Modular Behavior Tree

Draft 8

Table of Contents

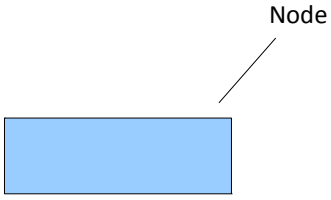
Core Concept

The Modular Behavior Tree is a collection of concepts that allow people to author behaviors for computer-controlled agents.

Instead of writing lots of complicated code in C++, or some other general purpose programming language, the Modular Behavior Tree lets you describe behaviors on a high level without having to think about pointers, memory, compilers, classes etc.

The concepts and the implementation is focused on rapid iteration and re-use.

Node

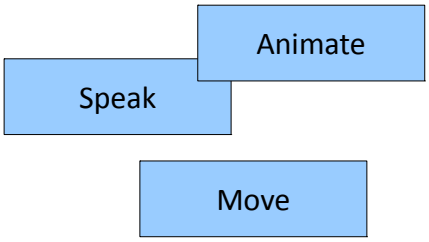


The node is the most fundamental concept you need to understand. It's like a Lego brick. You build behaviors by combining many different nodes.

All nodes have the same interface – it carries out a task that can either *succeed* or *fail*. However, it's always helpful to categorize so I'll cover a few common node patterns.

Action Node

We usually say that a node is an *action node* when it performs some sort of action. For instance, you can have one node that makes an agent play an animation, another that makes it speak, and a third that makes it move somewhere.



An action node is no different from any other node, I can't stress this enough! The “action” term is merely used to ease up communication.

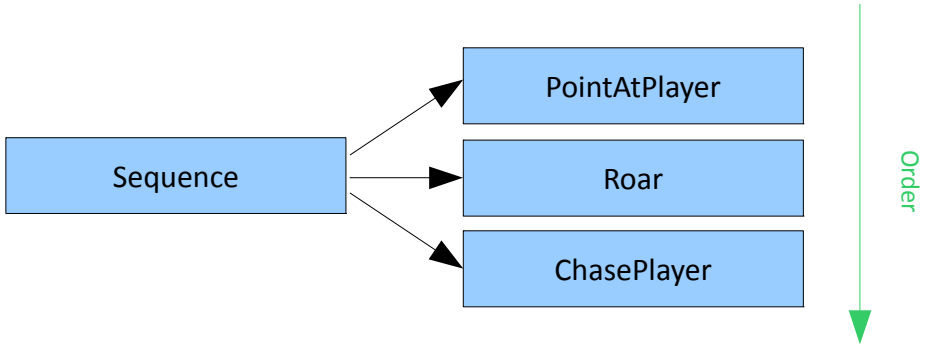
Composite Node

Imagine you want to create a monster that points at the player, roars, and then chases him or her. You'll want to perform these things in a sequence, and this is where *composite nodes* come in handy.

They contain a collection of nodes that are executed in a certain order. The nodes in this collection are referred to as children.

Sequence

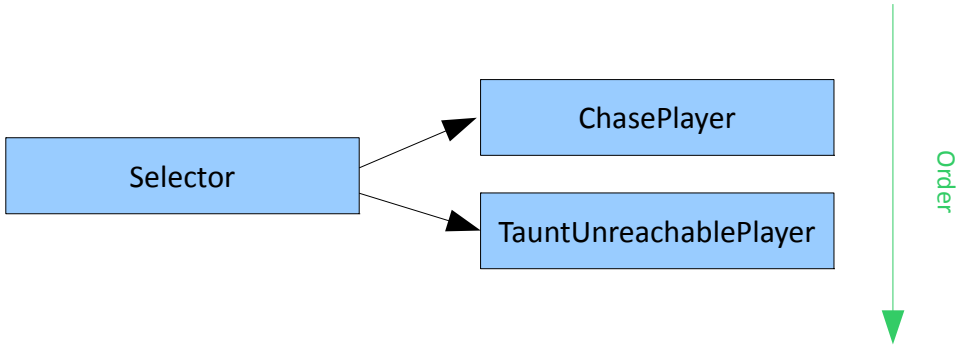
The *Sequence* node, for instance, sequentially executes its children (its collection of nodes) in order, just like in our example above with point, roar, and chase. It will first point and if that *succeeded* we start to roar. If that *succeeded* as well we start chasing. If all children *succeeded* we consider the *Sequence* itself to have *succeeded*.



If, however, any of the children *failed* we immediately consider the *Sequence* itself to have *failed,* without proceeding to the next child node.

Selector

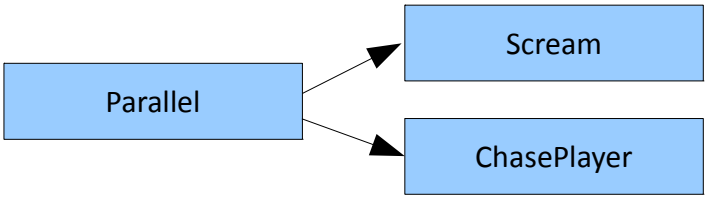
Let's imagine another scenario! This time we want our monster to chase the player, but if he can't reach him he should scream *"Come and fight me, you coward!"* This behavior can be accomplished by using a *Selector* composite node with two children (a collection of two nodes). The first child being a *ChasePlayer* node and the second child being a *TauntUnreachablePlayer* node. The *Selector* will first execute *ChasePlayer* which will fail because there's no way up to the building. The *Selector* then proceeds with the next node in its collection, which in our case is *TauntUnreachablePlayer.* This works out just fine, and the monster taunts the player.



The *Selector* will *succeed* as soon as a child node *succeeds*.
The *Selector* will *fail* if all child nodes were tried and all of them *failed*.
It's perfectly suited for trying different tactics in a game, or creating fallback behaviors in case something unexpected goes wrong.

Parallel

Imagine we want a monster that will scream and chase the player at the same time, not one before the other. You'd want them both to happen in parallel, and you can do just that by making *Scream* and *ChasePlayer* children of a *Parallel* composite node.



Decorator Node

A *decorator node* is a node which adds some sort of functionality to another node, without knowing how that other node works or what it does.

Loop

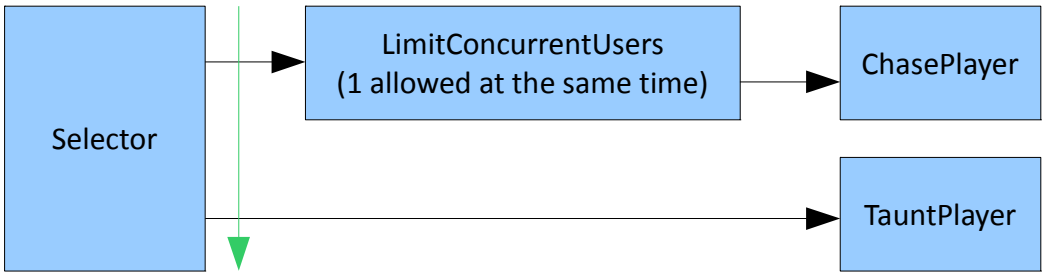
Imagine you want to play an animation 5 times in a row. Now, you could just go ahead and implement an *Animate* node that plays an animation 5 times before it *succeeds*. That's perfectly fine. But what if you later realize you want to scream 4 times in a row. Once again, you could re-implement the same looping functionality in the *Scream* node as well. However, it's strongly recommended you keep an eye out for potential re-use and implement a decorator node that does the job instead.

In our case here, we'd just create a *Loop* decorator node that has one child node. We set the Loop node to loop 5 times, and then we add an Animate node as its child node. When the Loop node is updated (or “ticked”) it will update its child node (in our case Animate). When the animation finishes playing, it will report that it succeeded. The Loop node keeps track of this and will once again tick the Animate node. After 5 times the Loop node will report that it *succeeded.*

Limit Concurrent Users

The *LimitConcurrentUsers* decorator node allows you to specify how many users should be allowed to concurrently use a node.

Scenario! Imagine you have three monsters and you want one of them to chase the player and the rest to taunt the player. To achieve this we could create this structure:



The first monster will flow through the Selector to the LimitConcurrentUsers node. The LimitConcurrentUsers node says that there are currently no one using it's child node, and let's us through to ChasePlayer. LimitConcurrentUsers now remembers that we've passed through and effectively locks the door.

The second monster will also flow through the Selector to the LimitConcurrentUsers node. Only this time, we're told there's already someone in here and we're not allowed in. The LimitConcurrentUsers node reports that a *failure.* If you read the Selector section you might remember that it will now try its next child, which in this case is TauntPlayer. The monster now taunts.

The third monster gets the same story as the second – the LimitConcurrentUsers node tells us that we can't flow through because there's already someone in there.

This is very powerful! You can tweak the amount of users allowed to chase the player simultaneously, and you can make the ChasePlayer behavior as complex or as simple as you'd like. It's quick and easy to try out ideas and you don't have to worry so much about designing up-front.

Provided Nodes

Many building blocks (state machines etc.) are already provided. Read more at http://confluence/display/C3/Modular+Behavior+Tree+Nodes+Documentation

Tree

The structures and collections of nodes we've seen in the previous section are referred to as behavior trees – because they all have a root and branches out more and more into leaves. In my visual representations they look like trees that have fallen over, but you get the point.

Markup Language (XML)

Any markup language would be fine, but we decided to describe trees using XML since we already had an XML reader and used XML files for many other systems so people in the team were comfortable with it.

The behavior trees are hot-loaded every time the user jumps into game in the editor.

Example: Monster

File: Scripts/AI/BehaviorTrees/MonsterTree.xml

One monster at a time is allowed to chase the player. The other monsters will stand around and taunt the player.

```
<BehaviorTree>
	<Root>
		<Selector>
			<LimitConcurrentUsers max=”1”>
				<ChasePlayer />
			</LimitConcurrentUsers>
			<TauntPlayer />
		</Selector>
	</Root>
</BehaviorTree>
```

C++ Implementation

You'll find all the Modular Behavior Tree code encapsulated in the *BehaviorTree* namespace.

Memory Model

The Modular Behavior Tree has a relatively small memory footprint. It accomplishes this by sharing immutable (read-only) data between instances of a tree, and by only having memory allocated for things that are necessary for the current situation.

Memory is divided into two categories: *configuration data* and *runtime data*.

Configuration Data

Let me explain how the following tree gets loaded in.

```
1     <Sequence>
2           <Communicate name=”Hello” />
3           <Animate name=”LookAround” />
4           <Wait duration=”2.0” />
5           <Communicate name=”WeShouldGetSomeFood” />
6     </Sequence>
```

When you load the behavior tree above from an XML file you'll end up with a *Behavior Tree Template* that holds all the configuration data you can see above. There will be a sequence node with four children; a communicate node, an animate node, a wait node, and yet another communicate node.

The configuration data will be animation name, duration, etc. This data doesn't change – ever.

The memory for the configuration data is allocated from the level heap and is freed on level unload when running the game through the launcher, or when the player exits game mode and goes back to edit mode in the editor.

Runtime Data

When you spawn an agent using this behavior tree, a *Behavior Tree Instance* will be created and associated with the agent. It points back to the template and contains only a few instance-specific things such as variables and timestamps (which we'll see later).

When the tree is ticked with our agent in mind it will first visit the Sequence node. The core system will see that it's the first time and allocates a runtime data object for this particular node configuration for this particular agent. Did you follow that? Each agent will get its own runtime data object when it's visiting nodes in the tree. The runtime data object persists as long as the agent is visiting a node (this can be several frames) but will be freed as soon as the agent leaves a node.

The memory for the runtime data is allocated from a bucket allocator. This is to minimize fragmentation as the runtime data is usually just a few bytes and is frequently allocated and freed, and would then make it harder to find a contiguous block of memory for large chunks of data. The bucket allocator is cleaned up on level unload.

Smart Pointers

Boost smart pointers are used throughout the modular behavior tree code. It's used to be able to pass around data safely and to avoid raw pointers as much as possible. Memory management is taken care of by the core of the system. In some places *unique_ptr* from C++11 would be a perfect fit, but we use boost's *shared_ptr* for compatibility reasons.

Implementing a New Node

Creating Your Node

```
#include <BehaviorTree/Node.h>

class MyNode : public BehaviorTree::Node
{
    typedef BehaviorTree::Node BaseClass;

public:

    // Every instance of a node in a tree for an agent will have a
    // runtime data object.
    //
    // This data is persistent from the point when a node is visited
    // until the point when it's left.
    //
    // If this struct is left out the code won't compile.
```

```
    //
    // This would be variables like 'bestPostureID', 'shotsFired' etc.
    struct RuntimeData
    {
    };

    MyNode() : m_speed(0.0f)
    {
    }

    // This is where you'll load the configuration data from the XML file
    // into members of the node. They can only be written to during the loading phase
    // and are conceptually immutable (read-only) later while the game is running.
    virtual LoadResult LoadFromXml(const XmlNodeRef& xml, const LoadContext& context)
    {
         if (BaseClass::LoadFromXml(xml, context) == LoadFailure)
              return LoadFailure;

         xml->getAttr("speed", m_speed);

         return LoadSuccess;
    }

protected:

    // Called right before the first update
    virtual void OnInitialize(const UpdateContext& context)
    {
         BaseClass::OnInitialize(context);

         // Optional: access runtime data like this
         RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

    // Called when the node is terminated
    virtual void OnTerminate(const UpdateContext& context)
    {
         BaseClass::OnTerminate(context);

         // Optional: access runtime data like this
         RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

    virtual Status Update(const UpdateContext& context)
    {
         // Perform your update code and report back whether your
         // node succeeded, failed or is running and needs more
         // time to carry out its task.

         // Optional: access runtime data like this
         RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);

         return Success;
    }

    // Handle any incoming events sent to this node
    virtual void HandleEvent(const EventContext& context, const Event& event)
    {
         // Optional: access runtime data like this
         RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

private:

    // Store any configuration data for the node right here.
    // This would be immutable things like 'maxSpeed', 'duration',
    // 'threshold', 'impulsePower', 'soundName', etc.
    float m_speed;
};

// Generate an object specialized to create a node of your type upon
// request by the node factory. The macro drops a global variable here.
GenerateBehaviorTreeNodeCreator(MyNode);
```

Exposing Your Node

To be able to use your new node you'll need to expose it to the *Node Factory*.

```
BehaviorTree::INodeFactory& factory = gEnv->pAISystem->GetIBehaviorTreeManager()->GetNodeFactory();
ExposeBehaviorTreeNodeToFactory(factory, MyNode);
```

Reporting Errors in Your Node

Use the *ErrorReporter* class to report errors and warnings in your node. It will let you log a printf-formatted message and automatically include any available information about your node, such as XML line number, tree name, node type etc.

```
ErrorReporter(*this, context).LogError("Failed to compile Lua code '%s'", code.c_str());
```

Naming Your Node

See the Best Practices chapter.

Variables

Ideally we would've implemented a general purpose variable/register/blackboard system, but since we were short on time we reused the selection variable code from the Behavior Selection Tree (BST) instead. We can statically declare variables in XML and describe how they'd change when we receive AI signals (a concept on it's own, but basically just named text messages within the AI system).

Example of an agent that moves to the target if it's visible, otherwise animates:

```
<BehaviorTree>
	<Variables>
		<Variable name="TargetVisible" />
	</Variables>
	<SignalVariables>
		<Signal name="OnEnemySeen" variable="TargetVisible" value="true" />
		<Signal name="OnLostSightOfTarget" variable="TargetVisible" value="false" />
	</SignalVariables>
	<Root>
		<Selector>
			<IfCondition condition=”TargetVisible”>
				<Move to=”Target” />
			</IfCondition>
			<Animate name=”LookAroundForTarget” />
		</Selector>
	</Root>
</BehaviorTree>
```

Lua Scripting

You can execute Lua code embedded in the tree, either to run fire-and-forget code or to control the flow in the tree. It's perfect for prototyping or extending functionality without having to create new nodes.

- ExecuteLua runs a bit of Lua code. It always succeeds.

```
<ExecuteLua code=”DoSomething()” />
```

- LuaWrapper runs one bit of Lua code before running its child, and runs another bit of Lua code once the child succeeded or failed.

```
<LuaWrapper onEnter=”StartParticleEffect()” onExit=”StopParticleEffect()”>
        <Move to=”Cover” />
</LuaWrapper>
```

- LuaGate runs a bit of Lua code and runs the child if the code returned true. It then returns the status of the child node. If the Lua code returns false or fails to execute, LuaGate will return failure.

```
<LuaGate code=”return IsAppleGreen()”>
        <EatApple />
</LuaGate>
```

- AssertLua lets you make a statement. If that statement is true the node succeeds, and if the statement is false the node fails.

```
<Sequence>
        <AssertLua code=”return entity.someCounter == 75” />
        <AssertCondition condition=”TargetVisible” />
        <Move to=”Target” />
</Sequence>
```

You'll have access to the 'entity' variable in all the Lua nodes.

The code is compiled once at level load in pure game to reduce fragmentation, and only code for behavior trees that are going to be used at one point in the level is compiled.

Timestamps

A timestamp is the knowledge of *when* something happened. For instance, this is useful if you want to query when a character was last shot at or hit, or maybe when he last could see the player. Or when he last moved to a new cover location.

```
<BehaviorTree>
	<Timestamps>
		<Timestamp name="TargetSpotted" setOnEvent="OnEnemySeen" />
		<Timestamp name="ReceivedDamage" setOnEvent="OnEnemyDamage" />
		<Timestamp name="GroupMemberDied" setOnEvent="GroupMemberDied" />
	</Timestamps>
	<Root>
		<Sequence>
			<WaitUntilTime since=”ReceivedDamage” isMoreThan=”5” orNeverBeenSet=”1” />
			<Selector>
				<IfTime since="GroupMemberDied" isLessThan=”10”>
					<MoveCautiouslyTowardsTarget />
				</IfTime>
				<MoveConfidentallyTowardsTarget />
			</Selector>
		</Sequence>
	</Root>
</BehaviorTree>
```

Optional: timestamps can be declared as exclusive to each other. For instance, TargetSpotted and TargetLost could be exclusive to one another so that the previous one is cleared out. You can't see the player and at the same time consider him lost.

Events

The AI system has something called AI signals. Fundamentally, they are named text messages sent to agents. Examples are OnBulletRain, OnEnemySeen etc. We wanted the nodes in our tree to be able to listen and react to these.

We didn't want the modular behavior tree to be too tightly coupled with the AI signals so we have an internal concept called the *modular behavior tree event.*

AI signals are picked up and converted to modular behavior tree events and dispatched to the root node, which passes them along down the running nodes in the tree.

```
<Sequence>
	<WaitForEvent name=”OnEnemySeen” />
	<Communicate name=”ThereHeIs” />
</Sequence>
```

Debugging and Tree Visualization

Slashing Agents

Set ai_DebugDraw to 0 or 1 (default is -1).

Look towards an agent and press numpad '/' (“slash”) and the agent's tree will be visualized. There are other ways of selecting an agent to start debugging:

- Call "ai_DebugAgent closest" to pick the agent closest to the camera.
- Call "ai_DebugAgent centerview" to pick the agent centermost in the camera view (same as slash).
- Call "ai_DebugAgent MrPickles" to pick the agent named MrPickles.
- Call "ai_DebugAgent" without any parameters to reset the debugged agent.

The tree is color-coded:

- White for nodes with custom information (more on that in a later chapter)
- Blue for leaf nodes (they often carry special weight when debugging)
- Dim gray for the rest

The name of the agent is at the top of the screen and there is a small green dot over the character in the world. The numbers on the left are line numbers in the XML file.

Adding Custom Debug Text

The Tree Visualizer supports custom node information. This allows us to get a more in-depth view of the currently running parts of the tree.

For instance, you can see the name of an event the WaitForEvent node is waiting for, or how much longer Timeout is going to run before it times out.

Override GetDebugTextForVisualizer to achieve this in your node.

```
#ifdef STORE_INFORMATION_FOR_BEHAVIOR_TREE_VISUALIZER
virtual void GetDebugTextForVisualizer(
        const UpdateContext& updateContext,
        stack_string& debugText) const
{
        debugText.Format("Speed %f", m_speed);
}
#endif
```

Logging and Tracing

You can quickly diagnose what's going on by tracing log messages. Output like this:

```
<Log message=”Got shot at, running away” />
```

Logging is so common that there's native support for it:

```
<Sequence>
          <QueryTPS name="CoverFromTarget" _startLog="Finding cover" _failureLog="Failed to find cover" />
          <Move to="Cover" _startLog="Advancing" _failureLog="Failed to advance" _successLog="Advanced" />
</Sequence>
```

The reserved attributes _startLog, _successLog and _failureLog are automatically read in.

The log messages will be routed through an object deriving from the BehaviorTree::ILogRouter interface, so you decide where the information ends up. For Crysis 3 we route it to the personal log – which is a short history of log messages for each agent that gets rendered to the screen for the currently “slashed” agent.

Log messages are also recorded in the AI Recorder so you can go back and forth in time to observe the order of events and what happened.

Breakpoint

If you want to break the behavior tree in Visual Studio you can add a Breakpoint node in the tree. When it's hit you'll get control in Visual Studio and you can step ahead.

Compile With Debug Information

DEBUG_MODULAR_BEHAVIOR_TREE needs to be defined. Crysis 3 does this:

```
#if !defined(_RELEASE) && (defined(WIN32) || defined(WIN64))
# define DEBUG_MODULAR_BEHAVIOR_TREE
#endif
```

Falling Through The Root

If your tree fails or succeeds all the way down through the root node you'll be notified in the console window and a list of recently visited nodes and their line numbers will be

presented.

[Error] Modular Behavior Tree: The root node for entity 'HumanSoldier' FAILED. Rebooting the tree next frame. (124) Move. (122) Selector. (121) Sequence.

As stated above, the tree will be rebooted the next frame. However, we see this as a sign that you didn't design your behavior tree to handle a situation that just occurred.

Good Practices

Consider this information as suggestions based on what seems to make things nice and easy to read. It increases the team's communication speed if the naming is easy to understand. Everyone is different and this might not apply to your case.

Naming Nodes

Name action nodes such that they sound like a word or sentence you'd use to command it to perform a task for you.

Good

- ✔ Loop
- ✔ Animate
- ✔ LimitConcurrentUsers
- ✔ ExecuteLua
- ✔ Shoot
- ✔ AdjustCoverStance

Bad

- ✗ Fast
- ✗ PathPredictor
- ✗ Banana
- ✗ Script
- ✗ ActivationProcess

Naming Timestamps

It seems to work well to name timestamps so that it sounds like something that happened, considering they all happened in the past:

- TargetSpotted
- ReceivedDamage
- GroupMemberDied