Polypostors: 2D Polygonal Impostors for 3D Crowds

Ladislav Kavan* [1] Simon Dobbyn [1] Steven Collins [1] Jiří Žára [2] Carol O'Sullivan [1]

[1] Trinity College Dublin, [2] Czech Technical University in Prague

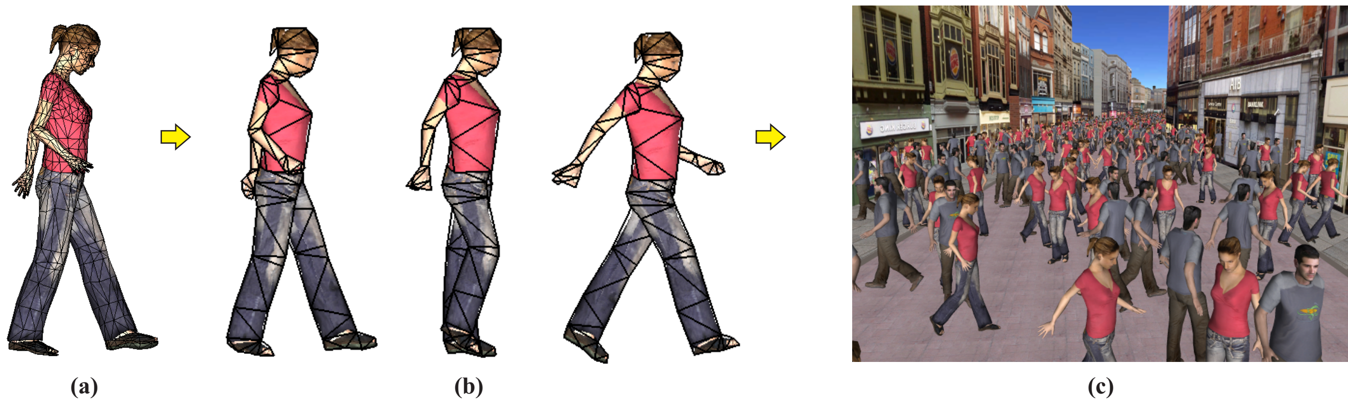

Figure 1: Our algorithm converts a 3D polygonal character (a) to 2D textured polygons, animated efficiently by displacing their vertices with associated texture coordinates (b). This view-dependent representation (which we call Polypostors) achieves dramatic simplification with low texture memory overhead. The intended application is real time rendering of large crowds (c).

Abstract

Various methods have been proposed to animate and render large crowds of humans in real time for applications such as games and interactive walkthroughs. Recent methods have been developed to render large numbers of pre-computed image-based human representations (*Impostors*) by exploiting commodity graphics hardware, thus achieving very high frame-rates while maintaining visual fidelity. Unfortunately, these images consume a lot of texture memory, no in-betweening is possible, and the variety of animations that can be shown is severely restricted. This paper proposes an alternative method that significantly improves upon pre-computed impostors: automatically generated 2D polygonal characters (or *Polypostors*). When compared with image-based crowd rendering systems, Polypostors exhibit a similarly high level of rendering efficiency and visual fidelity, with considerably lower memory requirements (up to a factor of 30 in our test cases). Furthermore, Polypostors enable simple in-betweening and can thus deliver a greater variety of animations at any required level of smoothness with almost no overhead.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: crowd animation, impostors, animation compression

*e-mail: kavanl@cs.tcd.ie

1 Introduction

In recent years, we have seen impressive computer generated crowd scenes in feature films. The next challenge is to achieve the same functionality in real time.

The problem with crowd rendering is that due to the usual shape of the viewing frustum, the vast majority of characters to be rendered are far from the viewer and thus appear small. Level-of-detail techniques offer a direct solution to this problem, by reducing the complexity of 3D animated geometry in proportion to distance to the viewer [DeCoro and Rusinkiewicz 2005]. However, there are lower limits on the maximal possible simplification [McDonnell et al. 2005]. Intuitively, the human body is a complex object and thus cannot be convincingly represented in 3D using just a few triangles.

Image-based crowd rendering has recently gained in popularity [Tecchia and Chrysanthou 2000]. This is because it allows extremely fast drawing – just one textured quad per character, and the preparation of impostors is fully automatic. The drawback is in the consumption of texture memory, which is unfortunate especially when we realize that the stored images are actually highly correlated; decorrelation or interpolation of the images is expensive and usually prohibitive with a given time budget.

This paper proposes a novel representation of virtual characters, somewhere between geometry and impostors. The idea is to use 2D animated polygonal characters, pre-generated for a discrete set of viewing directions (as with classical impostors). However, unlike those impostors that store an image per animation key-frame, Polypostors are animated by displacing vertices of 2D polygons (using just one texture for the whole animation). Since vertex displacements can be stored very compactly, this results in much more efficient memory usage. Also, interpolation becomes an easy task, computable entirely by vertex shaders. The advantage of Polypostors over 3D geometry rendering is that in the 2D domain, much more drastic simplification is possible (e.g., our Polypostors, as shown in Figure 1, use 90 triangles per character, while the orig-

inal 3D mesh uses 3198 triangles).

We describe an algorithm for converting 3D animated character models to 2D textured polygons (for a given viewing direction). The input to this algorithm is a 3D character (using either bone based skinning or animated mesh), cut into several body parts in order to resolve occlusion issues. For the first frame of the animation, each body part is rendered and converted into textured 2D polygons. 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. At run-time, the deformed polygons are composited in depth order, creating the illusion of an animated 3D character.

2 Related Work

This section presents a brief overview of related work on real time crowd rendering. For a more detailed survey please refer to [Ryder and Day 2005] or [Thalmann et al. 2006].

Geometric Level Of Detail (LOD) is a technique that has been used to improve the performance of real time crowd rendering [Ulicny et al. 2004]. This technique reduces the number of rendered polygons per frame by using several representations of an object with decreasing complexity. However, care has to be taken when generating these low resolution meshes, as removing too much detail can produce blocky results. Animation artifacts due to the loss of joint vertices can also occur, thus reducing the overall visual realism of the virtual human. Additionally, it has been found that a low resolution model is not perceptually equivalent to its high resolution counterpart at conveying subtle variations in motion [McDonnell et al. 2005], illustrating the importance of accounting for animation when selecting LOD schemes. Billboard clouds present an appealing alternative to extreme geometric LOD simplification [Décoret et al. 2003].

Another problem with rendering thousands of animated meshes is the cost associated with each API call. To address this issue, Gosselin et al. [2005] reduce the number of API calls needed to draw multiple characters' meshes by packing a number of instances of character vertex data into a single vertex buffer. A recent NVIDIA whitepaper [Dudash 2007] applies a new DirectX 10 feature called *instancing* in order to overcome the same problem.

Planar impostors have been widely used in crowd rendering since they provide a good visual approximation to complex objects at a fraction of the rendering cost. In [Tecchia and Chrysanthou 2000; Tecchia et al. 2002], pre-generated impostors are used for rendering several thousand virtual humans walking around a virtual city at interactive frame-rates. This involves pre-rendering an image of a character for a collection of viewpoints around the object for multiple key-frames of animation. At run-time, the most appropriate viewpoint image is selected for the current key-frame and displayed on a quadrilateral, which is dynamically oriented towards the viewer. However, the main drawback of this approach is the amount of texture memory consumed (which depends on the number of viewpoints and key-frames at which the impostor images are pre-generated). While dynamically generated impostors [Aubel et al. 2000] use less memory (since no storage space is devoted to any impostor image that is not actively in use), this method only works for crowds of individuals with similar orientation and animation, since it relies on reusing the current dynamically generated image over several frames in order to be efficient.

The primary visual problem when using an impostor to represent a virtual human is that, once the human is close to the viewpoint, the impostor's flat and pixellated appearance becomes quite obvious. To solve this problem, Dobbyn et al. [2005] developed the Geopostor system, which provides for a hybrid combination of pre-generated impostor and detailed geometric rendering techniques for virtual humans. By switching between the two representations, based on a pixel to texel ratio, their system allows visual quality and performance to be balanced.

Textured depth meshes [Jeschke and Wimmer 2002] are similar to the Polypostor representation as they simplify an object by triangulating a rendered image of the object based on the image's depth values. However, they are expensive to generate and can also suffer from occlusion artifacts resulting in image gaps. While this representation can accelerate the rendering of static polygonal models, it is not suitable for animated crowds. Other related research is the interactive system developed by Igarashi et al. [2005] which allows a user to deform a 2D shape without using a skeleton or freeform deformation. By triangulating an image of the shape, the system allows the user to create simple 2D animations by setting the position of the vertices for each key-frame while minimizing distortion. Polypostors can also be considered as a method of animation compression [Alexa and Müller 2000; Briceno et al. 2003].

At the core of our technique is an algorithm for matching two textured polygons (such as those in Figure 2) in an optimal way with respect to a chosen error metric. This is reminiscent of the problem of morphing and especially of the solution [Sederberg and Greenwood 1992], which also uses dynamic programming. However, Sederberg and Greenwood's approach, as well as more modern methods [Liu et al. 2004], find correspondences according to the deformation of a polygon's outline, i.e., a planar curve. In contrast, our algorithm matches a source polygon with the target one based on the deformation of its interior, i.e., the texture. In the computer vision community, a similar algorithm is used to detect shapes in images [Felzenszwalb 2003] (which is, however, a much more computationally complex process). Also, an idea similar to Polypostors has been proposed for person tracking [Ju et al. 1996].

3 Polypostor construction

As input, we have a 3D animated character. Before constructing the Polypostor representation, the character mesh is cut into several pieces, to allow us to account for visibility issues (Section 3.2).

In our experiments, we manually decompose the character into two legs, two arms and torso, and cap the holes that result from disconnecting the mesh. The caps are manually textured so that if visible they will not produce unpleasant artifacts. The original skeletal animation is then applied to the body parts – when composed together, they give exactly the same animation as was originally provided (the hole capping is performed because subsequent steps may introduce errors that can cause the holes to become visible).

The first step in generating Polypostors is the same as when creating impostors: discretize the set of viewing directions. This is accomplished by placing the segmented 3D character at the origin and enclosing it within a hemisphere. This hemisphere is sampled by a fixed number of points that are uniformly distributed [Rusin 1998]. They correspond to camera positions with the camera oriented towards the origin (twist around the viewing axis is not considered). For every such direction, a Polypostor is generated. Our actual implementation uses 137 camera directions (Polypostors) per character (in order to obtain comparable results with Geopostors [Dobbyn et al. 2005], where 136 viewing directions are used).

We restrict the following discussion to one fixed viewing direction

and one specific body part. The final algorithm simply iterates the process over all viewing directions and all body parts. We pose the character according to the first frame of its animation. The body part in question is rendered from the given camera direction at a fixed resolution (given by the maximal projected size of our characters; we use 128×128). Using two special purpose fragment shaders, we separately render a color and a normal image, which will enable us to achieve more realistic lighting at run-time [Dobbyn et al. 2005].

The next step encloses each rendered body part within a polygon (not necessarily convex, but without self-intersections). To this end, we apply a standard contour tracing algorithm [Pavlidis 1981]. This yields a highly-detailed boundary representation. In order to reduce the number of vertices, we use standard simplification techniques (adapted to 2D) [Luebke et al. 2002]. In particular, we remove vertices in a greedy way, i.e., we always discard the vertex **v** that has the smallest distance from the edge formed by omitting **v**. However, we also take into account the ratio of the longest to shortest edge, in order to encourage uniform distribution of resulting vertices. In our experiments, this simplification is terminated when we have reduced the polygons to 40 vertices. We denote this polygon as S (source) and tessellate it using constrained Delaunay triangulation (see Figure 2 left). We do not allow any internal vertices, which is important for polygon fitting (Section 3.1). Note that the final simplification (producing the primitives that will actually be rendered) will be computed subsequently.

The same process is repeated for every other frame of the animation (sampled at 10Hz in our experiments), simply producing more vertices for higher accuracy (in our case 50). This results in a textured polygon such as in Figure 2 right, which we will denote as T (target). The task now is to map vertices from S (representing the first frame of the animation) to vertices T (representing some arbitrary frame in the animation) so that the stretched source texture matches the target one as closely as possible. Our first approach was to exploit the existing 3D model in order to guide the fitting process (basically, binding the 2D vertices to the 3D mesh). Unfortunately, we did not find this approach to be sufficiently robust. The main problem was that 3D vertices associated with vertices visible in the source polygon can become occluded in the target polygon and vice-versa. Therefore, we resort to a slower yet robust algorithm that finds the mapping from S to T with minimal image-based error (see Section 3.1). Specifically, this error is defined as the image correlation [Gonzalez and Woods 2002] between the stretched source texture with the (unstretched) target one, e.g., the sum of squared differences of pixel intensities.

After the vertices have been mapped (and therefore vertex correspondences established for all frames), we perform deformation sensitive simplification of S to its final form (in our case, to 20 vertices). We use the same simplification algorithm as before, only averaging the error metric over all frames and leaving out the uniformity requirement (i.e., the longest to shortest edge ratio). According to our experiments, this produces a well adapted polygon – with higher concentration of vertices in more deformable areas. This polygon is then re-tessellated using constrained Delaunay triangulation and stripified using NVIDIA's NvTriStrip library for efficient rendering. Note that the initial pre-simplification was done in order to cut down the costs of the polygon mapping algorithm (Section 3.1). Varying the number of vertices produced during pre-simplification enables us to trade speed for accuracy.

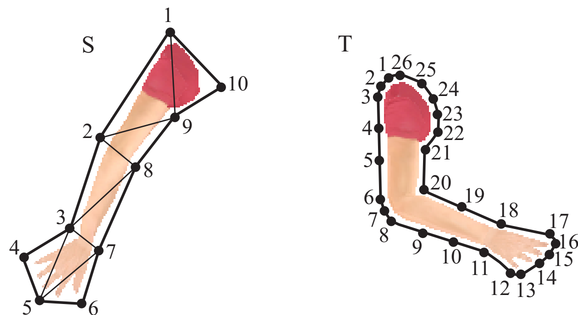

Figure 2: Polygon fitting: the task is to map vertices from source polygon (S) to the target one (T) so that the textures match as closely as possible.

3.1 Polygon Fitting

In this section, we present our algorithm to find a mapping P from vertices $1,\ldots,n$ of polygon S to vertices $1,\ldots,m$ of polygon T so that the given error metric is minimized. Polygon S has an associated triangulation; it is not difficult to show that the number of triangles in S will always be $n-2$. For the example in Figure 2, $n = 10$, $m = 26$ and the resulting optimal mapping determined by our algorithm would be $P(1) = 2$, $P(2) = 7$, $P(3) = 11$, $P(4) = 12$, $P(5) = 14$, $P(6) = 16$, $P(7) = 18$, $P(8) = 20$, $P(9) = 21$ and $P(10) = 24$. Note that it is possible to use a more general error metric than the one mentioned above, as long as it is computable by summing errors on individual triangles.

In theory, there are m^n possible mappings P (we generally allow triangles to degenerate). Therefore, a brute force solution is not an option, even though polygon fitting is done as a pre-process. In the following, we present an algorithm based on dynamic programming with time complexity $O(nm^3)$.

The algorithm starts by ordering triangles from S by sequentially removing triangles with only one neighbor (at least one such triangle always exists, because we did not allow internal vertices). For example, one possible ordering of triangles in Figure 2 is: 1-9-10, 2-9-1, 2-8-9, 3-8-2, 3-7-8, 5-3-4, 5-7-3 and the last triangle is 5-6-7. The triangles are then fitted to T, one by one, and the ordering ensures that all neighbors of the actual triangle have already been processed – up to one. For example, when fitting triangle 5-7-3, it is important that both neighbors 3-7-8 and 5-3-4 have already been optimized.

We will call the triangle's edge shared with the unprocessed neighbor the *base* edge. Let us assume that we are now processing triangle *a*-*b*-*c*, where *a*-*b* is the base edge and *c* is its opposite vertex. For simplicity, let us consider the first step of the algorithm, where $a = 1$, $b = 9$, $c = 10$ and there are no processed neighbors. Every base edge is a chord of S and thus cuts the polygon S into two subpolygons, one on either side of the chord. In the first step, one subpolygon is triangle 1-9-10 and the other subpolygon consists of the remaining triangles in S. We examine all m^2 possible choices of $P(a)$ and $P(b)$ and for each of them, we try all possibilities of $P(c)$, evaluating the error metric of the subpolygon created by chord *a*-*b*. We store the optimal $P(c)$ together with its fitting error for every $P(a)$, $P(b)$ (thus we need a table of size $O(m^2)$ for each base edge).

Treatment of the remaining triangles is similar, requiring only that we account for the already processed neighbors. We proceed as before, finding optimal choices of $P(c)$ for each $P(a)$, $P(b)$ pair. For example, for triangle $a = 2$, $b = 9$, $c = 1$ (see Figure 3), the subpolygon of *a*-*b* is quad 2-9-10-1. The error of mapping triangle *a*-*b*-*c* to

$P(a)$-$P(b)$-$P(c)$ is computed as before, using our error metric. To account for the remaining part of the quad, i.e., triangle 1-9-10, we make use of the fact that the edge 1-9 has already been optimized in the previous step. That is, for every choice of $P(1)$ and $P(9)$, the optimal $P(10)$ is readily available in the table computed previously. Because of this, the resulting time complexity is the same as in the first step, i.e., $O(m^3)$.

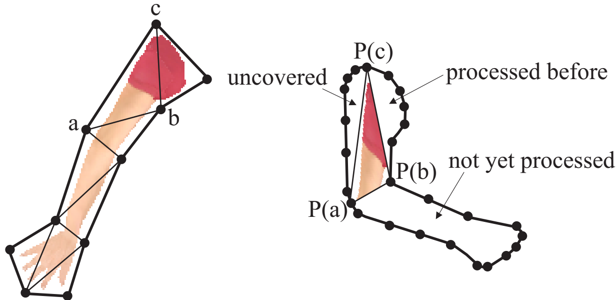

Figure 3: Fitting triangle a-b-c from the source polygon to triangle $P(a)$-$P(b)$-$P(c)$ in the target polygon. Obviously, in this case the choice of $P(a)$-$P(b)$-$P(c)$ is not the optimal one.

After processing the last base edge (in our example $a = 5$, $b = 6$), we can find the optimal mapping of the whole polygon by searching for the $P(a)$ and $P(b)$ with the smallest error. A table lookup gives us the optimal position of $P(c)$, i.e., in our case $P(7)$. Then we move to triangle 5-7-3, retrieve the optimal value of $P(3)$ and so on, processing triangles in reverse order. This "backtracking", typical with dynamic programming algorithms, finally reveals the optimal mapping P of all source vertices for our given error metric.

The basis for our error metric is image correlation. The correlation is computed by taking a fixed number of samples on triangles from S (for example, using scan conversion) and matching them with their counterparts from T. We compute both color and normal map correlation and sum the results together. This is motivated by the desire for color as well as geometric feature correlation – the normal map allows us to encode geometry in the correlation metric. This also improves robustness of our method in situations with little color map variation.

Each body part generates two images (color and normal map), which leads in this case to a total of 10 images per Polypostor. Creating a separate texture for each of them would result in wasting texture memory and the necessity of frequent texture binding. Therefore, we pack all images associated with one Polypostor into one texture (see Figure 4). This leads to the well known nesting problem (also called polygon packing) [Nielsen and Odgaard 2003]. In our current implementation, we apply only a simple brute-force approach, i.e., we translate incoming polygons from the lower-left to the upper-right texture corner (by fixed-length steps), terminating at the first intersection-free position.

3.2 Visibility

Theoretically, pixel-precise occlusions can be achieved using depth maps [Aubel et al. 2000]. Even though this method can be implemented in fragment shaders, it would require additional memory and introduce another constraint for the Polypostors fitting algorithm. Therefore, we settle for an approximate but more efficient visibility solution, based on our decomposition of the 3D character into several body parts and depth sorting using painter's algorithm [Foley et al. 1990]. The body parts are converted into textured 2D

Figure 4: Two example textures (for two different viewpoints), generated by our system; each one stores both color and normal maps.

polygons as described earlier. Now, it remains to find the ordering of these polygons which results in the most accurate visibility reconstruction. Note that we are implicitly making the assumption that the same ordering will be valid in the whole animation.

In order to achieve this ordering, we construct a visibility graph, using the original 3D model. Nodes of this graph correspond to individual body parts. Two nodes A, B are connected by a directed edge if A occludes B, and the weight of this edge is the number of B pixels occluded by A (averaged over all frames of the animation). If the resulting graph is acyclic, we have a well-defined order, because its topological sorting yields a (perfectly correct) visibility ordering. If the graph contains cycles, we determine the best ordering by deleting such subset of edges that yields an acyclic graph and has the smallest total weight. Since in our case visibility graphs have only five nodes, we do this by examining all subsets of edges (in case of more body parts, a polynomial-time algorithm [Bertsekas 1991] can be used instead).

4 Run-time System

Once Polypostors have been pre-computed (see Figure 5) their runtime display is fairly simple. For each character to be drawn, we first compute its distance from the camera and determine whether to render geometry or Polypostor. The switching distance is selected so that we obtain a one to one pixel to texel ratio, as in [Dobbyn et al. 2005]. Geometry is drawn using standard skinned meshes, with skinning computed on the GPU [Lindholm et al. 2001].

When rendering Polypostors, the first step is to determine the actual Polypostor plane, i.e., the one most perpendicular to the actual viewing direction. This is the same method as would be used for classic impostors. Subsequently, we bind the appropriate texture (such as in Figure 4) and draw triangle strips for each body part. Rendering of the body parts in the order described in Section 3.2 is not sufficient, because Z-fighting artifacts would creep in. To display the resulting character correctly, we perturb the depth values in a fragment shader based on the pre-computed ordering. Note that this is generally not recommended, as it usually results in disabling early-Z, typically associated with a performance hit. However, this was not a problem in our case, as our fragment shader is very simple. With more complex fragment shaders, it would be advisable to perturb vertex positions rather than fragment depths.

Interpolation of Polypostors is handled in a vertex shader, simply by linearly interpolating polygon vertices. The fragment shader combines information from the color and normal maps with the light direction in order to compute shading. If required, it can also vary the characters' appearance using Dobbyn et al.'s method [2006].

For rendering of both Polypostors and skinned meshes we employ

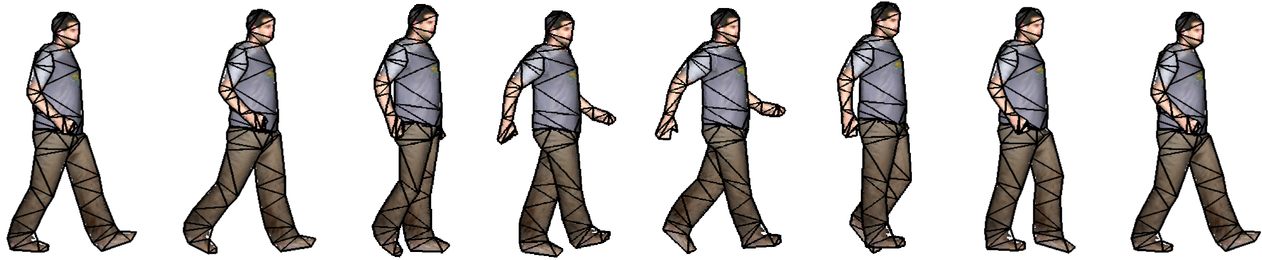

Figure 5: An example of a polypostor animation (overlaid with wireframe). Note that the character animation is created simply by displacing polygon vertices (stretching the texture accordingly).

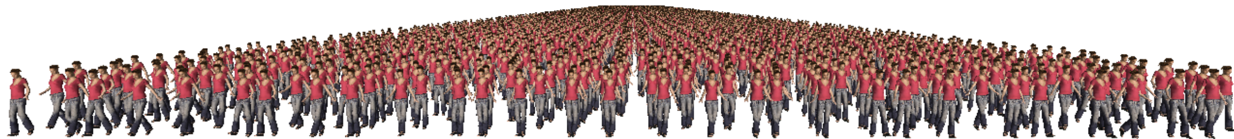

Figure 6: The testing scenario used to compare run-time efficiency of different character representations (10.000 humans in this picture).

OpenGL display lists. The performance of the Polypostor renderer can be slightly improved by batching, i.e., drawing multiple entities in a single draw call. In order to make use of batching, we first sort the array of characters based on their type and viewpoint, thereby grouping characters that can be drawn together. In our current implementation, we use batches consisting of only four characters, as we observed only negligible improvement with larger batches. This is probably because the current bottleneck is in transferring the character animation data from CPU to GPU. According to our experiments, batching did not improve performance of skinned meshes considerably, probably because their bottleneck is in the vertex processing (rather than in the CPU overhead associated with draw calls).

4.1 Experiments and Results

We measured the run-time performance of different virtual human representations using a Pentium D 3.7GHz processor with 2GB RAM and a GeForce 7950 graphics card with 512MB of video memory. In our first test (see Figure 6), the whole crowd is onscreen and all characters are using the same representation (i.e., either Impostors, Polypostors or skinned meshes). For Impostors, we use the same format as described in [Dobbyn et al. 2005]. With Polypostors, every character uses 90 triangles (18 per body part), while the skinned mesh has 3198 triangles for the female model (2476 in the case of the male model).

The results for crowd sizes from two to twenty thousand individuals are reported in Figure 7. From the graph, we see that rendering with Polypostors is almost as fast as with impostors. This is because the geometric complexity of Polypostors does not present a bottleneck in the rendering pipeline (in contrast to the case of unsimplified skinned meshes), and the per-pixel complexity is equivalent.

The total memory requirements for impostors and Polypostors are compared in Figure 8. The reported values account for all viewpoints of one character and one animation. Impostors animated at 10Hz do not always assure smooth animation (see the accompanying video). This can be improved by generating impostors at higher frequency, but at a cost of increased memory consumption. This is

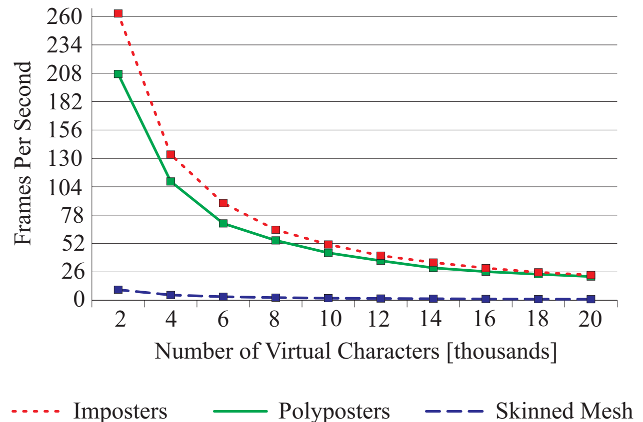

Figure 7: Run-time performance for different representations of virtual characters.

unfortunate because when the impostors do not fit into video memory, texture thrashing occurs, slowing the application down (thereby virtually defeating the advantage of using impostors). In contrast to impostors, Polypostors can be easily interpolated, thus allowing an arbitrary output frequency. The Polypostors' vertex animation is sampled at 10Hz, which is probably more than sufficient for our animations (although different sampling rates would not affect the memory consumption considerably, as the Polypostor vertex data is quite compact).

In our next experiment, we used a mixed geometry/Polypostor rendering within a virtual city (see Figure 1c). The city model itself contains over 100,000 polygons and uses over 200MB of textures. In this experiment, we took advantage of both frustum and occlusion culling (based on hardware occlusion queries). The resulting frame-rates for varying crowd sizes are reported in Figure 9. We see that real time frame-rates are maintained for crowds of up to 120,000 virtual characters (however, note that only fraction of all these characters is visible at each frame).

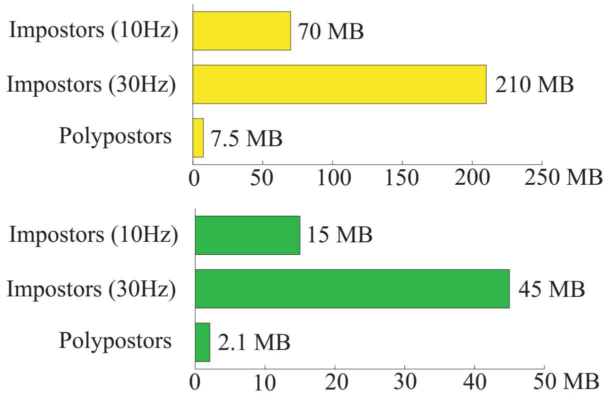

Figure 8: Total memory (in megabytes) consumed by one character and one animation. Top: uncompressed textures, bottom: compressed textures (using DXT3 compression).

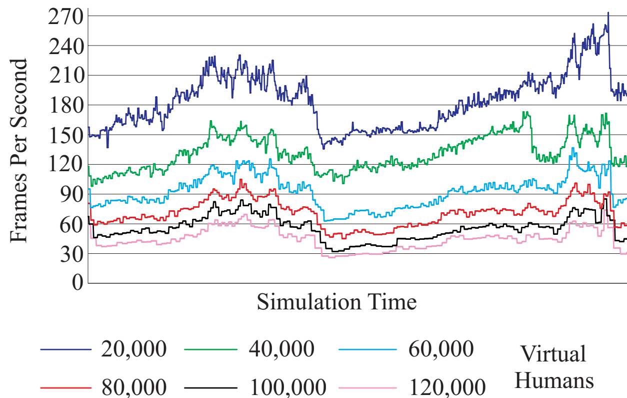

Figure 9: Run-time performance during a walkthrough in a virtual city for different numbers of characters (combined geometry/Polypostor rendering).

5 Limitations and Future Work

Polypostors are not as accurate as impostors, as they use only one texture per viewpoint and approximate the rest of the animation by deforming the texture. This implies that Polypostors are applicable only for animations that can be described as deformations of the initial key-frame. According to our experiments, this works well for simple walk cycles. However, we have not tested more challenging animations.

Polypostors may produce artifacts, especially with overhead views where there is a lack of texture information in the first key-frame, as shown in Figure 10. A simple remedy would be to identify the key-frame which contains the best texture information, then use it instead of the first frame. The extreme case would be to add as many textures as used by impostors. This would not result in memory saving, but would retain the benefits of efficient interpolation. An interesting avenue of future work would be to find the best compromise between these two extremes, i.e., to store only the relevant texture data while avoiding replication.

Another slight disadvantage of Polypostor construction is the necessity of manually disconnecting the mesh into close-to-convex parts. This could be automatized, e.g, using the approach of Lien and Amato [2006], in order to obtain a fully automatic Polypostor generation pipeline. Polypostors used in our current implementation have a fixed size of 90 triangles per character. However, this number can be reduced (particularly for distant Polypostors) by applying geometric level-of-detail techniques.

Figure 10: Left: first key-frame used to generate Polypostor texture, right: subsequent key-frame generated by deforming the texture data from the first frame. Note the artifacts due to lack of texture data in areas that have become visible.

Further speed-up could be achieved by executing the crowd simulation on multi-core processors or even on GPUs, as discussed recently [Reynolds 2006; Millan and Rudomin 2006]. This should help to eliminate our current CPU to GPU transfer bottleneck. Finally, a three level system could be considered, using skinned meshes in the near field, Polypostors in the middle and impostors in the distance. Since this would only need impostor images with low resolutions, their memory requirements would not be excessive.

6 Conclusions

This paper proposes Polypostors, a low-cost character animation representation based on 2D polygons. We present an algorithm that automatically constructs Polypostors from a segmented 3D character, thus reducing its geometric complexity more than would be possible with 3D simplification techniques. While this comes with a memory overhead cost, these requirements are significantly less than those associated with classical impostors. Moreover, Polypostors offer efficient GPU implementation and support animation interpolation at almost no extra cost. Our proposed system achieves real time frame-rates for crowds of up to 120,000 individuals, as we have verified in our virtual city simulation.

7 Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by Science Foundation Ireland (project Metropolis).

References

ALEXA, M., AND MÜLLER, W. 2000. Representing animations by principal components. *Comput. Graph. Forum 19*, 3, 411–418.

AUBEL, A., BOULIC, R., AND THALMANN, D. 2000. Real-time display of virtual humans: Levels of details and impostors. *IEEE Transactions on Circuits and Systems for Video Technology 10*, 2, 207–217.

BERTSEKAS, D. P. 1991. *Linear network optimization: algorithms and codes*. MIT Press, Cambridge, MA, USA.

BRICENO, H. M., SANDER, P. V., MCMILLAN, L., GORTLER, S., AND HOPPE, H. 2003. Geometry videos: a new representation for 3D animations. *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 136–146.

DÉCORET, X., DURAND, F., SILLION, F. X., AND DORSEY, J. 2003. Billboard clouds for extreme model simplification. *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 689–696.

DECORO, C., AND RUSINKIEWICZ, S. 2005. Pose-independent simplification of articulated meshes. In *SI3D '05*, ACM Press, 17–24.

DOBBYN, S., HAMILL, J., O'CONOR, K., AND O'SULLIVAN, C. 2005. Geopostors: a real-time geometry / impostor crowd rendering system. In *SI3D '05*, ACM Press, 95–102.

DOBBYN, S., MCDONNELL, R., KAVAN, L., COLLINS, S., AND O'SULLIVAN, C. 2006. Clothing the masses: Real-time clothed crowds with variation. In *Eurographics Short Papers*, 103–106.

DUDASH, B., 2007. Skinned instancing. NVIDIA Direct3D SDK 10 Code Samples.

FELZENSZWALB, P. F. 2003. *Representation and Detection of Shapes in Images*. PhD thesis, Massachusetts Institute of Technology.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GONZALEZ, R. C., AND WOODS, R. E. 2002. *Digital Image Processing*. Prentice Hall.

GOSSELIN, D., SANDER, P., AND MITCHELL, J. 2005. *ShaderX3 - Drawing a Crowd*. Charles River Media, 505–517.

IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. As-rigid-as-possible shape manipulation. *ACM Transactions on Computer Graphics 24*, 3, 1134–1141.

JESCHKE, S., AND WIMMER, M. 2002. Textured depth meshes for real time rendering of arbitrary scenes. *EGRW '02: Proceedings of the 13th Eurographics Workshop on Rendering*, 181–190.

JU, S. X., BLACK, M. J., AND YACOOB, Y. 1996. Cardboard people: A parameterized model of articulated motion. *International Conference on Automatic Face and Gesture Recognition*, 38–44.

LIEN, J.-M., AND AMATO, N. M., 2006. Approximate convex decomposition of polyhedra. Technical report TR06-002, Parasol Lab, Texas A&M University.

LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 149–158.

LIU, L., WANG, G., ZHANG, B., GUO, B., AND SHUM, H.-Y. 2004. Perceptually based approach for planar shape morphing. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, IEEE Computer Society, Washington, DC, USA, 111–120.

LUEBKE, D., WATSON, B., COHEN, J. D., REDDY, M., AND VARSHNEY, A. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA.

MCDONNELL, R., DOBBYN, S., AND O'SULLIVAN, C. 2005. LOD human representations: A comparative study. *Proceedings of the First International Workshop on Crowd Simulation*, 101–115.

MILLAN, E., AND RUDOMIN, I. 2006. 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*, ACM Press, New York, NY, USA, 49–55.

NIELSEN, B. K., AND ODGAARD, A., 2003. Fast neighborhood search for the nesting problem. Technical Report no. 03/02, DIKU, University of Copenhagen.

PAVLIDIS, T. 1981. *Algorithms for Graphics and Image Processing*. Computer Science Press.

REYNOLDS, C. 2006. Big fast crowds on PS3. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, ACM Press, New York, NY, USA, 113–121.

RUSIN, D., 1998. Topics on sphere distributions. `http://www.math.niu.edu/~rusin/known-math/95/sphere.faq`.

RYDER, G., AND DAY, A. M. 2005. Survey of real-time rendering techniques for crowds. *Computer Graphics Forum 24*, 2, 203–215.

SEDERBERG, T. W., AND GREENWOOD, E. 1992. A physically based approach to 2-D shape blending. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 25–34.

TECCHIA, F., AND CHRYSANTHOU, Y. 2000. Real-time rendering of densely populated urban environments. *Proceedings of the Eurographics Workshop on Rendering Techniques*, 83–88.

TECCHIA, F., LOSCOS, C., AND CHRYSANTHOU, Y. 2002. Visualizing crowds in real-time. *Computer Graphics Forum 21*, 4, 753–765.

THALMANN, D., O'SULLIVAN, C., DE HERAS CIECHOMSKI, P., AND DOBBYN, S. 2006. Populating virtual environments with crowds. In *Eurographics 2006: Tutorials*, 869–963.

ULICNY, B., DE HERAS CIECHOMSKI, P., AND THALMANN, D. 2004. Crowdbrush: Interactive authoring of real-time crowd scenes. *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, 243–252.