Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2015, Article ID 271296, 30 pages
http://dx.doi.org/10.1155/2015/271296

Research Article

Artificial Intelligence in Video Games: Towards a Unified Framework

Firas Safadi, Raphael Fonteneau, and Damien Ernst

Université de Liège, Grande Traverse 10, Sart Tilman, 4000 Liège, Belgium

Correspondence should be addressed to Firas Safadi; fsafadi@ulg.ac.be

Received 27 August 2014; Revised 26 December 2014; Accepted 8 February 2015

Academic Editor: Alexander Pasko

With modern video games frequently featuring sophisticated and realistic environments, the need for smart and comprehensive agents that understand the various aspects of complex environments is pressing. Since video game AI is often specifically designed for each game, video game AI tools currently focus on allowing video game developers to quickly and efficiently create specific AI. One issue with this approach is that it does not efficiently exploit the numerous similarities that exist between video games not only of the same genre, but of different genres too, resulting in a difficulty to handle the many aspects of a complex environment independently for each video game. Inspired by the human ability to detect analogies between games and apply similar behavior on a conceptual level, this paper suggests an approach based on the use of a unified conceptual framework to enable the development of conceptual AI which relies on conceptual views and actions to define basic yet reasonable and robust behavior. The approach is illustrated using two video games, *Raven* and *StarCraft: Brood War*.

1. Introduction

Because artificial intelligence (AI) is a broad notion in video games, it is important to start by defining the scope of this work. A video game can be considered to have two main aspects, the context and the game. The game includes the elements that define the actual challenges players face and the problems they have to solve, such as rules and objectives. On the other hand, the context encompasses all the elements that make up the setting in which these problems appear, such as characters and plot. This work focuses on game AI, that is, AI which is concerned with solving the problems in the game such as defeating an opponent in combat or navigating in a maze. Conversely, context AI would deal with context-specific tasks such as making a character perform a series of actions to advance the plot or reacting to player choices. Thus, the scope of discussion is limited to the game aspect in this work.

Since video games are designed for human beings, it is only natural that they focus on their cognitive skills and physical abilities. The richer and more complex a game is, the more skills and abilities it requires. Thus, creating a truly smart and fully autonomous agent for a complex video game can be as challenging as replicating a large part of the complete human intelligence. On the other hand, AI is usually independently designed for each game. This makes it difficult to create thoroughly robust AI because its development is constrained to the scope of an individual game project. Although each video game is unique, they can share a number of concepts depending on their genre. Genres are used to categorize video games according to the way players interact with them as well as their rules. On a conceptual level, video games of the same genre typically feature similar challenges based on the same concepts. These similar challenges then involve common problems for which basic behavior can be defined and applied regardless of the problem instance. For example, in a first-person shooter one-on-one match, players face problems such as weapon selection, opponent position prediction and navigation. Each moment, a player needs to evaluate the situation and switch to the most appropriate weapon, predict where the opponent likely is or is heading and find the best route to get there. All of these problems can be reasoned about on a conceptual level using data such as the rate of fire of a weapon, the current health of the opponent and the location of health packs. These concepts are common to many first-person shooter games and are enough

to define effective behavior regardless of the details of their interpretation. Such solutions already exist for certain navigation problems for instance and are used across many video games. Moreover, human players can often effortlessly use the experience acquired from one video game in another of the same genre. A player with experience in first-person shooter games will in most cases perform better in a new first-person shooter game than one without any experience and can even perform better than a player with some experience in the new game, indicating that it is possible to apply the behavior learned for one game in another game featuring similar concepts to perform well without knowing the details of the latter. Obviously, when the details are discovered, they can be used to further improve the basic conceptual behavior or even override it. It may therefore be possible to create cross-game AI by identifying and targeting conceptual problems rather than their game-specific instances. Detaching AI or a part of it from the development of video games would remove the project constraints that push developers to limit it and allow it to have a continuous and more thorough design process.

This paper includes seven sections in addition to Introduction and Conclusion. Section 2 presents some related work and explains how this work positions itself beside it. Sections 3–6 present a development model for video game AI based on the use of a unified conceptual framework. Section 3 suggests conceptualization as a means to achieve unification. Section 4 discusses the design of conceptual AI while Section 5 discusses conceptual problems. Section 6 then focuses on the integration of conceptual AI in video games. Sections 7-8 include some applications of the development model presented in the previous sections. Section 7 describes a design experiment conducted on an open-source video game in order to concretize the idea of introducing a conceptual layer between the game and the AI. Section 8 then describes a second experiment which makes use of the resulting code base to integrate a simple conceptual AI in two different games. The Conclusion section ends the paper by discussing some of the merits of the proposed approach and noting a few perspectives for the extension of this research.

2. Related Work

Conceptualizing video games is a process which involves abstraction and is similar to many other approaches that share the same goal, namely, that of factoring AI in video games. More generally, abstraction makes it possible to create solutions for entire families of problems that are essentially the same when a certain level of detail is omitted. For example, the problem of sorting an array can take different forms depending on the type of elements in the array, but considering an abstract data type and comparison function allows a programmer to write a solution that can sort any type of array. This prevents unnecessary code duplication and helps programmers make use of existing solutions as much as possible so as to minimize development efforts. Another example of widely used abstraction application is hardware abstraction. Physical components in a computer can be seen as abstract devices in order to simplify software development. Different physical components that serve the same purpose, storage for example, can be abstracted into a single abstract storage device type, allowing software developers to write storage applications that work with any kind of storage component. Such a mechanism is used in operating systems such as NetBSD [1] and the Windows NT operating system family [2].

The idea of creating a unified video game AI middleware is not new. The International Game Developers Association (IGDA) launched an Artificial Intelligence Interface Standards Committee (AIISC) in 2002 whose goal was to create a standard AI interface to make it possible to recycle and even outsource AI code [3]. The committee was composed of several groups, each group focusing on a specific issue. There was a group working on world interfacing, one on steering, one on pathfinding, one on finite state machines, one on rule-based systems and one on goal-oriented action planning, though the group working on rule-based systems ended up being dissolved [3–5]. Thus, the committee was concerned not only with the creation of a standard communication interface between video games and AI, but with the creation of standard AI as well [6]. It was suggested that establishing AI standards could lead to the creation of specialized AI hardware.

The idea of creating an AI middleware for video games is also discussed in Karlsson [7], where technical issues and approaches for creating such middleware are explored. Among other things, it is argued that when state systems are considered, video game developers require a solution in between simple finite state machines and complex cognitive models. Another interesting argument is that functionality libraries would be more appropriate than comprehensive agent solutions because they provide more flexibility while still allowing agent-based solutions to be created. Here too, the possibility of creating specialized AI hardware was mentioned and a parallel with the impact mainstream graphics acceleration cards had on the evolution of computer graphics was drawn.

An Open AI Standard Interface Specification (OASIS) is proposed in Berndt et al. [8], aiming at making it easier to integrate AI in video games. The OASIS framework is designed to support knowledge representation as well as reasoning and learning and comprises five layers each dealing with different levels of abstraction, such as the object level or the domain level, or providing different services such as access, translation or goal arbitration services. The lower layers are concerned with interacting with the game while the upper layers deal with representing knowledge and reasoning.

Evidently, video game AI middleware can be found in video game engines too. Video game engines such as *Unity* [9], *Unreal Engine* [10], *CryEngine* [11], and *Havok* [12], though it may not be their primary focus, increasingly aim at not only providing building blocks to create realistic virtual environments but realistic agents as well.

Another approach that, albeit not concerned with AI in particular, also shares a similar goal, which is to factor development efforts in the video game industry, is game patterns. Game design patterns allow game developers to document recurring design problems and solutions in such a way that they can be used for different games while helping them understand the design choices involved in developing

a game of specific genre. Kreimeier [13] proposes a pattern formalism to help expanding knowledge about game design. The formalism describes game patterns using four elements. These are the name, the problem, the solution and the consequence. The problem describes the objective and the obstacles that can be encountered as well as the context in which it appears. The solution describes the abstract mechanisms and entities used to solve the problem. As for the consequence, it describes the effect of the design choice on other parts of the development and its costs and benefits.

Björk et al. [14] differentiates between a structural framework which describes game components and game design patterns which describe player interaction while playing. The structural framework includes three categories of components. These are the bounding category, which includes components that are used to describe what activities are allowed or not in the game such as rules and game modes, the temporal category which includes components that are involved in the temporal execution of the game such as actions and events, and the objective category which includes concrete game elements such as players or characters. More details about this framework can be found in Björk and Holopainen [15]. As for game design patterns, they do not include problem and solution elements as they do in Kreimeier [13]. They are described using five elements which are name, description, consequences, using the pattern and relations. The consequences element here focuses more on the characteristics of the pattern rather than its impact on development and other design choices to consider, which is the role of the using the pattern element. The relations element is used to describe relations between patterns, such as subpatterns in patterns and conflicting patterns.

In Olsson et al. [16], design patterns are integrated within a conceptual relationship model which is used to clarify the separation of concerns between game patterns and game mechanics. In that model, game mechanics are derived from game patterns through a contextualization layer whose role is to concretize those patterns. Conversely, new patterns can be extracted from the specific implementation of these game mechanics, which in the model is represented as code.

Also comparable are approaches which focus on solving specific AI issues. It is easy to see why, since these approaches typically aim at providing standard solutions for common AI problems in video games, thereby factoring AI development. For instance, creating models for intelligent video game characters is a widely researched problem for which many approaches have been suggested. Behavior languages aim to provide an agent design model which makes it possible to define behavior intuitively and factor common processes. Loyall and Bates [17] presents a goal-driven reactive agent architecture which allows events that alter the appropriateness of current behavior to be recognized and reacted to. ABL, a reactive planning language designed for the creation of believable agents which supports multicharacter coordination, is described in Mateas and Stern [18] and Mateas and Stern [19].

Situation calculus was suggested as a means of enabling high-level reasoning and control in Funge [20]. It allows the character to see the world as a sequence of situations and understand how it can change from one situation to another under the effect of different actions in order to be able to make decisions and achieve goals. A cognitive modeling language (CML) used to specify behavior outlines for autonomous characters and which employs situation calculus and exploits interval methods to enable characters to generate action plans in highly complex worlds is also proposed in Funge et al. [21], Funge [22].

It was argued in Orkin [23, 24] that real-time planning is a better suited approach than scripting or finite state machines for defining agent behavior as it allows unexpected situations to be handled more naturally. A modular goal-oriented action planning architecture for game agents similar to the one used in Mateas and Stern [18, 19] is presented. The main difference with the ABL language is that a separation is made between implementation and data. With ABL, designers implement the behavior directly. Here, the implementation is done by programmers and designers define behavior using data.

Anderson [25] suggests another language for the design of intelligent characters. The avatar definition language (AvDL) enables the definition of both deterministic and goal directed behavior for virtual entities in general. It was extended by the Simple Entity Annotation Language (SEAL) which allows behavior definitions to be directly embedded in the objects in a virtual world by annotating and enabling characters to exchange information with them [26, 27].

Finally, learning constitutes a different approach which, again, leads to the same goal. By creating agents capable of learning from and adapting to their environment, the issue of designing intelligent video game characters is solved in a more general and reusable way. Video games have drawn extensive interest from the machine learning community in the last decade and several attempts at integrating learning in video games have been made with varying degrees of success. Some of the methods used are similar to the previously mentioned approaches in that they use abstraction or concepts to deal with the large diversity found in video games. Case-based reasoning techniques generalize game state information to make AI behave more consistently across distinct yet similar configurations. The possibility of using case-based plan recognition to reduce the predictability of real-time strategy computer players is discussed in Cheng and Thawonmas [28]. Aha et al. [29] presents a case learning and plan selection approach used in an agent that learns to win against a number of different AI opponents in *Wargus*. In Ontañón et al. [30], a case based planning framework for real-time strategy games which allows agents to automatically extract behavioral knowledge from annotated expert replays is developed and successfully tested in Wargus as well. More work using Wargus as a test platform includes Weber and Mateas [31] and Weber and Mateas [32] which demonstrate how conceptual neighborhoods can be used for retrieval in case-based reasoning approaches.

Transfer learning approaches attempt to use the experience learned from some task to improve behavior in other tasks. In Sharma et al. [33], transfer learning is achieved by combining case-based reasoning and reinforcement learning and used to improve performance over successive games against the AI in *MadRTS*. Lee-Urban et al. [34] also uses

MadRTS to apply transfer learning using a modular architecture which integrates hierarchical task network (HTN) planning and concept learning. Transfer of structure skills and concepts between disparate tasks using a cognitive architecture is achieved in Shapiro et al. [35].

Although machine learning technology may lead to the creation of a unified AI that can be used across multiple games, it currently suffers from a lack of maturity. Even if some techniques have been successfully applied to a few commercial games, it may take a long time before they are reliable enough to become mainstream. On the other hand, video game engines are commonly used and constitute a more practical approach at factoring game development processes to improve the quality of video games. They are however comprehensive tools which developers need to adopt for the entire game design rather than just their AI. Furthermore, they allow no freedom in the fundamental architecture of the agents they drive.

The approach presented in this paper bears the most resemblance to that of creating a unified AI middleware. It is however not an AI middleware, strictly speaking. It makes use of a conceptual framework as the primary component which enables communication between video games and AI, allowing video game developers to use conceptual, game-independent AI in their games at the cost of handling the necessary synchronization between game data and conceptual data. A key difference with previous work is that it makes no assumptions whatsoever on the way AI should be designed, such as imposing an agent model or specific modules. Solutions can be designed for any kind of AI problem and in any way. A clear separation is made between the development of the conceptual framework, that of AI and that of video games. Because AI development is completely separated from the conceptual framework, its adoption should be easier as it leaves complete freedom for AI developers to design and implement AI in whichever way they are accustomed to. Furthermore, the simplicity of the approach made it possible to provide a complete deployment example detailing how an entire video game was rewritten following the proposed design. In addition, the resulting limited conceptual framework prototype was successfully employed to reuse some of the game AI modules in a completely different game.

3. Conceptualize and Conquer

Since video games, despite their apparent diversity, share concepts extensively, creating AI that operates solely on concepts should allow developers to use it for multiple games. This raises an important question however, namely, that of the availability of a conceptual interpretation of video games. In reality, for AI to handle conceptual objects, it must have access to a conceptual view of game data during runtime.

When humans play a video game, they use their faculty of abstraction to detect analogies between the game and others they have played in the past. Abstraction in this context can be seen as a process of discarding details and extracting features from raw data. By recalling previous instances of the same conceptual case, the experience acquired from the other games is generalized and transformed into a conceptual policy (i.e., conceptualized). For example, a player could have learned in a role-playing game (RPG) to use ranged attacks on an enemy while staying out of its reach. This behavior is known as kiting. Later, in a real-time strategy (RTS) game, that player may be faced with the same conceptual situation with a ranged unit and an enemy. If, at that time, the concept of kiting is not clearly established in the player's mind, they may remember the experience acquired in the RPG and realize that they are facing a similar situation: control over an entity with a ranged attack and the ability to move and the presence of an enemy. The player will thereby conceptualize the technique learned in the RPG and attempt to apply it in the RTS game. On the other hand, if the player is familiar with the concept of kiting, a simple abstraction of the situation will lead to the retrieval of the conceptual policy associated with it, without requiring the recall of previous instances and associated experiences and their conceptualization.

Note that kiting can be defined using only concepts, such as distance, attack range and movement. Distance can have several distinct interpretations, for example yards, tiles or hops. Attack range can be a spell range, a firearm range or a gravity range. Walking, driving and teleporting are all different forms of movement. Kiting itself being a concept, it is clear that concepts can be used to define other concepts. In fact, in order to define conceptual policies, different types of concepts are necessary, such as objects, relationships, conditions and actions. Weapon, enmity, mobility (The condition of being mobile.) and hiding are all examples of concepts.

According to the process shown in Figure 1, conceptual AI, that is AI which operates entirely on concepts, could be used in video games under the premise that three requirements are met. These would be:

(1) the ability to translate game states into conceptual states,

(2) the ability to translate conceptual actions into game actions,

(3) and the ability to define conceptual policies. (A conceptual policy maps conceptual states to conceptual actions.)

Though the third requirement raises no immediate questions, the other two appear more problematic, as translating states and actions needs to be done in real-time and there currently exists no reliable replacement for the human faculty of abstraction. It follows from the latter assertion that this translation must be manually programmed at the time of development. This means that the game developer must have access to a library of concepts during development and write code to provide access at runtime to both conceptual views and conceptual controls of the game for the AI to work with. Using such a process, both the real-time availability and the quality conditions of the translation are satisfied.

As is hinted in Figure 2, rather than translating game states into conceptual states discretely, it is easier to simply maintain a conceptual state in the conceptual data space (CDS). In other words, the conceptual state is synchronized with the game state. Every change in the game state, such as object creation, modification or destruction, is directly

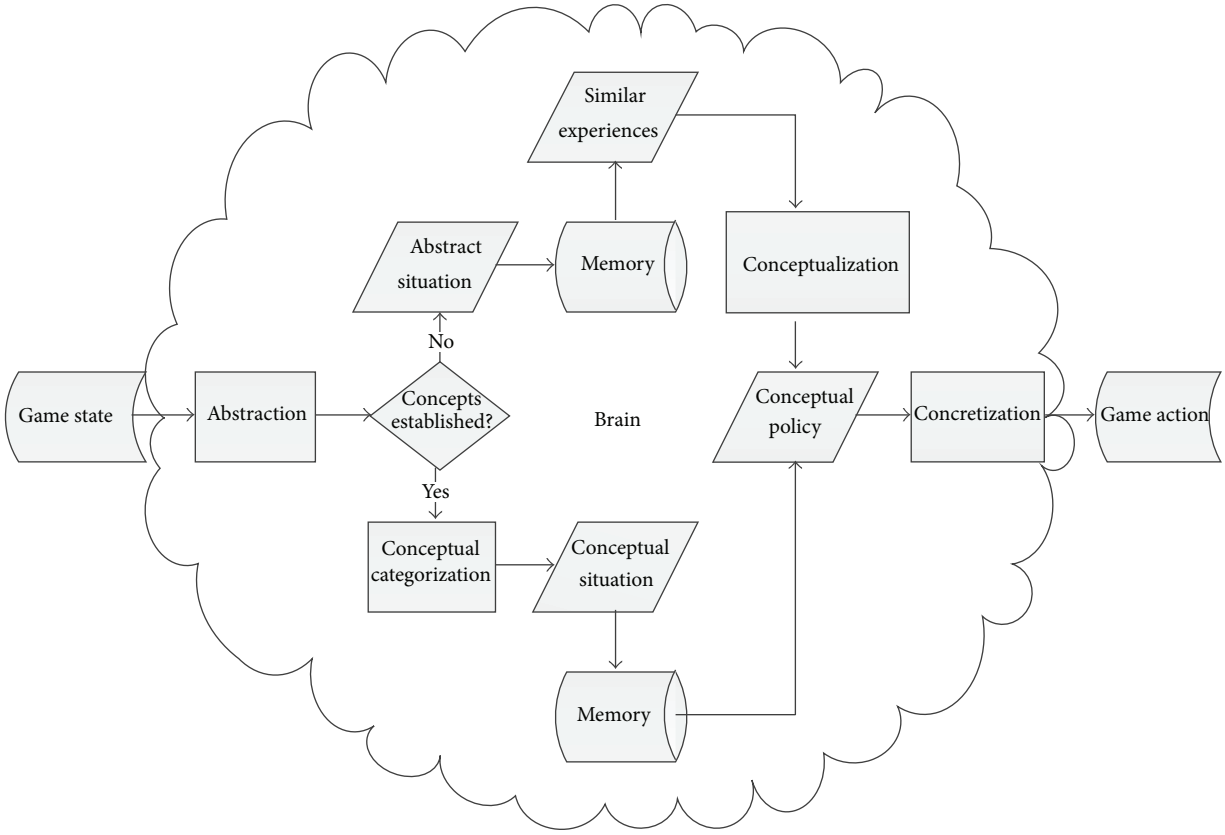

Figure 1: Possible process of human decision making in a video game using conceptual policies, as described above. If memory queries do not yield any results, a concrete policy is computed in real-time using other cognitive faculties such as logic or emotion.

propagated to the conceptual state. Note that there is no dynamic whatsoever in the CDS. A change in the CDS can only be caused by a change on the game side, wherein the game engine lies.

Obviously, this design calls for a unified conceptual framework (CF). That is, different developers would use the same conceptual libraries. This would allow each of them to use any AI written using this unique framework. For example, a single AI could drive agents in different games featuring conceptually similar environments, such as a first-person shooter (FPS) arena. This is illustrated in Figure 3.

From a responsibility standpoint, the design clearly distinguishes three actors:

(1) the game developers,

(2) the AI developers,

(3) and the CF developers.

The responsibilities of game developers include deciding which AI they need and adding code to their game to maintain in the CDS the conceptual views required by the AI as well as implementing the conceptual control interfaces it uses to command game agents. Thus, game developers neither need to worry about designing AI nor conceptualizing games. Instead, they only need to establish the links between their particular interpretation of a concept and the concept itself.

On the opposite side, AI developers can write conceptual AI without worrying about any particular game. Using only conceptual elements, they define the behavior of all sorts of agents. They also need to specify the requirements for each AI in terms of conceptual views and controls.

Finally, the role of CF developers is to extract concepts from games (i.e., conceptualize) and write libraries to create and interact with these concepts. This includes writing the interfaces used by game developers to create and maintain conceptual views and by AI developers to access these views and control agents.

Because the CF should be unique and is the central component with which both game developers and AI developers interact, it should be developed using an open-source and extensible model. This would allow experienced developers from different organizations and backgrounds to collaborate and quickly produce a rich and accessible framework. Incidentally, it would allow game developers to write their own AI while extending the framework with any missing concepts.

4. Designing Conceptual AI

From a technical perspective, writing conceptual AI is similar to writing regular AI. That is, developers are free to design their AI any way they see fit. Conceptual AI does not require a specific form. The only difference between conceptual AI and

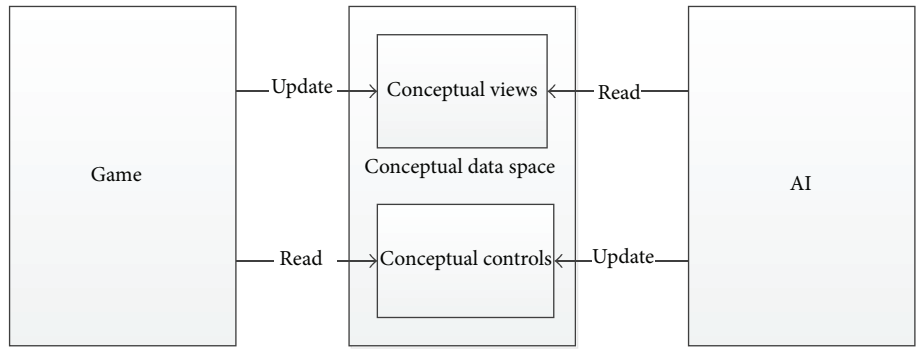

Figure 2: Basic architecture of a video game using conceptual AI. The game maintains a conceptual view of its internal state. A conceptual view is the projection of a part of the game state into conceptual space. Based on this conceptual data, the AI controls an agent in the game by issuing conceptual commands, which the game translates back into game actions.

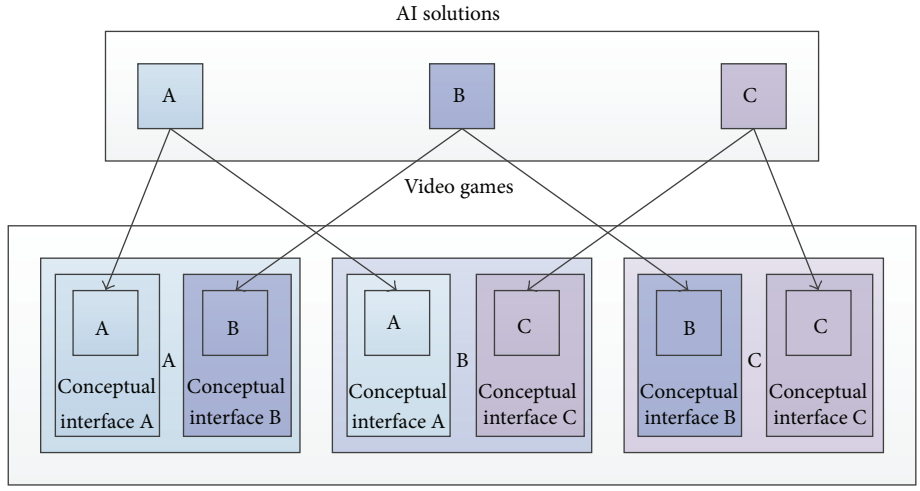

Figure 3: Using the same AI in multiple games. AI A can run in games A and B because both implement the conceptual interface A it requires. A conceptual interface is a set of conceptual views and controls.

regular AI is that the former is restricted to the use of conceptual data. Rather than operating on concrete objects, such as knights, lightning guns or fireball spells, it deals with concepts such as melee (Opposite of ranged, can only attack within grappling distance.) tanking units (A tanking unit, or tank, is a unit who can withstand large amounts of damage and whose primary role is to draw enemy attacks in order to ensure the survival of weaker allied units.), long-range hitscan weapons (A hitscan weapon is a weapon that instantly hits the target when fired (i.e., no traveling projectile).) and typed area-of-effect damage projectile abilities. (Area-of-effect abilities target an entire area rather than a single unit.) Likewise, actions involve conceptual objects and properties instead of concrete game elements and can consist in producing an anti-air unit or equipping a damage reduction accessory. This difference is illustrated in Algorithms 1 and 2.

Algorithm 1 shows an excerpt from the combat code of a Fortress Defender, a melee non-player character (NPC) in a RPG. A Fortress Defender can immobilize enemies, a useful ability against ranged opponents who might attempt to kite it. Before commanding the NPC to attack an encountered enemy, the code checks whether the type of opponent is one of those who use a ranged weapon and starts by using its immobilization ability if it is the case.

Algorithm 2 shows a possible conceptualization of the same code. Note how the design remains identical and the only change consists in replacing game elements with conceptual ones. As a result, the new code mimics a more conceptual reasoning. In order to prevent a ranged enemy from kiting the melee NPC, the latter checks whether a movement-impairing ability is available and uses it on the target before moving towards it. Whether the actual ability turns out to slow, immobilize or completely stun the opponent holds little significance as long as the conceptual objective of preventing it from kiting the NPC is accomplished. Although this requires developers to think in a more abstract way, they do retain the freedom of designing their AI however they are accustomed to.

Despite this technical similarity, the idea of conceptualizing video games suggests looking at AI in a more problem-driven way. There are two obvious reasons. First, conceptual AI does not target any game in particular, meaning that it

```
void handle_enemy(pc_t & enemy)
{
  ...
  if (enemy.type() == pc_t::cleric || enemy.type()
       == pc_t::sorcerer || enemy.type() == pc_t::ranger)
    queue_action(use_skill(Skill::root, enemy));
  queue_action(attack(enemy));
  ...
}
```

Algorithm 1: Fortress Defender combat code snippet.

```
void handle_enemy(pc_t & enemy)
{
  ...
  if (enemy.ranged() && can_impair_movement())
    queue_action(use_skill(get_skill(SkillType::
        disable_move), enemy));
  queue_action(attack(enemy));
  ...
}
```

Algorithm 2: Conceptual combat code snippet.

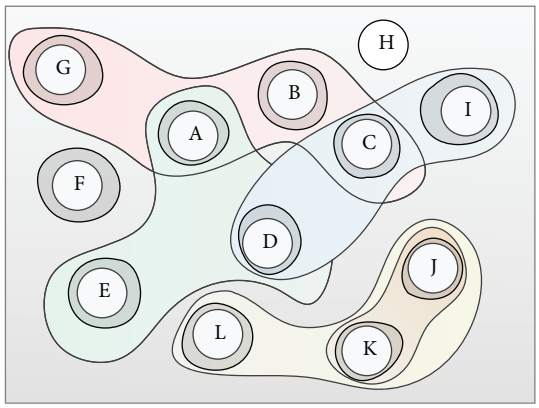

Figure 4: Conceptual problems (circles) and solutions (irregular forms). Instead of looking at the whole ADE and CDI problems found in two different games and solving them directly, solving problem D twice in the process, it is more interesting to identify the individual problems A, C, D, E, and I and solve them once first. A solution based on those of the individual problems can then be developed for each game without having to solve them again.

should not be defined as a complete solution for an entire game. Second, with the various interpretation details omitted, AI developers can more easily identify the conceptual problems that are common to games of different genres and target the base problems first rather than their combinations in order to leverage the full factoring potential of conceptualization. The idea of solving the base conceptual problems and combining conceptual solutions is illustrated in Figure 4.

Besides combining them, it can be necessary to establish dependencies between solutions. An AI module may rely on data computed by another module and require it to be running to function properly. For example, an ability planner module could require a target selection module by planning abilities for a unit or character according to its current target. This can be transparent to game developers when the solutions with dependencies are combined together into a larger solution. When they are not however, game developers need to know whether an AI module they plan on using has any dependencies in order to take into account the conceptual interfaces required by those dependencies. This means that AI developers have to specify not only the conceptual interface an AI solution uses, but also those required by its dependencies. Dependencies in combined and individual AI solutions are illustrated in Figure 5.

It can be argued that problems are actual video game elements. The difference between them and other elements such as objects is that they are rarely defined explicitly. They might be in games where the rules are simple enough to be listed exhaustively in a complete description of the problem the player is facing, but often in video games the rules are complex and numerous and a complete definition of the problems players must face would be difficult to not only write, but also read an understand. Instead, a description of the game based on features such as genres, environments or missions convey the problems awaiting players in a more intuitive way. With such implicit definitions, there can be many ways of breaking down video games into conceptual problems. Different AI developers might consider different problems and compositions. There are no right or wrong configurations of conceptual problems, though some may allow developers to produce AI more efficiently than others, just like the concepts making up the CF. It was suggested that the CF should be developed

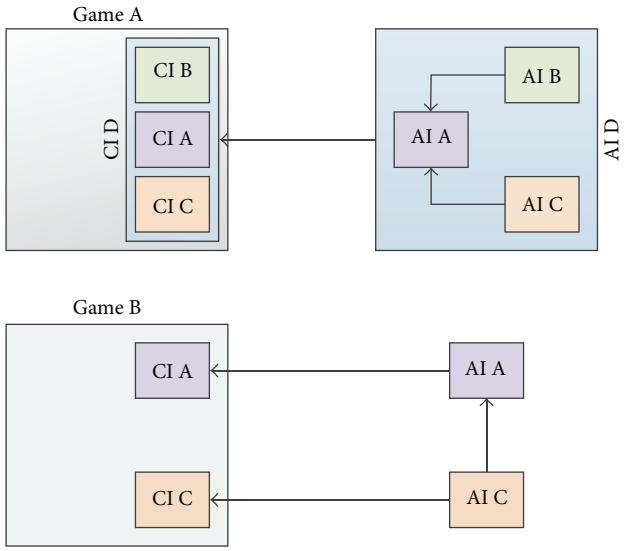

Figure 5: AI dependency in combined and individual solutions. Arrows represent requirement. A combination of AI solutions (AI D) has its own conceptual interface (CI D) which includes those of its components, making AI dependency transparent to game developers. In the case of separate AI solutions, a dependency (AI C requires AI A) translates into an additional conceptual interface (CI A) for game developers to provide.

using an open-source model to quickly cover the numerous existing concepts through collaboration and ease the addition of new ones. The same suggestion could be made for conceptual problems. If conceptual problems are listed and organized in the CF, AI developers can focus on solving conceptual problems instead of identifying them. As with concepts, as conceptual problems are identified and included in the CF, they become part of the AI developers' toolkit and allow them to better design their solutions. This task can be added to the responsibilities of CF developers, though since AI developers are the ones facing these conceptual problems and dealing with their hidden intricacies, they are likely to detect similarities between solutions to seemingly distinct problems, and in extension similarities between problems, and could collaborate with CF developers to restructure problems or contribute to the CF directly. Similarly, game developers deal with the details of the explicit elements and may have valuable contributions to make to the CF. In a way, CF developers can include both game and AI developers who could be assuming a double role either as direct contributors or as external collaborators. Such an organization together with the idea of breaking down video games into conceptual problems and using these as targets for conceptual AI is shown in Figure 6. The AI used in a video game could thus be described as solutions to elementary or composite conceptual problems.

5. Identifying Conceptual Problems

Conceptual problems are the heart of this video game AI development model. Indeed, it would serve little purpose to conceptualize video games if the resulting concepts could not be used to identify problems that are common to multiple games. Problem recurrence in video games is the raison d'être of such a model and why factoring video game AI is worth pursuing. The amount of factoring that can be achieved depends on how well recurring problems are isolated in video games not only of the same genre, but of any genre. This could be used as a measure of the efficiency of the model, as could be the amount of redundancy in AI solutions to disjoint problems. Clearly identifying and organizing conceptual problems is therefore a crucial dimension of this development model.

Problems and their solutions can either be elementary or composite. Elementary problems are problems whose decomposition into lesser problems would not result in any AI being factored. They are the building blocks of composite problems. The latter combine several problems, elementary or composite, into a single package to be handled by a complete AI solution. For example, an agent for a FPS arena deathmatch can be seen as a solution to the problem of control of a character in that setting. This problem could be decomposed into smaller problems which can be found in different settings such as real-time combat and navigation.

Navigation is a popular and well-studied problem found in many video games. Navigation in a virtual world often involves pathfinding. Common definitions as well as optimal solutions already exist for pathfinding problems. Examples include the A* search algorithm, which solves the single-pair shortest path problem (Find the least-cost path between a source node and a destination node in a graph.), and Dijkstra's algorithm, which solves the single-source shortest path problem (Find the least-cost path between a root node and all other nodes in a graph.). Although standard implementations can be found in developer frameworks and toolboxes, it is not unusual for developers to commit to their own implementation for environment-based customization.

A problem decomposition is often reflected in the AI design of a video game. For example, the AI in a RTS game may be divided into two main components. One component would deal with the problem of unit behavior and define behavior for units in different states such as being idle or following specific orders. This AI component could in turn include subcomponents for subproblems such as pathfinding. Defining autonomous unit behavior involves elements such as the design of unit response to a threat, an attack or the presence of an ally and is a problem that can be found in other games such as RPGs and FPSs. The other main component would deal with the problem of playing the RTS game to make it possible for a human player to face opponents without requiring other human players. This component could be organized in a number of modules to deal with the various tasks a player has to handle in a RTS game. A strategy manager can handle decisions such as when to attack and which units to produce. A production manager can handle construction tasks and assign workers to mining squads. A combat manager can designate assault locations and assign military units to combat squads. Squad managers can handle tasks such as unit formations and coordination and target selection. These AI components can provide insight on the different conceptual problems they attempt to solve and their organization. Coordination between a group of units

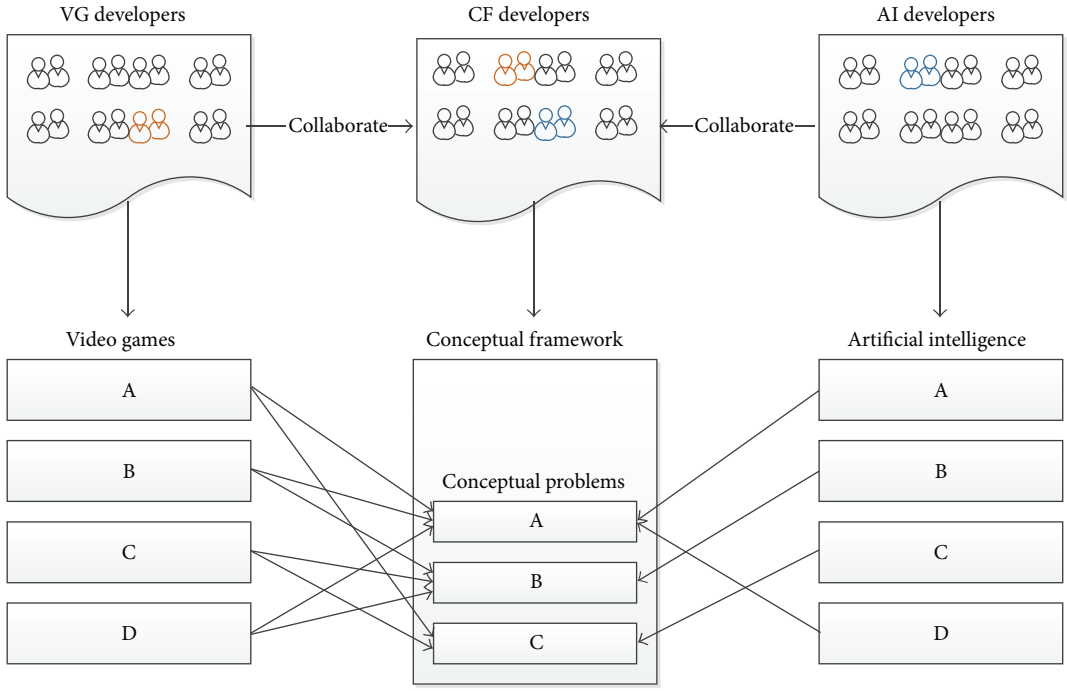

Figure 6: Collaboration between developers. Some game and AI developers, possibly large organizations or pioneers, also help developing the CF. Others only use it. Conceptual problems (CP) are listed and organized in the CF. A conceptual problem can be included in multiple games (CP B is included in VG B, C, and D) and can have multiple solutions (AI A and D are two different solutions for CP A).

to select a common target or distribute targets among units and maneuver units can be included in the larger problem of real-time combat which is not exclusive to the RTS genre. On the other hand, production-related decisions could be taken based on generic data such as total air firepower or total ground armor, making it possible for the same conceptual policy to be used for any RTS game providing a conceptual view through which such data can be computed.

More conceptual problems could be derived from these AI components. The real-time combat problem is a complex recurring problem found in many different games and may incorporate problems such as role management, equipment tuning, positioning, target selection and ability planning.

Many such problems have already been studied and the video game AI literature is rich in books which explore common problems in depth. Examples include *Programming Game AI by Example* by Buckland [36], *AI Game Engine Programming* by Schwab [37], *Artificial Intelligence for Games* by Millington and Funge [38], *Artificial Intelligence: A Modern Approach* by Russel and Norvig [39] and the *AI Game Programming Wisdom* books by Rabin [40–43]. More specific publications that focus on positioning for example also exist, such as work from Straatman and Beij [44] and Pottinger [45].

The remainder of this section briefly presents some of these problems and attempts to reason about them conceptually.

5.1. Role Management. Role management in combat is a recurring problem in video games. Role management deals with the distribution of responsibilities, such as damaging, tanking, healing and disabling, among a group of units or characters fighting together. This problem is often encountered in popular genres such as RPG (e.g., *World of Warcraft*, Blizzard Entertainment 2004), RTS (e.g., *Command & Conquer: Red Alert 3*, Electronic Arts 2008) and FPS (e.g., *Left 4 Dead*, Valve Corporation 2008). Roles can be determined based on several factors, including unit type or character class, attributes and abilities, equipment and items, unit or character state and even player skill or preference. Without targeting any specific game, it is possible to define effective policies for role management using conceptual data only. The data can be static like a sorted list of role proficiencies indicating in order which roles a unit or character is inherently suited for. Such information can be used by the AI to assign roles in a group of units of different type in combat. Dynamic data can also be used to control roles in battle, like current hit points (The amount of damage a unit can withstand.), passive damage reduction against a typed attack and available abilities of a unit. For instance, these can be used together to estimate the current tanking capacity for units of the same type. Naturally, the interpretation of these concepts varies from one game to another. Yet a conceptual policy remains effective in any case.

In a RPG, if a party is composed of a gladiator, an assassin and two clerics, the gladiator may assume the role of tank while a cleric assumes the role of healer and both the assassin and the other cleric assume the role of damage dealers. This distribution can vary significantly however. For example, the gladiator may be very well equipped and manage to assume the double role of tank and damage dealer, or conversely, the assassin may be dealing too much damage and

become the target. If the tank dies, the healer may become the target (Healing often increases the aggression level of a monster towards the healer, sometimes more than damaging the monster would.) and assume both the role of tank and healer. In this case, the other cleric may switch to a healer role because the tanking cleric could get disabled by the enemy or simply because the lack of defense compared to a gladiator could cause the damage received to increase drastically, making two healers necessary to sustain enemy attacks. Roles can thus be attributed during combat depending on character affinities and on current state data too.

A similar reasoning process can be used for units in a RTS game. In a squad composed of knights, sorcerers and priests, knights will be assuming the role of tanks and fighting at the frontlines, while priests would be positioned behind them and followed by the sorcerers. Sorcerers would thus be launching spells from afar while knights prevent enemy units from getting to them and priests heal both injured knights and sorcerers. Even among knights, some might be more suited for tanking than others depending on their state. Heavily injured knights should not be tanking lest they not survive a powerful attack. They should instead move back and wait for priests to heal them while using any long range abilities they might have. Unit state includes not only attributes such as current hit points but also status effects (A status effect is a temporary alteration of a unit's attributes such as speed or defense.) and available abilities. Abilities can significantly impact the tanking capacity of a unit. Abilities could create a powerful shield around a unit, drastically increase the health regeneration of a unit or even render a unit completely invulnerable for a short amount of time. Likewise, healing and damage dealing capacities can vary depending on available abilities. The healing or damage dealing capacity of a unit may be severely reduced for some time if the unit possesses powerful but high-cooldown (The cooldown of an ability is the minimum amount of time required between two consecutive activations. It is used to regulate the impact of an ability in a battle.) abilities which have been used recently. If the knights fall, either priests stay at the front and become the tanks or they move to the back and let the sorcerers tank depending on who of the two has the higher tanking capacity. Again, conceptual data can be used to generate operating rules to dynamically assign roles among units.

Algorithm 3 shows a conceptual AI function which can be used to determine the primary tank in a group. The primary tank is usually the unit or character that engages the enemy and is more likely to initiate a battle. Algorithm 4 details a possible implementation of the scoring function. It estimates the total amount of damage a unit could withstand based on its hit points and the overall damage reduction factor it could benefit from that can be expected during the battle given the abilities of both sides. A damage reduction factor is just one way of conceptualizing defensive attributes such as armor or evasion. The `dmgred_abilities` function could create a list of available damage reduction abilities and average their effects. For each ability, the amount of reduction it contributes to the average can be estimated using the reduction factor it adds, the duration of the effect, the cooldown of the ability as well as its cast time. In the case of conflicting abilities (i.e., abilities whose effects override each other), the average reduction bonus could be estimated by spreading the abilities over the cooldown period of the one with the strongest effect. The `dmgamp_abilities` function could work with damage amplification abilities in a similar way. It could also take into account the unit's resistance to status effects.

Any form of distribution of responsibilities between units or characters fighting together can be considered role management. Role management does not assume any objective in particular. Depending on the goal of the group, different distribution strategies can be devised. The problem of role management in combat can therefore be described as follows. Given an objective, two or more units or characters and a set of roles, define a policy which dynamically assigns a number of roles to each unit or character during combat in a way which makes the completion of the objective more likely than it would be if units or characters each assumed all responsibilities individually. An example of objective is defeating an enemy unit. Roles do not have to include multiple responsibilities. They can be simple and represent specific responsibilities such as acting as a decoy or baiting the enemy.

5.2. Ability Planning. Another common problem in video games is ability planning. It can be found in genres such as multiplayer online battle arena (MOBA) (e.g., *League of Legends*, Riot Games 2009 and *Dota 2*, Valve Corporation 2013) and RPG (e.g., *Aion: The Tower of Eternity*, NCsoft 2008 and *Diablo III*, Blizzard Entertainment 2012). Units or characters may possess several abilities which can be used during combat. For instance, a wizard can have an ice needle spell which inflicts water damage on an enemy and slows it for a short duration, a mana shield spell which temporarily causes damage received to reduce mana points (Also called magic points or ability points. Using abilities usually consumes mana points.) instead of health points and a dodge skill which can be used to perform a quick sidestep to evade an attack. Each of these abilities is likely to have a cost such as consuming a certain amount of mana points or ability points and a cooldown to limit its use. Units or characters thus need to plan abilities according to their objective to know when and in what order they should be used. As with role management, both static and dynamic data can serve in planning abilities. For example, if the enemy's class specializes in damage dealing, disabling abilities or protective abilities could take precedence over damaging abilities because its damage potential may be dangerously high. However, if the enemy's currently equipped weapon is known to be weak or its powerful abilities are known to be on cooldown, the use of protective abilities may be unnecessary.

Although abilities can be numerous, the number of ability types is often limited. These may include movement abilities, damaging abilities, protective abilities, curative abilities, enhancing abilities, weakening abilities and disabling abilities. Evidently, it is possible for an ability to belong to multiple categories. Abilities can be described in a generic way using various conceptual properties such as damage dealt, travel distance, conceptual attribute modification such as increasing hit points, effect duration, conceptual attribute cost such as action point cost, and cooldown duration. Abilities could also

```
void set_tank(UnitList & grp)
{
  //Get a list of the enemies the group is fighting
  UnitList enemies = get_nearby_threats(grp);
  Unit* toughest = NULL;
  double score = 0;
  //For each unit in the group, estimate its toughness against the enemy
  UnitList::iterator u;
  for (u = grp.begin(); u != grp.end(); ++u)
  {
    double cs = score_tanking(*u, grp, enemies);
    if (cs > score)
    {
      toughest = *u;
      score = cs;
    }
  }
  //Assign the role of tank to the toughest unit in the group
  if (score > 0)
    set_role(toughest, Role::tank);
}
```

Algorithm 3: Primary tank designation. This function could be used to determine which unit or character should engage the enemy.

```
double score_tanking(Unit* u, UnitList & grp, UnitList & enemies)
{
  //Set the base score to the current unit hit points
  double score = u->hitpts();
  //Get primary damage type of enemy
  DamageType dt = get_primary_dtype(enemies);
  //Get current damage reduction of the unit
  double dr = u->dmgred(dt);
  //Factor in average reduction bonus from ally abilities
  dr += dmgred_abilities(u, grp, dt);
  //Factor in average amplification bonus from enemy abilities
  dr -= dmgamp_abilities(u, enemies, dt);
  if (dr >= 1.0)
    return numeric_limits<double>::infinity();
  //Estimate effective hit points
  score *= 1.0/(1.0 - dr);
  return score;
}
```

Algorithm 4: Tanking capacity estimation. This function could be used to evaluate how fit of a tank a unit or character is.

be linked together for chaining, such as using an ability to temporarily unlock another. Ability planning can then be achieved without considering the materialization of the abilities in a particular world. Even special abilities used under certain conditions, such as a boss attack that is executed when the hit points of the boss fall under a specific threshold, can be handled by conceptual policies. For instance, a powerful special ability of a boss monster can be unavailable until a condition is met. At that point, a policy that scans abilities and selects the most powerful one available would automatically result in the use of the special ability. If the ability must be used only once, a long cooldown can stop subsequent uses assuming cooldowns are reset if the boss exits combat (This is to ensure that the boss can use the special ability again in a new battle in case its opponents are defeated or run away.).

Abilities can be planned according to some goal. For example, the goal could be to maximize the amount of damage dealt over a long period of time, also called damage per second (DPS). Maximizing DPS involves determining a rotation of the most powerful abilities with minimum downtime (A state where all useful abilities are on cooldown.). Conversely, the goal could be maximizing the amount of damage dealt over a short period of time, or dealing as much damage as possible in the shortest amount of time, also called

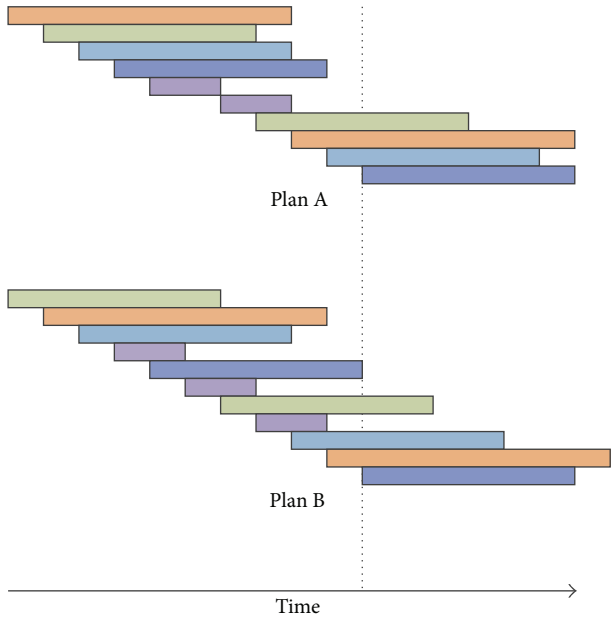

Figure 7: Ability planning using a burst strategy (Plan A) and a DPS strategy (Plan B). Rectangles represent cooldown periods of abilities. Each color corresponds to a different ability. Cast time is represented by a delay between the use of two consecutive abilities.

burst damage. A burst plan is compared to a DPS plan in Figure 7. While the burst strategy (Plan A) obviously deals more damage at the beginning, it is clear that the DPS strategy (Plan B) results in more damage over the entire period. The DPS plan orders long-cooldown abilities in a way that avoids simultaneous cooldown resets because these powerful abilities need to be used as soon as they are ready to make the most out of them, which is not possible if multiple ones become ready at the same time. It also avoids the downtime between the two consecutive uses of the purple ability in Plan A by better interleaving its casts throughout the time period. This leads to a higher output overall. Note that the burst strategy eventually converges towards the the DPS strategy.

When combat is largely based on abilities, predicting and taking into account enemy abilities becomes crucial for effective ability planning. If a lethal enemy attack is predicted, a unit or character can use a protective ability such as casting a shield just before the attack is launched to nullify its effect. Alternatively, it can use a disabling ability to prevent the enemy from using the ability or interrupt it. Known enemy abilities could be evaluated in order to predict the enemy's likely course of action and plan abilities accordingly. Just like role management, ability planning can be dealt with by defining interesting conceptual policies for various frequently encountered objectives.

In Algorithm 5, the DPS of an ability chain is estimated by adding up the damage and duration of each ability in the chain. Ability chains can be useful to represent linked abilities, for example when an ability can only be activated after another. They can also be used to generate different versions of the same ability in cases where using an ability after a specific one alters the attributes of the ability. If activating ability Y after X increases the damage of Y by 100% or reduces its use time by 50%, X and Y may be interesting from a DPS standpoint in cases where they otherwise are not when considered individually. The attribute values of Y can then be different from their default ones depending on the chain in which they appear. Of course, this function only estimates a theoretical damage and is more useful to generate all-purpose ability rotations than to plan abilities against a specific enemy. DPS can be more accurately estimated by factoring in the attributes and status effects of both the user and the target. If the target is very resistant against a particular type of damage, powerful abilities of this type may be outranked by less powerful ones dealing a different type of damage. The attributes or the status effects of the user can also affect the effectiveness of different abilities in different ways. One ability may have a high base damage value but gain nothing from the strength of the user, while another ability may have a low base damage but greatly benefit from the strength attribute and end up out-damaging the former. Use time can also vary depending on the user's attributes. Note that the use time corresponds to the total time during which the user is busy using an ability and cannot use another. Some abilities may involve both a cast time (i.e., a phase where the user channels energy without the ability being activated) and an activation duration (i.e., the time between the activation of the ability and the time the user goes back to an idle state). This function does not calculate other costs either. If abilities cost ability points or mana points to use in combat, these additional costs can be estimated for the chain together with the time cost since they usually cannot be ignored during a battle.

The concept of abilities is used in several genres. They usually correspond to actions that can be taken in addition to base actions, such as moving, at a cost. Given an objective and a set of abilities, the problem of ability planning is to produce a sequence of abilities which leads to the completion of the objective. Note that the set of abilities does not have to belong to a single entity. Like in role management, the objective can be fairly abstract and common, such as running away, disabling an enemy or protecting an ally.

5.3. Positioning. A frequently encountered problem in video games is positioning in the context of combat. Many genres include it, such as action-adventure (AA) (e.g., *The Legend of Zelda: Ocarina of Time*, Nintendo 1998), RTS (e.g., *StarCraft II: Wings of Liberty*, Blizzard Entertainment 2010) and RPG (e.g., *TERA: Rising*, Bluehole Studio 2011). Maneuverable units or characters have to continuously adjust their position according to their role or plan. A character whose role is to defend other characters will move to a position from which it can cover most of its allies from enemy attacks. An archer will attempt to stay outside the range of its enemies but close enough to reach them. A warrior with strong melee area-of-effect (AoE) attacks must move to the center of a group of enemies so as to hit as many of them as possible with each attack. An assassin may need to stick to the back of an enemy in order to maximize its damage. A specialized unit with poor defense could remain behind its allies in order to easily retreat in case it becomes targeted. This kind of behavior results from conceptual reasoning and needs not be specific to any one game.

```
double calc_dps(AbilityChain & ac)
{
  double dmg = 0;
  double dur = 0;
  AbilityChain::iterator a;
  //Add up the damage and duration of each ability in the chain
  for (a = ac.begin(); a != ac.end(); ++a)
  {
    dmg += (*a)->damage();
    dur += (*a)->usetime();
  }
  //DPS = total damage/total execution time
  if (zero(dmg))
    return 0;
  if (zero(dur))
    return numeric_limits<double>::infinity();
  return dmg/dur;
}
```

Algorithm 5: DPS estimation of an ability chain. This function can be useful for creating optimal DPS plans.

While navigation deals with the problem of traveling from one position to another, positioning is more concerned with finding new positions to move to. New positions can be explicitly designated for a unit or character or they could be implicitly selected by adjusting movement forces. For example, a unit may need to step outside the range of an enemy tower by moving to a specific position, or it could avoid bumping into a wall while chasing another unit by adding a force that is normal to the direction of the wall to its steering forces instead of selecting a position to move to. When positions are explicitly calculated, navigation may be involved to reach target positions. This can lead to a dependency between solutions to positioning problems and solutions to navigation problems.

Algorithm 6 shows a function which moves a unit out of the attack range of a group of enemies. For each enemy, it creates a circular area based on the enemy's attack range and centered on its predicted position. The latter is simply calculated by adding the enemy's current velocity to its position. This function ignores enemies that are faster than the unit because even if the unit is currently outside their range, it would eventually fall and remain within their reach. This could be delayed however. A list of immediate threats is thus created and used to compute a force to direct the unit away from the center of threats as quickly as possible. Note that this code does not differentiate between threats. It can be improved by weighting each position in the calculation of the center according to an estimation of the danger the threat represents. The more dangerous the threat, the larger the weight can be. This would cause the unit to avoid pressing threats with higher priority. This function could be used for kiting.

The code in Algorithm 7 shows how a straight line projectile can be dodged by a unit. A ray is created from the current position of the projectile and used to determine whether a collision with the unit is imminent. If this is the case, the unit is instructed to move sideways to avoid collision. The bounding radius of the projectile as well as that of the unit are used to determine the distance which must be traveled. The side on which the unit moves depends on its relative position vis-à-vis the projectile course. Of course, this function does not take into account the speed of the projectile and could therefore be better. If the projectile is slow compared to the unit, the movement could be delayed. On the other hand, if it is too fast, dodging may be impossible and the unit would not need to waste any time trying to do that.

Clearly, both code examples presented above follow a purely conceptual reasoning and could apply to a multitude of video games. They operate solely on conceptual objects and properties such as units, positions, velocities, steering forces and distances. Creating a comprehensive collection of general policies to deal with positioning problems can be time-consuming, making it unlikely to be profitable for a video game developer. When the solutions are conceptual and target all video games however, they may become profitable, providing incentive for AI developers to undertake the challenge.

Like role management and ability planning, positioning exists within the context of an objective. It is possible to design conceptual yet effective positioning policies for generic objectives such as maximizing damage dealt or minimizing damage received. Given an objective, the problem of positioning is to control the movement of a maneuverable entity in a way which serves the completion of the objective. Note that objectives could automatically be derived from roles. Depending on the space and the type of movement, different positioning problems could be considered. For example, it may be more interesting to consider 2D positioning and 3D positioning separately than to consider a single multidimensional positioning problem.

6. Integrating Conceptual AI in Video Games

Since conceptual AI is designed independently from games, an integration mechanism is necessary for it to be used by game developers. Game developers must be able to choose

```
void stay_safe(Unit* u, UnitList* enemies)
{
  UnitList threats;
  UnitList::iterator e;
  //Iterate on enemies to detect immediate threats
  for (e = enemies.begin(); e != enemies.end(); ++e)
  {
    //Ignore enemies that can't be outrun
    if (u->maxspeed() > (*e)->maxspeed() &&
      distance(u->position(), (*e)->position() + (*e)
         ->velocity()) <= (*e)->maxrange())
      threats.add(*e);
  }
  //Get the center of the threats
  Vector c = center(threats);
  //If the unit is located at the center, drop one of the threats
  if (c == u->position())
    c = center(remove_weakest(threats));
  //Create a force that pulls the unit away from the center
  Vector dir = u->position() - c;
  //Add a steering force of maximum magnitude
  u->addforce(dir*u->maxforce()/dir.norm());
}
```

Algorithm 6: Avoiding enemy attacks by staying out of range. This function can be used for kiting.

```
void dodge_projectile(Unit* u, Projectile* p)
{
  //Create a ray for the projectile course
  Ray r(p->position(), p->velocity());
  //Get a list of objects intersecting the ray
  ObjectList is = intersection(r, p->radius());
  //Only dodge if u is the first object to intersect the ray
  if (is.front() == u)
  {
    //Is u exactly on the projectile course?
    if (r.passthru(u->position()))
    {
      //Move perpendicularly by a distance equal to the sum of bounding radiuses
      u->move(u->position() + r.normal()*(p->radius() + u->radius()));
      return;
    }
    //Project the unit position on the projectile course
    Vector pr = r->project(u->position());
    //Get a normal to the projectile course with a norm equal to the distance between
      the unit position and its projection
    Vector mv = u->position() - pr;
    //Rescale it to the width of the intersection
    mv *= (p->radius() - (mv.norm() - u->radius()))/mv.norm();
    //Follow the normal to avoid collision
    u->move(u->position() + mv);
  }
}
```

Algorithm 7: Dodging a straight line projectile. This function assumes that the projectile is not penetrating.

and connect AI solutions to a game. This is achieved by registering AI controllers with conceptual objects. To assign control, partial or complete, of an entity in the game to a particular AI, the corresponding controller must be instantiated and registered with the projection of the entity in CDS. The AI then controls the conceptual entity, effectively controlling the entity in the game. For example, a game developer could use two AI solutions for a racing game, one for controlling computer opponents on the tracks and another for dynamically adjusting the difficulty of a race to the player's performance. Each time a computer opponent is added to the race, a new instance of the driving AI is created and registered with its conceptual projection. As for the difficulty AI, it can be created at the beginning of the race and registered with a real-time player performance evaluation object.

For each controllable conceptual object defined by the CF developers, a controller interface is defined together with it. This interface describes functions the AI must implement in order to be able to properly assume control over the conceptual object. These are not to be confused with the conceptual controls, also defined by the CF developers, which the AI can use to control the conceptual object and which are implemented by the game developers. Figure 8 illustrates the distinction.

It is possible for multiple controllers to share control of the same object. For example, a NPC could be controlled by different AI solutions depending on its state. It may have a sophisticated combat AI which kicks in only when the NPC enters a combat state and otherwise remains on standby, while a different AI is used when the NPC is in an idle state to make it roam, wander or rest. Multiple controllers however may lead to conflict in cases with overlapping control. One way to resolve conflicts is for AI controllers to have a table indicating a priority level for each conceptual control. Conceptual control calls would then be issued by controllers with their respective priorities and queued for arbitration. Of course, when multiple AI controllers are integrated into a complete solution, this issue can be handled by the author of the solution in whatever way they may choose and only the complete controller can be required to provide a priority table for conceptual controls.

Figure 9 shows how multiple controllers can be registered with a conceptual object. First, an object in the game, an Undead Peon, is created. Following this, its projection in CDS, a NPC, is created and linked to the Undead Peon. Finally, several independent AI controllers, one for generating idle behavior when the Undead Peon is idle, another for generating social behavior when the Undead Peon is around other Undead Peons and other types of NPCs and another for generating combat behavior when the Undead Peon is facing enemies, are created and registered with the NPC in CDS. In this case, there is no overlap in the control of the NPC by the different AI solutions. Using this registration mechanism, an AI controller can also verify that its dependencies are running and access them via the conceptual object.

Examples of functions found in controller interfaces are an update function and event handlers. An update function is used to update the internal state of the AI and can be called every game cycle or at an arbitrarily set update rate. This function is illustrated in Figure 10. Note how the NPC in CDS has no internal state update cycle. This is because there is no dynamic in the CDS. Objects in CDS are projections of game objects and are only modified as a result of a change in game objects. Event handlers are used to notify AI controllers of game events, such as a unit being killed by another. When an event occurs in the game, a conceptual projection is fired at the projection of the involved game object. The events that can involve a conceptual object are determined by the CF developers and used to create the controller interface. An AI controller does not necessarily need to handle all events. This is obvious for partial controllers. Therefore, it is possible for AI controllers to ignore some events. Event handlers are illustrated in Figure 11. Other examples are functions for suspending and resuming the controller.

When game developers link AI solutions to their games, they can either link them statically at build time or load them dynamically at runtime. Loading AI at runtime makes it easier to test different AI solutions and can also allow players to hook their own AI to the game. Typically, the AI would be running within the video game process, though it can be interesting to consider separating their execution. Deploying the AI in a separate process means it can run on a different machine. The latter could be optimized for AI processing or it could even be on the Internet, making it possible for AI developers to offer AI as a service. A multiprocess design can easily be imagined, as shown in Figure 12.

7. Graven: A Design Experiment

7.1. Description. The Graven experiment consists in rebuilding an open-source video game called *Raven* according to the design presented in the previous sections. (See Figure 2.) Namely, the AI is separated from the game and a conceptual layer is added in between. The AI is adapted to interact with the conceptual layer rather than directly with the game and the latter is modified to maintain a conceptual view in memory and use the conceptual AI. Albeit basic, Raven involves enough concepts to use as a decent specimen for conducting experiments relating to the deployment and use of a CF. The goal of the experiment is twofold.

(1) Concretize the design architecture as well as key processes in a working example.

(2) Obtain a code base to use as a limited prototype for testing conceptual AI in multiple games.

Note that the Graven experiment does not directly aim at demonstrating the efficiency of conceptual AI.

7.2. Raven. *Raven* is an open-source game written by Mat Buckland. A detailed presentation of the game as well as the code can be found in *Programming Game AI by Example* Buckland [36], where it is used to demonstrate a number of AI techniques such as path planning, goal-driven behavior, and fuzzy logic. It is a single-player, top-down 2D shooter featuring a deathmatch mode.

Maps are made of walls and doors and define spawn locations for players as well as items. When players die,

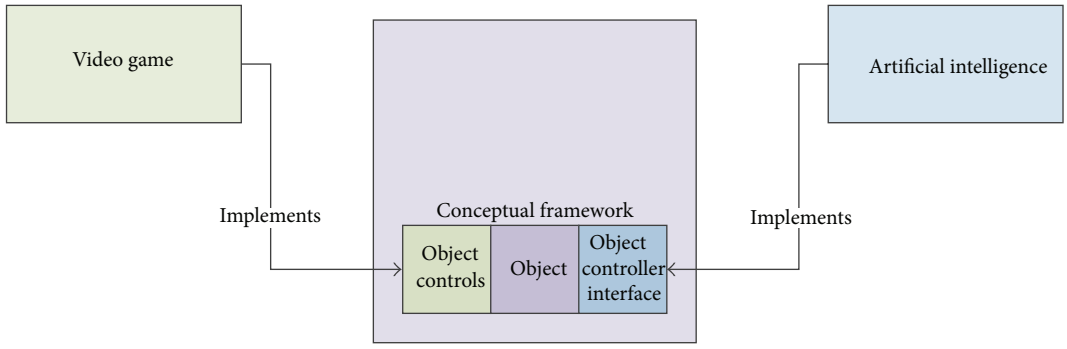

Figure 8: Conceptual controls and controller interface. Both are defined by the CF developers. Conceptual controls have to be implemented by game developers while the controller interface has to be implemented by AI developers.

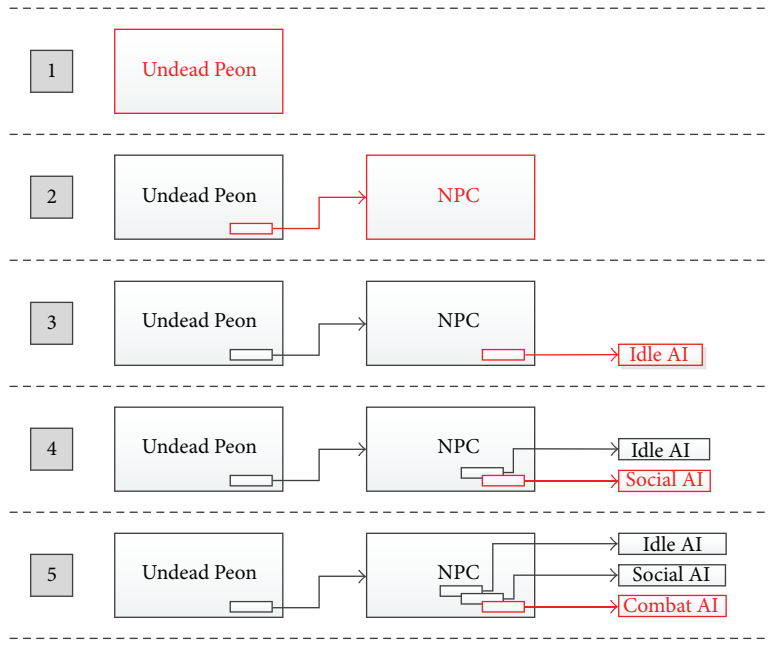

Figure 9: Registering multiple controllers with a conceptual object. Depending on its state, the Undead Peon is controlled by one of the three AI solutions.

they randomly respawn at one of the fixed spawn locations. Items also respawn at fixed time intervals after they are picked up. There are two types of items in Raven, weapons and health packs. Three weapons can be picked up. These are the Shotgun, the Rocket Launcher and the Railgun. A fourth weapon, the Blaster, is automatically included in every player's inventory at spawn time.

Each weapon is characterized by a unique set of features such as a firing rate and the maximum quantity of ammunition that can be carried for it. The Blaster is a basic projectile weapon with unlimited ammo. The Shotgun is a hitscan weapon which fires several pellets that spread out. The Rocket Launcher is a projectile weapon which fires rockets that deal AoE damage when they explode either on impact or after traveling a certain distance. The Railgun is a hitscan weapon which fires penetrating slugs that are only stopped by walls. Players can pick up weapons they already have. In that case, only the additional ammo is added to their inventory.

Initially, a default number of bots are spawned depending on the map. Bots can then be added to and removed from the game. The player can possess one of the existing bots to participate in a match. The left and right mouse buttons can be used to fire and move respectively, while numbers on the keyboard can be used to switch weapons. Despite their adorable look, these bots will compute the shortest paths to navigate the map, avoid walls, pick up ammo and health when needed, estimate their opponent's position to aim projectiles properly, use the most appropriate weapon depending on the situation, remember where they last saw or heard an opponent, chase or run away from an opponent, perform evasive maneuvers and, of course, kill. A preview of the game is shown in Figures 13 and 14.

The world in a Raven game is essentially composed of a map, bots and projectiles. The map is composed of walls and includes a navigation graph used for pathfinding as well as triggers. Triggers are used to define item pick up locations as

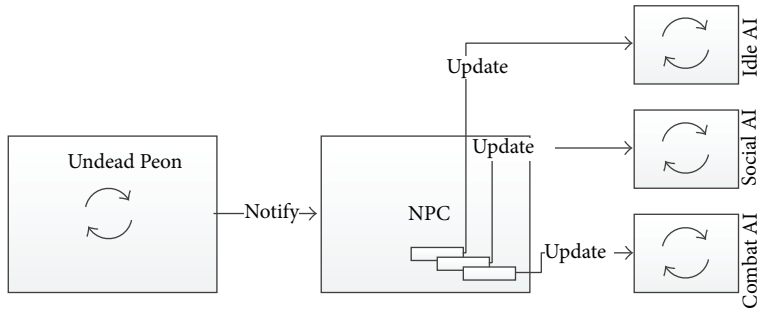

Figure 10: Updating the internal state of AI controllers when game objects update theirs. Note how objects in CDS do not have an update cycle.

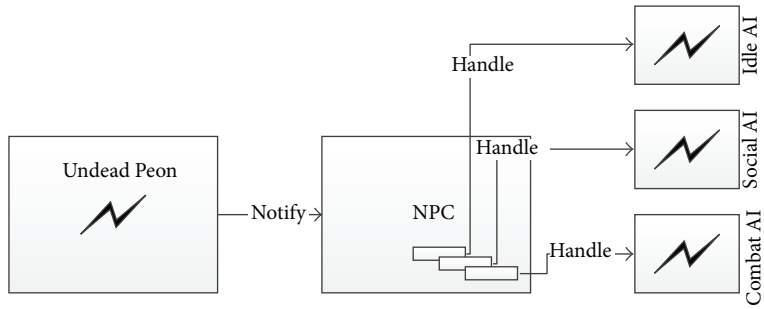

Figure 11: Event handling by AI controllers. Game events are projected into CDS before being pushed to AI controllers.

well as temporary sound sources. This composition is illustrated in Figure 15. The bot AI is primarily made of 6 interdependent modules, as shown in Figure 16. The brain module handles abstract goals and takes decisions such as attacking the current target or retrieving an item. The steering module manages steering forces resulting from multiple simultaneous behaviors such as avoiding a wall while seeking an enemy. The path planner module handles navigation requests by computing paths between nodes in the navigation graph. The sensory memory module keeps track of all the information the bot senses and remembers, such as visible enemies, hidden enemies and gunshot sound locations. The target selection module is used to select a target among a group of enemies. Finally, the weapon system module handles aiming and shooting and also includes per-weapon specific modules to evaluate the desirability of each weapon given the current situation.

7.3. Overview of the Code Structure. The code structure in Graven comprises five categories of components:

(1) the Raven classes,

(2) the conceptual view classes,

(3) the conceptual AI classes,

(4) the conceptual controls,

(5) and the Raven control classes.

The Raven classes are the game classes and an adaptation of the original code where all the AI components are removed and code to synchronize the conceptual view with the game state is added. The second category is a library of objects representing concepts corresponding to the Raven objects. The conceptual AI classes are a modification of the original AI code in which the AI is made to interact with the conceptual layer rather than the game. The fourth category includes a set of conceptual controls used by the conceptual AI to control bots. Finally, the Raven control classes implement these conceptual controls. Note that from a design perspective, the conceptual controls belong in the conceptual layer classes and their implementation in the game classes. They are separated in the code structure for the purpose of clarity.

7.4. Conceptualization. Raven is primarily composed of generic elements, as can be seen in Figure 15. A 2-dimensional world, projectiles, or walls are concepts commonly found in many video games. The added conceptual layer thus largely consists of clones of the objects in Raven. Unlike their Raven counterpart, however, conceptual objects are entirely static and do not update their own state. Instead, their state is only modified as a result of a modification on the game side. This is illustrated in Algorithm 8.

In Algorithm 8, the `Raven_Weapon` class declares a `ShootAt` function which is used to fire the weapon and which is implemented by each of the four Raven weapon classes. It also defines an `IncrementRounds` function which is used to update the number of rounds left for the weapon when a bot picks up additional ammo. In the corresponding `CptWeapon` class, the `ShootAt` function has been removed, and the `IncrementRounds` function has been replaced with

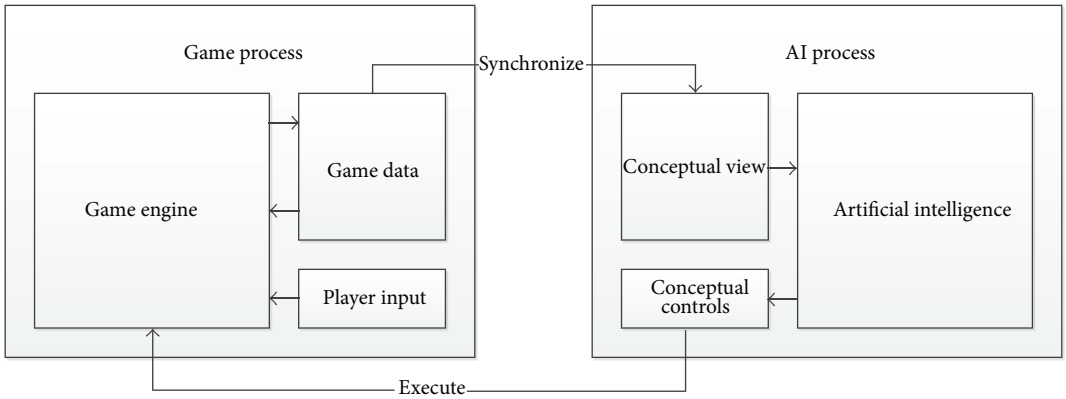

Figure 12: Running AI in a separate process. Synchronizing a conceptual view with game data requires an inter-process communication mechanism such as sockets or remote procedure call (RPC) systems. The mechanism is also required for using conceptual controls.

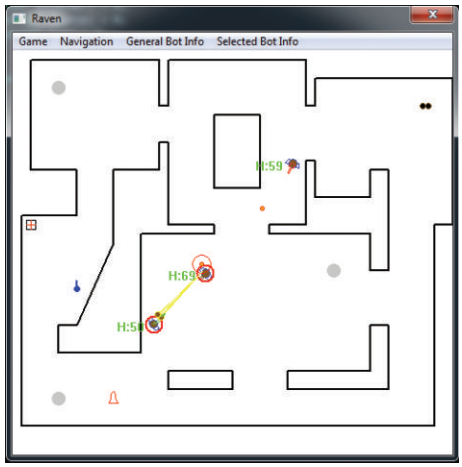

Figure 13: Screenshot taken from the Raven game. Player spawn locations are drawn in gray. On the top right corner is a Shotgun in black. At the bottom is a Rocket Launcher. At the left are a Railgun and a health pack. Each bot has its current hit points drawn next to it.

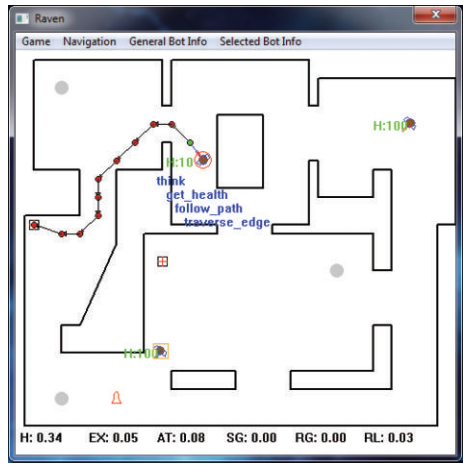

Figure 14: The AI information of a selected bot in Raven. Are shown are the goal stack, the path the bot is currently following, the current target of the bot (shown as a colored square around another bot) as well as a number of numerical desirabilities which indicate how important some of the actions the bot is thinking about are. From left to right, these are getting health, exploring, attacking the current target, getting a Shotgun, getting a Railgun and getting a Rocket Launcher.

a `SetNumRoundsLeft` function which can be used by the game to update the number of rounds left for the weapon in CDS. The synchronization process is detailed in a subsequent section.

Four conceptual controls have been defined. These are used by the conceptual AI to control the bots in Graven and are shown in Algorithm 9. The `ApplyForce` function can be used to apply a steering force to a bot and control its movement. The `RotateToward` function can be used to rotate a bot and control the direction of its field of view. The `EquipWeapon` function can be used to switch a bot's weapon to any of those it holds in its inventory. Lastly, The `ShootAt` function can be used to fire a bot's equipped weapon. These conceptual controls can be applied to a `CptMvAgent2D` object, the conceptual projection of a Raven bot in CDS. They are implemented game-side.

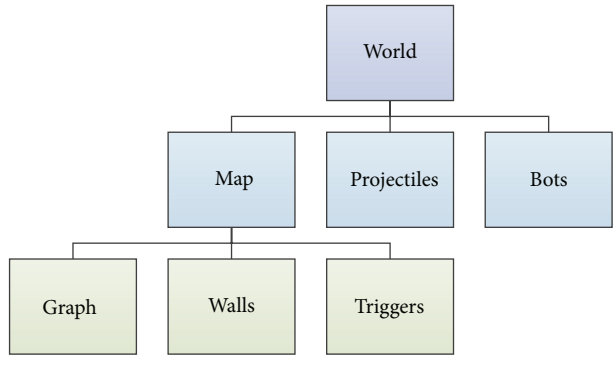

Figure 15: Overview of the Raven world composition.

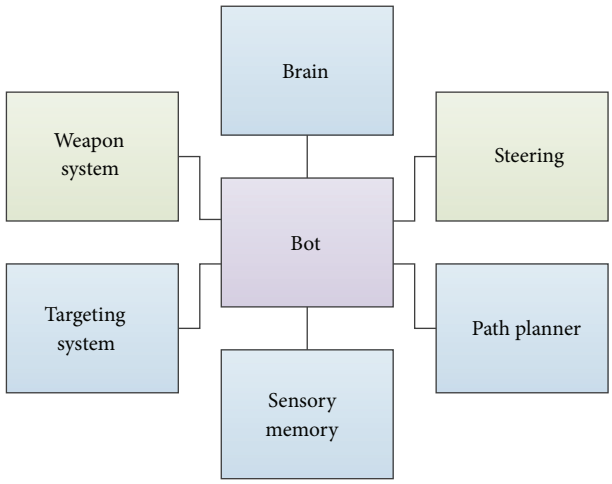

Figure 16: Overview of the Raven bot AI structure. Concrete actions such as firing a weapon or applying a steering force are taken by the green modules.

On the AI side, a `CptMvAgent2D` represents a controllable object and therefore the class comes with a controller interface. For an AI to be recognized as a valid controller by the game, it has to implement this interface. The interface is shown in Algorithm 10. It includes six functions. The `KilledBy_Handler` function is called whenever a bot is killed by an opponent and allows the controller to retrieve information about the killer. The `HeardSound_Handler` function is called when a bot hears a gunshot and can be used by the AI to find the origin of the sound. The `BotRemoved_Handler` function is called when a player removes a bot from the game via the main menu and can be used to notify other bots that the removed bot no longer exists. The `Suspend` and `Resume` functions serve to temporarily disable the controller when a bot is possessed by the player. The last `Update` function is used to allow the AI to update its state every game cycle.

Functionally, the AI in Graven is the same as the original Raven AI. It slightly differs in its structure, however. In Raven, the `Raven_WeaponSystem` class serves as a weapon inventory and handles weapon switching and also aiming and shooting, whereas weapon selection and aiming and shooting are separated in Graven. The central AI module through which other AI modules interact is the `CptBot` class. It resembles the original `Raven_Bot` class, though there are two significant differences. One, it interacts solely with the conceptual layer instead of the game. Two, it does not host any game state data such as current position and velocity, which is found in the `CptMvAgent2D` it controls. The AI state is thus clearly separated from the game state.

7.5. Creating a Conceptual View. The following process is used to synchronize the conceptual view with the Raven game state. For each class representing an object in the Raven game which has some projection in the CDS, a pointer to an object of the corresponding conceptual class is added to its data members. Then, following each statement that directly modifies a member of the class (without calling a function), a second operation is added to update the conceptual object accordingly. The conceptual object is created at the beginning of the class constructor and destroyed at the end of its destructor. By confining the synchronization code of an object to its class, its synchronization is done only once and never mixed with that of other objects. This idea is illustrated in Algorithm 11.

One problem with this technique is that it cannot be used directly with virtual classes because, even if they have corresponding conceptual classes, they do not represent actual objects with an independent projection in the CDS. The projection of a virtual class only exists as a part of the projection of a concrete class (i.e., a conceptual concrete class) and can only be accessed through this conceptual concrete class. A remedy for this problem is using a pure virtual getter implemented by its concrete subclasses, as shown in Algorithm 12. This involves another problem however, since virtual functions cannot be called in the constructor. (In C++, the virtual table of an object is only initialized after its construction is complete.) This is solved by moving the synchronization code in the constructor into an additional `sync` function for each class. This applies even to concrete classes. The `sync` function in a subclass always starts by calling the `sync` function of its superclass, ensuring that the synchronization code of an object remains confined within its class definition. A call to the `sync` function is added immediately after the creation of a conceptual object in the constructor of a concrete class, effectively executing the synchronization code of all its superclass constructors.

In order to properly synchronize certain template classes in Raven, it is necessary to use additional data type parameters to accommodate conceptual data types associated with the base parameters. For example, the `Trigger_Respawning` template class in Raven takes an entity type parameter which determines the type of game object that can activate the trigger. The class `Trigger_WeaponGiver` which extends `Trigger_Respawning` uses a `Raven_Bot` as parameter. However, its conceptual projection, a `CptTrigger WeaponGiver`, requires a `CptMvAgent2D` parameter. For this reason, the `Trigger_Respawning` class takes two parameters in Graven, one for the game data type and one for the corresponding conceptual data type.

7.6. Registering the Conceptual AI. The `CptBot` class implements the `CptMvAgent2D_Controller` interface and provides the AI functionality of the original Raven. The `CptMvAgent2D` class defines an `AddController` function which can be used by the game to register `CptMvAgent2D_Controller` objects with its instances. All registered controllers are updated and notified through the `CptMvAgent2D` instance. This is shown in Algorithm 13.

A `DMController` module can be used to instantiate and register `CptBot` objects without exposing the class to the game. Algorithm 14 shows how a controller is registered in the constructor of the `Raven_Bot` class. After creating and synchronizing a corresponding `CptMvAgent2D`, the `RegisterDMController` function is used to relegate the control of the bot to the conceptual AI.

```
class Raven_Weapon
{
  ...
  //this discharges a projectile from the weapon at the given target position (provided
      the weapon is ready to be discharged... every weapon has its own rate of fire)
  virtual void ShootAt(Vector2D pos) = 0;
  void IncrementRounds(int num)
  {
    m_iNumRoundsLeft += num;
    Clamp(m_iNumRoundsLeft, 0, m_iMaxRoundsCarried);
    //Synchronize rounds in CDS
    GetCptWeapon()->SetNumRoundsLeft(m_iNumRoundsLeft);
  }
  ...
};
class CptWeapon
{
  ...
  //Removed
  //virtual void ShootAt(Vector2D pos) = 0;
  void SetNumRoundsLeft(int n)
  {
    m_iNumRoundsLeft = n;
  }
  ...
};
```

Algorithm 8: Modifications in the conceptual copy of a Raven class. No game behavior is defined in CDS, which hosts nothing more than a projection of the game state.

```
//Applies a steering force to a bot
void ApplyForce(int agent_id, Vector2D force);
//Rotates the facing direction of a bot
void RotateToward(int agent_id, Vector2D position);
//Switches the equipped weapon of a bot
void EquipWeapon(int agent_id, int weapon_type);
//Fires the equipped weapon of a bot
void ShootAt(int agent_id, Vector2D position);
```

Algorithm 9: Conceptual controls used by the AI in Graven. In order, these are used by the conceptual AI to send commands to apply a steering force to a bot, to rotate a bot toward a certain position, to switch the currently equipped weapon of a bot, and to fire a bot's weapon at a given position. Together, these conceptual controls are sufficient to replicate the intricate behavior from the original code.

8. Using the Graven Targeting AI in StarCraft

8.1. Description. Following the Graven experiment which produced a limited CF prototype as well as a number of conceptual AI solutions, a second experiment was conducted to assess the work involved in using a simple conceptual AI solution in different games. Two games were used in this experiment, *Raven* and *StarCraft: Brood War* (BW), a real-time strategy game developed by Blizzard Entertainment. Albeit very different, these two games share a common conceptual problem, namely, target selection. Target selection in combat deals with deciding which enemy should be targeted in the presence of multiple ones. In Raven, a bot may face multiple opponents at the same time. Likewise in BW, a unit may face multiple enemy units on the battlefield. This experiment consists in using the same solution to this targeting problem in both Raven and BW, resulting in having the exact same code drive the targeting behavior of both bots in Raven and military units in BW.

8.2. StarCraft and the Brood War API. Although BW is not open-source, hackers have long been able to tamper with the game process by breaking into its memory space. Eventually, a development framework was built on top of this hacking. The Brood War Application Programming Interface (BWAPI) [46] is an open source C++ framework which allows AI developers to create custom agents by providing them with means to access the game state and issue commands. More information regarding the features and the design of the API can be found on the project's web page.

```
class CptMvAgent2D_Controller
{
protected:
  CptMvAgent2D* m_pOwner;
public:
  virtual ~CptMvAgent2D_Controller() {}
  //Called when a bot is killed by an opponent
  virtual void KilledBy_Handler(CptMvAgent2D* attacker) = 0;
  //Called when a bot hears a gunshot
  virtual void HeardSound_Handler(CptMvAgent2D* source) = 0;
  //Called when the player removes a bot from the game
  virtual void BotRemoved_Handler(CptMvAgent2D* bot) = 0;
  //Called when the player takes control of a bot
  virtual void Suspend() = 0;
  //Called when the player hands back control to a bot
  virtual void Resume() = 0;
  //Called every game update cycle
  virtual void Update() = 0;
};
```

Algorithm 10: The controller interface of a `CptMvAgent2D`. These functions are used by the `CptMvAgent2D` class to relay events to the AI.

```
void Raven_Bot::Spawn(Vector2D pos)
{
  ...
  //Direct modification: sync!
  m_iHealth = m_iMaxHealth;
  cpt->SetHealth(m_iHealth);
  //Function call: don't sync, already done in function definition!
  SetAlive();
  //Different class: don't sync, WeaponSystem has its own sync code!
  m_pWeaponSys->Initialize();
  ...
}
```

Algorithm 11: Conceptual data synchronization in Graven. Synchronization code in a class is added whenever its members are modified directly.

8.3. Targeting in Graven. The targeting system module in Graven, `CptTargetingSystem`, is used by the main AI module `CptBot`. To function, it requires another module, the sensory memory module `CptSensoryMemory`, which determines which enemies the bot currently senses. The targeting system works by setting an internal target variable which the bot module can read to find out which enemy it should aim at.

The original AI selects targets based on their distance to the bot and prioritizes closer enemies. It was modified to instead select targets based on their health and prioritize weaker enemies, a more interesting strategy for this experiment because the default unit AI in BW also uses distance as the primary factor in target selection. The main module function is shown in Algorithm 15. The vision update function in the sensory module is shown in Algorithm 16.

8.4. Completing the Graven Conceptual Layer. In terms of conceptual view, the requirements of the targeting module include those of its dependencies (i.e., the sensory memory module). The solution requirements can quickly be determined by looking at Algorithms 15 and 16. It requires a 2D world with the list of targetable entities that exist in it as well as a list of vision-blocking obstacles such as walls typically defined in a map. The entities must have their position, facing direction, field of view and health attributes synchronized. All of these concepts are already defined in the conceptual layer used in Graven.

In addition to those, BW involves three more concepts which are not present in Raven and which need to be defined. First, the concept of entity ownership is required to specify the player a unit belongs to. In Raven, a player is associated with a single bot. In BW, a player is associated with multiple units. Therefore, an owner property is required for units to differentiate between allies and enemies. The second concept is that of sight range. In Raven, a bot has a 180 degree field of view but its vision range is only limited by obstacles. In BW, a unit has a 360 degree field of view but can only

```
class MovingEntity: public BaseGameEntity
{
  ...
  //Virtual accessor – Retrieves the conceptual projection of this entity
  virtual CptMvEntity2D* GetCptMvEntity2D() const = 0;
  ...
  void SetVelocity(const Vector2D & NewVel)
  {
    m_vVelocity = NewVel;
    //Velocity changed, update conceptual data
    GetCptMvEntity2D()->SetVelocity(m_vVelocity);
  }
  ...
}
class Raven_Bot: public MovingEntity
{
protected:
  //The conceptual projection
  CptMvAgent2D* cpt;
  ...
public:
  //Returns the entire conceptual projection of this bot
  CptMvAgent2D* GetCptMvAgent2D() const { return cpt; }
  //Returns the conceptual projection of the MovingEntity part of this bot
  CptMvEntity2D* GetCptMvEntity2D() const { return cpt; }
  //Returns the conceptual projection of the BaseGameEntity part of this bot
  CptEntity2D* GetCptEntity2D() const { return cpt; }
  ...
}
```

Algorithm 12: Synchronization with virtual classes. The virtual class `MovingEntity` uses a pure virtual getter implemented by its concrete subclass `Raven_Bot` for its synchronization code.

```
class CptMvAgent2D: public CptMvEntity2D
{
private:
  //List of registered controllers
  std::list<CptMvAgent2D_Controller*> controllers;
  ...
public:
  //Registers a new controller
  void AddController(CptMvAgent2D_Controller* c)
  {
    controllers.push_back(c);
  }
  //Notifies controllers that a bot has been removed from the game
  void BotRemoved(CptMvAgent2D* bot)
  {
    std::list<CptMvAgent2D_Controller*>::iterator it;
    for (it = controllers.begin(); it != controllers.end(); ++it)
    {
      (*it)->BotRemoved_Handler(bot);
    }
  }
  ...
}
```

Algorithm 13: Controller management in the `CptMvAgent2D` class.

```
Raven_Bot::Raven_Bot(Raven_Game* world,Vector2D pos):
      ...
{
  //Create the conceptual projection
  cpt = new CptMvAgent2D(world->GetCptWorld2D());
  //Synchronize initialization
  sync();
  ...
  //Instantiate and register a DMController
  RegisterDMController(cpt);
}
```

Algorithm 14: Conceptual AI registration in Graven. The `RegisterDMController` function is defined in the `DMController` module and is used to instantiate the `CptBot` class.

```
void CptTargetingSystem::Update()
{
  int LowestHPSoFar = MaxInt;
  m_pCurrentTarget = 0;
  //grab a list of all the opponents the owner can sense
  std::list<CptMvAgent2D*> SensedBots;
  SensedBots = m_pOwner->GetSensoryMem()->GetListOfRecentlySensedOpponents();
  std::list<CptMvAgent2D*>::const_iterator curBot = SensedBots.begin();
  for (curBot; curBot != SensedBots.end(); ++curBot)
  {
    //make sure the bot is alive and that it is not the owner
    if ((*curBot)->isAlive() && (*curBot != m_pOwner->GetAgent()))
    {
      int hp = (*curBot)->Health();
      if (hp < LowestHPSoFar)
      {
        LowestHPSoFar = hp;
        m_pCurrentTarget = *curBot;
      }
    }
  }
}
```

Algorithm 15: Modified target selection in Graven. Health is compared instead of distance.

see up to a certain radius. A sight range property is thus required. The third concept is the plane. The world in BW is two-dimensional but there are ground and air units. Ground units are not always able to attack air units and vice versa. A property to indicate the plane in which a unit exists and which planes it can target is thus needed. As a result, five new members are added to the `CptMvAgent2D` class, a player ID, a sight range, a plane flag, and two plane reach flags. Note that the sensory memory module is slightly modified to take into account this information, though this has no impact on its functionality in Raven.

As far as conceptual controls are concerned, the aiming and shooting controls in Graven are not necessary for BW. When a unit in BW is given an order to attack another unit, the target only needs to be within firing range to be automatically attacked continuously. Only one conceptual control, an attack command, is required for this experiment and added to the conceptual framework.

8.5. Integrating the Targeting AI in StarCraft. In order to use the targeting AI from Graven in BW, there are a few tasks that need to be completed. These are

(1) adding code to the game to maintain in memory a conceptual view including the elements mentioned above,

(2) implementing the attack conceptual control,

(3) and creating an AI solution which makes use of the targeting AI to control units.

8.5.1. Conceptual View. The conceptual view is maintained using 3 callback functions provided by the BWAPI, the `onStart` function which is called at the beginning of a BW game, the `onEnd` function which is called at the end of the game and the `onFrame` function which is called every game frame. The code added in each of these functions is shown

```
void CptSensoryMemory::UpdateVision()
{
  //for each bot in the world test to see if it is visible to the owner of this class
  const std::list<CptMvAgent2D* > & bots = m_pOwner->GetWorld()->GetAllBots();
  std::list<CptMvAgent2D* >::const_iterator curBot;
  for (curBot = bots.begin(); curBot != bots.end(); ++curBot)
  {
    //make sure the bot being examined is not this bot
    if (m_pOwner->GetAgent() != *curBot)
    {
      //make sure it is part of the memory map
      MakeNewRecordIfNotAlreadyPresent(*curBot);
      //get a reference to this bot's data
      CptMemoryRecord & info = m_MemoryMap[*curBot];
      //test if there is LOS between bots
      if (m_pOwner->GetWorld()->isLOSOkay(m_pOwner->GetAgent()->Pos(), (*curBot)->Pos()))
      {
        info.bShootable = true;
        //test if the bot is within FOV
        if (isSecondInFOVOfFirst(m_pOwner->GetAgent()->Pos(),
             m_pOwner->GetAgent()->Facing(),
             (*curBot)->Pos(), m_pOwner->GetAgent()->FieldOfView()))
        {
          info.fTimeLastSensed = Clock->GetCurrentTime();
          info.vLastSensedPosition = (*curBot)->Pos();
          info.fTimeLastVisible = Clock->GetCurrentTime();
          if (info.bWithinFOV == false)
          {
            info.bWithinFOV = true;
            info.fTimeBecameVisible = info.fTimeLastSensed;
          }
        }
        else
        {
          info.bWithinFOV = false;
        }
      }
      else
      {
        info.bShootable = false;
        info.bWithinFOV = false;
      }
    }
  }//next bot
}
```

Algorithm 16: Vision update in the Graven sensory memory module.

in Algorithms 17, 18, and 19, respectively. The `syncUnit` function is shown in Algorithms 20.

Because the source code of BW is not available, the synchronization process is different from the one used in the Graven experiment. Every game cycle, the game state is scanned and new and destroyed units are added to and removed from the conceptual view and the states in CDS are synchronized with unit states in the game.

8.5.2. Conceptual Controls. The `Attack` conceptual control is easily implemented using the basic `attack` command players can give to units in BW. The implementation is shown in Algorithms 21.

8.5.3. Conceptual AI. For units capable of attacking, an attack AI is added to the list of controllers of their projection using the `RegisterDMController` function. This function instantiates the `CptBot` class, which is similar to the one in Graven but which has been modified to only use the sensory memory and targeting system modules. The update function of the `CptBot` module is shown in Algorithms 22. Note that the sensory memory module only registers reachable enemy units. Allied units are ignored.

```
void GravenAIModule::onStart()
{
  ...
  //Create 2D world
  cptWorld = new CptWorld2D();
  //Add an empty map
  cptWorld->pSetMap(new CptMap2D());
  //Set map dimensions
  cptWorld->GetMap()->pSetSizeX(Broodwar->mapWidth() * 32);
  cptWorld->GetMap()->pSetSizeY(Broodwar->mapHeight() * 32);
}
```

Algorithm 17: Conceptual view code in the `onStart` callback function. Map dimensions in BW are given in build tiles, each build tile representing a 32 by 32 area.

```
void GravenAIModule::onEnd(bool isWinner)
{
  ...
  //Destroy world
  delete cptWorld;
}
```

Algorithm 18: Conceptual view code in the `onEnd` callback function. The conceptual world destructor also destroys associated objects.

8.6. Results. The same targeting AI was successfully used in both Raven and BW, as shown in Figures 17 and 18. Unsurprisingly, the CF prototype (more specifically the `CptMvAgent2D` class) built from Raven, a very simple 2D shooter, had to be slightly extended for this experiment. Even so, the effort required to integrate the Graven targeting AI in BW was minimal. Of course, the AI was minimal too. This shows however that the work involved in creating conceptual AI that can be used in different games does not have to grow significantly with the number of games it can be applied to and that when a conceptual problem is clearly identified, it can be solved independently of the game it appears in.

Obviously, though it may not have been the goal of the experiment, the modified unit AI performs better in combat than the original one for ranged units, since it uses a better a strategy. In the presence of enemies, the original unit AI acquires a target by randomly selecting one within firing range. The modified unit AI on the other hand selects among targets within its sight radius the one with the lowest health. Because the sight range of a ranged unit is often close to its firing range, this behavior is similar to the original one in the sense that the unit does not move to reach a target when another target that is already in firing range exists. The behavior is therefore close but the unit does target weak enemies first in order to reduce their firepower as fast as possible. Moreover, setting a short memory span in the `CptSensoryMemory` class prevents units from remembering runaway targets for too long and starting to look for them. This helps maintain similarity between the original and modified unit AI. That way, the original unit behavior is maintained, making it harder for players to notice any difference other than the improved targeting strategy. Needless to say, the targeting AI remains completely unchanged. Note that modifying the sensory module to pick up targets that are within firing range rather than sight range makes the strategy work for melee units as well.

Table 1: Units lost in each battle for each group with the modified and original unit AI.

Battle	1	2	3	4	5	6	7	8	9	10
Units lost										
Modified AI	3	3	1	3	2	3	3	4	4	3
Original AI	5	5	5	5	5	5	5	5	5	5

The modified unit AI was tested using 10 battles of 5 Terran Ghosts versus 5 Terran Ghosts, one group being controlled by the modified unit AI and the other by the original unit AI. Ghosts are ranged ground units. The group with the modified unit AI won every battle. The number of Ghosts lost during each battle is reported in Table 1.

9. Conclusion

The main contribution of this research is an approach for the development of AI for video games based on the use of a unified conceptual framework to create a conceptual layer between the game and the AI. (The AI referred to here is game related and does not include context related AI as specified at the beginning of this work.) This approach is inspired by an interpretation of human behavior. Human players have the ability to detect analogies between games and generalize, or conceptualize, the knowledge acquired in one game and apply it in another. By conceptualizing video games and asking game developers to create conceptual views of their games using a unified framework, it becomes possible to create solutions for common conceptual problems and use them across multiple video games. Developing solutions for conceptual problems rather than specific video games means that AI design is no longer confined to the scope of individual game projects and can be more efficiently refined over time. Such conceptual AI can then serve as a core engine

```
void GravenAIModule::onFrame()
{
  ...
  //For each unit visible to the player
  Unitset units = Broodwar->getAllUnits();
  for (Unitset::iterator u = units.begin(); u != units.end(); ++u)
  {
    //Ignore neutral units which include mineral fields and critters
    if (u->getPlayer()->isNeutral())
      continue;
    //Get the projection of the unit in CDS
    CptMvAgent2D* cptUnit = dynamic_cast<CptMvAgent2D*>(cptEntityMgr->GetEntityFromID
      (u->getID()));
    //Projection found, synchronize state and update controllers
    if (cptUnit)
    {
      syncUnit(cptUnit, *u);
      cptUnit->Update();
    }
    //Projection not found, create one
    else if (u->exists() && u->isCompleted())
    {
      cptUnit = new CptMvAgent2D(cptWorld);
      syncUnit(cptUnit, *u);
      cptWorld->pAddBot(cptUnit);
      cptEntityMgr->RegisterEntity(cptUnit);
      //If the unit can attack, register the targeting AI
      if (u->getPlayer() == Broodwar->self() && u->canAttack())
      {
        RegisterDMController(cptUnit);
      }
    }
  }
  //Remove projections of units that no longer exist in the game
  std::list<CptMvAgent2D*> cptUnits = cptWorld->GetAllBots();
  for (std::list<CptMvAgent2D*>::iterator c = cptUnits.begin(); c != cptUnits.end(); ++c)
  {
    if (!Broodwar->getUnit((*c)->ID()) || !Broodwar->getUnit((*c)->ID())->exists())
    {
      cptEntityMgr->RemoveEntity(*c);
      cptWorld->pRemoveBot((*c)->ID());
    }
  }
}
```

Algorithm 19: Conceptual view code in the `onFrame` callback function. The `RegisterDMController` creates a `CptBot`, which uses the Graven targeting AI to attack enemies, and adds it to the list of controllers of the `CptMvAgent2D`.

for driving agents in a variety of video games which can be complemented by game developers specifically for each game. This would both reduce AI redundancy and facilitate the development of robust AI.

Such an approach can result in a number of advantages for game developers. First, it means that they no longer need to spend a lot of resources to design robust game AI unless they want to and can simply use existing AI solutions. Even though they have to add code for the creation of conceptual views, not having to worry about game AI can result in significant cuts in development time. For example, they would not even need to plan for coordination mechanisms between multiple agents in the game. Moreover, they do not need to use conceptual AI for all tasks. They can select the problems they want to handle using conceptual AI and use regular AI for other tasks. Story and environment related AI, which this approach does not apply to, can be designed using existing tools and techniques, such as scripting engines and behavior trees, which make it easy to implement specific behavior. In addition, the continuous development of conceptual AI is likely to yield better quality solutions over time than what can be achieved through independent game projects. It may also be that clearly identifying and organizing the conceptual problems that make up the challenges offered by video games

```
void GravenAIModule::syncUnit(CptMvAgent2D* u, Unit unit)
{
  //Synchronize CptEntity2D attributes
  u->SetID(unit->getID());
  u->SetEntityType(cpttype_bot);
  u->SetScale(1);
  u->SetBRadius(MAX(unit->getType().height() / 2, unit->getType().width() / 2));
  u->SetPos(Vector2D(unit->getPosition().x, unit->getPosition().y));
  //Synchronize CptMvEntity2D attributes
  u->SetHeading(Vector2D(unit->getVelocityX(), unit->getVelocityY()));
  u->SetVelocity(Vector2D(unit->getVelocityX(), unit->getVelocityY()));
  u->SetMass(1);
  u->SetMaxSpeed(unit->getType().topSpeed());
  u->SetMaxTurnRate(unit->getType().turnRadius());
  u->SetMaxForce(unit->getType().acceleration());
  //Synchronize CptMvAgent2D attributes
  u->SetMaxHealth(unit->getType().maxHitPoints() + unit->getType().maxShields());
  u->SetHealth(unit->getHitPoints() + unit->getShields());
  u->SetScore(unit->getKillCount());
  u->SetPossessed(false);
  u->SetFieldOfView(360);
  u->Face(Vector2D(unit->getVelocityX(), unit->getVelocityY()));
  u->SetWorld(this->cptWorld);
  u->SetStatus(unit->exists() ? CptMvAgent2D::alive: CptMvAgent2D::dead);
  u->SetPlayer(unit->getPlayer()->getID());
  u->SetSightRange(unit->getType().sightRange());
  u->SetPlane(unit->isFlying());
  u->SetAirReach(unit->getType().airWeapon() != WeaponTypes::None);
  u->SetGroundReach(unit->getType().groundWeapon() != WeaponTypes::None);
}
```

Algorithm 20: Synchronizing conceptual unit state. Some attributes are not required by the targeting AI and only serve as illustrations.

could allow game developers to compose new challenges more easily.

Since this approach allows AI development to progress independently of video games, it could lead to the birth of a new game AI business. AI developers could compete to create the best AI solutions and commercialize them or they could collaborate to design a solid open-source AI core which would be perfected over time. Additionally, machine learning techniques would be more straightforward to apply with a unified conceptual representation of game elements. These techniques can be used to learn specialized behavior for each game which can enhance the basic generic behavior. This is similar to the way humans tune their generic experience as they learn specific data about a video game they are playing to improve their performance in that particular game.

With an open-source unified conceptual framework, incentive for both game developers and AI developers to contribute to the development of the framework and the conceptualization of video games would exist. AI developers would benefit from a better conceptual framework because it would help factor AI better and allow more efficient AI development, resulting in better quality AI which benefits game developers directly when they integrate it in their games to create smarter, more challenging and more realistic agents.

Because the conceptual layer constitutes a sort of middleware, a new version of the conceptual framework may not be compatible with AI developed prior to its update. Even if it is, legacy AI may require an update in order to benefit from the improved conceptual framework. Another disadvantage of the approach is that it requires more computational resources in order to maintain a conceptual view in memory during runtime, though this may not represent a major obstacle with mainstream hardware featuring increasingly more processing cores and system memory. Other issues may also arise from the separation of AI from video games. Indeed, game developers could lose some control over the components of their games and subsequently over the ability to balance them. For instance, it may be necessary to design new mechanisms to allow game developers to retain control over the difficulty of the game and adjust the skill level of their agents. Furthermore, although machine learning techniques such as imitation learning could benefit from a larger learning set as a unified conceptual representation would give them access to data from many games, they would require a translation process to project human actions into conceptual data space since, unlike AI actions, those are not conceptual. In other words, without a translation process, conceptual game states could only be linked to concrete game actions.

Though an implementation of the approach was presented to illustrate some applications, alternative implementations can easily be imagined. For example, even if the AI code was compiled alongside the game code in Graven, it

```
void Attack(int agent_id, int target_id)
{
  Unit u = Broodwar->getUnit(agent_id);
  Unit v = Broodwar->getUnit(target_id);
  // Don't attack under explicit move orders
  if (u->getOrder().getID() == Orders::Move)
    return;
  // Already attacking that target
  if (u->getLastCommand().getTarget() != NULL && u->
      getLastCommand().getTarget()->getID() == target_id)
    return;
  if (v->getType().isFlyer())
  {
    if (u->getType().airWeapon() != WeaponTypes::None)
      u->attack(v);
    return;
  }
  else
  {
    if (u->getType().groundWeapon() != WeaponTypes::None && u->exists())
      u->attack(v);
    return;
  }
}
```

Algorithm 21: Implementation of the `Attack` conceptual control in BW. Because the targeting AI only selects targets the unit can attack, the test to see whether the unit is flying could be discarded.

```
void CptBot::Update()
{
  //if the bot is under AI control but not scripted
  if (!GetAgent()->isPossessed())
  {
    //examine all the opponents in the bots sensory memory and select one
    //to be the current target
    if (m_pTargetSelectionRegulator->isReady())
    {
      m_pTargSys->Update();
    }
    //update the sensory memory with any visual stimulus
    if (m_pVisionUpdateRegulator->isReady())
    {
      m_pSensoryMem->UpdateVision();
    }
    //Attack
    if (m_pAttackRegulator->isReady() && m_pTargSys->isTargetPresent())
    {
      Attack(m_pOwner->ID(), m_pTargSys->GetTarget()->ID());
    }
  }
}
```

Algorithm 22: The update function of the `CptBot` class. The function uses the `Attack` conceptual control to issue commands to the units in the game.

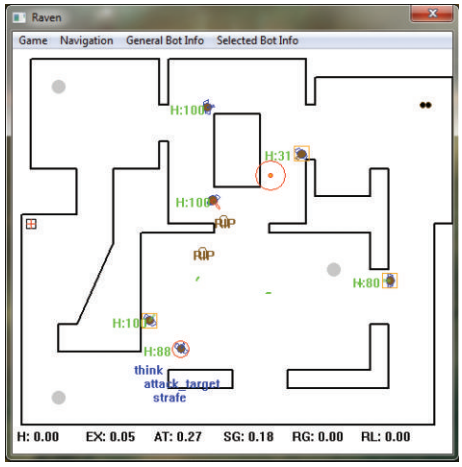

Figure 17: Raven with the modified targeting AI. The selected bot can be seen aiming at the enemy with low health (31), instead of the one close to it.

Figure 18: StarCraft: Brood War with the modified unit AI. The selected Goliaths are prioritizing Dragoons instead of the Archons in front of them because of their lower health.

was designed to be independent. AI modules can be compiled independently from game code and either linked to the game statically or dynamically loaded at runtime. An implementation using the latter option would benefit from easier testing of different AI solutions. When deployed, it would allow players to switch between different solutions too. This may not be desirable however, as untested solutions may result in unexpected behavior. A security mechanism could be added to prevent the game from loading unverified AI modules.

Perhaps the most exciting extension to this research would be a study of the world of conceptual problems found in video games. Both the video game industry and the scientific community would benefit from tools for describing and organizing problems using a set of convenient standards, perhaps a bit like game design patterns. This would help better categorize and hierarchically structure problems and result in a clearer view and understanding of the complexity of video games.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] The NetBSD Foundation, "Portability and supported hardware platforms," http://netbsd.org/about/portability.html.

[2] Microsoft, Windows NT Hardware Abstraction Layer (HAL), http://support.microsoft.com/kb/99588.

[3] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2003 report of the IGDA's artificial intelligence interface standards committee," Tech. Rep., International Game Developers Association, 2003, http://www.igda.org/ai/report-2003/report-2003.html, http://archive.org/web/.

[4] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2004 report of the IGDA's artificial intelligence interface standards committee," Tech. Rep., International Game Developers Association, 2004, http://www.igda.org/ai/report-2004/report-2004.html.

[5] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2005 report of the IGDA's artificial intelligence interface standards committee," Tech. Rep., International Game Developers Association, 2005, http://www.igda.org/ai/report-2005/report-2005.html, http://archive.org/web/.

[6] B. Yue and P. de Byl, "The state of the art in game AI standardisation," in *Proceedings of the 2006 International Conference on Game Research and Development*, pp. 41–46, Murdoch University., 2006.

[7] B. F. F. Karlsson, "Issues and approaches in artificial intelligence middleware development for digital games and entertainment products," CEP 50740:540, 2003.

[8] C. Berndt, I. Watson, and H. Guesgen, "OASIS: an open AI standard interface specification to support reasoning, representation and learning in computer games," in *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games (IJCAI '05)*, pp. 19–24, 2005.

[9] Unity Technologies, "Unity—Game Engine," http://unity3d.com/.

[10] Epic Games, Unreal Engine Technology — Home, https://www.unrealengine.com/.

[11] Crytek, CRYENGINE: The complete solution for next generation game development by Crytek, http://cryengine.com/.

[12] Havok, http://www.havok.com/.

[13] B. Kreimeier, The case for game design patterns, 2002, http://www.gamasutra.com/view/feature/132649/the_case_for_game_design_patterns.php?print=1.

[14] S. Björk, L. Sus, and H. Jussi, "Game design patterns," in *Proceedings of the Level Up-1st International Digital Games Research Conference*, Utrecht, The Netherlands, November 2003.

[15] S. Björk and J. Holopainen, "Describing games—an interaction-centric structural framework," in *Level Up: Proceedings of Digital Games Research Conference*, 2003.

[16] C. M. Olsson, S. Björk, and S. Dahlskog, "The conceptual relationship model: understanding patterns and mechanics in game design," in *Proceedings of the DiGRA International Conference (DiGRA '14)*, 2014.

[17] A. B. Loyall and J. Bates, "Hap: a reactive, adaptive architecture for agents," Tech. Rep. CMU-CS-97-123, Carnegie Mellon University, School of Computer Science, 1991.

[18] M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE Intelligent Systems and Their Applications*, vol. 17, no. 4, pp. 39–47, 2002.

[19] M. Mateas and A. Stern, "A behavior language: joint action and behavioral idioms," in *Life-Like Characters*, Cognitive Technologies, pp. 135–161, Springer, Berlin, Germany, 2004.

[20] J. D. Funge, "Making them behave: cognitive models for computer animation," 1998.

[21] J. Funge, X. Tu, and D. Terzopoulos, "Cognitive modeling: knowledge, reasoning and planning for intelligent characters," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 29–38, ACM Press/Addison-Wesley, 1999.

[22] J. Funge, "Representing knowledge within the situation calculus using interval-valued epistemic fluents," *Reliable Computing*, vol. 5, no. 1, pp. 35–61, 1999.

[23] J. Orkin, "Symbolic representation of game world state: toward real-time planning in games," in *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, 2004.

[24] J. Orkin, "Agent architecture considerations for real-time planning in games," in *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment (AIIDE '05)*, pp. 105–110, 2005.

[25] E. F. Anderson, "Scripting behaviour—towards a new language for making NPCs act intelligently," in *Proceedings of the zfxCON05 2nd Conference on Game Development*, 2005.

[26] E. F. Anderson, "SEAL—a simple entity annotation language," in *Proccedings of zfxCON05-2nd Conference on Game Development*, pp. 70–73, Stefan Zerbst, Braunschweig, Germany, 2005.

[27] E. F. Anderson, "Scripted smarts in an intelligent virtual environment," in *Proceedings of the Conference on Future Play: Research, Play, Share*, pp. 185–188, ACM, 2008.

[28] D. C. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE '04)*, pp. 36–40, 2004.

[29] D. W. Aha, M. Molineaux, and M. J. V. Ponsen, "Learning to win: case-based plan selection in a real-time strategy game," in *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR '05)*, pp. 5–20, August 2005.

[30] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Case-Based Reasoning Research and Development: 7th International Conference on Case-Based Reasoning, ICCBR 2007 Belfast, Northern Ireland, UK, August 13–16, 2007 Proceedings*, vol. 4626, pp. 164–178, Springer, Berlin, Germany, 2007.

[31] B. Weber and M. Mateas, "Conceptual neighborhoods for retrieval in case-based reasoning," in *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR '09)*, pp. 343–357, 2009.

[32] B. G. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09)*, pp. 106–111, October 2009.

[33] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram, "Transfer learning in real-time strategy games using hybrid CBR/RL," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*, pp. 1041–1046, January 2007.

[34] S. Lee-Urban, H. Muñoz-Avila, A. Parker, U. Kuter, and D. Nau, "Transfer learning of hierarchical task-network planning methods in a real-time strategy game," in *Proceedings of the 17th International Conference on Automated Planning & Scheduling (ICAPS '07), Workshop on AI Planning and Learning (AIPL)*, 2007.

[35] D. Shapiro, T. Könik, and P. O'Rorke, "Achieving far transfer in an integrated cognitive architecture," in *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI '08)*, pp. 1325–1330, July 2008.

[36] M. Buckland, *Programming Game AI by Example*, Jones & Bartlett Learning, 2004.

[37] B. Schwab, *AI Game Engine Programming*, Cengage Learning, 2008.

[38] I. Millington and J. Funge, *Artificial Intelligence for Games*, CRC Press, Boca Raton, Fla, USA, 2009.

[39] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2009.

[40] S. Rabin, *AI Game Programming Wisdom*, Charles River Media, 2002.

[41] S. Rabin, *AI Game Programming Wisdom 2*, Cengage Learning, 2003.

[42] S. Rabin, *AI Game Programming Wisdom 3*, Cengage Learning, Boston, Mass, USA, 2006.

[43] S. Rabin, *AI Game Programming Wisdom 4*, Charles River Media Group, 2008.

[44] R. Straatman and A. Beij, "Killzone's AI: dynamic procedural combat tactics," in *Proceedings of the Game Developers Conference*, 2005.

[45] D. Pottinger, "Implementing coordinated movement," *Game Developer Magazine*, pp. 48–58, 1999.

[46] bwapi—An API for interacting with Starcraft: Broodwar (1.16.1)—Google Project Hosting, https://code.google.com/p/bwapi/.