Lightweight Simulation Scripting with Proto

Jacob Beal
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: jakebeal@bbn.com

Kyle Usbeck
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: kusbeck@bbn.com

Brian Krisler
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: bkrisler@bbn.com

***Abstract*—Modern game engines make it easy to create complex realistic environments for entertainment or for training, but scripting the behavior of agents in these environments is still a major challenge. Spatial computing languages such as Proto [1] provide a possible solution, but need to be adapted for practical scripting use. We have begun to address this problem by linking Proto with the Unity game engine and by creating a Proto library for scripting the behavior of groups of agents. We validate our approach by demonstrating compact scripting of three complex agent interaction scenarios.**

I. Introduction

Modern game engines [2], provide the core components necessary to quickly produce simulations that just a few years ago required complex, custom solutions. The proliferation of these generic engines has led to the emergence of a new category of games, referred to as serious games [3]. The main focus area for serious games is training, where systems such as the US Navy VESSEL trainer [4] are used to reduce classroom lecture times and promote active learning.

Every game engine has a scripting environment that provides a language and core API for customizing interactions with and within the game. While these APIs are typically robust and allow for complete control of all objects within the game world, they are seriously limited in their support for quickly scripting behaviors for large groups of autonomous agents. For example, in the creation of a training game where a trainee would have to function in a large crowd, providing the movement flow and heterogeneous interactions typical of a realistic crowd would require many complex pieces of custom code, perhaps down to the level of individual agents. This requirement limits the inclusion of many autonomous agents in a training scenario.

In this paper, we address this problem by linking the Proto spatial computing language to Unity [5], a widely used modern game engine. We then create a library for scripting the behavior of groups of agents and demonstrate how our approach allows compact scripting for large groups of agents in a realistic simulation environment.

II. Background

Although there has been much previous work on agent behavior programming and simulation, there are significant gaps in the capabilities of existing solutions. These current solutions can be classified into three categories: single-agent behavioral models, multi-agent toolkits, and spatial computing platforms. A detailed review discussing many of the approaches described in this section can be found in [6].

Many game engines simply use conventional programming languages (or their own domain-specific variants) for their scripting languages. For example, Unity uses scripts written in JavaScript, C#, and Boo. More sophisticated single-agent behavioral models include conceptual models of agent behavior and agent frameworks (for implementing agent behaviors). Conceptual behavior models, such as the Belief-Desire-Intent (BDI) agent model [7], offer high-level descriptions of agent internals. Agent frameworks (often called "agent architectures") described thoroughly in [8] and [9], provide tools (e.g., agent administration, messaging, mobility, logging, etc.) for implementing agent behaviors. These frameworks and behavioral models, however, rarely provide aggregate programming/modeling tools that are useful for MAS control.

Multi-agent System (MAS) modeling and simulation toolkits tend to focus on interactions: both inter-agent interactions and interactions between agents and their environment. MAS modeling and simulation toolkits include languages for modeling the MAS, and tools for simulating the running agent system. For example, NetLogo [10] extends the Logo language to allow agent coordination and provides a graphical tool for simulating the agent behaviors. Most existing MAS modeling and simulation toolkits lack realism in their simulation environments, and therefore do not provide language features for realistic behavior. Furthermore, most MAS toolkits lack features for spatial aggregate programming, which we describe next, which enable scalable descriptions of aggregate behavior (i.e., the number of agents don't need to be specified *a priori*).

A more-recent approach is *spatial computing*, which assumes communication is constrained to agents near one another in space. The implication of this assumption is that it becomes necessary to consider the spatial structure of the system in planning the solution. Proto [1] is a purely-functional LISP-like language that is designed with spatial constructs (i.e., operations to measure and manipulate space-time, compute spatial patterns, and evolve dynamically). General purpose spatial languages such as Proto or MGS [11] are capable of elegantly and concisely describing aggregate MAS behavior (sometimes labeled "emergent behavior") [12], but often have unusual programming models. For example, Proto is a purely

Work partially sponsored by DARPA; the views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

functional language and does not offer the imperative-style MAS scripting that is familiar to game-based agent behavior developers. Furthermore, the simulators used for running and testing spatial languages tend to lack the realism that is available from recent physics simulators and game engines.

III. Approach

Our approach for creating a scalable aggregate scripting language for realistic simulation environment has three components: *(1)* the Unity game engine [5] provides realism in the simulation environment (e.g., terrain modeling, realistic physics simulation, entity modeling), *(2)* the Proto spatial computing language provides constructs for scalable aggregate programming, and *(3)* agent behavior scripting is facilitated by a novel Proto library comprising group behavior primitives and a novel macro library for imperative-style scripting.

A. Connecting Proto and Unity

Proto has three main components: (1) the spatial language, (2) a global-to-local compiler which accepts a global behavior description (in Proto language) and outputs a virtual machine (VM) binary for the Proto VM, and (3) the VM that interprets Proto VM instructions on each device. In order to be able to execute global Proto programs from within the realistic Unity simulation environment, we first make the Proto compiler invokable from within Unity. Next, we create an implementation of the Proto VM for reading information from and performing operations upon Unity agents. Finally, we create an interface for developers to control parameters of the Proto-Unity plugin.

1) Invoking the Proto Compiler: Proto uses a global-to-local compiler to convert global behavior descriptions into local (i.e., per-device) programs. It is important that this compiler be integrated into the final solution so that end-users can write programs for groups of agents within the modeling and simulation toolkit.

The reference implementation of the Proto compiler is written in C++ and Unity has a C/C++ API for its plugins, so one option would be to directly integrate the Proto compiler into a Unity plugin. This would have required maintaining a branch of the Proto compiler with a Unity-friendly interface, however. Instead, we created a simpler plugin that invokes an external installation of the standard Proto compiler. Thus, Unity, supplied with a Proto program, invokes the external Proto compiler on that program and receives in return the Proto VM instructions (a.k.a., Proto opcodes) that specify how each agent should act.

2) A Proto VM Implementation for Unity: Next, we need a mechanism for controlling agents within Unity according to the behaviors described by the local Proto VM instructions. The Proto reference implementation already contains a VM suitable for most environments—requiring the developer to implement only a small set of platform-specific functions (e.g., how the machine allocates memory, broadcasts messages to other devices, etc.). Likewise, the continuous time model of Proto programs means there is no problem matching simulation rates: the VM execution rate can simply be derived from its Unity environment. Importing the Proto VM to Unity was not significantly different or more difficult than prior imports on various embedded platforms: to do so, we constructed a Unity plugin that implements the required platform-specific functions using tools from the Unity API. For example, one such function uses the Unity utility for computing the distance between Unity agents to implement a unit-disc communication model. Of course, this model could be extended to incorporate other information available from Unity (e.g., line-of-sight) for improved realism.

3) An Interface to the Proto-Unity Plugin: Finally, we created an interface for controlling the Proto-Unity plugin. This engineering interface is not meant for end-users, but instead is designed to help agent script developers by providing functionality similar to that of the reference implementation of the Proto simulator. For example, the interface can show the network topology of the Unity agents by drawing lines between the agents within communication range and allows the developer to change the devices' communication radii on-the-fly. This allows a developer to "tweak" simulation parameters during development, then set them and remove the interface when the simulation is finished and provided to users.

B. Group Behavior Primitives

We next need a library of "primitives" for group behaviors—simple ways of describing what we want a collection of agents to do. These will be the basis for the agent scripts that we build. We build this library after the fashion of [13], as Proto functions that compute vector fields for the desired motion of agents. We can then produce complex behaviors by mixing these vector fields together in various ways.

We have created an initial library of eight behaviors. Figure 1 shows examples of these behaviors being applied to agents in Unity using the Proto/Unity bridge. Here we present only the API for the behaviors; their implementation is similar (or in some cases identical) to code presented in [13].

(`random-walk`)		
Parameter	*Type*	*Description*
RETURN	`TUPLE`	Vector direction for agent to move

Random-walk moves each agent in a random direction. This is like `brownian` in [13], except that speed is also randomized.

(`flock` `DIRECTION`)		
Parameter	*Type*	*Description*
`DIRECTION`	`TUPLE`	Preferred direction to flock toward
RETURN	`TUPLE`	Vector direction for agent to move

The `flock` behavior, also from [13], moves agent groups by repelling the closest agents, aligning with moderately-proximate agents, and weakly attracting distant agents. This allows a group of agents to "flock" together in partially-coherent group motion. The `DIRECTION` argument guides the motion of the flock with a preferred direction supplied to some or all members, as investigated in [14].

(`flock-to` `LOCATION`)		
Parameter	*Type*	*Description*
`LOCATION`	`TUPLE`	Coordinates to flock to
RETURN	`TUPLE`	Vector direction for agent to move

(a) Random-Walk (b) Flock/Flock-To (c) Disperse/Scatter (d) Toward (e) Cluster-By

Fig. 1. Examples of agents being controlled by group behavior primitives written in Proto.

The `flock-to` behavior is like `flock`, except the agents move coherently to a location, rather than toward a direction.

(disperse)		
Parameter	*Type*	*Description*
RETURN	TUPLE	Vector direction for agent to move

The `disperse` behavior, our last adaptation from [13], repels agents away from one another with a force proportional to the inverse square of the distance separating them.

(scatter DIRECTION)		
Parameter	*Type*	*Description*
DIRECTION	TUPLE	Vector biasing scatter direction
RETURN	TUPLE	Vector direction for agent to move

The `scatter` behavior is much like `disperse`, except that agents do not slow down when they start getting far apart and they have a directional bias, `DIRECTION`.

(toward TARGET)		
Parameter	*Type*	*Description*
TARGET	BOOLEAN	Boolean indicator that is `true` if an agent is a target
RETURN	TUPLE	Vector direction for agent to move

The `toward` behavior finds the direction toward the mean location of all neighbors with a `TARGET` property.

(away-from TARGET)		
Parameter	*Type*	*Description*
TARGET	BOOLEAN	Boolean indicator that is `true` if an agent is a target
RETURN	TUPLE	Vector direction for agent to move

The `away-from` behavior is the inverse of `toward`.

(cluster-by GROUP-ID)		
Parameter	*Type*	*Description*
GROUP-ID	INTEGER	Identifier for an agent group
RETURN	TUPLE	Vector direction for agent to move

Finally, `cluster-by`, sorts agents into groups: all agents repel each other weakly and are strongly attracted to others with the same `GROUP-ID` identifier. This will tend to separate the group into clusters by identifier, though if the agents are widely scattered, there may be more than one cluster for any given identifier.

C. Agent Scripting Library

The last ingredient needed is a means of composing together these group behavior primitives to form useful agent behavior scripts, which will typically be much more complicated. Proto's native approach is one of purely functional composition—mathematically elegant, but not well suited for the way that simulation designers often like to think. Instead, we would like to be able to talk about a script in terms of concepts like particular groups being assigned particular behaviors, responding to triggers, or progressing through a planned sequence one stage at a time.

Technically, Proto's functional model can already provide all of these capabilities. The problem is that the code to do so is often awkward and does not "look" like the kind of state-based programming that is more familiar for this sort of scripting. Fortunately, Proto has recently been extended with a capability for syntactic macros. We use this macro programming facility to create new syntactic constructs suitable for agent scripting. The macros transform these new syntactic constructs into implementation in terms of standard Proto primitives.

Our initial agent scripting library comprises five constructs, selected as examples of group and individual behavior selection and sequencing; other important categories not yet included are behavior planning, collective decision making, etc. In our initial library, `group-case` and `where` assign behaviors to groups of agents, `on-trigger` sets up a triggered action for a group of agents, `priority-list` assigns behavior based on the relative importance of competing priorities, and `sequence` moves a group of agents through a planned sequence of actions. All of these are defined with the assumption that the return value is intended to be a vector field specifying the movement of agents. We will now detail each of these constructs in turn, then demonstrate their use with the examples in the next section.

The syntax of the `group-case` construct is:

```
(group-case
 (behavior-of MEMBERSHIP-TEST BEHAVIOR
 (behavior-of MEMBERSHIP-TEST BEHAVIOR
  ...
 (default BEHAVIOR)...)))
```

This operates much like an ordinary case statement: each `MEMBERSHIP-TEST` must be a boolean-value expression, and agents use the `BEHAVIOR` of the first `behavior-of` case they match. If an agent is not a member of any group, then it uses the `default` group's `BEHAVIOR`.

Within each `behavior-of` construct, there is a special variable `in-group` defined. Computations for a group's behavior normally extend over both agents in the group and agents outside of the group, allowing agents to react to information from others outside of their group. The `in-group` variable is true only for those agents in the group, and can thus be used to restrict computation to only within the group.

The `where` construct is a good way to do such a restriction:

```
(where TEST BEHAVIOR)
```

This computes BEHAVIOR over the set of agents where TEST is true, much like the standard Proto if construct, except that all other devices default to a tuple of zeros.

The on-trigger construct has identical syntactic structure:

```
(on-trigger TRIGGER BEHAVIOR)
```

Its function, however, is to enable a BEHAVIOR in a dormant group of agents as soon as TRIGGER becomes true for at least one member of the group. Once enabled, the group stays enabled and continues to act.

The syntax of the priority-list construct is:

```
(priority-list
 (priority NAME TEST BEHAVIOR
 (priority NAME TEST BEHAVIOR
 ...)))
```

Each agent walks the list of priorities in descending order, treating the first entry as highest priority. When a TEST evalutes to true, the agent executes the associated BEHAVIOR. If no priority holds, then the agent does nothing.

Finally, the sequence construct, which moves agents through a sequence of actions over time, is:

```
(sequence
 ([stage|group-stage] NAME ACTION TERMINATION
 ([stage|group-stage] NAME ACTION TERMINATION
 ...
 [end-sequence|repeat])...))
```

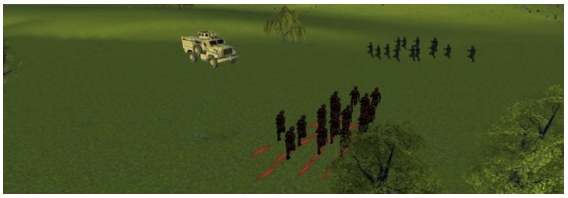
Agents transition individually through stage constructs and transition collectively out of group-stage constructs. The sequence begins with the first stage, executing ACTION until the TERMINATION condition is met. When an agent finishes a stage, it just moves on and begins executing the next stage's ACTION. For a group-stage, on the other hand, the agent also informs all neighboring agents, which move on to the next stage and inform their neighbors as well, and so on until all agents with reach of communication have changed stages. Thus, a group-stage terminates when *any* agent in the group reaches its TERMINATION condition.

When agents reach the final action in the sequence, their behavior depends on the final keyword. If the keyword is end-sequence, the agents stop moving; if it is repeat, the sequence begins again. Optionally, the keyword ongoing may be substituted for the last stage's TERMINATION, in which case the last stage continues indefinitely instead.

Although these five constructs are just a beginning of the type of constructs that are necessary to make up a full-fledged agent scripting library, they demonstrate that Proto macros can allow more "natural" scripting for agent behaviors.

IV. Validation

We now have an agent scripting library written in Proto and the ability to execute Proto programs in Unity—all of the ingredients necessary for validating our approach to lightweight simulation scripting. In this section, we demonstrate the power of our approach by constructing three simulations where groups of agents need to interact and to coordinate their behaviors with one another.



(a) Red team advances on Blue team

(b) Blue team notices incoming Red team

(c) Blue team scatters

Fig. 2. The red-advance script running on 30 agents.

For these simulations, we consider environments with two teams of agents: "Red team" aggressors and "Blue team" defenders. In each simulation, agents from both teams are placed onto a geo-typical terrain where they can then execute their group behaviors within the physics and terrain based constraints of the environment. We run these simulations with 10 to 30 agents; since the code is written in Proto, however, the same simulations can be executed on any number of agents.

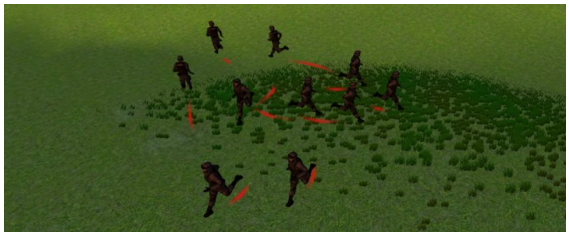
A. Red Advances on Blue

We begin with a simple scenario where Red team advances on Blue team and Blue scatters and flees when Red gets close:

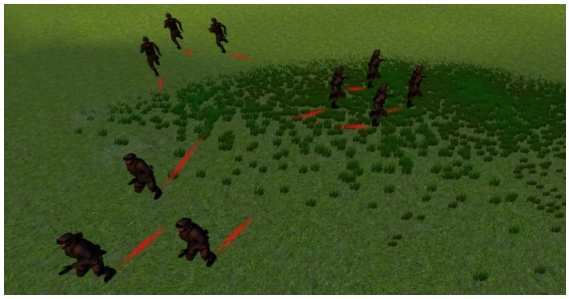
```
(def red-advance (red-team blue-team)
  (group-case
   (behavior-of red-team       ;; Red team behavior:
    (where in-group
     (flock-to (tup 0 0)))     ;; go to Blue starting location
   (behavior-of blue-team      ;; Blue team behavior:
    (on-trigger (can-see red-team) ;; when Red is near...
     (scatter (away-from red-team))) ;; flee from Red!
   (default (tup 0 0))))))
```

We do this by using the group-case construct to specify behavior by team. For Red team, we use flock-to to advance coherently towards the starting location of Blue team. For Blue team, we use on-trigger to scatter when any Blue agent notices an advancing Red, using the bias argument to making sure that the Blue agents move away-from Red. If any agent is not on either Red or Blue team, it does nothing. Figure 2 shows agents executing this scenario in Unity.

(a) Team moving as a group



(b) Team breaking up into three sub-groups



(c) Sub-groups moving to different destinations

Fig. 3. The `deploy` script running on 10 agents.

B. Red Deploys from a Vehicle

The next scenario has Red team deploying out of an armored transport vehicle into three squads:

```
(def deploy (squadID)
  (sequence
   (stage leave-vehicle          ;; First stage:
    (flock (tup -1 0 0))           ;; move left...
    (timeout 20)                   ;; ... for twenty seconds.
   (stage group-by-squad         ;; Second stage:
    (cluster-by squadID)           ;; group into squads...
    (timeout 50)                   ;; ... for fifty seconds.
   (stage deploy-to-destination ;; Third stage:
    (group-case        ;; Each squad goes to a different location:
     (behavior-of (= squadID 0)    ;; First squad ...
      (flock-to (tup 50 100))      ;; ... goes to (50, 100)
     (behavior-of (= squadID 1)    ;; Second squad ...
      (flock-to (tup -200 0))      ;; ... goes to (-200, 0)
     (behavior-of (= squadID 2)    ;; Third squad ...
      (flock-to (tup -100 -100))   ;; ... goes to (-100, -100)
     (default (tup 0 0))))))
    ongoing                  ;; Sequence doesn't end or repeat
   end-sequence)))))
```

Here, we use the `sequence` construct to break the deployment into three phases. First, the agents all `flock` for 20 seconds to leave the vehicle together. Next, the agents use `cluster-by` to sort themselves out into squads, giving 50 seconds for the squads to organize themselves. Finally, we use `group-case` to have each of the three squads `flock-to` its own destination. Figure 3 shows agents executing this scenario in Unity.

(a) Blue team on patrol — looking for Red team.

(b) Blue team members break off to chase Red team.

Fig. 4. The `patrol-encounter` script running on 30 agents.

C. Red Tries to Sneak Past a Blue Patrol

Our third scenario is the most complex: Blue team is trying to defend against Red team while patrolling a regular pattern. Meanwhile, Red team is trying to pass through the area that Blue team is guarding without being caught.

We first define the patrol pattern to be used by Blue team:

```
(def patrol ()
  (sequence
   (group-stage checkpoint-1 ;; First stage:
    (flock-to (tup 100 50))       ;; Go toward (100, 50) ...
    ;; ... until somebody in the patrol is within 5 meters of the place ...
    (< (vlen (- (coord) (tup 100 50))) 5)
   (group-stage checkpoint-2 ;; Second stage:
    (flock-to (tup 0 50))         ;; ... now go to (0, 50) ...
    (< (vlen (- (coord) (tup 0 50))) 5)
   (group-stage checkpoint-3 ;; Third stage:
    (flock-to (tup 50 -50))       ;; ... and then to (50, -50) ...
    (< (vlen (- (coord) (tup 50 -50))) 5)
   repeat)))))                    ;; ... and repeat
```

Here, we use a repeating `sequence` construct to define a cyclic patrol around three checkpoints. For each checkpoint, the agents use `flock-to` to move as a group to that checkpoint. Once any agent in the group reaches the checkpoint, the `group-stage` construct means its information will spread, causing the whole group to head for the next checkpoint even if some members have not yet reached the current checkpoint.

We then use this script as a behavior in the scripts for the overall encounter between Red team and Blue team:

```
(def patrol-encounter (red-team blue-team)
  (group-case
   (behavior-of red-team ;; Red team behaviors
    (priority-list
     (priority defend-self
      (can-see blue-team)              ;; if Blue shows up...
      (scatter (away-from blue-team));; ... then run away
     (priority invade                  ;; otherwise,
      (timeout 500)                ;; when the script says start
      (flock-to (tup 200 0))))) ;; try to pass by Blue team
```

```
(behavior-of blue-team ;; Blue team behaviors
 (priority-list
  (priority attack-red
   (can-see red-team)            ;; if Red shows up...
   (let ((dir (toward red-team))) ;; ... then track ...
     (where in-group (flock dir)));; ... and chase them
  (default                       ;; otherwise,
   (where in-group (patrol)))));; walk your patrol route.

(default (tup 0 0))))))
```

As before, we use `group-case` to specify a behavior for each team. We now also use `priority-list` to give each team multiple possible behaviors, depending on circumstances. Red team begins to invade the area 500 seconds after the simulation starts, attempting to `flock-to` its target. If a Red team agent encounters Blue team, however, this goal will be pre-empted by the goal of defending itself, and it will `scatter`, fleeing `away-from` Blue team.

The Blue team has a complementary `priority-list` script: when there are no Red team agents nearby, they default to patrolling on the three-checkpoint pattern that we defined above. If a Red team agent is nearby, though, they will break off to attack, using `flock` to move toward nearby red-team agents. Figure 4 shows agents executing this scenario in Unity.

Taken together, these demonstrate the power of our approach to scripting of agent-based simulations. Unity provides realistic physics simulation, while Proto and our new agent scripting library allow for compact scripting of scenarios in which groups of agents engage in many types of interactions.

The scenarios presented are remarkably compact in code, requiring only 8 lines, 19 lines, and 31 lines respectively. The same scenarios written in a conventional scripting language would typically take at least an order of magnitude more code. We can measure this in some cases by comparing against similar scripts available on the Unity community site, `http://www.unifycommunity.com/`. For example, the Proto `flock` code presented above takes only 12 lines, while a Unity JavaScript version takes 77 lines, yet must "cheat" in its calculations and can only run a single flock. Similarly, a single-agent waypoint following Program in Unity is implemented with 65 lines of JavaScript code, while it takes only 7 lines of Proto code to script coherent waypoint following for groups of agents. Even single agent functions are often much simpler in Proto: a Unity C# script for a wandering agent requires 40 lines, while the equivalent Proto function `dither` (part of the standard Proto library) requires only 5 lines.

Also of note is the relatively light computational burden of Proto; in the scenarios presented, the limiting factor on Unity simulation speed appears to be the cost of rendering the agents, with the cost of Proto computation and communication insignificant.

While not yet a definitive study, these results do clearly indicate that it is reasonable to expect large benefits from scripting agent-based simulations in Proto. We believe such a drastic reduction in size is likely to be due to two factors. First, as a functional programming language, Proto tends to produce more compact code. More importantly, however, Proto's ability to program aggregates and to execute routines on subgroups makes it just as easy to script a group behavior as a behavior for a single individual. The relative importance of these two factors, however, is not yet established.

V. Contributions

We have presented a novel approach to construction of agent-based simulation, based on the integration of the Unity simulation engine with the Proto spatial computing programming language. We have developed a library of agent group behaviors and scripting constructs aimed at programming this environment, and have demonstrated that the combination allows succinct specification of complex simulation scenarios with large numbers of interacting agents.

While the results presented in this paper demonstrate the potential for major improvements in agent-based simulation programming, there is much more that can be accomplished. Future work includes better integration between Proto and Unity, refinement and extension of the behavior library and scripting constructs, and construction of virtual sensors to allow Proto-controlled agents access to more information available from the Unity simulator, such as terrain properties, physical contact and line-of-sight.

References

[1] J. Beal and J. Bachrach, "Infrastructure for engineered emergence on sensor/actuator networks," *IEEE Intelligent Systems*, 2006.

[2] M. Lewis, J. Jacobson, and C. based Games, "Game engines in scientific research," 2002.

[3] T. Susi, M. Johannesson, and P. Backlund, "Serious games – an overview," 2007.

[4] T. Hussain, B. Roberts, C. Bowers, J. Cannon-Bowers, E. Menaker, S. Coleman, C. Murphy, K. Pounds, A. Koenig, R. Wainess, and J. Lee, "Designing and developing effective training games for the US Navy," in *2009 Interservice/Industry Training, Simulation and Education Conference.*, 2009.

[5] "Unity — 3D game engine," Available: `http://unity3d.com/`, Retrieved March 4, 2012.

[6] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," *CoRR*, vol. abs/1202.5509, 2012.

[7] A. Rao and M. Georgeff, "BDI agents: From theory to practice," in *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, 1995, pp. 312–319.

[8] W. C. Regli, I. Mayk, C. J. Dugan, J. B. Kopena, R. N. Lass, P. J. Modi, W. M. Mongan, J. K. Salvage, and E. A. Sultanik, "Development and specification of a reference model for agent-based systems," *Trans. Sys. Man Cyber Part C*, vol. 39, pp. 572–596, September 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1656816.1656823

[9] D. N. Nguyen, K. Usbeck, W. M. Mongan, C. T. Cannon, R. N. Lass, J. Salvage, and W. C. Regli, "A methodology for developing an agent systems reference architecture," in *11th International Workshop on Agent-oriented Software Engineering*, Toronto, ON, May 2010.

[10] E. Sklar, "Netlogo, a multi-agent simulation environment," *Artificial life*, vol. 13, no. 3, pp. 303–311, 2007.

[11] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," Univerite d'Evry, LaMI, Tech. Rep. 103-2004, 2004.

[12] K. Usbeck and J. Beal, "An agent framework for agent societies," *Systems, Programming, Languages and Applications: Software for Humanity*, 2011.

[13] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous space programs for robotic swarms," *Neural Computing and Applications*, vol. 19, no. 6, pp. 825–847, 2010.

[14] I. Couzin, J. Krause, N. Franks, and S. Levin, "Effective leadership and decision-making in animal groups on the move," *Nature*, vol. 433, no. 7025, pp. 513–516, 2005.