Scalable Dynamical Systems for Multi-Agent Steering and Simulation

Siome Goldenstein[1] Menelaos Karavelas[2] Dimitris Metaxas[1] Leonidas Guibas[2]
Ambarish Goswami[3]

[1]University of Pennsylvania {siome,dnm}@graphics.cis.upenn,
[2]Stanford University {menelaos,guibas}@graphics.stanford.edu,
[3]Discreet ambarish.goswami@autodesk.com

Abstract

We present a new methodology for agent modeling that is scalable and efficient. It is based on the integration of nonlinear dynamical systems and kinetic data structures. The method consists of three-layers that model steering, flocking, and crowding agent behaviors among moving and static obstacles in 2 and 3D. The first layer, the *local layer* is based on the use of nonlinear dynamical systems theory and models low level behaviors, it is fast and efficient, and does not depend on the total number of agents in the environment. The use of dynamical systems allows the use of continuous numerical parameters with which we can modify the interaction of each agent with the environment. This creates controllable distinctive behaviors. The second layer, a global *environment layer* consists of a specifically designed kinetic data structure to track efficiently the immediate environment of each agent and know which obstacles/agents are near or visible to the given agent. This layer reduces the complexity in the local layer. In the third layer, a *global planning layer*, the problem of target tracking is generalized in a way that allows navigation in maze-like terrains, avoidance of local minima and cooperation between agents. We implement this layer based on two approaches that are suitable for different applications. One is to track the closest single moving or static target. The second is to use a pre-specified vector field. This vector can be generated automatically (with harmonic functions, for example) or based on user input to achieve tht desired output.

We demonstrate the power of the approach through a series of experiments simulating single/multiple agents and crowds moving towards moving/static targets in complex environments.

1 Introduction

The modeling of autonomous digital agents and the simulation of their behavior in virtual environments is becoming increasingly important in computer graphics and robotics. In virtual reality applications, for example, each agent interacts with the other agents and the environment. Complex interactions in real time are necessary to achieve nontrivial behavioral scenarios. Modern game applications require the creation of smart autonomous agents with varying degrees of intelligence to allow for multiple levels of game complexity and a variety of behaviors. These behaviors must allow for complex interactions and must be adaptive in terms of both time and space (continuous changes in the environment). Finally, the modeling approach should scale well with the complexity of the environment geometry, the number and intelligence of the agents, and the level of the various environment-agent interactions.

There have been several promising approaches towards achieving the above goal, but many of them are restrictive in terms of their application domain and they do not scale well with the complexity of the environment. This paper attempts to develop a mathematically rigorous approach to modeling complex low-level behaviors in real time that is scalable, adaptive and suitable to a distributed application.

We develop a three-layer approach to modeling autonomous agents based on the integration of nonlinear dynamical system theory, kinetic data structures, and harmonic functions. The first layer consists of differential equations based on nonlinear dynamic system theory which model the behavior of the autonomous agent in a complex environment. The second layer incorporates the motion of the agents, the obstacles and the targets into a kinetic data structure, and provides a very efficient and scalable approach for adapting an agent's motion based on its changing local environment. Finally, differential equations based on harmonic functions, the third layer, provide a method for determining a global course of action for an agent that is used as an initialization to the differential equations from the first layer, guiding the agent and keeping it from getting stuck in local minima.

In the first layer, through the use of nonlinear dynamical systems theory, we characterize in a mathematically precise way the behavior of our agents in complex dynamic environments. The agents lie in a constantly changing environment consisting of obstacles, targets and other agents. Depending on the application, agents reach one or multiple goals, while avoiding multiple obstacles. Both targets and obstacles can be static and/or moving.

Our agent modeling is based on the coupling of a set of nonlinear dynamical systems. The first dynamical system is responsible for the control of the agent's angular velocity. It uses carefully designed nonlinear *attractor* and *repeller* functions for targets and obstacles, respectively, to change the facing direction. Due to the nonlinearity of these functions a direct summation can generate undesired attractors that would lead to collisions and other unsuitable behaviors. To remedy this problem we use a second nonlinear dynamical system which automatically computes the correct weighted contribution of the above functions at each time instant. A third dynamical system controls the agent's forward velocity which, depending on the situation may or may not take into account the value of the angular velocity.

To model low-level personality attributes, such as agility or aggressiveness, we extend the above set of equations through additional parameterization. The attributes are modeled using this additional parameterization of the governing equations. Agents with different personalities will react differently in the same environment, given the same set of initial conditions. Our approach is general, and other low-level behaviors can be modeled easily if needed in a given application.

Each agent is described by its position, geometrical shape, heading angle, forward velocity and personality. The nonlinear dynamical systems which model the changes in the above parameters are based on local decisions. Therefore, to avoid local minima in the solution and to generate a *nominal* global trajectory we use *harmonic function* theory. This nominal trajectory is computed using only the static objects in the environment. Harmonic functions are solutions to the *Laplace equation*, and they create an artificial potential field for the agent to follow.

In addition to the “intrinsic” personality attributes of each agent, the agent’s behavior is affected by “extrinsic” factors in its immediate environment, such as the presence of obstacles, other friendly or hostile agents, etc. In order to make our approach efficient and scalable with the number of agents and the geometric complexity of the environment, we use *kinetic data structures* (KDS for short) to track the immediate environment of each agent and know which obstacles/agents are near or visible to the given agent. KDS [4, 15] is a general framework for designing algorithms tracking attributes of a continuously evolving system so as to optimally exploit the continuity or coherence of the motion. A kinetic structure works by caching a certain set of assertions about the environment, certifying the value of the attribute of interest, and then maintaining this assertion cache and the attribute of interest as assertions fail.

We demonstrate the power of our approach through a series of examples exhibiting low-level behaviors including flocking and a multitude of continuously adaptive behaviors. The formulation lends itself naturally to parallelism and can be used as a basis for modeling high-level behaviors.

The paper is organized as follows. In Section 2 we list some of the related work in the field, and how they relate to our approach. In Section 3 we describe our basic framework while Sections 4, 5 and 6 describe each layer in detail. Section 7 explains the demos that are in the video and Section 8 contains the conclusions and future work.

2 Related Work

Modeling autonomous agents has been an active area of research across different fields, from AI to CG, where the work of Reynolds [29] first tried to simulate flocks of animals as agents. Since then, distinct methodologies have been employed to address this problem [7] and have been used in several domains, including game applications [35, 28]. AI techniques have shown very promising results, [5, 12], but they generally require complex inferencing mechanisms and explicit rules, making the explicit modeling of time very difficult. Typically, such methods do not scale well with the number of autonomous agents and obstacles, especially when every agent has a different set of governing rules. On the other hand, learning, perception and dynamics-based techniques [30, 34, 23, 19] can easily adapt to constantly changing environments.

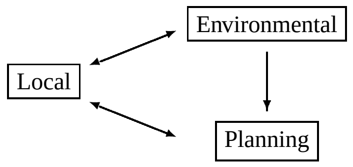
Some initial attempts to use nonlinear dynamical systems for low-level behavior modeling have appeared recently in [13, 14]. Finally, methods from computational geometry have also been employed [36, 16].

Different systems and approaches have been developed to add distinct behaviors to autonomous agents [34, 5, 6], while the usefulness of a layered approach to modeling low and high-level behaviors has been advocated by several researchers [21, 1, 26]. Behavioral robotics deals with similar issues, usually dealing with only two dimensional environments [2, 33, 22].

In this paper our novel integration of nonlinear dynamical systems, kinetic data structures and harmonic functions results in a layered, efficient and scalable approach to modeling low level behaviors, higher level planning and smart environment management.

3 Framework Design

We describe a three-layer framework for autonomous agent modeling and motion generation. Each layer is responsible for a different stage of the process, and has to communicate with the others. Different approaches can be taken for each layer depending on the particular application.



We term the three layers as *local*, *environmental* and *global*:

Local This layer is responsible for basic reactive actions: obstacle avoidance and moving in the desired direction. This stage needs to be fast, scalable, robust and have a parameterization used to control the particular way it does its basic tasks. For the system to be scalable this layer cannot depend on the total number of elements in the environment.

Global Environment This layer has knowledge about the spatial layout (e.g., agents, obstacles etc.) of all the objects in the environment. For each of the agents it maintains its immediate environment, which in turn is used to determine the agent’s behavior. It is possible to imagine a real-time application where each agent has a sensor system that can detect the presence of obstacles, other agents, etc. within a certain viewing frustum. Such information can then be transmitted to and shared by other agents, so that each agent can build a more complete model of its environment. This layer is also responsible for collision detection, and for notification to the agents involved in it, so that appropriate responses can be generated.

Global Planning Since the local layer only knows how to deal with very local operations, it is necessary to give the agents some intelligence, or at least a more general knowledge of their environment. This layer gives each agent the general direction it should follow, at each given point in time towards a target. An interpretation of the result of this layer is a time-varying adaptive flow field. This is used in the local layer as an educated guess of where the agent should try to go to achieve its goal.

In a cooperative situation this layer is responsible for organizing different agents to do different tasks in order to achieve the common goal. It needs to take into account each agent’s attributes.

This layer can also change the attributes of each agent, so as to achieve higher level behaviors. An agent can get “tired” as time goes by, moving slower or in a more clumsy way. If a critical task has to be achieved no matter the cost, an agent should allow itself to get closer to moving obstacles, increasing the risk of being involved in a collision.

4 Local system

We model every agent by a nonlinear dynamical system. The movement of the agent at any instant is defined by its heading angle and forward velocity. The agent’s behavior is modeled through a parameterization of the nonlinear system governing the agent’s movements.

We first present the modeling of the agent’s heading angle, which is defined through a pair of coupled nonlinear dynamical systems. Then we present the modeling of the agent’s forward velocity. Our methodology is an adaptation of nonlinear dynamical system theory first used in behavioral robotics [33, 22]. Intuitively, our method amounts to computing an agent's motion so that it always turns away from obstacles and moves in the direction of the targets. The agent's behavior is adaptive and intelligent, since it dynamically computes the varying contributions of the surrounding objects to determine its path towards a target.

4.1 Agent Heading Angle

The agent's heading angle, ϕ, is computed through a dynamical system of the form

$$\dot{\phi} = f(\phi, \mathbf{env}), \tag{1}$$

where f is a nonlinear function, and **env** represents the state of the environment, i.e, the position and size of obstacles, targets and other agents.

The function f is constructed so that each agent avoids obstacles and pursues the desired direction. The first step in the design of f is to model the contribution of obstacles and target direction. We model their contribution so that the direction of an obstacle will act as an angular *repeller*, while the goal direction will act as an angular *attractor*.

Angular attractors and repellers are located at the zero crossings of f (1). At these points an attractor or repeller property will be determined by the derivative of f with respect to ϕ. These points are called *critical points* or *fixed points*: since $\dot{\phi} = 0$, ϕ does not change at these points.

A fixed point behaves as an attractor or repeller depending on the value of the derivative of f with respect to ϕ, i.e., $\partial f/\partial\phi = \partial\dot{\phi}/\partial\phi$. If the derivative is positive, any ϕ slightly off of the critical point will deviate farther away with time. This is a repeller. The inverse situation occurs when the derivative is negative. Values of ϕ slightly away from the critical point will converge towards it. This is an attractor. Shiftings of these functions will place repellers and attractors to the specified positions of obstacles and goal direction respectively.

Figure 1 shows several parameters in a situation involving an agent and two obstacles. ψ_i is the angle of obstacle i with respect to the horizontal, $\Delta\psi_i$ is the angle subtended by obstacle i on the agent and r_i is the distance between obstacle i and the agent. We will use these parameters to define a repeller and an attractor function that correspond to the targets' and obstacles' contribution to f, f_{tar} and f_{obs}, respectively. The design of new

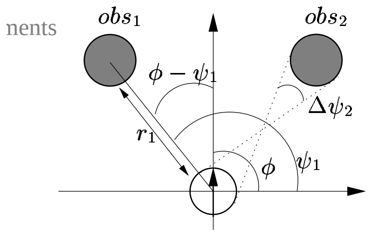


Figure 1: Two obstacles in front of an agent.

attractor/repeller functions can be a complex task, but once it is done it results in a very fast computer implementation – it only amounts to function evaluations. Figure 2 shows the plotting of two typical examples of these functions. All equations are in Appendix A. More details and design choices can be found in [14]. The definitions of the attractor f_{tar} (12) and repeller f_{obs} (13) functions are used to define the environment function f that takes

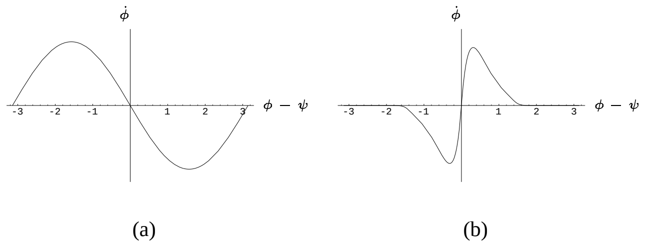


Figure 2: In 2(a), attractor function for a target in $\psi = 0$. In 2(b) an repeller with $\psi = 0$, $\Delta\psi = \pi/10$,$r_i = 3$, $d_0 = 6$ and $\delta = 0.8$.

into account both obstacles and goal direction. A simplistic approach would be to sum the obstacle and target contributions to obtain

$$\dot{\phi}(\phi, \mathbf{env}) = f = f_{tar} + f_{obs}. \tag{2}$$

Unfortunately this approach does not always produce the desired result. Undesirable behaviors may result due to the nonlinearity of the summed functions [22]. To overcome this problem we use an adaptive weighted sum of the target and obstacle contributions (3) instead of a direct sum.

4.1.1 Adaptive Weighting of Env. Contributions

The repeller functions, f_{obs}, are designed so that they will work as expected when there is more than one obstacle, but their direct sum with a target's function f_{tar} will not always produce the expected result. This case occurs, for example, when two obstacles are too close together to allow the agent to move between them, and a target is hidden behind them. We would expect that the agent will go around the obstacles. Instead, the resulting direct sum of these functions creates an undesired attractor just in front of the agent, leading eventually to a collision.

One way to avoid this problem is to introduce a mechanism for selecting the relative weights for target and obstacle contributions in the sum. We use a nonlinear dynamical system to produce a real-time computation of the necessary adaptive weight functions. These functions are used to determine a *weighted sum* of the target and obstacle functions.

Based on our application domain, we use a two dimensional weight space (obstacles and goal), but more classes can be used (allowing different types of obstacles to be treated differently, for example) depending on the application and the system's stability conditions (similar approaches are used in *Hopfield Neural Networks* [18]). Therefore the computation of the heading angle is based on a nonlinear system of the form

$$\dot{\phi} = f(\phi, \mathbf{env}) = |w_{tar}| f_{tar} + |w_{obs}| f_{obs} + n, \tag{3}$$

where w_{tar} and w_{obs} are the weights for the targets and obstacles, respectively, and n is a noise term used to escape from the unstable fixed points created by obstacles. These weights are the *fixed points* of the following nonlinear constraint competition dynamical system:

$$\begin{cases} \dot{w}_{tar} = \alpha_1 w_{tar}(1 - w_{tar}^2) - \gamma_{12} w_{tar} w_{obs}^2 + n \\ \dot{w}_{obs} = \alpha_2 w_{obs}(1 - w_{obs}^2) - \gamma_{21} w_{obs} w_{tar}^2 + n \end{cases}, \tag{4}$$

where $\alpha_1, \alpha_2, \gamma_{12}$ and γ_{21} are parameter functions to be designed. Based on our approach, at every iteration during the computation of $\dot{\phi}$ using (3), we compute w_{tar} and w_{obs} based on the computation of the fixed points (since $\dot{w}_{tar} = \dot{w}_{obs} = 0$, there are two nonlinear equations with two unknowns w_{tar} and w_{obs}) of the system defined by (4).

However, a system like (4) chosen arbitrarily might not converge to a fixed point. Therefore, a careful stability analysis is required to determine its convergence properties. The stability analysis is done by first identifying the positions of the *fixed points*, $\dot{\mathbf{w}} = \dot{w}_{tar}, \dot{w}_{obs} = \mathbf{0}$, and then for each fixed point check the conditions such that all the eigenvalues of the *Jacobian matrix* $(\partial \dot{w}_i / \partial w_j)$ at that point are less than zero. This is a classical nonlinear dynamical system stability analysis [25]. Based on this procedure we have computed that the system in (4) has four critical points: $(0,0)$, $(\pm 1, 0)$, $(0, \pm 1)$ and $(\pm A_1, \pm A_2)$, where

$$A_1 = \pm\sqrt{\frac{\alpha_2(\alpha_1 - \gamma_{12})}{\alpha_1\alpha_2 - \gamma_{12}\gamma_{21}}}, \qquad A_2 = \pm\sqrt{\frac{\alpha_1(\alpha_2 - \gamma_{21})}{\alpha_1\alpha_2 - \gamma_{12}\gamma_{21}}}. \quad (5)$$

Appendix A contains the expressions for α and γ. Full proofs and explanations can be found in [14].

4.2 Agent's Forward Velocity

Many different approaches can be used for modeling the forward velocity. A possible approach is to assign a constant value to the forward velocity. This approach has drawbacks in a real-time environment: if an obstacle is suddenly in front of the agent, there might not be enough time for the agent to change direction, and a collision may occur. A better approach is to have the agent move faster when there are no objects around and slower in a crowded area. The agent should also retreat when it is too close to an obstacle. An equation for the forward velocity that satisfies the above design criteria is the following

$$v = \frac{r_{min} - d_1}{t2c}, \qquad (6)$$

where r_{min} is the distance to the closest obstacle, d_1 the *safety distance* and $t2c$ is the *time to contact*. This method basically applies a *constant time to contact* approach (see [32] for similar approaches). Note that only obstacles in front of the agent should be considered for this calculation.

4.3 Extension to Three Dimensions

In three dimensions angular velocities are no longer scalars, but this does not change the general approach. Basically each contribution now is a vectorial angular velocity, these velocities can then be vectorially added to each other.

The contribution of an attractor or repeller is obtained through the normal 2D method in the plane that contains both the current direction the agent is facing and the direction to the given object. This scalar contribution is then converted in a fully 3D angular velocity by multiplying it by the normal of this plane ($p - pos \times v_{dir}$). We see this situation in Figure 3.

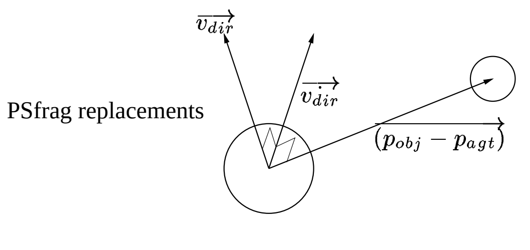

PSfrag replacements
$\overrightarrow{v_{dir}}$
$\overrightarrow{\dot{v}_{dir}}$
$\overrightarrow{(p_{obj} - p_{agt})}$

Figure 3: Obtaining the three dimensional angular velocity.

4.4 Modeling Low-Level Personality Attributes

The above dynamical system models every agent's movement in the same way. This can be a limitation when modeling more complex environments and behaviors. We add three scalar parameters (in the interval $[0, 1]$) that change significantly the way agents react to a same environment.

This approach gives each agent a certain degree of personality or uniqueness. It is then possible to create a simulation with a large number of agents, each one with its own individual characteristics. Each parameter of a group of agents can then be defined as a normal distribution with a given average and standard deviation. The group will act with some common characteristics but each agent will still have a particular way of doing things. This approach gives the simulation designer a great deal of flexibility. Finally, because these parameters only change the way numerical functions are evaluated, there is no further impact on the complexity of the system.

4.4.1 Angular Acceleration – Moment of Inertia

Intuitively this trait tries to model the agent's agility, i.e., how fast it can change direction. In our system this means imposing some kind of controllable (depending on the trait's value) limitation on the change in the heading angle.

We use this parameter to model how fast $\dot{\phi}$ can change. The above designed system (3) gives us a new value for $\dot{\phi}$ at each time instant, without any consideration of the previous history of the system. We need some kind of memory that will take into account the previous values of $\dot{\phi}$, combine it with the new proposed value, and give the final value $\dot{\phi}$.

A possible realization of the above idea is to use a low-pass filter for the value of $\dot{\phi}$ [24]. We need a very efficient solution that allows some control over the filtering. A larger amount of "dexterity" allows faster agent angle changes (larger band) while a smaller one limits the angle change (smaller band). We therefore choose to use a IIR (Infinite Impulse Response) filter, whose discrete implementation is shown in Figure 4. The equation describing this filter is:

$$\dot{\phi}(n\tau) = (1-k) * \dot{\phi}((n-1)\tau) + k * \tilde{\dot{\phi}}(n\tau), \qquad (7)$$

where $\tilde{\dot{\phi}}(t)$ is the output of (3) at time t, and $\dot{\phi}(t-\tau)$ is its previous evaluated value. The parameter k defines how fast is the response of the system to sudden variations. k is obtained as an affine transformation from the raw parameter. This way we can avoid the pathological case where $k = 0$ and the value of ϕ does not change. Notice that this model is extremely efficient, since it requires only a single addition and two multiplications.

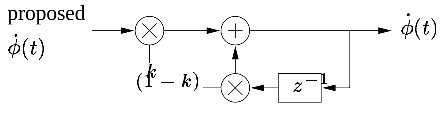


Figure 4: Two obstacles in front of the agent.

4.4.2 Forward Acceleration – Mass

We use the second parameter to model how fast the agent can change its forward velocity. This is a kind of imposition on the forward acceleration, and the problem is very similar to the one analyzed in the last section for the dexterity.

We model this contribution using the same kind of low-pass filter as for the dexterity trait, but now the forward velocity is modified instead of $\dot{\phi}$.

4.4.3 Aggressiveness – A subjective quantity

Intuitively, this parameter changes how the agent is willing to take risks, how close it is willing to get to the obstacles, and how fast it should move when close to them.

In the original system (3), the constant d_0 (equation 17 in Appendix A) controls the distance where obstacles start to be taken into account: larger values mean larger distances. Therefore, to model the willingness of the agent to get close to obstacles we use a mapping from courage to d_0, where large values of courage lead to small values of d_0 and small values of courage lead to large d_0. The mapping we use is

$$d_0(\text{co}) = e^{k(1-\text{co})} - 1.0, \tag{8}$$

where $k = 2\log(d_{0base} + 1)$. For small values of the parameter, it will result in large values for d_0. When the parameter is equal to zero, it will give d_{0base} and when it is 1, $d_0 = 0$. In our nonlinear dynamic system approach, the forward velocity also depends on the constant d_1 (6). The agent will even retreat when the distance to the closest obstacle is smaller that d_1. It is important then to reduce d_1 when this aggressiveness parameter increases. We model

$$d_1 = d_{0vel} - v_0, \tag{9}$$

where

$$d_{0vel} = \begin{cases} d_{0base} & \text{if co} < 0.5 \\ 2d_{0base}(1-\text{co}) & \text{if co} \geq 0.5. \end{cases} \tag{10}$$

5 Global Environment Control

We now describe our data structures for keeping track of the near environment of each agent. We model the near environment of an agent x by a disk C_x centered at the agent. We may wish to know what obstacles are near the agent, and what other agents, friendly or hostile, are close to x and visible to it, etc. This information can be used for calculating environmental contributions to the dynamics of x, for collision avoidance and detection, and so on. The challenge is here is how to keep this information up-to-date as the agents move and the environment changes. Though in principle this information can be recomputed from scratch at each time step, this is wasteful and would not allow our system to scale well to situations where large numbers of agents are interacting.

We have developed a new kinetic data structure tailored to this task. The KDS exploits continuity of motion and temporal coherence by focusing only on those relevant relationships in the environment that can be the ones to change next. Effectively, the KDS maintains a mathematical proof that the objects in C_x, the near environment of agent x, are the ones the KDS knows about. This proof is supported by a number of simple atomic relationships (such as distance comparisons); the set of objects forming the near environment of x cannot change unless one of those support relationships fails. The KDS tracks these atomic relationships, called the emphcertificates of the KDS, and, when one of them fails, it updates both the description of the near environment of x and the associated proof incrementally. A well-designed KDS for this problem attempts to come as close as possible to the goal of processing such certificate failures only when the near environment of x actually changes.

To make these general ideas concrete, consider the simple case where we model the agents as n moving points in the plane and there are no obstacles. As x moves, and C_x moves with it, we wish to track the set of other points that are inside C_x. We could do so by testing the distance of all other points from x at each time step; if we were to do this for a fraction of all the points, we have to deal with a $\Theta(n^2)$ update algorithm at each time step. Instead, our KDS solution proceeds as follows. We compute the well-known Delaunay triangulation of the n points; this structure is appropriate for our purposes, because it contains a lot of proximity information about the points. Furthermore, it is known how to maintain the Delaunay triangulation as the points move continuously, by simply doing certain edge-flip operations [17]. We have discovered the surprising fact that the only points that can enter or leave C_x are the endpoints of Delaunay edges crossed (exactly once) by the boundary of C_x. Thus, if we maintain the set of Delaunay edges crossed by C_x, we need to focus on a much smaller set of points when it comes to tracking points that may enter or exit C_x. This is depicted in Figure 5, where the endpoints of the solid edges are the only ones can make a transition next. The KDS certificates in this case include those certifying the Delaunay triangulation (these allow us to know when edge flips need to be done) and the ones asserting that a certain set of edges is crossed by C_x. Updating the Delaunay triangulation, the set of points in C_x, and the set of crossed edges at certificate failure times is straightforward. The details are given in appendix A.3. The appendix also discusses the extension of these ideas to handle obstacles, visibility conditions, and generalization to 3D. As compared with the naive solution to tracking the contents of C_x, the KDS solution has many advantages.

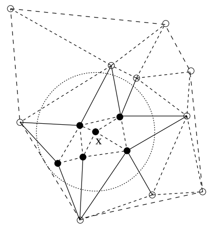


Figure 5: Tracking the agents that may enter or exit C_x.

- Conservative estimates for the failure times of the KDS certificates can be computed using motion estimates provided by the non-linear dynamical system; this means that most KDS certificates need not be verified at each time step by only when necessary.

- The number of other agents that can potentially enter or exit C_x has been reduced to the endpoints of Delaunay edges crossed by C_x. Though exactly how many these are depends on the radius of C_x, it is reasonable to assume that agents have a minimum size or separation, and that the radius of C_x is small compared to the size of the entire environment. These conditions imply that this set of interest has size which is a small constant, irrespective of the overall number of agents. Thus in general our update cost will be $O(1)$ per agent, or $O(n)$ overall.

6 Global Planning Layer

The global planning layer tells each agent, at each moment in time, what is the direction it should pursue to reach the goal. A simple solution for that is to establish the direction that points directly to the desired target as the correct course. In many applications this is a good and simple enough solution

The advantage of having an independent layer for doing this calculation is that, as long as it communicates the same way with the other layers, it is straightforward to replace it with a more sophisticated approach if needed.

Using local knowledge of the surroundings to determine how to reach the goal is not always enough. The agent will likely fail to reach the goal under trap situations like an U-shaped wall. In this example, using just the direction to the target, the agent enters the dead-end region until it decides that it has to turn off the target contribution. By then it turns around and leaves the U-shaped obstacle, but shortly after, it again takes the target into consideration, reestablishing the cycle. Every now and then, because of the noise inserted into the equations, the agent might escape this dead-end cycle — but there is no guarantee of how or when it would happen.

A more sofisticated way to determine what the local layer should use as the goal direction, a precomputed velocity field. The ability to use arbitrary velocity fields gives the designer of the environment/simulation a great flexibility to achieve effects that might not be normal or expected, but that are desired or necessary for a given application. An initial velocity field can also be obtained through automatic procedures, like the use of *harmonic functions* (i.e. also know as *potential functions*), to obtain a good initial global guess of how to avoid local traps.

The potential field construction is a computationally expensive task, and to do it in every integration step would rule out any real/interactive time application. What we do is precompute the potential field for only static obstacles known in advance. This is not a limitation, since main landmarks of the environment are usually already known. In a real terrain simulation it is absolutely reasonable to expect that the position of rivers, crevices and buildings are known in advance; in games and VR applications the "map" is already known, and many times there are even precomputed BSP-trees.

Moving obstacles, as well as the static ones, are taken into account by the steering and collision avoidance capabilities of the two other layers, and the direction of the velocity field is then fed into the local layer whenever the target is not in clear direct sight. This technique avoids local minima situations like the one described in the Section 6. There, the goal direction in the "U-Shaped" room would be towards the exit and not in the local direct line to the target.

Potential functions have already been used in robotic path planning applications [20]. Due to certain inherent properties such as the guaranteed absence of local minima, harmonic potential functions are particularly suitable when robust trajectory generation is important [9]. Harmonic potentials are also used in several physical domains, such as hydrodynamics and electrostatics, to analyze complex fluid flow situations and to compute electric potentials. Modeling the agent motion as a flow allows the use of the well-developed underlying mathematical theory and the associated powerful tools.

6.1 Harmonic Functions

Harmonic functions are used, among other things, to model the behavior of ideal (incompressible and non-viscous) fluids. They are solutions to the *Laplace equation*:

$$\nabla^2\lambda = \frac{\partial^2\lambda}{\partial x^2} + \frac{\partial^2\lambda}{\partial y^2} = 0, \qquad (11)$$

where, $\nabla^2 = \nabla \cdot \nabla$ is the *Laplacian* operator.

The path adopted by a fluid particle in a steady flow is called a *streamline*. A streamline is the integral curve of (11) and may be represented by the gradient of the harmonic potential. Streamlines may not intersect each other but external perturbations, or even integration errors, will cause a particle to jump from one streamline to another.

For our purposes, we only use the direction (and ignore the magnitude) of the local gradient vector as the nominal goal direction of our agent. Our boundary conditions in Laplace's equation are the target potential (the potential of the region covered by the target) and the obstacles' potential, which include the configuration space boundary. Typically, the target potential is set to a low value (the global minimum) and the obstacle potential is set to a high value. Since fluid tends to flow towards a decreasing potential, all the streamlines generated in this set-up will eventually converge to the goal point. The implementation details for the potential field can be found in the Appendix A.

7 Experiments

We demonstrate the power of our approach with a series of two and three dimensional experiments [1]. We demonstrate how changes in the intrinsic parameters of an agent can generate different behaviors, in situations with few agents in crowd simulations. All our experiments run in the range of 1-700 frames per second on a Pentium II 400MHz Linux workstation.

In the two dimensional flocking example we used a random set of intrinsic attributes for each of the agents. We show how some of the chosen attributes are not sufficient for the agents to reach the target. For example, there are two agents that sometimes get stuck behind obstacles. Their attributes allow them to get really close to the target but do not give them enough flexibility to steer away in time. The forward velocity component of the system is responsible to avoid actual collisions.

We use harmonic functions on the two maze-structured examples, and in the theater example in order to provide a global direction that guarantees the agents do not get stuck in dead-ends.

In the first pair of three dimensional examples we illustrate how the intrinsic parameters of the agents affect the general flocking property of a group.

8 Conclusions and Future Work

In this paper we have presented a three-layered system for single and multiple agents steering with unique behaviors in real time environments. The system uses nonlinear dynamical systems, together with harmonic functions and a specially designed kinetic data structure to navigate among static and moving obstacles, reach static and/or moving targets. Using our principled approach we have shown a multitude of examples including flocking simulations which are by themselves difficult and challenging areas. We have also shown a variety of low-level continuously adaptive behaviors generated through the personality parameters. Systems based on our approach will make it easier for users to achieve efficiently complex low-level agent behaviors and easily build new personality traits. Our approach can serve as a basis for modeling higher level behaviors, since it provides a continuum of complex low-level behaviors. The embedded controlling parameters allow time-changing agent behaviors. It is possible to simulate fatigue in plane pilots, adrenaline rush, a highly inertial vehicle, etc, all within the realm of the low-level layer.

The kinetic data structure (KDS) perfectly suits the environment layer. It keeps lots of information and deals with continuous time collision detection (in precise points between integration

[1]http://www.cis.upenn.edu/∼siome/research/nonlin/results

steps) all with no impact on the complexity. It allows a single computer implementation to simulate the agent perception of its local surroundings keeping the complexity in.

Interesting possibilities arise by the possible combination of AI and computational geometry techniques on the global planning layer of this system, where a centralized reasoning system can collect information from the environment and plan different strategies for each agent in order to achieve some common goal. The layered approach allows these algorithms to overlook low level actions, like obstacle avoidance, and concentrate on the more abstract tasks.

A Formulas and Equations

In this appendix we provide some equations and details omitted from the main text of this paper. These details, however, are still necessary for reproducing our work. Size contraints, unfortunately, do not allow us to fully explain all the theory — for a smooth overview the reader is invited to look at some of the references.

A.1 Non Linear Dynamical System Equations

A.1.1 Heading Attractor and Repeller

The model we use for the attractor function f_{tar} is defined as a sinusoidal shifted by ψ, whose argument is the agent's heading angle, i.e., the relative angle between the agent's direction and the object direction:

$$f_{tar} = -a\sin(\phi - \psi). \tag{12}$$

To define a repeller function, a simple sinusoidal function is not enough (see reference). Therefore, as a model for a repeller we use the multiplication of three different functions, R_i, D_i, W_i, each modeling a different property, i.e.,

$$f_{obs_i} = R_i\, W_i\, D_i. \tag{13}$$

Function R_i models a generic repelling property of an obstacle i, and is defined as:

$$R_i = \frac{(\phi-\psi_i)}{\Delta\psi_i} e^{\left(1-\frac{\phi-\psi_i}{\Delta\psi_i}\right)}, \tag{14}$$

where $\phi - \psi_i$ is the angle of obstacle i with respect to the agent's direction and $\Delta\psi_i$ is the subtended angle of the obstacle by the agent.

The second function, W_i, is responsible for limiting the angular range of the repeller's influence in the environment and is defined as

$$W_i = \frac{1}{2}[\tanh(h_1(\cos(\phi-\psi_i) - \cos(2\Delta\psi_i + \sigma))) + 1], \tag{15}$$

which models a window-shaped function. The variable h_1 is responsible for the inclination of the window's sides and is defined as

$$h_1 = 4/(\cos(2\Delta\psi) - \cos(2\Delta\psi + \delta)), x \tag{16}$$

where δ is a safety margin constant (in all our applications we kept $\delta = 0.8$).

The third function, D_i, models the influence of the i^{th} obstacle in the environment based on the distance of the obstacle from the agent. It is modeled as

$$D_i = e^{-\frac{r_i}{d_0}}, \tag{17}$$

where r_i is the relative distance between the obstacle and the agent, and d_0 controls the strength of this influence as the distance changes (exponential decay).

A.1.2 Adaptive Weighting of Contributions

In our application, the dynamical system in (4) has four critical points, their status (as stable or unstable) at each moment in the simulation will depend on the α and γ coefficients. They are carefully designed to achieve the desired properties, please see [14] for a full explanation of the procedure, as well as for some of the proofs..

$$P_{tar} = \text{sgn}\left(\frac{\partial f_{tar}}{\partial\phi}\right) e^{c_1|f_{tar}|}, \tag{18}$$

$$P_{obs} = \text{sgn}\left(\frac{\partial f_{obs}}{\partial\phi}\right) e^{c_1|f_{obs}|}\left(\sum_i W_i\right) \tag{19}$$

Using P_{tar} and P_{obs}, we can design the α and γ coefficients as follows:

$$\gamma_{12} = \frac{e^{-c_2 P_{tar} P_{obs}}}{e^{c_2}}, \qquad \gamma_{21} = 0.05, \tag{20}$$

$$\alpha_1 = 0.4(1-\alpha_2), \qquad \alpha_2 = \tanh\left(\sum_i D_i\right). \tag{21}$$

A.2 Potential Field Implementation Details

This section presents the computational scheme for solving (11) and is adapted from [31, 3, 27]. In a numerical scheme the configuration space is discretized and is replaced by a grid. We first set the target potential to 0 and the obstacle and boundary potential to 1. To initiate the iterations, the empty configuration space nodes are arbitrarily set to a value of 0.5. In the Gauss-Seidel [3] approach, the m^{th} iteration step for node points $k = 1, 2, \ldots, N-1$ and $j = 1, 2, \ldots, N-1$ in an $N \times N$ mesh is expressed as,

$$\lambda_h^{(m)}(x_j, y_k) = \frac{1}{4}[\lambda_h^{(m-1)}(x_{j+1}, y_k) + \lambda_h^{(m-1)}(x_j, y_{k+1}) + \lambda_h^{(m)}(x_{j-1}, y_k) + \lambda_h^{(m)}(x_j, y_{k-1})]. \tag{22}$$

whereas for the boundary point nodes,

$$\lambda_h^{(m)}(x_j, y_k) = 0 \text{ or } 1, \text{ for all } m \geq 0. \tag{23}$$

We terminate the iterations when the difference between the nodal values at the current and previous time steps is below a chosen threshold (in all our experiments we set it to 10^{-15}). The Gauss-Seidel method may be significantly accelerated by using the *successive over relaxation* or the SOR method [3].

A.3 Kinetic near neighbor maintenance

In this subsection we provide details about the kinetic data structure for maintaining the environment around each agent. Geometrically, the agents are modeled as moving points, obstacles (fixed or moving) as line segments, and the near environment as a ball of a certain radius around an agent. The key geometric structure we deploy is the Delaunay triangulation of the moving agents.

A.3.1 Nearest neighbors in 2D

Let $G(V, E)$ be a time-varying planar straight-line graph. The vertices of V model moving agents or obstacle boundaries, while the edges in E are the obstacles themselves. With each point p in a subset S of V we associate a disk C_p centered at the p that is of radius r_p. The set S represents the agents whose environments we desire to track, and the associated disks represent these near environments for each agent. Let T be a triangulation on V conforming to G (i.e., all edges in E are also in T). We call a point q

in V *approachable* from a point p in S if q is inside C_p and there exists a path from p to q in T that lies entirely in C_p. We say that an edge e of T *crosses* C_p if one endpoint of e lies outside of C_p and the other endpoint of e is approachable from p.

We consider two cases: the *unconstrained* case, when there are no obstacles present (the edge set E is empty), and the *constrained* case, with obstacles. In the unconstrained case, we maintain for each point p in S the set of points of V that are inside C_p. In the constrained case, we maintain the set of points that are inside C_p and which are not separated from p by an edge in E both of whose endpoints are outside C_p (these correspond to the agents near to and visible from the given agent). It turns out that, if T is the Constrained Delaunay Triangulation (CDT) [8], the set that we want to maintain for a point p in S is the set of points that are approachable from p. In particular, in the unconstrained case, every point inside C_p is approachable from p; in the constrained case points inside C_p that are not approachable from p are exactly those that are blocked by a constrained edge that with endpoints outside C_p.

The key observation that enables us to maintain the above mentioned sets is captured in the following lemma:

Lemma 1. *Let T be the CDT of G and let C_p be a circle centered at $p \in S$. If point $q \in V$ enters/exits the circle C_p at some time t_0 and is visible from at least one point inside C_p, then at t_0 there exists an edge of T between q and a point inside C_p.*

Proof. At time t_0, q is on the boundary of C_p. Let $\{C_r\}$ be the family of circles with center r that pass through q, where r is a point on the segment pq. Consider the circle $C_{r'}$ such that r' is at maximal distance from q, visible from q and the circle $C_{r'}$ contains no points of V. Note that because the set of points of V that are visible from q at t_0 is non-empty by assumption, such a circle $C_{r'}$ always exists. Clearly the edge qr' is a constrained Delaunay edge.□

This observation immediately provides a way to track the neighbors of every point p in S, if we maintain the CDT of S. For every p in S keep the set of approachable points A_p and the set of crossing edges E_p. If the combinatorial structure of T does not change, then the endpoints of these edges are the only points that can possibly enter or exit C_p. In other words the maintenance of the neighbors of the points in S consists of two parts: one is the maintenance of the triangulation itself, and the second is the maintenance of the triangulation edges crossed by the circles C_p.

The CDT can be maintained using standard edge-flip operations, just like the regular Delaunay triangulation [10, 17]. Whenever an edge flip happens, however, we also need to update the sets E_p, $p \in S$: when an edge in E_p, for some p, disappears we need to delete that edge from the edge set E_p, whereas when an edge that intersects the circle C_p, for some p, appears we need to add it to E_p.

The situation is somewhat more complicated if a point enters or exits the circle. In the case that a point q enters the circle C_p we have to look at q's neighbors. For those neighbors that are outside we only need to add the corresponding edges to E_p. For those that are inside and approachable we need to remove the corresponding edges from the edge set E_p. Finally for the neighbors that are inside but not approachable (this can only occur in the constrained case) we need to add them to the point set A_p and perform the same tests for their neighbors recursively. When a point q exits C_p the situation is entirely symmetric: for all the neighbors that are outside we delete the corresponding edges from E_p. For the neighbors that are inside and remain approachable after the point exits, we need to add the corresponding edges to the set E_p. Finally as far as the remaining neighbors are concerned, we have to delete them from the set A_p of approachable neighbors, as well as delete any edges in E_p that adjacent to them and recursively do the same for their neighbors.

A.3.2 Nearest neighbors in 3D

In three dimensions we have implemented only the case where there are no obstacles present. For every point $p \in S$ we now associate a sphere C_p of radius r_p. The definitions of *approachability* and *proper intersection* are the same as in the two-dimensional case.

Our goal is to maintain for each point $p \in S$ the set of points that are inside C_p. As in two dimensions, if T is the Delaunay triangulation, this set is the same as the set of A_p of approachable points, and Lemma 1 is still valid.

The three-dimensional Delaunay triangulation is maintained by performing face-edge of edge-face flips [11]. Hence, the 3D unconstrained case is essentially analogous to the 2D unconstrained case. While in a 2D flip always exactly one edge appears and one disappears, in a 3D flip the number of edges in the triangulation may increase or decrease by one.

References

[1] N. Magnenat-Thalman adn D. Thalmann. Digital actors for interactive television. *Proc. IEEE (Special Issue in Digital Television, Part 2)*, 83(7):1022–1031, July 1995.

[2] R. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Journal of Robotics and Autonomous Systems*, 6:105–122, 1990.

[3] K. E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley & Sons., New York, 1978.

[4] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

[5] C. Beardon and V. Ye. Using behavioral rules in animation. In *Computer Graphics: Developments in Virtual Environments*, pages 217–234. Academic Press, 1995.

[6] B. Blumberg. Go with the flow: Synthetic vision for autonomous animated creatures. In *Poster on AIII Conference on Autonomous Agents*, 1997.

[7] Mark Cavazza, Rae Earnshaw, Nadia Magnenat-Thalmann, and Daniel Thalmann. Survey: Motion control for virtual humans. *IEEE Computer Graphics and Applications*, 18(5):24–31, Sep/Oct 1998.

[8] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[9] C. I. Connolly. Application of harmonic functions to robotics. In *International Symposium on Intelligent Control*, 1992.

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[11] M. A. Facello. Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions. *Comput. Aided Geom. Design*, 12(4):349–370, June 1995.

[12] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. *Proc. of SIGGRAPH 99*, pages 29–38, 1999.

[13] S. Goldenstein, E. Large, and D. Metaxas. Dynamic autonomous agents: Game applications. In *Proceedings of Computer Animation 98*, June 1998.

[14] S. Goldenstein, E. Large, and D. Metaxas. Non-linear dynamical system apprach to behavior modeling. *The Visual Computer*, 15:349–369, 1999.

[15] L. J. Guibas. Kinetic data structures — a state of the art report. In *Proc. 3rd Workshop on Algorithmic Foundations of Robotics (WAFR)*, pages 191–209, 1998.

[16] L. J. Guibas, M. Sharir, and S. Sifrony. On the general motion planning problem with two degrees of freedom. In *Proc.* 4^{th} *ACM Symp. Computational Geometry*, pages 289–298. ACM Press, 1988.

[17] Leonidas J. Guibas, Joseph S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In G. Schmidt and R. Berghammer, editors, *Proc. 17th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, volume 570 of *Lecture Notes Comput. Sci.*, pages 113–125. Springer-Verlag, 1991.

[18] Simon S. Haykin. *Neural Networks : A Comprehensive Foundation*. Prentice Hall, 1998.

[19] Jessica K. Hodgins and Nancy S. Pollard. Adapting simulated behaviors for new characters. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 153–162, 1997.

[20] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 500–505, March 1985.

[21] D. Kurlander and D. T. Ling. Planning-based control of interface animation. In *Proc CHI95 Conf.*, pages 472–479. ACM Press, 1995.

[22] E. Large, H. Christensen, and R. Bajcsy. Scaling the dynamic approach to path planning and control: Competition among behavioral constraints. *International Journal of Robotics Research*, 18(1):37–58, 1999.

[23] H. Noser, O. Renault, D. Thalmann, and N. Thalmann. Navigation for digital actors based on synthetic vision, memory and learning. *Computer and Graphics*, 1995.

[24] Alan Oppenheim, Alan Willsky, and Ian Young. *Signal and Systems*. Prentice-Hall, 1983.

[25] L. Perko. *Differential Equations and Dynamical Systems*. Number ISBN-0387974431 in Texts in Applied Mathematics. Springer Verlag, Berlin, February 1991.

[26] Ken Perlin and Athomas Goldberg. IMPROV: A system for scripting interactive actors in virtual worlds. In *SIGGRAPH 96 Conference Proceedings*, pages 205–216. ACM SIGGRAPH, 1996.

[27] W. H. Press, S. A. Teukolsky, T. V. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 1992.

[28] C. Reinolds. Steering behaviors for autonomous characters. In *Proc. of Game Developers Conference*, 1999.

[29] C. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Proc. SIGGRAPH '87*, volume 21, pages 25–34, 1987.

[30] G. Ridsdale. Connectionist modelling of skill dynamics. *Journal of Visualization and Computer Animation*, 1(2):66–72, 1990.

[31] K. Sato. Deadlock-free motion planning using the laplace potential field. *Advanced Robotics*, 7(5):449–461, 1993.

[32] G. Schöner, M. Dose, and C. Engels. Dynamics of behaviour: theory and applications for autonomous robot architectures. *Robotics and Autonomous Systems*, 16(2–4):213 – 246, 1996.

[33] A. Steinhage and G. Schöner. The dynamic approach to autonomous robot navigation. In *Proceedings IEEE International Symposium on Industrial Electronics*, 1997.

[34] X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proc. of SIGGRAPH '94*, pages 43–50, 1994.

[35] Swen Vinckle. Real-time pathfinding for multiple objects. *Game Developer*, June 1997.

[36] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proc.* 4^{th} *ACM Symp. Comp. Geometry*, pages 179–288, 1988.