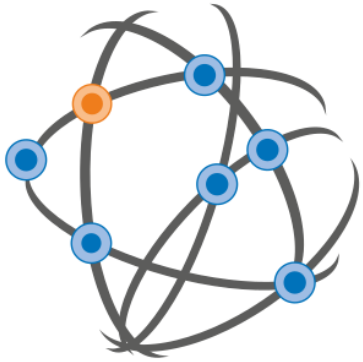
Direct AI

White Paper



masa
group

Direct AI White Paper

AI is thriving. No longer a curiosity, AI-based techniques are now a cornerstone of modern software, and increasingly hardware, development. There was a time when only biological intelligence was capable of adapting to its environment, by perceiving and reasoning, learning and planning. Today, the new generation of intelligent, AI-based, consumer products and applications is also capable of adapting and learning to their user and their environment, offering new services, and a better and unique experience.

This white paper introduces Direct AI, a technology engineers can use to model, design and implement adaptive software agents, with the aim of bringing their products and applications to life. MASA has been in the AI business for more than two decades, and Direct AI is one of its earliest technologies, used in games and many research projects. The artificial brains behind the simulated units of SWORD, MASA's simulation solution for the defense and security markets, are also implemented using the latest generation of Direct AI.

This document is aimed at technically-minded readers, and, more specifically, software engineers that are interested in learning about Direct AI from a technical perspective. However, the introductory sections, even if they necessarily contain some technical terms, are accessible to a broader audience.

What is Direct AI?

Direct AI is a software development kit (SDK), middleware clients can use to introduce rich AI-based behavior into their products and applications. More specifically, software engineers can use Direct AI to model, design, and implement the rich behaviors of **AI software agents** that **think** and **act**, to any degree of autonomy, within a product or application environment.

The Direct AI SDK comprises:

- A thin C, C++, and Lua framework allowing Direct AI to be integrated into any application. Usually, a basic integration of Direct AI takes around 1 (one) working day.
- A structured behavior design and implementation language that AI developers, the software engineers that will work with Direct AI on a daily basis, use to model, design and implement both the behaviors of AI software agents, and their environment (or world).
- A behavior authoring integrated development environment (IDE), with fully fledged navigation, edition, validation, and Lua debugging tools.
- A Lua profiler for troubleshooting.

Direct AI is not a newcomer in the industry. It is a mature, robust, heavily-tested, and optimized product that is in constant evolution. It is the synthesis of many years of experience crafting the rich and diverse behavior libraries for the defense and security markets, which are key components of SWORD, MASA's simulation solution. Direct AI has been designed from the ground up to grow. Its plugin-based architecture, and the module nature of Lua, allows us to extend the runtime with new features without bloating client applications with features they do not need or care about.

Basic principles

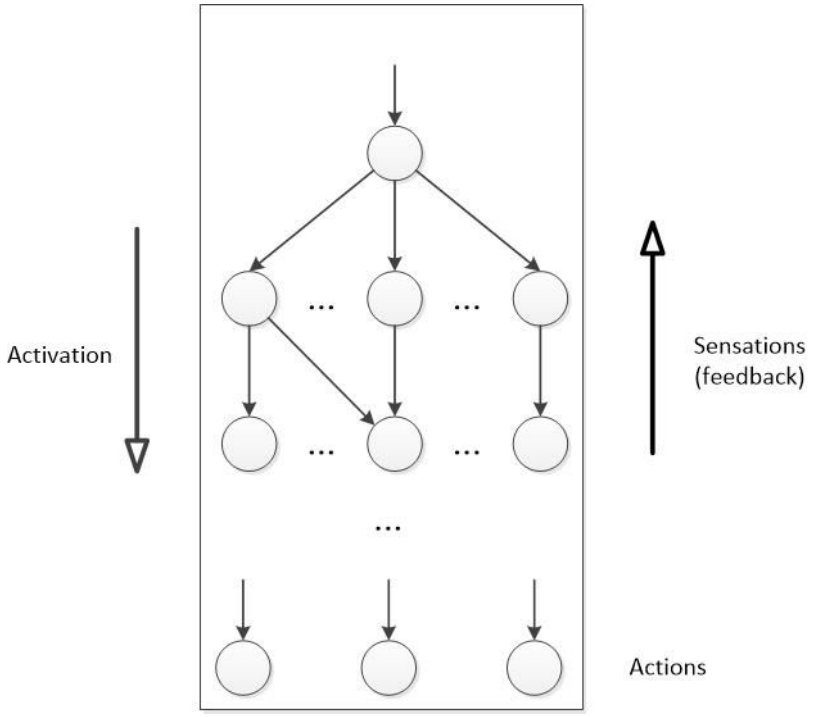
In a nutshell, Direct AI is a technology for engineering an agent's **brain**, the part of the agent responsible for its actions in the application environment. In fact, the problem for which Direct AI provides a design and implementation solution is that of **action selection**, that is, the problem of deciding what to do next; and, as we will see shortly, the way we reason, design, and express behaviors in Direct AI, its paradigm, is especially adequate when it comes to deciding on a course of action that is a **compromise** between many different, sometimes contradictory or conflicting, courses of action. A military soldier may be confronted with a behavior that suggests a number of positions for observing an enemy, and a different behavior, a safeguarding instinct, that suggests adopting stealth positions to avoid taking unnecessary risks of being seen.

A brain decides how to act on its application environment based on its **perception** and **knowledge** of its environment. We say the agent is **situated** in the environment, since its actions may not only change the environment in which it takes active part, they may also have an effect on the agent itself, and perhaps on other agents. This, in turn, might have an effect on the agent's future decisions. An agent thinks and acts in a changing, unpredictable, environment. The fact that the agent is situated means that the agent has a **partial** and **local** perception of the environment; it can only sense the environment around it, and may not know everything there is to know about it, just what it can learn about it. A virtual military soldier's brain cannot commonly see through walls, and requires special virtual equipment to keep in touch with fellow soldiers in its unit.

The shape of a brain

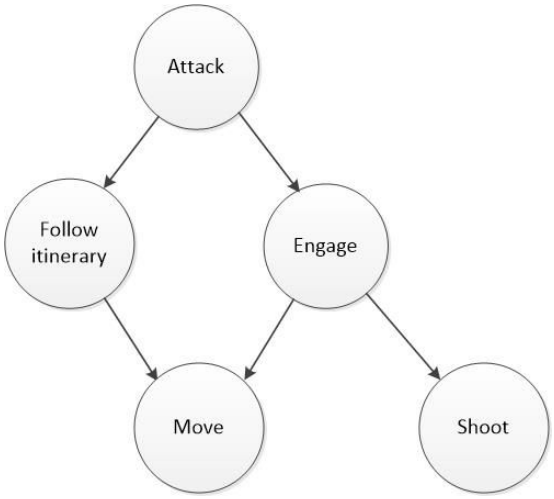
Inspired by biology, the overall shape of a brain is that of a network of cells, or **nodes** in Direct AI jargon, which form a hierarchy where nodes higher up in the hierarchy only excite, or **activate**, nodes lower down in the hierarchy. The node is the simplest, most basic, form of behavior in Direct AI. The lowest level nodes are always **action** nodes: they carry out the actions that modify the application environment. A soldier may have actions like walking or running to a specific location, grabbing a weapon, aiming or shooting at a specific target, or simply becoming stressed, which is an example of an action that affects the (internal) state of the agent itself. The overall behavior of the agent takes place when one or more of the nodes at

the top is activated by the application. When activated, a node decides, or **thinks**, about which lower-level nodes to activate in turn. This process continues until action nodes are reached.



As illustrated above, at any stage in the process, a node may rely on **sensory** information, perhaps obtained upon request before the application, or any other knowledge gained or just given, to carry out its behavior. Information, carried by activations, is not restricted to propagating downwards in the hierarchy; it can also propagate upwards. A node may receive feedback, or **sensations**, from the nodes it activates, which allows the node to adapt accordingly. If a soldier agent decides to shoot, but a lower-level node finds out it does not have the appropriate equipment, it has a way of notifying its caller, allowing the caller to take an alternative course of action, or simply propagate the problem up the hierarchy. Because information can flow freely in a brain, the paradigm of action selection used by Direct AI is known in the literature under the name of **free-flow hierarchy** (FFH).

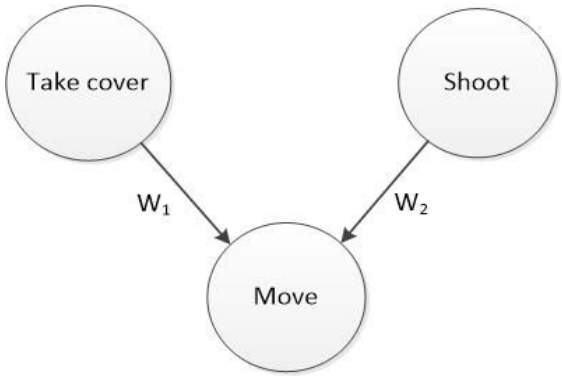
A typical FFH would look as follows:



In this example, the goal of the *Attack* behavior (node) is to engage enemies positioned along an itinerary. Both *Follow itinerary* and *Engage* activate the *Move* action to effectively move the agent engage positions along a given itinerary (which can be a component of the *Attack* activation). Finally, *Engage* activates the *Shoot* action to perform the actual shooting.

The power of compromise

Unlike most of the other commonly used paradigms in the industry (briefly mentioned in the following section), when activating an FFH behavior node, the activator or caller can choose to specify a **preference** in the form of a numerical **weight**. The default weight is 1 (one) if none is specified. Leaving out the details of how weights are propagated down the node hierarchy, the important application of activation weights is that, whenever there are two conflicting actions, the Direct AI runtime will choose the one with the highest preference. For example, suppose we have two nodes, one implementing the (instinctive) behavior of keeping out of sight of enemies (by always preferring a cover position), and another that, on the contrary, will seek the best position to shoot at an enemy (possibly at the risk of exposure). Using FFH, instead of resolving the issue explicitly in Lua, we can use preferences to let FFH resolve the issue by itself. When implementing both nodes, we just activate the action node that will move the agent to, respectively, a cover position or a shooting position. Even if the preferences are the same (for example, the default one), because preferences propagate down the hierarchy, if the instinctive behavior of seeking cover has a higher preference (w_1 in the diagram below), the cover position will be selected; otherwise, the shooting position will be selected.



In this very simple example, we have the choice between two, possibly contradictory, positions; it is one or the other, black or white. If, more interestingly, we are interested in the agent finding a **compromise** between these two possible extremes, we can activate the action nodes many times in a row, with one or more candidate positions, and introduce a preference value that is a measure in one case of how suitable a position is for taking cover, and, in the other case, how good a position is for shooting. If the positions are chosen from a discrete set (that is, they are not completely arbitrary), and the measures are comparable (for example, they both measure distances with equivalent units), the best common candidate position that is *both* good as a cover *and* a shooting position will be selected. In practice, since specifying weights requires some discipline, and has an impact on performance (since many node instances must be considered for selection), one uses this form of compromise sparingly. However, if used correctly, and performance is not an issue, it can be used to implement rich **adaptive** behaviors.

Other paradigms

There are other paradigms in use by the industry to develop situated agents. Even if they are all competing paradigms, in the sense that they are all alternative solutions to the action selection problem, each has its own strengths and weaknesses. The most commonly used are, in order of their expressivity: decision trees (i.e., if-then rules), finite-state machines (FSM), and behavior trees (BT).

All paradigms, when used in non-trivial projects, are introduced in a hierarchical manner, which in this case means that a behavior is not implemented using a unique FSM (or BT), but a collection of FSM (or BT), where each behavior is responsible for a particular aspect of the behavior, as is the case with FFH. The hierarchical nature is needed to tame complexity, to be able to reason about a complex behavior in terms of simpler behaviors, and allow for the reusability of common patterns of behavior.

Traditionally, the FSM, BT, and FFH, have natural graphical representations, even if not all implementations propose graphical editors. BT is relatively new; like FFH, it subsumes FSM (i.e., there is a simple transformation from FSM to BT or FFH), but is more expressive than FSM since it allows an agent to consider several courses of action simultaneously. When used in

non-trivial projects, most implementations of any paradigm introduce some form of scripting, or introduce some language of some sort to express conditions and actions. FFH nodes are essentially scripted; some BT nodes are scripted, but BT-specific nodes (sequence, parallel, conditional, and so on), are not; and, FSM proposals may implement transitions and actions using some form of scripting.

BT permits a more compact, and more user-friendly, graphical representation than FSM, due to its rich catalog of behavior patterns, that applications may enrich as they see fit. However, implementing standard programming logic graphically for non-trivial trees is less readable than using the traditional textual notation developers are accustomed to (and learn about at school). Our experience using FFH has shown that behaviors that are expressed as, for example, sequences of simpler behaviors, are better and more compactly expressed using specific (sequence) nodes, and can be grasped at a glance if represented graphically. This is the direction we have taken with Spark, introduced below. The idea is to allow behavior developers to decide where they draw the line between a behavior that is better expressed using a specific syntax that provides a readable graphical representation, or a complex behavior node whose details are better expressed in Lua, and should be hidden from the graphical representation.

Finally, FFH implements an action selection mechanism where conflicts between competing actions are transparently resolved by selecting the action with the highest preference value. This allows behavior developers to implement interesting and rich patterns of adaptive behavior.

Introducing Spark

Engineering a brain, especially one offering a large variety of behaviors, requires a sound and structured approach. Direct AI brains are networks of nodes, and as the number of behavior nodes increases, it becomes critical to adopt rules and conventions of good practice to be able to tame complexity.

For example, a *Patrol* behavior may require one or more network nodes for its implementation. A distinguished node, the top node, is the one to activate first for the agent to start behaving, whereas the other nodes are conceptually support nodes, implementation details, and therefore not intended to be reused or activated by other nodes elsewhere in the brain. A convention of good practice would suggest the *Patrol* behavior be stored in a Lua file named *Patrol.lua*, possibly in a folder whose name provides a hint about the sort of behaviors stored there, such as *military/platoon*. Also, the convention would also specify that the top node should be named in such a way as to avoid name clashes, like *military.platoon.Patrol*; and, also importantly, to name the implementation nodes in such a way that their relation to the top nodes as implementation nodes is clearly stated. In other words, non-trivial behavior libraries cannot be engineered without introducing some sort of structure, governed by sound rules that avoid common pitfalls during behavior development. If this sounds like common sense, it is just an example among many.

One step up the ladder

Direct AI 5, the new generation of Direct AI, is all about stepping one level up the abstraction ladder. It introduces Spark (as in a “spark of life”), a novel extension of Direct AI that embodies a decade of experience developing brains using the free-flow hierarchy (FFH) paradigm. Spark is a technology comprising the following features:

- The **Spark language**, a new language of behavior design and implementation. All Direct AI 5 developers use Spark now to engineer brains, as it allows them to concentrate on behavior design and forget about the low-level details of FFH network construction.
- An **integrated development environment** (IDE), based on the Eclipse platform, with new advanced edition and debugging perspectives.
- The **Spark Lua API** and **Spark core library**, that behavior developers and integrators can use to interact with the new Spark entities. For example, behavior integrators use the Spark Lua API introspection capabilities to discover about the behaviors available for an agent.
- Early, development-time, **validation** of behavior library contracts; and, optionally, runtime validation of type constraints.

A structured approach

The Spark language introduces new structuring and high-level behavior design constructs known as **entities**. They can be classified into three groups:

- the **skill**, **task**, and **resource** entities, are used to design and implement behavior;
- the **class**, **interface**, and **query** entities, are used to design and implement the environment, or world; and, finally,
- the **role** specifies the agent’s characteristics;
- the **category** entity stands for a group of related entities, gathered together under a category name for organization purposes. It is the equivalent of the namespace in programming languages.

With Spark, behavior developers model, design, and implement behaviors, and their environment or world, at a higher level of abstraction than that of FFH networks and pure Lua. When troubleshooting, behavior developers must be aware of the fact that Spark brains are FFH networks, but they do not need to build the networks by hand themselves as was the case in the past.

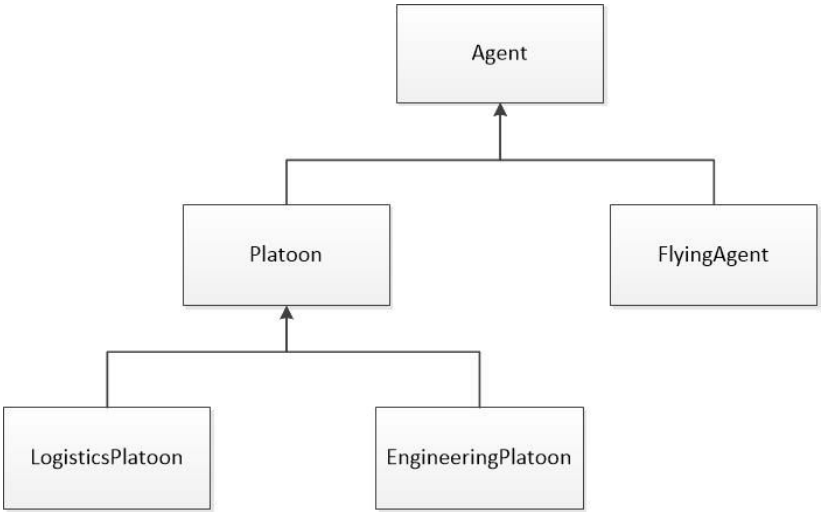
We explain the meaning of the entities in more detail in the following section, where we discuss behavior design and implementation using Spark.

Behavior authoring

Since Spark was introduced in Direct AI, all behavior authoring takes place inside Direct AI's integrated development environment (IDE). A key technical aspect of our framework is that, once Direct AI has been integrated into an application, the behavior author does not need to bother generating a new version of the application each time the AI changes. Because Spark behaviors are written in XML and Lua, behaviors are already executable. Direct AI will construct and optimize brains at runtime. Why have we chosen Lua as Direct AI's behavior implementation language? Because Lua is a lightweight, simple, easy-to-learn language, and, most importantly, it is one of the fastest languages of its generation. (Users may decide to pre-compile Lua sources beforehand, but parsing is so fast it is in practice just not worth it.)

The product a behavior author delivers is a **behavior library**, a collection of XML and Lua sources neatly organized into **categories**, where each category corresponds to a folder in the file system. (The approach is akin to that of the Java programming language.) A behavior library may, of course, reuse behaviors in other behavior libraries. In fact, all behavior libraries implicitly depend on, and so may reference entities from, the Spark core library.

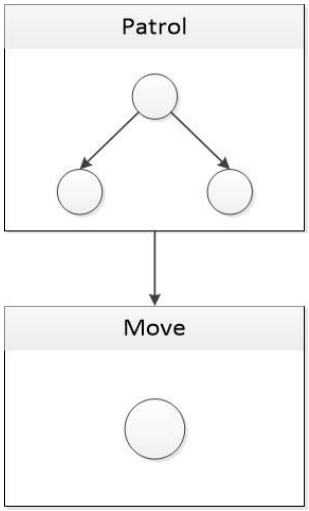
A behavior library may introduce one or more agent **roles**. A role specifies the agent's characteristics, the most important of which is the type of the agent's **body**, which can be a class, an interface, or an intersection of interfaces. (The type language of Spark includes **intersection types**.) The brain **controls** the agent's body, and through the body, it can modify its environment. Whereas the role is a behavioral entity, the body is not. Many roles can specify the same body type, differing in other characteristics; however, in general, when a new role is introduced it will most likely specify a different body type. A role can inherit characteristics from a parent role, including its parent's body type. Roles therefore form a hierarchy as well, known as the **agent typology**. The following is a typical example of agent typology:



When instantiating a Direct AI brain, one must specify its role and a body, which is an instance of a class. The body class is the place to define the state of the agent, as well as the action methods ultimately invoked from within the action behaviors. The role therefore puts restrictions on the capabilities the agent can effectively perform. In this sense, it establishes a **contract**, that Spark will be able to check at runtime.

Two kinds of behavior

Behavior entities in Spark come in two kinds: **skills** and **tasks**. A skill is a behavior implemented as a Direct AI network of nodes (technically, a sub-network of the final FFH network). For example, a *Patrol* skill, could comprise three nodes, where the top node would be the behavior's **root node**, the activation entry node, and the other two would be support nodes, which, in Spark, are local to the skill, and hidden from the rest of the behavior library. This is how Spark achieves encapsulation in the context of an FFH. All three nodes are conventionally located in a single file, *Patrol.lua*.

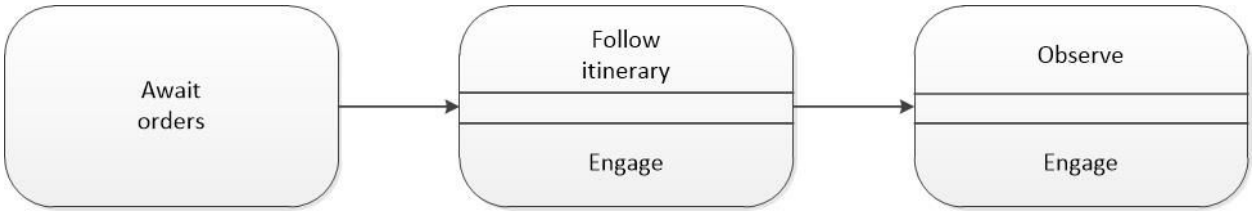


The contract of the skill, that is, its interface, which specifies how to invoke the skill from other behaviors, is written in a file *Patrol.xml*. The skill interface, among other properties, specifies the skill **parameters** and their types, since behaviors in Spark are all parametric. The explicit contract allows users to find out about the skill (and read its documentation if given), without having to delve into the details of its implementation, and it allows Spark to check skill activations at runtime.

The other kind of behavior, the **task**, is really interesting. Our experience has shown that many behaviors can be expressed as a **sequence**, where each stage in the sequence corresponds to one or many skills activated simultaneously. The sequence is actually so common that we thought it deserved a special syntax (as is the case, for example, in a behavior tree). Of course, it is not difficult to write a sequence using a skill in Lua, but the result is a lot more verbose, and,

therefore, much less explicit. A task is written entirely in XML using the task editor in the IDE, and because the task editor is a special-purpose graphical editor, no knowledge of XML (and perhaps programming in general) is required.

The following diagram illustrates a task with three stages. Once the agent has received specific instructions from its superior (first stage), it will follow an itinerary, perhaps engaging any enemies it finds along the way until it reaches a destination position (second stage). Then, the agent will hold the destination position, engaging any enemies that might get close to it (third stage).



Perception and the world

As the philosopher Emmanuel Kant would have asserted, to be is to do. An agent comes into being when a behavior is activated with the required parameters. A behavior parameter may be a simple value, a composite value like a Lua array, or a Spark class instance or **object**.

Lua does not have classes or interfaces, so Spark has introduced special entities to be able to define them. An **interface** is just a collection of method declarations, entirely written in XML; and a **class** is a collection of method and attribute declarations, and optionally a constructor declaration, where the class interface is written in XML and any (non-abstract) method implementations are written in Lua. Lua is not statically typed, but Spark is, so all method declarations must bear types. A Spark interface may be an extension of another Spark interface, inheriting any method declarations, and allowing for method overriding. A Spark class may be an extension of another Spark class, and may choose to implement one or many interfaces. (Both classes and interfaces work as in the Java programming language.)

The body, when it exists, is a Spark object; and, in general, **perception** information is modeled using Spark classes and interfaces. For example, if an agent perceives other agents, we could have a class somewhere called *Agent* denoting a perceived agent. Whether the *Agent* class contains some knowledge of the actual agent (like its position, state, and so on), or is the actual agent object of the application itself, is up to the behavior author.

The behavior author also has to decide which perception mechanism is the most appropriate. For performance reasons, Spark does not force upon the behavior author or integrator any technique in particular. A very simple technique is the **blackboard**. Because the behavior nodes in a Direct AI brain are implemented in Lua, their execution takes place in a specific Lua

environment. In fact, a brain has access to two **Lua environments**: its own private Lua environment (also called the **brain environment**), and a shared environment (with other brains in the same group). Any perception information can be placed, by the behavior integrator in either the private of the shared Lua environment via a simple C/C++ API call.

Another perception mechanism relies on **queries**. A query is a Lua function that is implemented as a named entity in a behavior library (like the stored procedure in a relational database). A query is called like any other Lua function, except that because it has a contract (defined in XML), it can be checked at runtime. Queries are also interesting because they can be used inside tasks. A query stands for a request to the application to return perception data (although it can be used for other purposes as well). However, any Lua function or method that polls the application for knowledge about the environment is a perception function.

Since both actions and perceptions take place by ultimately calling application-bound Lua functions, the collection of these functions is an API that establishes a contract between the behavior integrator and the behavior developer or author. It is what we call the **integration layer**, as detailed below.

Behavior integration

The Direct AI SDK contains the C/C++ framework that software engineers need to integrate Direct AI into an application. Usually, a basic integration of Direct AI takes around 1 (one) working day. It is not required that the host application be itself written in C/C++ to be able use Direct AI. The framework comes with C bindings to facilitate the integration using other languages, such as C#, Python, or any other language with a C native interface. In fact, our demos are written in C# using the game development framework Unity.

Integration tasks

In a nutshell, the integration of Direct AI into an application requires the application developer to:

- Add autonomy, to any desired degree, to an application by attaching to it one or several Direct AI brains. For example, a character in a game or simulation can become autonomous or semi-autonomous by having a brain attached to it that will take control of its actions. A brain can also be used to control a whole set of characters, like a crowd. But brains need not control only virtual agents; a brain (or brains) can be used to control a robot or connected object. Finally, a brain can exist disembodied, like an assistant that provides hints to application users based on usage information. The possibilities are endless.
- Let brains think and act timely. For example, agents may be allowed to behave every 100ms, instead of at every tick of the clock. It is also not necessary for agents to behave

in every situation; an agent whose actions are not observable (for example, because it is far away), does not need to waste priceless battery power thinking.
- Introduce an **integration layer**, or interface between the application and the brain behavior nodes, in the form of a specific, application-bound, Lua API allowing nodes to ask for **perception** information or perform **actions** on it. Technically, both requests for perceptions or actions are function (or pseudo-method) calls to the Lua API. A perception, or any other information the application passes to the brain via the activation of nodes at the top of the hierarchy, are Lua values, which may include pseudo-objects.

Direct AI adheres, by design, to the good engineering practice of considering brain design and integration as separate concerns. Brain logic can be reasoned and can evolve in isolation from the rest of the application logic, as they only interface as specified by contract in the **integration layer**. Integrators do not need to know about the details of behavior development, and behavior developers can forget about the implementation details of the application.

In practice, for performance reasons, and other considerations, behavior development is a close collaboration between the integration and the behavior developer.

About the future

Direct AI is a never-ending story; and we could never have predicted the way this story would unfold. The shape the product is in today is the result of our experience using Direct AI in our own products, research projects, and, recently, in the hands of clients in other companies.

Our commitments

Instead of just showing the list of features that will be released in future versions, which is beyond the scope of this white paper, we will summarize the guidelines that drive the future development of Direct AI:

- The integration of our AI into a client application should be simple and require as little work as possible. An example is our all-in-one C API for facilitating the integration using languages other than C++.
- AI developers should have the necessary tools to streamline daily AI development. Most of our investment is dedicated to the Direct AI IDE, and more specifically, to the development of better behavior editors that help developers express their intent more easily and efficiently. Also, facilitating navigation, searching, and visualization of behaviors is essential to our goal of offering a better, and more productive, behavior authoring experience.
- AI developers should have the necessary tools to help them resolve issues. Perhaps the best example is the Lua debugger that ships with the IDE. Also, the Lua profiler keeps getting better. Better than resolving issues is having the tools to spot errors early during

development. This is one of the main motivations behind the newer versions of Spark, with better validation tools.

- Our AI framework, especially our behavior design language Spark, should be as simple and easy to learn and adopt as possible. Indeed, newer versions of Spark are smaller than previous ones, but their expressive power has grown; and this is the trend we would like to commit to for future versions. We had in the past a lot of ideas we thought were useful, but experience has shown that they were either difficult or annoying to use, possessed very little added value, or, simply, clients did not adopt them. These features ended up being deprecated and removed from the framework.
- Performance should never be neglected. Newer versions of Spark are faster than any previous ones, and it is now possible to use more than one processor to boost the evaluation of brains. Performance is important since a fast AI supports more agents, or smarter agents, than a slow one.