34

A Simple and Robust Knowledge Representation System

Phil Carlisle

34.1 Introduction

Knowledge comes in many forms, and how the knowledge is represented can have a significant impact on the efficiency and development time of any game AI. The representation chosen is particularly important for any game that involves complex characters. This chapter describes a system that was created for the game *EverSky*, an indie first-person exploration game for the PC and tablet devices.

The goal of the game was to push the boundaries of companion style AI and allow for a cast of characters that would be constantly alongside the player. This required that each character have deep knowledge of the world and each be capable of portraying complex attitudes to events, other characters, and objects within it. The complexity of the knowledge required led to a number of iterations on the design of how the data was represented, leading to a system that favored flexibility and iteration speed over raw efficiency, both in terms of memory and CPU usage.

34.2 Design Requirements

Before we begin, we note that increasing the longevity of characters also increases the requirements on the knowledge supporting the characters' behaviors. We focus on

long-lived companions that require a complex knowledge representation. It is often the case that nonplayer characters in a video game have a relatively short lifespan and thus do not require any form of complex knowledge representation.

That said, the guiding design principles for this system were:

- Minimize iteration time (sometimes at the expense of execution speed, we could always optimize later).
- Trade memory for efficiency (as a PC-based game, memory is not that restrictive).
- Easily integrate with scripts.
- Work flexibly with a component-oriented architecture (more on that later).
- Allow for varying types of knowledge and be flexible in structure.

If we think of logic as simple sets of `if (condition) do action` clauses, then we must consider what the condition part of that clause actually means. In order to obtain any form of complex behavior, the conditional part of any logical clause must contain any number of conditions, the complexity of the conditions being generally proportional to the complexity of the behavior. The key realization for anyone working on game artificial intelligence is that these conditions rely heavily on knowledge and that knowledge representation can form a large part of the requirements for any complex behaviors.

The knowledge representation for *EverSky* went through a number of iterations on the basic structure of knowledge. The first iteration involved a simple blackboard architecture that contained a number of get and set functions to add and retrieve information from the blackboard, which were also available to scripts. It soon became apparent that this was not a very well structured approach to the problem of knowledge representation for a character with complex behavior and high expected longevity. The problem was that having get and set methods for every single type of information was simply not possible. The amount of time required to add a new type of knowledge to the blackboard code was too great and added to the iteration time. It was time to go back and rethink the problem.

What should we consider as the basis for a knowledge representation system?

Knowledge generally consists of elements such as:

- entity attributes (position, velocity)
- existence
- classification or type
- set membership
- relationships/attitudes
- knowledge of others
- semantic knowledge
- knowledge of events

Let's examine these types of knowledge in more detail:

34.2.1 Entity Attributes

Entity attributes are simply data associated with a particular entity. It could be things like position, velocity, name, facing direction, current animation, etc. Each entity that a particular character senses (more on that later) needs to be stored along with useful attributes

of that entity. This information goes in an `Entities` array, which is a dynamic array of all entities a given character is aware of.

34.2.2 Existence

The notion of existence is very important as often we can gain efficiency by only considering objects or sets of objects about which the character has knowledge. For instance, conditional logic required to select the appropriate enemy to target during combat requires the knowledge that an enemy exists, before it is useful to consider knowledge for which enemy is the most desirable. In many ways it is useful to use the existence of something as the primary condition and then to drill down into secondary conditions, only if that primary condition is true. So in a typical situation, you may see an `is_enemy_seen()` method that returns true if an enemy is seen by the NPC, which then feeds into other conditions such as selecting from a list of seen enemies. This concept of "existence" can be easily represented by the count of any specific set membership, a nonzero value representing the existence of an enemy in the `enemies` set for instance.

34.2.3 Classification

The notion of "enemy" in this example is an interesting one. What we mean by "enemy" is that the particular entity belongs to a set of entities which elicit a specific reaction from our logic relating to that particular classification. But is it enough to simply classify entities as a single type? Imagine the case of Romeo and Juliet. Both could be classed as Montague or Capulet, but they could also be classed as male and female, they could be classed both as human, or they could both be classified as lovers. The problem is that any given entity may be part of a number of classifications depending on the context. Romeo can be classified as male, human, Montague, and lover all at the same time. If the number of classifications is relatively small, then it is possible to use a simple bit-flag representation and binary logic to represent them, but a more flexible approach is often preferable.

34.2.4 Set Membership

Sets can be represented as a simple dynamic array of member ID's. If we allow for an arbitrary number of sets, each containing an arbitrary number of members, along with a way to name the sets and lookup the members, then we can cope with any number of different classifications.

34.2.5 Relationships

Relationships are a slightly more complex version of set membership. In relationships we have a membership, as well as a positive or negative value on each member in the set or on the set overall. Consider the relationship between Romeo and Juliet again. In this case there is the overall relationship between Montague and Capulet (which implies that Romeo and Juliet are members of those sets), but there is an overall negative relationship value between those two sets.

But in the specific case of Romeo and Juliet themselves, they also have a positive relationship, which implies another set membership, where the members are the two lovers. Ultimately, the representation chosen for relationships is a simple one. Each membership set is represented as a named dynamic array of members, where members are stored using their ID value. Each entity stores the names of each set it is a member of in a dynamic array that basically forms a

hierarchy of membership from most to least specific. So the first element might be membership of the "lover" set, the next might be "Montague," and the final set might be "human."

Each membership set also has a `valence` value that can be positive or negative to denote the positive or negative associations with that set. The reason for this flattened hierarchy is that when considering a behavior we will often want to consider more specific behavior first, followed by more generalized behavior. If we store the set membership in this fashion, we simply need to read the array from left to right and consider the behavior for each set membership as appropriate. So for example, if we find we are in the set "lover," we might choose a display of affection. However, if we are not a member of this set, then we may be a member of "Montague" and subsequently select a display of dominance, with the final fallback of "human" allowing a default for when no other membership is available.

34.2.6 Attitudes

Attitudes are dealt with using the exact same structure as relationships (attitudes in this sense are just relationships to simple objects or events rather than characters).

34.2.7 Knowledge of Others

If we are creating behavior for companions, we need them to appear as though they also have a model of what those around them know. In this case, we can simply store a set of "companions" for each entity and then query their knowledge via lookup if required.

34.2.8 Semantic Knowledge

Perhaps the thorniest issue in knowledge representation is the concept of semantic knowledge. Suppose we have a character that likes fruit; as we are exploring, we come across an object we have not encountered before. It looks like a fruit that the character is familiar with, so the character decides to try and eat it. The question is, what made the character decide to try and eat it? It was the semantic knowledge that this new object was "like-a" fruit.

Terms such as "like-a" and "is-a" are interesting concepts. A potato is not a fruit and yet it is edible. It has an "is-a" relationship with the concept "food" in the same way that fruit has an "is-a" relationship with the food concept. In order to allow for inference of this kind, we need to consider representing semantic knowledge. We note that experience shows that the vast majority of game AI behavior does not require semantic knowledge like this; thus we will leave experimenting with this semantic knowledge as an exercise for the reader.

34.2.9 Events

These are records of any events relevant to the character; they might be events such as "just heard a grenade land" to "just seen Romeo kiss Juliet." Event knowledge is useful for rapid reactions to things that require immediate attention, and they are also useful as aggregate statistics that affect things like mood. They can be used for example to change a value that represents battle fatigue or to alter a value that aggregates combat experience over time.

To sum up, what is needed is a system that allows storing of arbitrary information in arbitrary data structures that nonetheless allows efficient retrieval and querying, while also allowing automated access to knowledge representations from within scripts and conditional statements from the decision logic.

34.3 Implementation

Each entity in *EverSky* is represented by a unique per-instance integer value, which allows for fast lookup of entities by using the entity ID as the hash in a hashtable. Entities themselves are represented using the `GameObject` class, which is a simple container for a flat list of `GameComponent` derived classes. For more information on component-based architectures, it might be useful to refer to Jason Gregory's excellent book *Game Engine Architecture* [Gregory 09], but in general you can think of components as classes which deal with specific functionality in a narrow scope.

The advantage of component-based architectures is that you can composite entities by simply adding or removing components and get quite different behavior. All entities in the game are composed at runtime from a named XML template that describes all the components of the entity. Levels are described in another XML file, which contains the entity template name, unique ID, and other relevant attributes for each entity in the game.

It might be useful to refer to the first *AI Game Programming Wisdom's* chapter on blackboards [Isla and Blumberg 02]. In general, a blackboard is a simple repository of data with corresponding access methods. Initially, when designing the knowledge representation for *EverSky*, a simple blackboard with a number of lists of named integers, floats, and ID values with member functions (used to retrieve information from each list) was used. In addition, it had a number of floating-point and integer values used to represent knowledge such as emotional state and counts of items owned, etc. The downside to this approach is that every new piece of information had to have some method of setting and getting that information added to the blackboard class, along with defining the interface for access by scripts, etc. This ended up being overly complicated and not particularly useful for the purpose of prototyping gameplay. Eventually, it was decided to use variant type data, which allowed arbitrary data to be stored, but also simplified the interface for both native code and script code. In reality there is always a trade-off between performance and accessibility and the use of variant data types was deemed to be a reasonable compromise.

In the knowledge representation system described in this chapter and available on the book's website (http://www.gameaipro.com), you will find a system that is based around the blackboard concept. However, it has been modified to appear very similar to a property-based system, in that it stores arbitrary named data in a variant format that allows querying of the data by name, but also retrieval of the data in any suitable format with reasonable conversion. The data itself is stored in variants which have a name string and a notional "type" with methods to convert to other common types. In practice this means that most data is stored as native types with no conversion, but has the advantage that common conversion such as string access for passing to scripts is trivial.

While this system is not as efficient as a simple list of integers or floats, it is efficient in terms of ease of use and automation of access; this trade-off is one that is appropriate for a system which is meant to change as the design of the behaviors adapts to gameplay requirements. The main caveat to this approach is that it is not always possible to convert from one type of data to another. For instance it is not possible to convert from any arbitrary string to a numerical format. The way *EverSky* deals with mismatched data type access is to use the POCO class libraries exception handling routines; however, in a nonprototype game context, or on a shipping title, it would make more sense to simply assert the data access and fix the code that is incorrectly accessing or storing data.

Because this was an indie project and in general we are supporters of open source code, the implementation is built using an open source variant implementation provided by the POCO C++ class libraries [Applied Informatics 12]. In particular the variant type `poco::dynamicany` allows the storage of variant data without type in much the same way dynamically typed languages allow variable storage.

The variant system used in the example code was extended in three ways. First, we added hierarchies of variant types. This means that each variant can store either a single value or a dynamic array of other variants, which enables any single data access to be able to access anything between a single value and a complete hierarchy as part of that access. Second, we added a "name" property to the variant such that variants can be queried by name, which is useful for script access. Finally, we added a method to describe the expiry value for the variant data. This expiry value is either zero to denote a nonexpiring piece of data, or any nonzero value, which is then counted down during the update of the blackboard. The variant system requires that any variant data access specifies the type of data that is returned through the use of a template parameter. The variant data is stored and accessed via the blackboard class.

34.4 Execution

In the case of *EverSky*, an AI component stores a blackboard instance within it, plus it allows for accessing other "named" blackboards. The AI component also stores a behavior tree instance, which has an `Execute` method called during update of the AI component. This execute method is passed a pointer to the AI component along with the elapsed time. As the behavior tree executes, it can access the blackboard via the AI component interface, as well as add actions for the AI component to execute. In this way, the decision logic of the behavior tree class is separated from the action execution logic of the AI component.

During the AI component update, the blackboard itself is updated with the elapsed time, which allows for the blackboard data to be cleaned up as expiring data can be pruned from the dataset. This expiry is simply a loop over the entities list. Values that are nonzero have the elapsed time subtracted from them and any that would lead to either a zero or negative value are considered to have expired and are removed from the list. Note that this expiry system allows for any individual piece of data to expire either at the aggregate level, as in the case of an individual entity and all its attributes, or at the attribute level for specific features of an entity such as position or velocity. During this update, another method allows for new event data to be aggregated using an `appraisal` class. This class is game specific and allows for aggregation of events and other knowledge generally useful for implementing features such as character emotions and moods.

34.5 Sensing

Data can be added to the blackboard during entity construction via an XML file reference or it may also be added in the game update loop via the sensory system. As each entity is sensed, it is added to the blackboard along with a number of common attributes such as position, direction, velocity, etc. If an entity is sensed and is already in the blackboard, the variant for that entity is updated with a new expiry time, if appropriate, along with any new attributes that may have changed.

34.6 Handling Events

Events are added as they are received from the central event handling system called `MessageManager` and are parsed into variant structures for adding to the `Events` variant array in the blackboard class using the `AddEventMemory` member function.

34.7 Accessing the Knowledge

Accessing the various representations of knowledge can either be by named element or by index values if a variant array is stored. Typically, access takes the form:

```
<type> variable = blackboard<type>[elementname];
```

where `<type>` is any supported data type and `elementname` is the string name given to the element.

In place of `elementname`, an integer index can be used for variants which are known to store arrays of other variants. The Boolean member function `IsArray()` can be used to determine whether the variant storage has an array stored within it. The unsigned integer member function `Count()` returns the number of elements in the array.

The reader is advised to consult the various `condition` nodes available in the source code for examples of accessing the various variant data types.

34.8 Debugging, Scripts, and Serialization

One of the primary benefits of the variant approach is that exposing the data to other systems becomes trivial. Each variant knows how to serialize itself to any given data stream, whether this be for debugging, logging, or entity serialization. So the process becomes a recursive one where you simply call a function on the topmost variant and allow it to serialize itself and its child hierarchies automatically. Because each variant has a name, it is relatively easy to understand what each value represents visually within the stream. For example, this feature is used within *EverSky* to serialize the blackboard to a JSON format for display in a web browser, which is accessed via an embedded web server. Script interfacing is similarly simplified, because a simple function can be written which exposes a given named variant to the script engine via a string conversion using the variant name to allow access. Variant types that store arrays are usually exposed as tables in scripting languages that support them such as LUA.

34.9 Conclusion

While developing the behavior for a game, reducing the iteration time required to get any single behavior implemented is of paramount importance. The knowledge representation system described builds on the concept of blackboards and adds a more dynamic form of data representation that more easily deals with changing requirements. While this allows for a flexible data representation, it does have tradeoffs in terms of performance that may not be desirable in a final shipping product. It is useful to point out that this system derives many of the benefits and pitfalls of dynamically typed languages. A good solution to the pitfalls is to optimize the representation once it is known what the final

behavior implementation requires and to allow type specific data accesses once this has been finalized.

Although we have not explored the use of semantic knowledge, this is perhaps the last area of development for *EverSky* that is required to build fully adaptive and believable characters. It is left as an exercise for the reader to consider what use of semantic knowledge might add to their own games.

References

[Applied Informatics 12] Applied Informatics. "Poco Project." 2012. Available at (http://pocoproject.org/)

[Gregory 09] J. Gregory. "*Game Engine Architecture.*" Boca Raton, FL: CRC Press, 2009.

[Isla and Blumberg. 02] D. Isla and B. Blumberg. "Blackboard architectures." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002, pp. 333–345.