IMPLEMENTING AUTONOMOUS CROWDS IN A COMPUTER GENERATED FEATURE FILM

A Thesis

by

JOHN ANDRE PATTERSON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Visualization Sciences

IMPLEMENTING AUTONOMOUS CROWDS IN A COMPUTER GENERATED FEATURE FILM

A Thesis

by

JOHN ANDRE PATTERSON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Visualization Sciences

ABSTRACT

Implementing Autonomous Crowds
in a Computer Generated Feature Film. (December 2005)

John Andre Patterson, B.A., Boston University;
M.S., The University of Texas at Austin
Chair of Advisory Committee: Dr. Ergun Akleman

The implementation of autonomous, flocking crowds of background characters in the feature film “Robots” is discussed. The techniques for obstacle avoidance and goal seeking are described. An overview of the implementation of the system as part of the production pipeline for the film is also provided.

To Jen, Grover and Sergio

ACKNOWLEDGMENTS

I would like to acknowledge the contributions of Mitch Kopelman, Rob Cavaleri, Alen Lai, Matt Simmons, Aaron Halifax, Paul Downs, Ronny Kim, Jake Richards, Eric Maurer and Tim Speltz, without whom none of this would have been possible. In particular, Mitch Kopelman designed the crowd system at Blue Sky Studios and got it on its feet. Rob Cavaleri was also instrumental in the early design and development of the system and the user interface. Of course, I would also like to thank Ergun Akleman, Carol Lafayette and Jianer Chen for their time and patient guidance.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND MOTIVATION

This thesis is an investigation of autonomous crowd systems in a computer generated feature film. I will discuss how to populate an animated film with hundreds of background characters efficiently and effectively. I will use the movie “Robots” (2005) as a case study.

This thesis reflects my experiences working at Blue Sky Studios on the film “Robots” which was released in theaters on March 11, 2005. Blue Sky Studios was purchased by Fox Feature Films in 1997 with the intention that the studio would produce visual effects for movies made at other studios. This plan was not pursued, and Fox did not know what to do with the studio. The executives at Fox decided that Blue Sky would produce the film “Ice Age” (2002). The scope of the film was modest, and some of the voice talent was already under contract to Fox. The film’s budget was approximately $65 million [1]. This was significantly less than other comparable films at the time.

Blue Sky finished the film on time and on budget, and it went on to earn $176 million in domestic gross box office receipts [1]. The executives at Fox had the confidence to give Blue Sky a more ambitious project. Chris Wedge, one of the co-founders of Blue Sky, and the director of Ice Age, was given the chance to do “Robots”, a project he had wanted to do for several years.

The production of the animated feature film “Robots” involved the creation of an entire imaginary world, including a robot city with highways, buildings, cars and people. All of the tools and techniques that were used to make “Ice Age” were revisited

The journal model is *IEEE Transactions on Automatic Control.*

and expanded. Every step of the production pipeline took on more ambitious and complex methods in order to bring the robot world to life.

The director, Chris Wedge, set the tone for the production crew by telling us "I want it to look like a film crew was sent to Robot land and brought back the film" [2]. One of the needs identified early in production was to fill the movie with background characters. It was obvious that some type of automated system would be needed. The decision was made to tackle this problem "in-house" by building a system from scratch. This thesis is a description and analysis of that process and it's results.

CHAPTER II

PREVIOUS WORK

This section describes both the theoretical and practical work that has been done in flocking and crowd simulation in the computer graphics field. Flocking is the computer graphics term for techniques that mimic the behaviors of schools of fish or swarms of insects. The first widely publicized work on flocking, Craig Reynolds' 1987 paper, introduced the concept of "boids" [3]. Boids do not have intelligent behavior. They move based on some simple rules that each member of the flock must follow for separation, alignment, cohesion, and obstacle avoidance.

Reynolds was fascinated by how flocks of birds moved. He was interested in the idea that a flock of birds was not organized by a single controlling intelligence, but emerged from the individual flight plans of all the members of the flock. He set out to simulate this process on the computer, and the result was his "boids" system.

Reynolds' work set off a flurry of research activity. The concepts have been applied to aerospace, gaming, robotics, and even artificial life [4]. For example, two researchers used "boids" as the inspiration for a model that simulates pilot retention in the Air Force [5]. This was an effort to model complex human behaviors in a complex environment. They classified their model as a "complex adaptive system". The goal was to figure out how to keep trained pilots from leaving the Air Force for jobs in the private sector. The authors argued that a model using autonomous agents and rule based behaviors was the most appropriate for their task.

Agents are a step above boids. They can have intelligent behavior. They can communicate with each other and process information about their surroundings. They are capable of cooperating to achieve goals.

Clearly, the computer graphics field has embraced artificial intelligence, artificial life and robotics as areas that provide novel research opportunities. Craig Reynolds' boids research tied in with ongoing research about complex systems and artificial intelligence. These ideas have been adapted to the feature film industry to varying degrees, according to the budgets, ambitions, and abilities of the film makers.

The practical work in the movie industry spans from physically based modeling to artificial intelligence. Craig Reynolds' boids evoke a sense of complicated group behavior with a few simple rules, making them more than simply objects reacting to gravity or other dynamic forces [6]. At the other end of the spectrum, the software used in "Lord of the Rings" (2001) is an actual artificial intelligence system. The agents learn from one simulation run to the next, and their behavior changes.

Despite the existence of sophisticated commercial crowd system packages like Massive^{TM}and Behavior^{TM}, many studios still use in-house software. For instance, in the movie "The Lion King" (1994) Disney chose an in-house approach. Although the film was produced mostly using traditional hand drawn animation, the story included a stampede of hundreds of wildebeests. The stampede sequence was produced using 3D computer graphics techniques. These images were composited with the hand drawn animation of the lead characters.

The stampede in "The Lion King" was produced by creating groups of wildebeests that followed designated lead wildebeests as their goals. The animators combined different gallops, head tosses and leaps to produce a varied look.

The industry standard for agent based software packages is Massive^{TM}, which was written specifically for the "Lord of the Rings" films [7]. Massive^{TM}is built on a rule-based system. Each agent in Massive^{TM}can have dozens of traits, and dozens of behaviors. The system uses fuzzy logic to string these behavior snippets together, and thus create crowd behaviors.

Fuzzy logic is a concept first proposed by Lotfi Zadeh. As its name suggests, fuzzy logic is a way to use imprecise or “noisy” data as input to a control system [8]. Zadeh’s goal was to mimic human thinking in its imprecision [9]: “The approach outlined is based on the premise that the key elements in human thinking are not numbers, but labels of fuzzy sets - i.e., classes of objects in which the transition from membership to nonmembership is gradual rather than abrupt.”

Massive$^{\text{TM}}$is also based in part on the work of Karl Sims [10] who outlined a genetic programming approach to modeling complex characters and behaviors. Genetic programming is analogous to the Darwinian principle of survival of the fittest. Multiple solutions to a given problem are generated either randomly, or through some type of heuristic or guided approach. The solutions are then tested against a “fitness criteria”, and the best solutions survive to be mutated in the next step of the optimization process.

The examples of “The Lion King” and “Lord of The Rings” illustrate the breadth of tools and approaches being used to handle procedural crowds in feature films. The system developed for “Robots” is probably more general and powerful than the stampede solution in “The Lion King”, but it is not an artificial intelligence system, like Massive$^{\text{TM}}$. The system at Blue Sky is more similar to an implementation of flocking on a 2 dimensional plane.

One important consideration in choosing appropriate solutions to the crowd problem is the expectations of the audience. “Lord of The Rings” is a live action film. The audience is expecting to see real characters in real environments. “The Lion King” is a hand drawn traditional animation film. The audience brings a different set of expectations to this film. Disney films have their own particular style, which can almost be thought of as a visual language. All of the basic principles of animation that can be found in the classic book “The Imitation of Life” can be seen throughout a film like “The Lion King”. Characters squash and stretch, and the motion is stylized to exaggerate anticipation and follow-through. The character designs are simplified, cuter versions of their real world counterparts. This context brings a completely different set of requirements than one has in “The Lord of The Rings”.

“Robots” is a film that falls somewhere between live action and hand drawn animation. The renderer at Blue Sky Studios is based on a physically accurate lighting model. There is a constant effort to create realistic shadows, reflections, refractions, and material properties. On the other hand, “Robots” was marketed in the same way a traditional hand drawn animated film would be. In essence, a new dialect of the language of film is being created by companies like Blue Sky Studios and Pixar. Audiences are receptive to this new technology, and their expectations about the look of a computer animated film have not been fixed. Each film brings a new element to the look.

CHAPTER III

METHODOLOGY

At Blue Sky, we currently use a system developed by Mitch Kopelman, a former Texas A&M Visualization Sciences student who now works at Pixar. The paper "Scalable Nonlinear Dynamical Systems for Agent Steering and Crowd Simulation" by Goldenstein et al. [11]. was used as a theoretical basis for the crowd system in Robots. That paper describes a three layer system. The first layer manages low level behavior. The second layer manages collision detection. The third layer guides the agents toward their goals and avoids local minima. The idea behind the paper is to divide the problem into three sub-problems - locomotion, collision detection, and goal seeking - and then attack each one separately.

The locomotion calculations, (angular heading and speed), for each agent are done in a way that makes them fast, regardless of the number of elements in the total system. The angular heading and speed of each agent are calculated using a set of dynamical system equations. There is one equation for goals that exert an attractive force on the agent, and one equation for objects that exert a repelling force, including walls, obstacles and other agents. A third dynamical system equation weights the contributions of the first two.

Dynamic systems are equations that are evaluated repeatedly - at each step the output from the previous evaluation is used as the input to the equation. Any equation can be used this way. They will fall into a few categories. Some equations will "blow up" to infinity. For example:

$$f(x) = x^2 \tag{3.1}$$

Starting with an input of 3 yields 9. Inputting 9 yields 81, and so on. This is not a very interesting or useful dynamical system.

$$F(x) = x/2.0 \tag{3.2}$$

This dynamical system tends toward 0.0. Again start with 3. This yields 1.5, then .75. Inputting .75 yields .375, etc. The graphs in figure 1 describe the equations used in the Goldenstein paper[11].

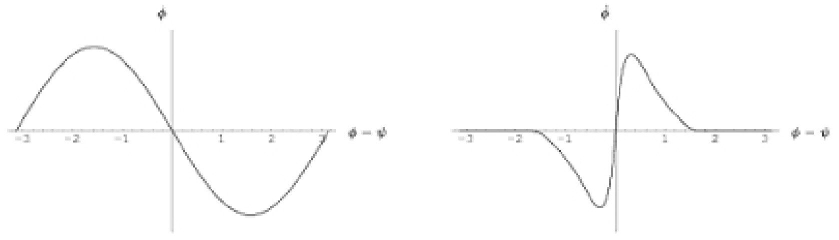

Fig. 1. Goldenstein's dynamical systems curves.

The graph on the left has an attracting fixed point at 0, meaning that starting values near 0 will tend to move closer to 0. The graph on the right will repel input values that are close to 0. The equation on the left is used to pull agents toward their goals. The equation represented on the right is used to repel objects away from obstacles and to avoid agent collisions.

In the second layer of the system, the input data that is needed to do the locomotion calculations is handled separately, using a Delaunay triangulation as the basis for a kinetic data structure. So, the state of the total environment is handled completely separately, and in a completely different way, from the heading and speed calculations. This layer passes information among the agents to help them avoid collisions. The data structure will optimize its work by tagging agents that do not need to be evaluated for a few frames. For example, if two agents are close to each other, but headed in opposite directions, the data structure is able to avoid a collision detection test on that pair of agents for several frames. This makes the kinetic data structure efficient and increases the scalability of the whole system.

The more complicated goal-seeking behaviors are handled in the third layer. This is where the system can be adjusted to alter the traits of individual agents, or to change the environment. This layer is available to the user to choreograph the crowd simulation. The user can input a map of the environment, or a custom vector field, like a texture map, or any type of procedurally based forces to influence the agents. This three layer approach has the advantage of being scalable, adaptive to environmental changes, and mathematically rigorous.

CHAPTER IV

THE CROWD SYSTEM FOR “ROBOTS”

The Blue Sky crowd system provides similar functionality to the one described in the paper above, but with a simpler implementation. The Blue Sky crowd system does collision avoidance and goal seeking, but does not use multiple levels of data management, and does not pass information among the agents. Rather than creating a kinetic data structure, the Blue Sky system loops over all the agents in the crowd at each frame of the simulation. Global planning is achieved by placing NURBS geometry in the scene.

The basic idea of the system at Blue Sky is to move characters on a ground plane while they execute animation cycles, with goal seeking and collision avoidance. The system was implemented as a plug-in to the Maya^{TM}software package. It is a custom field that operates on a particle system, similar to gravity or turbulence. The plug -in was written in C++ using the Maya^{TM}API. Mitch Kopelman designed and implemented the basic structure of the crowd system.

The crowd system was used inside Maya^{TM}. The user saw sliding wire frame proxy agents that represented the final rendered robots, as shown in figure 2. These proxies were attached to particles that carried data, including position, velocity, character type, what animation cycle was being executed and the current frame of that cycle, character radius, maximum turning per frame (in radians), and parameters that modulated target and obstacle influences. This data was used by the custom C++ crowd field to determine position and velocity for each agent at each frame of the simulation.

Simulation data was written out to binary files that were then interpreted by another system that would place the robot in world space and assign it body geometry

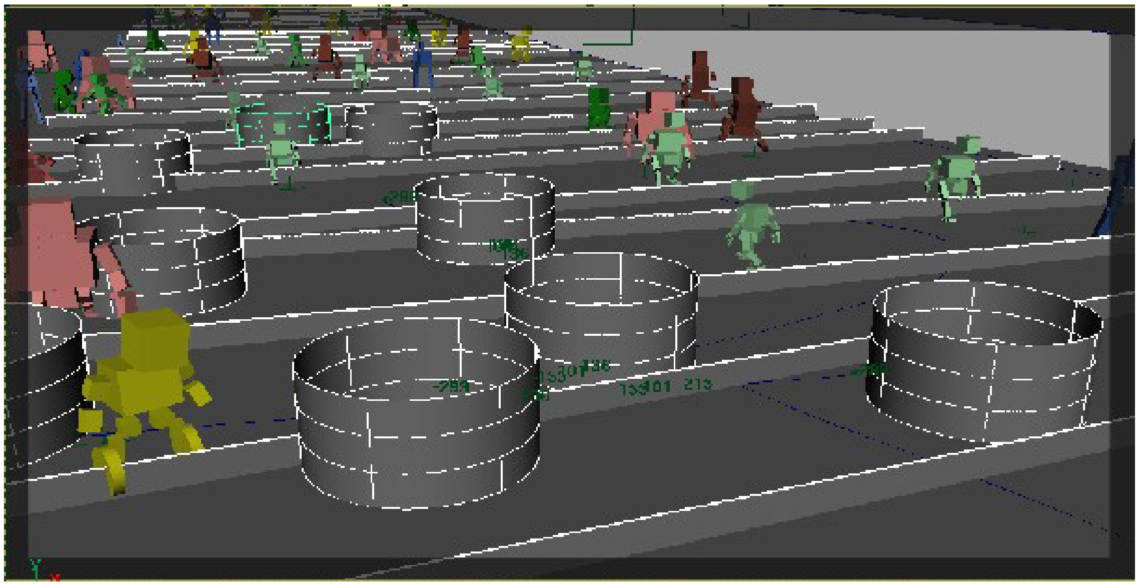

Fig. 2. The crowd system as used in Maya^{TM}.

to replace the low resolution proxies. The robots would then be added to the list of objects to be rendered. This pre-rendering work could take several minutes.

A. The Maya^{TM}Plug-in

The heart of the crowd flocking code was broken into four functions. There was one for avoiding other agents, one for avoiding walls, one for avoiding obstacles, and one for goal seeking. Each function looped over the agents in the system and calculated influences for speed and heading. These function loops provided the same functionality that was described in the first two layers of the Goldenstein paper. The targets attracted the agents nonlinearly as a function of distance. The obstacles exerted a steeply nonlinear repelling force. The attracting and repelling function curves are analogous to the local layer dynamical system equations in the Goldenstein paper. Global planning was addressed by the careful placement of targets and obstacles by the user of the system. These objects were NURBS geometry. The targets and walls were usually planes, and the obstacles were usually made out of cylinders.

1. Distance and Angle Influences

The three repelling functions mentioned above - for obstacles, walls, and other agents - each calculated an influence factor that would turn the agent left or right. These three functions in turn relied on two other functions that separated the influence into angle and distance based components. The distance based influence function was named "D" and the angle based influence function was named "repulsion".

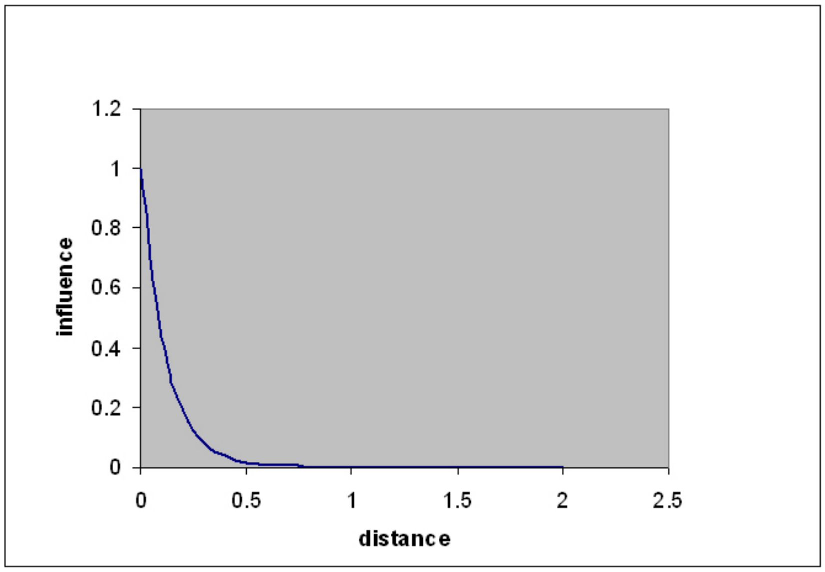

Fig. 3. Object influence as a function of distance.

The function D returned essentially

$$1/(2^{distance}) \tag{4.1}$$

where distance was the distance between the agent and the repelling object. This meant that the effect of an object diminished quickly as a function of distance. After experimentation, this raw value was actually raised to the 12th power, yielding a very steep fall-off, as shown in figure 3. This very steep curve allowed agents to get closer to objects in the environment and yielded a more natural looking simulation.

The second component of an object's repelling force was a function of the angle between the agent's velocity vector and the vector pointing from the agent to the object. This aspect was handled by a function called repulsion. Define the following two vectors:

$$\vec{v} = velocity \quad vector \quad of \quad agent \tag{4.2}$$

$$\vec{o} = agent \quad position \quad - \quad object \quad position \tag{4.3}$$

So, v points in the direction we're traveling, and o points toward an object we want to avoid. If we normalize these two vectors and then take their dot product, the result is the cosine of the angle between the two vectors, meaning it will be between -1 and 1.

$$angle = \vec{v} * \vec{o} \tag{4.4}$$

The repulsion function returned the value:

$$repulsion = ((1 + angle)/2)^2 \tag{4.5}$$

This was exactly what was needed. The repulsion function would return it's highest value if the agent were heading straight toward an object, as shown on the right side of figure 4. It would return 0 if the agent were headed exactly away from the object, as shown on the left side of figure 4.

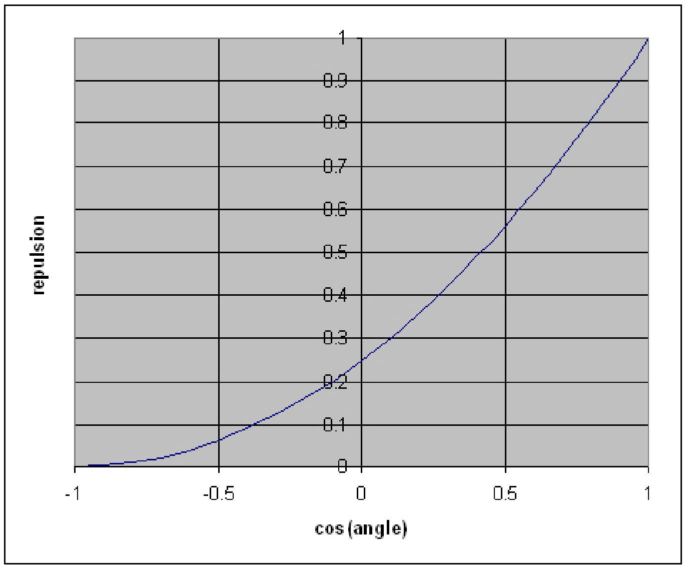

Fig. 4. Object influence as a function of angle.

For example, if an agent is headed along the x axis:

$$\vec{v} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{4.6}$$

and there is an object straight ahead. The normalized vector between the agent and the object is:

$$\vec{o} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{4.7}$$

$$angle = \begin{bmatrix} 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 \end{bmatrix} = 1 \tag{4.8}$$

This implies:

$$repulsion = ((1+1)/2)^2 = 1 \tag{4.9}$$

Now, imagine that the object is 90° to the left:

$$\vec{o} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{4.10}$$

$$angle = 0 \tag{4.11}$$

$$repulsion = ((1+0)/2)^2 = .25 \tag{4.12}$$

So, when the object is off to the side, it exerts less repulsion on the agent. For an object that is behind us:

$$angle = \begin{bmatrix} 1 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 \end{bmatrix} = -1 \tag{4.13}$$

which yields:

$$repulsion = 0 \tag{4.14}$$

In this case the object has a smaller influence. If the object is behind us the repulsion function returns a value of 0. This function provided natural motion and worked well throughout the production without any major modifications.

The repulsion function also calculated the cross product of the same two v and o vectors. This was done to determine if the object was to the right or left. If the detected object was to the right the repulsion force would be multiplied by -1. In the agent's local coordinate space positive angles were to the left of the velocity vector. Looking down on an agent from above, positive angles were in the counter-clockwise direction. In that sense the crowd system used a "right-hand rule". If the object were to the agent's left the cross product vector would point up. If the object were to the agent's right the cross product vector pointed down, and the sign of the repulsion value would be flipped. The images below illustrate the use of the cross product to determine orientation. The left side of figure 5 illustrates the case when the object is to the left of the agent. The right side of figure 5 illustrates the case when the object is to the right of the agent.

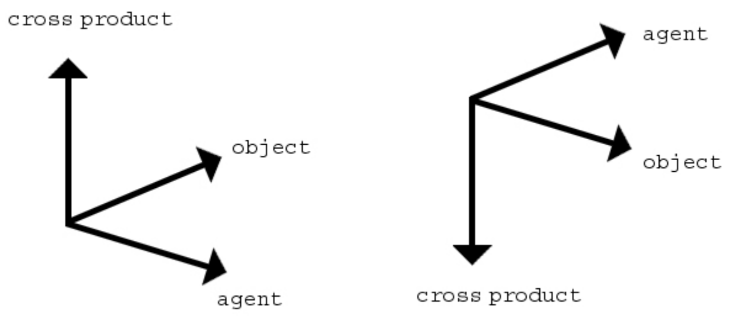

Fig. 5. Cross products of object and agent vectors.

In the case on the right, the repulsion value would be multiplied by -1.

The repulsion and D values were used to directly weight the agent's heading. Generally the maximum allowed turn was 5° per frame. The various objects' repulsion and D forces were summed, some canceling others, and the agent's velocity vector would be rotated and scaled accordingly. The values were weighted using parameters that could be set on a per-agent basis. This provided a highly flexible control mechanism.

2. Avoiding Other Agents

The agent to agent collision function looped over all the other agents in the system and determined the influence on speed and heading due to any other nearby agents. It used the angle and distance calculators, repulsion and D, that were described above. This collision detection function had one special feature to handle agents walking in line one behind the other. In that case, we did not want the agent turning to avoid the agent directly ahead of him. This would lead to violent shaking in dense crowds.

If another agent was detected nearby, a variable called "chase" was calculated. It was the dot product of the (normalized) velocity vectors of the current agent and the other nearby agent:

$$chase = agent \quad velocity \quad * \quad other \quad agent \quad velocity \tag{4.15}$$

This value was used to decrease the influence of the other agent if it were traveling in the same direction as the subject agent. If chase were close to 1.0, that meant that the velocity vectors were almost identical, and the agents were traveling in more or less the same direction. In this case, the agent should not try to turn away from the other agent that is nearby. In pseudo-code, this is how the chase value was used to decrease the influence of the other agent:

$$If \quad (chase > 0)\{influence = influence * (1.0 - chase)\} \tag{4.16}$$

The conditional restricts the influence of the chase variable so that it only comes into play when the other agent is traveling in the same basic direction as the current agent. The agents' influence on each other would decrease as their paths became more parallel to each other and the chase value got bigger.

It should be kept in mind that the chase parameter was used to modify the repulsion value. When the repulsion value was small, the attenuation would be slight - the chase parameter would not have a big influence on the repulsion value. That was the case when the agents were walking side by side.

On the other hand, when the repulsion force was strong, when the two velocity vectors were almost parallel and the other agent was almost straight ahead, then the chase parameter had its strongest effect. This adjustment took a lot of instability out of the system, and allowed the crowds to be denser, which was the ultimate goal of a lot of the early work on the flocking code.

3. Avoiding Walls

This third function looked for walls in the agent's radius. The wall's location was determined by finding the closest point on the wall to the agent. The agent's heading was adjusted to avoid the wall. The walls were not used very frequently. In most cases the cylindrical obstacles proved to be much more useful.

4. Avoiding Obstacles

An obstacle was similar to a wall, except that it was treated as a point with a radius. Again, the repulsive force was calculated based on angle and distance using the two functions mentioned previously. In Maya^{TM}, NURBS cylinders (see fig. 2) represented obstacles. These would often be animated to represent hand animated characters that were already placed in the scenes earlier in the pipeline.

5. Goal Seeking

The simulations could also be directed by setting up goal objects. These could be any piece of NURBS geometry, but were generally planes. These had an exponentially decaying attractive force as a function of distance. The chart below represents the parameter that controlled goal influence as a function of distance. This value was calculated by the same distance function, D, that was used for all the repelling loops above. In this case though, the value was not raised to the 12th power. Figure 6 reflects this difference, and illustrates that the goal objects effected the agents at greater distances.

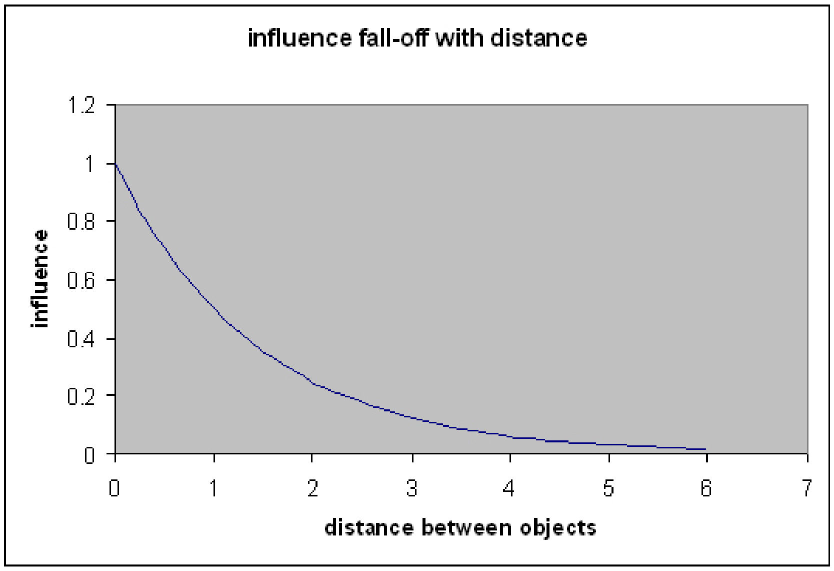

Fig. 6. Goal influence as a function of distance.

The agents would pick the closest object to them, and after moving to within that object's radius, they would move to the next closest object. A tool was created in Maya^{TM}to interactively place targets by clicking with the mouse. The target planes had to intersect the ground plane that the crowd was on. Targets could be removed by lifting them off the surface. Targets could also be animated, although this wasn't necessary very often.

6. Prioritization

One of the biggest design issues for the crowd system was to try to create a system that would correctly prioritize the agents' multiple objectives. Targets would draw the agents into collisions with each other, and obstacles would force the agents through walls. This problem lingered as part of the design, although the addition of per-agent handles allowed the user to adjust the prioritization for individual agents, and thus fine tune the simulations.

Reynolds mentions this issue in his work on boids [3]. His solution is similar to ours. He avoids artificial intelligence techniques in favor of parceling out the allowed acceleration according to a pre-determined prioritization list.

This whole problem was made more severe by the fact that the agents were running animation loops of fixed length. So, if an agent was pushed into an alley made of two walls configured in a "V" they were bound to violate a wall. They were bound to walk until they reached the end of their cycle. The walls would exert a symmetrical force that would prevent escape. This is similar to reaching a global minimum in an optimization problem.

The crowd system was modified to include handles for each agent that would change the relative weighting of the targets, obstacles, walls and other agents. With these handles the user could make one agent blind to walls, and make another extremely goal-oriented. Modifying these priorities on a per-agent basis was a useful way to fine tune the simulations. Adjusting individual performances while minimizing the changes to the rest of the simulation was a huge benefit. Changing the parameter values on individual agents, rather than the crowd as a whole, gave more control to the user. If the user had been restricted to changing the priorities of the crowd as a whole, the simulations would have been more unwieldy. These handles allowed localized changes.

B. Animation Database

The crowd system was designed essentially as a system to move characters along while they executed animation cycles. The animation database was made up of the inventory of animation cycles that were available to the extras in the crowd system.

The crowd animations were done on a group of characters that used special rigs. The rigs were part of a sister technology called frankenbots. The crowd system and the frankenbot system were designed and built by Mitch Kopelman to work together hand in hand.

Frankenbots were simplified robot rigs that had rectangular boxes for all their major body parts. At render time these boxes were filled with high resolution, fully modeled robot parts. So, the crowd system also used a database of robot body parts. Figure 2 shows some of the bounding box characters.

In the simplest case those boxes would be replaced by a set of randomly assigned body parts. For ease of use, though, the parts database was organized into groups or wardrobes. This was done mostly to distinguish the old, low end robots from the new, shiny high end robots. The parts were also grouped in response to art direction. Certain heads turned out to be objectionable, for example. As the production proceeded the wardrobe groupings were groomed to use a minimal set of parts. This was done to reduce RAM usage at render time. Through trial and error it was found that putting a large variety of heads on robots that all shared a very small collection of body parts would provide almost enough variation for most situations.

There were several different frankenbot rigs, with names like bipedThinA, bipedTallB, biwheelA, uniwheelB. All of the biped rigs were scaled versions of an original, basic biped rig. The body parts were scaled differently to create different characters. As the names above suggest, the rig could be adjusted to produce characters with larger torsos, longer arms, or larger hips. The animators found that they were able to transfer animations between rigs with only minor cleanup. The keyframed curves were copied directly from one character to another.

The cycles that did not involve locomotion were categorized as “idle cycles”. These were the easiest to handle in the crowd system. There was no flocking happening, so the scene did not need targets or obstacles, and the simulation was guaranteed to be almost flawless right from the start.

In contrast, the walking characters often needed a lot of herding and coaxing to behave as required. Targets would be set up, obstacles added and shifted, and individual agent attributes would be adjusted.

As mentioned previously, one of the limitations of the crowd system was the fact that the agents were running animation loops of fixed length. The crowd system did not include any functionality to interpolate between animations. A character could not be switched from the middle of a walk cycle to an idle loop. These transitions had to be animated. The agents could switch from one loop to another, but the animation had to be seamless between two loops.

The problem with running animation loops was that the agents had to keep moving until the end of the current loop. This could be problematic in dense crowds. There were cases when there was no place to go, but the characters had to keep moving because of the animation loop they were running. This problem was dealt with by careful choreography of the crowds. This usually meant adding obstacles to change an agent’s direction.

Achieving high density crowds was a major hurdle. In the earliest versions of the crowd system the agents would shake violently and walk through each other if the crowds became too dense. The first thing that was done to try to alleviate this problem was to increase the agents' radii. This solved the shaking and penetration problems, but also kept the agents too far apart from each other.

The next solution to this problem was to slow the characters down. The characters' maximum allowed speed would be reduced as they approached other objects in the scene. Reducing their allowed speed in confined spaces, as mentioned above, would effectively give them more frames in which to avoid an object in the scene, turning at 5° per frame. Dense crowds would start to move in slow motion, which still looked wrong.

The next step in achieving density was to lift the constraint on the agents' maximum turning per frame. Initially the characters were allowed to change their heading by a maximum of 5° every frame. This restriction was lifted if they got very close to an object.

Rather than going into slow motion, we would lift the constraint on the agents' maximum turning when they really needed to make a tight turn. This created a new problem - the agents would pivot unnaturally, but this was less conspicuous than having them move in slow motion.

In the end, both adjustments were used. When characters got into tight spaces they were allowed to slow down to as little as 90% of their base speed. Their maximum turn per frame was allowed to increase by up to a factor of 26. The turning constraint was relaxed with the square of the distance to the object, as illustrated in the pseudocode below:

```
dist = distance to object
turn = maximum turn per frame
```

if (dist < 5.0) {

$$turn = turn * (1.0 + (5.0 - dist)^2) \qquad (4.17)$$

}

If the object were 3 units away, (quite close), the turn restriction would be relaxed to allow a turn of 25°. At 2 units away the maximum allowed turn would be 50°. This is illustrated in figure 7.

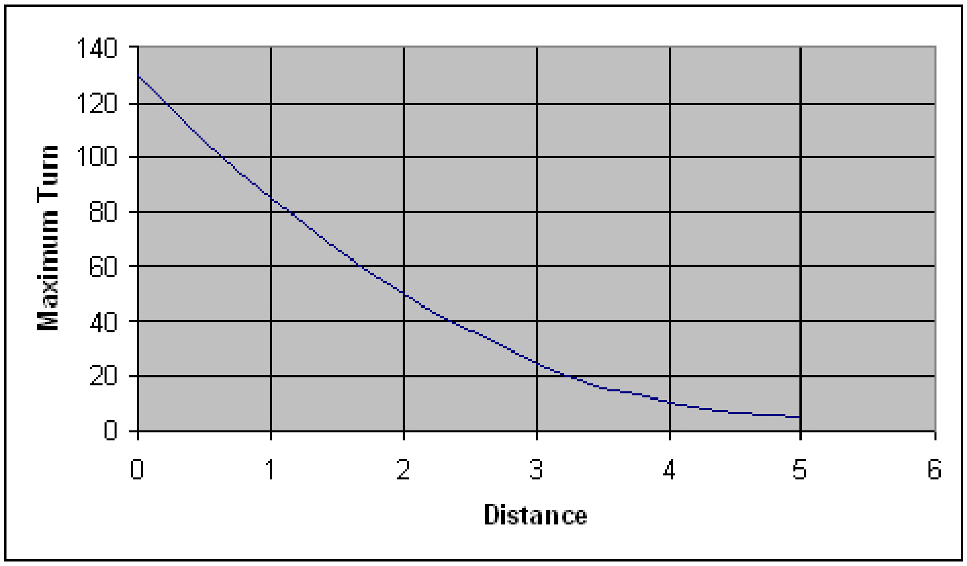

Fig. 7. Sharper turns were allowed as objects got closer.

These adjustments to the crowd field were not perfect, but they allowed for denser crowds. The one thing that was to our advantage was that dense crowds tended to mask individual problems. Among 200 agents it was less noticeable if one character was pivoting or slowing down.

Careful arrangement of the agents and objects in the scene was the key to achieving an acceptable look. When problems were unavoidable in the simulations, often the simplest solution was to remove the problem agent from the render, so that the rest of the simulation could still be used. Re-running a simulation to try to correct one bad agent would usually change the entire simulation in unpredictable ways. Often this new simulation would have its own new problems - awkward motion or unfortunate random geometry assignments. This would begin a back and forth process that was time-consuming and unsatisfying for all involved. Eliminating the offending agent from the render would retain the otherwise acceptable simulation, and reduce the amount of reworking by the lighters and the crowd crew. Sometimes the crowds were made slightly denser than required, with the knowledge that the lighting crew would pick and choose which crowd agents they wanted to keep.

One other minor aspect of the crowd animation database is worth mentioning. The animators were instructed to animate characters moving in a straight line - along the positive Z axis. This gave a vector in the character's local coordinate space that could be matched with the particles velocity, so that the characters moved in the direction they were walking. This was the amount to move the particle each frame to assure that the character did not slip. If the speed were wrong a walking characters's planted foot would slide when it was supposed to be stationary. It would look like the character was on a moving sidewalk.

In the beginning of the production the characters were restricted to moving at a constant rate. As the crowd system matured that restriction was lifted. If the animators provided a keyframed curve of the character's Z translation, that curve could be used in the crowd system. The absolute Z translation numbers were "differentiated" to yield per-frame Z translation values. These values were stored as a new curve. This curve was accessed as needed, so that if an agent were on frame 17 of a walk cycle, the translation rate for frame 17 would be used to move the character in world space.

C. User Interface

The user interface had a few simple elements. There was a window that was used as an interface to the crowd field, which was the plug-in that calculated each character's heading and speed. This window gave access to the field's data - such as which ground plane the crowd was using, which body types were being used, and which animations were available to each character. The crowd field also kept track of which groups of geometry, ("wardrobes"), were available to the characters, and this list could also be edited through the window interface.

The other main user interface window was used to place the crowd agents on the ground plane, set their initial heading, and assign body types and animations. This window also had a button for deleting agents.

Deleting agents actually involved a lot of work behind the scenes. Maya^{TM}does not allow the user to delete a particle from a particle system. Deleting an agent was accomplished by making a new empty particle system and putting duplicates of all the undeleted agents into the new system. The old system was deleted, the new system was given the name of the original system, and all of the connections to the other nodes in Maya^{TM}had to be recreated.

A tool was developed to automate the creation of targets. The user would create a NURBS curve by clicking with the mouse. A very simple script would then constrain a NURBS plane to each control vertex of the curve. The planes were constrained to be perpendicular to the tangent of the curve. The targets could be moved as a group by moving the curve, and moving the control vertices of the curve would move the targets. Often these constraints were broken as fine tuning progressed, but the tool was still useful for quick setup.

A tool called “botInfo” was developed for trouble shooting. If the user selected one or more particles then hit the botInfo button, a list of useful information would be printed out. This was usually the first step in trying to fix an agent that was misbehaving, or had objectionable geometry. Typical output would be something like this:

```
Robot ID: 78
Rig: BipedThinA
Cycle: WalkA
Frame: 12
Scheme: LowEnd_good_heads
```

The scheme was the wardrobe - the group of geometry that would be used to replace the agents bounding box body parts. This tool was an afterthought, but it was tremendously useful.

CHAPTER V

RESULTS

A. Crosstown Express

The crowd system went from research and development to full production with little warning. The first sequence that required crowds was the "Cross-town Express" scene. This was the first establishing scene that gave the audience a sense of the scope of the robot world. The scene was animated and mostly lit before it was determined that the city lacked a sense of scale, and looked lifeless. The sequence did not give a sense of the massive scale of Robot City.

The crowd system was used to bring life and a sense of scale to the scene. Although the crowd system itself was in fairly good shape, the crowd robots did not have very good materials, and the animation database had only a few test cycles available. Some of the shots from this scene were reworked slightly at a later time to address some of the animation and materials issues.

A few things became clear as soon as the first shots were passed down the pipeline to the lighting department. Some animation cycles were distracting because they were so distinctive. There was one bouncy strut that could be picked out immediately anytime it was on screen.

There was also one run cycle that would draw the viewer's eye away from the primary action of the shot. The running cycles were also problematic because the characters would often run in a path that was nonsensical. They would run off-screen, and then come back, or run to the far background of the set, turn sharply around, and then run toward the camera. This was the cause of much lamentation during lighting rounds. These animation cycles were culled from the offending shots, and were avoided for the duration of the production.

One other issue that came to light at this stage in the production was that certain heads were not meeting the standards of the art director. These were almost spherical heads that somehow looked wrong. Investigating this issue led to two decisions. One was to eliminate the offending heads from the wardrobes. The other adjustment was to scale the remaining heads uniformly in X, Y and Z directions. The default behavior of the frankenbot system was to scale the parts as needed to fill the rectangular bounding boxes. For example, a thigh might be shrunken slightly in its length and expanded in its diameter, if the rig were for a short, stocky character. When the heads were scaled in this non-uniform way the eyes went from round to elliptical, and the heads looked inflated and sickly.

For some of the final shots in the crosstown express sequence one long simulation was run to preserve continuity. The simulation was used in the conversation at the end of the sequence, where it would have been jarring to see different crowds behind the characters as the camera cut back and forth during a conversation. The shots indexed into the simulation so that the conversation appeared to be happening in real time. For example, if the first shot were 89 frames long, it would use up the first 89 frames of the simulation. The first frame of the second shot would start with frame 90 of the simulation, and continue on from there. This approach had good and bad aspects. On one hand, it was efficient to re-use the same simulation for multiple shots. On the other hand, it was generally more challenging to get longer simulations to behave as desired. In this case it was not so difficult to get a satisfactory long simulation because the characters were not densely packed, and there was no specific note about what they had to do. As long as they weren't drawing attention away from the main characters the simulation was acceptable.

The cross-town express was a valuable learning experience. It would have been very difficult to predict the types of issues that eventually arose in that first production scenario.

B. Train Station

The train station sequence was the next chance to use the crowd system extensively. By this time there were more animation loops in the database, and even more characters to use. The materials had also been improved significantly, and the characters were blending into the shots without drawing as much unwanted attention.

The train station was supposed to be overwhelming and frenetic, like Grand Central Station. The crowd system was a more integral part of the sequence, and much larger crowds were used than in the cross-town express sequence.

The large numbers of agents led to a problem for the lighters. The crowd agents would use up all of the 2.8 gigabytes of RAM on the render processors, making them unrenderable. The solution was to break the crowds into renderable layers. A simple sorting script was written to rank the agents by distance from camera, and they would be assigned to different layers that would each be rendered separately. These could then be composited one over the other.

The first method for splitting the crowds into layers was to run the sorting script on every frame. This was occasionally problematic, because characters would pop from one layer to another. That was not a problem by itself, but if the character were casting a shadow on another agent, the shadowed character would brighten visibly as the shadow left with the transient agent. The solution, obviously, was to pick a frame in the middle of the shot and use only that frame for calculating distance to camera.

The layering solved the problem of rendering the agents, but it left another problem unsolved. The train station floor was polished and reflective, and the lighters needed to include most of the scene geometry in the render of the floor. Although this geometry was not rendered in the image, it had to be loaded into memory in order to cast correct reflections on the floor. The crowds had to be included in the floor renders because they were casting reflections on the floor. When the crowds were included with all of the other scene geometry that was also reflecting on the floor, the RAM usage issue became even more acute.

The RAM was being used up by the crowd system during the frankenbot phase, when the polygonal bounding boxes were being replaced with real robot geometry. The solution was to simply skip that step. In the reflection passes for the train station floor the crowd agents were made out of low resolution polygonal rectangles. The same materials were assigned to the boxy robots that were assigned to the rendered robots in the other crowd layers, so that the reflections look correct. Also, the roughness coefficients on the floor material were increased so that the reflections were not so distinct or precise.

Another interesting extension of the crowd system that was developed during the train station sequence was getting agents to sit on the benches in the station. This seemed like it was going to be a problem, but with a little cooperation from the animation department some conventions and standards were set up, and the animators were able to provide sitting loops that matched the benches in the set. These were input into the crowd system, and used extensively. The only caveat to having characters sit on benches was that they could not be scaled. The default behavior of the crowd system was to randomly scale the characters slightly. That was done so that two characters of the same type that were doing the same animation would be of slightly different height, and travel at slightly different speeds. This gave the crowds more variation.

Since the agents were scaled from a pivot point on the floor, increasing a sitting agent's scale would lift it off the bench, and scaling it smaller would make it sink into the bench. The seated agents were handled as separate crowds. These were systems without flocking that went through the crowd pipeline very smoothly.

The default mode of the crowd system was to add a small amount of random scaling to each agent. This created more variation in the crowds, and also varied the speeds of the agents a little bit. The random factor was added in this form:

$$height = height + rand(-.1, .1) * height \tag{5.1}$$

where the rand() function is a uniformly distributed random number generator. Any statistician would be disappointed to hear this, and might wonder why a bell curve was not used. The reason is that a bug was discovered in Maya^{TM}that prevented effective seeding of the "gauss" random number generator. This bug would change the agents' heights every time the simulation was re-run or re-output. This was hugely distracting to the lighters, and so a uniform distribution, with stable seeding, was used. It should be noted that no one noticed that the agents' heights were uniformly distributed.

The other innovation that occurred during the production of the train station sequence was to give the agents props. The animation department produced some cycles of characters carrying luggage. Although the system had not been designed to specifically handle characters carrying things with them, the implementation was almost transparent. Mitch Kopelman's initial design of the crowd and frankenbot systems was straightforward and robust. The animators had to be careful about keeping the props within the character hierarchy. Other than that, any geometry could be included in the crowd system.

This idea was explored further. The animation department needed a method to populate the city streets with vehicles. Some vehicles were put into the crowd system. Adding the vehicles was very simple, but eventually a separate, simpler instancing system was developed that allowed the animators to use their normal work flow to generate background vehicular traffic.

C. Minions

The next major hurdle for the crowd system was the final fight sequence in Madame Gasket's chop shop. This sequence used even more agents than the train station sequence, although there were fewer shots. The main challenge in the chop shop sequence was putting the minion characters into the crowd system.

The minions were made out of pieces of rusty scrap and junk. They were not part of the frankenbot system. Their torsos were procedurally populated with random pieces of junk. The junk was placed randomly on a hidden NURBS torso surface. The sampling parameters could be adjusted to change the look of the characters. The scale of the pieces could be changed, and there were handles to control their orientation and displacement off the sampling surface.

The hidden torso surface was a piece of deforming geometry. The crowd system could not handle the deforming geometry because, unlike the polygonal bounding boxes used with the frankenbots, the deforming geometry was defined by a list of points, rather than a transformation matrix.

The solution was to create a new version of the minion that had three non-deforming torso pieces in place of the deforming geometry of the original torso. This new minion had to be shepherded through the normal modeling and rigging pipeline, which actually went fairly quickly and smoothly. The new minion then had to be adjusted to match the look of the non-crowd minions.

These new crowd characters used much less RAM than the frankenbots. It was possible to render more than 200 minions in one layer, compared to 50 or 60 frankenbots. The instancing of their body parts worked more effectively than the frankenbots, so that the memory usage was very low, and rose slowly after the initial overhead of loading the first minion into RAM. Instancing a piece of geometry means loading it into memory once, and then accessing that location in memory any time that part is called for again, anywhere else in the scene. In the case of the minions, their torsos were all made out of the same 16 pieces of junk. The random placement and orientation of the pieces made each minion unique.

There was one other small bug that was addressed at this time - the parts were being instanced, but the materials were not. Although this was a known issue, it had been inconsequential until the crowd minions started using the same 16 pieces hundreds and hundreds of times. Instancing materials yielded significant RAM savings.

In summary, the crowd system essentially worked as designed and intended. The Art Director called out the shots and on-screen locations where crowds were desired. The Maya$^{\text{TM}}$scene files were stripped down to include just the manageable set pieces that were needed - camera, ground planes, and buildings that were needed for reference. Agents were placed, along with targets, walls and obstacles. The crowd simulation was run and adjusted, then the data were written out to binary files. The crowds were delivered to the lighters, who incorporated them into the shots for final rendering.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Crowd systems are becoming more common in live action and computer generated films. They seem to have two distinct uses - either to fill in random passers by, or to overwhelm the viewer with staggering numbers of characters. “Robots” mostly used crowds in the first way - to bring life to a fully synthetic environment, although it was also used for establishing shots during the large climactic battle scene. Reviews of the film praised the look of the film generally, and the rich level of detail that was brought to the world [6]. In that sense, one could argue that the crowd system was a success.

There were problems, however. The crowd tool was controlled by the Effects Department, and was never handed off to the Animation Department to use. The animators at Blue Sky Studios are extremely talented, and take great pride in their work. They were not always comfortable seeing animation added to shots after the shots had left their department. This created a tendency on their part to avoid the crowd system. They would sometimes animate background characters by brute force, using very large, heavy scene files that would slow their computers to a crawl.

There were also technical limitations that prevented more widespread use of the crowd system. Although repeated body parts were instanced as much as possible, RAM usage was still a problem. The first effort to address this was to break the crowds into layers that could be rendered separately and then composited together. This led to issues with shadows and reflections, and would sometimes create extra work for the lighting crew.

Another problem with the crowd system was the fact that the frankenbot system would take several minutes to load in all the robot geometry. This made the lighters’ workflow more time-consuming and unpleasant. Although we provided a caching system, it was not adopted widely. This was probably because of the stern warnings about deleting the caches before final render. The caches had to be deleted for two reasons. One was that the cache might have become out of date. The second reason was that the files were large. The cache files were big enough that it would not have been practical to cache every frame of every crowd simulation, even though that would have completely eliminated the slow pre-processing step.

There are areas for future work. The crowd system was used with characters that held props - drinks, luggage, spears and other weapons. In one case a crowd was made out of just weapons without using the characters that were holding the weapons - the characters were hidden by the low camera angle, but their tall spears were visible. This brings up the possibility that the crowd system could be used as a general tool for set dressing. Using a database of set pieces would save hard disk memory, and would reduce work. Changes could be made through the crowd system, and would eliminate the need to make the changes in the scene files. This idea would really turn the crowd system into a different tool, but the possibility is there. It could make the pipeline a little more modular.

The other most useful augmentation to the crowd system would be the ability to bake simulations. That is, it would be useful to be able to keyframe simulations that worked well. The keyframed agents would change categories from simulation to animation. One possible scenario would then be that additional agents could be added to the system. They would react to the baked agents, but the baked agents would not react to them. Other agents could be added to the system without changing the behavior of the agents that were already behaving as desired. Also, baked motion paths could be copied and pasted onto new agents to allow for very quick and flexible choreography of large crowds.

Some work was done toward capturing simulations, but it proved to be slightly more difficult than expected. The simplest approach was to move a locator to the agent's position at every frame and keyframe the position of that locator. This created very heavy maya$^{\text{TM}}$files that made the user interface sluggish. Also, the keyframing produced inaccuracies. The keyframes created curves with tangents. The tangents would effect the animation. Clearly, the problem of capturing desirable simulations could be solved, given the motivation and time.

As mentioned previously, Blue Sky Studios considered buying an off-the-shelf commercial product to handle the crowds in "Robots". It seems worthwhile to ask the question whether or not it was a good decision to tackle the problem in-house.

In the computer graphics field today there are several commercial packages that provide flocking behavior of agents. During the production of "Robots", Blue Sky Studios considered purchasing Softimage's Behavior$^{\text{TM}}$system, which is able to blend smoothly between animations, and to put multiple animations on a character at once. Behavior$^{\text{TM}}$could also work on hilly terrain, but it did not give enough flexibility for the movie, and would have required a substantial reworking of the existing pipeline at Blue Sky. In addition, at $10,000 per seat, Behavior$^{\text{TM}}$would have been a costly solution to the problem.

Although there was a good deal of interest in Softimage's Behavior$^{\text{TM}}$system on the part of the animation department, the feeling that prevailed among the ultimate decision makers was that it would be much wiser to develop a tool from the ground up. There were many advantages to this approach.

Avoiding the dependence on an outside software vendor was a huge advantage. Feature requests could be brought literally down the hall. There was no risk of having to pay for custom features in the future. Owning all the source code for our own tool meant that the entire tool was transparent. Problems could be addressed without having to make guesses about what the tool was doing, or what was going on “under the hood”.

Integrating a commercial product into the “Robots” pipeline would have required a substantial effort on it’s own. This effort was probably much better spent writing flocking code. These integration tools would have required maintenance as the commercial software went through version changes, and bug fixes. The effort to integrate outside products into the pipeline would not have been a wise investment in core technology.

There were some nice features in the commercial products that the in-house product never delivered, but it is a safe bet that many unforseen problems were avoided as well.

REFERENCES

[1] B. Nash, *Box Office History for Ice Age Movies.* http://www.the-numbers.com/movies/series/IceAge.php, 2005.

[2] C. Wedge, Personal Communication, Blue Sky Studios, 44 S. Broadway, White Plains NY 10601, March 24, 2003.

[3] C. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Proc. SIGGRAPH '87*, pp. 25-34, 1987.

[4] C.Reynolds, *Boids, Background and Update.* http://www.red3d.com/cwr/boids/, 2005.

[5] M. Gaupp and R. Hill, "Using Adaptive Agents in Java to Simulate U.S. Air Force Pilot Retention," *Proc. 1999 Winter Simulation Conference*, pp. 1152-1159, 1999.

[6] M. Cavazza, R. Earnshaw, N. Magnenat-Thalmann, and D. Thalmann,"Survey: Motion Control of Virtual Humans," *IEEE Computer Graphics and Applications*, vol. 18, no. 5, pp. 24-31, Sep/Oct, 1998.

[7] Massive Software, *Rhythm & Hues Uses Massive's Smart Stunts for Elektra.* http://www.post601.com/news/2730, 2005.

[8] L.A. Zadeh, *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems.* World Scientific Publishing, 1996.

[9] L.A. Zadeh, "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes," *IEEE Transactions on Systems, Man, and Cybernetics* vol. 1, no. 3, pp. 28-44, Jan. 1973.

[10] K. Sims, "Evolving Virtual Creatures," *Proc. ACM SIGGRAPH '94,* pp. 15-22, 1994.

[11] S. Goldenstein, M. Karavelas, D. Metaxas, L. Guibas, E. Aaron, and A. Goswami, "Scalable Nonlinear Dynamical Systems for Agent Steering and Crowd Simulation," *Computers and Graphics,* vol. 25, no. 6, pp. 983-998, 2001.

VITA

John Andre Patterson

5-33 49th Avenue

Long Island City, NY 11101

yesthatjohn@hotmail.com

Education

M.S. in Visualization Sciences	Texas A&M University, December 2005
M.S. in Operations Research	The University of Texas at Austin, January 1994
B.A. in Pure and Applied Mathematics	Boston University, January 1990

The typist for this thesis was John Patterson.