Chapter No. 6
"Decision Making"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.6 "Decision Making"
A synopsis of the book’s content
Information on where to buy this book

About the Author

David Young is a professional software engineer who works within the game industry. He started his career at NASA's Deep Space Network and later moved to NASA's Jet Propulsion Laboratory for the Curiosity rover mission. After leaving NASA, he worked on the platform that powers Riot Game's League of Legends. David is pursuing a PhD at the University of Southern California, focusing on graphics research in the field of real-time hair rendering and simulation.

I would like to thank my wife; without her support, this book would not have been possible.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Learning Game AI Programming with Lua

Game AI is a combination of decision making and animation playback. Although classic or academic AI is focused solely on finding the right decision to make, game AI is responsible for acting on numerous decisions over time. Treating game AI independent from animation is a classic mistake that this book avoids by integrating animation and locomotion systems immediately into the AI systems. This subtle difference of decision making and execution changes many of the aspects that game AI programmers have to focus on.

The other large issue with game AI is regarding the specific needs and implementation strategies that are genre-specific. In order to prevent a watered-down approach, this book focuses on one specific genre, which is the first- and third-person action genre. Limiting the AI to this decision allows for an in-depth, tutorial-based approach of creating a full AI system. The overall goal of this book is to create an AI sandbox composed of professional level C and C++ open source libraries exposed to an AI system scripted in Lua.

What This Book Covers

Chapter 1, Getting Started with AI Sandbox, begins with learning the overview of how projects are set up as well as how the Lua script interacts with C++ code. Here, the beginnings of the AI sandbox are built from open source technologies, starting with a framework that integrates Lua, Ogre3D, OpenSteer, and Bullet Physics.

Chapter 2, Creating and Moving Agents, starts off with examples of the lowest layer of AI interaction with the world, local steering, and movement. Here, agent seeking, avoiding, and group movement are introduced into the sandbox through the use of the OpenSteer library.

Chapter 3, Character Animations, continues with the AI sandbox by exposing Ogre3D's animation playback and resource handling of Lua scripts. Low-level structures for controlling animation clips, animation state machines, and layered animations are integrated into the sandbox.

Chapter 4, Mind Body Control, combines animation handling with local steering and agent movement. Two different approaches toward mind and body interactions will be implemented. The first will focus on the latency between agent decisions and actions, while the second approach will focus on the perceived quality of the agent's actions.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Chapter 5, *Navigation*, builds up from local movement and moves on to long distance movement and planning. Navigation mesh generation provided by the Recast library will be integrated into the AI sandbox in order to allow A* pathfinding provided by the Detour library.

Chapter 6, *Decision Making*, adds intelligence to the choices the AI agents make. Different data structures and approaches to creating modular and reusable decision logic are covered through Lua scripts. Decision trees, finite state machines, and behavior trees are integrated into the sandbox.

Chapter 7, *Knowledge Representation*, adds the ability to store long-term and short-term information for individual agents. A centralized approach to storing and propagating agent knowledge about the world is exposed to Lua.

Chapter 8, *Perception*, exposes the services that are available to agents for them to query information about the world. Approaches toward visual- and communication-based information is integrated into the sandbox.

Chapter 9, *Tactics*, exposes a high-level spatial knowledge of the environment to the sandbox. Through a grid-based representation of the world, different knowledge sources are combined in order to give you an accurate tactical view of the environment for decision making.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision Making

In this chapter, we will cover the following topics:

• Creating reusable actions for agent behaviors
• Building conditional evaluators for decision making
• Creating a decisions tree structure that builds autonomous agents
• Creating a finite state machine that handles state-based agents
• Creating behavior trees for reactive agents

Now that we have agents that can animate and maneuver through their environments, we'll add high-level decision making to our agents. These data structures will finally give our agents autonomy in how they interact with the world as well as other agents.

Creating userdata

So far we've been using global data to store information about our agents. As we're going to create decision structures that require information about our agents, we'll create a local userdata table variable that contains our specific agent data as well as the agent controller in order to manage animation handling:

```
local userData = {
    alive, -- terminal flag
    agent, -- Sandbox agent
    ammo, -- current ammo
    controller, -- Agent animation controller
    enemy, -- current enemy, can be nil
    health, -- current health
    maxHealth -- max Health
};
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Moving forward, we will encapsulate more and more data as a means of isolating our systems from global variables. A `userData` table is perfect for storing any arbitrary piece of agent data that the agent doesn't already possess and provides a common storage area for data structures to manipulate agent data. So far, the listed data members are some common pieces of information we'll be storing; when we start creating individual behaviors, we'll access and modify this data.

### Agent actions

Ultimately, any decision logic or structure we create for our agents comes down to deciding what action our agent should perform. Actions themselves are isolated structures that will be constructed from three distinct states:

- Uninitialized
- Running
- Terminated

The typical lifespan of an action begins in uninitialized state and will then become initialized through a onetime initialization, and then, it is considered to be running. After an action completes the running phase, it moves to a terminated state where cleanup is performed. Once the cleanup of an action has been completed, actions are once again set to uninitialized until they wait to be reactivated.

We'll start defining an action by declaring the three different states in which actions can be as well as a type specifier, so our data structures will know that a specific Lua table should be treated as an action.

```
Action = {};  
Action.Status = {  
    RUNNING = "RUNNING",  
    TERMINATED = "TERMINATED",  
    UNINITIALIZED = "UNINITIALIZED"  
};  
Action.Type = "Action";
```

Remember, even though we use Lua in an object-oriented manner, Lua itself merely creates each instance of an object as a primitive table. It is up to the code we write to correctly interpret different tables as different objects. The use of a `Type` variable that is moving forward will be used to distinguish one class type from another.

---

For More Information:
Adding data members

To create an action, we'll pass three functions that the action will use for the initialization, updating, and cleanup. Additional information such as the name of the action and a `userData` variable, used for passing information to each callback function, is passed in during the construction time.

Moving our systems away from global data and into instanced object-oriented patterns requires each instance of an object to store its own data. As our Action class is generic, we use a custom data member, which is `userData`, to store action-specific information.

Whenever a callback function for the action is executed, the same `userData` table passed in during the construction time will be passed into each function. The update callback will receive an additional `deltaTimeInMillis` parameter in order to perform any time specific update logic.

To flush out the `Action` class' constructor function, we'll store each of the callback functions as well as initialize some common data members:

Action.lua:

```lua
function Action.new(name, initializeFunction, updateFunction, cleanUpFunction, userData)
    local action = {}
    -- The Action's data members.
    action.cleanUpFunction_ = cleanUpFunction;
    action.initializeFunction_ = initializeFunction;
    action.updateFunction_ = updateFunction;
    action.name_ = name or "";
    action.status_ = Action.Status.UNINITIALIZED;
    action.type_ = Action.Type;
    action.userData_ = userData;

    return action;
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision Making

Initializing an action

Initializing an action begins by calling the action's initialize callback and then immediately sets the action into a running state. This transitions the action into a standard update loop that is moving forward:

Action.lua:

```lua
function Action.Initialize(self)
    -- Run the initialize function if one is specified.
    if (self.status_ == Action.Status.UNINITIALIZED) then
        if (self.initializeFunction_) then
            self.initializeFunction_(self.userData_);
        end
    end
    -- Set the action to running after initializing.
    self.status_ = Action.Status.RUNNING;
end
```

Updating an action

Once an action has transitioned to a running state, it will receive callbacks to the update function every time the agent itself is updated, until the action decides to terminate. To avoid an infinite loop case, the update function must return a terminated status when a condition is met; otherwise, our agents will never be able to finish the running action.

An update function isn't a hard requirement for our actions, as actions terminate immediately by default if no callback function is present.

Action.lua:

```lua
function Action.Update(self, deltaTimeInMillis)
    if (self.status_ == Action.Status.TERMINATED) then
        -- Immediately return if the Action has already
        -- terminated.
        return Action.Status.TERMINATED;
    elseif (self.status_ == Action.Status.RUNNING) then
        if (self.updateFunction_) then
            -- Run the update function if one is specified.
            self.status_ = self.updateFunction_(deltaTimeInMillis, self.userData_);
            -- Ensure that a status was returned by the update
            -- function.
            assert(self.status_);
        else
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
-- If no update function is present move the action
-- into a terminated state.
self.status_ = Action.Status.TERMINATED;
end
end
return self.status_;
end

Action cleanup
Terminating an action is very similar to initializing an action, and it sets the status
of the action to uninitialized once the cleanup callback has an opportunity to finish
any processing of the action.

If a cleanup callback function isn't defined, the action will
immediately move to an uninitialized state upon cleanup.

During action cleanup, we'll check to make sure the action has fully terminated,
and then run a cleanup function if one is specified.

Action.lua:

function Action.CleanUp(self)
  if (self.status_ == Action.Status.TERMINATED) then
    if (self.cleanUpFunction_) then
      self.cleanUpFunction_(self.userData_);
    end
  end
  self.status_ = Action.Status.UNINITIALIZED;
end

Action member functions
Now that we've created the basic, initialize, update, and terminate functionalities,
we can update our action constructor with CleanUp, Initialize, and Update
member functions:

Action.lua:

function Action.new(name, initializeFunction, updateFunction,
cleanUpFunction, userData)
  ...
  -- The Action's accessor functions.
  action.CleanUp = Action.CleanUp;
  ...

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Creating actions

With a basic action class out of the way, we can start implementing specific action logic that our agents can use. Each action will consist of three callback functions—initialization, update, and cleanup—that we'll use when we instantiate our action instances.

The idle action

The first action we'll create is the basic and default choice from our agents that are going forward. The idle action wraps the IDLE animation request to our soldier's animation controller. As the animation controller will continue looping our IDLE command until a new command is queued, we'll time our idle action to run for 2 seconds, and then terminate it to allow another action to run:

SoldierActions.lua:

```lua
function SoldierActions_IdleInitialize(userData)
    userData.controller:QueueCommand(
        userData.agent,
        SoldierController.Commands.IDLE);

    -- Since idle is a looping animation, cut off the idle
    -- Action after 2 seconds.
    local sandboxTimeInMillis = Sandbox.GetTimeInMillis(
        userData.agent:GetSandbox());
    userData.idleEndTime = sandboxTimeInMillis + 2000;
end
```

Updating our action requires that we check how much time has passed; if the 2 seconds have gone by, we terminate the action by returning the terminated state; otherwise, we return that the action is still running:

SoldierActions.lua:

```lua
function SoldierActions_IdleUpdate(deltaTimeInMillis, userData)
    local sandboxTimeInMillis = Sandbox.GetTimeInMillis(
```
userData.agent:GetSandbox();
if (sandboxTimeInMillis >= userData.idleEndTime) then
    userData.idleEndTime = nil;
    return Action.Status.TERMINATED;
end
    return Action.Status.RUNNING;
end

As we'll be using our idle action numerous times, we'll create a wrapper around initializing our action based on our three functions:

**SoldierLogic.lua:**

```lua
local function IdleAction(userData)
    return Action.new("idle",
        SoldierActions_IdleInitialize,
        SoldierActions_IdleUpdate,
        SoldierActions_IdleCleanUp,
        userData);
end
```

**The die action**

Creating a basic death action is very similar to our idle action. In this case, as death in our animation controller is a terminating state, all we need to do is request that the DIE command be immediately executed. From this point, our die action is complete, and it's the responsibility of a higher-level system to stop any additional processing of logic behavior.

Typically, our agents will request this state when their health drops to zero. In the special case that our agent dies due to falling, the soldier's animation controller will manage the correct animation playback and set the soldier's health to zero:

**SoldierActions.lua:**

```lua
function SoldierActions_DieCleanUp(userData)
    -- No cleanup is required for death.
end

function SoldierActions_DieInitialize(userData)
    -- Issue a die command and immediately terminate.
    userData.controller:ImmediateCommand(userData.agent, SoldierController.Commands.DIE);
    return Action.Status.TERMINATED;
end
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
Creating a wrapper function to instantiate a death action is identical to our idle action:

```
local function DieAction(userData)
    return Action.new("die",
        SoldierActions_DieInitialize,
        SoldierActions_DieUpdate,
        SoldierActions_DieCleanUp,
        userData);
end
```

The reload action

Reloading is the first action that requires an animation to complete before we can consider the action complete, as the behavior will refill our agent's current ammunition count. As our animation controller is queue-based, the action itself never knows how many commands must be processed before the reload command has finished executing.

To account for this during the update loop of our action, we wait till the command queue is empty, as the reload action will be the last command that will be added to the queue. Once the queue is empty, we can terminate the action and allow the cleanup function to award the ammo:

```
function SoldierActions_ReloadCleanUp(userData)
    userData.ammo = userData.maxAmmo;
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
end

return Action.Status.TERMINATED;
end

SoldierLogic.lua:

local function ReloadAction(userData)
  return Action.new(
    "reload",
    SoldierActions_ReloadInitialize,
    SoldierActions_ReloadUpdate,
    SoldierActions_ReloadCleanUp,
    userData);
end

The shoot action

Shooting is the first action that directly interacts with another agent. In order to apply damage to another agent, we need to modify how the soldier's shots deal with impacts. Previously, when the soldier shot bullets out of his rifle, we added a callback function to handle the cleanup of particles; now, we'll add an additional functionality in order to decrement an agent's health if the particle impacts an agent:

Soldier.lua:

local function ParticleImpact(sandbox, collision)
  Sandbox.RemoveObject(sandbox, collision.objectA);

  local particleImpact = Core.CreateParticle(  
    sandbox, "BulletImpact");
  Core.SetPosition(particleImpact, collision.pointA);
  Core.SetParticleDirection(    
    particleImpact, collision.normalOnB);

  table.insert(    
    impactParticles,
    { particle = particleImpact, ttl = 2.0 } );

  if (Agent.IsAgent(collision.objectB)) then
    -- Deal 5 damage per shot.
    Agent.SetHealth(      
      collision.objectB,      
      Agent.GetHealth(collision.objectB) - 5);
  end
end
Creating the shooting action requires more than just queuing up a shoot command to the animation controller. As the SHOOT command loops, we'll queue an IDLE command immediately afterward so that the shoot action will terminate after a single bullet is fired. To have a chance at actually shooting an enemy agent, though, we first need to orient our agent to face toward its enemy. During the normal update loop of the action, we will forcefully set the agent to point in the enemy's direction.

```
function SoldierActions_ShootUpdate(deltaTimeInMillis, userData)
  -- Point toward the enemy so the Agent's rifle will shoot correctly.
  local forwardToEnemy = userData.enemy:GetPosition() – userData.agent:GetPosition();
  Agent.SetForward(userData.agent, forwardToEnemy);

  if (userData.controller:QueueLength() > 0) then
    return Action.Status.RUNNING;
  end

  -- Subtract a single bullet per shot.
  userData.ammo = userData.ammo - 1;
  return Action.Status.TERMINATED;
end
```

 FORCEFULLY SETTING THE AGENT'S FORWARD DIRECTION DURING AN ACTION WILL ALLOW OUR SOLDIER TO SHOOT BUT CREATES A VISUAL ARTIFACT WHERE THE AGENT WILL POP TO THE CORRECT FORWARD DIRECTION. SEE WHETHER YOU CAN MODIFY THE SHOOT ACTION'S UPDATE TO INTERPOLATE TO THE CORRECT FORWARD DIRECTION FOR BETTER VISUAL RESULTS.
SoldierLogic.lua:

```lua
local function ShootAction(userData)
    return Action.new(
        "shoot",
        SoldierActions_ShootInitialize,
        SoldierActions_ShootUpdate,
        SoldierActions_ShootCleanUp,
        userData);
end
```

The random move action

Randomly moving is an action that chooses a random point on the navmesh to be moved to. This action is very similar to other actions that move, except that this action doesn't perform the moving itself. Instead, the random move action only chooses a valid point to move to and requires the move action to perform the movement:

SoldierActions.lua:

```lua
function SoldierActions_RandomMoveCleanUp(userData)
end

function SoldierActions_RandomMoveInitialize(userData)
    local sandbox = userData.agent:GetSandbox();
    local endPoint = Sandbox.RandomPoint(sandbox, "default");
    local path = Sandbox.FindPath(
        sandbox,
        "default",
        userData.agent:GetPosition(),
        endPoint);
    while #path == 0 do
        endPoint = Sandbox.RandomPoint(sandbox, "default");
        path = Sandbox.FindPath(
            sandbox,
            "default",
            userData.agent:GetPosition(),
            endPoint);
    end
    userData.agent:SetPath(path);
    userData.agent:SetTarget(endPoint);
    userData.movePosition = endPoint;
    return Action.Status.TERMINATED;
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
function SoldierActions_RandomMoveUpdate(userData)  
  return Action.Status.TERMINATED;  
end

SoldierLogic.lua:

local function RandomMoveAction(userData)  
  return Action.new(  
    "randomMove",  
    SoldierActions_RandomMoveInitialize,  
    SoldierActions_RandomMoveUpdate,  
    SoldierActions_RandomMoveCleanUp,  
    userData);  
end

The move action

Our movement action is similar to an idle action, as the agent's walk animation will loop infinitely. In order for the agent to complete a move action, though, the agent must reach within a certain distance of its target position or timeout. In this case, we can use 1.5 meters, as that's close enough to the target position to terminate the move action and half a second to indicate how long the move action can run for:

SoldierActions.lua:

function SoldierActions_MoveToCleanUp(userData)  
  userData.moveEndTime = nil;  
end

function SoldierActions_MoveToInitialize(userData)  
  userData.controller:QueueCommand(  
    userData.agent,  
    SoldierController.Commands.MOVE);  
  -- Since movement is a looping animation, cut off the move  
  -- Action after 0.5 seconds.  
  local sandboxTimeInMillis =  
    Sandbox.GetTimeInMillis(userData.agent:GetSandbox());  
  userData.moveEndTime = sandboxTimeInMillis + 500;  
  return Action.Status.RUNNING;  
end

When applying the move action onto our agents, the indirect soldier controller will manage all animation playback and steer our agent along their path.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Chapter 6

The agent moving to a random position

Setting a time limit for the move action will still allow our agents to move to their final target position, but gives other actions a chance to execute in case the situation has changed. Movement paths can be long, and it is undesirable to not handle situations such as death until the move action has terminated:

**SoldierActions.lua:**

```lua
function SoldierActions_MoveToUpdate(deltaTimeInMillis, userData)

-- Terminate the action after the allotted 0.5 seconds. The decision structure will simply repath if the Agent needs to move again.
local sandboxTimeInMillis = Sandbox.GetTimeInMillis(userData.agent:GetSandbox());
if (sandboxTimeInMillis >= userData.moveEndTime) then
    userData.moveEndTime = nil;
    return Action.Status.TERMINATED;
end

path = userData.agent:GetPath();
if (#path ~= 0) then
    offset = Vector.new(0, 0.05, 0);
end
```

For More Information:

Decision Making

DebugUtilities_DrawPath(
    path, false, offset, DebugUtilities.Orange);
Core.DrawCircle(
    path[#path] + offset, 1.5, DebugUtilities.Orange);
end

-- Terminate movement is the Agent is close enough to the
-- target.
if (Vector.Distance(userData.agent:GetPosition(),
    userData.agent:GetTarget()) < 1.5) then
    Agent.RemovePath(userData.agent);
    return Action.Status.TERMINATED;
end
return Action.Status.RUNNING;
end

SoldierLogic.lua:

local function MoveAction(userData)
    return Action.new(
        "move",
        SoldierActions_MoveToInitialize,
        SoldierActions_MoveToUpdate,
        SoldierActions_MoveToCleanUp,
        userData);
end

The flee action

To create a flee action that causes our agent to run away from its enemy, we need to first find a valid path that is at least 4.0 meters away. Picking an arbitrary point on the navmesh will work, but we must ensure that there is a valid path to that point from the agent's current position:

SoldierActions.lua:

function SoldierActions_FleeCleanUp(userData)
    -- No cleanup is required for fleeing.
end

function SoldierActions_FleeInitialize(userData)
    local sandbox = userData.agent:GetSandbox();
    if (userData.enemy) then
        local endPoint = Sandbox.RandomPoint(sandbox, "default");
        if (userData.enemy) then
            local endPoint = Sandbox.RandomPoint(sandbox, "default");
        end
    end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
local path = Sandbox.FindPath(
    sandbox,
    "default",
    userData.agent:GetPosition(),
    endPoint);

-- Find a valid position at least 16 units away from the current enemy.
-- Note: Since pathfinding is not affected by the enemy, it is entirely possible to generate paths that move the Agent into the enemy, instead of away from the enemy.
while #path == 0 do
    endPoint = Sandbox.RandomPoint(sandbox, "default");
    while Vector.DistanceSquared(endPoint, userData.enemy:GetPosition()) < 16.0 do
        endPoint = Sandbox.RandomPoint(
            sandbox, "default");
    end
    path = Sandbox.FindPath(
        sandbox,
        "default",
        userData.agent:GetPosition(),
        endPoint);
end

Soldier_SetPath(userData.agent, path, false);
userData.agent:SetTarget(endPoint);
userData.movePosition = endPoint;
else
    -- Randomly move anywhere if the Agent has no current enemy.
    SoldierActions_RandomMoveInitialize(userData);
end

userData.controller:QueueCommand(
    userData.agent,
    SoldierController.Commands.MOVE);

return Action.Status.RUNNING;
end
Once a valid target position and path is found, the flee action acts very similar to a move action with one exception, which is health. As our flee action continues to move our agent until it reaches the target position, we must check whether the agent is still alive; otherwise, we have to terminate the action early:

SoldierActions.lua:

```lua
function SoldierActions_FleeUpdate(deltaTimeInMillis, userData)
    -- Terminate the Action if the agent is dead.
    if (Agent.GetHealth(userData.agent) <= 0) then
        return Action.Status.TERMINATED;
    end

    path = userData.agent:GetPath();
    DebugUtilities_DrawPath(
        path, false, Vector.new(), DebugUtilities.Blue);
    Core.DrawCircle(
        path[#path], 1.5, DebugUtilities.Blue);

    if (Vector.Distance( 
            userData.agent:GetPosition(), 
            userData.agent:GetTarget()) < 1.5) then 
        Agent.RemovePath(userData.agent);
        return Action.Status.TERMINATED;
    end

    return Action.Status.RUNNING;
end
```

When our agents start fleeing from their pursuers, they will now draw a blue path indicating the position they are fleeing to.
Agents fleeing from one another

Instantiating the `flee` action requires creating a new action and specifying the initialize, update, and cleanup functions.

```
local function FleeAction(userData)
    return Action.new(
        "flee",
        SoldierActions_FleeInitialize,
        SoldierActions_FleeUpdate,
        SoldierActions_FleeCleanUp,
        userData);
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision Making

The pursue action

Pursue is the exact opposite of flee, as our agent will track down its enemy instead of running from it. For this, we first find a path to the enemy and then begin moving toward it. Typically, our agents will default to this behavior if they have a known enemy and sufficient health to engage the enemy:

SoldierActions.lua:

```lua
function SoldierActions_PursueCleanUp(userData)
    -- No cleanup is required for pursuit.
end

function SoldierActions_PursueInitialize(userData)
    local sandbox = userData.agent:GetSandbox();
    local endPoint = userData.enemy:GetPosition();
    local path = Sandbox.FindPath(
        sandbox,
        "default",
        userData.agent:GetPosition(),
        endPoint);

    -- Path to the enemy if possible, otherwise idle and
    -- constantly try to repath to the enemy.
    if (#path ~= 0) then
        Soldier_SetPath(userData.agent, path, false);
        userData.agent:SetTarget(endPoint);
        userData.movePosition = endPoint;

        userData.controller:QueueCommand(
            userData.agent,
            SoldierController.Commands.MOVE);
    end

    return Action.Status.RUNNING;
end
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
As the enemy agent will typically be moving, we need to update our agent's path during the update loop so that the agent can track down the enemy's new position. If our agent gets within 3.0 meters of the enemy, the pursuit ends and another action can be run. As pursuit is a long running action, we also check for the health condition of our agent that can terminate pursuits early:

SoldierActions.lua:

```lua
function SoldierActions_PursueUpdate(deltaTimeInMillis, userData)
  -- Terminate the Action if the agent dies.
  if (Agent.GetHealth(userData.agent) <= 0) then
    return Action.Status.TERMINATED;
  end

  -- Constantly repath to the enemy's new position.
  local sandbox = userData.agent:GetSandbox();
  local endPoint = userData.enemy:GetPosition();
  local path = Sandbox.FindPath(
    sandbox,
    "default",
    userData.agent:GetPosition(),
    endPoint);
  if (#path ~= 0) then
    Soldier_SetPath(userData.agent, path, false);
    userData.agent:SetTarget(endPoint);
    userData.movePosition = endPoint;

    offset = Vector.new(0, 0.1, 0);
    path = userData.agent:GetPath();
    DebugUtilities_DrawPath(
      path, false, offset, DebugUtilities.Red);
    Core.DrawCircle(
      path[#path] + offset, 3, DebugUtilities.Red);
  end

  -- Terminate the pursuit Action when the Agent is within
  -- shooting distance to the enemy.
  if (Vector.Distance(userData.agent:GetPosition(),
    userData.agent:GetTarget()) < 3) then
    Agent.RemovePath(userData.agent);
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
return Action.Status.TERMINATED;
end

return Action.Status.RUNNING;
end

When our soldiers are pursuing their enemy, we'll now see a red path that is constantly updated to move toward the enemy position.

Agents pursuing one another

Instantiating a pursue action is identical to the previous actions we've created and requires passing our pursue initialize, update, and cleanup functions to each new action instance.

SoldierLogic.lua:

local function PursueAction(userData)
    return Action.new("pursue",
        SoldierActions_PursueInitialize,
        SoldierActions_PursueUpdate,
        SoldierActions_PursueCleanUp,
        userData);
end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Evaluators

Evaluators are the principal method of handling conditional checks in our decision structures. While actions perform the eventual behaviors that our agents exhibit, it's the responsibility of evaluators to determine which action is allowed to run at what time.

Creating an evaluator object simply wraps a function call that returns true or false when the `userData` table is passed into the function:

Evaluator.lua:

```lua
function Evaluator.Evaluate(self)
    return self.function_(self.userData_);
end

function Evaluator.new(name, evalFunction, userData)
    local evaluator = {}

        -- data members
        evaluator.function_ = evalFunction;
        evaluator.name_   = name or "";
        evaluator.type_   = Evaluator.Type;
        evaluator.userData_ = userData;

        -- object functions
        evaluator.evaluate_ = Evaluate;

    return evaluator;
end
```

Creating evaluators

Creating evaluators relies on simple functions that perform isolated operations on the agent's `userData` table. Typically, most evaluators will only perform calculations based on `userData` instead of modifying the data itself, although there is no limitation on doing this. As the same evaluator might appear within a decision structure, care must be taken to create consistent decision choices.

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
**Decision Making**

**Constant evaluators**

First, we'll need to create the two most basic evaluators; one that always returns true, and another that always returns false. These evaluators come in handy as a means of enabling or disabling actions during development:

**SoldierEvaluators.lua**:

```lua
function SoldierEvaluators_True(userData)
    return true;
end

function SoldierEvaluators_False(userData)
    return false;
end
```

**Has ammo evaluator**

Next, we can perform a simple ammo check to see whether the agent has any remaining ammo, as well as the inverse to check whether the agent has no ammo:

**SoldierEvaluators.lua**:

```lua
function SoldierEvaluators_HasAmmo(userData)
    return userData.ammo ~= nil and userData.ammo > 0;
end

function SoldierEvaluators_HasNoAmmo(userData)
    return not SoldierEvaluators_HasAmmo(userData);
end
```

**Has critical health evaluator**

A critical health check evaluator returns true if our agent has less than 20 percent of its health; otherwise, it evaluates to false:

**SoldierEvaluators.lua**:

```lua
function SoldierEvaluators_HasCriticalHealth(userData)
    return Agent.GetHealth(userData.agent) < (userData.maxHealth * 0.2);
end
```
Has enemy evaluator

Has enemy is the first evaluator that actually modifies the userData table passed in to the evaluator. The HasEnemy function calculates the best enemy the agent should consider to be its enemy. Iterating over all agents in the sandbox, the closest pathable enemy will be selected. If no enemy that meets these requirements is found, the HasEnemy evaluator returns false.

As HasEnemy modifies and performs non-simple calculations, it should be used sparingly within any logic structure.

Since our agents will need to know when they have an enemy as well as when there are no valid enemies, we'll create a normal HasEnemy function evaluator and the inverse HasNoEnemy function.

SoldierEvaluators.lua:

```lua
function SoldierEvaluators_HasEnemy(userData)
    local sandbox = userData.agent:GetSandbox();
    local position = Agent.GetPosition(userData.agent);
    local agents = Sandbox.GetAgents(userData.agent:GetSandbox());

    local closestEnemy;
    local distanceToEnemy;

    for index=1, #agents do
        local agent = agents[index];
        if (Agent.GetId(agent) ~= Agent.GetId(userData.agent) and
            Agent.GetHealth(agent) > 0) then
            -- Find the closest enemy.
            local distanceToAgent = Vector.DistanceSquared(
                position, Agent.GetPosition(agent));
            if (closestEnemy == nil or
                distanceToAgent < distanceToEnemy) then
                local path = Sandbox.FindPath(
                    sandbox,
                    "default",
                    position,
                    agent:GetPosition());
                -- If the agent can path to the enemy, use this
                -- enemy as the best possible enemy.
                if (#path ~= 0) then
                    closestEnemy = agent;
                    distanceToEnemy = distanceToAgent;
            end
        end
    end
end
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision Making

end
end end

userData.enemy = closestEnemy;
return userData.enemy ~= nil;
end

function SoldierEvaluators_HasNoEnemy(userData)
return not SoldierEvaluators_HasEnemy(userData);
end

Has move position evaluator

The HasMovePosition evaluator calculates whether the agent is within 1.5 meters of its target position. This allows for agents to terminate their move behaviors once they’ve reached their target position:

SoldierEvaluators.lua:

function SoldierEvaluators_HasMovePosition(userData)
return userData.movePosition ~= nil and (Vector.Distance(userData.agent:GetPosition(), userData.movePosition) > 1.5);
end

Is alive evaluator

IsAlive simply informs you whether the agent has health left, and the IsNotAlive evaluator returns the negation:

SoldierEvaluators.lua:

function SoldierEvaluators_IsAlive(userData)
return Agent.GetHealth(userData.agent) > 0;
end

function SoldierEvaluators_IsNotAlive(userData)
return not SoldierEvaluators_IsAlive(userData);
end
Can shoot enemy evaluator

To determine if our agent can shoot an enemy agent, we can perform a distance check to see whether our agent is within 3.0 meters of the enemy. While this might produce some false positives, it's effective enough to allow our agents to shoot one another with a relatively high accuracy:

SoldierEvaluators.lua:

```lua
function SoldierEvaluators_CanShootAgent(userData)
    if (userData.enemy ~= nil and
        Agent.GetHealth(userData.enemy) > 0 and
        Vector.Distance(
            userData.agent:GetPosition(),
            userData.enemy:GetPosition()) < 3) then
        return true;
    end;
    return false;
end
```

50/50 chance evaluator

A 50/50 chance evaluator is a random roll and returns true half of the time:

SoldierEvaluators.lua:

```lua
function SoldierEvaluators_Random(userData)
    return math.random() >= 0.5;
end
```

Decision structures

With actions and evaluators at our disposal, we'll begin to create different types of logic structures that use both of these primitive operators to build our agent's behaviors. While each decision structure uses different approaches and techniques, we'll create similar behaving agents based on the actions and evaluators we have.

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision trees

Decision trees will be the first structure we'll implement and are, by far, the easiest way to understand how a decision was made. A decision tree is composed of branches and leaves. Each branch in the tree will wrap an evaluator, while each leaf will be composed of an action. Through a sequence of branch conditions, our decision tree will always result in a final action that our agent will perform.

To create a decision tree structure, we'll implement an update loop for our tree, which evaluates the root branch within the tree and then proceeds to process the resulting action. Once the action has been initialized, updated, and eventually, terminated, the decision tree will re-evaluate the tree from the root branch to determine the next action to be executed:

DecisionTree.lua:

```lua
require "Action"

DecisionTree = {};

function DecisionTree.SetBranch(self, branch)
    self.branch_ = branch;
end

function DecisionTree.Update(self, deltaTimeInMillis)
    -- Skip execution if the tree hasn't been setup yet.
    if (self.branch_ == nil) then
        return;
    end

    -- Search the tree for an Action to run if not currently executing an Action.
    if (self.currentAction_ == nil) then
        self.currentAction_ = self.branch_:Evaluate();
        self.currentAction_:Initialize();
    end

    local status = self.currentAction_:Update(deltaTimeInMillis);

    -- Clean up the Action once it has terminated.
    if (status == Action.Status.TERMINATED) then
        self.currentAction_:CleanUp();
        self.currentAction_ = nil;
    end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
function DecisionTree.new()
    local decisionTree = {};

    -- The DecisionTree's data members.
    decisionTree.branch_ = nil;
    decisionTree.currentAction_ = nil;

    -- The DecisionTree's accessor functions.
    decisionTree.SetBranch = DecisionTree.SetBranch;
    decisionTree.Update = DecisionTree.Update;

    return decisionTree;
end

Branches

Branches in a decision tree consist of a conditional evaluator that determines which child is executed. It is the responsibility of the evaluator to return a value that ranges from 1 to the maximum number of children in the branch. Even though we’ll only be creating binary decision trees, the structure itself can branch out to any number of children, as shown in the following diagram:

![An example of a decision tree branch](image)

Our decision branch class will have basic assessors such as adding additional children as well as setting the evaluator function used during branch calculation.

DecisionBranch.lua:

    DecisionBranch = {}
    DecisionBranch.Type = "DecisionBranch";
    function DecisionBranch.new()
        local branch = {};
        -- The DecisionBranch's data members.
        branch.children_ = {};

For More Information:
branch.evaluator_ = nil;
branch.type_ = DecisionBranch.Type;
-- The DecisionBranch's accessor functions.
branch.AddChild = DecisionBranch.AddChild;
branch.Evaluate = DecisionBranch.Evaluate;
branch.SetEvaluator = DecisionBranch.SetEvaluator;
return branch;
end

function DecisionBranch.AddChild(self, child, index)
  -- Add the child at the specified index, or as the last child.
  index = index or (#self.children_ + 1);
  table.insert(self.children_, index, child);
end

function DecisionBranch.SetEvaluator(self, evaluator)
  self.evaluator_ = evaluator;
end

Decision leaves
As the leaves of the decision tree are merely actions, we can completely encase each leaf action into the branches themselves without the need for any additional structures. The use of the type variable allows us to determine whether a child of the branch is another branch or an action to be executed.

Branch evaluation
To evaluate a branch, we execute the evaluator and use the return value to further process the tree. Once a choice is made, we either return an action node if the selected child is a leaf, otherwise we recursively evaluate another branch until an action is found.

Every branch within a decision tree must eventually end with an action node; trees without actions as leafs are malformed and will not evaluate properly.

To implement evaluation, we'll use the type field to determine if a child should be considered as a branch or as an action to return.

DecisionBranch.lua:

function DecisionBranch.Evaluate(self)
  -- Execute the branch's evaluator function, this much return a numeric value which indicates what child should execute.
end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
local eval = self.evaluator_();
local choice = self.children_[eval];

if (choice.type_ == DecisionBranch.Type) then
    -- Recursively evaluate children that are decisions
    -- branches.
    return choice:Evaluate();
else
    -- Return the leaf action.
    return choice;
end
end

Building a decision tree

Building a decision tree starts with instantiating an instance of a decision tree, creating each branch within our tree, connecting the conditional branches, and adding actions:

SoldierLogic.lua:

        function SoldierLogic_DecisionTree(userData)
            local tree = DecisionTree.new();
            return tree;
        end

Creating branches

The tree we’ll be creating combines each of the actions and evaluators we implemented previously and gives our agents the ability to pursue, flee, move, shoot, idle, reload, and die.

Critical health decision branch

First we’ll create each branch instance that our decision tree will contain before adding any evaluators or actions.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Once we've created each branch in our decision tree, we'll begin to hook up the parent-child relationships between branches as well as add leaf node actions.

```lua
function SoldierLogic_DecisionTree(userData)
    local tree = DecisionTree.new();
    local isAliveBranch = DecisionBranch.new();
    local criticalBranch = DecisionBranch.new();
    local moveFleeBranch = DecisionBranch.new();
    local enemyBranch = DecisionBranch.new();
    local ammoBranch = DecisionBranch.new();
    local shootBranch = DecisionBranch.new();
    local moveRandomBranch = DecisionBranch.new();
    local randomBranch = DecisionBranch.new();

    tree:SetBranch(isAliveBranch);
    return tree;
end
```

As our decision tree follows a binary tree design, each branch will typically have one action and another branch. Branches at the tips of the tree will end with two different actions:

```lua
function SoldierLogic_DecisionTree(userData)
    ...

    isAliveBranch:AddChild(criticalBranch);
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
isAliveBranch:AddChild(DieAction(userData));
isAliveBranch:SetEvaluator(
    function()
        if SoldierEvaluators_IsNotAlive(userData) then
            return 2;
        end
        return 1;
    end);

criticalBranch:AddChild(moveFleeBranch);
criticalBranch:AddChild(enemyBranch);
criticalBranch:SetEvaluator(
    function()
        if SoldierEvaluators_HasCriticalHealth(userData) then
            return 1;
        end
        return 2;
    end);

moveFleeBranch:AddChild(MoveAction(userData));
moveFleeBranch:AddChild(FleeAction(userData));
moveFleeBranch:SetEvaluator(
    function()
        if SoldierEvaluators_HasMovePosition(userData) then
            return 1;
        end
        return 2;
    end);

tree:SetBranch(isAliveBranch);
return tree;
end

So far, we've added death, move, and flee actions; now, we'll add the remaining reload, shoot, pursue, move, random move, and idle actions:

SoldierLogic.lua:

    function SoldierLogic_DecisionTree(userData)
        ...
        energyBranch:AddChild(ammoBranch);
        energyBranch:AddChild(moveRandomBranch);
enemyBranch:SetEvaluator(
    function()
        if SoldierEvaluators_HasAmmo(userData) then
            return 2;
        end
        return 1;
    end);

ammoBranch:AddChild(shootBranch);
ammoBranch:AddChild(ReloadAction(userData));
ammoBranch:SetEvaluator(
    function()
        if SoldierEvaluators_HasAmmo(userData) then
            return 1;
        end
        return 2;
    end);

shootBranch:AddChild(ShootAction(userData));
shootBranch:AddChild(PursueAction(userData));
shootBranch:SetEvaluator(
    function()
        if SoldierEvaluators_CanShootAgent(userData) then
            return 1;
        end
        return 2;
    end);

moveRandomBranch:AddChild(MoveAction(userData));
moveRandomBranch:AddChild(randomBranch);
moveRandomBranch:SetEvaluator(
    function()
        if SoldierEvaluators_HasMovePosition(userData) then
            return 1;
        end
        return 2;
    end);

randomBranch:AddChild(RandomMoveAction(userData));
randomBranch:AddChild(IdleAction(userData));
randomBranch:SetEvaluator(
    function()
        if SoldierEvaluators_Random(userData) then
            return 1;
        end
        return 2;
    end);
Creating a decision tree agent

To create an agent whose logic is controlled by our decision tree, we'll modify our indirect soldier agent, as the initial setup of an animation state machine and soldier controller is already done for us.

We'll first create the `userData` table and associate the initial values so that our decision tree can interact with the agent. Once we've populated the `userData` table, we can instantiate our decision tree. We'll change the agent's update loop to process the decision tree as well as the soldier controller. As our decision tree expects that the execution will end when an agent dies, we'll add a conditional check that halts updates when this occurs:

**IndirectSoldierAgent.lua:**

```lua
local soldier;
local soldierController;
local soldierDecisionTree;
local soldierUserData;

function Agent_Initialize(agent)
    Soldier_InitializeAgent(agent);
    soldier = Soldier_CreateSoldier(agent);
    weapon = Soldier_CreateWeapon(agent);
    soldierController = SoldierController.new(
        agent, soldier, weapon);
    Soldier_AttachWeapon(soldier, weapon);
    weapon = nil;

    soldierUserData = {};
    soldierUserData.agent = agent;
    soldierUserData.controller = soldierController;
    soldierUserData.maxHealth = soldierUserData.health;
    soldierUserData.alive = true;
    soldierUserData.ammo = 10;
```
soldierUserData.maxAmmo = 10;

soldierDecisionTree = SoldierLogic_DecisionTree(soldierUserData);
end

function Agent_Update(agent, deltaTimeInMillis)
    if (soldierUserData.alive) then
        soldierDecisionTree:Update(deltaTimeInMillis);
    end

    soldierController:Update(agent, deltaTimeInMillis);
end

**Strengths of decision trees**

After implementing a decision tree, it's relatively easy to tell how decisions are made, as evaluators essentially serve as nested if...else structures. Any possible case can be modeled by the decision tree, as every action our agent supports will be a leaf node within the tree.

**Pitfalls of decision trees**

Although decision trees are easy to understand, their weaknesses stem from trying to implement complicated logical conditions where every possible outcome must also be accounted for. With a large number of branch possibilities, a decision tree will also need to be balanced; otherwise, parts of the tree will end up needing to be replicated, further increasing the complexity of the tree.

**Finite state machines**

Creating a finite state machine (FSM) for modeling logic will resemble the animation state machines we've created previously, except that transitioning to a new state within the state machine is handled automatically through the evaluation of transitions. Once one evaluator returns true, the state machine will transition to the new state and invoke the associated state's action.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
States
States within an FSM are responsible for associating an action with the state. We create a state by passing an action and naming the state for debug convenience:

FiniteState.lua:

```lua
require "Action";
require "FiniteState";
require "FiniteStateTransition";

FiniteState = {};

function FiniteState.new(name, action)
    local state = {};

    -- The FiniteState's data members.
    state.name_ = name;
    state.action_ = action;

    return state;
end
```

Transitions
Transitions encapsulate the state to be transitioned to as well as the evaluator that determines whether the transition should be taken. The responsibility for evaluating transitions is left to the finite state machine itself:

FiniteStateTransition.lua:

```lua
FiniteStateTransition = {};

function FiniteStateTransition.new(toStateName, evaluator)
    local transition = {};

    -- The FiniteStateTransition's data members.
    transition.evaluator_ = evaluator;
    transition.toStateName_ = toStateName;

    return transition;
end
```
Finite state machine structure

The FSM consists of ever logical state the agent can execute on, as well as a two-dimensional table of every possible state transition. The transition_ variable is keyed by the state name and consists of transition objects:

FiniteStateMachine.lua:

```
require "Action";
require "FiniteState";
require "FiniteStateTransition";

function FiniteStateMachine.new(userData)
    local fsm = {};
    -- The FiniteStateMachine's data members.
    fsm.currentState_ = nil;
    fsm.states_ = {};
    fsm.transitions_ = {};
    fsm.userData_ = userData;
    end
```

Helper functions

We can implement additional helper functions in order to allow interactions with the FSM in a safe manner:

FiniteStateMachine.lua:

```
function FiniteStateMachine.ContainsState(self, stateName)
    return self.states_[stateName] ~= nil;
end

function FiniteStateMachine.ContainsTransition(
    self, fromStateName, toStateName)
    return self.transitions_[fromStateName] ~= nil and
    self.transitions_[fromStateName][toStateName] ~= nil;
end

function FiniteStateMachine.GetCurrentStateName(self)
    if (self.currentState_) then
        return self.currentState_.name_;
    end
end

function FiniteStateMachine.GetCurrentStateStatus(self)
    if (self.currentState_) then
        return self.currentState_.action_.status_;
    end
end
```
function FiniteStateMachine.SetState(self, stateName)
    if (self:ContainsState(stateName)) then
        if (self.currentState_) then
            self.currentState_.action_:CleanUp();
        end
        self.currentState_ = self.states_[stateName];
        self.currentState_.action_:Initialize();
    end
end

Adding states and transitions
As states are contained in a table where the state's name is the look-up key, we
simply create a new FiniteState instance and add it to the table. Adding transitions
requires checking whether each of the transitions to and from states exist within the
FSM, and then inserting the transition within the transitions_class variable.

Going forward, as we create states and transitions, we'll represent each state as well
as every possible transition using the following diagram:

Creating our AddState and AddTransition functions will modify the internal tables
held by the FSM in order to add additional states and map one state to another
using transitions

FiniteStateMachine.lua:

    function FiniteStateMachine.AddState(self, name, action)
        self.states_[name] = FiniteState.new(name, action);
    end

    function FiniteStateMachine.AddTransition(
        self, fromStateName, toStateName, evaluator)
        -- Ensure both states exist within the FSM.
        if (self:ContainsState(fromStateName) and

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
self:ContainsState(toStateName)) then
    if (self.transitions_[fromStateName] == nil) then
        self.transitions_[fromStateName] = {};
    end
    -- Add the new transition to the "from" state.
    table.insert(
        self.transitions_[fromStateName],
        FiniteStateTransition.new(toStateName, evaluator));
end

-- Add the new transition to the "from" state.

---

**Updating the finite state machine**

Updating the state machine internally performs two different operations. The first is to execute or continue executing a running action, and the second is to select a new state once the action is complete. As transitions exist in a priority order, we iterate over all transitions executing their evaluators to determine the next state for the FSM to transition to. The first evaluator that returns true determines the state the FSM moves to.

Note that there is no validation that the finite state machine actually picks an action to transition to. If this occurs, the FSM will attempt to iterate over each transition in the next update call in order to find a valid transition. Using an evaluator that always returns true as the last possible transition is a best practice to prevent cases where our agent is unable to select any action to perform.

First we'll create an EvaluateTransitions function to determine the next state our FSM will move to once a state has finished. Afterwards we can create the FSM Update function that manages a running action and determines when state transitions occur.

**FiniteStateMachine.lua**:

```
local function EvaluateTransitions(self, transitions)
    for index = 1, #transitions do
        -- Find the first transition that evaluates to true,
        -- return the state the transition points to.
        if (transitions[index].evaluator_(self.userData_)) then
            return transitions[index].toStateName_;
        end
    end
end

function FiniteStateMachine.Update(self, deltaTimeInMillis)
    if (self.currentState_) then
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
local status = self:GetCurrentStateStatus();

if (status == Action.Status.RUNNING) then
    self.currentState_.action_:Update(deltaTimeInMillis);
elseif (status == Action.Status.TERMINATED) then
    local toStateName = EvaluateTransitions(
        self,
        self.transitions_[self.currentState_.name_]);
    if (self.states_[toStateName] ~= nil) then
        self.currentState_.action_:CleanUp();
        self.currentState_ = self.states_[toStateName];
        self.currentState_.action_:Initialize();
    end
end
end

Adding instance functions

With helper functions and the main update loop out of the way, we can modify our FSM mapping to point to our new local functions:

FiniteStateMachine.lua:

    function FiniteStateMachine.new()
        local fsm = {

        ...

    -- The FiniteStateMachine's accessor functions.
        fsm.AddState = FiniteStateMachine.AddState;
        fsm.AddTransition = FiniteStateMachine.AddTransition;
        fsm.ContainsState = FiniteStateMachine.ContainsState;
        fsm.ContainsTransition = FiniteStateMachine.ContainsTransition;
        fsm.GetCurrentStateName = FiniteStateMachine.GetCurrentStateName;
        fsm.SetState = FiniteStateMachine.SetState;
        fsm.Update = FiniteStateMachine.Update;

        return fsm;
    end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Building a finite state machine

To build a finite state machine for our agent, we'll create the initial states that wrap each possible action. As the state machine needs a starting point, we'll set the state explicitly to the idle state to begin with. Once the idle state has finished, our state machine will automatically pick the most relevant action to execute afterward:

**SoldierLogic.lua:**

```lua
function SoldierLogic_FiniteStateMachine(userData)
    local fsm = FiniteStateMachine.new(userData);
    fsm:AddState("die", DieAction(userData));
    fsm:AddState("flee", FleeAction(userData));
    fsm:AddState("idle", IdleAction(userData));
    fsm:AddState("move", MoveAction(userData));
    fsm:AddState("pursue", PursueAction(userData));
    fsm:AddState("randomMove", RandomMoveAction(userData));
    fsm:AddState("reload", ReloadAction(userData));

    fsm:SetState("idle");
    return fsm;
end
```

The idle state

Creating the idle state consists of adding every possible transition from the idle, which also allows for looping back to itself.

![The soldier's FSM idle state](image)

**For More Information:**

As transitions are evaluated in a priority order, it is imperative to add the most important actions first, followed by less important actions. In this case, death will always be the most important action, followed by our agent's sense of preservation to flee from their enemies. If both of these cases don't exist, our agent will move into combat behaviors such as reload, shoot, pursue, or randomly wander around:

**SoldierLogic.lua:**

```lua
function SoldierLogic_FiniteStateMachine(userData)

  ...

  -- idle action
  fsm:AddTransition(
    "idle", "die", SoldierEvaluators_IsNotAlive);
  fsm:AddTransition(
    "idle", "flee", SoldierEvaluators_HasCriticalHealth);
  fsm:AddTransition(
    "idle", "reload", SoldierEvaluators_HasNoAmmo);
  fsm:AddTransition(
    "idle", "shoot", SoldierEvaluators_CanShootAgent);
  fsm:AddTransition(
    "idle", "pursue", SoldierEvaluators_HasEnemy);
  fsm:AddTransition(
    "idle", "randomMove", SoldierEvaluators_Random);
  fsm:AddTransition("idle", "idle", SoldierEvaluators_True);

  fsm:SetState("idle");
  return fsm;
end
```

For More Information:

The movement state

The move state is nearly identical to the idle state, as every possibility is equally valid except for the addition of the looping movement.

As our move action is time-based, instead of terminating when our agent reaches its target position, we need to continue looping within the state until a better option becomes available:

SoldierLogic.lua:

```lua
function SoldierLogic_FiniteStateMachine(userData)
...

   -- move action
   fsm:AddTransition(
      "move", "die", SoldierEvaluators_IsNotAlive);
   fsm:AddTransition(
      "move", "flee", SoldierEvaluators_HasCriticalHealth);
   fsm:AddTransition(
      "move", "reload", SoldierEvaluators_HasNoAmmo);
   fsm:AddTransition(
      "move", "shoot", SoldierEvaluators_CanShootAgent);
   fsm:AddTransition(
      "move", "pursue", SoldierEvaluators_HasEnemy);
   fsm:AddTransition(
      "move", "move", SoldierEvaluators_HasMovePosition);
   fsm:AddTransition(
      "move", "randomMove", SoldierEvaluators_Random);
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
The random movement state

A random movement state is only responsible for picking a location to move to; once a location is found, our agent will proceed to the move position state to complete the action.

Separating position selection and processing of a position allows us to minimize all possible transitions that need to be taken care of from the random movement state:

SoldierLogic.lua:

```lua
function SoldierLogic_FiniteStateMachine(userData)
    ...

    -- random move action
    fsm:AddTransition(
        "randomMove", "die", SoldierEvaluators_IsNotAlive);
    fsm:AddTransition(
        "randomMove", "move", SoldierEvaluators_True);

    fsm:SetState("idle");
    return fsm;
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
**The shoot state**

Shooting is another state that is nearly identical to our idle state.

As our state machine only has eight possible states, states such as idle and shoot are nearly fully connected to any possible action our agent can perform. Having fully connected states produces more reactive agents overall:

**SoldierLogic.lua:**

```lua
function SoldierLogic_FiniteStateMachine(userData)
    ...

    -- shoot action
    fsm:AddTransition("shoot", "die", SoldierEvaluators_IsNotAlive);
    fsm:AddTransition("shoot", "flee", SoldierEvaluators_HasCriticalHealth);
    fsm:AddTransition("shoot", "reload", SoldierEvaluators_HasNoAmmo);
    fsm:AddTransition("shoot", "shoot", SoldierEvaluators_CanShootAgent);
    fsm:AddTransition("shoot", "pursue", SoldierEvaluators_HasEnemy);
    fsm:AddTransition("shoot", "randomMove", SoldierEvaluators_Random);

    fsm:SetState("idle");
    return fsm;
end
```

---

For More Information:

The flee state

As fleeing is a state our agents cannot exit from, we can loop directly into another flee action until our agents die.

![The soldier's FSM flee state](image)

The only way for our agent to exit from fleeing its enemy is through death:

`SoldierLogic.lua`:

```lua
function SoldierLogic_FiniteStateMachine(userData)
...

    -- flee action
    fsm:AddTransition(
        "flee", "die", SoldierEvaluators_IsNotAlive);
    fsm:AddTransition("flee", "move", SoldierEvaluators_True);

    fsm:SetState("idle");
    return fsm;
end
```

The die state

The death state is the only terminal state within the state machine and has no transitions. The processing of the state machine is expected to seize once the death state finishes.

![The soldier's FSM die state](image)
The pursue state
As pursuing an enemy generates new paths during each update call, our agent doesn't need another state to handle movements.

![The soldier's FSM pursue state](image)

In this case, the only thing our agent cares about while pursuing an enemy is to flee in order to prevent death, shoot the enemy when it comes within range, or idle if the enemy is no longer valid. As there are limited transitions from pursuit, our agent will sometimes need to jump through another state to find a valid action, such as reload. Even though cases like these exist, very little noticeable latency is introduced, and the finite state machine is less complex because of the reduced number of transitions:

SoldierLogic.lua:

```
function SoldierLogic_FiniteStateMachine(userData)

    ...

    -- pursue action
    fsm:AddTransition(
        "pursue", "die", SoldierEvaluators_IsNotAlive);
    fsm:AddTransition(
        "pursue", "shoot", SoldierEvaluators_CanShootAgent);
    fsm:AddTransition(
        "pursue", "move", SoldierEvaluators_HasMovePosition);
    fsm:AddTransition("pursue", "idle", SoldierEvaluators_True);

    fsm:SetState("idle");
    return fsm;
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
The reload state

As reloading is such a short-lived action, every transition out of reload is either a combat action or an idling action.

![The soldier's FSM reload state](image)

This reduced number of transitions will also incur latency in behavior selection, but it further reduces the complexity of our agent:

**SoldierLogic.lua**:

```lua
function SoldierLogic_FiniteStateMachine(userData)
    ...
    -- reload action
    fsm:AddTransition("reload", "die", SoldierEvaluators_IsNotAlive);
    fsm:AddTransition("reload", "shoot", SoldierEvaluators_CanShootAgent);
    fsm:AddTransition("reload", "pursue", SoldierEvaluators_HasEnemy);
    fsm:AddTransition("reload", "randomMove", SoldierEvaluators_Random);
    fsm:AddTransition("reload", "idle", SoldierEvaluators_True);

    fsm:SetState("idle");
    return fsm;
end
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Creating a finite state machine agent

Now that we've created a finite state machine, we can simply replace the decision tree that powers our indirect soldier agent with a finite state machine. The initialization, updating, and termination of our FSM are identical to the decision tree.

With an abstract data structure, we can now create both decision-tree-based agents and finite state machine agents simultaneously:

IndirectSoldierAgent.lua:

```lua
local soldier;
local soldierController;
local soldierFSM;
local soldierUserData;

function Agent_Initialize(agent)
...
    soldierFSM = SoldierLogic_FiniteStateMachine(  
        soldierUserData);
end

function Agent_Update(agent, deltaTimeInMillis)
    if (soldierUserData.alive) then
        soldierFSM:Update(deltaTimeInMillis);
    end

    soldierController:Update(agent, deltaTimeInMillis);
end
```

Strengths of finite state machines

The straightforwardness of a state machine relies heavily on large amounts of data. One of the key strengths with such a structure lies in the fact that the statefulness of an agent is inherent within the logical structure. Compared to a decision tree, finite states isolate the amount of possible actions that can follow from another action. To create the same sort of flow in a decision tree would be inherently difficult and would require embedding some sort of userdata that maintains the statefulness we get for free with a finite state machine.
Pitfalls of finite state machines
The downside of state machines, though, are the exponential possible connections that come with the addition of each new state. To reduce latency or unexpected chains of actions, state machines need to be well connected so that agents can quickly select the best action.

Our implementation of state transitions is a simple, priority-based approach that might not fit all circumstances, in which case, a weighted or search-based approach might become necessary. This additional complexity can further complicate logic selection, where actions are chosen for reasons that aren't apparent.

Behavior trees
With decision trees focusing on the if...else style of action selection and state machines focusing on the statefulness of actions, behavior trees fill a nice middle ground with reaction-based decision making.

The behavior tree node
Behavior trees are composed solely of different types of nodes. Based on the node type, the behavior tree will interpret child nodes as actions and evaluators. As we'll need to distinguish each node instance type from one another, we can create an enumeration of all supported node types: actions, conditions, selectors, and sequences.

Creating an instance of a behavior tree node merely sets the node type and the name of the node:

BehaviorTreeNode.lua:

```lua
BehaviorTreeNode = {};

BehaviorTreeNode.Type = {
  ACTION = "ACTION",
  CONDITION = "CONDITION",
  SELECTOR = "SELECTOR",
  SEQUENCE = "SEQUENCE"
};

function BehaviorTreeNode.new(name, type)
  local node = {};

  -- The BehaviorTreeNode's data members.
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
node.action_ = nil;
node.children_ = {};
node.evaluator_ = nil;
node.name_ = name or "";
node.parent_ = nil;
node.type_ = type or BehaviorTreeNode.Type.ACTION;
end

Helper functions
Adding some object-oriented helper functions to handle child management as well as a backward link to the nodes parent will allow the behavior tree to evaluate the nodes more efficiently:

BehaviorTreeNode.lua:

function BehaviorTreeNode.AddChild(self, child, index)
    index = index or (#self.children_ + 1);

    table.insert(self.children_, index, child);

    child.parent_ = self;
end

function BehaviorTreeNode.ChildIndex(self, child)
    for index=1, #self.children_ do
        if (self.children_[index] == child) then
            return index;
        end
    end

    return -1;
end

function BehaviorTreeNode.GetChild(self, childIndex)
    return self.children_[childIndex];
end

function BehaviorTreeNode.GetNumberOfChildren(self)
    return #self.children_;
end

function BehaviorTreeNode.GetParent(self)
    return self.parent_;
function BehaviorTreeNode.SetAction(self, action)
    self.action_ = action;
end

function BehaviorTreeNode.SetEvaluator(self, evaluator)
    self.evaulator_ = evaluator;
end

function BehaviorTreeNode.SetType(self, type)
    self.type_ = type;
end

Updating the behavior tree node

With local functions fleshed out, we can fully encapsulate our behavior tree node functionality:

BehaviorTreeNode.lua:

    function BehaviorTreeNode.new(name, type)
        local node = {};

        ...

        -- The BehaviorTreeNode's accessor functions.
        node.AddChild = BehaviorTreeNode.AddChild;
        node.ChildIndex = BehaviorTreeNode.ChildIndex;
        node.GetChild = BehaviorTreeNode.GetChild;
        node.GetNumberOfChildren =
            BehaviorTreeNode.GetNumberOfChildren;
        node.SetAction = BehaviorTreeNode.SetAction;
        node.SetEvaluator = BehaviorTreeNode.SetEvaluator;
        node.SetType = BehaviorTreeNode.SetType;

        return node;
    end
**Actions**

The first node type is a basic action. We can create a wrapper function that will instantiate an action node and set the internal action accordingly. Actions are only designed to execute behaviors on an agent and shouldn't be assigned any children. They should be considered leaves in a behavior tree:

**SoldierLogic.lua**:

```lua
local function CreateAction(name, action)
    local node = BehaviorTreeNode.new(
        name, BehaviorTreeNode.Type.ACTION);
    node:SetAction(action);
    return node;
end
```

**Conditions**

Conditions are similar to actions and are also leaves in a behavior tree. Condition nodes will execute the evaluator assigned to them and return the result to the caller to determine how they should be processed.

**SoldierLogic.lua**:

```lua
local function CreateCondition(name, evaluator)
    local condition = BehaviorTreeNode.new(
        name, BehaviorTreeNode.Type.CONDITION);
    condition:SetEvaluator(evaluator);
    return condition;
end
```

**Selectors**

Selectors are the first type of nodes that can have children within the behavior tree. A selector can have any number of children, but will only execute the first child that is available for execution. Essentially, selectors act as `if, if...else, and else` structures within behavior trees. A selector will return true if at least one child node is able to run; otherwise, the selector returns false:

**SoldierLogic.lua**:

```lua
local function CreateSelector()
    return BehaviorTreeNode.new(
        "selector", BehaviorTreeNode.Type.SELECTOR);
end
```
Sequences

Lastly, we have sequences, which act as sequential blocks of execution that will execute each of their children in an order until a condition, selector, or child sequence fails to execute. Sequences will return true if all their children run successfully; if any one of their children returns false, the sequence immediately exits and returns false in turn:

SoldierLogic.lua:

```lua
local function CreateSequence()
    return BehaviorTreeNode.new(
        "sequence", BehaviorTreeNode.Type.SEQUENCE);
end
```

Creating a behavior tree object

Creating a behavior tree object is simple, as it primarily consists of a root node, evaluation function, and update function:

BehaviorTree.lua:

```lua
require "BehaviorTreeNode"
BehaviorTree = {};

local _EvaluateSelector;
local _EvaluateSequence;

function BehaviorTree.SetNode(self, node)
    tree.node_ = node;
end

function BehaviorTree.new()
    local tree = {};

    -- The BehaviorTree's data members.
    tree.currentNode_ = nil;
    tree.node_ = nil;

    return tree;
end
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
Behavior tree helper functions

Four primary evaluators are used to process the behavior tree structure: selector evaluation, sequence evaluation, actual node evaluation, and finally, a continue evaluation function that continues where a sequence's child finishes:

BehaviorTree.lua:

    local EvaluateSelector;
    local EvaluateSequence;

Selector evaluation

As selectors only return false if all child nodes have executed without returning true, we can iterate over all children and return the first positive result we get back. We need to return two values, the evaluation result as well as an action node if one is found. To do this, we'll return a table containing both values.

As our behavior tree can be of any arbitrary depth, we will recursively evaluate both selectors and sequences till we have a return result:

BehaviorTree.lua:

    _EvaluateSelector = function(self, node, deltaTimeInMillis)
        -- Try and evaluate all children. Returns the first child
        -- that can execute. If no child can successfully execute the
        -- selector fails.

        for index = 1, #node.children_ do
            local child = node:GetChild(index);

            if (child.type_ == BehaviorTreeNode.Type.ACTION) then
                -- Execute all Actions, since Actions cannot fail.
                return { node = child, result = true};
            elseif (child.type_ == BehaviorTreeNode.Type.CONDITION) then
                -- Conditions are only valid within sequences, if one
                -- is encountered in a selector the tree is malformed.
                assert(false);
                return { result = false };
            elseif (child.type_ == BehaviorTreeNode.Type.SELECTOR) then

                -- Recursively evaluate child selectors.
                local result = _EvaluateSelector(

            end
        end

        return { result = false };
    end
Sequence evaluation

A sequence is nearly the opposite of a selector where the first failure will result in the sequence returning a failure. As sequences can execute multiple actions sequentially, we can take in an index number that represents the current child from which we should start our evaluation. This allows the behavior tree to continue the evaluation from where it left off:

BehaviorTree.lua:

```lua
_behaviorTreeNodeType.Sequence = function(self, node, deltaTimeInMillis, index)
  -- Try and evaluate all children. Returns a false result if a
  -- child is unable to execute, such as a condition failing or
  -- child sequence/selector being unable to find a valid Action
  -- to run.
  index = index or 1;

  for count=index, #node.children_ do
    local child = node:GetChild(count);

    if (child.type_ == BehaviorTreeNode.Type.ACTION) then
      -- Execute all Actions, since Actions cannot fail.
      _EvaluateSequence(self, child, deltaTimeInMillis);
    end
  end

  return { result = false };  
  
```

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Decision Making

    return { node = child, result = true};
else if (child.type_ ==
    BehaviorTreeNode.Type.CONDITION) then

    local result = child.evaluator_ (self.userData_);
    -- Break out of execution if a condition fails.
    if (not child.evaluator_ (self.userData_)) then
        return { result = false };  
    end
    elseif (child.type_ ==
    BehaviorTreeNode.Type.SELECTOR) then

        local result = _EvaluateSelector(
            self, child, deltaTimeInMillis);
        -- Unable to find an Action to run, return failure.
        if (not result.result) then
            return { result = false };  
        elseif (result.result and result.node ~= nil) then
            -- Found an Action to execute, pass the result
            -- back to the caller.
            return result;
    end
    -- A selector must return an Action to be considered
    -- successful, if no Action was found, then the
    -- selector failed.
elseif (child.type_ ==
    BehaviorTreeNode.Type.SEQUENCE) then

        local result = _EvaluateSequence(
            self, child, deltaTimeInMillis);
        -- Sequence reported failure, propagate failure to the
        -- caller.
        if (not result.result) then
            return { result = false };  
        elseif (result.result and result.node ~= nil) then
            -- Found an Action to execute, pass the result
            -- back to the caller.
            return result;
    end
    -- There is a third possible case, the sequence
    -- completed successfully and has no additional

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
-- children to execute. In that case let the sequence
-- continue executing additional children.
end

-- Move to the next child to execute.
count = count + 1;
end

-- Returns success without an Action to run if all children
-- executed successfully.
return { result = true }; end

Node evaluation

Basic node evaluation is only used to begin the recursive tree evaluation of the root node and handles all four possible root node type evaluations. If the root node is a condition, an assert is called to indicate a malformed tree:

BehaviorTree.lua:

local function _EvaluateNode(self, node, deltaTimeInMillis)
  if (node.type_ == BehaviorTreeNode.Type.ACTION) then
    -- No further evaluation is necessary if an Action is
    -- found.
    return node;
  elseif (node.type_ == BehaviorTreeNode.Type.CONDITION) then
    -- Conditions should be evaluated immediately, if the
    -- behavior tree is trying to evaluate this node, there is
    -- something structurally wrong in the behavior tree.
    assert(false); -- invalid structure
  elseif (node.type_ == BehaviorTreeNode.Type.SELECTOR) then
    -- Treat the node like a selector and find the first valid
    -- child action.
    local result = _EvaluateSelector(
      self, node, deltaTimeInMillis);

    if (result.result) then
      return result.node;
    end
  elseif (node.type_ == BehaviorTreeNode.Type.SEQUENCE) then
    -- Treat the node like a sequence and find the first valid
    -- child action.
    local result = _EvaluateSequence(
      self, node, deltaTimeInMillis);

    if (result.result) then
      return result.node;
    end
  else
    -- Handle other node types here.
  end
end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
if (result.result) then
    return result.node;
end
end

Continue behavior tree evaluation

Continuing evaluation of the behavior tree is required whenever an action is completed, as the action's parentage could contain a sequence that the behavior tree must continue evaluating. In order to continue every possible sequence, we can use the currently executing node's parent to determine whether it's part of a sequence. We will continue moving through parents until the root node is processed and only continue sequence evaluation if one of the encountered parents is a sequence:

BehaviorTree.lua:

local function _ContinueEvaluation(self, node, deltaTimeInMillis)
    local parentNode = node:GetParent();
    local childNode = node;

    -- Navigates upward within the tree to find any sequences that
    -- require continued evaluation.
    while (parentNode ~= nil) do
        if (parentNode.type_ == BehaviorTreeNode.Type.SEQUENCE) then
            -- Found a sequence, continue evaluating from the
            -- current executing node within the sequence.
            local childIndex = parentNode:ChildIndex(childNode);

            -- So long as the executing child was not the last
            -- node within the sequence, evaluate the sequence
            -- starting on the next child node.
            if (childIndex <
                parentNode:GetNumberOfChildren()) then
                return _EvaluateSequence(
                    self,
                    parentNode,
                    deltaTimeInMillis,
                    childIndex + 1);
            end
        end
    end
end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
end

-- Move one parent up in the tree.
childNode = parentNode;
parentNode = childNode:GetParent();
end

end

The behavior tree update loop

The main behavior tree update loop is responsible for both picking the current action that should be executing as well as initializing, updating, and terminating any running action:

BehaviorTree.lua:

function BehaviorTree.Update(self, deltaTimeInMillis)
  if (self.currentNode_ == nil) then
    -- Find the first valid Action to execute.
    self.currentNode_ = _EvaluateNode(
      self, self.node_, deltaTimeInMillis);
  end

  if (self.currentNode_ ~= nil) then
    local status = self.currentNode_.action_.status_

    if (status == Action.Status.UNINITIALIZED) then
      self.currentNode_.action_:Initialize();
    elseif (status == Action.Status.TERMINATED) then
      self.currentNode_.action_:CleanUp();
    elseif (status == Action.Status.RUNNING) then
      self.currentNode_.action_:Update(deltaTimeInMillis);
    end
  end
end

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Updating the behavior tree

Now that the internals of our behavior tree are fleshed out, we can update the behavior tree class with the `Update` and `SetNode` functions:

**BehaviorTree.lua**:

```lua
function BehaviorTree.new(userId)
  local tree = {};

  -- The BehaviorTree's data members.
  tree.currentNode_ = nil;
  tree.node_ = nil;
  tree.userData_ = userId;

  -- The BehaviorTree's accessor functions.
  tree.SetNode = BehaviorTree.SetNode;
  tree.Update = BehaviorTree.Update;

  return tree;
end
```

Building a behavior tree

Building a behavior tree is very similar to building a decision tree, except for the addition of selectors and sequence node types.
We can start creating a behavior tree using a similarly wrapped function that instantiates a behavior tree and creates the first selector node for the tree:

**Soldier Logic.lua**:

```lua
function SoldierLogic_BehaviorTree(userData)
    local tree = BehaviorTree.new(userData);
    local node;
    local child;

    node = CreateSelector();
    tree:SetNode(node);
    return tree;
end
```

The soldier's behavior tree

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
The death behavior

To add the first action, which is death, we add the required sequence, condition, and action nodes. As the behavior tree will completely re-evaluate once an action completes, we don’t have to worry about other actions knowing about death. As prioritizing actions is based on how early in the tree they appear, we add death first, because it has the highest priority:

SoldierLogic.lua:

```lua
function SoldierLogic_BehaviorTree(userData)
    ...
    -- die action
    child = CreateSequence()
    node:AddChild(child);
    node = child;
    child = CreateCondition(
        "is not alive", SoldierEvaluators_IsNotAlive);
    node:AddChild(child);
    node = child;
    node = child:GetParent();
    child = CreateAction("die", DieAction(userData));
    node:AddChild(child);
    node = child;

    return tree;
end
```

The flee behavior

Fleeing is another important behavior that extends the agent's lifespan. As fleeing is less important than death, the flee action has a lower priority in the behavior tree than the death behavior:

SoldierLogic.lua:

```lua
function SoldierLogic_BehaviorTree(userData)
    ...
    -- flee action
    node = node:GetParent();
    child = CreateAction("flee", FleeAction(userData));
    node:AddChild(child);
    node = child;
```

For More Information:

www.packtpub.com/game-development/learning-game-ai-programming-lua
node = node:GetParent();
child = CreateSequence();
node:AddChild(child);
node = child;

child = CreateCondition(
    "has critical health",
    SoldierEvaluators_HasCriticalHealth);
node:AddChild(child);
node = child;

node = node:GetParent();
child = CreateAction("flee", FleeAction(userData));
node:AddChild(child);
node = child;

return tree;
end

**Combat behaviors**

Combat behaviors encompass reloading, shooting, and pursuit; all have a common SoldierEvaluators_HasEnemy condition that must be true for any of these actions to execute. The strength of a behavior tree allows you to group common behaviors under common conditionals to reduce costly evaluations.

As our agent must choose a combat behavior if they have an enemy, the fallback action, pursuit requires no additional conditions before executing. If the behavior isn't able to process reloading or shooting, the pursuit action will be chosen automatically:

**SoldierLogic.lua:**

```lua
function SoldierLogic_BehaviorTree(userData)
...

-- reload/shoot/pursue actions
node = node:GetParent();
node = node:GetParent();
child = CreateSequence();
```
Decision Making

```lua
node:AddChild(child);
node = child;

child = CreateCondition("has enemy", SoldierEvaluators_HasEnemy);
node:AddChild(child);
node = child;

node = node:GetParent();
child = CreateSelector();
node:AddChild(child);
node = child;
```

The reload behavior
Creating a reload behavior consists of a sequence with a single conditional and the main reload action:

SoldierLogic.lua:

```lua
function SoldierLogic_BehaviorTree(userData)

...

-- reload action
child = CreateSequence();
node:AddChild(child);
node = child;

child = CreateCondition("has no ammo", SoldierEvaluators_HasNoAmmo);
node:AddChild(child);
node = child;

node = node:GetParent();
child = CreateAction("reload", ReloadAction(userData));
node:AddChild(child);
node = child;
```

For More Information:
The shoot behavior

The shoot behavior is identical to reloading, except that we must traverse back to the parent selector node to add the shoot sequence:

SoldierLogic.lua:

```lua
function SoldierLogic_BehaviorTree(userData)

    ...

    -- shoot action
    node = node:GetParent();
    node = node:GetParent();
    child = CreateSequence();
    node:AddChild(child);
    node = child;

    child = CreateCondition(
        "can shoot enemy", SoldierEvaluators_CanShootAgent);
    node:AddChild(child);
    node = child;

    node = node:GetParent();
    child = CreateAction("shoot", ShootAction(userData));
    node:AddChild(child);
    node = child;

    ...

The pursue behavior

As pursuit is the action our agents can perform if they have an enemy, no condition is necessary:

SoldierLogic.lua:

```lua
function SoldierLogic_BehaviorTree(userData)

    ...

    -- pursuit action
    node = node:GetParent();
    node = node:GetParent();
```
child = CreateAction("pursue", PursueAction(userData));
node:AddChild(child);
node = child;

return tree;
end

The move behavior
If our agent is unable to pursue an enemy and isn't fleeing, a general move behavior should operate on any target position our agent is currently moving toward. As target positions are only set by random move actions, the move behavior must have higher priority than a random move action; otherwise our agent will never be able to act on its target position:

SoldierLogic.lua:

function SoldierLogic_BehaviorTree(userData)
...
-- move action
node = node:GetParent();
node = node:GetParent();
node = node:GetParent();
child = CreateSequence();
node:AddChild(child);
node = child;

child = CreateCondition("has move position", SoldierEvaluators_HasMovePosition);
node:AddChild(child);
node = child;

node = node:GetParent();
child = CreateAction(
    "move to position", MoveAction(userData));
node:AddChild(child);
node = child;

return tree;
end
The random move behavior
Randomly moving is one of the least prioritized behaviors and is a tossup between randomly choosing a spot to move to and just idling at a place:

```lua
function SoldierLogic_BehaviorTree(userData)

...

-- random action
node = node:GetParent();
node = node:GetParent();
child = CreateSequence();
node:AddChild(child);
node = child;

child = CreateCondition(
    "50/50 chance", SoldierEvaluators_Random);
node:AddChild(child);
node = child;

node = node:GetParent();
child = CreateAction(
    "random move", RandomMoveAction(userData));
node:AddChild(child);
node = child;

return tree;
end
```
The idle behavior

Just in case every other action has failed, the last action in a behavior tree should be something that can always execute. Behavior trees must find an action to execute regardless of the state of the agent. In our case, we simply allow the agent in order to idle at a place:

```lua
function SoldierLogic_BehaviorTree(userData)
    ...
    -- idle action
    node = node:GetParent();
    node = node:GetParent();
    child = CreateAction("idle", IdleAction(userData));
    node:AddChild(child);
    node = child;

    return tree;
end
```

Creating a behavior tree agent

To use our behavior tree, we can replace the finite state machine and perform the same update loop on our behavior tree instead. Regardless, if our agent is using a decision tree, state machine, or behavior tree, its actions will be nearly identical, as the logic is merely translated from one decision structure to another:

`IndirectSoldierAgent.lua`:

```lua
local soldier;
local soldierController;
local soldierBT;
local soldierUserData;

function Agent_Initialize(agent)
    ...
    soldierBT = SoldierLogic_BehaviorTree(
```
function Agent_Update(agent, deltaTimeInMillis)
    if (soldierUserData.alive) then
        soldierBT:Update(deltaTimeInMillis);
    end

    soldierController:Update(agent, deltaTimeInMillis);
end

Strengths of behavior trees
Compared to decision trees, behavior tree actions know very little about actions other than the priority they show up in the tree. This allows for actions to be modular in nature and can reduce the need to rebalance a complex tree when more actions are added.

Pitfalls of behavior trees
With reactive actions, behavior trees have shortcomings; they represent stateful logic very poorly. If statefulness needs to be preserved within a behavior tree, typically high-level conditions will dictate which branch is currently active. For instance, the noncombat and combat state of our agent isolates a lot of the behaviors that are available at any point in time.

Summary
With some rudimentary actions and decision-making logic controlling our agents, we can now begin to enhance how our agents see the world, as well as how they store information about the world.

In the next chapter, we'll create a data structure that can store knowledge as well as create senses for our agents to actually see and hear.

For More Information:
www.packtpub.com/game-development/learning-game-ai-programming-lua
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.