10

From Behavior to Animation

A Reactive AI Architecture for Networked First-Person Shooter Games

Sumeet Jakatdar

10.1 Introduction

First-person shooters (FPS) are a very specific and popular genre of video games. Many of the general AI principles and techniques are applicable for FPS AI, but they need to be modified to cater to the needs of fast-paced action. The challenge lies in selecting, modifying, and combining algorithms together efficiently, so that AIs can react quickly to player actions. For an AAA FPS game with strict deadlines, the AI system needs to be ready early in the development cycle, allowing content creators to start using it. The system also needs to be simple to use, data driven, and flexible enough to be able to support creating a variety of AI archetypes. The system also needs to support networked gameplay without needing any extra effort from design and animation teams.

This chapter will provide a high-level and simple overview of an AI system that could be used for a networked FPS game. We will look into modifications to well-known AI algorithms such as behavior trees (BTs) and animation state machines (ASMs), which were made to suit specific needs of the game. We will discuss ways to keep the BT simple but still reactive to high-priority events by introducing a concept called *interrupts*. Furthermore, we will touch upon animation layers that are designed to handle aiming and shooting animations independent of the BT. Here, we will discuss modifications to traditional ASMs to help network animations, allowing cooperative or competitive gameplay across the internet.

We will also look into a Blackboard system to solve behavior and animation selection problems. Here we will introduce techniques that will help keep the Blackboard attributes up-to-date by using a *function-based* approach.

Whenever applicable, we will provide pseudocode and real examples to better explain the concepts presented. Many of you probably know these techniques individually. Hopefully, by reading this chapter, you will get an idea of how they can work together for networked FPS games.

10.2 Client-Server Engine and AI System

We will assume that our game engine is built using a client-server paradigm, where the server is authoritative in nature. This means that server dictates the position, orientation, and gameplay logic for all game entities.

From time to time, the client receives a *network snapshot* from the server for all of the entities. While processing this snapshot, each entity's position and orientation will be corrected to match the server. The frequency of these snapshots depend upon the network bandwidth and CPU performance requirements of the game. Between network snapshots, the client interpolates entities for smoother 60 fps gameplay. It is important to keep the size of the network snapshots (measured in bits) small to support hundreds of entities and to avoid packet fragmentation. The upper limit for network snapshots is usually referred to as the *maximum transmission unit* (MTU), and the recommended size is approximately 1200–1500 bytes for Ethernet connections. To be able to achieve the smallest possible MTU, we will look for opportunities to infer logic on the client side by sending across minimalistic information about entities.

The AI system is just one part of this engine. In addition, the server is responsible for handling many other systems, such as controller logic, player animation, physics, navigation, and scripting. Similarly, the client is responsible for handling effects, sounds, and most importantly rendering.

Figure 10.1 shows a high-level overview of the system and important AI modules in the engine. The BTs are responsible for choosing behavior using the knowledge stored in the Blackboard (a shared space for communication). The ASM is responsible for selecting animations and managing the animation pose of the AI. The client-side ASM has the same responsibilities as the server, but the implementation is slightly different from the server, as it does not make any decisions but merely follows the server. We will explore the details of the ASM further in this chapter.

At first, we will concentrate on the AI system on the server side. After we are done with the server, we will move on to the client side. We will finish by looking at the networking layer between the server and client AI systems. The pseudocode provided in this chapter will look very similar to C, but it should be fairly easy to translate these concepts in an object-oriented coding environment.

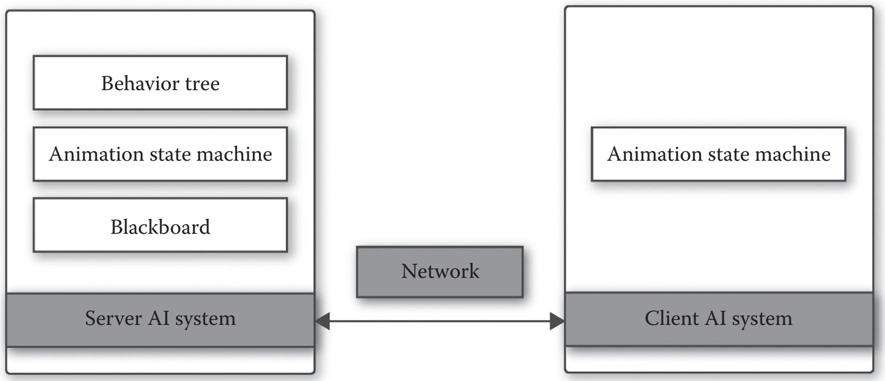

Figure 10.1

Client-server engine and AI system.

10.3 The AI Agent

Before moving on to create various AI modules, we will define the `AIAgent` structure that lies at the core of our AI system, as shown in Listing 10.1. All three of our AI modules will be contained within this structure. Both server and client will have their own representation of the `AIAgent`.

Listing 10.1. Defining `AIAgent`.

```
struct AIAgent
{
    // Blackboard
    // Behavior Tree
    // Animation State Machine
    // ...
};
```

10.4 The Blackboard

The Blackboard holds key information about the AI agent, which can be manipulated and queried by other AI modules for choosing behaviors, animations, and so on. In some cases, the Blackboard is also used as a communication layer between modules.

Our implementation of the Blackboard will support three different types of variables; strings, floats, and integers. We will call them *value-based* variables. We will also support a special variable, referred in this chapter as *function-based*.

10.4.1 Value-Based Variables (Strings, Integers, and Floats)

Table 10.1 creates four Blackboard variables for the human soldier archetype. From the first three Blackboard variables, we can infer that the human soldier is currently *standing*.

Table 10.1 Sample Blackboard for the Human Soldier Archetype

Variable	Type	Current_value	Update_function
Weapon	string	rifle_longrange	null
Number_of_bullets	int	50	null
Stance	string	stand	null
Angle_to_player	float		getangletoplayer

He or she is holding a weapon, a *long range rifle* with *50 bullets* in its magazine. There is no specific range, or possible set of values associated with these variables. So, for example, the number of bullets can be negative, which might be illogical. In such cases, it is the responsibility of other AI logic to handle that situation gracefully. However, adding support for ranges or enums should be fairly straightforward and is recommended in a final implementation.

10.4.2 Function-Based Variables

The first three Blackboard variables in Table 10.1 will be modified by external systems. For these variables, the Blackboard can be considered as some form of storage. But sometimes, we need the Blackboard to execute logic to calculate the latest value of a given variable. This is the purpose of the `update _ function`.

When a human soldier needs to turn and face the player, it will query the Blackboard to get the latest value of the `angle_to_player`. In this case, the Blackboard will execute the `getangletoplayer` function and return the latest yaw value. We call these types of variables *function-based*, as a function is executed when they are requested.

10.4.3 Implementation of the Blackboard

Let us look at some implementation details for our Blackboard. Listing 10.2 shows the implementation for `BlackboardValue`, which will essentially hold only one of the three types of values: a string, float, or integer. We have also created an enum to indicate the type of Blackboard variable and use it in the `BlackboardVariable` structure. Remember, the `update_function` is only defined and used for *function-based* Blackboard variables.

Listing 10.2. Definition of `BlackboardVariable` and `BlackboardValue`.

```
struct BlackboardValue
{
    union
    {
        const char* stringValue;
        int intValue;
        float floatValue;
    };
};
```

(*Continued*)

```
enum BlackboardVariableType
{
    BLACKBOARD_INT,
    BLACKBOARD_FLOAT,
    BLACKBOARD_STRING,
};
typedef BlackboardValue
    (*BlackboardUpdateFunction)(AIAgent *agent);

struct BlackboardVariable
{
    const char* name;
    BlackboardValue value;
    BlackboardVariableType type;
    BlackboardUpdateFunction updateFunction;
};
```

The `Blackboard`, in Listing 10.3, is essentially a collection of many `BlackboardVariable` structures stored as a fixed-size array. We also add `numVariables` to easily keep track of the actual number of variables in use. Finally, a Blackboard is added to the `AIAgent` definition. When an AI agent spawns, a script is responsible for populating and assigning default values to the AI agent's blackboard.

Listing 10.3. Setting up the Blackboard for the `AIAgent`.

```
struct Blackboard
{
    BlackboardVariable variables[MAX_BLACKBOARD_VARIABLES];
    int numVariables;
};

struct AIAgent
{
    // AI agent's blackboard
    Blackboard blackboard; }
```

Listing 10.4 shows an example of creating one of the Blackboard variables for a human soldier.

Listing 10.4. `BlackboardVariable` example for human soldier.

```
BlackboardValue value;
Value.name = "weapon";
value.stringValue = "rifle_longrange";
value.type = BLACKBOARD_STRING;
value.updateFunction = null;
```

10.5 The Behavior Tree

At the core of an AI agent's behaviors is a way to select various actions. A BT consists of many of these actions arranged in a tree format and inherently provides priority in its hierarchy. Every action will have prerequisite conditions, which are needed to be satisfied for that action to be chosen. If you need to get familiar with the basics of BTs, please refer to chapters in earlier *AI Game Pro* and *AI Game Programming Wisdom* books (Isla 2005, Champandard 2008, Champandard and Dunstan 2013). You can also refer to the many online articles available at *AiGameDev.com* (AiGameDev 2015) and *Gamasutra.com*. Here, we will focus on some specific modifications to known BT paradigms.

Our AI agent is going to have many behaviors. Every behavior will consist of one or more actions. However, it is important to understand that only one action will be active at a given time. For example, the human soldier behavior for suppressing the enemy can be composed of two actions; first the *throwing grenade* action and then the *charging toward the player* action. We will design the BTs with this consideration in mind.

10.5.1 Creating a Behavior Tree for the AI Agent

We will start by defining a `BehaviorTree`, and then look at the `BehaviorTreeNode` definition.

Listing 10.5. Setting up `BehaviorTree` for `AIAgent`.

```
struct BehaviorTree
{
    const char* name;
    BehaviorTreeNode nodes[MAX_BT_NODES];
    int numNodes;
};

struct AIAgent
{
    // AI agent's own blackboard
    Blackboard blackboard;
    // pointer to behavior tree definition
    const BehaviorTree *behaviorTree;
    //...
}
```

As seen in Listing 10.5, the `BehaviorTree` itself is just a fixed-size array of `BehaviorTreeNode` nodes. This size depends upon the complexity of various AI agent's behaviors, but generally it is a good idea to keep the size under 1000 nodes. If the tree gets bigger than this, then it might be worth looking at the granularity of condition nodes and finding opportunities to consolidate them.

We add one more variable, `numNodes`, to easily keep track of number of nodes used in a `BehaviorTree`. Another important assumption made here is that the "root node" will always appear first, at the 0th index of the nodes array. Each BT also has a unique name, which is used only for debugging purposes.

You would have noticed that there is only one definition for nodes, `BehaviorTreeNode`. In the next section, we will explore how it is used for supporting different types of nodes.

We saw earlier that, every AI agent stores its own copy of the Blackboard, but this is not the case with a BT. At runtime, only one, nonmodifiable copy will reside in memory for a given BT definition. Every AI agent who uses this BT will store a pointer to this definition and refer to it time and again to choose an action.

10.5.2 Behavior Tree Nodes

Now that we have represented the BT in the `BehaviorTree` structure, let us move on to representing actual BT nodes, as shown in Listing 10.6.

Listing 10.6. Implementing Behavior Tree nodes.

```
enum BTNodeType
{
    BT_NODE_ACTION,            // action
    BT_NODE_CONDITION,         // condition
    BT_NODE_PARALLEL,          // parallel
    BT_NODE_SEQUENCE,          // sequence
    BT_NODE_SELECTOR,          // selector
    //...
};

enum BTNodeResult
{
    BT_SUCCESS,
    BT_FAILURE,
    BT_RUNNNING,
    //...
};

typedef BTNodeResult(*BTFunction)
    (AIAgent *agent, int nodeIndex);

struct BehaviorTreeNode
{
    const char* name;
    // type of the node
    BTNodeType type;
    // parent node index
    int parentNodeIndex;
    // children nodes
    int childrenNodeIndices[MAX_BT_CHILDREN_NODES];
    int numChildrenNodes;

    // condition node attributes
    BTFunction condition;
    // action node attributes
    BTFunction onActionStart;
    BTFunction onActionUpdate;
    BTFunction onActionTerminate;
};
```

We start by specifying the types of BT nodes we want to support using the `BTNodeType` enum. Typically, a BT supports conditions, actions, parallels, selectors, and sequences. You can add support for additional nodes, but the core idea of the implementation remains the same.

As we have only one definition for all of the nodes, the type will be used to figure out how the node is processed during the BT evaluation.

The child and parent relationship of a `BehaviorTreeNode` is defined by storing `parentNodeIndex` and an array of `childrenNodeIndices`. This setup is going to make it easy to traverse up and down the tree. It is possible that certain types of nodes are not allowed to have any children at all. In our implementation, the leaf nodes, action, and conditions are not allowed to have children. If we need to have a condition to be successful before executing further, then we can just use the sequence node with that condition as its first child. Similarly, two actions can be put together under one sequence if one action needs to follow another action, only if the first one was successful. This approach helps keep the tree simple and readable, and we always know to reevaluate the tree from the root if an action fails.

Actions and conditions can only be part of a composite node in our BT, which guarantees that there will not be any logical issues with our tree. Allowing conditions to have children adds ambiguity about the nature of that node. Unless we add more attributes to the condition, it will be difficult to decide if the condition node needs to be treated as a sequence or a parallel. We rather choose to do it in more elegant way by using the composite nodes.

Only the composite node types such as selectors, parallels, and sequences are allowed to have children. So, if we need to create a behavior consisting of one condition and one action, then it would be represented using a sequence or a parallel node. Listing 10.7 shows an example of a sequence node for a human soldier.

Listing 10.7. Turn behavior for a human soldier.

```
{
    "name": "turnBehavior",
    "type": "sequence",
    "children":
    [
        {
            "type": "condition",
            "name": "turnCondition",
            "condition": "isPlayerBehind"
        },
        {
            "type": "action",
            "name": "turnAction",
            "onActionStart": "sayDialogue"
        }
    ]
}
```

It is worth mentioning that there is only one parent for a node in our implementation. We will validate all these requirements while populating BTs at runtime.

Just knowing the relationship between BT nodes is not enough. We need to be able to associate some logic to nodes, so that we can execute that logic and find out the result. This is where `BTFunction` comes in handy. To start with, we only need functions for two types of nodes: conditions and actions. For action nodes in particular, we need three functions: `onActionStart`, `OnActionUpdate`, and `onActionTerminate`.

These functions can be used to execute additional gameplay logic when an action is active. If any one of these functions returns `BT _ FAILURE`, then we will have to reevaluate the tree.

To explain this a little better, we go back to the human soldier example again. Let us say that one of the behaviors for our human is a *Turn Behavior*. If the player is behind, our agent will turn to face the player and say a line of dialogue while turning. For this behavior, we will need three nodes in our BT. Listing 10.7 shows a raw behavior tree asset that we will use to populate the human soldier's BT at runtime. Many of you might recognize the JSON format. If you plan to write your own tools to create and edit BT, then a format like JSON is worth considering as there are many available parsers and editors available which might save you time.

The `isPlayerBehind` condition will make use of the Blackboard variable `angle _ to _ player` to decide if the player is behind. If the condition is successful, then the sequence will continue and select `turnAction` and call the `sayDialogue` function once, as the action starts.

10.5.3 Handling Parallel Conditions

The only node in our BT that will last for a longer period of time is the action node. In our case, an action will continue while the corresponding animation state is active. We will look into animation states a little later in this chapter.

For some actions, it is important that all of the prerequisite conditions remain valid for the duration of the action and not just at the start. We will achieve this by using parallel nodes. Let us say our human soldier has another behavior called *Relaxed Behavior*. He or she continues to remain relaxed until he or she sees a threat and then switches to another behavior. So, when he or she is relaxing, we need a way to continue to make sure that there is no threat.

In Listing 10.8, notice that the `relaxedBehavior` node is a parallel node. This node will act exactly the same as a sequence node while evaluating the tree to find an action.

Listing 10.8. Relaxed behavior for a human soldier using parallel nodes.

```
{
    "name": "relaxedBehavior",
    "type": "parallel",
    "children":
    [
        {
            "type": "condition",
            "name": "checkThreat",
            "condition": "noThreatInSight"
        },
        {
            "type": "action",
            "name": "relaxedAction"
        }
    ]
}
```

Although, once the relaxedAction is chosen and started, the parallel node will be used to populate all of the conditions that are needed to remain valid throughout this behavior.

In a full-blown BT, there will be many parallel nodes active for a given action. We will need a way to store all active parallel nodes for every AI agent independently, so we can execute them every frame and make sure that they are valid to continue executing the current action.

Listing 10.9 extends the AIAgent definition to support active parallel nodes. Once the action is chosen by the BT, then the PopulateActiveParallelNodes function will be called to generate the list of all parallel nodes that were executed on the way to the current action nodes. This is achieved by traversing back up the tree, all the way to the root node. We make use of parentNodeIndex to quickly traverse back up the tree. While updating the action, the BT logic will go through all the active parallel nodes and process their children condition nodes. If any of those conditions are invalid, then the action is

Listing 10.9. Extending AIAgent to support parallel conditions.

```
struct AIAgent
{
    // AI agent's own blackboard
    Blackboard blackboard;
    // pointer to behavior tree definition
    const BehaviorTree *behaviorTree;
    // active parallel conditions
    int  activeNodes[MAX_ACTIVE_PARALLELS];
    int  numActiveNodes;

    //...
}

void PopulateActiveParallelNodes
    (AIAgent *agent, int actionNodeIndex)
{

    const BehaviorTreeNode* btNode
          = &agent->behaviorTree->nodes[actionNodeIndex];
    agent->numActiveNodes = 0;

    while(btNode->index != 0)
    {
        int parentNodeIndex = btNode->parentNodeIndex;
        btNode
           = &agent->behaviorTree->nodes[parentNodeIndex];

        if(btNode->type == BT_NODE_PARALLEL)
        {
          agent->activeNodes[agent->numActiveNodes]
                                   = btNode->index;
          agent->numActiveNodes++;
        }
    }
}
```

considered to be invalid. At this time, the BT will be reevaluated from root node to find a new, more suitable action.

This approach helps to create an extremely efficient BT, as we do not evaluate the complete tree every frame. It also keeps the reactiveness of the tree intact. However, extra care has to be taken while designing the tree, so that all the possible conditions are accounted for. This is a tradeoff that we could live with, as efficiency is very important.

Sequence nodes can be handled in very similar fashion with a small implementation difference. There is no need to store any active sequences, as we are only interested in advancing the immediate parent sequence when an action is complete.

10.5.4 Handling Interruption Events in the Behavior Tree

After implementing the behavior tree architecture, we still have one more problem to deal with. We need a way to forcefully reevaluate the tree for certain key events, which can happen any time. In the case of a human soldier, *damage* is a high-priority event. Most of the time, we want humans to immediately react to *damage*.

To ensure that the AI will always react to *damage,* we found ourselves duplicating the same parallel condition for most of our actions. This approach works, but it is not ideal as it makes the tree unnecessarily complex. To handle this problem, we will introduce a new concept called interrupts.

Interrupts are basically events that will force the behavior tree logic to immediately invalidate the current action and initiate a full reevaluation of the tree. Interrupts themselves know nothing about AI behaviors in the tree at all. They usually last only until the next update of the BT.

When an interrupt occurs, the full reevaluation update should lead us to an action that was meant to be chosen when this interrupt is present and other conditions are met. Hence, some condition nodes need to have additional parameters so that they will only be valid during an interrupt. To achieve this, we need to add another attribute to the condition nodes, as shown in Listing 10.10.

Listing 10.10. Adding support for interrupts to `BehaviorTreeNode`.

```
struct BehaviorTreeNode
{
    //...
    // only used by condition nodes
    const char* interrupt;
};
```

The `interrupt` attribute is optional and only used by condition nodes. However, if it is specified for a condition node, then that node will only be evaluated when the specified interrupt event is being processed.

This simple modification helps reduce the complexity of the tree greatly. This approach works well when there are only handful of interrupt events associated with the behavior tree. It is possible that interrupt events are not mutually exclusive and may occur on the same frame. This problem can be solved by having a predefined priority list of interrupt events and only processing the highest interrupt event on that list.

A good optimization is to ignore an interrupt if none of the nodes in a BT refer to it. For this to work, you can populate a separate list of referenced interrupts while creating the tree at runtime.

10.6 Animation State Machine

Now that we have a system to choose an action, it is time to figure out how our AI agent will be animated. Let us look at two basic types of animations we need to support for our AI agents in our ASM.

A *full-body* animation forms the base pose of the AI. In most cases, an AI agent needs to only play one *full-body* animation. This is not a limitation of the animation system but rather a choice we made to keep things simple. Then comes the *additive* animation type, which modifies the base pose. Contrary to the *full-body* animation, an AI agent can play multiple additive animations at a time. In fact, we will make use of *additive* animations to allow our agents to aim and shoot at the player. Let us start with the problem of selecting animations.

10.6.1 Animation Tables

The most important job of the ASM is to select animations, and it does so by using Animation Tables (ATs).

ATs can be thought of as a simple database of all possible animations, given a set of predefined conditions. These conditions are nothing but Blackboard values we defined earlier in our system. In our ASM, every animation state will have one or more ATs associated with it.

Table 10.2 shows one of the ATs for our human soldier. We will use this table to find one *full-body* animation when the human soldier is in an idle animation state. There are two types of columns for ATs. One is an *input column*, and another is *output column*. *Input columns* are helpful to form a query of Blackboard variable values. This query is then used to find a *first fitting row* by matching the value of each variable against that row. Once a row is found, the AT will return the corresponding entry in its animation column.

For example, if a human soldier is `crouching` and holding a `shotgun` in his hands, then the AT will end up choosing the `shotgun _ crouch _ idle` animation by selecting row number 1. It is that simple!

You might have noticed the "–" symbol in Table 10.2. This dash signifies that the value of that Blackboard variable can be ignored while evaluating that row. This is very useful, as we can always have a fallback row, which will make sure that we always find an animation to play. In this example, the fifth row is a fallback row. We also have an explicit column for *row numbers*. This column is actually not stored as part of the ATs, but it is there to explain another concept later in this chapter (so for now, let us ignore it).

Table 10.2 Sample "Idle" Animation Table for Human Soldier

Row	Stance	Weapon	Animation
0	stand	shotgun	shotgun_stand_idle
1	crouch	shotgun	shotgun_crouch_idle
2	–	shotgun	shotgun_prone_idle
3	prone	–	prone_idle
4	crouch	–	crouch_idle
5	–	–	stand_idle

Table 10.3 Aim Table for the Human Soldier

Row	Weapon	anim_aim_left	anim_aim_right	anim_aim_up	anim_aim_down
0	shotgun	shotgun_aim_left	shotgun_aim_right	shotgun_aim_up	shotgun_aim_down
1	–	rifle_aim_left	rifle_aim_right	rifle_aim_up	rifle_aim_down

As shown in Table 10.3, ATs can have multiple *input* and *output columns*. In fact, we use this to return more than one animation for the aiming and shooting of *additive* animations. In the case of the *Aim Table*, the AT will return *left*, *right*, *up*, and *down* animations. It will be the responsibility of ASM to figure out the blend weights for these animations to achieve the desired aiming pose.

Let us look at how we can implement ATs in our AI system, as shown in Listing 10.11.

Listing 10.11. Animation Table column and row definitions.

```
enum AnimationTableColumType
{
    AT_COLUMN_INPUT,
    AT_COLUMN_OUTPUT
};

struct AnimationTableColumn
{
    const char* blackboardVariableName;
    BlackboardValue expectedValue;
    AnimationTableColumType type;
};

struct AnimationTableRow
{
    AnimationTableColumn columns[MAX_COLUMNS_PER_ROW];
    int numColumnsInUse;
};

struct AnimationTable
{
    const char* name;
    AnimationTableRow rows[MAX_ROWS_PER_TABLE];
    int numRowsInUse;
};
```

Essentially, `AnimationTableColumn` stores a Blackboard variable name that it refers to and an `expectedValue` of that variable to compare against. We can easily look up the type of Blackboard variable in the `blackboard` array in our `AIAgent` definition. In the final implementation however, it is ideal to use hashes to avoid string comparisons. The structure `AnimationTableRow` is a collection of columns, and finally `AnimationTable` is a collection of rows.

Depending on the number of AI agents alive, the amount of queries can be a performance concern. To help improve performance, we can also implement a caching mechanism for the results.

10.6.2 Animation States

As mentioned earlier, every animation state is responsible for finding and applying one *full-body* animation. If it cannot find one, then there is a bug that needs to be fixed.

Depending on the state, we also need *additive* animations to modify the base pose. *Additive* animations are optional and may not be used by every animation state. In this chapter, we will assume that there are only two possible *additive* animations per animation state. One for aiming and another for the shooting animation.

To achieve this, we will need to refer to three different ATs in our animation state, as shown in Listing 10.12.

Listing 10.12. `Animation State` definition.

```
struct AnimationState
{
    const char* name;
    const AnimationTable *fullBodyTable;
    const AnimationTable *aimTable;
    const AnimationTable *shootTable;
};
```

10.6.3 Choosing the Animation State for an AI Agent

For a given action, we need to choose an animation state. We could infer the animation state by using the Blackboard variables alone, but this approach is complex and potentially unreliable. However, this approach is successfully used in many other games, and it does work very well. Keeping up with the theme of simplicity, we opt for another, rather easy solution. In our system, the BT action is responsible for choosing the corresponding animation state.

As shown in Listing 10.13, we added an animation state index to the `BehaviorTreeNode` definition. When an action is selected by the BT, a corresponding animation state will be requested by the BT and then immediately processed by the ASM.

Listing 10.13. Animation state for the current BT action.

```
struct BehaviorTreeNode
{
    // ...
    // used by action nodes to request animation state
    int animationStateIndex;
};
```

10.6.4 Managing Animations with the ASM

Given an animation state, we are now equipped to find animations. Although, once they are found, we will need to apply those animations to our agent. In the case of aiming with the given aim animations, we would need to calculate blend weights based on the direction to the player. In the case of shooting, we would need a way to select one of the animations based on the weapon agent is holding. We will achieve this by adding three functions to our ASM definition. These functions will query the ATs when the animation state is changed. They will also manage blend weights for the selected animations based on specific gameplay logic.

Listing 10.14. Definition of the ASM.

```
typedef void(*ATFunction)(AIAgent *agent, AnimationTable* table);

struct AnimationStateMachine
{
    AnimationState states[MAX_ANIMATION_STATES];
    int numAnimationStatesInUse;

    ATFunction fullBodyAnimUpdate;
    ATFunction aimAnimUpdate;
    ATFunction shootAnimUpdate;
};
```

As shown in Listing 10.14, we have created an `AnimationStateMachine`, which is essentially a collection of many animation states. We have also added three functions for managing animations for all three different animation tables.

Listing 10.15. Storing current animations and Animation State for an `AIAgent`.

```
struct AIAgent
{
    //...
    AnimationStateMachine *animationStateMachine;

    int currentStateIndex;
    int currentFullBodyRowIndex;
    int currentAimRowIndex;
    int currentShootRowIndex;
    //...
};
```

Finally, in Listing 10.15, we add a reference to the ASM. Additionally, we store the indices of the current rows we got our animations from in `currentStateIndex`. This will come in handy when we look at networking AI animations later in this chapter.

10.6.5 Transitions

It is trivial to add the concept of a transition to our ASM. Transitions are very similar to the animation states, except they are not requested specifically by a BT action. They are chosen by the ASM while transitioning from one animation state to a state requested by the BT. The BT itself is unaware of transitions and leaves this responsibility to the ASM completely. The transition can have their own *full-body*, *additive* layers, and related ATs. While searching for an animation in an AT for a transition, it is acceptable if no fitting animation is found. In that case, the ASM will just blend animations from the previous to next state directly. This helps the animation team, as they only add transition animations where they fit and rely on animation blending in other cases.

10.6.6 Aiming and Shooting

While designing the ASM in the earlier section, we gave the responsibility of managing the aiming and shooting logic to the ASM. In our case, using the logic in `ATFunction`, the ASM will decide to aim and shoot independent of the BT action based on the existence of tables and their respective functions. This is where our FPS version of the ASM is slightly different from traditional state machines.

This method helps to keep our BT simpler, as it will not have to worry about managing shooting and aiming. The BT can still influence animation selection for aiming and shooting by changing Blackboard variables. We can also disable the aiming and shooting logic completely if needed. In the case of an FPS game, whenever possible, the AI agents shoot the players, or choose to perform melee combat at a close range. This solution solves the problem in the ASM instead of the BT, making it easier for the BT to handle high-level logic.

In some games, shooting needs to be controlled as a behavior in the BT. In such cases, this approach may not be suitable as it does not give fine-grain control over shooting using behaviors. One suggestion is to split the shooting and aiming logic into another, lightweight state machine. Then this state machine can be controlled independently by adding more attributes and logic to the BT actions.

10.6.7 Animation Alias Tables

So far, whenever we referred to an animation in an AT, we were actually referring to an Animation Alias. An AT is purely a one-to-many mapping, responsible for choosing one of the many variations of a given animation. This simple table empowers animators to create a lot of animation variety by truly staying independent of AI logic and behaviors. Table 10.4 shows an example of an AAT for a human soldier.

We complete our `AIAgent` definition by adding a reference to an `AnimationAlias Table` in Listing 10.16.

At runtime, we can allow switching between different AATs. In fact, we can use them to change the AI's look and feel in order to make them more interesting. For example, when a human soldier is shot, we can switch to another AAT with wounded animations. This is achieved by using one default AAT and another override AAT, as seen in Listing 10.16.

Table 10.4 Example of Animation Alias Table for a Human Soldier

animation_alias	variation 0	variation1	variation2
rifle_idle	rifle_idle_lookaround	rifle_idle_smoke	rifle_idle_checkgun

Listing 10.16. Adding Animation Alias Tables (AATs) to an `AIAgent`.

```
struct AIAgent
{
    //...
    AnimationAliasTable *aliasTableDefault;
    AnimationAliasTable *aliasTableOverride;
    //...
};
```

We also allow multiple active layers of tables and search for animations starting with the aliasTableOverride table first. If no override animation is found, then we will fall back to the aliasTableDefault table. This allows creating smaller batches of animation variations.

It is important to validate all of the animations within one AnimationAlias to ensure that they are compatible with one another. Some of the validations include animation length, animation event markers, and most importantly positions of certain key bones relative to the root bone.

10.7 Networking AI

Players connect to the game as clients, and all of the systems we have created so far work only on the server side. At this point, we need to add a network component to our AI system to see the AI in action on the client. As mentioned earlier, we will assume that an entity's position and orientation are already parts of the general network layer. In the AI system, we are more concerned about animations as no generic animation networking layer exists in the engine otherwise.

To be able to achieve this, we will need to send across enough information to the client so that it can replicate the animation pose of the AI. This is where the ASM and ATs come in very handy, as the definitions for both are available on the client as well.

Table 10.5 lists the minimized AIAgent data that are sent over to the client, which is used to choose the same animations as the server. First is currentStateIndex, which allows the client to use the same animation state as the server. Now, the client can refer to the same AT tables as the server though the selection animation state, but it does not have any ability to choose animations yet.

On the server, ATs choose animations using a query of Blackboard variables. Unfortunately, there is no Blackboard available on the client, as our server handles all the decision-making authoritatively. If we could somehow tell the client the row number in the table, then we can look up the animation directly without needing a Blackboard at all. This is exactly why we send over row indices for all three ATs to the client: *full-body*, *additive* aim, and shoot.

With this setup, it is guaranteed that the client will always choose the same animations as the server. We need to run the same ATFunction on the client to be able to apply the chosen animations in the same way as server. Mostly, the server and client versions of ATFunction are very similar to each other in this case.

Using the row indices, we are merely choosing animation aliases and not actual animations. It is critically important that both server and client choose the same animation variation for a given animation alias. If this does not work properly, then the server and client may generate a different animation pose which can result in bugs. Let us take an example of a player

Table 10.5 Animation State Machine Data Sent Over the Network to the Client AI System

Data	Number of Bits
currentstateindex	8 bits (up to 256 states)
currentfullbodyrowindex	7 bits (up to 128 rows)
currentaimrowindex	4 bits (up to 16 rows)
currentshootrowindex	4 bits (up to 16 rows)

shooting and damaging the AI. The bullet collision will be performed on the server using the animation pose on the server. If the animation chosen is different on the client as compared to server, then sometimes players might see that they are clearly hitting the AI on client, but in fact, on the server they are not. We will have to make sure that this never happens.

We could send an extra couple of bits for animations to solve this problem, but a more optimal solution is using a deterministic random seed, which results in the same variation on both ends.

10.8 Conclusion

Presented in this chapter is a basic AI system which supports a variety of behaviors, animations, and networked gameplay. There are many more aspects to this system that we could not discuss in this chapter, but hopefully you now have a solid idea of the foundation that can be extended to match your own needs.

Let us quickly recap what we covered in this chapter. We started with laying down the design and technical requirements and then moved on to implement a Blackboard system. In addition to basic *value-based* variables, the Blackboard supported *function-based* variables to calculate up-to-date values. We used BTs as our behavior selection algorithm and also looked at how parallel conditions can be implemented to incorporate reactivity without complete evaluation of the tree every frame. We also introduced *interrupts* to handle some common high-priority events.

Then we moved on to Animation Tables, which are smart animation databases capable of selecting animations based on Blackboard variables. We also added support for selecting more than one animation and used it for shooting and aiming animations. Next up was the ASM, which made use of Animation Tables to network AI animations. Finally, we created an `AIAgent` definition that keeps track of all these systems for every AI in the game. At this time, we considered our basic AI system complete and ready for prime time.

Although there are a myriad of AI architectures and techniques available, it is important to choose appropriate ones by keeping simplicity and flexibility of a system in mind, at least in the early stages of development. Once you choose the right architecture, you can always add more paradigms to your system as you go. Usually the best approach is to start with proven, simple AI techniques and mold them to your needs in order to get something working quickly. Then you can iterate based on feedback from the team and the changing needs of the game.

References

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

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

Champandard, A. J. 2015. Behavior Trees for Next-Gen Game AI AiGameDev.com. 2015. http://www.aigamedev.com/.

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