Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Simulation Modelling Practice and Theory 13 (2005) 741–751

SIMULATION MODELLING PRACTICE AND THEORY

www.elsevier.com/locate/simpat

Fragment shaders for agent animation using finite state machines

Isaac Rudomín *, Erik Millán, Benjamín Hernández

Instituto Tecnológico y de Estudios Superiores Monterrey, Campus Estado de México, Monterrey, Mexico

Abstract

In a previous paper we generated animated agents and their behavior using a combination of XML and images. The behavior of agents was specified as a finite state machine (FSM) in XML. We used images to determine properties of the world that agents react to. While this approach is very flexible, it can be made much faster by using the power available in modern GPUs. In this paper we implement FSMs as fragment shaders using three kinds of images: world space images, agent space images and FSM table images. We show a simple example and compare performance of CPU and GPU implementations. Then we examine a more complex example involving more maps and two types of agents (predator–prey). Furthermore we explore how to render agents in 3D more efficiently by using a variation on pseudoinstancing.
© 2005 Elsevier B.V. All rights reserved.

Keywords: FSM-based simulations; Virtual characters; GPU; Shader; Maps; Crowds; Behavior simulation; Finite state machine

* Corresponding author.
E-mail addresses: rudomin@itesm.mx (I. Rudomín), emillan@itesm.mx (E. Millán), hbenjamin@itesm.mx (B. Hernández).
URL: http://rudomin.cem.itesm.mx/~rudomin (I. Rudomín).

1569-190X/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2005.08.008

1. Introduction

In virtual environments and video games, it is common to find different animated characters that interact with the user, with other virtual characters, and with their surrounding environment. The behavior of these characters has been defined in many different ways: One early approach is emergent behavior (boids, by Reynolds [1]). A very common and simple way to specify behavior for virtual characters is by using finite state machines (FSMs). For instance, as done by Cremer et al. [2], behavior is controlled through communicating, hierarchical and concurrent state machines. Work by Musse and Thalmann [3], has a hierarchical FSM architecture that models the behavior of crowds of characters. Devillers and Donikian [4], developed a scenario language, similar to structured programming languages, that uses FSMs to specify most of the instructions. Becheiraz and Thalmann [5] included more complex behaviors involving emotions in the FSMs.

There are some interesting ideas on the use of textures in different state-based simulations. James [6] developed a Game of Life simulation on the GPU, where, according to the cell state and to its neighbors, the output for the rule was selected from a lookup table. Similarly, Miyazaki et al. [7] use a set of rules similar to cellular automata, where each cell contains floating values, to simulate cloud formation. Wei et al. [8] simulate the motion of particles in fluids applying a local set of rules over a grid of cells.

In previous work [9], we created a system that specified behavior by using XML scripting and images. While our approach proved useful for defining simple behaviors for semi-autonomous characters, we decided that, since it used images extensively, it could be useful, and deliver better performance, if we would implement agent behavior based on finite state machines as fragment shaders.

Not much has happened in A.I. simulation in the GPU. There is some very interesting preliminary work for a class in UNC [10] that seems to take an approach similar to ours, based on Swarm Intelligence [11], but programmed in the GPU by using a modification of coupled map lattices [12] implemented as shaders. It is difficult to judge how far this effort has progressed. De Chiara et al. [13] simulated the behavior of flocks of birds, including obstacle avoidance, using the graphics hardware.

There is also some work in path finding using graphics hardware [14,15]. Here, the basic idea is to discretize the world and propagate proximity to initial or destination cells; when this propagation arrives to the destination cell, it can be said that a path has been found.

2. FSM shaders

Modern GPUs allow using several textures and dependent texture lookups. Based on this idea, we implement a very simple system involving three kinds of maps that will be queried and used by agents:

(1) world space maps,
(2) agent space maps,
(3) FSM maps.

World space maps are what we normally consider maps. They codify some value for each location (pixel) in the map. This would cover many types of maps: texture maps, height maps, normal maps, collision maps, interest area maps, action maps. In general, all such maps can be considered labeled maps with some color on each pixel. The information represented by the colors in the maps, as well as the number of such maps that one could use in a particular simulation is up to the application designer.

Agent maps, on the other hand, have a pixel for each agent. They can be used to codify the state of the agent, the type of agent, the x, y position of the agent in the world map, the agent's velocity in said world map, or any other information of the agent.

A finite state machine can be represented as a table where given a certain state of the agent and a certain input, one can obtain a new state (and position) of the agent. This table can also be considered a map. Maps can be built for deterministic state machines. If there are several types of agents, there could be a portion of the table devoted to each type of agent. In a similar way one could describe probabilistic FSMs, hierarchical FSMs or layered FSMs. For the moment we consider a simple system

(1) A single labeled color texture represents the world where the agents will move. Suppose the image is size $X \times Y$ and there are L possible color labels.
(2) Another texture represents our agents (with N agents represented as a RGB $N \times 1$ pixel image (R is state, G and B the xy position of the agent in the world). Each agent can have up to S different states.
(3) Finally, there is a FSM image of size $S \times L$, indexed by state and input, as one can see in Fig. 1a.

All we must do then, is to consult the FSM map to update the agent map, and then render the world map with the agents overlayed. There is another step, however, since the FSM map is indexed by state, which can be obtained from the agent map, but also by input (or label color), which must be obtained from the agent's position in the world. Therefore, the basic mechanism, expressed as pseudocode, is extremely simple:

```
given agent i
  s = agent[i].s; x = agent[i].x; y = agent[i].y;
  l = world[x, y];
  agent[i].s = fsm[s, l];
  draw agent[i];
```

One would update the positions of the agent in a similar fashion. As you can see all we are doing are texture lookups. The basic mechanism can be seen in Fig. 1b.

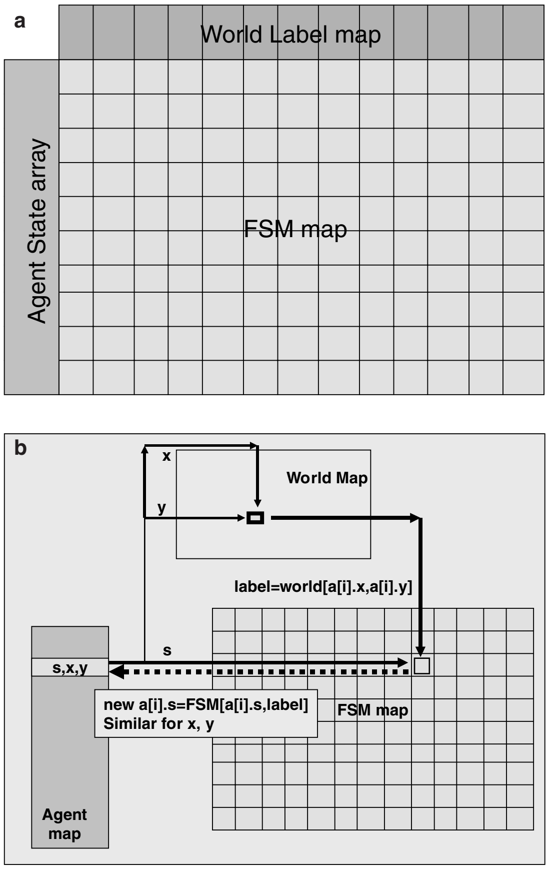


Fig. 1. (a) FSM map indexed by state and input, (b) basic mechanism involving agent, world and FSM maps.

3. Results

The system has been implemented as fragment shaders using GLSL and runs on a PC with a 6800 Nvidia card. We will discuss two examples. First, we present a simple example that we will compare with a software implementation. Then we will discuss a more complex predator–prey example.

3.1. Simple example

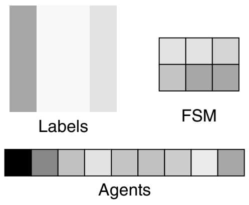
The first example is extremely simple. Agents are generated at random positions. A very simple labeled world map is constructed with three vertical strips: an agent will consult the color of the labeled world map and follow a very simple FSM that causes it to move right until it reaches the strip in the right, at which point the agent changes state and starts moving left until it gets to the strip in the left. The FSM map in this example was hand crafted from the table associated with the diagram in Fig. 2b. We have chosen the appropriate color for the table by using different R values for the states and different GB values for representing δ*x*, δ*y*. The resulting FSM map as well as the label and agent maps used for this example can be seen in Fig. 2a.

Building the FSM map by hand seems simple enough for diagrams such as the one shown here, but one can see that it is probably not the best way to construct the maps for more complex FSMs. This is a problem that we will try to solve in the future.

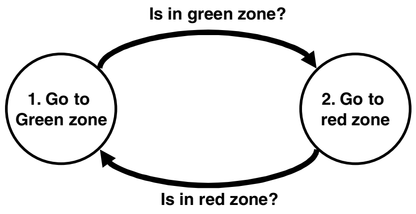
To implement the basic mechanism in the case of this simple example is the following fragment shader in GLSL:

```
uniform sampler2DRect fsm;
uniform sampler2DRect agentsPos;
uniform sampler2DRect soilLabels;
void main ()
{vec3 agentLookUp = texture2DRect (agentsPos, gl_TexCoord
[0].st).rgb;
vec2 agentPos = agentLookUp.rg;
```

a



b



f(L,S):(S,Δx, Δy)	L=Red	L=Yellow	L=Green
S=State 1	1,1,0 *RGB[0, 255, 0]*	1,1,0 *RGB[0, 255, 0]*	2,0,0 *RGB[255, 128, 0]*
S=State 2	1,0,0 *RGB[0, 128, 0]*	2,-1,0 *RGB[255, 0, 0]*	2,-1,0 *RGB[255, 0, 0]*

Fig. 2. (a) Example label map, FSM map and agent map, (b) FSM to simulate characters and construction of the FSM map. Each cell is equivalent to a pixel in the map. Red component encodes the state *S*, green component the *x* increment Δ*x*, and blue component the *y* increment Δ*y*. RGB values for these pixels in the final map are also included.

```
float agentState = agentLookUp.b;
  float soilLabelsLookUp = texture2DRect (soilLabels,
agentPos).r;
  vec3 fsmOut = texture2DRect
(fsm, vec2(soilLabelsLookUp.r, agentState)).rgb;
  gl_FragColor = vec4 (agentPos.x +
(fsmOut.r − 1), agentPos.y + fsmOut.g, fsmOut.b, 1);
}
```

Frames from a sample run of this example (for easier visualization a small number of agents was used) can be seen in Fig. 3. To do this, once we have applied the FSM using the fragment program shown above, we have to render these results so users could see the agents behavior. For testing purposes we have decided to render these agents as a points.

As we have mentioned, the FSM results are stored in a color texture or more precisely in a Pixel Buffer Object (PBO) for the GPU implementation. We take advantage of the fact that the PBO is resident in video memory so we can use the technique called "render to vertex array", were the color information of the PBO is directly converted to vertex position information using a shader program and Vertex Buffer Objects (VBO).

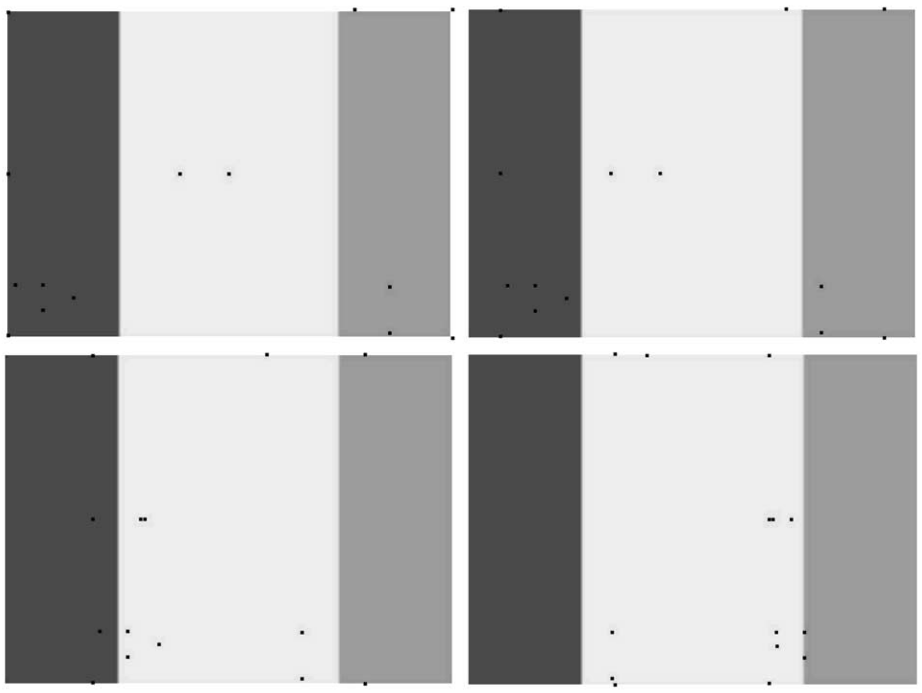


Fig. 3. Frames from simple example.

We programmed a similar algorithm running on the CPU to compare our results. For the CPU case, since the FSM results are stored in main memory, i.e. an array, we cannot use the same technique used in the GPU case. Nevertheless we still are using Vertex Buffer Objects. These VBOs are loaded with the array information so render performance is just affected by the intrinsic transfer of information from main memory to video memory. This makes the rendering in both implementations basically the same, and so simulation results we will see shortly, comparable.

If we run the simulation, it proceeds at interactive rates even with a large number of agents. We compare with a software FSM implementation. Frames per second resulting with different numbers of agents can be seen in Fig. 4 (the same rendering algorithm is used in both cases).

It is clear from these numbers that even for larger numbers of agents (a million), using the GPU can give us a significant advantage. The GPU implementation stays within the interactive range above 30 fps while the same algorithm running entirely on the CPU is now down to 3 fps.

3.2. Predator–prey example

GPU agent animation is not restricted only to the generation of the simple character behaviors from our first example. More complex behaviors can be achieved through the use of static and dynamic maps that can modify the outcome of the FSM. One example to illustrate this point is a predator–prey simulation.

The general idea of the simulation of a predator is described in Fig. 5. Fig. 5a, contains a general description of the FSM: the predator is wandering within its environment until it locates a nearby prey. When this happens, it starts following its prey until it escapes, and then goes on wandering.

This FSM is encoded in the map shown in Fig. 5b. Here, the red component is used, as in the first example, to encode the current state of the automata. However, rather than containing the actual motion for the agent, the FSM contains a reference for this motion in its green component. According to the current state, the agent will decide either to stay in its own location or to check the most appropriate direction of motion.

agents	CPU	GPU
16384	428	713
65536	85	356
262144	11.8	137
1048576	3	33.9

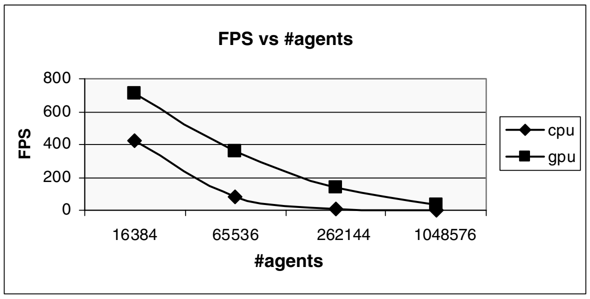
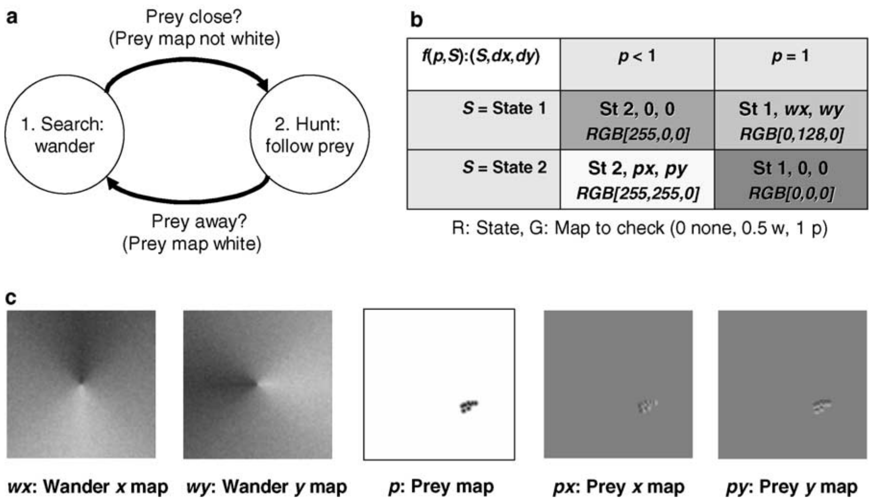


Fig. 4. CPU vs GPU fps.



f(*p*,*S*):(*S*,*dx*,*dy*)	*p* < 1	*p* = 1
S = State 1	St 2, 0, 0 *RGB[255,0,0]*	St 1, *wx*, *wy* *RGB[0,128,0]*
S = State 2	St 2, *px*, *py* *RGB[255,255,0]*	St 1, 0, 0 *RGB[0,0,0]*

R: State, G: Map to check (0 none, 0.5 w, 1 p)

Fig. 5. (a) FSM for the predator in the predator–prey example, (b) FSM map for the predator in the predator–prey example. Red component encodes the state S, green component encodes a reference to the map where the x and y increment (dx and dy) will be searched. Blue component is ignored. (c) Prey map p used to detect proximity of preys, and maps used for agent position increment when wandering (wx and wy) and hunting preys (px and py).

In the wandering state, there is no clear hint on where to locate a prey. In this case, the agent uses a couple of static maps, shown in Fig. 5c as *wander* maps. The agent selects a pixel from each map according to its current location, in a similar way to the lookup presented in [6]. The values from these pixels are added to the current location of the agent.

The maps used for the *hunt* state are used in a similar way to those used in the wandering state, but they are calculated dynamically; after each simulation step, these maps should be recalculated to produce dynamic behavior.

Initially, the location of all prey is marked in a collision map similar to the one used in [9], where each agent corresponds to one pixel according to its location in the map. A Gaussian blur filter is applied to this collision map in order to obtain the prey map. In this way, a predator can locate only prey within its vision range. When a predator is located on a white pixel, no prey can be seen; but in a darker pixel, prey are seen and predators may follow them.

Another advantage of using the Gaussian blur filter to obtain the visibility regions is that it is simple to infer the direction in which a predator would follow this prey. By applying horizontal and vertical sobel filters to the prey map we obtain the direction to follow prey, shown in Prey x and Prey y maps in Fig. 5c. These maps are interpreted just as wander maps are, simplifying the behavior implemented in the shader.

Back in the shader, transitions are selected according to the value of the prey map. If a predator is over a white area of the map, no prey is near, so the wander maps are used to obtain agent displacement. When prey are close, prey maps are used to select agent displacement. This modifies slightly the script from the simple example by querying a different map for the displacement instead of obtaining it from the FSM map, but the basic algorithm remains very similar and simple.

The results are similar to the ones in the simple example, and some animation frames are shown in Fig. 6. Here, it is possible how to appreciate how the predators, the white agents, start following the wander map, and then start following prey, the dark agents, whenever they get close to them.

3.3. Optimizing 3D render

Up to now we have rendered each agent as a point. The idea of the simulation, however, is to be able to render many 3D agents interactively for games or similar applications. If we render each 3D agent in the scene in the usual fashion, the frame rate decreases significantly, but remains interactive for a reduced, yet still significant, number of agents. Techniques such as instancing allow us to increase this number significantly.

Our approach to instancing consists of

(1) generating a vertex buffer with the agent positions,
(2) transforming vertices for each character with a shader.

To do this, we use the character position in the agent map (the same one we use for the simulation) to display a set of more complex virtual animated characters, instead of points as we did above. A very efficient option would be reading the agent position directly from the texture where this information is stored, avoiding unnecessary traffic from the GPU to the CPU, then using this data to render agents. Unfortunately, we did not find this option as efficient as expected, since texture lookups within vertex

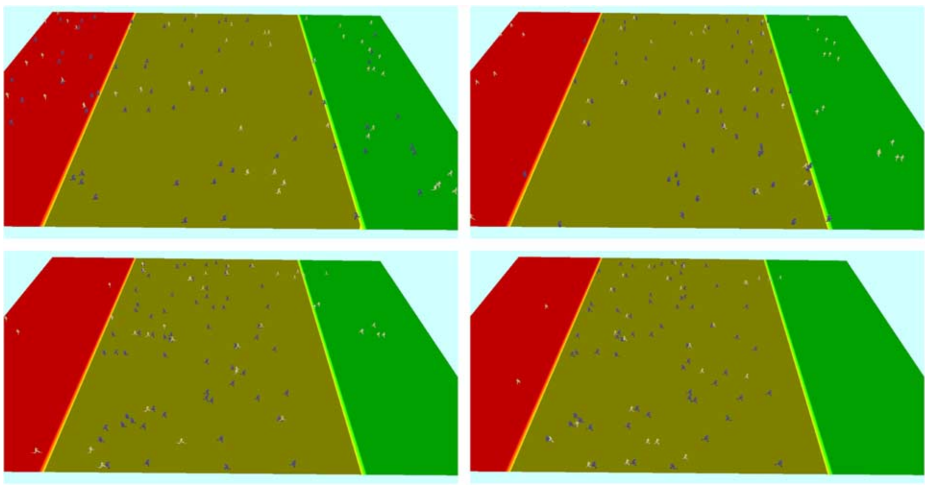


Fig. 6. Frames from predator–prey example.

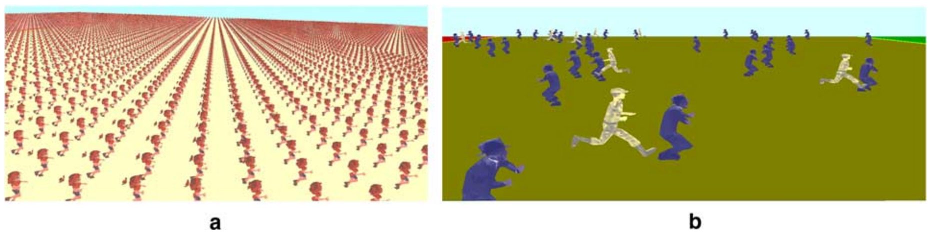


Fig. 7. Frames from examples, rendering 3D characters: (a) example with 16,384 instances running at 5.45 fps for the simple agent simulation, (b) pseudoinstancing for the predator–prey simulation.

shaders are not particularly efficient in current graphics hardware, at least in GLSL. However, next generation graphic cards will privilege this option.

Instead, we used the following algorithm, which is similar to the pseudoinstancing technique used in [16], but extended to display animated characters. First, we update the animation frame for a character. While we used the same animated character for all agents, different animation frames could be used to increase diversity in character appearance. We used keyframed animations for simplicity. The current frame of this animated character is sent to the graphics hardware as a set of vertex arrays and compiled into a display list.

Then, for each agent instance, current and previous agent positions are sent as color-value parameters for a vertex shader. Using this vertex shader, the display list for the animated model is rendered. The vertex shader uses the agent position, encoded as a color-value, to translate each vertex to the position of the character. This is done by adding the position to each vertex coordinates.

The shader uses the difference between the current and the old agent position, sent as color parameters, to obtain the character direction. This difference, when normalized, represents the sine and cosine of the rotation angle. Once known, rotation of vertices is done using these values so that the agent looks towards its motion direction.

Our pseudoinstancing technique was applied to the first simulation discussed previously. This implementation, initialized with 16,384 agents (512×512), runs at 5.45 frames per second without any further optimization. It is important to mention that animated models are used for this render. A frame from this simulation with the above mentioned pseudoinstancing, with 16,384 agents, can be seen in Fig. 7a. This pseudoinstancing technique was also used in the predator–prey simulation, showing similar results. A frame from this implementation can be seen in Fig. 7b.

4. Conclusions

We have presented a simple approach to generating agent behavior implementing Finite State Machines on the GPU. The basic idea is to use dependent texture lookups of three kinds of maps: agent space, world space and FSM maps within a fragment shader. This works at interactive rates for even a large number of agents in current generation GPUs. We have tried a simple and a more complex example. In the future we will attempt even more complex situations using several maps and more FSMs (probabilistic, layered, hierarchical). We are also working on a simpler, better way for users to construct FSM maps, such as interpreting an XML scripting language similar to [9].

We have also implemented pseudoinstancing. While interesting, the performance is not yet good enough or general enough for what is needed. Further optimizations (culling, for example) will be attempted. While simple culling is viable, we are working on a version that takes advantage on the map and shader paradigm.

References

[1] C.W. Reynolds, Flocks, herds, and schools: a distributed behavioral model, Computer Graphics 21 (4) (1987) 25–34.

[2] J. Cremer, J. Kearney, Y.E. Papelis, HCSM: a framework for behavior and scenario in virtual environments, Modeling and Computer Simulation 5 (3) (1995) 242–267.

[3] S.R. Musse, D. Thalmann, Hierarchical model for real time simulation of virtual human crowds, IEEE Transactions on Visualization and Computer Graphics 7 (2) (2001) 152–164.

[4] F. Devillers, S. Donikian, A scenario language to orchestrate virtual world evolution, in: SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, 2003, pp. 265–275.

[5] P. Becheiraz, D. Thalmann, A behavioral animation system for autonomous actors personified by emotions, in: Proceedings of the First Workshop on Embodied Conversational Characters (WECC'98), 1998, pp. 57–65.

[6] G. James, Operations for hardware-accelerated procedural texture animation, in: Game Programming Gems 2, Charles River Media, 2001, pp. 497–509.

[7] R. Miyazaki, S. Yoshida, T. Nishita, Y. Dobashi. A method for modeling clouds based on atmospheric fluid dynamics, in: PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications, IEEE Computer Society, 2001, pp. 363–372.

[8] X. Wei, W. Li, K. Mueller, A.E. Kaufman, The lattice-Boltzmann method for simulating gaseous phenomena, IEEE Transactions on Visualization and Computer Graphics 10 (2) (2004) 164–176.

[9] I. Rudomín, E. Millán, Xml scripting and images for specifying behavior of virtual characters and crowds, in: Proceedings CASA 2004, University Press, 2004, pp. 121–128.

[10] C. Hantak, Comparison of parallel hardware based and graphics hardware based platforms for swarm intelligence simulations, Class report, Available from: <http://www.cs.unc.edu/hantak/ip.html>.

[11] G. Theraulaz, J. Deneubourg, On formal constraints in swarm dynamics, in: Proceedings of the 1992 IEEE International Symposium on Intelligent Control, IEEE Computer Society Press, 1992, pp. 225–233.

[12] M. Harris, G. Coombe, T. Scheuermann, A. Lastra, Physically-based visual simulation on graphics hardware, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, September 01–02, 2002, Saarbucken, Germany.

[13] R. De Chiara, U. Erra, V. Scarano, M. Tatafiore, Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance, in: Proceedings of the Vision, Modeling, and Visualization Conference 2004 (VMV 2004), 2004, pp. 233–240.

[14] S .Henriksen, GPU Path Finder demo. in: ShaderTech GPU Programming. Available from: <http://www.shadertech.com/shaders>, September 2004.

[15] K. Phong Ly, Pathfinding on the GPU, in: ShaderTech GPU Programming. Available from: <http://www.shadertech.com/shaders>, September 2004.

[16] J. Zelsnack, GLSL Pseudo-Instancing, Technical Report, NVIDIA Corporation, 2004.