8

Simulating Behavior Trees

A Behavior Tree/Planner Hybrid Approach

Daniel Hilburn

8.1 Introduction

Game AI must handle a high degree of complexity. Designers often represent AI with complex state diagrams that must be implemented and thoroughly tested. AI agents exist in a complex game world and must efficiently query this world and construct models of what is known about it. Animation states must be managed correctly for the AI to interact with the world properly. AI must simultaneously provide a range of control from fully autonomous to fully designer-driven.

At the same time, game AI must be flexible. Designers change AI structure quickly and often. AI implementations depend heavily on other AI and game system implementations, which change often as well. Any assumptions made about these external systems can easily become invalid—often with little warning. Game AI must also interact appropriately with a range of player styles, allowing many players to have fun playing your game.

There are many techniques available to the AI programmer to help solve these issues. As with any technique, all have their strengths and weaknesses. In the next couple of sections, we'll give a brief overview of two of these techniques: behavior trees and planners. We'll briefly outline which problems they attempt to solve and the ones with which they struggle. Then, we'll discuss a hybrid implementation which draws on the strengths of both approaches.

8.1.1 Behavior Trees

A behavior tree in game AI is used to model the behaviors that an agent can perform. The tree structure allows elemental actions (e.g., jump, kick) to be combined to create

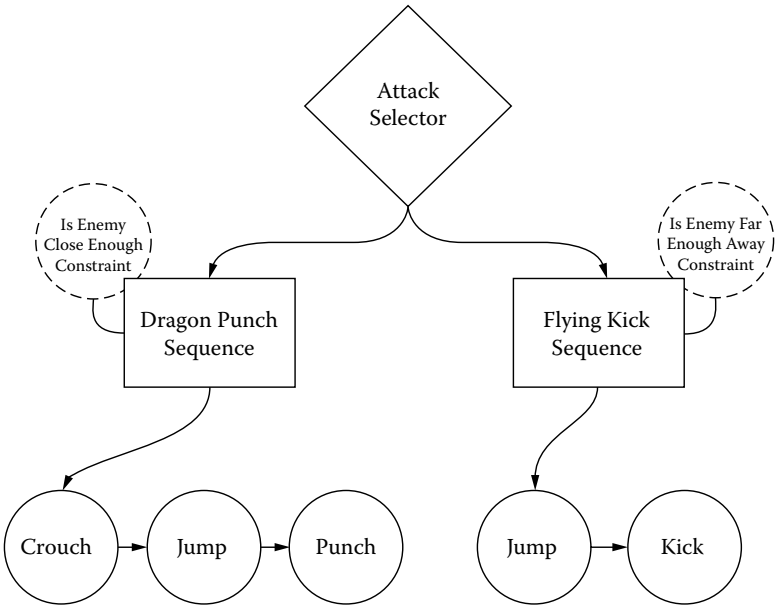

Figure 8.1

Example of a behavior tree.

a higher level behavior (e.g., flying kick). This higher level behavior can then be treated as an elemental behavior and used to compose even higher level behaviors (e.g., attack). Behavior trees also include the concept of *constraints*, which can be attached to behaviors at any level in the tree to keep that behavior from being selected when the state of the world does not match the state required by the constraint (Figure 8.1).

Behavior trees are great at modeling what an AI *can* do. They allow designers to take very low-level actions and combine them to create exactly the set of high-level actions that the designer wants available to the AI. The tree structure easily models any complex design that is thrown at it. Conveniently, this structure also closely resembles the diagrams that designers often use to describe the AI, which allows designers and programmers to speak the same language when discussing AI implementation. The tree structure is also easily configurable, especially if a graphical design tool is available. This allows the designer to rapidly iterate and refine the AI design.

Unfortunately, behavior trees are not so great at specifying what an AI *should* do. In order for the tree to know which action it should perform, it must have intimate knowledge about the world state, including how other AI or game systems are implemented. It must also know how each of its behaviors affects—and is affected by—changes in the world state. This results in a web of dependencies on other systems which are likely to change. If any of these systems change, you'll have to update your behavior tree accordingly. This sort of design is far more brittle than we would like. It is much more preferable that the behavior tree works properly with no modifications even when other systems or its own internal structure changes. Later, we'll discuss ways to solve these issues by taking some cues from AI planners.

8.1.2 Planners

A planner in game AI is used to create a sequence of elemental actions to achieve some goal, given the current world state. This sequence of actions is called a *plan*. The planner maintains a model of the world state, a collection of all elemental actions available to an AI, and a goal heuristic. The world state model contains any information about the world that the heuristic needs. For example, a world state might include a list of available enemies and their health values. The planner understands how each action affects the world state, and since a plan is simply a sequence of these actions, the planner also understands how any plan affects the world state. For example, a kick action deals some damage to an enemy if it is close by, while a jump action moves the AI closer to an enemy.

The goal heuristic scores a given plan by how much it achieves the heuristic's goal. In our example, a combat heuristic would give a high score to a plan that results in enemies being damaged. So, if the AI is close to an enemy, a high scoring plan might consist of just a kick action. However, if the AI is too far from an enemy for the kick to deal damage, the plan will receive a low score. But if we insert a jump action before the kick, now the AI can move in and attack an enemy, a plan which would receive a high score. With all of these pieces available, the planner can create plans that achieve high-level goals dynamically, regardless of the current world state (Figure 8.2).

As you can see, planners are great at managing what an AI *should* do. They allow designers to specify high-level goals for the AI by evaluating world states in the planner's heuristic, rather than trying design specific behaviors for specific situations. Planners are able to do this by keeping a very strict separation between what an AI *does* (actions) and what the AI *should do* (heuristics). This also makes the AI more flexible and durable in the face of design changes. If the jump action gets cut because the team didn't have time to

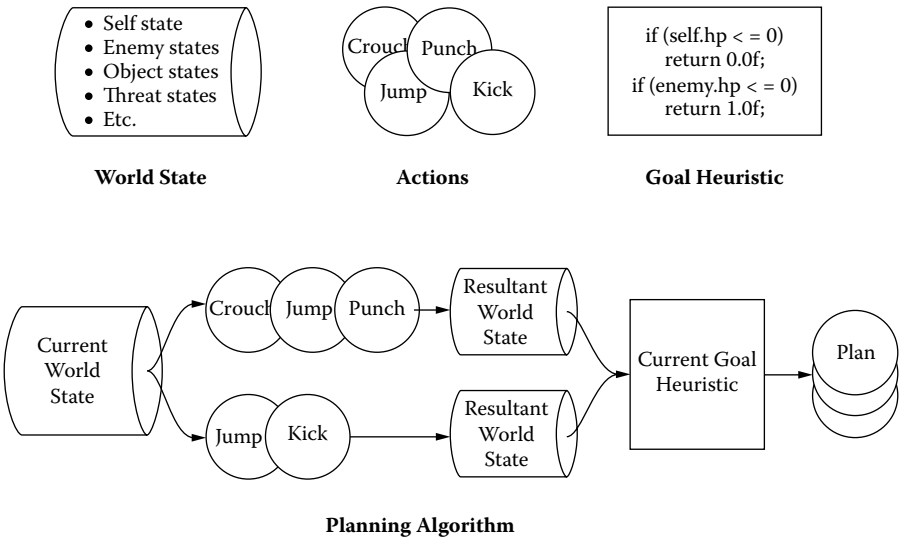

Figure 8.2

Example of a planner.

polish the animations, just remove it. The AI will still create the best possible plan for its current world state. If the kick action suddenly also sends out a shockwave, you only need to add that to the kick action's result description. You don't need to change what the AI *should do* just because you changed what it *does*.

While the flexibility of planners is a great strength, it can also be a great weakness. Often, designers will want to have more control over the sequences of actions that an AI can perform. While it is cool that your AI can create a jump kick plan on its own, it could also create a sequence of 27 consecutive jumps. This breaks the illusion of intelligence our AI should produce, which is obviously not what we want to happen. There are techniques to prevent such undesirable plans, but it is difficult to predict all of the situations where a planner can break down. This, understandably, causes distaste for planners among many designers, as they often prefer more control over how their characters behave.

This is the classic tradeoff that AI designers and programmers have to deal with constantly: the choice between the fully designed (brittle) AI that behavior trees provide and the fully autonomous (unpredictable) AI that planners provide. While this battle has been raging for a while now, it is happily not a binary choice. The space between these two approaches is where the best solutions lie. There are numerous implementations of behavior trees and planners and many other techniques that attempt to solve this problem, one of which is our next topic. I'll quickly describe how this approach works, and then we'll dive into how I implemented it on a recent project.

8.1.3 A Behavior Tree/Planner Hybrid

The basic premise of the hybrid approach is simple: combine the strengths of behavior trees and planners to produce an AI system that is flexible and durable in the face of design changes while allowing the designers full control over the structure of the actions available to the AI. It uses a world state model and heuristic, just like the planner. However, where a planner builds sequences of elemental actions dynamically and uses its heuristic to choose the best one, the hybrid approach uses its heuristic to choose between branches of a premade behavior tree.

Using the behavior tree allows the designers to have full control over what actions are available and to easily redesign the structure of these actions while iterating. However, as we mentioned previously, the behavior tree is usually pretty resistant to design changes, as changing the internal structure of an action must be reflected in parent selector nodes back up the tree. This is where our planner half swoops in to save the day.

Remember that our approach also includes a planner's world state model and a heuristic. We incorporate these into the behavior tree by implementing a simulation step for each node type. For *leaf* nodes, this simulation step simply returns a resultant world state just like elemental actions in the planner system. However, the recursive structure of the behavior tree allows us to recursively simulate composite behaviors as well. *Sequence* nodes simulate each of their child behaviors in sequence and then return the accumulated result. *Selector* nodes simulate each of their child behaviors to determine what the resultant world state would be if that node was selected. These results are then fed through the heuristic function to generate a score for each child behavior. The *selector* node then uses these scores to determine which node to select and returns that node's result as its own.

This design allows us to construct behavior trees that know nothing of their internal structure. We can make an *Attack Selector* node that is composed of any number of attack

behaviors, and it will choose the one most appropriate to the current world state without knowing about *Dragon Punch* or *Flying Kick* or when they are most appropriate. Any selector node just needs to simulate each of its children and select the one with the highest heuristic score. This allows us to change the internal structure of the tree without changing any code further up the hierarchy. It also allows us to change how a leaf action affects the world without worrying about updating the entire tree to compensate for the changed design. This is what we are looking for. The behavior tree structure allows designers to have full control over what the AI *can* do, while the planner mechanism handles determining what the AI *should* do.

8.2 The Jedi AI in *Kinect Star Wars*™

We developed this behavior tree/planner hybrid approach while working on the Jedi AI for *Kinect Star Wars*™ at Terminal Reality Inc.™, which has been gracious enough to provide the source code for this article as example material. We only have space to go over a subset of the system in this article, but the entire system with sample code is provided on the book's website (http://www.gameaipro.com). The proprietary engine stuff is stubbed out, but all of the Jedi AI bits are there if you are interested in looking them over. Without further ado, let's make a Jedi!

8.2.1 Jedi Memory

The first thing our Jedi needs is an internal knowledge of the world. This is the world state model from our previous discussion, which we called the Jedi AI Memory (see Listing 8.1). It encapsulates everything that our actions can manipulate in the world, including the Jedi, the Jedi's current victim, any nearby enemies, and any incoming threats. It also provides a `simulate()` method, which allows any action to update the parts of memory that change over time (e.g., position), and a `simulateDamage()` method, which allows any behavior to easily simulate damage dealt to a given enemy.

8.2.2 Jedi Behavior Tree

Now that we have our world state representation, let's look at the Jedi's behavior tree implementation. All of our behavior tree nodes, which we called *Actions*, provide the standard begin/update/end behavior tree node interfaces. These nodes return an `EJediAiActionResult` value to their parent from their begin and update operations to let the parent know the Action's current status (see Listing 8.2).

The *Actions* also provide a `checkConstraints()` method, which iterates over a list of attached *Constraint* objects (see Listing 8.3). This method may also be overridden to allow *Action* subclasses to check *Constraints* which are specific to those subclasses. The *Constraint* objects provide options to skip the constraint while the action is in progress or while the action is being simulated, which allow the constraint subclasses a bit of stability. For example, let's consider the *Distance Constraint* attached to the *Dragon Punch* sequence to prevent our AI from executing it when the enemy is too far away. If we start the sequence and the enemy moves far enough away to cause the constraint to fail, the AI will immediately bail out of the sequence, which may not be desirable. It may be more desirable for the AI to continue executing the sequence and simply miss the enemy. If we set up the constraint to be skipped while the action is in progress, this is exactly what will happen.

Listing 8.1. The following is an example of the Jedi AI Memory (world state) that our AI uses to keep track of the world and simulate behaviors.

```
class CJediAiMemory {
public:
    //simulate this AI memory over a given timestep
    void simulate(float dt);
    //simulate damage to an actor
    void simulateDamage(float dmg, SJediAiActorState &actor);
    //data about my self's current state
    struct SSelfState {
        float skillLevel, hitPoints;
        CVector pos, frontDir, rightDir;
    } selfState;
    //knowledge container for world entities
    struct SJediAiEntityState {
        CVector pos, velocity;
        CVector frontDir, rightDir, toSelfDir;
        float distanceToSelf, selfFacePct, faceSelfPct;
    };
    //knowledge container for other actors
    struct SJediAiActorState : SJediAiEntityState {
        EJediEnemyType type;
        float hitpoints;
    };
    //victim state
    SJediAiEntityState *victimState;
    //enemy state list
    enum {kEnemyStateListSize = 8};
    int enemyStateCount;
    SJediAiActorState enemyStates[kEnemyStateListSize];
    //knowledge container for threats
    struct SJediAiThreatState : SJediAiEntityState {
        EJediThreatType type;
        float damage;
    };
    //threat state list
    enum {kThreatStateListSize = 8};
    int threatStateCount;
    SJediAiThreatState threatStates[kThreatStateListSize];
};
```

Finally, the *Actions* provide our simulation interface. Each *Action* contains a simulation summary object which encapsulates everything that our heuristic function cares about. This summary also contains an `EJediAiActionSimResult` value, which is computed by the heuristic and specifies the desirability of the action. Originally, we used a floating-point number between 0 and 1 to specify this value, but it was very difficult to get stable, predictable results from the heuristic that way. We simplified the result to the values in Listing 8.4.

Now that we have specified all of the pieces of an *Action*, we can bring them all together in the `CJediAiAction` class in Listing 8.5. It provides the standard begin/update/end interface, the simulation interface, and the Constraint interface.

Listing 8.2. The returned results of an Action's begin and update operations:

```
//jedi ai action results
enum EJediAiActionResult {
    eJediAiActionResult_Success = 0,
    eJediAiActionResult_InProgress,
    eJediAiActionResult_Failure,
    eJediAiActionResult_Count
};
```

Listing 8.3. The behavior tree's Constraint implementation.

```
//base class for all jedi ai constraints
class CJediAiActionConstraint {
public:
    //next constraint in the list
    CJediAiActionConstraint *nextConstraint;
    //don't check this constraint while in progress
    bool skipWhileInProgress;
    //don't check this constraint while simulating
    bool skipWhileSimulating;
    //check our constraint
    virtual EJediAiActionResult checkConstraint(
        const CJediAiMemory &memory,
        const CJediAiAction &action,
        bool simulating) const = 0;
};
```

Listing 8.4. The behavior tree's simulation summary, which the heuristic uses to score an Action's desirability.

```
//jedi ai action simulation result
enum EJediAiActionSimResult {
    eJediAiActionSimResult_Impossible,
    eJediAiActionSimResult_Hurtful,
    eJediAiActionSimResult_Irrelevant,
    eJediAiActionSimResult_Cosmetic,
    eJediAiActionSimResult_Beneficial,
    eJediAiActionSimResult_Urgent,
    eJediAiActionSimResult_Count
};
//jedi ai action simulation summary data
struct SJediAiActionSimSummary {
    EJediAiActionSimResult result;
    float selfHitPoints, victimHitPoints, threatLevel;
};
```

Listing 8.5. The behavior tree's abstract base Action class.

```
class CJediAiAction {
public:
    //standard begin/update/end interface
    virtual EJediAiActionResult onBegin();
    virtual EJediAiActionResult update(float dt) = 0;
    virtual void onEnd();
    //simulate this action on the specified memory object
    virtual void simulate(
        CJediAiMemory &simMemory,
        SJediAiActionSimSummary &simSummary) = 0;
    //check my constraints
    virtual EJediAiActionResult checkConstraints(
        const CJediAiMemory &memory, bool simulating) const;
};
```

Listing 8.6. The behavior tree's abstract base Composite Action class.

```
class CJediAiActionComposite : public CJediAiAction {
public:
    //child actions accessors
    CJediAiAction *getAction(int index);
    virtual CJediAiAction **getActionTable(int *count) = 0;
};
```

Next, we define a *Composite Action*, the base class of all nodes which are composed of subnodes (e.g., *Sequence* or *Selector*). It is pretty simple, providing a common interface for accessing the list of child nodes (see Listing 8.6).

Next, let's look at the *Sequence Action* (see Listing 8.7). It simply runs all of its child *Actions* in sequence, using the method `beginNextAction()`. If any of the actions fail, the *Sequence Action* fails as well. Also, simulating a sequence simulates each of its children, starting with the currently running child if the *Sequence* is currently executing. Each child is simulated using the resultant world state of the previous child's simulation. After all children have been simulated, the *Sequence* computes its own simulation result from the resultant world state.

The *Sequence* class provides a few parameters to let you customize how it operates. One thing that you'll notice is that we encapsulate the parameters into their own object. Encapsulating the parameters this way allows a simple `memset()` to initialize all of the parameter variables, preventing you from forgetting to initialize a new parameter.

Next up is the most important part of the behavior tree: the *Selector* (see Listing 8.8). This class is what decides what the AI will or won't do. The *Selector* does this by calling `selectAction(CJediAiMemory *memory)`, which simulates each of its child behaviors using the provided memory to generate simulation summaries for each. It then calls `compareAndSelectAction()`, which compares these *Action* summaries and selects the *Action* whose summary has the highest result.

Listing 8.7. The behavior tree Sequence class.

```
class CJediAiActionSequence : public CJediAiActionComposite {
public:
    //parameters
    struct {
        //specify a delay between each action in the Sequence
        float timeBetweenActions;
        //allows the Sequence to loop on completion
        bool loop;
        //allows the Sequence to skip over failed actions
        bool allowActionFailure;
        //specify what action result is considered a failure
        EJediAiActionSimResult minFailureResult;
    } sequenceParams;
    //get the next available action in the sequence,
    //starting with the specified index
    virtual CJediAiAction *getNextAction(int &nextActionIndex);
    //begin the next available action in the sequence
    virtual EJediAiActionResult beginNextAction();
};
```

Listing 8.8. The behavior tree Selector class.

```
class CJediAiActionSelector : public CJediAiActionComposite {
public:
    //parameters
    struct SSelectorParams {
        //specify how often we reselect an action
        float selectFrequency;
        //prevents the selected action from being reselected
        bool debounceActions;
        //allow hurtful actions to be selected
        bool allowNegativeActions;
        //if results are equal, reselect the selected action
        bool ifEqualUseCurrentAction;//default is true
    } selectorParams;
    //simulate each action and select which one is best
    virtual CJediAiAction *selectAction(CJediAiMemory *memory);
    //compare action simulation summaries and select one
    virtual int compareAndSelectAction(
        int actionCount, CJediAiAction *const actionTable[]);
};
```

8.2.3 Jedi Simulation

Now that we've defined our behavior tree components, let's have a look at the planner side of things: the simulation. When we begin simulating an *Action*, we create a summary of the current world state. Then, we modify the world state in the same way that the simulating *Action* actually would if it were executed. For example, when simulating the

Listing 8.9. This shows how we condense a Jedi Memory object into a Simulation Summary.

```
//condense the specified memory into a summary
void setSimSummaryMemoryData(
    SJediAiActionSimSummary &summary,
    const CJediAiMemory &memory);
//initialize a summary from the specified memory
void initSimSummary(
    SJediAiActionSimSummary &summary,
    const CJediAiMemory &memory)
{
    summary.result = eJediAiActionSimResult_Impossible;
    setSimSummaryMemoryData(summary, memory);
}
//compute the resultant world state summary
void setSimSummary(
    SJediAiActionSimSummary &summary,
    const CJediAiMemory &memory)
{
    summary.result = computeSimResult(summary, memory);
    setSimSummaryMemoryData(summary, memory);
}
```

SwingSaber Action, we apply damage to the victim and run the world state simulation forward by the same amount of time that it takes to swing our lightsaber. After the simulation is complete, we create a summary of the resultant world state and compute the desirability of this new state compared to the summary of the initial state (see Listing 8.9). This final summary is passed back to the parent *Action* and will be used by the behavior tree when selecting this *Action* from a set of other *Actions*.

The real meat of this system is the planner heuristic, where we compute the simulation result (see Listing 8.10). This function represents our AI's current goal. In this case, the Jedi's only goal was to avoid damage and threats while causing damage to his victim. The heuristic does this by classifying an *Action*'s post-simulation world state as one of the `EJediAiActionSimResult` values (impossible, hurtful, irrelevant, beneficial, etc.).

Now that we've defined how our AI's simulation result is computed, let's have a look at how it fits into an actual simulation step: the *SwingSaber Action* (see Listing 8.11).

8.3 Now Let's Throw in a Few Monkey Wrenches …

Game development is an iterative process, and your system will change many times between conception and final product. Even when your system isn't being redesigned, design changes in other systems can change how your implementation behaves. So, it is imperative that our system handles these changes well. As we discussed earlier, the whole point of our hybrid system is to provide flexibility to handle these changes with as few changes as possible. So let's see how well it does by looking at some design changes from *Kinect Star Wars*™.

Listing 8.10. The planner heuristic, which computes the simulation result.

```
//determine the result of a simulation by comparing a summary
//of the initial state to the post-simulation state
EJediAiActionSimResult computeSimResult(
    SJediAiActionSimSummary &summary,
    const CJediAiMemory &memory)
{
    //if we are more hurt than before, the action is hurtful
    //if we are dead, the action is deadly
    if (memory.selfState.hitPoints < summary.selfHitPoints) {
        if (memory.selfState.hitPoints <= 0.0f) {
            return eJediAiActionSimResult_Deadly;
        } else {
            return eJediAiActionSimResult_Hurtful;
        }
    //if our threat level increased, the action is hurtful
    } else if (memory.threatLevel > summary.threatLevel) {
        return eJediAiActionSimResult_Hurtful;
    //if our threat level decreased, the action is helpful
    //if it decreased by a lot, the action is urgent
    } else if (memory.threatLevel < summary.threatLevel) {
        float d = (summary.threatLevel - memory.threatLevel);
        if (d < 0.05f) {
            return eJediAiActionSimResult_Safe;
        } else {
            return eJediAiActionSimResult_Urgent;
        }
    //if victim was hurt, the action is helpful
    } else if (memory.victimState->hitPoints < summary.victimHitPoints) {
        return eJediAiActionSimResult_Beneficial;
    }
    //otherwise, the sim was irrelevant
    return eJediAiActionSimResult_Irrelevant;
}
```

8.3.1 Jedi Skill Level

Kinect Star Wars™ featured three different types of Jedi: Master Jedi, low-level Padawan Jedi, and the second player Jedi. Originally, these were all implemented using the same design. Later, the design team added the caveat that each Jedi should have a skill level to specify how competent he was at combat. This would allow us to make a Master Jedi, like Mavra Zane, more capable in a fight than your Jedi buddy or the other Padawan Jedi in the game.

We implemented this by having the skill level specify how quickly the Jedi could defeat each enemy type. This allowed Mavra to dispatch enemies quickly, while the Padawan Jedi took much longer. To make this work, we added a `victimTimer` member to our world state to track how much time had elapsed since we acquired our current victim. Then, we added a statement to the heuristic to discourage killing the victim before timer specified by the current skill level had expired (see Listing 8.10).

That was it. We didn't have to change any behavior tree *Actions* or simulation code. The heuristic was already aware if a given action would kill the victim, because we were

Listing 8.11. The SwingSaber Action's simulation method.

```
void CJediAiActionSwingSaber::simulate(
    CJediAiMemory &simMemory,
    SJediAiActionSimSummary &simSummary)
{
    initSimSummary(simSummary, simMemory);
    EJediAiActionResult result;
    for (int i = data.swingCount; i < params.numSwings; ++i)
    {
        //simulate a single swing's duration
        CJediAiMemory::SSimulateParams simParams;
        simMemory.simulate(
            kJediSwingSaberDuration, simParams);
        //apply damage to my target
        simMemory.simulateDamage(
            simMemory.selfState.saberDamage,
            *simMemory.victimState);
        //if my target is dead, I'm done
        if (simMemory.victimState->hitPoints <= 0.0f)
            break;
    }
    setSimSummary(simSummary, simMemory);
}
```

simulating each action instead of hard-coding the selection logic into the selectors. So the planner held up its end of the bargain, allowing us to change goals without modifying any *Actions*.

8.3.2 Jedi Mistakes

Another wrinkle that arose was the idea of mistakes. It isn't realistic for the Jedi to always defeat their enemies; they should sometimes fail. Also, the designers wanted the Jedi AI to demonstrate what not to do against various enemy types. However, our entire system is built on the idea that the Jedi will choose the **best** option. We could make a custom selector that chooses the worst option instead of the best option, but it would still return a negative simulation result to its parent, which would then not select it to run.

At first this seemed like a flaw in the system, until we thought about what defines a "mistake." Obviously, the Jedi will always *try* to choose the best Action available. But what if they made a miscalculation and chose an Action which actually was hurtful? This would pass correctly back up the behavior tree and the hurtful Action would then be chosen. In order to create this miscalculation, we needed to insert incorrect information into the simulation step for any *Action*. Rather than add these special cases to each *Action*, we added a special *Action* called a *FakeSim*. The *FakeSim Action* is a special type of *Composite Action* called a *decorator*, which wraps another *Action* to add extra functionality to it. The *FakeSim* was responsible for adding incorrect information to the wrapped *Action*'s simulation step by modifying the world state directly. For example, there are some enemies that have a shield which makes them invulnerable to lightsaber attacks. If we want a Jedi to attack the enemy to demonstrate that the enemy is invulnerable while the shield is up,

we can wrap the *SwingSaber Action* with a *FakeSim Decorator* which lowers the victim's shield during the simulation step. Then, the *SwingSaber* simulation will think that the Jedi can damage the enemy and give it a good simulation result. This would allow *SwingSaber* to be chosen, even though it won't actually be beneficial.

This ended up being a great way to handle this design requirement. It allows us to insert specific mistakes anywhere in the system without modifying any of the *Action* classes. And it allows us to avoid writing special case code to handle inserting these mistakes. We simply insert a bit of incorrect domain knowledge into the system, which reflects how people make mistakes in real life. So the behavior tree held up its end of the bargain, allowing us to easily design very specific *Action* sequences that the planner couldn't handle on its own.

8.4 Conclusion

We've discussed some of the strengths and weakness with both behavior trees and planners. Behavior trees are great at allowing designers to define exactly what an AI *can* do, and planners are great at allowing designers to easily specify what an AI *should* do. And we've discussed how we can utilize a hybrid approach to realize the strengths of both approaches. Finally, we looked at how this system was used in *Kinect Star Wars*™ to create the Jedi AI. This approach provides designers with all of the control of a behavior tree and all of the durability and flexibility of a planner, allowing it to handle design changes smoothly and with few changes to the code. And that is really the whole point.