

NPC Mechanics and Actions

Ver. 0.1

Yordan Gyurchev
yordan@gyurchev.com

Introduction

Character mechanics is part of what people consider game AI. It is responsible for playing correct animations at the right times, blending between them and keeping the character behaviour mechanically correct.

Character mechanics is pure common sense logic. For example: if your AI decides that your NPC wants to “duck” but it is currently playing a wounded animation it would be stupid and visually wrong to interrupt the wounded animation and play “duck”.

Character mechanics also executes very simple sequence logic. For example an enemy has been shot and it is prone on the ground. It needs to get up. These are tree animations: “shot”, “prone” and “getup” that need to be played in a sequence. And it can grow even more complicated if the NPC can react while prone.

Most often character mechanics ends up being part of the AI itself. We however believe that it works best when it is isolated into a layer of its own. Dividing the AI into two separate parts making it easier to develop and debug.

This is what this document is all about.

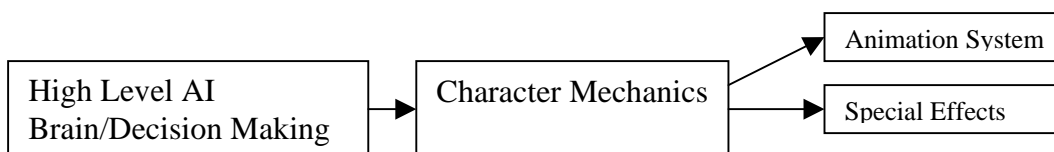
1. AI = Decision making + Mechanics

When we write AI we want to be thinking high level. We want to build behaviours and be concerned with the brain process of the NPCs instead of which exactly animation should we play now and how long can we wait before we can play the next one.

We separate our NPC into two distinct layers: AI and Mechanics

The AI (brain) yields a command every update to what the NPC “should” be doing (for example “go that way”). The mechanics reads this command and tries to comply with it within its logical boundaries.

The character mechanics sends signals to the animation system to play (or blend) animations, to effects system to play special effects, play sounds, etc.



Example: NPC is prone on the ground and is trying to get up. The AI is telling it go move into a certain direction. NPC Mechanics is going to wait till the getup animation is finished before complying with the move command.

This “ignore” capability in the name of consistency is the key.

In fact this is also close to how things happen in real life. Our brain can issue commands to our muscles to execute some action but the laws of physics and mechanics are going to come first.

This way AI can work independently of the mechanics and focus on the high level thinking process of the NPC. AI would always have an idea (command) what the NPC should be doing but the final result will always be filtered through the mechanics and what is physically possible and logical.

1.1 State machines, anyone?

State machines have always been a must-know AI concept. They are a powerful tool AI programmers use to fight the ever so complex problems they face. So lets assume that our high-level AI is going to be a state machine.

Do we put another state machine in the mechanics? What if the two state machines go out of sync? Or the high-level AI insists on a command we don't have a transition for? Do we make all possible transitions?

The way we tackle this problem is to eliminate the state transitions logic from the mechanics and leave just states. This way we can make any transition based on the commands coming from the AI but we also have the opportunity to not comply with it if the current mechanics state is “more important”. We also need some system for identifying and manipulating mechanics state priority.

In our system we call the mechanics states “actions”. Actions are states without transitions that have some priority information added to them.

So to sum it up: AI updates every frame (or less often based on some load balancing) and decides what action needs to be played. Mechanics looks up the action and decides based on action priorities if it can replace the current action with the new one. Mechanics updates the current action every frame.

1.2 Encapsulation

As we hinted before we want to keep the advantages of the state object. We want our action to not be dependent on other actions. This makes development of different actions less error prone and stable in time. It also allows developers to work on different aspects of the same NPC without getting in each other's way.

The usual action class derives from a base action and has zero fan-in¹. This means that “nothing” references the derived class and all calls are to the base interface. Its instantiated trough a factory from serialized data. We could remove an action class

¹ FAN-IN is the count of unique places that call a given class functions either directly, or ultimately, via other functions.

from the project and the game should still compile. To this day we haven't found an action that doesn't satisfy this principle.

This also allows us to replace one action with another and experiment with different ideas with zero complexity code overheads.

1.3 Where is the line?

Where is the line between mechanics and high-level AI? Between brain and action?

I once heard a good definition: "A mechanics state is an atomic coherent action the character can perform"

Mechanics can have minimal amount of logic (what variations of animations to play for example) as long as it doesn't go out of the "atomic and coherent" boundaries. The moment we start asking ourselves "should my action "decide" this or that" we are stepping into the land of "decision making". The action is what character does as a result of the decision making process in the high-level AI.

Here are some examples of actions:

IDLE - can choose to play some idle animations if it likes – it would still do what the brain is telling it – idle

WALK

RUN

WOUND - can select appropriate animations and sounds based on different zones of wound or power of damage.

STRIKE

And here are some examples of high-level/brain states:

SEARCH – uses the AI facilities to select interest points in the environment and navigate to them (uses IDLE, WALK actions)

ATTACK – NPC approaches the target appropriately using the AI systems and attacks it (uses IDLE, WALK, RUN, STRIKE)

You can notice in our example (although very simplified) that WOUND action is never used by the brain states. This is because the WOUND action is triggered by a mechanical reaction in response to the NPC being "damaged" (shot/attacked).

There are number of actions (like WOUND) that are activated externally. If an NPC walks (or is pushed) off a cliff we want play a FALL action.

1.4 Priorities

As mentioned earlier we don't have state transitions (action transitions). Theoretically all transitions are possible. In practice an action can change only if the new required action have better (higher) priority values than the current one.

We have two types of priority values: Start priority and current priority. If the new action "start" priority is higher than the current action "current" priority then it is more important and needs to take over.

It is important to separate the priority values into two groups to avoid deadlocks and also to be able to manipulate the current priority if needed. If our action were to play a single animation we would have a high current priority in the start of the animation and decrease it at the end of the animation giving the next action a chance.

A WOUND action would have a very high starting priority, as it needs to be able to play in most cases. A FALL action would have higher priority than WOUND.

A DEAD action would have a high start priority and maintain its current priority to high – after all NPC cant perform any actions while being dead.

1.5 Classes, Objects and Data

Not all actions need to be represented by unique classes. Most often NPC actions are the same action class with different sets of data.

For example an action animation class can be a “look around” action or an “idle” action depending on the animation it will be playing.

2. Implementation

We already described most of the components of our system. We have Actions, ActionMachine and an AI Command entities.

The actions are the building blocks of the system. They have a common base class that provides the interface and framework for building actions.

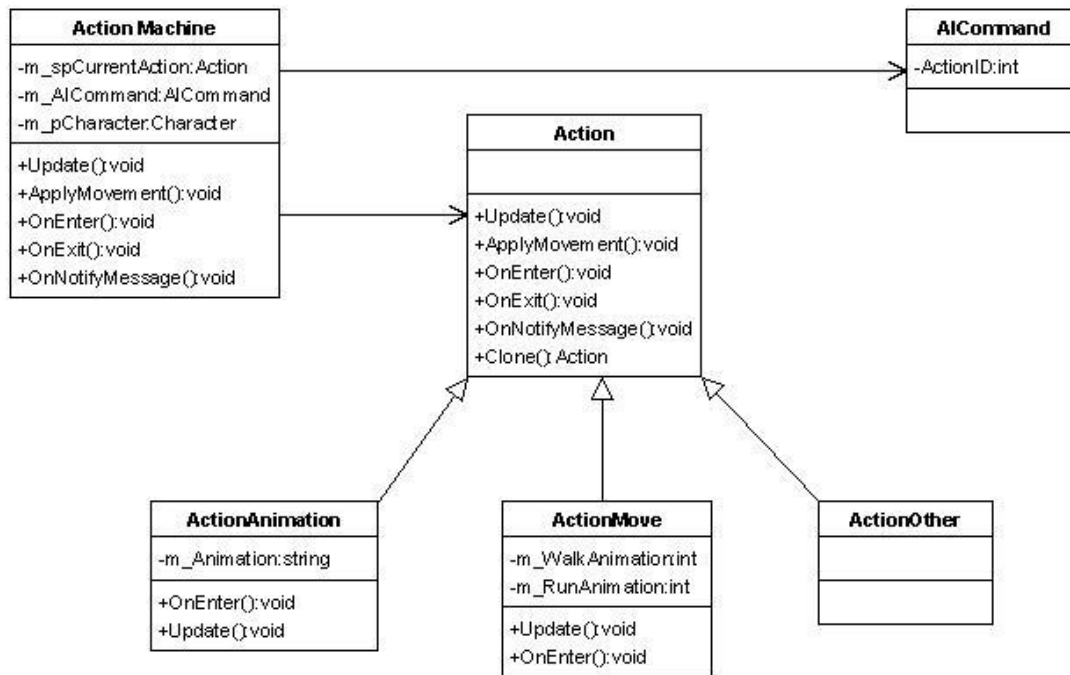
The action machine class is what puts the picture together. It has the context information: NPC pointer, AI Command, list of available actions. Every frame the AI Command is provided from the high-level AI and the action machine has responsibility to check the priority values and try to activate the specified action (if available).

AI Command is a POD. It contains an action identifier and additional data (like velocity, target, look at direction, etc)

For simplicity I have omitted the constructors, destructors and simple accessor functions.

2.1 Interfaces and Class design

The following diagram shows the classes and the relationships between them.



2.2 Action

```

class NPCAction
{
public:
    //every frame functions
    virtual void Update(float fTime);
    virtual void ApplyMovement(HfVector& velocity);

    //initialize and clean up
    virtual void OnEnter();
    virtual void OnExit();

    //external influence
    virtual bool OnNotifyMessage(const GameMessage& msg);

    //clone functionality
    virtual NPCAction * Clone() const;

private:
    unsigned int m_ActionID;
    //priorities
    unsigned int m_nRunPriority;
    unsigned int m_nStartPriority;

    //current can be changed by the action ... drop/raise etc
    unsigned int m_nRunPriorityCurrent;
}
  
```

2.2.1 Update

Every frame the current action is updated. It is given the delta time (since last update cycle). Derived actions are supposed to execute their internal logic (if any) during the update call.

2.2.2 Apply Movement

Apply movement is a concept giving the action option to choose (or modify) the velocity of the NPC character while executing that action.

In most cases we want to use the animation provided velocity (if any) to move the character so his feet don't slide on the ground. In some cases however (cue the FALL action) we want to manually provide/override the NPC velocity.

2.2.3 OnEnter/OnExit

Like any other state class the Action should be allowed to initialize itself and clean up afterwards.

2.2.3 OnNotifyMessage

Gives the action option to react on external influences that don't change the action but can be taken in consideration.

Example: NPC is playing a WOUND action lying of the ground after a powerful hit. However while lying it takes another blow. We could play another WOUND_ON_GROUND action or we can just notify the WOUND action that another damage hit is taken and it will choose the animation accordingly. This would save us from duplicating a lot of code and logic for "getting up" in the two potential "wound" actions.

2.2.3 Clone

Cloning is good way of keeping instanced classes to minimum. We can't have every NPC to have a full set of instanced actions – that would just be a ridiculous memory overhead. There fore once an action is scheduled for activation the ActionMachine "clones" it and then activates the clone object.

2.3 Action Machine

```
class NPCActionMachine
{
public:
    bool Update(const AICommand& command, float fTime);
    HfReal ApplyMovement(HfVector& velocity);

    bool OnNotifyMessage(const HfMessageHeader& msg);

    void SetAvailableActions(const RhNPCActionCollection&);

private:
    NPCActionPtr          m_spCurrentAction;
    AICommand             m_AICommand;
    NPCActionCollection   m_AvailableActions;
    Character*            m_pCharacter;
}
```

2.3.1 Update

The action machine is updated by the NPC that owns it. The NPC has access to the brain so it can acquire the current AI Command and pass it down to the action system.

Update receives the command as parameter and is responsible for checking the priority parameters and changing the action if needed (and possible).

It also needs to “update” the currently active action.

On action change the action machine will call OnExit on the old action. Clone the new one, call OnEnter and set it as current.

2.3.2 ApplyMovement

This call is passed down to the current action for processing.

2.3.3 OnNotifyMessage

This call is passed down to the current action respective function.

2.3 AI Command

```
class AICommand
{
    unsigned int    m_ActionID;

    float           m_Speed;
    Vector          m_Direction;
    Vector          m_LookAtDirection;
    void*           m_pUserData;
    float           m_ExampleParam1;
}
```

The Action ID uniquely identifies the action and allows the action system to clone the right object.

The rest of the parameters are provided by the brain (high-level AI) as additional parameters to the specified action. Actions can use some of the parameters or none depending on what the action is supposed to do.

For example: a WALK action will use the direction parameter to orient the character into that direction and play the walk animation. Alternatively a BLENDED_MOVE action could use the direction to orient the character and the speed to choose a blend factor between run and walk animations.

3 Details and future work

We now have the basics of the action system. Here are some details and options how the system can be used, expanded and refined.

3.1 Feed back

For 99% of the time the information traffic is one way: from high-level AI to the mechanics. Sometimes, however, the high-level AI needs some information back from the mechanics. For example we have some complex attack patterns that can have multiple results that can influence the high-level AI.

3.2 Enough animation

On many occasions animation is the only thing mechanics is concerned with. In some cases however we want to be doing additional things like special effects for example. The action class is the perfect place to link our special effects to whatever it is the NPC is doing (good example would be “casting a spell”).

Imagine a NPC throwing objects. The action is the place where (at specific point in the animation) the throw object is going to be created and thrown in the game world.

A sword trail needs to accompany the attack move – no problem.

3.3 Action Set

The list of actions for a certain NPC represents all possible low-level behaviours for that NPC. It would be good if these action sets can be modified with a tool separately from the game code (as they don't contain anything else than pure data). This way we can tweak our character mechanics without even touching the source code.

3.4 Reusability

Most action classes are generic. This means that they can be reused for different character AI throughout the game. The more generic your actions are the more building blocks you have to build your mechanics.

3.5 Replication

If NPC behaviour is to be replicated over the network the system needs to send only the AI Command as the NPC mechanics are going to be the same.

3.6 Random brains

To test your NPC mechanics you can generate a random brain – a brain that randomly selects the AI commands (actions). Although erratic in decision-making the NPC should never look wrong. There shouldn't be any animation artifacts and the NPC should look as if it obeys the physical laws (although very stupid).

Summary

The described system focuses on the character mechanics. It separates it from the AI thus making it easier to develop and debug.

The actions are encapsulated and dependencies are minimized making development faster and code more robust.