Battlecity Revived: Game Design with BDI.net

Yifan Li, Petr Musilek, and Lukasz Kurgan
Department of Electrical and Computer Engineering, W5-020 ECERF
University of Alberta, Edmonton, Alberta, T6G 2V4, Canada
yifan@ee.ualberta.ca

Abstract
Can interactive entertainment industry benefit from intelligent agents? The answer is: quite possible. This paper strives to demonstrate this possibility through a sample arcade style game. Modern computer games are, without doubt, complex software systems, and they have grown more and more reliant on AI techniques in order to better entertain the customers. These two properties make agent-oriented approach an ideal candidate for game development as it addresses both the complexity issue and the intelligence issue.

Keywords
Agent, BDI, Game

1. Introduction
The exact meaning of the term *agent* has never been unanimously agreed upon; however, there is an increasing consensus on the following properties described as the *weak notion of agency* [1]:

- *Autonomy* providing agents with control over both their internal states and their actions, i.e. the ability to operate without direct human intervention;
- *Reactivity* giving agents the power to feel their environments and act upon them;
- *Pro-activeness* meaning that agents can actively pursue after their goals. In other words, agents may take initiatives; and
- *Social ability* allowing agents to communicate with human or other agents.

To some extent, one can think of an agent as a 'living' software entity that carries its own purposes.

There is no prescription on which, if any, specific AI techniques agents should employ. However, the nature of the agents as autonomous entities with clearly defined borders against their environments does make them excellent media for AI applications, since the complexity of the environments is effectively separated from the complexity of the problems of concern.

By treating agents as basic entities of software systems – in the same way that classes (or objects) are regarded as basic building blocks in the object-oriented software engineering approach, additional benefits can be expected from the software engineering processes. Due to their higher granularity and tighter encapsulation, complex systems can be better modeled, designed, and built with agents. In fact agent-oriented approach has been proposed as one of the next main stream software engineering paradigms due to its suitability for constructing complex systems [2].

Both of the aforementioned benefits should appeal to the interactive entertainment industry. Players seek the sense of fulfillment in games, and the games meet their needs by providing adequate challenges. There are excellent simple games, such as Tetris, that achieve this without sophisticated plot or intelligence. Yet in most other games the challenges have to be provided by the virtual creatures that inhabit the games. A common practice in game design is to use hard-coded scripting to control the storyline and provide creature behaviors. This design works especially well in situations where a firm grasp of the storyline is needed, however at the cost of the flexibility of creature behaviors and overall design. On the other hand, the agent-based approach provides excellent flexibility – each creature becomes an autonomous entity that has sensory input, reasoning and reaction of its own, allowing easy modification of its behaviors, and even online learning. In addition, there is the bonus of endless combinations of emergent behaviors. Since the end users (players) perceive the game creatures as intelligent individuals, modeling them as such in the beginning would make it easier to meet that expectation during later development stages. The software architectures of the games gain flexibility from the new approach as well. Agents guard not only their implementation details but also their states from the rest of the system and interact through high level communication languages. The design of such interactions is focused on the contents being exchanged rather than the action of exchange itself, posing a sharp contrast to the rigid stipulations that glues the objects together.

This paper tries to provide an anatomy of how agent technology can be applied to game design. It is organized as follows: A brief introduction to the Belief-Desire-Intention (BDI) agent theory [3], [4], [5] and an overview of the BDI.net agent framework [6] is first given in section 2, followed by close examinations of BattleCity.net, a re-implementation of the old arcade game BattleCity, in section 3. Finally, section 4 summarizes the main conclusions and indicates possible directions for future work.

2. Background
The above-mentioned agent definition provides no hint about the internal structures and operations of an agent. These are usually defined by a particular agent theory. Among the many theories of agent surveyed in [1], the BDI model is probably the most popular and influential one.

During the past decade, the BDI model has been well studied and formal models have been established [7], [8]. The strong theoretical basis is also supplemented by a number of successful industrial applications ranging from the early NASA projects [9] to the recent air traffic management [10] and air combat simulation [11].

The BDI model is rooted in the philosophical work of Bratman [12], which studies intention and its relations with other mental

attitudes. As its name implies, BDI features three major mental attitudes as its building blocks - belief, desire, and intention:

- *Belief* is the agent's knowledge about its environment and itself.
- *Desire* describes the agent's goal: a system state that the agent wants to achieve.
- *Intention* is the course of action that the agent has chosen to achieve that goal.

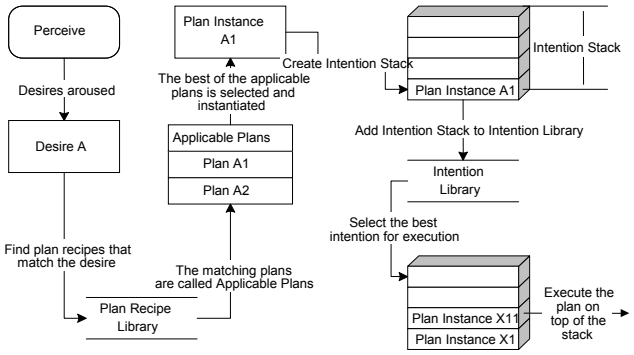
Another important concept in the BDI model is *plan*. Plans are used as recipes for achieving certain goals, guiding the deliberation process of the agents so they do not have to search through the entire space of possible solutions [13].

BDI.net is a Microsoft Visual C# [14] implementation of the AgentSpeak [15] BDI model. It is designed to be a lightweight framework for easy BDI implementation by casual programmers. This design philosophy leads to some favorable characteristics in comparison to the other existing BDI implementations, such as UM-PRS [16], JACK [17], JAM [18], dMARS [7], and ZEUS [19]. In particular, BDI.net brings the following advantages:

- Better exposure to the programmers. BDI.net is written using a popular programming language, and can be interfaced with all programming languages that support the .NET framework.
- Better alignment with the programmers' habits. Programmers have always embedded their knowledge in code and BDI.net respects that tradition instead of forcing them to resort to symbolic logic.
- Testability: BDI.net is designed with testability in mind, providing special facilities and considerations to aid the debugging process.
- Explicit communication support: facilities for agent communication are built-in.

A BDI.net agent operates through iterations of its execution cycle as depicted in Figure 1.

Figure 1: BDI.net execution cycle



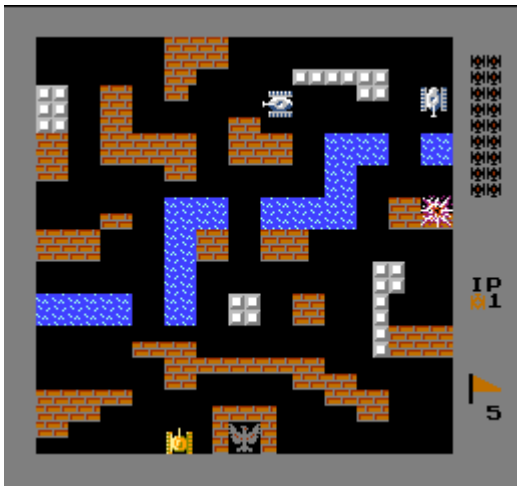
At the beginning of each cycle, the agent senses its environment and updates its belief. Consequently, desires may arouse in response to the stimulations it has just perceived. To fulfill its desires, the agent starts to search through its stock recipes (plans) for ones that match the desires. One desire may have several matching plans or no corresponding plans at all, in which case the desire is simply omitted. Only the best plan is selected for each desire to form a new intention for execution. At the end of the cycle, the intention with the highest utility value will be chosen from the intention library to be actually carried out. Sometimes during execution one plan may need to achieve a certain sub-goal that is beyond its control. In such scenarios it can stimulate a new desire and the agent will try to find proper plans to fulfill the desire in a similar fashion to that described earlier. For the originating plan, this is a synchronous call – it waits until the sub-goal is achieved or is believed to be unobtainable and devise further actions based on the results.

It is hard to overestimate the importance of the role that communication plays in any serious agent application – without communication, it is almost impossible to exercise any control on a multi-agent system, let alone collaborative problem solving. The Agent Communication Language (ACL) [20], along with the content language and ontology specifications, are the enabling technologies of agent communication. BDI.net implements the Foundation for Intelligent Physical Agents (FIPA) [21] ACL, a standard message language that specifies the encoding, semantics and pragmatics of the messages. Compared to the equally popular KQML [22], the FIPA ACL provides formal semantics, support for XML (which is relatively easy to parse), and other benefits such as specifications for interaction protocols

3. BattleCity Revived

BattleCity (Figure 2) is an old Nintendo Entertainment System (NES) game released by Namco group in 1985. The plot of the game is simple: the players (2 maximum) are supposed to protect their own base (the eagle on the bottom) and destroy all enemy tanks, then move on to the next stage. The game provides different terrain elements – rivers, bricks, stones, and trees, each featuring unique behavior. The players are supposed to conceive strategies in correspondent with the terrain configurations of the stages to protect themselves and effectively clear the hostile tanks.

Figure 2: BattleCity



The game is certainly interesting but is not a particularly challenging one. There are different kinds of enemy tanks that come with various speed and armor, but they all share the same stochastic behavior and can be easily destroyed.

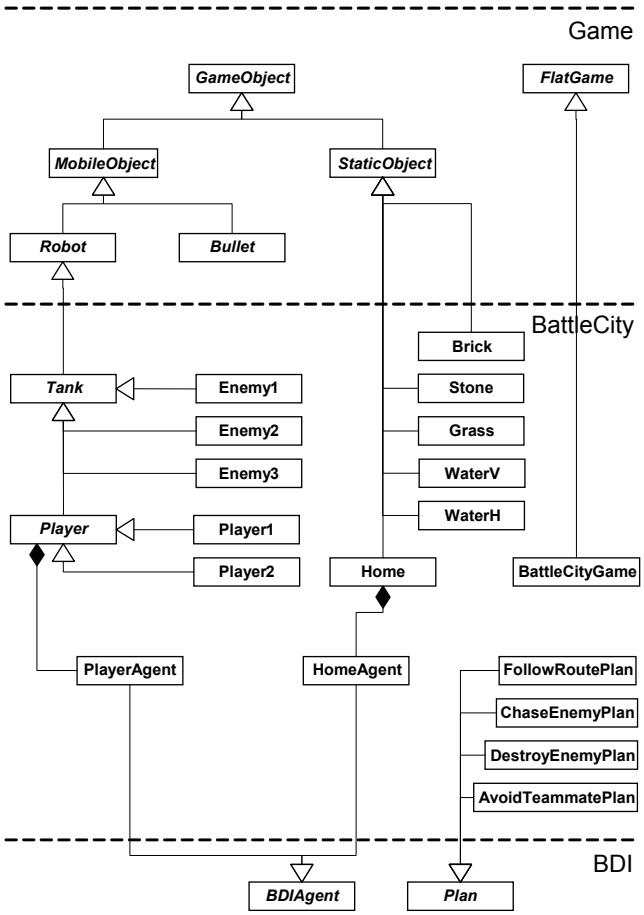
BattleCity.net is a remake of the old BattleCity game using the BDI.net framework. The rules and graphic elements are transplanted as is, but for the convenience of research the new game changed the plot to a computer vs. computer battle. While the enemy tanks remain stochastic, the player tanks and the base

are taken over by intelligent agents, so there is no human intervention needed.

3.1 Design

In an agent-oriented approach the target system is decomposed into autonomous entities and inanimate objects. The first step is to decompose the system properly and identify the agents. It does not take much effort to find out that there are nine types of entities – four terrain elements, three different enemy tanks, the base, and the player tank. In theory, all entities can be agents but that would result in a waste of precious resources. So which entities should be designed as agents? As in many other design issues there is no straightforward answer here. However a simple rule of thumb can provide some hint: if an entity has to perform actions to change the environment, and the motivation for performing these actions originates from the environment, it can be made an agent. In BattleCity.net, the player tank and the base are designed as agents, leaving the rest as plain objects. To get a more realistic behavior, the player agents have limited sight, which in turn demand the base to monitor its own vicinity and alert the player agents when enemies approach.

Figure 3 BattleCity.net Design (Selected Parts)

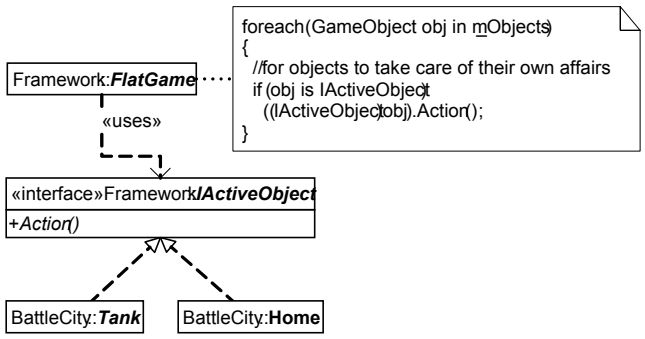


The agent-oriented approach is not a radical departure from the object-oriented approach, but rather its natural extension. Agent-based games can still take advantage of existing game development frameworks. BattleCity.net makes use of a 2D grid game development framework that provides rendering, sound, collision detection, and basic object prototypes. BDI agents are embedded into the game objects to control their behaviors, but the actual actions are still performed by the objects, cf. Figure 3. This approach is analogous to replacing the puppets used in a puppet show with robots – the investments on the costume and stage is preserved, the puppet show is getting more interesting, but the hassle of manipulating the puppets directly with threads is exempted.

3.2 Implementation

The game framework provides a control loop driven by an external timer to handle animations and collisions. It also gives the game entities a chance to handle their own affairs by calling their `Action` method through the `IActiveObject` interface, as illustrated in Figure 4.

Figure 4 IActiveObject Interface



The enemy tanks take this chance to make random moves and fire occasionally; the player tanks and the base pass the control to their 'brains', i.e. the agents. The terrain elements do not implement the `IActiveObject` interface as they are not active entities.

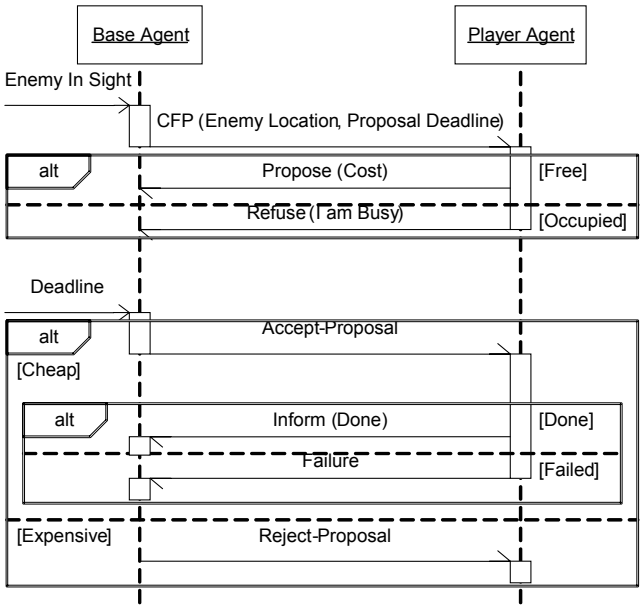
While the agent framework covers the mental operations of an agent, plans have to be set up to instruct the agent what can be done and how to do it. There are six major tasks for the player agent: to avoid bullets, to protect the base, to destroy enemies that can be fired at, to track the enemies down, to avoid collision with teammates, and to explore the battlefield when there is nothing better to do. Each of these situations has to be completely handled by at least one plan. Taken that the game is not complicated, one plan is conceived for each respective task. Similarly, for the base agent, only one plan is needed, which is to call for help.

Aside from the main control loop described earlier, each BDI agent also runs its own execution cycle in a separate thread. These loops have to be synchronized with the main control loop properly in order to avoid undesirable results. In other words, considering each timer event as one step, the agents should only be able to perform one set of actions at any step. BDI.net provides support synchronization at two levels – the agent execution cycle and the plans, both through semaphores. At the agent execution cycle level, agents can be configured to wait for a ready-to-go signal at the beginning of each cycle. The plans can also be interrupted and resumed at a later execution cycle, which is useful when a plan, such as chasing the enemy, takes multiple steps to finish. However special care has to be taken when implementing such plans – the agent's status has to be closely monitored to make sure that the plans are still valid under the current

circumstances, because new situations may arise while a plan is temporarily suspended and other plans of higher importance may be executed, invalidating the original conditions when the plan resumes.

Without centralized control, the game agents have to coordinate among themselves. Calling for help is one situation that demands such coordination. As illustrated below in Figure 5, when the base senses enemy tanks around, it sets up a `contract net` conversation with both player agents by broadcasting a Call for Proposal (CFP) message that indicates the enemy's position and the deadline of proposal submission. If a player agent is not too busy it will reply with the cost of performing such rescue, i.e. the time it takes to reach the spot. The base agent then chooses a winner from the submitted proposals. Once a contract is awarded to the player agent, it initiates a `subscribe` conversation with the base agent so that it will be kept posted the up-to-date information about the threat.

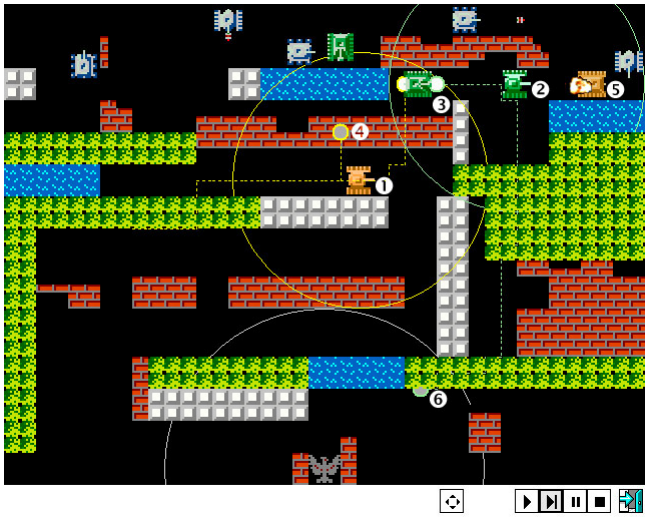
Figure 5 Call for Help Contract Net Conversation



At the current stage the rule-based approach (in which a set of if-else statements define the stimuli and the corresponding reactions) is used to generate the behaviors of the agents, and the standard A* search algorithm [23] is responsible for path finding. The same configuration would be used without problem in a game that does not employ the agent-based design. However, since all the agents are planning their own paths locally, there is a possibility that the player agents may run into each other and, in the worst case, result in a deadlock. One solution to this problem is to have the agents negotiate with each other in case of collision, but a simpler approach is taken here: both parties in collision will stop and wait for a random period of time and re-evaluate their situation. This way the first recovered from the 'coma' will always have to find a new path.

Figure 6 is a portrait of BattleCity.net in action showing the player tanks (❶ and ❷) actively engaged in battle. The small filled circles (❹, ❻, and the two near ❸)indicate the agents' intended destinations and the dotted lines leading to the circles depict the paths that the agents ought to follow. As shown in the screenshot, both player agents have multiple intentions. Player ❶ is committed in chasing after one of the enemy tanks (❸) but has not forgotten about its original intention of visiting a nearby spot (❹). Similarly, player ❷ is attacking one of the enemies (❺) but also has chasing ❸ and visiting ❻ in mind.

Figure 6 BattleCity.net Screenshot



4. Conclusions

The Agent-oriented approach to game development offers many benefits throughout the development cycle. It provides a natural way of modeling the game creatures at the very beginning, followed by a software architecture of high flexibility and low coupling which in turn paves the way to large scale and parallel development. In addition, developers can easily integrate their old game development frameworks with the new design approach.

Agents will certainly play a key role in game development in the near future. However the agent-oriented approach is not without drawbacks. The most serious problem is the conflict between the need to maintain a storyline and the autonomous nature of the agents. The storyline often demands precise control over certain creature's properties, but the autonomous agents may exhibit undesirable emergent behaviors due to the absence of centralized planning and control. Such unwanted emergent behaviors can be eliminated on a per-problem basis, like the teammate avoidance problem described earlier. While patch works can also be effective, the general solution to this kind of problems will be a hybrid architecture that features both centralized control and autonomous agents with 'back doors' for external control.

5. Acknowledgments

Support provided by Alberta Software Engineering Research Consortium is gratefully acknowledged.

6. References

[1] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice", The Knowledge Engineering Review, 10(2), pp.115-152, 1995.

[2] N. R. Jennings, "On agent-based software engineering", Artificial Intelligence, 177:277-296, 2000

[3] M. E. Bratman, D. Israel, M. Pollack, “Plans and Resource-Bounded Practical Reasoning”, Computational Intelligence, 4:349-355, 1988

[4] M. P. Georgeff and A. L. Lansky, “Reactive Reasoning and Planning”, Proceedings of the Sixth National Conference on Artificial Intelligence, volume 2, pp.677-682, Seattle, WA, 1987

[5] A. S. Rao and M. P. Georgeff, “Modeling Rational Agents within a BDI-architecture”, Proceedings of Knowledge Representation and Reasoning, pp.473-484, 1991

[6] Y. Li and P. Musílek, “BDI.net: A Lightweight Framework”, Proceedings of the Third ASERC Workshop on Quantitative and Soft Computing Based Software Engineering, pp. 49-53, 2003

[7] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, “A formal specification of dMARS”, Intelligent Agents IV, M. P. Singh, A. S. Rao, and M. Wooldridge, eds, pp. 155--176. Springer Verlag, Berlin, 1998

[8] M. d'Inverno and M. Luck, “Engineering AgentSpeak(L): A formal computational model”, Journal of Logic and Computation, 8(3), pp.233-260, 1998

[9] M. P. Georgeff and F. F. Ingrand, “Decision-Making in an Embedded Reasoning System”, Proc. Int'l Joint Conf. on Artificial Intelligence, pp. 972--978, August 1989

[10] M. Ljungberg and A. Lucas, “The oasis air traffic management system”, Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence, 1992

[11] A. S. Rao, A. Lucas, D. Morley, M. Selvestrel, G. Murray. “Agent-oriented architecture for air-combat simulation”, Technical Report Technical Note 42, The Australian Artificial Intelligence Institute, 1993

[12] M. E. Bratman, *Intentions, Plans and Practical Reasoning*, Harvard University Press, London, 1987

[13] A. S. Rao, “A Unified View of Plans as Recipes”, Contemporary Action Theory, Editors Ghita Holmstrom-Hintikka and Raimo Tuomela, Kulver Academic Publishers, The Netherlands, 1997

[14] Microsoft Visual C# .net Home Page, http://msdn.microsoft.com/vcsharp/, 2002

[15] A.S. Rao, “AgentSpeak(L) : BDI agents speak out in a logical computable language”, Proc. 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW'96, LNAI-1038, Springer Pub., 1996

[16] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny, “UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications.”, Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, Service, and Space, pp.842-849, 1994

[17] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas, “JACK Intelligent Agents – Components for Intelligent Agents in Java”, AOS Technical Report 1, 1999

[18] M. Huber, “Jam: a bdi-theoretic mobile agent architecture”, Proc. Intl. Conf. Autonomous Agents, pages 236–243, 1999

[19] H. Nwana, D. Ndumu, L. Lee, and J. Collis, “ZEUS: A tool-kit for building distributed multi-agent systems”, Applied Artificial Intelligence Journal, 13(1):129-186, 1999

[20] Y. Labrou, T. Finin, and Y. Peng, “Agent Communication Languages: The Current Landscape”, IEEE Intelligent Systems, 14(2):45-52, 1999

[21] Foundation for Intelligent Physical Agents Home Page, http://www.fipa.org/, 2003

[22] T. Finin et al, “Specification of the KQML Agent Communication Language”, Technical Report, DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993

[23] N. J. Nillson, “Problem-Solving Methods in Artificial Intelligence”, McGraw-Hill, 1971