.3.6 Minimal Supporting Geometry

A few recent techniques adopt a relief mapping approach to encode details in arbitrary 3D models with minimal supporting geometry. In [3], new algorithms were presented for the construction, selection and rendering of a compact view-dependent representation, called ORIS (Omnidirectional Relief Impostors) for interactive exploration of complex models. They optimize the set of viewing planes supporting a small collection of properly-oriented relief maps to encode arbitrary objects (see figure 2.21).

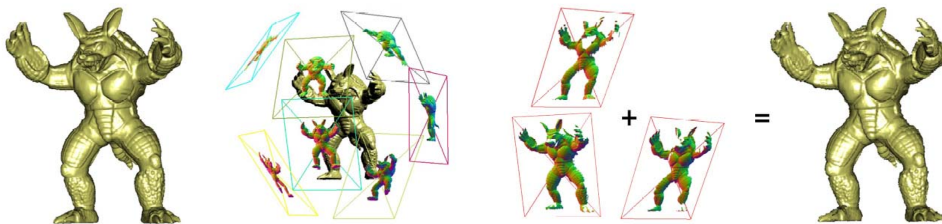


Figure 2.21: ORIs: Several relief maps are combined to provide an impostor set that can be rendered from arbitrary directions. [3]

Unfortunately, output-sensitive approaches like this one or [6] are limited to static geometry.

2.3.7 Animated Relief Impostors

Only a few works attempt to animate geometry encoded as relief impostors. Alternatively to the use of static textures cyclically mapped onto some polygons, texture-based animation can use image warping techniques.

Pamplona et. al [22] described a technique for animating relief impostors using radial basis functions (RBF). It allows the viewer to observe changes in occlusion and parallax during the animation, as relief impostors usually do.

To produce the animations, the user has to manually specify in a preprocessing step, a set of control points over the texture space of the relief impostor. When moving the control points over the 2D texture, makes the texture warp. The new positions of the control points provides new poses, which can then be used as animation key-frames (see figure 2.22).

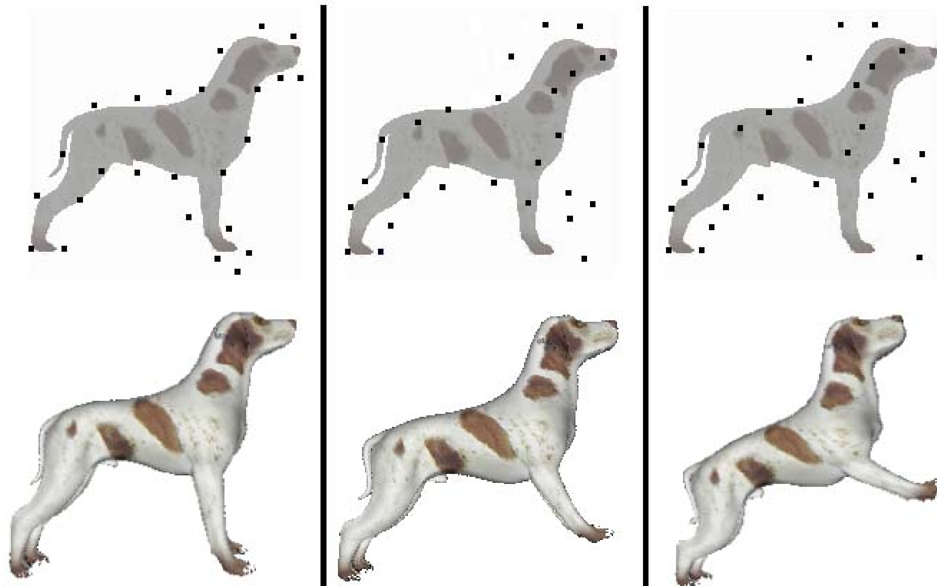


Figure 2.22: Control points (dark dots) are placed over the original relief impostor (left). When moved, the texture is warped, giving a new pose (center and right). [22]

The position of the control points is also interpolated for the inside frames, and a linear system is solved for each of them to obtain the interpolated RBF coefficients. The control points and coefficients can be stored in a textured and used at runtime in the fragment shader.

Interpolated control points and RBF coefficients allows interpolated warps, giving the effect of motion to the relief impostor. The RBF-based animation is added to the relief mapping pixel shader just before the call to the linear search.

The above method suffers from two major limitations: control points defining the animation are just moved in 2D, because image warping techniques are limited to some planar deformations, and it does not support standard skeletal animation which we need for character animation.

2.4 Conclusions

The most extended technology in character animation is the skeletal animation [20], where an underlying skeleton is animated and its bones move the associated vertices of the mesh in relation to certain weights. Due to its simplicity it is also the straight forward technique we would use for a crowd rendering system. The problem of this method is that it is a geometric approach, and therefore the performance of such a system will depend on the geometric complexity, i.e. number of rendered agents and number of polygons per agent. That is the main reason why this approach is not the most suitable for real-time crowd rendering.

To overpass this problem novel crowd rendering acceleration techniques started to appear in the literature. We could separate them in three types, which are geometry- based, image-based and hybrids. The first geometry-based approaches involved have different level of details (LOD) of simplified meshes depending on the distance [26]. Its main problem is to obtain those simplified meshes and the loss of vertices which produce artifacts when they are animated. Image-based approaches appeared with dynamic [4] and pre-generated [30, 31] impostors , where each character was represented by a texture projected on a simple polygon. The main problem of those approaches are some visibility problems, a huge memory cost (for pre-generated impostors) and a pixelization when they are too close to the camera. Hybrid systems [12, 10, 17] then used the same distance to the camera concept as LOD to switch between a geometry render and an impostor one. This systems reduce the artifacts for close agents (since they use geometry) but still have the memory problems. Finally, some other geometry-based approaches [21, 18] try to pre-load and re-use a maximum of meshes exploiting the temporal and intracoherence of crowds.

We have summarized in Table 2.1 all these approaches in crowd rendering. There we classify and compare them stating for each one what data they preprocess, what is their chosen representation for characters and the animation technique they use for them. We also remark the parameter or element that can be considered as the one causing the trade-off between visual quality and performance. In the last columns we evaluate (as high, medium or low) the limitations of each method in terms of memory cost, visual artifacts and computing cost.

It seems clear then that best crowd rendering approaches are the hybrid ones, where impostors are an important factor. To create those impostors, a

Approach	Year	Type	Preprocessed Data	Representation	Animation	Tradeoff	Limitations			
							Memory	Artifacts	Computational Cost	Other
LOD	1997	Geometry-based	Multiresolution meshes	Mesh	Skeletal	Distance or pixel size			XXX	Simplification problem
Dynamic Impostors	1998	Image-based		Oriented billboard	Skeletal	Refresh criterium			X	Limited reusability
Pre-generated Impostors	2000	Image-based	One texture per view and per frame, with color	Oriented billboard	Texture cyclical		XX	XX		
Geopostors	2005	Hybrid	One texture per view and per frame, with color	Oriented billboard / Mesh	Texture cyclical / Skeletal	Distance or pixel size	XX	X	X	
Volumetric Layered Impostors	2007	Hybrid	Several textures per view and per frame, with color, depth and normal	Orientedlayered billboards / Mesh	Texture cyclical / Skeletal	Distance or pixel size	XX		XX	
Polypostors	2008	Hybrid	One texture per view per keyframe, with color	One oriented billboard per body part / Mesh	Texture deformation / Skeletal	Distance or pixel size	X	XX		
Pseudo-Instancing	2006	Geometry-based		Mesh	Skeletal	#Meshes in the GPU			X	Too much data in the GPU
Dynamic Caching	2010	Geometry-based		Mesh	Closest pose (skeletal)	Caché size	X		X	

For each approach we indicate:

- **Year:** Year of publication.
- **Type:** Geometry-based, Image-based or Hybrid.
- **Preprocessed Data:** The data that is created during a preprocessing step.
- **Representation:** The geometric representation used to render one agent.
- **Animation:** The animation technique used to animate one agent.
- **Tradeoff:** The parameter of the approach that implies a tradeoff between visual quality and performance.
- **Limitation:** XXX (High), XX (Medium), X (Low) or none.
  This values have to be applied consequently to each kind of limitation as a reference to compare them with the other approaches.:
    - **Memory:** The required memory cost of the approach.
    - **Artifacts:** Visual or animation artifacts.
    - **Computational Cost:** The global cost of rendering a crowd

Table 2.1: Table comparing the analyzed approaches in crowd rendering.

good choice could be an image based approach as relief mapping [25], which is becoming very popular with the increased use and performance of new GPUs. Relief mapping allow us to retrieve geometry from relief textures on a simple polygons, thus reducing the geometric complexity of the scene (at the expense of a higher per fragment cost). All the relief mapping approaches [25, 29, 6, 13, 24, 3] are based in different kinds of searches for the intersection of a viewing ray with the surface encoded in the texture. The main limitations is the existing tradeoff between the performance and the arising visual artifacts, which are due to the sampling and the algorithm used for the ray intersection search. Finally, relief mapping approaches are made mostly for static objects, except one where impostors are animated warping a texture [22]. The limitation of this method is that it is completely independent from skeletal animation and requires too much human intervention.

As mentioned in the previous chapter, our main objective is to improve crowd simulations designing a new approach based on impostors which will allow us to represent a large number of agents in real time. The state of the art in rendering acceleration techniques, together with the limitations of the traditional impostors employed for animated characters, motivated our work to explore a new approach for rendering large numbers of animated characters in real time.

To take advantage of current skeletal animation techniques, we have worked on a new technique which combines the best of image-based crowd rendering and the best of skeletal animation. Instead of having a single impostor (or a single set of impostors) for each set "agent, animation frame, camera angle" we could use one reduced set of impostors for each bone of the skeleton of each agent, which is independent from both camera angle and animation frame. The impostors would then be animated with the same principle as the mesh is moved in the classical skinning, applying the rotations of the associated joint.

Chapter 3

Our Approach

This chapter first presents a short overview of the proposed approach. It is followed by the detailed explanations of the needed preprocessing steps with all its consequent problems. Finally we show our approach for the rendering process.

3.1 Overview

Based on our conclusions over the current state of the art in crowd rendering and relief mapping, we aim at using relief impostors for crowd rendering, and analyzing its viability. For this purpose we propose a hybrid approach where close characters are rendered with pure geometry, and far ones are rendered with our novel set of relief impostors.

Figure 3.1: A character in a reference pose

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 (see Figure 3.1).

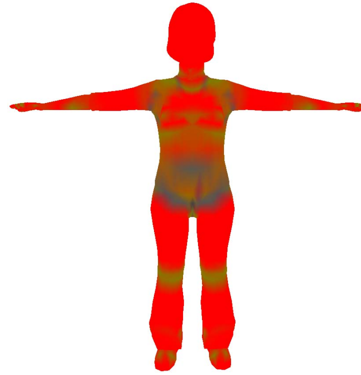


Figure 3.2: Bones influence over the mesh: Each vertex is influenced by a maximum of 4 bones. In this render, for each fragment, we have assigned the weights of at most 3 bones to the three color channels (RGB). So vertices in red zones are influenced only by one bone, while other zones are influenced by two or three different bones.

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 (see Figure 3.2).

Our approach for representing distant characters consists of a collection of oriented bounding boxes (OBB), one for each bone in the skeleton (see Figure 3.3), along with a collection of textures projected into the OBB faces, each texture encoding color and depth values (see Figure 3.4).

In a preprocessing step this representation needs to be constructed (see Figure 3.5). This construction is explained in the next section. During the rendering process, detailed in the last section of the chapter, 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

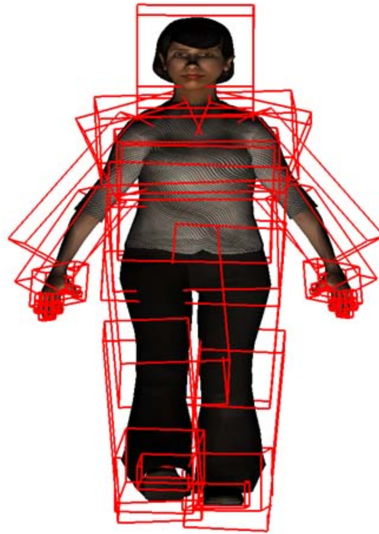


Figure 3.3: Oriented Bounding Boxes, one for each bone.

Figure 3.4: Collection of textures projected into the OBB faces.

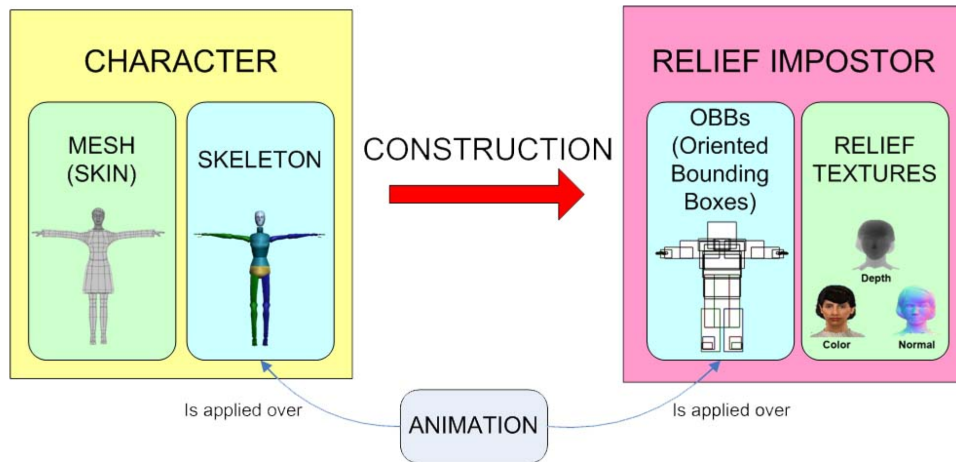


Figure 3.5: From a character composed of a mesh and a skeleton we want to construct a relief impostor with bounding boxes (one for each bone) and project on them relief textures. Animations applied to the skeleton will be applied to the bounding boxes.

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. Third, our technique allows a straight use of skeletal animations, which cannot be done with any of the previous image based rendering techniques. Finally, our method provides a detailed rendering for any character, viewpoint, and animation sequence.

Halca animation library

Our technique relies on the Halca animation library [27, 14] 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 Cal3D XML file format [1] to describe skeleton weighted meshes, animations, and materials.

This library uses shaders for rendering and skinning. Owing to the highly parallel nature of this problem, graphics hardware can carry out the required computations much more efficiently than the CPU. In addition, since only joint transformations are sent instead of all the vertices of the mesh, much less data is transferred between the CPU and the GPU.

Our current implementation works with any animated avatar and any animation that can be exported to the Cal3D format.

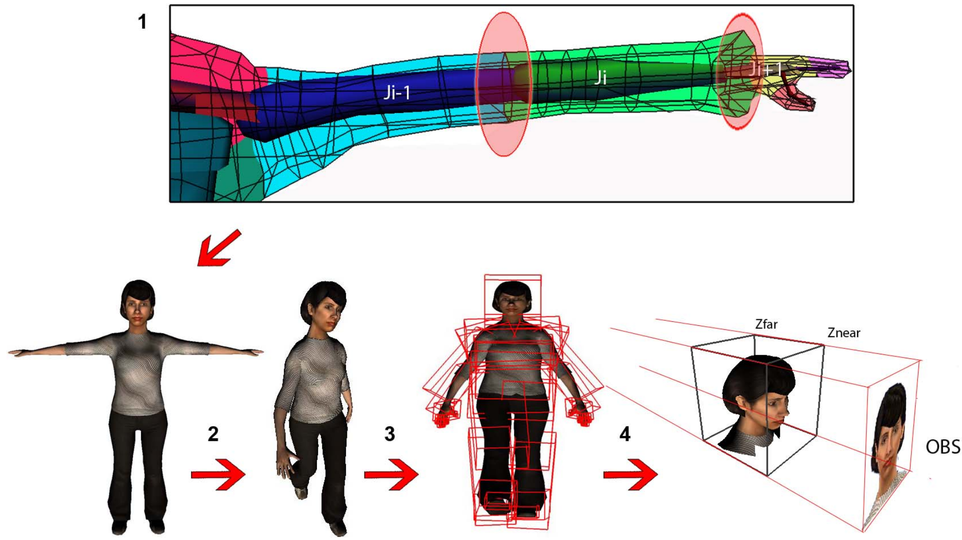


Figure 3.6: Construction steps of our relief impostors: 1) Each triangle has to be associated with one joint J_i. We might have problems with triangles affected by more than one joint (red zones). 2) We must choose a suitable pose to reduce the possible future artifacts. 3) We compute the bounding box of each joint. 4) We project the model onto each face of the bounding box using an orthonormal camera.

3.2 Construction of the Impostors

The construction of our relief impostors from a given 3D character proceeds through the following steps, described in detail below, and illustrated in figure 3.6:

1. **Associate mesh triangles with impostors:** Our new character representation will divide it in a set of bounding boxes of the bones. The textures we will capture will be projected over the OBBs, recovering the mesh geometry. We must then decide which geometry is recovered by which OBB textures. In order to do so we must define an association of the mesh triangles with our impostors.

2. **Select a suitable pose for capturing the impostors:** Using a reference pose when capturing the textures can derive in some artifacts later. This is due to inherent problems with the linear blend skinning technique. We must then find a better pose for capturing the impostors.

3. **Compute the bounding boxes with the chosen pose:** We have to compute the OBBs in function of the positions of the mesh' vertices in the chosen pose.

4. **Capture the textures of each bounding box:** Once everything is set up, we have to capture one texture for each of the 6 faces of each OBB with an orthonormal camera.

The following subsections detail each one of the above steps.

3.2.1 Step 1: Associate mesh triangles with impostors

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 $J_1, \ldots J_n$ with weights $w = (w_1, \ldots w_n)$. Now the challenge 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.

The challenge when assigning triangles to impostors appears because we cannot simply render the complete mesh when creating the textures of one concrete bone. If we did so, we would have several problems to deal with:

- **Occlusion problems:** Since we are going to need to render the different parts of our character from 6 different views (the ones defining the planes of the OBB), it is clear that some of these parts will be occluded by other parts of the mesh. Although we can avoid this adjusting the near and far clipping planes to the ones of the OBB, there could still some occlusion problems due to precision problems with the clipping values (see Figure 3.7 (a)).

- **Redundancy problems:** The OBB of one bone is computed in a way that it includes all the vertices influenced by that bone, but it can be in such a way that other vertices remain inside too. This can derive in parts of the mesh that appear inside more than one bounding box (see Figure 3.7 (b)). But moreover, this parts will be also move along with the bounding box when we will animate it, so they will have a wrong transformations and artifacts will arise.

With the purpose of keeping preprocessing as reduced and automatic as possible we studied two different approaches to assign triangles to impostors trying to avoid the cited problems.

To illustrate the different approaches that we explored we will use a section of the character's mesh which corresponds to a foot as an example (Figure 3.8). This area of the foot is linked to three different joints: the foot itself, the toes, and the ankle. For each approach we will show images of how the different solutions affect the triangles association to the foot's OBB.

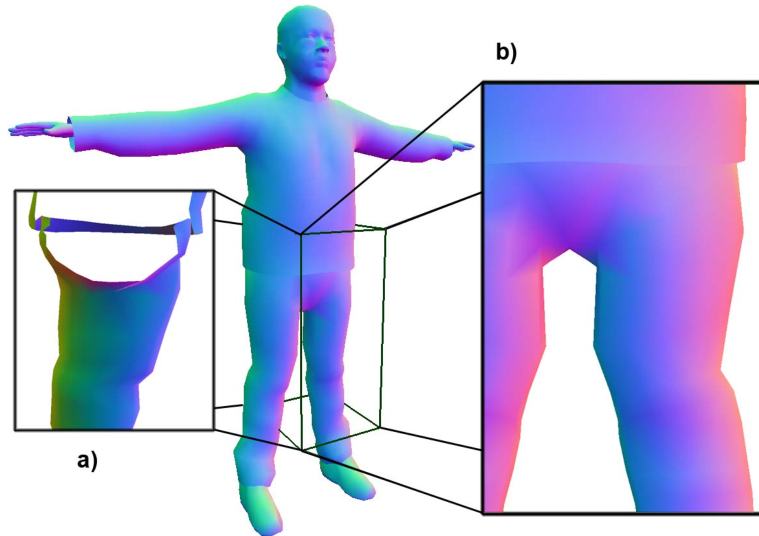


Figure 3.7: If we render all the mesh, we can have occlusion problems when rendering some of the faces of the OBB (a), or triangles which are associated to other bones can be rendered inside the current OBB (b).

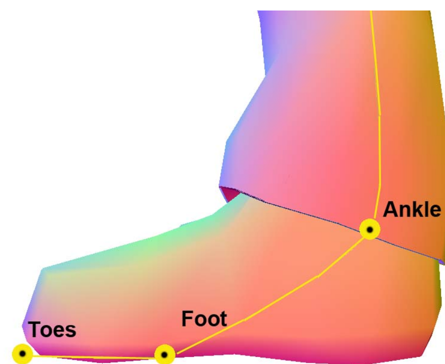


Figure 3.8: The section of a character's mesh representing a foot

Option 1: Triangles assigned to the bone with highest influence over one vertex of the triangle

The first 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). With this approach each triangle is associated to a unique joint. The main problem with this partition is that it tends to produce visible gaps around the joint boundaries during animation, the higher the deviation with respect to the reference pose, the larger the resulting gaps.

In the Figure 3.9 we can see how gaps are formed by triangles which have a maximum linking weight with the toes or the ankle.

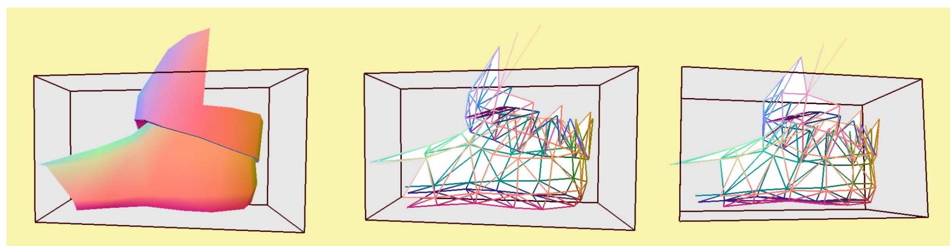


Figure 3.9: Option 1: A triangle is drawn if the maximum weight of its 3 vertices is assigned to the current bone (in this case, the foot). On the left image we can see how gaps are formed in the foot parts near to the toes and the ankle, since those parts had vertices with a higher weight associated to these bones. Center and right images show the same but in wireframe and with a minimal change. We can see how triangles are missing at the gap zones because vertices are discarted.

Option 2: Triangles assigned to all bones with some influence over one vertice of the triangle

To solve the previous problem where gaps appear at zones with vertices influenced by more than one bon, we studied a different approach. Each triangle is assigned to a bone if at least one of its vertices is influenced by the bone, regardless of the corresponding weight. Therefore those triangles around joints will be assigned to a variable number of impostors. Notice that the above strategy only uses vertex weights and thus is pose-independent.

Figure 3.10 shows that the gaps we had with the previous solution are now filled.

This approach avoided occlusion problems, reduced reasonably redundancy problems, and offered better visual results than our previous option by filling gaps. Thus we decided to implement this method in our work.

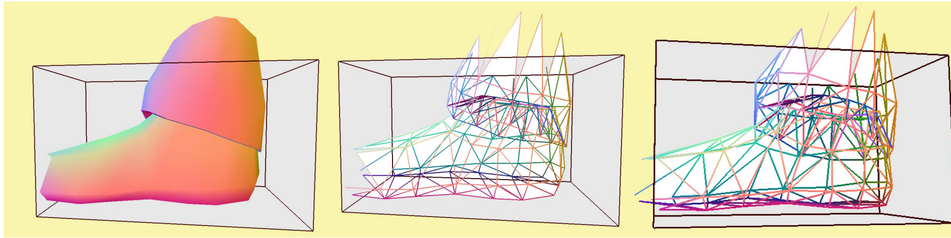


Figure 3.10: Option 2: A triangle is drawn if at least one of its 3 vertices is assigned to the current bone (has a weight greater than 0). The gaps of the previous option are filled and even some triangles out of the bounding box are drawn. Those triangles associated with more than one bone will be rendered when creating both correspondent bone impostor textures.

3.2.2 Step 2: Select a suitable pose for capturing the impostors

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 by using 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 fulfilling two requirements:

- (a) The pose should minimize the overall volume of the bounding boxes to save memory space.
- (b) The pose should represent an average pose of the animation sequence to minimize the visible artifacts during the animation.

In most cases the animations have a reference pose similar to that shown in Figure 3.1. This is a pose that usually has all the triangles "visible", because it is normally the pose used to model the mesh and texture its surface. Although this seems to be a good approach to capture the impostor texture because we would be covering all the triangles, it is not usually the case due to the inherent artifacts of the linear blend skinning technique. Thus selecting a pose with the above properties is necessary.

For example, Figure 3.11 shows the under arm of one character in a reference pose. Note that the surface is continuous and presents no creases, so all the triangles are visible. Figure 3.12 shows the same zone in a walking pose. We can see that in such conditions, it is common to have auto-intersections and triangles that are not visible anymore. So, if we capture the impostors with the reference pose we would be covering triangles, that we will not want to see during the animation.

Therefore, if the animation sequence shows a character walking with the arms in a rest position, it is better to capture the triangles around the

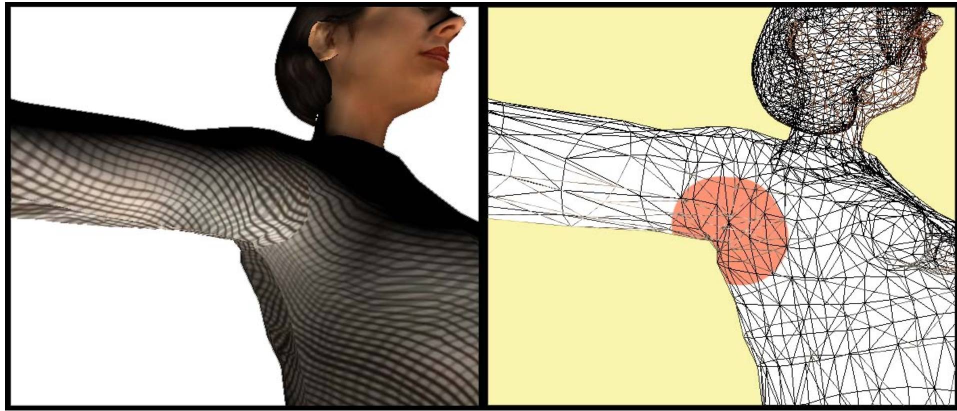


Figure 3.11: Under arm zone of the mesh in a reference pose (in wireframe mode on the right). This is the original mesh pose, the one used to model it. At the red zone, the surface is continuous and presents no creases, so all the triangles are visible.

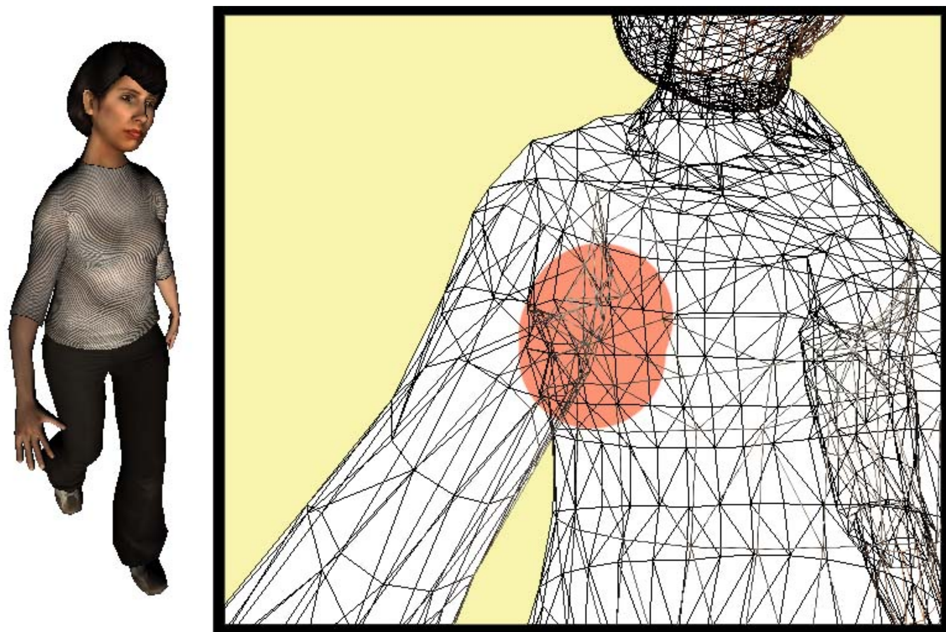


Figure 3.12: A character's mesh in a walking pose (on the left) and it's under arm zone with the same pose (in wireframe mode on the right). At the red zone, we can see how, due to linear blend skinning, we have some auto-intersections and triangles that are not visible anymore.

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. We do not guarantee finding an optimal pose but our results are reasonably good enough. 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.

3.2.3 Step 3: Compute the bounding boxes with the chosen pose

At first, the bounding boxes are computed when the character in its reference pose. The bounding box of each bone is computed as the axis-aligned bounding box (AABB) of the vertices with a weight associated to the bone. Once a suitable pose has been chosen, we need to apply it by transforming the bounding box.

To apply the chosen pose, the bounding box of the joint J is rigidly transformed accordingly by applying linear blend skinning to its vertices, i.e. the transformed bounding box vertex v' is computed as

$$v' = M_J v$$

where M_J is the rigid transformation matrix from the reference-pose of joint J to its actual position in the chosen posture. The bounding box of each impostor will then be oriented when we will capture the textures.

Another option we should contemplate is to compute bounding boxes aligned with the bone, called oriented bounding boxes (OBB), but once the character's mesh is deformed by the chosen pose with linear blend skinning. For an arm or a leg bone, it is easy to decide which axis the box should be aligned with. We might have some problems with some bones which are more spherical like the head, but in general we would align the box along the axis formed by the vector going from the current joint to its parent in the hierarchy. Unfortunately, at its current version, Halca just let us to compute AABB of the reference pose.

3.2.4 Step 4: Capture the textures of each bounding box

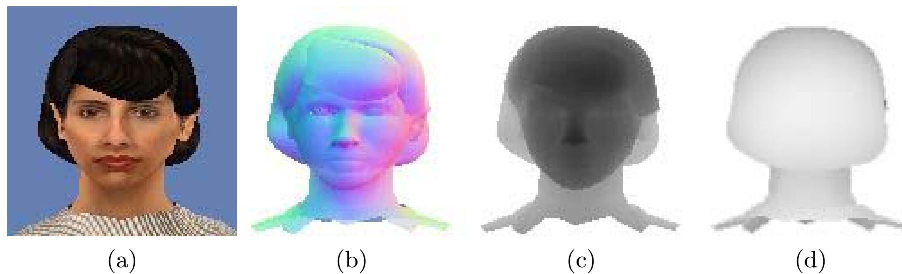
The last step is to render the deformed mesh to capture the relief maps corresponding to each one of the six faces of its bounding box. In order to do so, for each bounding box face we set up an orthographic camera with its viewing direction aligned with the face's normal vector, and then render the triangles assigned to the corresponding impostor.

To apply relief mapping we need relief textures, i.e. one or more depth layers. We also need the surface color, and surface normal if we want to apply lighting computation. The idea is then to have one texture for color values, and another for normal values. Each one of this textures can use its alpha channel to store one depth value, so we choose to use two depth layers: front depth values and back depth values.

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.

So, we store the following RGBA textures (Figure 3.13):

- 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.



(a) (b) (c) (d)

Figure 3.13: Color (a), normal (b), front depth (c) and back depth (d). Values are encoded as two RGBA textures. The first one encodes color (RGB channels) and front depth (Alpha channel). The second one stores the normal (RGB channels) and the back depth (Alpha channel).

3.2.5 Storage of the Impostors

Once the bounding boxes are computed and the relief textures are created, we might want to store them in order to not recompute them later. The bounding box computation is fast enough, so we do not need to store it in a hard drive memory, but we can store it graphic memory in order to re-use for all instances of the same type of character. The color and normal relief textures though, need to be stored in a hard drive memory, since

its generation is not so fast due to the number of them that we need to capture for each character. Although, when used at rendering time, as for the bounding boxes, we need to load the textures into the graphic memory using an efficient data structure.

The vertices of all bounding boxes of a character are stored in a single Vertex Buffer Object (VBO), which is used for all the instances of the same character. In fact, along with the vertex coordinates, we put in the VBO the needed face attributes that we will need to apply relief mapping.

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 64×64 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 [2] (one for color maps and one for normal maps) to avoid texture switching while rendering the instances of the same articulated character. If we would not use texture arrays, binding and unbinding the textures into the GPU could dramatically slow down the rendering performance.

The textures are arranged in the array using the bone's *id*, which is also stored in the VBO of the OBB along each vertex. This way, for the face $f \in \{0, 1, 2, 3, 4, 5\}$ of the joint J_i with $id = i$, the color and normal maps of our relief impostor will be in the position $i * 6 + f$ of the texture arrays.

3.3 Real-Time Crowd Rendering

Our proposed hybrid model 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 (see Figure 3.14). Since we are talking about humanoids, we assume that our characters will all have approximatively the same size. So the distance to the camera is inversely proportional to the projection size and can be used to decide whether to use or not impostors. For that reason we choose to switch between the two representation as a function of the distance instead of the pixel size of the character's projection.

Crowd Rendering Algorithm

Our crowd rendering algorithm, at each frame, works as follows:

1. For each agent, the Halca animation library [27] must update its loaded animation. In order to do so, the geometric transformations of each bone are updated depending on the current time and the speed at

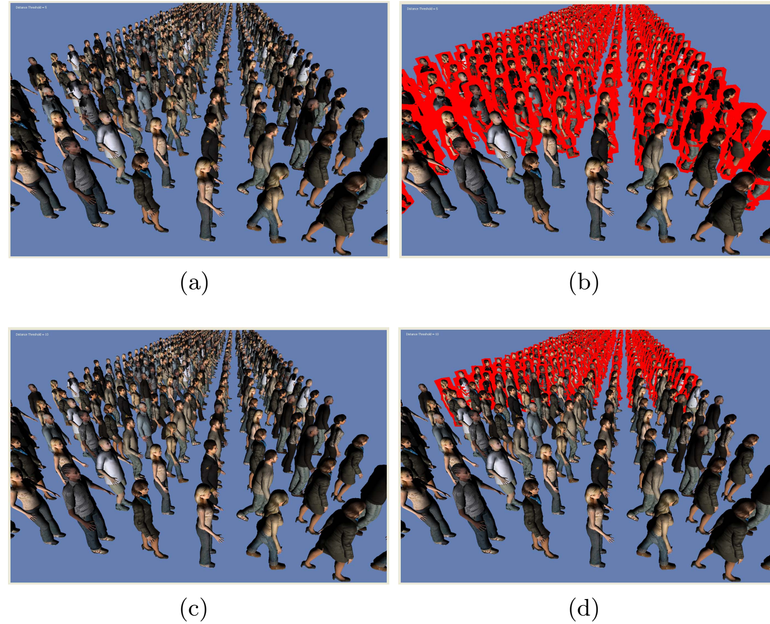


Figure 3.14: Distance Threshold: Agents closer than 5m (a and b) or than 10 m (c and d) are rendered with pure geometry while far ones are rendered with our impostors. Agents with a red aura (b and d) are the impostors.

which the animation is played. This produces the rigid transformation matrices $M_1, ..., M_n$ (where n is the number of bones of the current agent) of each bone corresponding to the current pose of the agent.

2. For each agent, its distance to the camera is computed and it is decided whether it will be rendered with pure geometry or with impostors, using a user defined threshold.

3. Agents are sorted by character type (mesh, textures, set of animations). In order to minimize rendering state changes we will render all instances of a same character type before changing to another character type. This way, all shared data between those instances will be sent to the GPU just once per frame and re-used for all of them.

4. For all agents, render nearby polygonal agents with the Halca animation library [27], using linear blend skinning to deform meshes.

5. For all agents, render the far ones as impostors. Each one of those characters is rendered through an adapted version of relief mapping over the fragments produced by the rasterization of the transformed bounding boxes.

The following subsections detail how our novel impostors are rendered. We decomposite its explanation in three parts: (1) the part containing all the operations to be done by the CPU, (2) the vertex shader with all the operations to be done for each of the bounding boxes vertices, and (3) the fragment shader part which is the most relevant part because it is where the geometry is retrieved from the relief textures.

3.3.1 CPU Process

The CPU-based part of the rendering algorithm proceeds through the following steps:

1. Bind to the GPU the corresponding texture arrays (color and normal maps) into different texture units.

2. Bind also to the GPU the vertex buffer object (VBO) with the geometry of OBBs, which we have computed, in a preprocessing step, in the same way as in the creation of the impostor textures. As mentioned in the creation section, this VBO contains other properties apart from the vertices coordinates, such as faces' normals or bone identifying number. These steps are performed only once per character type, as it will be reused for all instances of the same type.

3. Use the Halca animation library [27] to send to the GPU the uniform variables encoding the rigid transformation matrices $M_1, \ldots M_n$ (where n is the number of bones) of each bone corresponding to the current pose to make this information available to the shaders. Since each agent has its own independent animation been played, this step must be performed once per instance.

4. Draw the 6 faces of the OBB associated to each bone, just to ensure that a fragment will be created for any viewing ray intersecting the underlying geometry.

3.3.2 Vertex Shader Program

The purpose of the vertex shader is to transform the incoming vertices before the triangles are drawn, in order to achieved the motion and the desired animation of the whole character.

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. Also, we can send as varyings to the fragment shader all the vertex attributes containing information about the relief impostor. For each fragment, varyings are interpolated by the GPU from all

the vertices of the triangle from which the fragment comes. Since the varying parameters would have the same value for all the vertices of one face, they would not be altered when being interpolated.

These attributes are basically vectors encoding the orientation and location in space of each relief map. Since we stored them in our VBO when our character was at its initial pose, we must transform them too in the vertex shader as we transform the OBB where the relief texture will be projected. So we have to rotate them with the correspondent bone rotation matrix, which we can extract from the bone transform matrix used before. You can find the GLSL code of the vertex shader in the appendix A.

3.3.3 Fragment Shader Program

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. Our current prototype is based on the relief mapping algorithm described in [25].

The fragment shader, which GLSL code is in the appendix B, receives as input the following information (see also Figures 3.15 and 3.16):

- World space viewpoint coordinates E.
- World space fragment coordinates C.
- The origin $P0$ 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 $\vec{n}$ and two vectors $(\vec{u}, \vec{v})$ aligned along the horizontal and vertical sides of the transformed face.
- Impostor depth factors $(d_u, d_v) = (\frac{z_{far}-z_{near}}{\|\vec{u}\|}, \frac{z_{far}-z_{near}}{\|\vec{v}\|})$: where z_{far} and z_{near} are the distance to the clipping planes of the orthonormal camera used to render the relief texture.
- Impostor depth range $d_r = \|z_{far} - z_{near}\|$.

The viewpoint coordinates are available as a uniform variable which is 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 impostor depth factors and impostor depth range are also available through varying variables.

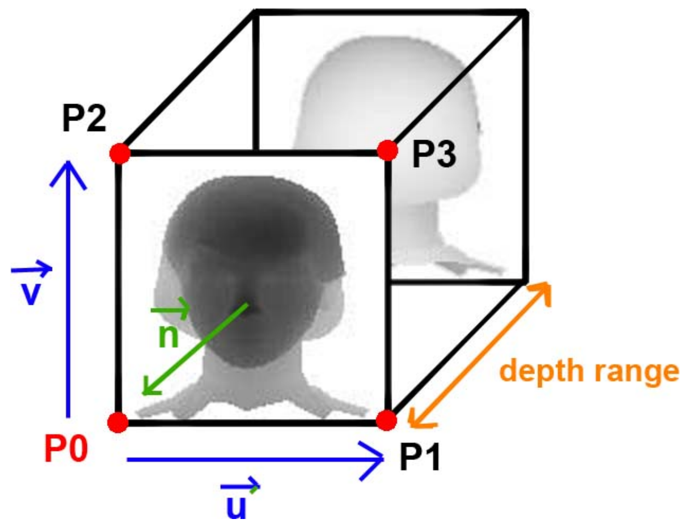


Figure 3.15: Impostor parameters stored in the VBO, transformed by the vertex shader and used by the fragment shader.

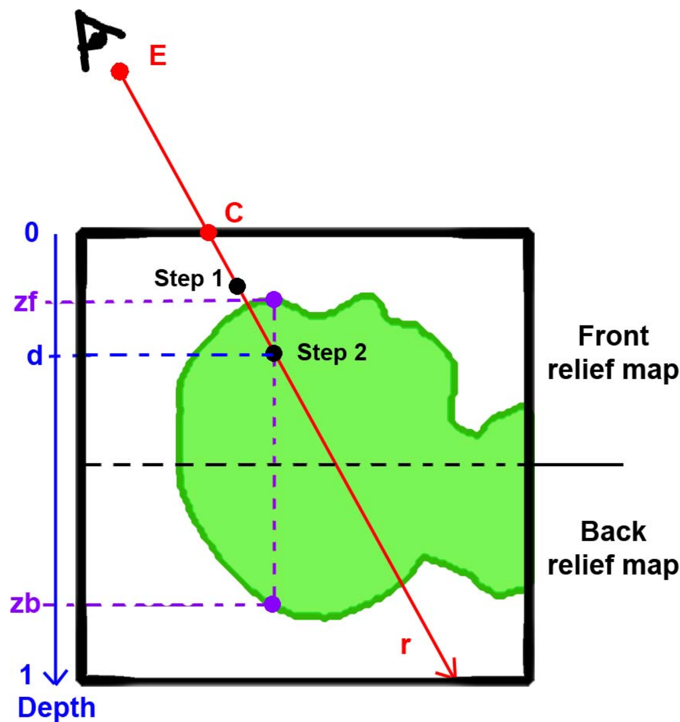


Figure 3.16: Ray-heightfield intersection search. At each search step, the current depth d is compared with the front depth zf and the back depth zb to check whether we are inside the geometry or not.

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 (see Figure 3.16). If no intersection is found, the fragment is discarded. As in [25], 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 zf (in the A component of the color map) and zb (in the A component of the normal map) per texel. During the search processes, a sample along the ray with depth d is classified as interior to the object iff $zf \leq d \leq zb$.

3.4 Conclusions

We have presented a new hybrid approach for crowd rendering in real-time using a novel kind of relief impostors. We have explained how to create these impostors, how to efficiently store them, and how to use them to render and animate distant characters of a crowd by using a classic relief mapping technique. We need now to objectively analyze the performance and the visual quality of our method, and evaluate if it is a valid method. This is done in the next chapter, where we also discuss some of its current limitations.

Chapter 4

Experimental Results and Discussion

In this chapter we show the experimental evaluation realized to analyze the viability of our proposed approach. These tests basically compare the achieved frames per second using our impostors against rendering with pure geometry, and also as a function of the number of pixels they occupy in the screen. We also explain the setup and results of our user study, which has helped us to validate the visual quality of this new approach. Finally we discuss some aspects of our work that might need to be improved.

4.1 Performance Tests

Our performance tests are based in the measure of the achieved frames per second of each considered approach, as a function of the total number of agents in the crowd simulation. We first compared performance using two radical approaches:

- Pure geometric rendering: The used characters have polygonal meshes having between 4K and 6K triangles, with 2048×2048 texture atlases for color and normal values, and skeletons of 53 bones (because they are fully articulated and for example, each finger is consider individually and formed by 3 bones).

- Pure impostor rendering: Each impostor was represented by 53 OBBs, one for each bone (we do not group individual finger bones in our current prototype), using 128×128 array textures for color, normal and depth values. Taking into account that for each OBB we have 6 faces, each one with 2 textures of 128×128 pixels, with 4 channels RGBA of 1 byte, this resulted in $53 \times 6 \times 2 \times 128^2 \times 4 = 41.6$ MB for character type. For all the impostors rendering, in the ray-heightfield

intersection search, the linear search was made with a maximum of 16 steps, and the binary search with a maximum of 7 steps.

We expected our impostor-based approach to have a better rendering performance, since they are output sensitive as opposed to being dependent on the number of agents, i.e. the geometric complexity. Figures 4.1 and 4.2 shows the results with a varying number of agents, using the same character type for all instances (4.1) or using ten character types (4.2). All times were measured on an Intel Core2 Quad Q6600 PC equipped with a GF 8800 GT, using a 1280×1024 viewport.

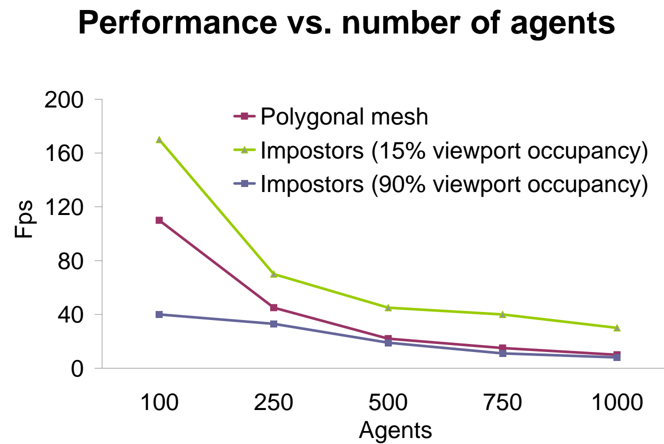


Figure 4.1: Performance results with one type of character

By *viewport occupancy* we mean the proportion of the screen pixels where the impostors are drawn, and thus need to be processed by the impostor fragment shader. In a typical impostor usage scenario with the distant agents covering a 15% of the viewport, as depicted in Figure 4.3, 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. If we have more than one character type (Figure 4.2), we have to bind and unbind the different impostors textures, so the achieved framerate is bit worst, but still better than with polygonal meshes.

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

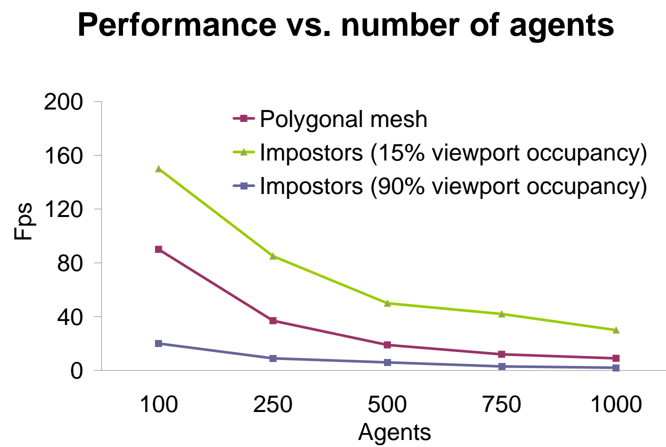


Figure 4.2: Performance results with ten types of characters

Figure 4.3: Crowd with about 5,000 agents, all of them rendered with our relief impostors.

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.

The problem is that, studying the performance of our relief impostors in a crowd simulation system, we have actually reached a different bottleneck. As we can see in Figures 4.1 and 4.2, from a certain number of agents (around 1000) the system looses all interactivity and the frames per second are close to 0. After studying the problem closely, we believe that the bottleneck is now in the transfer of the transformation matrices of each agent. Since each agent has its own independent animation, with 53 transform matrices to be sent to the GPU to be used by the vertex shader, its cost is still linearly dependent on the number of agents.

4.2 Image Quality

Figures 4.4 show images of three different types of characters rendered using relief impostors in a random walk pose. Figure 4.5 shows two of these characters in an walking animation sequence, showing how the OBB are rigidly animated give good results combined with our relief mapping rendering.

Figure 4.4: Agents rendered with relief impostors

Although the images show some artifacts around joints, these artifacts are very hard to perceive in the context of a crowd simulation since we would be rendering many animated agents far away from the camera. Also, retrieving the normal and computing a per-fragment lightning can help to diminish those artifacts.

We conducted a preliminary user study to evaluate our impostor-based approach in terms of image quality. The objective was to know the switch distance at where the impostor became unnoticeable to the human eye, when compared to the geometry rendering.

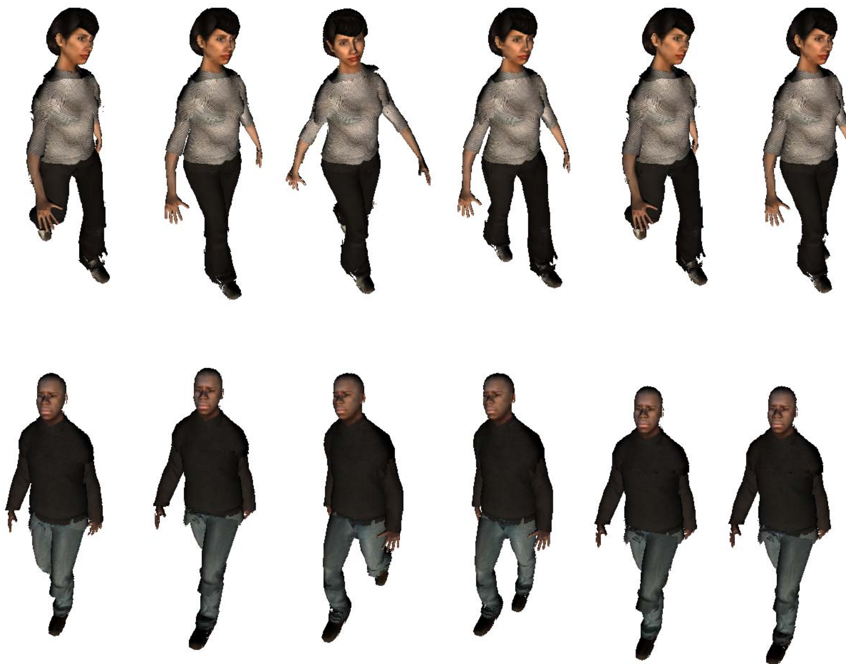


Figure 4.5: Relief impostors are rigidly animated by transforming the vertices of the supporting bounding boxes

4.2.1 User Study

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 ∞ (pure geometric rendering). We produced a 25 s movie for each resulting animation, with switch distances $d \in \{0,\ 10\ m,\ 15\ m,\ \infty\}$, i.e. switching to impostor rendering when the viewer-to-agent distance was above d.

In order to assess image quality with respect to the reference image, we grouped these movies in pairs, stacking vertically one of the movies using impostors with the one using pure geometry (see Figure 4.6, and the videos which can be downloaded from http://www.lsi.upc.edu/~abeacco/MasterThesisVideos.zip). The movie with pure geometry was stacked on the top or on the bottom randomly.

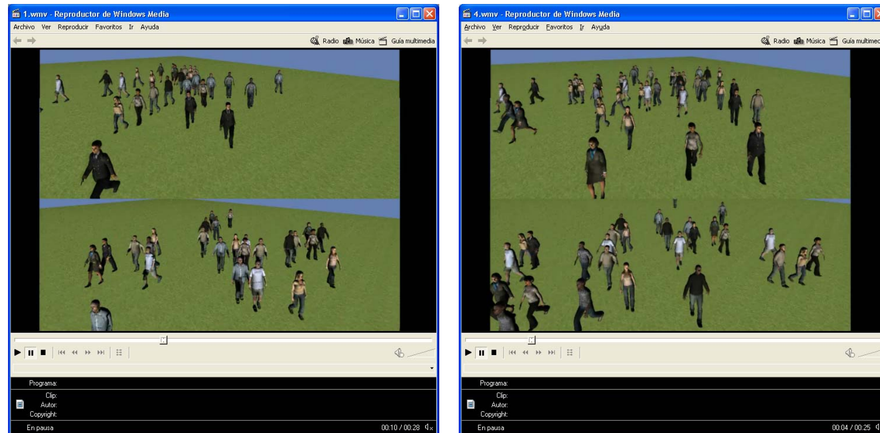


Figure 4.6: Two screenshots from two of our videos used in our user study.

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 geometry, the height of the camera above ground level was set so that the screen projection of each character was below 60×120 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 $(0, 10-15m)$; 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 $d = 10-15$ m. Notice for example that, when switching to impostors at $d = 10-15$ 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 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. This is best evaluated and explained in the next subsection.

4.2.2 Image Difference

To find an explanation to our user study results, and to have an objective and quantified evaluation of our method, we have proceed to make *image differences* of the same scenes with geometry and with impostors. The basic idea behind image difference is to compare each pixel color of two different renders by making the difference between each color component.

In order to do so, we have created *pseudocolored images* with Photoshop software . Pseudocolored images help to see the zones where differences are and its magnitude. The resulting image from making color differences is converted to a grayscale image, and from there each gray value is indexed to a color from a pseudocolor table.

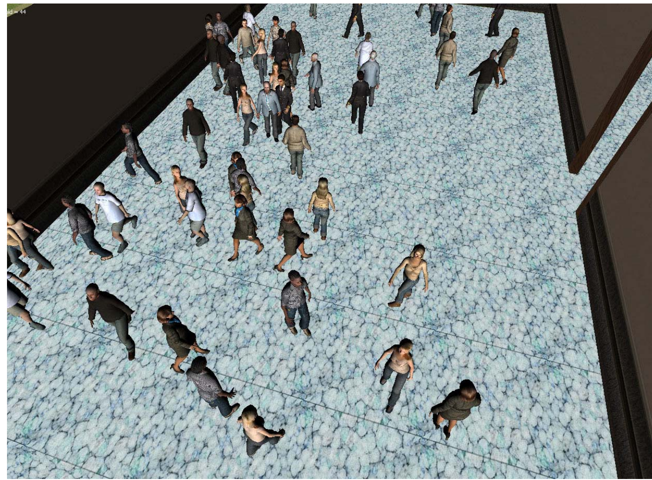
Figure 4.7) shows an example of image difference between the render of one scene with geometry and our impostors. The pseudocolor bar at the bottom of the pseudocolor image indicates the color scale used for each error value. This way, dark blue indicates images have exactly the same equal color at that pixel, while red indicates the maximum error between colors. As we can see in the figure, only a few pixels show to have minimal color differences between our two methods. Moreover, these differences seem to be only around the silhouettes of the impostors. This could be explain by some of the problems of relief mapping at retrieving surface on silhouettes: since it is where the view ray is tangent, the ray-heighfield intersection search might not find a point inside the surface, and therefore the sampling we made is more important. In any case, the difference is very small, and the impostors are shown from a distance big enough to be almost unnoticeable for the human eye.

4.3 Discussion

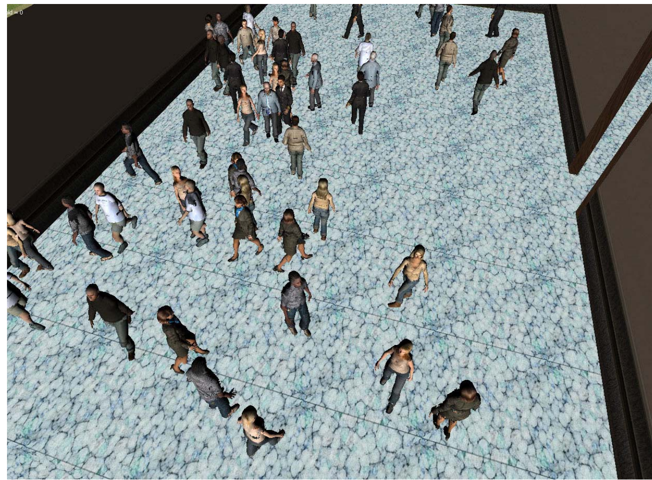
Despite the good results we obtained in terms of quality and performance, we find that our new approach has some aspects and limitations that could be handled in next phases of our research.

4.3.1 Distance Trade-Off

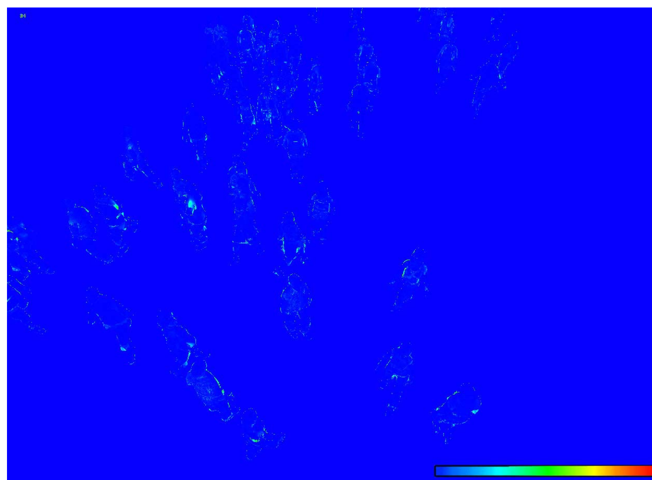
In section 3.3 we say that agents close to the camera are rendered with original geometry, while the others are rendered as impostors. Although this seems to be the most extended solution to decide whether to use or not impostors, we have not really searched for the optimal distance to switch between render modes.



(a)



(b)



(c)

Figure 4.7: Image rendered with polygonal meshes (a) and our impostors (b). Image difference is shown in (c).

When close agents are rendered with impostors, artifacts become more noticeable. As they occupy a bigger area on the screen, the performance is also drastically reduced. But at some distance they become visually indistinguishable and have better performance than geometry. So, clearly, there is a trade-off regarding this distance that we should take into account more seriously.

4.3.2 Texture Resolution

In our current implementation, as mentioned in section 3.2.4, we use texture arrays to avoid a continuous binding and unbinding of textures. The disadvantage of textures 2D arrays is that we must use for all the textures the same resolution. This is not perfectly convenient because we might not want to have the same resolution and detail for all the bounding boxes. For example we do not need the same resolution for a hand (where there are not many details) as we would need for the head (where we want to sample the complete face of the character). Figure 4.8 shows this with 3 different texture resolutions. An adaptive resolution for the relief maps would be an important enhancement of the system, since less memory would be used and transferred to the GPU while visual quality would be almost the same, with unnoticeable differences.

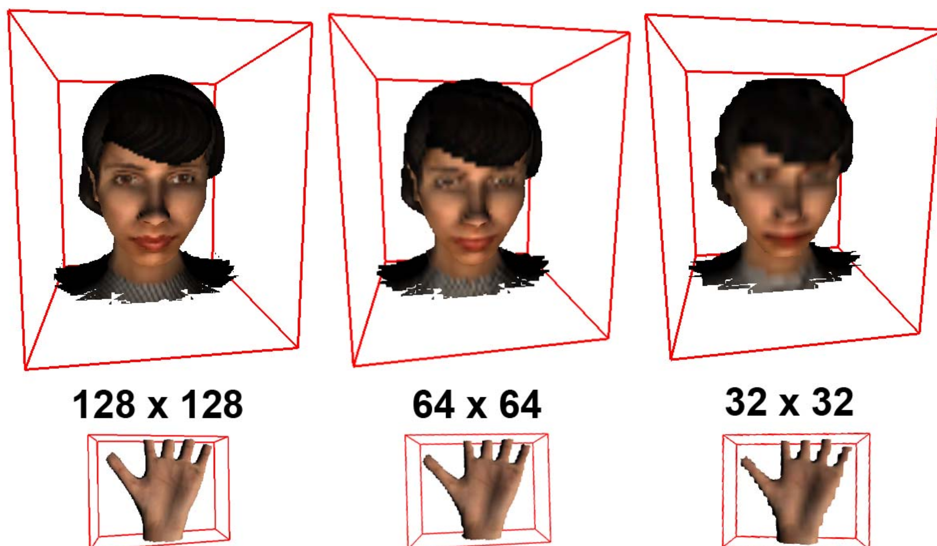


Figure 4.8: Reducing the resolution of relief textures reduces the sampling and visual quality of the retrieved surface. For some parts, like the head, a good resolution is needed as there is a lot of detail. For other parts, like a hand, which is smaller and with much less detail, a smaller resolution can be enough.

4.3.3 Ray-Heightfield Intersection Sampling

Regardless of the combination of several relief textures, the final visual quality of our approach, as well as its performance, still depends on the relief mapping technique and the chosen ray-heightfield intersection algorithm. More specifically, it critically depends on the sampling we made, i.e. the number of steps we take in the linear and binary searches for the ray-heighfield intersection.

Although we have made no study about the visual quality and the performance when varying these parameters, we expect that an adapted sampling version of our approach would improve our results. For example, a version where the number of steps would vary as a function of the viewing distance. Figure 4.9 shows a character rendered with impostors varying distance and sampling. As we can see, for far views we could use fewer steps without noticing any differences.

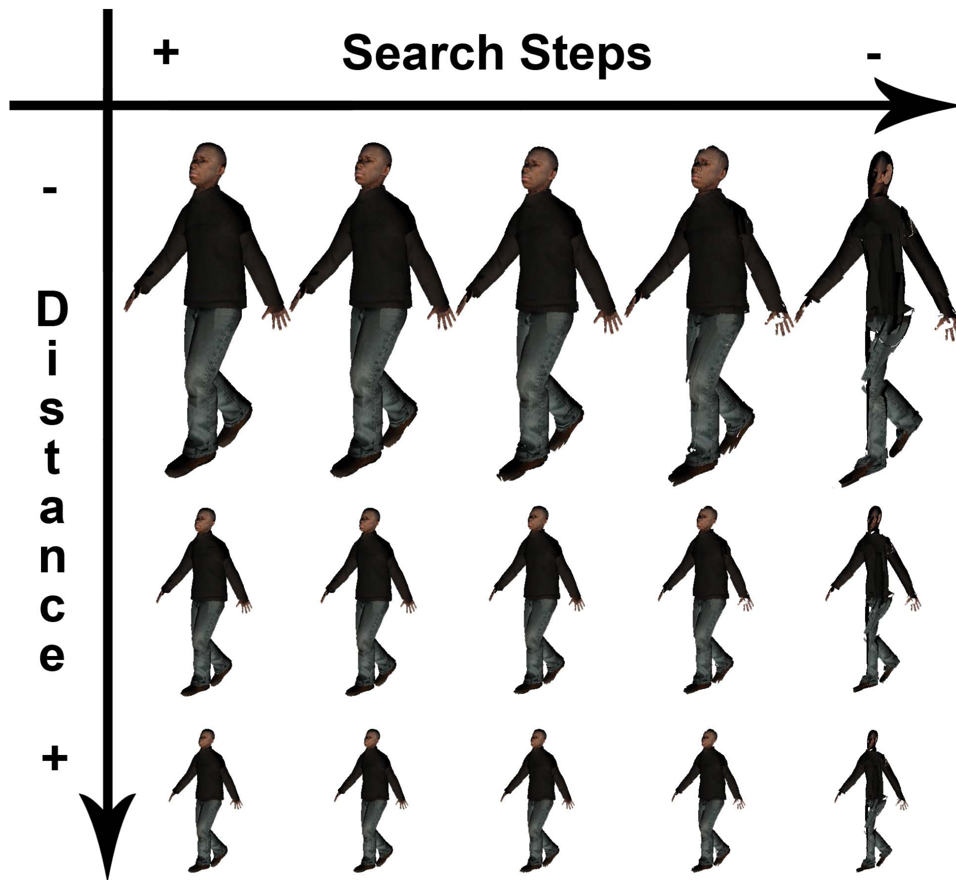


Figure 4.9: A character render with impostors at different distances and with different number of search steps in the linear and binary search for ray-heightfield intersection.

4.3.4 Artifacts due to Rigid Transformations

As seen in section 3.2, when capturing the textures of our impostors, we choose a walking pose instead of a reference one in order to dissimulate some artifacts. The artifacts we were talking about where the ones due to the inherent problems of skeletal animation and linear blend skinning. But there is another kind of artifacts due to the rigid transformations we are applying over the OBBs. When we animate our impostor characters, we are just translating and rotating those boxes, with its whole underlying geometry. We expected this kind of artifacts, but surprisingly these were not so noticeable when using a similar pose to create the impostors as the one we were trying to use (a walking one).

Figure 4.10 shows how are exactly these artifacts, produced above all the joints links. In this figure the impostors were captured in a reference pose. The problem is that we are not recreating the effect of linear blend skinning, where some vertices are influenced by more than one bone. In our approach, all the underlying geometry of an impostor OBB is fully influenced by its corresponding bone. Therefore the skin is not bended in any way.

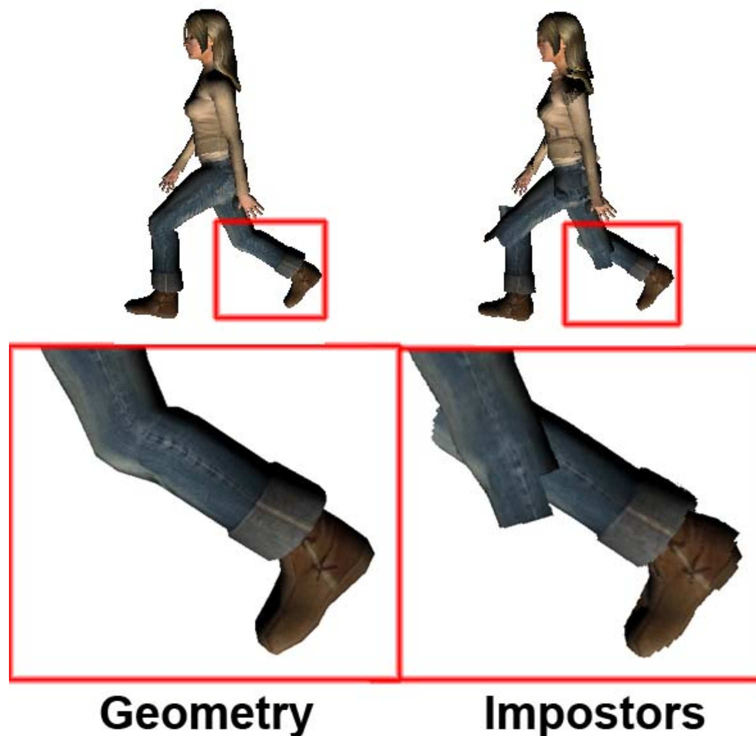


Figure 4.10: Due to the rigid animation we apply to all the impostor OBB, artifacts can arise at the joint links. HHere we can particularly notice these artifacts around the knees.

An elegant solution would be to really animate in a dynamic way our impostors. A similar approach to the one presented in [22], where relief textures are warped in time to simulate an animation, could help to really animate the underlying geometry. This is, in fact, an option we want to explore in our future research. The warping could be done as a function of the *rigging* weights, which could be encoded in another texture, for example.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The main problem behind crowd rendering lies in the amount of geometry we have to render and animate for each one of the represented agents. The more difficult is to render one character, the less agents we will be able to have in a crowd simulation. After studying the state of the art in crowd rendering and knowing its limitations, we have noticed that impostors, which are largely used in other areas, are starting to become the core of new crowd rendering approaches. Moreover, we have also studied the state of the art in a particular technique to create relief impostors: relief mapping. We have then proposed a novel approach to test the validity of using relief impostors for crowd rendering.

We have presented a new method to accelerate the rendering of crowds by using static relief impostors on rigidly animated bounding volumes. This 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.

The proposed approach 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, by the moment given by the user. 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. In summary, the validity of our new method encourages us to extend and improve this first approach using relief impostors for crowd rendering.

A paper with the content of this thesis has been submitted and accepted for publication in CEIG 2010 (*Congreso Español de Informática Gráfica*) [8], which also confirms that we are in a good research direction.

5.2 Future Work

As discussed in the previous chapter, there are some aspects and limitations we would like to work on in our future research. First we would like to consider a new kind of dynamic impostors, where the relief impostors would be dynamically animated, for instance by warping the textures, 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. Also, an adaptive sampling when searching along the ray as function of the distance to the camera of the agent could accelerate the system without loosing visual quality.

Uploading the animation to the GPU beforehand would also reduce bandwidth requirements and would enable to perform all motion-blending computations on the GPU. We would also like to study the possibility of adapting the texture size according to the relevance of the body part, for instance increasing resolution for the head textures with respect to those for the legs.

Finally, once some of these improvements were done, we would like to repeat with more users our user tests to further validate our approach.

Bibliography

[1] Cal3d. 3d character animation library.
http://home.gna.org/cal3d/.

[2] Nvidia texture array.
http://developer.download.nvidia.com/opengl/specs/
GL_EXT_texture_array.txt.

[3] C. Andujar, J. Boo, P. Brunet, M. Fairen, I. Navazo, P. Vazquez, and A. Vinacua. Omni-directional relief impostors. *Computer Graphics Forum*, 26(3):553–560, September 2007.

[4] A. Aubel, R. Boulic, and D. Thalmann. Animated impostors for real-time display of numerous virtual humans. In *VW '98: Proceedings of the First International Conference on Virtual Worlds*, pages 14–28, London, UK, 1998. Springer-Verlag.

[5] A. Aubel and D. Thalmann. Realistic deformation of human body shapes. In *Proc. Computer Animation and Simulation 2000*, pages 125–135, 2000.

[6] L. Baboud and X. Décoret. Rendering geometry with relief textures. In *GI '06: Proceedings of Graphics Interface 2006*, pages 195–201, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[7] I. Baran and J. Popović. Automatic rigging and animation of 3d characters. *ACM Trans. Graph.*, 26(3):72, 2007.

[8] A. Beacco, C. Andujar, B. Spanlang, and N. Pelechano. Output-sensitive rendering of detailed animated characters for crowd simulation. *Congreso Español de Informática Gráfica, CEIG10*, September.

[9] K. Cain, Y. Chrysanthou, and F. Silberman. A case study of a virtual audience in a reconstruction of an ancient roman odeon in aphrodisias. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 4, New York, NY, USA, 2005. ACM.

[10] J. Coic, C. Loscos, and A. Meyer. Three lod for the realistic and real-time rendering of crowds with dynamic lighting. Research Report RN/06/20, Université Claude Bernard, LIRIS, France, April 2007.

[11] F. Cordier and N. Magnenat-Thalmann. A data-driven approach for real-time clothes simulation. *Computer Graphics Forum*, 24:173–183, 2005.

[12] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry / impostor crowd rendering system. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 95–102, New York, NY, USA, 2005. ACM.

[13] J. Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. *IEEE Comput. Graph. Appl.*, 2006.

[14] M. Gillies and B. Spanlang. Real-time character engines comparing and evaluating real-time character engines for virtual environments. *Special Issue on Presence*, 2010.

[15] S. Jeschke, M. Wimmer, and W. Purgathofer. Image-based representations for accelerated rendering of complex scenes. In *Eurographics 05 State of the Art Reports [STAR]*, pages 1–20. The Eurographics Association and The Image Synthesis Group, 2005. Vortrag: Eurographics, Dublin, Irland; 2005-08-29 – 2005-09-02.

[16] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan. Geometric skinning with approximate dual quaternion blending. volume 27, page 105, New York, NY, USA, 2008. ACM Press.

[17] L. Kavan, S. Dobbyn, S. Collins, J. Žára, and C. O'Sullivan. Polypostors: 2d polygonal impostors for 3d crowds. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 149–155, New York, NY, USA, 2008. ACM.

[18] W. Lister, R.G. Laycock, and A.M. Day. A dynamic cache for real-time crowd rendering. *Computer Graphics Forum*, 2010.

[19] T. Lorach. Gpu blend shapes. *NVidia Whitepaper*, 2007.

[20] N. Magnenat-Thalmann, R. Laperrire, D. Thalmann, and Université De Montréal. Joint-dependent local deformations for hand animation and object grasping. In *In Proceedings on Graphics interface'88*, pages 26–33, 1988.

[21] E. Millan and I. Rudomin. Impostors and pseudo-instancing for gpu crowd rendering. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in*

Australasia and Southeast Asia, pages 49–55, New York, NY, USA, 2006. ACM.

[22] V. Pamplona, M. Oliveira, and L. Nedel. *Game Programming Gems VII*, chapter Animating Relief Impostors Using Radial Basis Functions Textures, pages 401–412. Charles River Media, Inc., Hingham, Massachusetts, 2008.

[23] J. Pettré, P. Ciechomski, J. Maïm, B. Yersin, J. Laumond, and D. Thalmann. Real-time navigating crowds: scalable simulation and rendering: Research articles. *Comput. Animat. Virtual Worlds*, 17(3-4):445–455, 2006.

[24] F. Policarpo and M. Oliveira. Relaxed cone stepping for relief mapping. In *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 409–428. Addison-Wesley Professional, 2007.

[25] F. Policarpo, M. Oliveira, and J. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM.

[26] D. Pratt, S. Pratt, P. Barham, R. Barker, M. Waldrop, J. Ehlert, and C. Chrislip. Humans in large-scale, networked virtual environments. *Presence*, 6(5):547–564, 1997.

[27] B. Spanlang. Halca hardware acclerated library for character animation. Technical report, 2009.

[28] L. Szirmay-Kalos and T. Umenhoffer. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(1), 2008.

[29] N. Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 63–69, New York, NY, USA, 2006. ACM.

[30] F. Tecchia and Y. Chrysanthou. Real-time rendering of densely populated urban environments. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 83–88, London, UK, 2000. Springer-Verlag.

[31] F. Tecchia, C. Loscos, and Y. Chrysanthou. Image-based crowd rendering. *IEEE Comput. Graph. Appl.*, 22(2):36–43, 2002.

[32] T. Winkler, J. Drieseberg, M. Alexa, and K. Hormann. Multi-scale geometry interpolation. *Computer Graphics Forum*, 29(2):309–318, May 2010. Proceedings of Eurographics.

Appendix A

Vertex Shader

The following is the vertex shader program of our impostors in GLSL.

```
// The array of transformation matrices of the bones. 68 is the
    considered maximum number of bones.
uniform mat4 transforms[68];

// All the vertex attributes coming to the shader from the VBO

// The OBB vertex coordinates
attribute vec3 inVertex;

// The vertex texture coordinates. The third component has the
    index of the texture in the texture array corresponding to
    the current face of the OBB.  The fourth one has the index to
     the texture of the opposite face of the OBB.
attribute vec4 texCoordinates;

// Numbering of the impostor vertices:
// p2———p3
// | |
// | |
// | |
// p0———p1

// The impostor vertex attributes (also in the VBO)
attribute vec3 impostorPzero; // impostor p0 (world space)
attribute vec3 impostorNormal; // impostor normal (world space)
attribute vec3 impostorU; // vector p1-p0 (normalized)
attribute vec3 impostorV; // vector p2-p0 (normalized)
attribute vec2 impostorDepthFactor; // vec2((zfar-znear)/||p1-p0
    ||, (zfar-znear)/||p0-p2|| )
attribute float impostorDepthRange; // abs(zfar-znear)

// The bone identifying number to which is associated the
    current vertex
attribute float boneId;
```

```
// All the varying parameters that have to be passed to the
    fragment shader but applying the correspondent transform
    matrix

varying vec3 coord; // Vertex coordinates (world space)
varying vec3 impPzero;
varying vec3 normal;
varying vec3 impU;
varying vec3 impV;
varying vec2 depthFactor;
varying float depthRange;

varying float bId;

varying vec3 lightDir;

varying mat3 rotMat;

void main()
{

    int bone = int(boneId);
    bId = boneId;

    // We take the transform matrix of the current OBB
    mat4 transformMat;
    transformMat = transforms[bone];

    // The rotation matrix comes from the transform matrix
    rotMat[0] = vec3(transformMat[0]); rotMat[1] = vec3(
        transformMat[1]); rotMat[2] = vec3(transformMat[2]);

    // inVertex has the vertex coordinates from the VBO.
    // coord needs to have the final coordinates, after the
        transform matrix is applied. That is why we multiply
        inVertex by the transformMat.
    coord = (transformMat * vec4(inVertex,1.0)).xyz;

    // impostorPzero is a point and must also be multiplied by
        transformMat
    impPzero = (transformMat * vec4(impostorPzero,1.0)).xyz;

    // impU and impU are vectors and just need to be rotated.
    // So, they are multiplied by rotMat.
    normal = rotMat*impostorNormal;
    impU = rotMat*impostorU;
    impV = rotMat*impostorV;

    // These are magnitude values.
    // Since the animations are rigid (translation + rotation),
        they do not need to be transformed.
    //If scaling was permitted, they would.
    depthFactor = impostorDepthFactor;
    depthRange = impostorDepthRange;
```

```
    // The texture coordinates do not change because of the
        animation
    gl_TexCoord[0] = texCoordinates;

    // We create a light direction vector
    lightDir = normalize(vec3(0,-1,1));

    // We finally apply the camera transformation to the vertex
        coordinates
    gl_Position = gl_ModelViewProjectionMatrix * vec4(coord,1.0)
        ;
}
```

Appendix B

Fragment Shader

The following is the fragment shader program of our impostors in GLSL. Due to its extension we have divided it in three parts. The first one has the declaration of all the input variables. To better understand them we refer to Figure 3.15. The second part includes the function performing the ray-heightfield intersection search with a linear search followed by a binary one. The third and last part includes the main function of the fragment shader.

B.1 Input

```
// Fragment shader used in the paper "Omnidirectional Relief
    Impostors"
// Modified for the thesis "Output-Sensitive Rendering of
    Detailed Animated Characters for Crowd Simulation"

// The array of transformation matrices of the bones. 68 is the
    considered maximum number of bones. It is used here to
    compute the correct z value of the fragment
uniform mat4 transforms[68];

// The number of steps of the linear and binary search are
    uniforms set by the user.
uniform int linear_search_steps; // 16 is recommended
uniform int binary_search_steps; // 7 is recommended

// The 2 texure arrays with the relief textures
uniform sampler2DArray sampler0; // RGBZ texture array (front
    relief map)
uniform sampler2DArray sampler1; // normal map array (back
    relief map)

uniform vec3 eye; // viewpoint coordinates (world space)

// The bone identifying number to which is associated the
    current fragment
```

```
varying float bId;

varying vec3 coord; // fragment coordinates (world space)

// Numbering of the impostor vertices:
// p2————p3
// |  |
// |  |
// |  |
// p0————p1

// The impostor parameters
varying vec3 impPzero; // impostor p0 (world space)
varying vec3 normal; // impostor normal (world space)
varying vec3 impU; // vector p1-p0 (normalized)
varying vec3 impV; // vector p2-p0 (normalized)
varying vec2 depthFactor; // vec2((zfar-znear)/||p1-p0||, (zfar-
    znear)/||p0-p2|| )
varying float depthRange; // abs(zfar-znear)

varying vec3 lightDir; // The light direction vector

varying mat3 rotMat; // The rotation matrix from the animation (
    to be applied to the normal and compute correct lightning)
```

B.2 Ray-Heightfiel Intersection Search

```
// Ray-surface intersection code. Based on the paper by
    Policarpo, Oliveira, Comba, Real-Time Relief Mapping on
    Arbitrary Polygonal Surfaces, I3D 2005

void ray_intersect_binary_search(vec2 dp, vec2 ds, out float
    best_depth, out float best_depth_outside)
{
    float depth_step = 1.0 / float(linear_search_steps);
    float size = depth_step;
    float depth = 0.0;
    best_depth = 1.0;
    best_depth_outside = 0.0;

    // linear search
    // search from front to back for first point inside the
        object

    bool found = false;
    for ( int i=0; i<linear_search_steps-1 && !found ;i++)
    {
        depth += size;
        vec2 newTexel = dp+ds*depth;

        // if the new texel has coordinates inside [0;1]
```

```
        if (newTexel.s<=1.0 && newTexel.s>=0.0 && newTexel.t
            <=1.0 && newTexel.t>=0.0)
        {
            // the current layer of the texture array is encoded
                 in the 4th texture coordinate
            vec4 t= texture(sampler0,vec3(newTexel,gl_TexCoord
                [0].p)); // The front depth
            vec4 t2= texture(sampler1,vec3(newTexel,gl_TexCoord
                [0].p)); // The back depth

            // if front and back depths are equal, we are on a
                special case where the ray might be tangent to
                the surface
            // therefore, we just check the intersection with
                the front depth
            if (t.a == t2.a)
            {
                if (t.a <= depth) // inside
                {
                    found = true;
                    best_depth=depth; // store best depth
                }
            }
            else // we check if we are inside the surface using
                the two depths
            {
                if (t.a <= depth && t2.a >= depth) // inside
                {
                    found = true;
                    best_depth=depth; // store best depth
                }
            }
        }
        else // if we are out of the texture
            discard;
    }

    best_depth_outside = depth - size;

    // if intersection not found -> discard the fragment
    if (!found) discard;

    depth = best_depth;

    // binary search around first point (depth) for closest
        match
    for ( int i=0; i<binary_search_steps;i++)
    {
        size*=0.5; // binary search
        vec2 newTexel = dp+ds*depth;

        vec4 t=texture(sampler0,vec3(newTexel,gl_TexCoord[0].p))
            ;
```

```
        vec4 t2=texture(sampler1,vec3(newTexel,gl_TexCoord[0].p)
            );

        // as before, if front and back depths are equal, we are
             on a special case where the ray might be tangent to
            the surface
            // therefore, we just check the intersection with
                the front depth
        if (t.a == t2.a)
        {
            if (t.a <= depth) // still inside
            {
                best_depth = depth; // best_depth inside
                depth -= 2.0 * size;
            }
            else
                best_depth_outside = depth;
        }
        else // we check if we are inside the surface using the
            two depths
        {
            if (t.a <= depth && t2.a >= depth) // still inside
            {
                best_depth = depth; // best_depth inside
                depth -= 2.0 * size;
            }
            else
                best_depth_outside = depth;
        }
        depth+=size;
    }
}
```

B.3 Main

```
void main()
{
    vec3 v = normalize(coord - eye); // view ray

    vec3 vp = normalize( vec3(dot(impU,v),dot(impV, v),-dot(
        normal,v)));

    vec2 startTexel = gl_TexCoord[0].st;

    vec2 dir = depthFactor * vp.xy / vp.z;

    float d_in, d_out;
    ray_intersect_binary_search(startTexel, dir, d_in, d_out);
    // after the search, d_in has the depth just inside the
        surface and d_out has the best_depth just outside the
```

```
        surface.

    vec2 st_in = startTexel + d_in*dir;
    vec2 st_out = startTexel + d_out*dir;

    vec4 color_in = texture(sampler0, vec3(st_in,gl_TexCoord[0].
        p));
    vec4 color_out = texture(sampler0, vec3(st_out,gl_TexCoord
        [0].p));

    // we select the best of the two depths (inside or out)
    vec2 st;
    vec4 color;
    if (abs(color_in.a - d_in) < abs(color_out.a - d_out))
    {
        st = st_in;
        color = color_in;
    }
    else
    {
        st = st_out;
        color = color_out;
    }

    vec3 surfaceNormal = texture(sampler1, vec3(st,gl_TexCoord
        [0].p)).xyz;

    // the surface normal is encoded with values between [0,1],
        we need to transform it to have them between [-1,1].
    // to have a correct lightning we have to rotate the normal
        with the animation transformation and multiply it by the
        normal matrix.
    surfaceNormal = gl_NormalMatrix*rotMat*(surfaceNormal*2.0 -
        vec3(1.0, 1.0, 1.0));
    // we apply a simple lightning
    float intens = dot(lightDir,surfaceNormal)*1.20;

    vec4 resColor=vec4(color.rgb*intens, 1.0);

    gl_FragColor = resColor;

}
```