6

The Behavior Tree Starter Kit

Alex J. Champandard and Philip Dunstan

6.1 Introduction

You've done your homework and found that behavior trees (BTs) are a proven and established technique that game developers regularly use to build their AI [Isla 05, Champandard 07]. Not only does a BT give you a solid foundation to build upon, but it also gives you a lot of flexibility to include other techniques in a way that gives you full control over behavior and performance. Now you're ready to start coding!

This article introduces the simplest piece of code that you can call a behavior tree, and builds it up incrementally. The associated source code is called the Behavior Tree Starter Kit [BTSK 12], and is intended to serve as an example of a working BT for you to learn from. Since it's under an open-source license, you can use this as a starting point for your own projects. This is not a reusable middleware library, however, and it's important that you understand the core concepts and take ownership of the code.

The first section of this article paints the big picture for behavior trees, introducing a simple example tree, and explaining how to build BTs and how to use them for making AI decisions. The second section dives into the implementation of a first-generation BT, along with all its building blocks (e.g., sequences and selectors) and a discussion of the API. Finally, the third section explains the principles of second-generation BTs, and the improvements they offer. You'll learn about memory optimizations and event-driven implementations that scale up significantly better on modern hardware.

Throughout this article, source code examples are provided to demonstrate the concepts. Note, that the code listings in this article are edited for print, in particular using C++11 syntax, and some shortcuts have been taken for the sake of brevity. For the original implementations see the source code at http://github.com/aigamedev.

6.2 The Big Picture

To assist in the demonstration of the components that comprise a behavior tree it is first useful to see how such a tree might be structured. Following is an example of how a simple behavior tree might be designed for a robot guard that is hunting the Player.

6.2.1 A Simple Example

This AI is split into three main behaviors. First, if the robot guard can see the Player then it will either shoot three times at the Player if it is close enough, or move closer to the Player. Second, if the robot has recently seen the Player but can no longer see them then it will move to the Player's last known position and look around. Third, in case the robot has not seen the Player for some time, the fallback behavior is to move to some random location and look around.

```
BehaviorTree* bt = BehaviorTreeBuilder()
    .activeSelector()
    .sequence()    //Attack the player if seen!
        .condition(IsPlayerVisible)
        .activeSelector()
            .sequence()
                .condition(IsPlayerInRange)
                .filter(Repeat, 3)
                    .action(FireAtPlayer)
            .action(MoveTowardsPlayer)
    .sequence()    //Search near last known position.
        .condition(HaveWeGotASuspectedLocation)
        .action(MoveToPlayersLastKnownPosition)
        .action(LookAround)
    .sequence()    //Randomly scanning nearby.
        .action(MoveToRandomPosition)
        .action(LookAround)
    .end();
```

The example above demonstrates the use of the Tree Builder Pattern to separate the construction of the behavior tree from the behavior tree nodes. (This example has been edited for print.) The full source code for the tree builder and its examples can be found with the rest of the Behavior Tree Starter Kit source code as described in Section 6.1.

6.2.2 Updating the Behavior Tree

Given a behavior tree, how does the game logic update it? How often is this done and does it involve traversing the tree from the root every time? These are common questions about BTs.

To assist in the updating of a behavior tree it is useful to introduce a `BehaviorTree` object, which will serve as the central point for storing and updating the tree. The BehaviorTree object is most often created by a `BehaviorTreeBuilder` as shown in the example in Section 6.2.1, or using another builder that loads a tree from a file.

```
class BehaviorTree {
protected:
    Behavior* m_pRoot;
public:
    void tick();
};
```

This is a first-generation BT, and as such, the `BehaviorTree` class remains simple. It contains a single pointer to the root of the behavior tree, and a `tick()` function which performs the traversal of the tree. This is the entry point of the BT, and is called anytime an update is needed. It is often not necessary to update the behavior tree every game frame, with many games deciding to update each behavior tree every other frame or at 5Hz so that the load for updating the behavior trees of all characters can be spread across multiple game frames.

While this example seems straightforward and the implementation simply delegates the `tick()` to the root node, Section 6.4 of this article shows the advantages of such a centralized `BehaviorTree` class. It enables several advanced features such as improving runtime performance by controlling the memory allocation of BT nodes (Section 6.4.3), and an event-driven traversal that modifies the functionality of the `tick()` function to reduce node accesses.

6.3 Building Blocks

Moving on from the big picture, this section jumps to the lowest level of the implementation, progressing bottom-up and building up complexity incrementally with new features.

6.3.1 Behaviors

The concept of a *behavior* is the most essential part of a BT. The easiest way to think of a behavior from a programming perspective is an abstract interface that can be activated, run, and deactivated. At the leaf nodes of the tree, *actions* (e.g., "open door," "move to cover," "reload weapon") and *conditions* (e.g., "do I have ammunition?" "am I under attack?") provide specific implementations of this interface. Branches in the tree can be thought of as high-level behaviors, hierarchically combining smaller behaviors to provide more complex and interesting behaviors.

Here is how such an interface is implemented in the BTSK:

```
class Behavior {
public:
    virtual void onInitialize()       {}
    virtual Status update()           = 0;
    virtual void onTerminate(Status)  {}
    /*... */
};
```

This API is the core of any BT, and it's critical that you establish a clear specification for these operations. For example, the code expects the following contracts to be respected:

- The `onInitialize()` method is called once, immediately before the first call to the behavior's update method.
- The `update()` method is called exactly once each time the behavior tree is updated, until it signals it has terminated thanks to its return status.
- The `onTerminate()` method is called once, immediately after the previous update signals it's no longer running.

When building behaviors that rely on other behaviors (such as the sequence and selector behaviors described later in this section), it's important to keep these API contracts in mind. To help make sure you don't break these assumptions, it can help to wrap these functions into a single entry point.

```
class Behavior {
protected:
    /* API identical to previous code listing. */
private:
    Status m_eStatus;
public:
    Behavior() : m_eStatus(BH_INVALID) {}
    virtual ~Behavior() {}
    Status tick() {
        if (m_eStatus != BH_RUNNING) onInitialize();
        m_eStatus = update();
        if (m_eStatus != BH_RUNNING) onTerminate(m_eStatus);
        return m_eStatus;
    }
};
```

This approach is a bit slower, since you must use conditional branches every `tick()`. Most composite behaviors could handle this more efficiently since they process the return statuses anyway. However, having such a wrapper function avoids many beginner mistakes.

6.3.2 Return Statuses

Each behavior, when executed, passes back a return status. The return status is a critical part of any behavior tree, without which it simply wouldn't work. In practice, return statuses plays two roles:

- **Completion Status**—If the behavior has terminated, the return status indicates whether it achieved its purpose. There are two completion statuses most commonly used: `SUCCESS` (indicates that everything went as expected) and `FAILURE` (specifies that something apparently went wrong).
- **Execution Hints**—While the behavior is running, each update of the behavior also returns a status code. Most of the time, this is `RUNNING`, but modern BTs can leverage this status code to provide much more efficient implementations. For example, the `SUSPENDED` status code is an essential part of an event-driven BT, as you will see in Section 6.4.4.

In certain special cases, you might be tempted to add additional return statuses to the list. For example, there are some implementations that distinguish between expected issues (`FAILURE`) and unforeseen problems (`ERROR`). However, this quickly makes the code in the rest of the tree much more complex, and does not make the BT any more powerful. A more convenient approach to deal with failures is to let behaviors check for specific types of failure they expect, and deal with those cases outside of the tree's return statuses.

6.3.3 Actions

In a behavior tree, the leaf nodes have the responsibility of accessing information from the world and making changes to the world. Leaf behaviors that make such changes are called `Actions`.

When an action succeeds in making a change in the world it returns `SUCCESS`; otherwise it's simply a `FAILURE`. A status code of `RUNNING` indicates processing is underway. Actions are little more than a `Behavior`, except for initialization and shutdown that require extra care.

- **Initialization**—All but the simplest of actions will need to interface with other systems and objects to do the work. For example, a particular action may need to fetch data from a blackboard, or make a request of the animation systems. In a well-defined modular architecture getting access to these other games systems can be problematic. Extra work is required during the setup of behavior tree actions to provide the systems that those actions will use. This is often solved by passing extra parameters during node instantiation or through the use of the Visitor software design pattern.
- **Shutdown**—Like initialization, the shutdown of actions can be problematic due to the dependencies on external systems. Special care must be taken when shutting down an action to ensure that freeing resources does not interfere with other behaviors. For example, you cannot simply reset the animation system once an action shuts down as another instance of that action may have recently been activated elsewhere.

Helper functions can be set up to facilitate initialization and shutdown of actions if needed. In most cases, actions will simply inherit the functionality from the base `Behavior` class.

An example action used in the robot guard example of Section 6.2.1 is the action to move to the Player's last known location if the robot can no longer see the Player. This action will likely instruct the navigation system to move the robot to that location and return a `SUCCESS` status code. If for some reason the navigation system is unable to perform the request—for instance, a door has closed and the navigation system is unable to find a path to the target location—the action will return a `FAILURE` status code.

6.3.4 Conditions

Conditions are also leaf nodes in the tree and are the tree's primary way of checking for information in the world. For example, conditions would be used to check if there's cover nearby, if an enemy is in range, or if an object is visible. All conditions are effectively Boolean, since they rely on the return statuses of behaviors (success and failure) to express True and False.

In practice, conditions are used in two particular cases:

- **Instant Check Mode**—See if the condition is true given the current state of the world at this point in time. The check is run once immediately and the condition terminates.
- **Monitoring Mode**—Keep checking a condition over time, and keep running every frame as long as it is True. If it becomes False, then exit with a `FAILURE` code.

As well as being able to specify the mode of execution of a Condition, it's also useful to provide a negation parameter that effectively tells the code to do the exact opposite. This allows for the simpler reuse of existing code, such as checking to see if an enemy is in range, to create a condition to test for the opposite, i.e., that the enemy is not within range. In *Check* mode, this is Boolean negation, but in *Monitoring* mode the condition would keep running as long as it is False.

6.3.5 Decorators

The next step to building up an interesting BT is to wrap behaviors with other behaviors, adding detail, subtlety, and nuance to its logic. Decorator behaviors, named after the object-oriented design pattern, allow you to do this. Think of them as a branch in the tree with only a single child, for example, a behavior that repeats its child behavior *n* times, a behavior that hides the failure of its child node, or a behavior that keeps going forever even if its child behavior exits. All of these are decorators, and are very useful for technically minded developers using BTs.

```
class Decorator : public Behavior {
protected:
    Behavior* m_pChild;
public:
    Decorator(Behavior* child) : m_pChild(child) {}
};
```

The base `Decorator` class provides all the common functionality for implementing a decorator efficiently, only storing a single child for instance. Specific types of decorators are implemented as derived classes; for instance the update method on the `Repeat` decorator might be implemented as follows.

```
Status Repeat::update() {
    while (true) {
        m_pChild->tick();
        if (m_pChild->getStatus() == BH_RUNNING) break;
        if (m_pChild->getStatus() == BH_FAILURE) return BH_FAILURE;
        if (++m_iCounter == m_iLimit) return BH_SUCCESS;
    }
}
```

In this example, the repeating behavior keeps executing its child behavior until a limit is reached. If the child fails, the decorator also fails. When the child behavior succeeds, its next execution happens immediately in the same update once it has been reset.

The robot guard example introduced in Section 6.2.1 uses a *Repeat* condition to fire three times if the player is within firing range. Decorators like this provide a simple way to introduce subtle behavior patterns to the behavior tree without duplicating nodes in the tree.

6.3.6 Composites

Branches with multiple children in a behavior tree are called *composite behaviors*. This follows the composite pattern in software engineering, which specifies how objects can be assembled together into collections to build complexity. In this case, we're making more interesting, intelligent behaviors by combining simpler behaviors together.

It's often a good idea to have a base class for composite behaviors to avoid redundant code in the subclasses. The helper functions to add, remove, and clear children can be implemented just once in this base class.

```
class Composite : public Behavior {
public:
    void addChild(Behavior*);
    void removeChild(Behavior*);
    void clearChildren();
protected:
    typedef vector<Behavior*> Behaviors;
    Behaviors m_Children;
};
```

Common composite behaviors, like sequences and selectors, derive from this base class.

6.3.7 Sequences

Sequences are one of the two most common branch types. Sequences allow the BT to purposefully follow "plans" that are hand-specified by the designers. Sequences execute each of their child behaviors in sequence until all of the children have executed successfully or until one of the child behaviors fail.

The example behavior tree introduced in Section 6.2.1 for the robot guard uses sequences at several points to group together behaviors into larger behaviors. An example is the branch that the guard uses to search near the player's last known position when the player is not visible. In this branch the first behavior node is a condition node to check whether the robot has a suspected location for the player. If this condition succeeds the action to move to the suspected location and search around that location will be run. Otherwise, if the condition fails the sequence node will fail and the behavior tree will continue on to the next branch outside the sequence—searching a random location.

In the code shown below, a sequence is a composite behavior, which just happens to chain together its child behaviors one by one.

```
class Sequence : public Composite {
protected:
    Behaviors::iterator m_CurrentChild;
    virtual void onInitialize() override {
        m_CurrentChild = m_Children.begin();
    }
    virtual Status update() override {
        //Keep going until a child behavior says it's running.
        while (true) {
            Status s = (*m_CurrentChild)->tick();
            //If child fails or keeps running, do the same.
            if (s != BH_SUCCESS) return s;
            //Move on until we hit the end of the array!
            if (++m_CurrentChild == m_Children.end())
                return BH_SUCCESS;
        }
        return BH_INVALID;//Unexpected loop exit.
    }
};
```

The initialization code for the sequence starts at the beginning of the array of children. The update processes each child behavior in the list one by one, bailing out if any of them fail. The sequence returns a `SUCCESS` status if all of the children execute successfully.

There's one important thing to note about this implementation; the next child behavior is processed immediately after the previous one succeeds. This is critical to make sure the BT does not miss an entire frame before having found a low-level action to run.

6.3.8 Filters and Preconditions

A filter is a branch in the tree that will not execute its child behavior under specific conditions. For instance, if an attack has a cooldown timer to prevent it from executing too often, or a behavior that is only valid at a specific distance away from a target, etc. Designers can easily use filters to customize the execution of common behaviors—for instance, customizing them for a specific character or situation.

Using the modular approach of the BTSK, it's trivial to implement a filter as a type of sequence. Assuming the filter has a single condition, you can attach it to the start of the sequence—ensuring that it gets executed (and therefore, checked) first. If the filter has a single branch (or action), it comes next in the sequence. Of course, it's equally easy to set up a sequence with multiple preconditions and multiple action branches afterwards.

```
class Filter : public Sequence {
public:
    void addCondition(Behavior* condition) {
        //Use insert() if you store children in std::vector
        m_Children.push_front(condition);
    }
    void addAction(Behavior* action) {
        m_Children.push_back(action);
    }
};
```

You can also easily create Boolean combinations of conditions to add to the filter, as a testimony to the power of core BT nodes like sequences (AND) and selectors (OR).

6.3.9 Selectors

Selectors are the other most common branch type. Selectors let the BT react to impediments in the world, and effectively trigger transitions between different fallback behaviors. A selector executes each of its child behaviors in order until it finds a child that either succeeds or that returns a RUNNING status.

In the robot guard behavior tree example described in Section 6.2.1 a selector is used to decide which of the three main behavior branches should be picked. The first branch—attacking the player—is first executed. If this branch fails—if the player is not visible, for instance—the sequence node will execute the second branch, searching near the player's last known position. If that behavior also fails the sequence will execute the final behavior—searching a random location.

From the code perspective, a selector is the counterpart of a sequence; the code not only derives from a composite, but looks very similar as well. Only the two lines dealing with the specific return statuses are different.

```
virtual Status update() {
    //Keep going until a child behavior says it's running.
    while (true) {
        Status s = (*m_Current)->tick();
        //If child succeeds or keeps running, do the same.
        if (s != BH_FAILURE) return s;
        //Continue search for fallback until the last child.
        if (++m_Current == m_Children.end())
            return BH_FAILURE;
    }
    return BH_INVALID;//"Unexpected loop exit."
}
```

The rest of the selector is identical to the sequence implementation. Similarly, note that the selector keeps searching for fallback behaviors in the same `update()` until a suitable behavior is found or the selector fails. This allows the whole BT to deal with failures within a single frame without pausing.

6.3.10 Parallels

A parallel node is another type of composite branch in the tree that allows you to do more advanced control structures, such as monitoring if assumptions have been invalidated while you are executing a behavior. Like other composites, it's made up of multiple behaviors; however, these are all executed at the same time! This allows multiple behaviors (including conditions) to be executed in parallel and for those behaviors to be aborted if some or all of them fail.

A parallel node is not about multithreading or optimizations, though. Logically speaking, all child behaviors are run at the same time. If you trace the code, their update functions would be called sequentially one after another in the same frame.

```
class Parallel : public Composite {
public:
    enum Policy {
        RequireOne,
        RequireAll,
    };
    Parallel(Policy success, Policy failure);
protected:
    Policy m_eSuccessPolicy;
    Policy m_eFailurePolicy;
    virtual Status update() override;
};
```

It's important for the parallel to be extremely precisely specified, so that it can be understood intuitively and relied upon without trying to second-guess the implementation. In this case, there are two parameters; one specifies the conditions under which the parallel succeeds and the other for failure. Does it require all child nodes to fail/succeed or just one before failing/succeeding? Instead of enumerations, you could also add counters to allow a specific number of behaviors to terminate the parallel, but that complicates the everyday use of the parallel without any additional power. Most useful BT structures can be expressed with these parameters, using decorators on child nodes if necessary to modify their return statuses.

```
virtual Status update() {
    size_t iSuccessCount = 0, iFailureCount = 0;
    for (auto it: m_Children) {
        Behavior& b = **it;
        if (!b.isTerminated()) b.tick();
        if (b.getStatus() == BH_SUCCESS) {
            ++iSuccessCount;
            if (m_eSuccessPolicy == RequireOne)
                return BH_SUCCESS;
        }
        if (b.getStatus() == BH_FAILURE) {
            ++iFailureCount;
            if (m_eFailurePolicy == RequireOne)
                return BH_FAILURE;
        }
    }
    if (m_eFailurePolicy == RequireAll && iFailureCount == size)
        return BH_FAILURE;
    if (m_eSuccessPolicy == RequireAll && iSuccessCount == size)
        return BH_SUCCESS;
    return BH_RUNNING;
}
```

The implementation of the parallel iterates through each child behavior, and updates it. Counters are kept for all terminated behaviors, so the failure policy and success policy can be checked afterwards. Note that failure takes priority over success since the BT itself should assume the worst case and deal with it rather than proceed regardless. Also, in this implementation the parallel terminates as soon as any policy is satisfied, even if there are behaviors not yet run.

When a parallel terminates early because its termination criteria are fulfilled, all other running behaviors must be terminated. This is done during the `onTerminate()` function that iterates through all the child nodes and handles their termination.

```
void Parallel::onTerminate(Status) {
    for (auto it: m_Children) {
        Behavior& b = **it;
        if (b.isRunning()) b.abort();
    }
}
```

Parallels are the foundation of more advanced BT control structures and therefore tend to uncover a wide variety of little issues, like how to cleanly shutdown (see Section 6.3.3) and how to handle interrupting behaviors. There are two schools of thought on how to handle behaviors that need to be interrupted before they terminate on their own, for example, when they are run together in a parallel node.

- All behaviors should have the option to keep running if they want to, effectively having a noninterruptible flag that will cause the parent behavior to wait for termination. In this case, the `abort()` function becomes a request that is taken into account during the next `update()` if appropriate. Low-level BTs, in particular those dealing with animation control directly, tend to benefit from this option.

- All behaviors should support immediate termination, though they are given the option to clean-up after themselves using `onTerminate()`, which can optionally be given a special `ABORTED` status code. High-level BTs work best this way, since they don't want to micro-manage low-level states; the noninterruptible animation can be handled elsewhere in supporting systems.

The BTSK takes this second approach. When the behavior tree switches from a branch to another, this is done instantly, and any transitions (e.g., audio, animation) must be set up during the switch and managed by external systems. Then, in the next branch, if the same system is requested, it will simply delay the BTs request until the transition is over (e.g., play sound, play animation).

6.3.11 Monitors

Arguably, continuously checking if assumptions are valid (i.e., monitoring conditions) is the most useful pattern that involves running behaviors in parallel. Many behaviors tend to have assumptions that should be maintained while a behavior is active, and if those assumptions are found invalid the whole sub-tree should exit. Some examples of this include using an object (assumes the object exists) or melee attacks (assumes the enemy is in range), and many others.

The easiest way to set this up is to reuse the parallel node implementation, as a Monitor node can be thought of as a parallel behavior with two sub-trees; one containing conditions which express the assumptions to be monitored (read-only), and the other tree of behaviors (read–write). Separating the conditions in one branch from the behaviors in the other prevents synchronization and contention problems, since only one sub-tree will be running actions that make changes in the world.

```
struct Monitor : public Parallel {
    //Implementation is identical to the Filter sequence.
    void addCondition(Behavior* condition);
    void addAction(Behavior* action);
};
```

In the exact same way as a `Filter`, the monitor provides simple helper functions to ensure the conditions are set up first in the parallel. These conditions will be checked first before the actions are executed, bailing out early if there are any problems. This API is useful only if you create your BTs in C++, but most likely you'll impose these orderings in your BT editing tool.

6.3.12 Active Selectors

One final building block you'll most likely need in a production BT is an "active" selector, which actively rechecks its decisions on a regular basis after having made them. This differs from the traditional "passive" selectors by using another form of parallelism to retry higher-priority behaviors than the one that was previously chosen. You can use this feature to dynamically check for risks or opportunities in select parts of the tree, for example, interrupting a patrol with a search behavior if a disturbance is reported.

Active selectors appear twice during the short behavior tree example in Section 6.2.1. At the top level of the behavior tree an active selector is used to allow the high priority

behavior of attacking a visible enemy to interrupt lower-priority behaviors such as searching for the player or randomly patrolling. A second instance of an active selector is during the evaluation of behaviors when the player is visible to the guard robot. Here, an active selector is used so that the higher-priority behavior of shooting at the player if the player is within range preempts the lower-priority behavior of moving towards the player if they are not in range.

One simple way to implement this is to reuse a Monitoring node within a passive selector from Section 6.3.9 that terminates the low-priority node if the higher-priority behavior's preconditions are met. This type of implementation can be easier for straightforward cases with one condition, and works efficiently for event-driven implementations discussed in Section 6.4.4. However, you'll most likely require a specialized implementation to deal with more complex situations.

```
Status ActiveSelector::update() {
    Behaviors::iterator prev = m_Current;
    Selector::onInitialize();
    Status result = Selector::update();
    if (prev != m_Children.end() && m_Current != prev)
        (*previous)->abort();
    return result;
}
```

This active selector implementation reuses the bulk of the underlying `Selector` code, and forces it to run every tick by calling `onInitialize()`. Then, if a different child node is selected the previous one is shutdown afterwards. Separately, the `m_Current` iterator is initialized to the end of the children vector. Keep in mind that forcefully aborting lower-priority behaviors can have unwanted side effects if you're not careful; see Section 6.3.3.

6.4 Advanced Behavior Tree Implementations

As BTs have grown in popularity in the games industry the forms of implementation have become increasingly diverse, from the original implementation in *Halo 2* [Isla 05] to *Bulletstorm*'s event-driven version [PSS 11]. Despite the diversity there have been some common changes over the past couple of years in the ways that behavior trees have been implemented. These changes led to our coining of the terms first- and second-generation behavior trees at the Paris Shooter Symposium 2011 [PSS 11].

6.4.1 First- versus Second-Generation Trees

While there are no hard rules for classifying a behavior tree implementation, there are some common patterns behind original implementations and more modern ones. In general, first-generation BTs have the following characteristics:

- Small and shallow trees with relatively few nodes in them.
- Large behaviors written in C++ with "complex" responsibilities.
- No (or little) sharing of data between multiple behavior tree instances.
- Simple implementations with no worry about performance.
- Often written in one `.h` and `.cpp` file, not necessarily reusable outside of AI.
- The behavior tree concept is mostly used as a pattern for writing C++.

In contrast, second-generation trees have had to deal with a console hardware transition and designs with additional complexity and scale. They are defined as follows:

- Larger and deeper trees with many more nodes.
- Smaller powerful nodes that better combine together.
- BT data that is shared between multiple instances wherever possible.
- A heavily optimized implementation to improve scalability.
- Written as a reusable library that can be applied to any game logic.
- The behavior tree becomes a DSL with efficient interpreter.

It's important to point out that first-generation implementations are not necessarily worse than their successor, they just fulfill different requirements. If you can use a simpler implementation that avoids the complexity of the second-generation, then don't hesitate to do so!

6.4.2 Sharing Behavior Trees Between Entities

A powerful extension to first-generation behavior trees is the ability to share data between multiple instances, in particular, the structure of the tree or common parameters. This can significantly reduce the memory requirements for BT-based systems, especially in scenes with large numbers of complex characters.

The most important requirement for sharing data is the separation of two concepts:

- **Nodes**—Express the static data of the BT nodes, for instance the pointers to children of a composite node, or common parameters.
- **Tasks**—The transient node data required to execute each BT node. For example, sequences need a current behavior pointer and actions often require context.

In practice, the `Task` is a base class for runtime instance data and managing execution.

```
class Task {
protected:
    Node* m_pNode;
public:
    Task(Node& node);
    //onInitialize(), update() and onTerminate() as 3.3.1
};
```

The `Task` refers to a tree node which stores the shared data, including tree structure and parameters. Then, the `Node` implementation effectively becomes a factory for creating these tasks at runtime, when the node is executed by the tree.

```
class Node {
public:
    virtual Task* create() = 0;
    virtual void destroy(Task*) = 0;
};
```

To make these two classes compatible with the `Behavior` of Section 6.3.1, all that's required is to keep a `Node` and a `Task` together as member variables, and track their

`Status`. The node's factory functions `create()` must be called before the behavior's `tick()` function can be run the first time, and `destroy()` after the task has terminated.

While separating these two forms of BT data is relatively simple mechanically, it has a profound impact on the rest of the code. All composite nodes, for instance, must be able to "convert" nodes into tasks before execution. This can be done locally in all composites, or the `Task` instances can be managed centrally in a `BehaviorTree` class similar to Section 6.2.2. Then, the decision remains whether to allocate these instances on-the-fly or pre-allocate them—which will often depend on the size and complexity of the tree. The next section discusses similar memory issues relating to the allocation of `Node` objects forming the tree structure.

6.4.3 Improving Memory Access Performance

One of the significant drawbacks of first-generation behavior trees on modern hardware is the memory access patterns exhibited when executing the BTs. This is especially an issue on current generation console hardware with small cache sizes and limited hardware memory prefetching.

The primary cause of the poor memory access patterns is the dynamic memory allocation of the behavior tree nodes. Without careful management, the BT node storage may be scattered throughout memory, resulting in frequent cache misses as the BT execution moves from one node to the next. By changing the layout of the BT nodes in memory, it is possible to significantly improve the memory performance of behavior trees.

6.4.3.1 Node Allocation

The core mechanism for changing the memory layout of the behavior tree nodes is the introduction of centralized memory allocation API to the central `BehaviorTree` object that was introduced in Section 6.2.2. These additional allocate functions will be responsible for all node allocations.

When this object is constructed it allocates a block of memory into which all of the BT nodes will be allocated. To instantiate a BT node, the templated `allocate` function is used rather than the normal allocation functions. This function uses in-place `new` to allocate the new node within the block of memory owned by the `BehaviorTree` object.

```
class BehaviorTree {
public:
    BehaviorTree()
    : m_pBuffer(new uint8_t[k_MaxBehaviorTreeMemory])
    , m_iOffset(0)
    {}
    ~BehaviorTree() {
        delete [] m_pBuffer;
    }
    void tick();
    template <typename T>
    T& allocate() {
        T* node = new ((void*)((uintptr_t)m_pBuffer+m_iOffset)) T;
        m_iOffset += sizeof(T);
        return *node;
    }
```

```
protected:
    uint8_t* m_pBuffer;
    size_t m_iOffset;
};
```

By allocating all of the BT nodes via the custom allocate function it is possible to ensure that all of the nodes for a tree are allocated in the same localized area of memory. Furthermore, by controlling the order in which nodes are allocated (depth- or breadth-first), it is possible to optimize the layout so that it reduces cache misses during traversal. Improving the memory usage patterns as a BT is traversed can have significant impacts on the runtime performance.

6.4.3.2 Composite Node Implementations

As this change only affects the node memory allocations, the internal implementations of many of the behavior tree nodes described in Section 6.3 are completely unaffected. There are, however, additional optimizations that can be made to some of these implementations to further improve the resultant memory layout.

The primary candidates for further optimizations are the Composite nodes: Sequences, Selectors, and Parallels. In the simple implementation described in Section 6.3.6 each of these node types stored their children in a `vector<Behavior*>` in the Composite base class, resulting in additional heap allocations by the vector class for the data storage. This can be prevented by replacing the vector storage by an internal static array as shown in the code below.

```
class Composite : public Behavior
{
public:
    Composite() : m_ChildCount(0) {}
    void addChild(Behavior& child) {
        ptrdiff_t p = (uintptr_t)&child - (uintptr_t)this;
        m_Children[m_ChildCount++] = static_cast<uint32_t>(p);
    }
protected:
    uint32_t m_Children[k_MaxChildrenPerComposite];
    uint32_t m_ChildCount;
};
```

In this example, the child nodes store the child node address information as part of the primary node data. Each composite node contains static storage for *n* child nodes. In most trees, a limit of 7 on the maximum number of child nodes is more than sufficient. In cases where there are more children than this, then it is typically possible to use additional composite nodes to split apart the larger composite nodes (e.g., multiple nested sequences or selectors).

Rather than storing pointers to each of the child nodes, it is also possible to take advantage of the spatial locality of the nodes in memory and store only the offset of each child node from the composite node. As compiling software with support for 64-bit pointers becomes more common, this can result in significant memory savings. The `Composite` node example shown here requires 32 bytes of memory, whereas a naïve implementation storing pointers would require 64 bytes, occupying half of a cache line on current generation consoles.

6.4.3.3 Transient Data Allocation

The same transformations that were applied to the memory layout of the BT nodes in Section 6.4.3.2 can equally by applied to the transient data that is stored when each node is executed.

It is useful to think of this transient data as a stack. As a node is executed, its transient data is pushed onto the top of the stack. When a node execution is terminated, the transient data is popped back off the stack. This stack-like data structure is perfect for the depth-first iteration used in behavior trees. It allows for very simple management of transient data and results in very cache-friendly memory access patterns.

6.4.4 Event-Driven Behavior Trees

A final approach for optimizing BT traversal involves event-driven techniques. Instead of traversing the tree from the root every frame, simply to find previously active behaviors, why not maintain them in a list (of size one or more) for fast access? You can think of this list as a *scheduler* that keeps active behaviors and ticks the ones that need updating.

This is the essence of an event-based approach, and there are two ways to maintain it:

- Traverse the whole tree from scratch if the currently executing behaviors terminate, or there are changes in the world (or its blackboard). This effectively repopulates the task list from scratch when necessary, in a similar way than a planner would.
- Update the list of active behaviors incrementally as they succeed or fail. The parent of behaviors that terminate can be requested to decide what to do next, rather than traversing the whole tree for the root.

The first approach is a simple optimization to a traditional first-generation BT, but the second requires a much more careful implementation of the scheduler, which we'll dig into in the following sections.

6.4.4.1 Behavior Observers

The most important part of an event-driven BT is the concept of an observer. When a behavior terminates, the scheduler that was updating it also fires a notification to the parent, which can deal with the information as appropriate.

```
typedef Delegate<void (Status)> BehaviorObserver;
```

In the BTSK, the observer is implemented using a template-based fast delegate implementation, but this could be replaced with any functor implementation.

6.4.4.2 Behavior Scheduler

The central piece of code responsible for managing the execution of an event-driven BT is called a *scheduler*. This can be a stand-alone class, or left for the `BehaviorTree` class to implement such as in the BTSK. Essentially, the class is responsible for updating behaviors in one central place rather than letting each composite manage and run its own children. The example object below expands on the `BehaviorTree` class API that was first introduced in Section 6.2.2 to provide the management of BT tasks.

```
class BehaviorTree {
protected:
    deque<Behavior*> m_Behaviors;
public:
    void tick();
    bool step();
    void start(Behavior& bh, BehaviorObserver* observer);
    void stop(Behavior& bh, Status result);
};
```

The scheduler stores a list of active behaviors, in this case in a `deque` that has behaviors taken from the front and pushed to the back as they are updated. The main entry point is the `tick()` function, which processes all behaviors until an end-of-update marker is found.

```
void tick() {
    //Insert an end-of-update marker into the list of tasks.
    m_Behaviors.push_back(NULL);
    //Keep going updating tasks until we encounter the marker.
    while (step()) {}
}
```

One of the many benefits of this type of implementation is support for single-stepping of behaviors. This can be done directly via the `step()` function if necessary.

```
bool step() {
    Behavior* current = m_Behaviors.front();
    m_Behaviors.pop_front();
    //If this is the end-of-update marker, stop processing.
    if (current == NULL) return false;
    //Perform the update on this individual behavior.
    current->tick();
    //Process the observer if the task terminated.
    if (current->isTerminated() && current->m_Observer)
        current->m_Observer(current->m_eStatus);
    else//Otherwise drop it into the queue for the next tick()
        m_Behaviors.push_back(current);
    return true;
}
```

As for managing the execution of behaviors, the implementations of `start()` simply pushes a behavior on the front of the queue, and `stop()` sets its status and fires the observer manually.

6.4.4.3 Event-Driven Composites

The event-driven paradigm is most obvious in the composite nodes in the tree. Every composite has to make a request to the scheduler to update its child nodes rather than execute them directly.

For example, let's take a look at how a `Sequence` would be implemented.

```
class Sequence : public Composite {
protected:
    BehaviorTree* m_pBehaviorTree;
    Behaviors::iterator m_Current;
```

```
public:
    Sequence(BehaviorTree& bt);
    virtual void onInitialize() override;
    void onChildComplete(Status);
}
```

Since the composites rely on the scheduler to update the child nodes, there's no need for an `update()` function. The setup of the first child in the sequence is done by the `onInitialize()` function.

```
void Sequence::onInitialize() {
    m_Current = m_Children.begin();
    auto observer = BehaviorObserver::                          \
        FROM_METHOD(Sequence, onChildComplete, this);
    m_pBehaviorTree->insert(**m_Current, &observer);
}
```

Every subsequent child in the sequence is set up in the `onChildComplete()` callback.

```
void onChildComplete() {
    Behavior& child = **m_Current;
    //The current child behavior failed, sequence must fail.
    if (child.m_eStatus == BH_FAILURE) {
        m_pBehaviorTree->terminate(*this, BH_FAILURE);
        return;
    }
    //The current child succeeded, is this the end of array?
    if (++m_Current == m_Children.end()) {
        m_pBehaviorTree->terminate(*this, BH_SUCCESS);
    }
    //Move on and schedule the next child behavior in array.
    else {
        BehaviorObserver observer = BehaviorObserver::       \
            FROM_METHOD(Sequence, onChildComplete, this);
        m_pBehaviorTree->insert(**m_Current, &observer);
    }
}
```

Event-driven code has a reputation for being more complex, but this composite code remains very easy to understand at a glance. However, it is a little harder to trace the tree of behaviors by simply looking at a callstack, unlike first-generation BT implementations.

6.4.4.4 Event-Driven Leaf Behaviors

Many actions and conditions remain unaffected by an event-driven implementation, though some could be optimized to use an event-driven approach rather than polling. For instance, a monitoring condition that's waiting for a callback from the navigation system to terminate does not need to be updated every frame. Instead, this condition can be rewritten to return the `SUSPENDED` status, in which case the scheduler would ignore the behavior during its tick.

Upon receiving an external notification (e.g., from the navigation system), an event-driven condition could then tell the scheduler to reactivate it and process it during the next step of the update. This kind of logic requires special care and ordering of behaviors during the update, but can save a lot of unnecessary function calls.

6.5 Conclusion

In this article, you saw multiple variations of BTs, including a simple first-generation implementation to learn from, as well as a second-generation version that you can use as a starting point in your own project. A few principles recur throughout the code:

- Modular behaviors with simple (unique) responsibilities, to be combined together to form more complex ones.
- Behaviors that are very well specified (even unit tested) and easy to understand, even by nontechnical designers.

While there are always exceptions to these rules in practice, this approach works reliably for creating in-game behaviors. If you're having trouble with code duplication, explaining a bug to a designer, or not being able to understand interactions between your behaviors, then it's best to break down the code into simpler behaviors.

The original un-edited source code for this article can be found online under an open-source license [BTSK 12]. If you'd like to see the details, especially the stuff that couldn't be covered in this article, you're highly encouraged to check out the code!

References

[BTSK 12] A. Champandard and P. Dunstan. The Behavior Tree Starter Kit. http://github.com/aigamedev, 2012.

[Champandard 07] A. Champandard. "Behavior trees for Next-Gen AI." *Game Developers Conference Europe*, 2007.

[Isla 05] D. Isla. "Handling complexity in the Halo 2 AI." *Game Developers Conference*, 2005.

[PSS 11] M. Martins, G. Robert, M. Vehkala, M. Zielinski, and A. Champandard (editor). "Part 3 on Behavior Trees." Paris Shooter Symposium, 2011. http://gameaiconf.com/.