Impostors and pseudo-instancing for GPU crowd rendering

Erik Millan* ITESM CEM

Isaac Rudomin† ITESM CEM

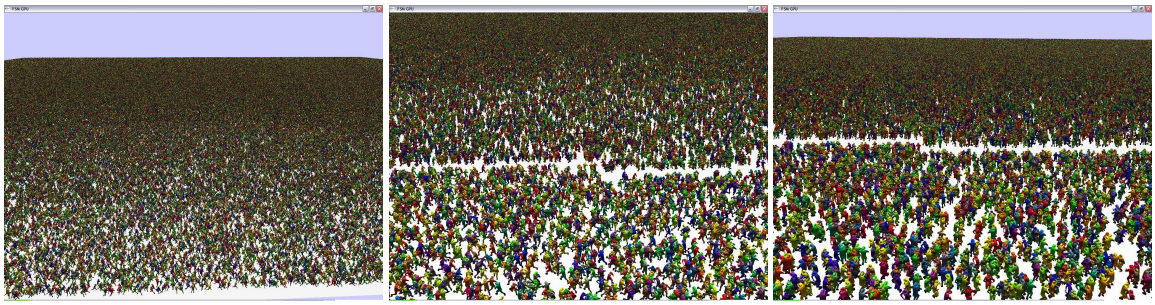


Figure 1: Rendering of a $1,048,576$ character crowd.

Abstract

Animated crowds are effective to increase realism in virtual reality applications. However, rendering crowds requires large computational power. In this paper, we present a technique suitable to render large crowds of characters that takes advantage of existing programmable graphics hardware. Impostors are used for low-detail representation, while pseudo-instancing is used for higher detail. A LOD map is used to select between both representations, based on a customizable threshold.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: crowds, level of detail, impostors, GPU

1 Introduction

Large computer-generated crowds have become a common feature in action films. Transition of these large crowds from movies to interactive applications requires efficient methods to simulate thousands of characters at interactive frame rates. One of the most time-consuming parts in this simulation is crowds rendering.

Many different techniques have been used to increase the number of rendered characters within an application. In order to reduce the processing requirements to render each character, these techniques usually decrease the detail on those characters displayed in small portions of the screen.

However, recent advances in graphics hardware present new opportunities to increase the size of real-time rendered crowds. In this paper we propose the use of a set of techniques in order to render large animated crowds at interactive frame rates. An instancing technique will be used to display detailed characters, while impostors will be used to show low-detail characters.

This paper is organized as follows. In section 2 some articles regarding different level-of-detail techniques are presented. Section 3 describes the techniques in this research to display large animated crowds, as well as some approaches to harness the power of graphics hardware using these techniques. Then, section 4 describes the LOD map, which is an efficient technique to select which rendering technique will be used to display each character. Section 5 presents an efficient implementation of crowd rendering using the graphics processor. Finally, section 6 presents results on the implementation of these techniques, and section 7 provides some conclusions and future work.

2 Related work

There are many possible approaches that can be used to reduce the detail of a geometric mesh, having different advantages and drawbacks for its implementation within the graphics processor. Initial mesh reduction approaches involved sampling a smaller set of vertices over a surface, to then join these points into a new polygonal structure. For instance, Turk [Turk 1992] sampled random points within polygons in a mesh, to then attach them to the original mesh and remove one by one the original vertices, producing models with a lower number of polygons. Progressive Meshes, introduced by Hoppe [Hoppe 1996], use a similar approach. Here, no new vertices are added to the mesh. Instead, one vertex is removed iteratively, and the mesh topology is updated. All these updates are stored into a hierarchical structure which allows to render the original mesh using any number of vertices.

Within mesh reduction approaches, the changing topology of lower-detail meshes requires the storage of a set of indices that will contain the connectivity among this reduced number of vertices. This, however, presents a limitation when we imply to use current graphics hardware, as selecting a set of indices from within a shader program is not a direct operation. To avoid storing connectivity information, point-based rendering approaches can be useful.

Point-based render group a set of algorithms that display complex objects as a dense array of points [Grossman and Dally 1998].

*e-mail: emillan@itesm.mx

†e-mail: rudomin@itesm.mx

These points may contain different attributes of the surface, such as color, normal vector or material. Approaches such as Q-splats [Rusinkiewicz and Levoy 2000] or surfels [Pfister et al. 2000] are point-based techniques that use different rendering methods to efficiently display large clouds of points as realistic solid objects. Point-based techniques have also been adapted to work with animated objects [Wand and Straßer 2002]. Point-based approaches solve the connectivity problem, as point primitives do not require information about the topology of the mesh for rendering. Nevertheless, a different number of points should be displayed for each character, which still presents problems in current graphics hardware, where shader programs lack the capacity of producing new vertices.

Image based approaches have also been used to simplify rendering of large crowds. In these approaches, 3D models are replaced by a small set of textured polygons that resemble the original geometry. Tecchia [Tecchia and Chrysanthou 2000] used an image-based technique to render large crowds in an urban simulation. A discrete set of possible views for a character is prerendered and stored in memory. Then, when rendering the character, the closest view from the set is used to display the character using a single quad. Dobbyn et al. [Dobbyn et al. 2005] present Geopostors, which combine detailed geometries with impostors to produce characters in an urban simulation. Final textures for characters are constructed by blending a set of image maps produced by normal maps, detail maps, and a set of customizable color materials. In this way, they achieve interactive frame rates and visually realistic simulations with large numbers of characters.

The use of image-based techniques provides with many features that may be harnessed by graphics processors to efficiently render large crowds of characters. Each object is represented by a single polygon with a fixed number of vertices, which adapts to the possibilities of current vertex programs. Animation is produced by mapping a texture to this polygon. While it is difficult to use a large number of different textures within a single shader program, individual textures can be obtained by selecting a sub-texture from a large texture. This reduces the texture selection process to a simple texture lookup. The main drawback of image-based techniques is that they require a large amount of memory for storing every frame and view for an animation. Nevertheless, the amount of memory in graphics hardware is increasingly high, making image-based approaches a good option for large crowd rendering.

Graphics hardware has also been used as an additional resource to improve rendering of large crowds in real time applications. De Heras et al. used shader programs to increase the diversity of large crowds in a heritage application [de Heras Ciechomski et al. 2005]. For each character, a polygonal mesh with different resolution is selected according to its distance to the camera. Polygonal models are useful here, as individual deformations can be specified for each character, producing a large variety of poses that would be impossible in image-based approaches. On the other hand, these large number of polygons per character decreases the number of characters that can be interactively rendered.

Instancing [Scott 2004], is a technique that can improve the rendering performance of large sets of similar objects based on features available on graphics hardware. While this technique was originally created for static objects, a similar technique has been adapted to render animated characters [Rudomín et al. 2005]. Nevertheless, though instancing can efficiently render a large number of polygons, all of these polygons must be actually rendered, consuming resources from the graphics hardware. Therefore, the use of image-based approaches, where a single quad is required to display a character, may yield more promising results. Instancing can then be used where detailed objects are required, as a detailed impostor would require large amounts of memory.

Geopostors [Dobbyn et al. 2005] use an image-based approach – impostors– and graphics hardware to blend different image maps for a set of characters, adding per-pixel illumination to dynamically lit each impostor, and producing visually realistic results for large numbers of characters. In addition, this technique uses polygonal models for characters that require more detail. However, the constant updates for impostor textures and character positions increase the traffic towards the graphics hardware and reduce the performance of the application. By using the graphics hardware to handle more parts of the rendering process, better results could be obtained.

3 Impostors and Pseudo-instancing

Aubel et al. [Aubel et al. 1998] presented impostors as an image-based approach to display virtual objects. An impostor displays a character as a textured polygon, commonly a square, which continuously faces toward the camera. The texture coordinates for this polygon will change according to the current viewing position of the camera. In the case of animated models, the texture coordinates will also change according to the current animation frame.

Most approaches based on impostors involve a preprocessing stage. Here, different views for a model are rendered and stored into an image. As it is impossible to render every possible view of a model, a discrete set of views is parameterized, usually by uniformly distributing these views over the surface of a bounding sphere or a bounding hemisphere.

The 3D model will be rendered from these views, and the set of renders will be stored into an image texture. This texture is usually organized as a grid, where columns share views from the same slice and rows share views of the same stack of the bounding sphere. In the case of animated models, a discrete set of frames will also be selected, and for each frame, a set of views will be rendered and stored. Images produced by this preprocessing stage will be later used to obtain the texture map applied for rendering impostors.

Impostors are drawn as textured quads. Using the camera view, the vertices of the quad are transformed so that the quad faces the camera. Then, texture coordinates for this quad will be calculated. The current view for an object is obtained based on the camera view and on the object heading. The closest available view in the impostor texture map will be then selected, and applied to the transformed quad.

As mentioned by Millan and Rudomin [Millán and Rudomín 2006], impostors present great advantages in current graphics hardware for character instancing. Through a shader program, it is possible to extract the viewing frustum from the current view, and by combining this view with the heading for each character, obtain its viewing angle. Then, texture coordinates can be calculated to extract the most similar image resembling the current view and animation pose, to then map this texture to a polygon that will display this character.

Impostors are a useful representation for low-detail crowd rendering. However, when impostors are rendered to an area larger than the impostor resolution, pixels from the impostor texture become noticeable. Therefore, a different representation should be used.

Instancing is not a level-of-detail technique by itself. Instead, instancing is a feature provided in modern graphics hardware that optimizes rendering an object several times. Instancing works by drawing multiple instances of the same model through a single draw call [Scott 2004]. Through instancing, graphics processor deals

with per-instance geometry transformations and appearance modifications, releasing the main processor from this task.

While instancing has been designed originally for static objects, it is possible to adapt this technique for its use in large crowds. In this case, geometry should be updated on every animation frame. Obtaining a real benefit from instancing requires that each model is used for as many instances as possible. Updating the animation of a model implies sending the modified data from the main memory to the graphics memory. As this operation is a well known bottleneck in the rendering pipeline, its use should be reduced as much as possible.

4 LOD Map

Two different approaches were thus selected to display characters: impostors for low-detailed, distant characters, and instancing for detailed, nearby characters. Distance to the camera to each character must then be estimated to select the appropriate detail. In addition, to improve rendering efficiency, those characters outside the viewing frustum should not be considered for rendering. An efficient technique to calculate this distance is through a LOD map.

A *LOD map* approximates distance calculations within an area through a discrete grid. This grid will describe the required detail in the entire surface of the scene. In this way, the detail for a character can be obtained by verifying the pixel corresponding to the position for that character. This can be achieved through a single texture lookup, which constitutes a single operation for a fragment shader program.

Calculation of a LOD map is also an efficient task. Whenever the camera location is updated, the viewing frustum should be intersected with the scene plane, which contains all the possible locations for a character. This intersection will produce a convex polygon, which will bound the area of the scene plane visible by the camera. Intensity of pixels outside this polygon will be set to zero, in order to specify that characters located in those areas will not be drawn.

To calculate detail within the visibility polygon, vertices of this polygon will be assigned a specific weight. This weight will be assigned according to the distance of the point p from the camera, according to this equation:

$$w = \frac{z_{far} - z_p}{z_{far}} \tag{1}$$

where z_{far} is the distance between the far plane and the near plane, and z_p is the distance between the point p and the near plane. Points located in the near plane will obtain a weight of 1, while for points at the far plane will obtain a weight of 0.

The visibility polygon will then be rendered on the visibility plane, using the weights as grayscale values. Linear blending will be used to interpolate colors within this polygon, producing the final LOD map. A sample LOD map is shown in Figure 2.

Finally, the detail for each character will be compared to a certain threshold. When the detail value is greater than this threshold, this character will be rendered using the instancing technique, while when the detail value is less than this threshold, an impostor will be used.

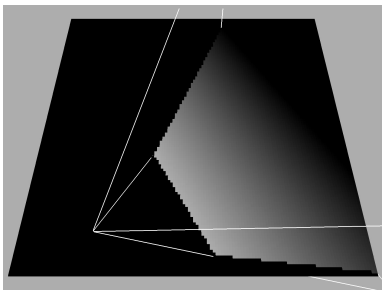


Figure 2: LOD map produced by the intersection of a scene plane with the viewing frustum.

5 GPU implementation

An example implementation was programmed to evaluate the efficiency of the presented techniques aided by the graphics hardware. A set of characters was randomly distributed over a plane. The same model was used for all characters, but in order to increase the appearance diversity, a random color was assigned to each character. Characters would then stand in their position, turning around using a running keyframe animation. The reference application was programmed using Open GL and GLSL.

An initialization step will load both the impostor and the polygonal representations for the animated characters. As impostor generation is produced in a preprocessing phase, a different application was used to produce impostor textures. Two animation frames for a custom impostor texture are shown in Figure 3. Animation frames will be contained within the same image to reduce the number of textures queried by the shader program, and thus simplify its code.

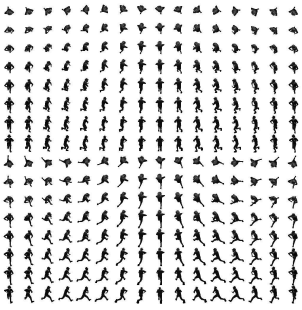


Figure 3: Animation frames from an impostor texture.

The set of animated character instances will be initialized. First, the location and heading for each character will be generated randomly. This information will then be copied to the graphics memory by using a pixel buffer. In this way, rendering and updating of characters may be performed internally by the graphics processor, avoiding sending information back and forth between main memory and graphics memory. Color information for each character will also be generated randomly and sent to the graphics card using a different pixel buffer. Rendering of the animated crowd involves a set of rendering passes.

Pass 1. Character update In this step, the position and heading for each character will be updated. A fragment program will read the character pixel buffer as a texture, and then will update the attributes for characters. In the presented example, characters will be spinning in their place, so the fragment shader will only update

the heading for characters. The new position and heading will be then copied to a new character pixel buffer. Both character pixels buffers can be swapped after the rendering procedure has finished. This pass can also include the update of the animation frame for characters.

Pass 2. LOD map generation Here, the LOD map will be updated according to the camera position. The LOD map will also be stored into a pixel buffer. This is a quick rendering pass, as it only involves the rasterization of a single, non-textured polygon using the fixed rendering pipeline.

Pass 3. Detail Selection This pass will define the detail for each character. A fragment shader will use the character position to query the LOD map texture, mapped from the pixel buffer produced by step 2. The obtained detail value will be compared to a threshold parameter to verify whether the character will be rendered as an impostor or as instanced geometry. Experimentally, thresholds of 0.95 produced a good frame rate, while restricting the size of impostors to its maximum display on-screen area – related to its resolution. The detail value will also be compared with a minimum display threshold, usually 0, to evaluate if that character is visible by the camera.

Conditional statements are expensive in current graphics hardware. Step functions are commonly used to efficiently produce an equivalent functionality. Step functions evaluate as 0 when a value is below a certain threshold, and as 1 otherwise. Hence, both thresholds comparisons required can be handled by the shader program as step functions. The sum of both functions will then be written on the character detail pixel buffer, and used to decide how to render each character by the following steps of the algorithm.

Pass 4. Character texture expansion The character pixel map contains the position and heading for all characters on the scene. In order to display individual objects, these pixels must be copied into a vertex buffer. However, it is not possible to render a character using the impostor technique with a single vertex. While the point sprites technique [ARB 2003] may display a point primitive as a textured quad, it is not yet possible to use programmable graphics hardware to provide the texture coordinates for such quad. Therefore, each vertex must be expanded into a quad for rendering.

An expanded pixel buffer will be filled with the information from the updated character pixel buffer. Unlike passes 1 and 3, where pixel buffers of the same size are produced, this pass requires a wider image buffer, where each row will contain four times more pixels than the original texture. Each pixel from the character pixel buffer will be rendered to four pixels in the expanded pixel buffer. In this way, a quad can be constructed in a later pass by using the four contiguous vertices that share the same character information.

This rendering pass includes some additional processing that will eliminate a bottleneck for the next rendering pass. Texture lookups are very expensive within vertex programs [Scott 2004]. In addition to the location and heading for each character, the resulting pixel buffer will also indicate whether the character will be rendered as an impostor. This will be stored as the alpha component of the color in the pixel buffer, and will be set to 1 or -1, depending on whether the character will or will not be displayed as an impostor.

After finishing this rendering pass, the expanded pixel buffer will be copied into a vertex buffer, which will be used to render impostors. The alpha components produced in this pass will become the w homogeneous coordinate for the pixel. When this coordinate is set to -1, this vertex is discarded by the graphics pipeline. This will discard those characters that are not shown as impostors from the next step.

Pass 5. Impostor rendering The expanded vertex buffer is now used by a vertex shader to display impostors in the frame buffer. In addition to character data, this pass receives the color vertex buffer that will contain the color for individual characters, as well as a corners vertex buffer.

The corners vertex buffer is created at the initialization phase, and will specify which one of the four vertices of the quad corresponds to which corner. The two components for this buffer are set to -1 or 1 depending on whether the vertex is the right or left – or upper or lower – corner of the quad. Additionally, the camera position and orientation, and the attributes of the impostor texture, are received as parameters.

First, the position V of vertices for the impostor quad are obtained. The right vector Q_x for this quad is calculated as the cross product of the camera up vector and the vector from the impostor location P to the camera. The up vector Q_y for the quad will be the camera up vector. Then, the coordinates for each vertex will be obtained as follows.

$$V = P + Q_x\delta_x + Q_y\delta_y \tag{2}$$

where δ_x and δ_y are the values obtained from the corners vertex buffer for the current vertex.

The texture coordinates for the current vertex will be obtained using the character heading and the direction from the character to the camera. From these parameters, the stack and slice for the current view can be obtained as the closest discrete value available. These variables, as well as the current animation frame, are combined to select from the impostor texture grid the closest subimage for the character. This subimage will be mapped to the impostor quad, and alpha testing will be used to discard pixels from the background.

Finally, the fragment shader modifies the mapped texture to produce different-colored characters. A character color vertex buffer, similar to the corners vertex buffer, is generated in the initialization phase, and will produce a different color for each character. It is important to store this information so that each character maintains the same color during the application and can be identified within the crowd. To produce the final render, the fragment shader multiplies each visible pixel by the character color.

Pass 6. Instanced geometry rendering Characters at full detail will be drawn using instancing. However, instancing is a feature only present in DirectX [Scott 2004]. As the example application is programmed in OpenGL, a different technique should be used. A technique based in pseudo-instancing will then be used. Pseudo-instancing [Zelsnack 2004] takes advantage on the efficiency of using persistent vertex attributes, such as color or transformations, to provide information for an entire instance. The main difference with instancing is that, in instancing, only one call is used to render all primitives, while pseudo-instancing requires one call to a display list to render each instance. However, these calls are very efficient in OpenGL, so similar performances are achieved by both techniques [Zelsnack 2004].

This pass requires first the generation of a set of display lists, which will later be used to display different instances. In the current example application, the same animation frame will be used for every character for simplicity. Hence, a single display list will be produced. This can be modified to support different animation frames

for each character, which should have no impact on impostor rendering, as it would simply modify the calculation of texture coordinates. However, to maintain the efficiency of the algorithm, a reduced number of frames should be sent to the graphics card, as vertices should be interpolated and sent to the graphics card on every frame, which will decrease rendering performance.

The next step consists in retrieving character information and detail from the graphics memory to the main memory. This is one of the most time consuming parts of the rendering procedure. Optimizations in this step will probably improve largely the performance of this approach. However, this was the most efficient way to display character instances available in current graphics hardware.

Retrieved character detail will then be used to verify which characters will be rendered as geometry instances. Characters selected for pseudo-instancing will then be rendered by calling the previously produced display lists. Character color, position and heading will be passed as primary and secondary color attributes. Position and heading will be used by a vertex program to transform the model geometry, while color will be used by a fragment program to modify the resulting color, using a similar algorithm to the one used for impostors.

A summary of the rendering procedure is shown in Figure 4.

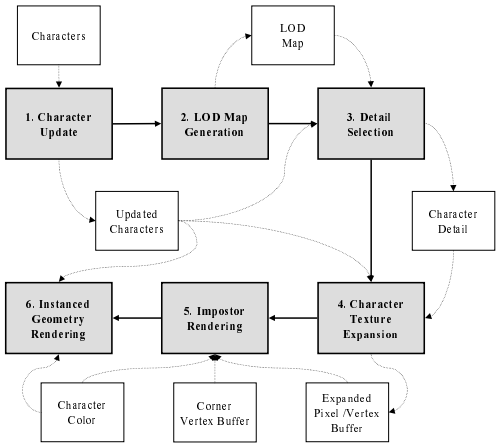


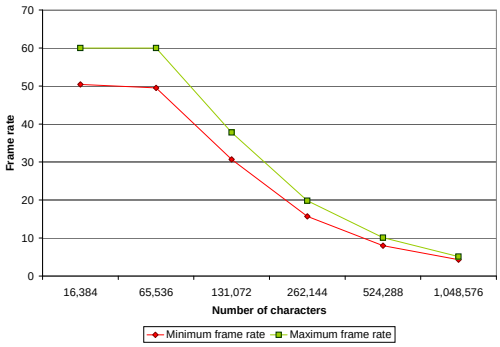
Figure 4: Overview of the crowd rendering procedure.

6 Results

The proposed technique was evaluated on a Pentium Xeon computer at 3.2Ghz using a QuadroFX 4400 graphics card with 512 MB of memory, and rendered to a 1280 x 1024 window. A different number of characters was used to evaluate the performance of the proposed technique. Rendering performance for different number of characters is shown in Figure 5.

Maximum frame rates were achieved when rendering all characters as impostors, while minimum frame rates involved a mixed rendering of impostors and instanced geometry. In order to select the detail for each character, a 256×256 LOD map was used.

Produced crowds had a maximum number of 2^{20} characters due to the maximum texture size of $2,048 \times 2,048$ pixels supported by our graphics hardware. As the expanded character texture requires four pixels per character, this allows a maximum number of $512 \times 2,048$ $(1,048,576)$ characters per texture.



Number of characters	Minimum frame rate	Maximum frame rate
2^{14} - 16,384	50.4 fps	60.0 fps
2^{16} - 65,536	49.5 fps	60.0 fps
2^{17} - 131,072	30.7 fps	37.8 fps
2^{18} - 262,144	15.7 fps	19.8 fps
2^{19} - 524,288	8.0 fps	10.1 fps
2^{20} - 1,048,576	4.3 fps	5.1 fps

Figure 5: Rendering performance for different number of characters.

Memory requirements of 128 bytes per character are constant for character rendering. Half-float textures were used to reduce memory usage to two bytes per channel, and eight bytes per RGBA pixel. Each buffer from Figure 4 has different memory requirements, shown in Figure 6. In general, 128 bytes are required for each character. LOD map generation does not depend on the number of characters, but good results can be achieved with 128×128 maps, adding 16K of memory to the algorithm.

128 MB of memory are required to produce a million characters. While is a large amount of memory, recent graphics hardware include up to 512 MB of memory, leaving a good amount for the scene itself. However, rendering a million characters is a rather complex task that requires many computational resources, and is hard to achieve at good frame rates using a single computer.

Memory requirements for the impostor textures vary according to the number of animations, frames, and viewpoints. A walking animation of 6 frames using 16×8 viewpoints with 64×64 pixels impostors requires 192 KB of texture memory. Changing the number of frames to 18 and the number of viewpoints to 16×16 increases the required memory to 288 KB, which is still a small amount compared to memory used by the character textures.

Number	1	16 K	64 K	256 K	1 M
Characters	8 B	128 KB	512 KB	2 MB	8 MB
Updated chars.	8 B	128 KB	512 KB	2 MB	8 MB
Character detail	8 B	128 KB	512 KB	2 MB	8 MB
Exp. Pixel	32 B	512 KB	2 MB	8 MB	32 MB
Exp. Vertex	32 B	512 KB	2 MB	8 MB	32 MB
corners	16 B	256 KB	1 MB	4 MB	16 MB
Character color	24 B	368 KB	1.5 MB	6 MB	24 MB
Total memory	128 B	2 MB	8 MB	32 MB	128 MB

Figure 6: Memory requirements for different number of characters.

Some of the renders for the example application are illustrated in Figure 7. Here, a gap is left between impostors and instanced geometry to help identifying both approaches on the image. The decision to use different colors for each character was motivated by these results. The use of a single colored model for every character would make it even more difficult to distinguish individual characters within the crowd.

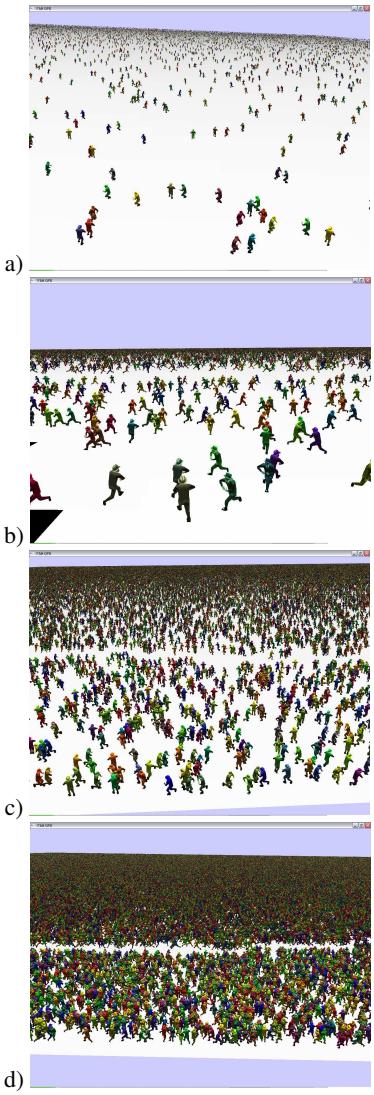


Figure 7: Display of different size crowds. a) 16 K characters. b) 64 K characters. c) 256 K characters. d) 1 M characters.

While the render of 1,048,576 characters provided a relatively low frame rate of between 4.3 and 5.0 frames per second, the produced crowd is very large for the window resolution used. A crowd of 262,144 characters produces a better frame rate and still gives a similar appearance. Character size may also be enlarged to give the appearance of a large crowd with a smaller number of characters. Here, a smaller character size was used to better appreciate the crowd density.

Having a set of characters spinning in their own place is not a very common behavior in videogames or in any graphics application. Hence, the proposed rendering technique was integrated with another application, where character behavior is simulated within the graphics processor [Rudomín et al. 2005]. In order to do this, step 1 from the rendering algorithm was replaced by the programmable Finite State Machine simulation proposed in our previous work. In the presented example, characters are programmed to go from one region to other and back.

Using only pseudo-instancing, the simulation was only capable of displaying 16,384 characters at only 5.45 frames per second. By integrating the proposed rendering technique, the resulting performance was very similar to that of the spinning characters: between 59 and 60 frames per second for 16,384 characters, between 29.9 and 60 fps for 65,536 characters, and between 3.1 and 4.6 fps for 1,048,576 characters. The main reason of the frame rate reduction is that in certain moments, the number of characters rendered at full detail is higher than in the spinning example. Two screenshots of this simulation are shown in Figure 8.

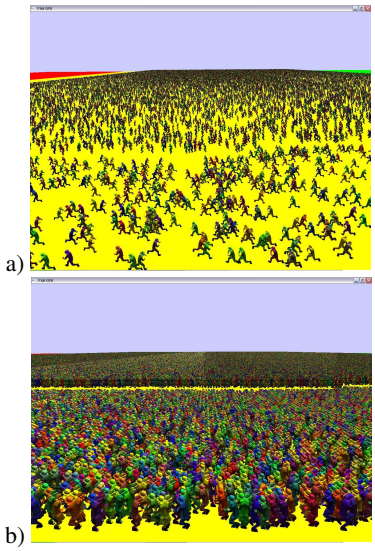


Figure 8: Display of different size crowds using FSM simulation on the GPU. a) 512 K characters. b) 1 M characters.

7 Conclusions and Future Work

An efficient technique has been presented to display large crowds of animated characters at interactive frame rates. As graphic processors become more powerful and common, the use of this approach may enable the interactive display of even larger crowds. Impostors are a well suited technique for graphics processor, as a constant single quad is only required to display each character, and its animation and transformations are based on simple texture lookups.

Different thresholds between impostors and pseudo-instancing can be used in different applications. A lower threshold can be used in realistic virtual reality simulations, where powerful hardware enables the production of higher resolution images. Here, as characters will be rendered to a larger on-screen area, the use of more instanced geometry will produce more realistic results.

In contrast, console videogames and applications for commodity hardware may rely on a higher threshold, improving the rendering performance for such applications. Furthermore, these applications may even eliminate the pseudo-instancing stage, avoiding the transfer of character data to main memory, and thus producing an even better performance, though with less detail.

While this distance threshold variability is useful to customize the detail of the characters according to the capabilities of the rendering hardware, a better metric may be used to switch between both representations. This metric could be obtained from the impostor generation phase; thresholds for image maps could be obtained as a function of the distance between the characters and the camera in the impostor generation phase, and of the image resolution of the produced impostor.

The number of displayed characters exceeds at least by an order of magnitude existing algorithms for rendering. Reynolds was able to produce crowds of 10,000 characters at 60 frames per second on Playstation 3 hardware [Reynolds 2006]. Other approaches have reached between 1,000 and 30,000 characters at interactive frame rates [de Heras Ciechomski et al. 2005; Dobbyn et al. 2005], adding features not present in our initial rendering approach. In particular, Dobbyn [Dobbyn et al. 2005] uses a set of texture maps to improve rendering: normal maps for per-pixel lighting, region maps to assign different colors to different parts of the character, and detail maps. This rendering technique produces a more realistic appearance.

Our rendering system uses a simplified version of behavior simulation and does not calculate illumination or shadows, but is capable of displaying more than 60,000 characters at 60 fps, and many more at interactive frame rates. While this is an initial approach, the efficiency obtained by these results encourage the extension of this algorithm to include further improvements in rendering.

Character behavior is extremely simple in the presented application. Better algorithms for character simulation, executed either by the main processor or by the graphics hardware, should be evaluated to produce more interesting character behaviors for videogames or other applications. Using finite state machines through image maps [Rudomín et al. 2005] is a starting point, but better ways to specify complex behaviors should be developed.

Finally, next generation graphics hardware should be thoroughly evaluated, as upcoming features, such as geometry shaders, animated instancing, or vertex shaders texture lookups, may provide with additional functionality that may be harnessed to improve the flexibility and efficiency of GPU crowd rendering.

References

ARB. 2003. ARB_point_sprite extension. Tech. rep., OpenGL Architecture Review Board.

AUBEL, A., BOULIC, R., AND THALMANN, D. 1998. Animated impostors for real-time display of numerous virtual humans. In *VW '98: Proceedings of the First International Conference on Virtual Worlds*, Springer-Verlag, London, UK, 14–28.

DE HERAS CIECHOMSKI, P., SCHERTENLEIB, S., MAM, J., MAUPU, D., AND THALMANN, D. 2005. Real-time shader rendering for crowds in virtual heritage. In *The 6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, Eurographics Association, Pisa, Italy, 91–98.

DOBBYN, S., HAMILL, J., O'CONOR, K., AND O'SULLIVAN, C. 2005. Geopostors: A real-time geometry / impostor crowd rendering system. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, 95–102.

GROSSMAN, J. P., AND DALLY, W. J. 1998. Point sample rendering. In *Rendering Techniques '98*, Springer, 181–192.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 99–108.

MILLÁN, E., AND RUDOMÍN, I. 2006. A comparison between impostors and point-based models for interactive rendering of animated models. In *Proceedings of the International Conference on Computer Animation and Social Agents (CASA) 2006*, University Press.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 335–342.

REYNOLDS, C. 2006. Crowd simulation on PS3. In *Game Developers Conference 2006*.

RUDOMÍN, I., MILLÁN, E., AND HERNÁNDEZ, B. 2005. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory 13*, 8 (November), 741–751.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 343–352.

SCOTT, P. 2004. Shader model 3.0, best practices. Tech. rep., NVIDIA Corporation. available online at http://developer.nvidia.com/object/SM3_0_best_practices.html.

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

TURK, G. 1992. Retiling polygonal surfaces. *Computer Graphics 26*, 2, 55–64.

WAND, M., AND STRASSER, W. 2002. Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum 21*, 3. Eurographics 2002.

ZELSNACK, J. 2004. GLSL pseudo-instancing. Tech. rep., NVIDIA Corporation. available online at http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html.