10

Building a Risk-Free Environment to Enhance Prototyping

Hinted-Execution Behavior Trees

Sergio Ocio Barriales

10.1 Introduction

Working on game technology is an iterative process. From game to game, we try to reuse as many systems as we can, but this leaves us in a situation where few substantial changes can be made to our already proven and solid solutions. At the same time, the creative nature of games is craving for changes, prototyping and testing new ideas, many of which come during production or even near the end of a project, when the risk of breaking things is at its peak. Can we do something to offer a risk-free environment to work on those potentially game changing ideas, or should we let them go?

Hinted-execution Behavior Trees (HeBTs) try to address this problem. The technology is an extension to the traditional behavior tree (BT) model that allows developers to dynamically modify the priorities of a BT based on some high-level logic; the new layer works in a plug-and-play fashion, which means it can be easily removed, leaving the base behavior untouched. This greatly reduces the inherent risk of changes.

In this chapter, we present the technology—which is a proven solution, successfully applied to the AI in *Driver: San Francisco*—then study how it works and show how HeBTs can be applied to real-world problems.

10.2 Explaining the Problem

Video games are a type of software that benefits from changes, and prototyping is necessary to develop fun. Building such experiences is a joint effort between programmers and designers. This is a two-way process: designers come up with ideas that are transformed into technology by programmers, but this technology, at the same time, refines the original idea and converts it into something feasible. This back-and-forth iterative process shapes what is going to be in the final game.

Let us focus in the technological aspect of the process. At a high-level, programmers produce black box systems with some tweakable parameters; game or level designers will use these black boxes to build what is going to be the final game. This workflow, which is shown in Figure 10.1, is very commonly used in the industry: ideas come from designers, who request the feature from engineering; after some implementation time, the new pieces are ready to use by design.

The key property of this type of process is that designers have very little control over what the new "box" is doing. In many situations, this is desirable, as programmers are the ones with the technical capabilities and designers do not need to worry about implementation details. However, it is worth noting that going from conception to being able to use the feature can be a long process and any change to a black box will restart the loop (i.e., generate another request that engineering will process and implement). Ideally, we would like design to be able to test or prototype new ideas faster, without going through engineering.

Although this scenario mitigates the potential long delays between conception and actual availability of features in game, it will most likely not be a feasible solution in most situations. The new workflow, as shown in Figure 10.2, could require working with very

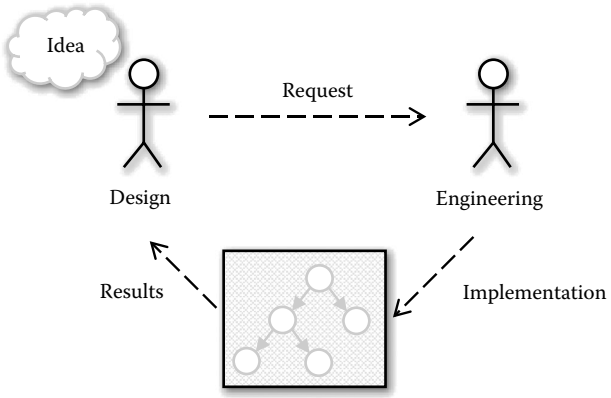

Figure 10.1

Traditional design/engineering collaboration workflow.

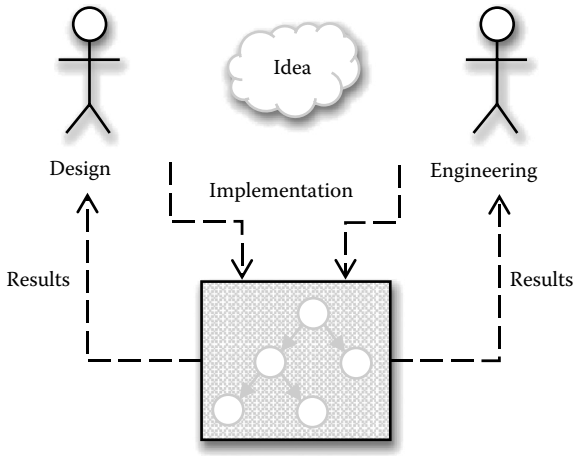

Figure 10.2

In an ideal world, both designers and engineers should have similar privileges when it comes to implementing and testing new ideas.

technical nonengineers that can manage, for example, modifying our BTs directly, which can be risky and produce more problems than benefits.

Another factor we have to take into account is that due to the complexity of video games, often new games use and improve previous games' codebases. This means we must be able to apply our solution to already existing code and technology.

The solution presented in this chapter, HeBTs [Ocio 10], is an extension to the traditional BT model. It tackles these problems and allows for fast and safe prototyping.

In the following sections, we will show how we can modify and manipulate an existing BT implementation to allow an extra high-level decision-making layer to dynamically change the priorities of certain sections of our behaviors.

10.3 Behavior Trees

The popularity of BTs has been growing steadily in the last 10 years and they have become a fundamental part of many games. In this section, we will cover the basics of what a BT is, as this knowledge is required to understand the rest of the chapter. Readers wanting to get better descriptions or some extra details about how a BT works should refer to [Isla 05, Champandard 08, Champandard 13] or the various materials available at AiGameDev.com [AIGameDev 15].

10.3.1 Simple BT

BTs are data-driven structures that can be easily represented in the form of a tree or graph. Nodes in a BT can be leaves or branches. Leaf nodes represent either an action or a conditional check, that is, nodes with a direct communication to the world. On the other hand, branch nodes do not perform any action but control the execution flow. In this category, we find nodes for control flow (selectors, sequences, parallels, etc.) or decorators.

Let us use a simple example, shown in Figure 10.3, to study how a BT works.

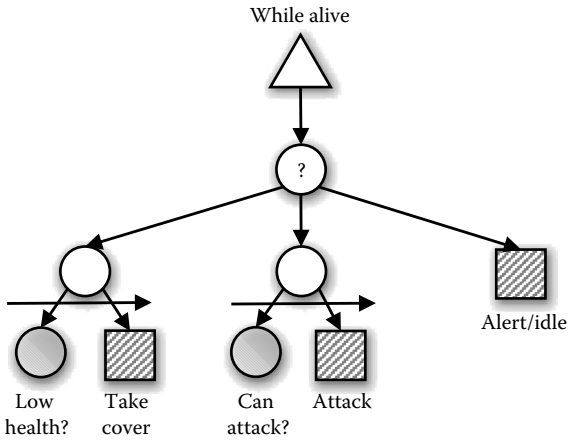

Figure 10.3

A simple BT.

In our example, we have modeled a very simple soldier behavior that will make the AI go and take cover if it is low in health, attack if it can, or just be alert (or idle) in any other case.

When a node is updated, it always returns a status value: "success," "failure," or "running." Return values are handled by parent nodes, which then decide what to do with this information. Thus, execution flow in a BT comes from its static structure (i.e., different branches will be activated in different orders all depending on what type of nodes we have used).

The most important node of the example tree is the selector (represented in the figure with a question mark). Selectors allow us to represent conditionals in a BT; translating it to what we could do in a programming language such as C++, selectors are the BT version of an "if-then-else" construction. It is this node's responsibility to decide what the AI is going to be doing at any given point. Priorities in the selector used in this example come from the order in which children of the selector are defined, that is, the leftmost node/branch is the highest priority and the rightmost, the lowest, resulting in static priorities.

Selectors try to pick the best possible child branch by testing each of them until one succeeds. In the example, we have represented two of the branches as sequences. A sequence is a special type of node that will run each of its children in order, succeeding if all the children succeed or failing otherwise. Due to this, sequences will very frequently be testing some preconditions as their first tasks and, if the preconditions pass, the actual actions can and will be run.

In the figure, our sequences have two children each: the first node is a condition node, or node that is just checking facts in the world; the second node is a proper action, or node that makes modifications in the state of the world.

Finally, we added a filter to our tree, which in the example is actually the root of the BT. The filter makes sure the behavior keeps running as long as the AI is alive.

For the sake of simplicity, we will continue using the example studied in the previous section, but the concepts presented in this chapter can be applied to BTs of any size.

10.3.2 Tree Complexity

The tree we are studying is very simple—it only has a handful of nodes—and making changes to it would be pretty straightforward. The problem is that, in a real-case scenario, trees can have dozens or hundreds of nodes and are not as easy to modify. There are some solutions to this, most of which involve having a good tool that allows us to work with the trees more easily (e.g., by expanding/collapsing branches or other UI improvements), but this does not remove the inherent complexity of the structure we are creating. Understanding the implications and side effects that a change might have in a complex tree is not trivial.

Going back to our example tree in Figure 10.3, let us say that at some point, we decide to bump the priority of the "attack" branch, because we want to model very brave (or, should we say, suicidal) soldiers that never retreat to take cover. In that situation, we would have to modify our tree, but that will make the "take cover" branch pretty much useless. What if, instead, we decide to only increase the priority of the attack in some circumstances?

Just by using the basic model, we can achieve this in a few different ways, like adding a new selector and duplicating some parts of the tree or by adding an extra precondition to the "take cover" branch, which is not that bad.

But, what if we would like the NPCs to attack for just 5 s then retreat and only do this for some specially flagged NPCs? Things can get complicated. In this case, we could end up with a tree similar to the one shown in Figure 10.4.

There are probably better ways to reorder the tree to accommodate for the new case, but the one presented in the figure is good enough to prove our point: modifying a BT to accommodate for new logic requires some thinking and always carries a risk.

In this chapter, we want to focus on a case like this one, particularly those in which we have to work with large BTs and where big changes are discouraged by deadlines or production milestones, but yet we need to keep iterating on our systems.

10.4 Extending the Model

BTs are a great technology, but they require good technical skills and hours of thinking to maintain them. The example of a change in the logic of a tree, studied in the previous section, showed how small changes in a simple tree can become difficult to understand very quickly. Logic changes are scary and, potentially, something we want to avoid, pleasing producers but hurting creativity.

In this section, we present a solution for these problems: HeBTs. The idea behind this extension is to allow extra layers of higher-level trees to run concurrently with our main tree and have the new logic dynamically modify priorities in the main BT.

10.4.1 Hint Concept

The main difference between a BT and a hinted-execution counterpart is that, while the execution flow in the former is defined by its own structure, HeBTs can reorder their branches dynamically to produce different results.

The system tries to imitate real-life command hierarchies, where lower-levels are told by higher-level ones what should be done, but, in the end, deciding how to do it is up to the individuals. In our system, the individual AI is a complex BT that controls the AI,

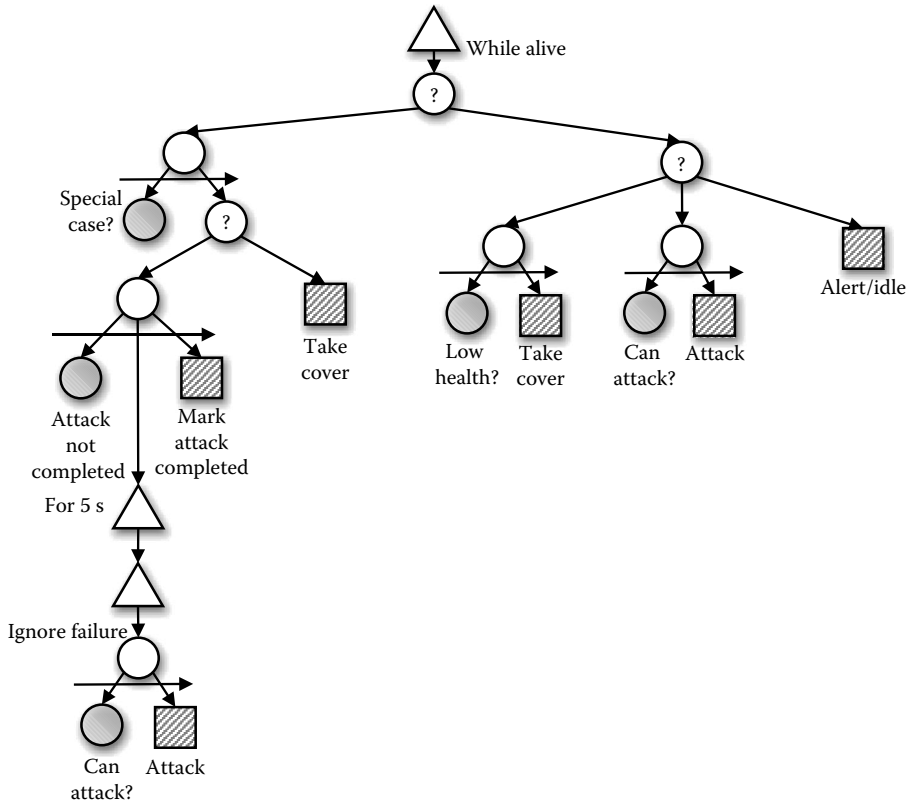

Figure 10.4

A more complex BT that adds special cases and extra checks.

so it behaves autonomously, but we want to open the higher-level layers to every member of the team, so they can test their ideas.

Nontechnical people will probably not be interested in how the AI works internally and will just want to tell it to do things (i.e., they just want to be able to order the AI to "kill an enemy," rather than "find a path to your enemy, then get closer, draw your weapon, and fire at your target, reloading your gun when you need to, etc."). These suggestions are called *hints* in the HeBT model.

A hint is a piece of information an AI can receive from a higher-level source and use it to produce an alternative behavior, as a consequence of a priority reorder. This means that an NPC, while maintaining its capability to respond properly to different situations, will take into account the requests coming higher in the command hierarchy to adapt its behavior to these petitions.

10.4.2 HeBT Selectors

Most of the decision making in a BT takes place in its selectors, which try different possibilities based on some priorities until a match is found.

In the simplest implementation of a selector, the priorities normally come from the order in which the branches were added to the selector node. So this means all of our priorities are *static*. HeBTs allow developers to change those priorities dynamically, resorting the branches associated to their selectors based on the information that a higher-level piece of logic has passed down to them. In order to do so, HeBTs introduce a new type of selector node.

Selectors, as composite nodes, have a list of children subbranches, each of which represents a possible action that a higher-level will, potentially, want the node to choose. We will talk further about these higher levels later on. In our new selectors, branches are assigned a unique identifier, which is assigned at creation time. This allows designers/engineers to name the branches and therefore create the hints that will favor each branch's execution.

Hints can be positive, negative, or neutral; if a hint is positive, the tree is being told to do something; if negative, it is being told not to do something; and, if neutral, the selector is not receiving the hint at all. Neutral hints are used to reset a priority to its default value.

The system works as follows: the AI is running a base BT (i.e., the one that contains all the logic for our game) and it can receive hints from a higher-level source. When a hint is received, the BT passes the information to all its selectors. The selectors will then recalculate the priorities of their branches.

For example, let us say we have a selector with five branches, named "A," "B," "C," "D," and "E." We have just implemented a HeBT system and our higher-level logic is telling us "D" and "E" are very desirable, but "A" is something we really should try to avoid. Figure 10.5 shows how the new selector would use this information.

The new selectors maintain four lists to control their children and their priorities. The first list just keeps track of the original priorities; the three extra lists store nodes that have been positively hinted (and thus, have more priority), nodes that have not been hinted (they are neutral), and nodes that have been negatively hinted (they have reduced priority). These extra lists are still sorted using the original order, so if two or more nodes are hinted, AIs will know which action is more important according to their original behavior.

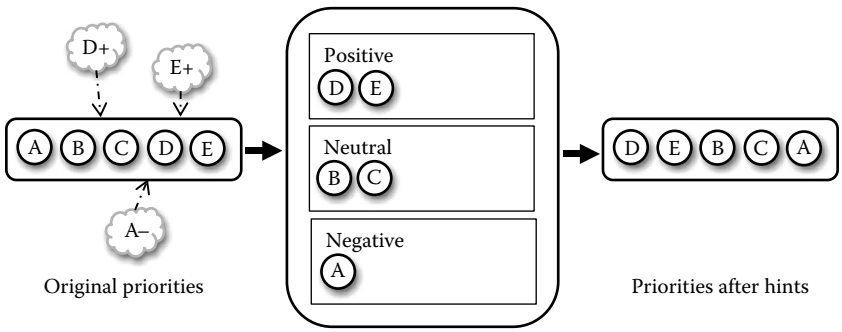

Figure 10.5

A HeBT selector sorts its children based on the hints it receives.

10.4.3 Hints and Conditions

With the modifications presented so far, we have made our trees capable of accepting hints and reordering the branches controlled by their selector nodes. It is up to tree designers to expose whatever logic they feel is important to expose to higher levels (i.e., to choose what hints the tree will accept).

As we have said, the execution flow in a BT is controlled by the type of nonleaf nodes we use and how we combine them. We can have many different type of nodes, but simplifying the traditional BT model, we could say most trees are collections of selectors and sequences, as shown in Figure 10.6.

Most of these sequences follow a basic pattern—shown in Figure 10.7—where some condition nodes are placed as the first children, followed by actual actions. This way, the actions will only get executed if these preconditions are met. If one of the conditions fails, the sequence will bail out, returning a failure, which will probably be caught by a selector that will then try to run a different branch.

This is not good for our hints in some situations, as the conditions could be making a hinted branch fail and not be executed. We can use the example shown in Figure 10.3 to illustrate this.

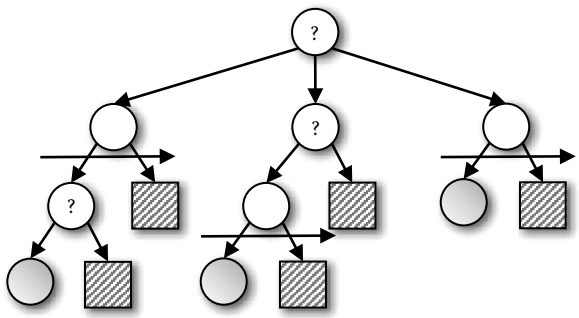

Figure 10.6

A basic BT can be seen as a series of selectors and sequences that control its execution flow.

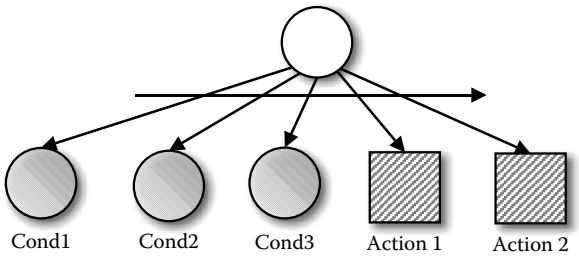

Figure 10.7

Basic sequence structure, where actions are preceded by a collection of preconditions.

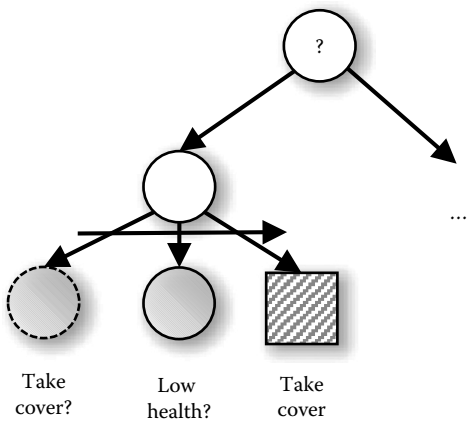

Figure 10.8

By using hint conditions, we can overcome problems caused by preconditions.

Let us say we have exposed the two first branches as hints. We will use the names "take cover" and "attack." Let us also say that we want to hint the AI to "take cover," by sending a positive hint to the tree.

The way we had defined our BT, the "take cover" branch already had the highest priority, so the selector does not really need to reorder its children (actually, it does reorder them, but this time, the order of the branches will not change). If we look closely at the first branch, we can see "take cover" is built as a sequence that checks a precondition (just as shown in Figure 10.7), called "low health?".

In the case where the AI has full health, the precondition will fail, making the sequence bail. The failure is ultimately propagated to the selector, which will run a different branch. Because we were hinting the AI to "take cover," we might be expecting it to take our suggestion into account, and not just ignore it blatantly.

So, we need a way to be able to ignore these preconditions *if that makes sense*, that is, if the condition is not really mandatory for the rest of the branch to be executed. In our example, we did not really need to be low on health to cover: this was just a design decision, probably trying to make the behavior more believable.

For that, HeBTs offer a *hint condition node*. This type of node is used to allow the BT to test if it is receiving a certain hint and what its type is (positive, negative, or neutral). We can modify our example's BT to modify the preconditions of the sequence, so our branch will look like what we show in Figure 10.8.

10.5 Multilevel Architecture

In the previous section, we introduced the concept of hints and how behaviors can be dynamically modified through them. These hints were sent to our trees by what we called "higher levels of logic."

Different approaches can be taken to implement these levels. For example, a quick and effective solution could be a layer of scripts that use the system to generate new behaviors.

However, the usage of scripts can make things harder to understand, as they normally require some technical background. A visual solution would be much more appropriate, as visualizing things is much simpler than learning a new language and its rules. Why not take advantage of the tools we have built to generate our base BTs, and expand it?

10.5.1 Behavior Controllers

BTs are constructed using a set of building blocks, among which we have actions; they are the nodes in charge of modifying the environment or the state of the AI instance itself. Depending on the granularity of the system, these actions can be more or less complex, ranging from subbehaviors, such as "take cover" to atomic actions such as "find cover spot." For users not interested in how the behaviors work—but just in the fact they do work—the coarser the granularity, the simpler the system will be for them.

Modifying a big BT can be complex and could require taking into account quite a lot of variables. Also, small changes in a tree could lead to undesirable behaviors, making AIs not work as expected. Because of this, we do not want new behaviors to be created from scratch; instead, we just want them to be flexible and malleable. So let us keep a base tree, maintained by engineers, and provide the team with the means to create new higher-level trees.

Higher-level trees use a different set of building blocks. Specifically, we will replace the action nodes with some new nodes that we call *hinters*. These nodes, as their name indicates, will send hints to the tree's immediate lower-level BT. The new trees will work on top of our base behavior, modifying it dynamically and allowing designers to prototype new ideas easily and safely, as the main BT is not modified permanently, thus reducing risks.

From now on, our AI instances will no longer be controlled by a single tree but by a number of layers of BTs. This set of trees is owned by a behavior controller. These controllers are in charge of maintaining the multiple levels of trees an AI can use and of running all of them to produce the final results.

A behavior controller works as a stack where we can push new trees. The top of the stack represents the highest level of logic, whereas the bottom contains the base BT. Every time we add a new tree, the controller informs the newly created high-level tree about what its immediate lower-level tree is, which will allow hints to be sent to the correct BT. This multilevel architecture is shown in Figure 10.9.

Once all the different trees have been created and registered with a behavior controller, it can run the final behavior. HeBTs are run from the top down, so higher levels are run first; this means that, by the time a tree is going to be executed, it would have already received all its hints, and their branches would be properly sorted. This process is shown in Figure 10.10.

By the end of each update, the AI will have run whatever action it has considered to have the higher priority, based on the information it has gathered from the environment *and* the hints it has received.

10.5.2 Exposing Hints to Higher Levels

High-level trees are built using a base BT to determine the hints that are available to them. When creating new trees, designers can name the branches of their selectors, which will

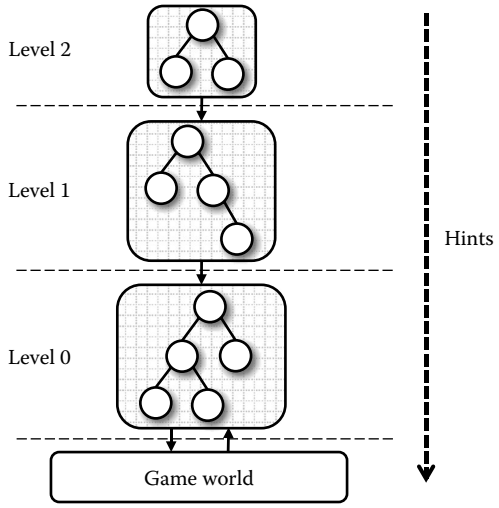

Figure 10.9

Multilevel structure of a HeBT.

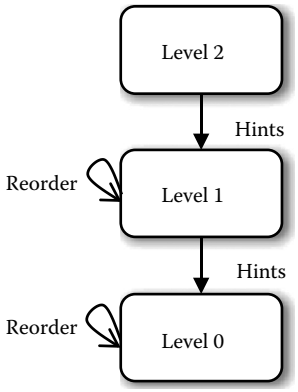

Figure 10.10

Each level in a HeBT will send hints to the level immediately below, causing the lower-level to reorder its priorities.

automatically expose the corresponding hints. In a similar way, if a condition hint is used anywhere in the low-level tree, the hint will automatically be exposed.

High-level trees cannot use actions; instead, they use hinters. Hinters allow trees to send any of the hints their lower-level trees are exposing to them, in order to produce new behaviors. Internally, they are very simple nodes: their logic is only executed once, and they bail out succeeding right after the hint has been sent.

It is important to note that hinters can send different types of hints, allowing us to send positive or negative hints. They can also set a hint back to neutral if necessary.

10.6 More Complex Example

So far, we have studied what a HeBT is and how it works internally. We have also been illustrating our exposition with a simple example. However, this example does not show the full potential of the system, so in this section, we will present an example that is closer to what we could find in a real game.

10.6.1 Prototyping New Ideas

There are many different ways to modify a behavior to obtain different responses from two AIs running the same logic. Some of them are even trivial to implement, such as the use of personality traits. However, adding more complex logic on top of an existing behavior starts getting complicated, especially if we do not want or cannot change the original BT.

In a real project, we are always subject to changes at any time, but new ideas and changes may pose a big risk to the project or require resources we cannot afford. As we saw, HeBTs allow us to generate this logic easily, just by using a high-level BT that will run on top of our base tree, guiding its normal execution toward what our new logic is suggesting should be done.

10.6.2 Base Behavior

Working on a new behavior requires that we have a base one working correctly, as it will define the way AIs in our game respond to different situations. In a real-life project, it would also have been thoroughly tested and optimized.

Let us say that for our example, our design team have decided the game needs some soldiers that

- Are able to patrol using a predefined route
- Detect the player as an enemy when they enter their cone of vision
- Attack the player once it is identified
- Try to find a cover position to keep attacking from it, if the agent takes damage

This behavior would be represented by a complex BT, and we show a simplified version of it in Figure 10.11.

Let us take a look at the base behavior. At a first glance, we can see there are three main branches controlled by a selector. We have named the branches "PATROL," "COVER," and "ATTACK;" this automatically exposes hints with the same names that can be used by a higher-level tree. The BT's root is a conditional loop that will keep the tree running until the AI is killed.

The first branch defines the agents' precombat behavior. In our case, we have chosen to have the soldiers patrol the area while they do not have an enemy. As we saw in a previous section, this condition might prevent the tree from behaving as expected when it receives the "PATROL" hint; to fix that, we have added a hint condition and put both condition nodes under a selector, which will allow us to enter the branch if either condition is true. It is also worth noting we have used a parallel node to run our conditions as an assertion (i.e., the conditions will be checked continuously to enforce they are always met); this way, the branch will be able to bail out as soon as the AI engages an enemy.

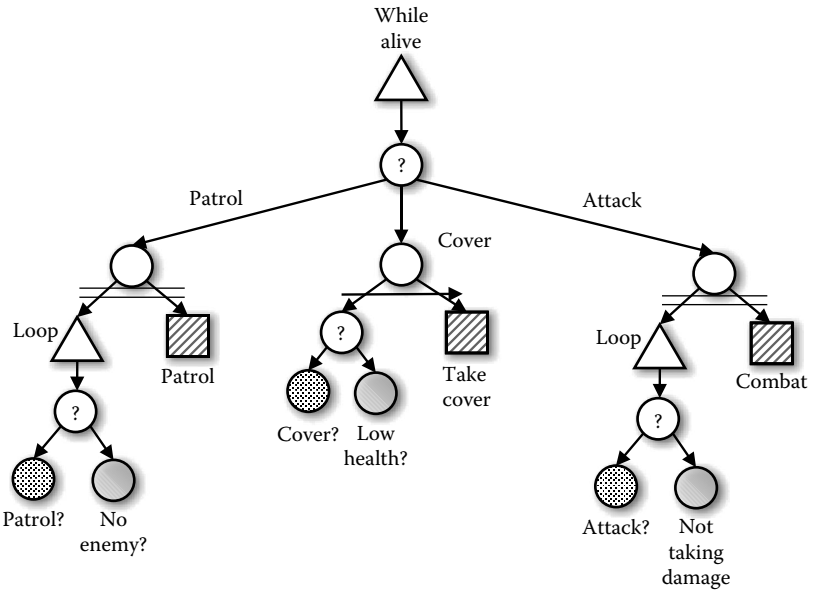

Figure 10.11

Simplified base BT our soldiers will run.

The second branch will make sure the agent takes cover when it is low on health. Similarly to the patrol branch, we have added a hint condition to make sure the tree will use hints properly.

Finally, the third branch is our combat behavior. Its structure is very similar to the "COVER" branch, with an assertion running to ensure combat is only triggered if we are not under fire (i.e., taking damage).

10.6.3 Prototype Idea

Once our AIs are able to run autonomously, in an ideal situation, most of the work for AI engineers will consist of debugging and polishing the behaviors. However, we could find ourselves in a situation when substantial changes to these behaviors are required, or maybe the team needs to keep testing new ideas to keep improving the game experience. This is where the power of HeBTs comes into play.

To demonstrate the capabilities of our new system, we will implement a "disguise" system just by adding a high-level tree to hint our base BT what should be done. The design for our feature is

- Players can wear the clothes of the enemies they kill, going unnoticed to other AIs if they do so.
- AIs should not recognize "disguised" players as enemies. However, they should react if the player damages them.

Basically, these changes would require gameplay and AI code modifications, and this new feature could not make it through to the final game. Because our game is using HeBTs,

we could delegate the prototyping of new ideas to the design team or at least let them play with new thoughts with minimal technical supervision (if we have the appropriate tools for the job).

We must bear in mind that if we want to have a system that requires virtually no programming work to be extended, we must start from designing our base behaviors correctly. Also, building a complete set of tree nodes and conditions can facilitate things further down the line.

10.6.4 Creating a High-Level Tree

So, as designers, the idea behind our new system is that we want AIs to ignore players that are "in disguise." So, basically, we want to hint the base level to prefer patrolling. The first pass at the high-level tree would be very similar to the one shown in Figure 10.12.

This is, in a nutshell, what our high-level tree should look like. However, we still have to define our condition. We want to check if the player is disguised, but we do not have a condition that does that.

Our system must have defined a way for AIs to maintain some knowledge about the world. A common way to do this is by using blackboards. Let us say we do have such a system, where we can write information and from where we can get details about the state of the world. In this case, our condition would be transformed to an "is enemy in disguise?" condition that checks for that information in the blackboard. But, if we need to read this information from the blackboard, we must have set it somewhere first.

For the sake of simplicity, we will use a "broadcast" action that allows us to write a value to the system's blackboard. What we want to do is to let other AIs in the world (by adding this information to the blackboard) about the new status of the player as soon as one AI dies. Since our first pass at the high-level tree was already checking if the AI was alive, let us extend the tree to modify the blackboard properly. We show this in Figure 10.13.

The first thing we have done is add an extra sequence as the new root of our tree. The idea behind it is that we want to run the "alive branch" (on the left) first, but always have a blackboard update following it.

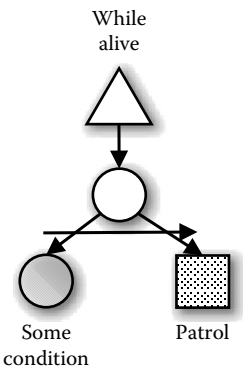

Figure 10.12

Basic idea behind the "disguise" system.

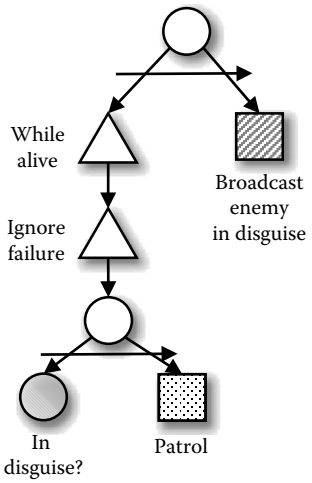

Figure 10.13

A first pass on a more complex high-level tree.

We have also added an extra decorator as a parent to our old sequence. The decorator's purpose is to ignore failures in the sequence, which will happen pretty frequently in our case (in fact, the sequence will fail every frame unless the player is in disguise); we never want the branch to fail, as it would break the root sequence.

So, with these changes, while the AI is alive, the tree will continue checking the blackboard to decide whether or not to hint the base BT to patrol; and, once the agent is dead, the blackboard will always be updated. When this happens, the remaining AIs will then have their blackboards updated, and they will start sending "PATROL" hints to their base BTs, causing those agents to ignore the player as intended.

Although this is a good first attempt at implementing the new feature, the tree is not completely correct yet, as AIs will not react to damage anymore if the player is disguised. To fix this problem, we need to clear the blackboard if an agent is under attack. The final high-level tree is shown in Figure 10.14.

In the final tree, we have added an extra selector that will catch whether the enemy has been attacked, clearing the disguise flag. The second branch of the selector is the same one our previous iteration had and, finally, the third branch is just making sure that, if nothing is going on, the "PATROL" hint is cleared.

10.6.5 Analyzing the Results

The key to this type prototyping is that the new logic is completely optional. We can just let the system know about it and see how the AI behaves with the extra feature or we can just remove the high-level tree, which will leave our original behavior untouched.

The base behavior requires minimal data changes, which are almost deactivated unless a high-level tree is used. Particularly, we have been using two types of base BT modifications: branch naming is a harmless change, as it does not affect the behavior at all; hint conditions do modify the original structure of the tree, but since they are straightforward

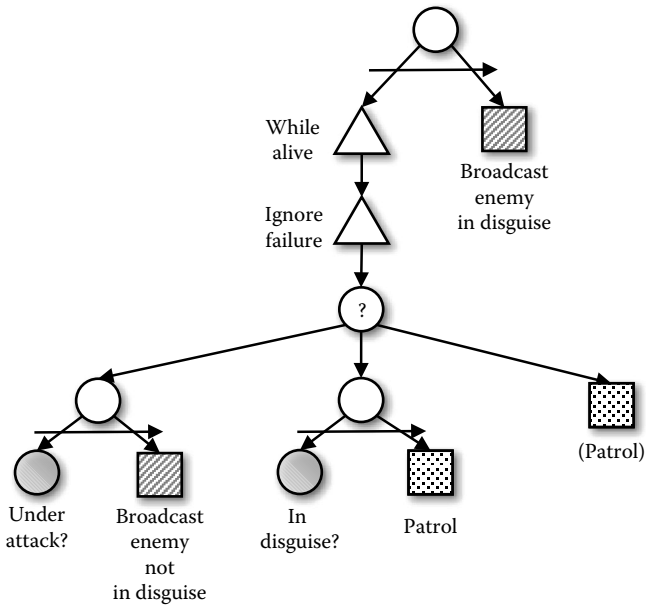

Figure 10.14

Final high-level tree that models the new feature.

flag (hint) checks, and the hints will never be enabled if a high-level tree is not present, it poses a very small risk. Hint conditions are also optional, and in some situations, we might not want to use them at all. Simply naming branches preemptively will expose dynamic priority control.

10.7 Other Applications

Over the course of this chapter, we have focused on the benefits HeBTs bring to quick and safe prototyping. These trees can also be used to help in other cases. In this section, we will present a couple of extra scenarios that can benefit from using HeBTs.

10.7.1 Adaptation

There are different ways to make our game more accessible to different types of players. Among them, we find the manual selection of a "difficulty level," which has been part of the game almost from the very beginning, or the adaptation of the difficulty level or situations to the player, which can allow us to offer a better and tailor-made experience to different groups of people.

HeBTs can help us offer different experiences to each player experience level and also allow us to modify things on the fly: we can define a different high-level tree per category and, in run-time, decide which one is most appropriate for our player. The base tree will recalculate its priorities based on the hints it is receiving and, hopefully, the player will enjoy our game better.

This is the approach Ubisoft's *Driver: San Francisco* used. In *Driver*, getaway drivers were able to adapt their route selection algorithm—which was controlled by a BT—by implementing a range of different high-level trees that could guide the route generation process [Ocio 12]. These high-level trees were called "presets," and they did things like making the route finder prefer straight routes (so casual players can catch the getaways easier) to zigzag routes or routes through dirt roads or alleyways.

10.7.2 Group Behaviors

Our hinted-execution model could also be used to create complex group behaviors based on command hierarchies. Hints would flow down the chain, allowing some AIs to have a better control over what others should do.

In a hint-based system, we would be able to create new links in our chain as high-level trees that are built on top of several base behaviors, rather than just one; in this case, each base tree would expose the orders that a particular class of AI can accept. Our higher-level tree would be able to broadcast hints to groups of AIs that are using the behaviors this level was based on.

An example of this would be a small army that has warriors, archers, and medics. A simplified version of their behaviors is shown in Figure 10.15.

We could use different generals, defining different high-level trees to create an intelligent army that obeys the orders we want to send. For instance, we could build a high-level AI that wants to attack with its archers first, holding off warriors and medics, and continuing with a melee-type attack that includes sending the medics along with the warriors to try and heal wounded units, and never allowing the units to retreat. The high-level tree that would control such behavior is shown in Figure 10.16.

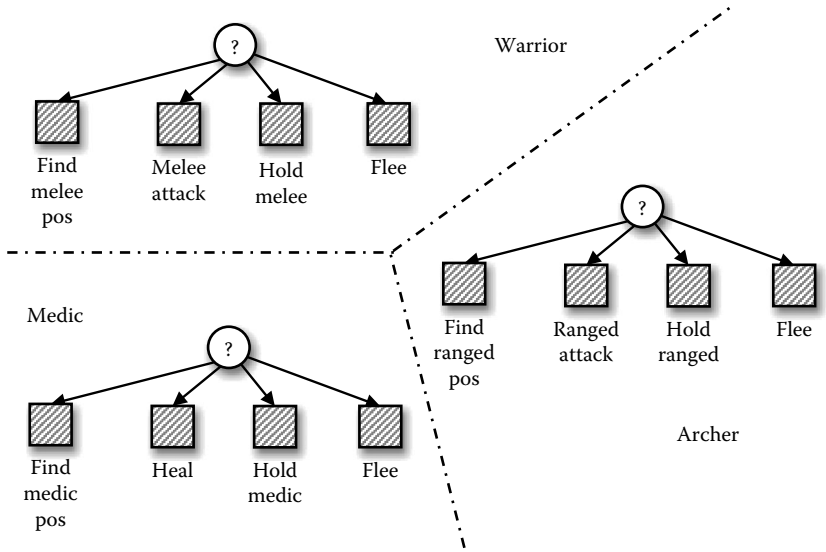

Figure 10.15

Base BTs controlling the different types of units in our army.

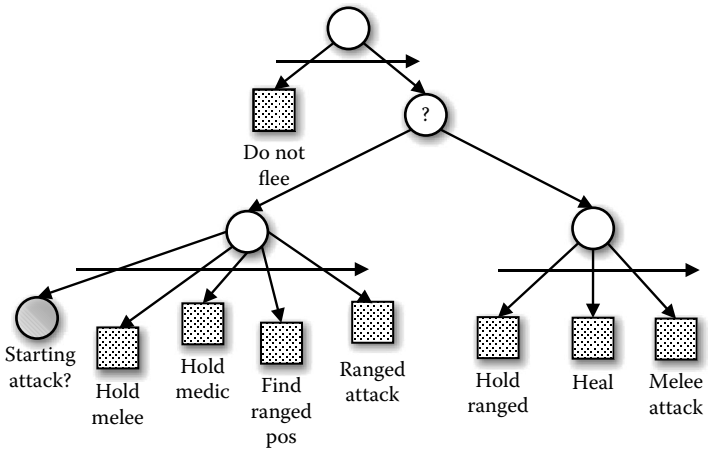

Figure 10.16

High-level tree that will define a tyrant high-level AI that never allows individuals to retreat.

Groups of units (as high-level entities) would also be controlled by HeBTs, so they could potentially receive hints too, producing a complex chain of command that can help us create more credible group behaviors.

10.8 Conclusion

BTs are a proven technology that has been used in many successful commercial games. However, as with any other technology, any change is risky, especially if these changes are made in the last stages of production.

HeBTs try to mitigate these risks by providing a way to do dynamic, revertible modifications to bigger, more complex BTs. These modifications are also controlled by another BT (that we call "high-level" BT), so we can still take advantage of the power and visual editing capabilities of the technology.

As we showed in this chapter, HeBTs help in many different problems, such as rapid prototyping, dynamic behavior adaptation, or group behaviors. In any case, risks are kept to a low, as we will never lose our base, tested behavior.

This technology is not hard to implement on top of an existing BT system and has also been used in an AAA game, *Driver: San Francisco*. HeBTs were key to the success of the game's AI, allowing its developers to adapt the behaviors of their getaway drivers to the skills of their players.

References

[AIGameDev 15] AIGameDev.com. http://www.aigamedev.com/.

[Champandard 08] Champandard, A. J. 2008. Getting started with decision making and control systems. *AI Game Programming Wisdom*, Vol. 4, pp. 257–263. Boston, MA: Course Technology.

[Champandard 13] Champandard, A. J. and Dunstan P. 2013. The behavior tree starter kit. In *Game AI Pro: Collected Wisdom of Game AI Professionals*. Boca Raton, FL: A K Peters/CRC Press.

[Isla 05] Isla, D. 2005. Handling complexity in the Halo 2 AI. In *Proceedings of the Game Developers Conference (GDC),* San Francisco, CA.

[Ocio 10] Ocio, S. 2010. A dynamic decision-making model for game AI adapted to players' gaming styles. PhD thesis. University of Oviedo, Asturias, Spain.

[Ocio 12] Ocio, S. 2012. Adapting AI behaviors to players in driver San Francisco: Hinted-execution behavior trees. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-12),* Stanford University, Stanford, CA.