Execution flow

This section describes at which points in time the game engine transfers execution to and from the user-supplied script functions.

The game itself is executed by the system running the game engine code, that is, the binary game code written by the developers, and embedded inside the game's executable files. Amnesia supports scripting by allowing you to associate each level with a corresponding script file (mapname.hps). On a conceptual level, as the game runs, at certain points the control is transferred to the script engine, which executes the user-supplied script, making level-specific things happen, before returning the control back to the game (which takes care of all the rest).

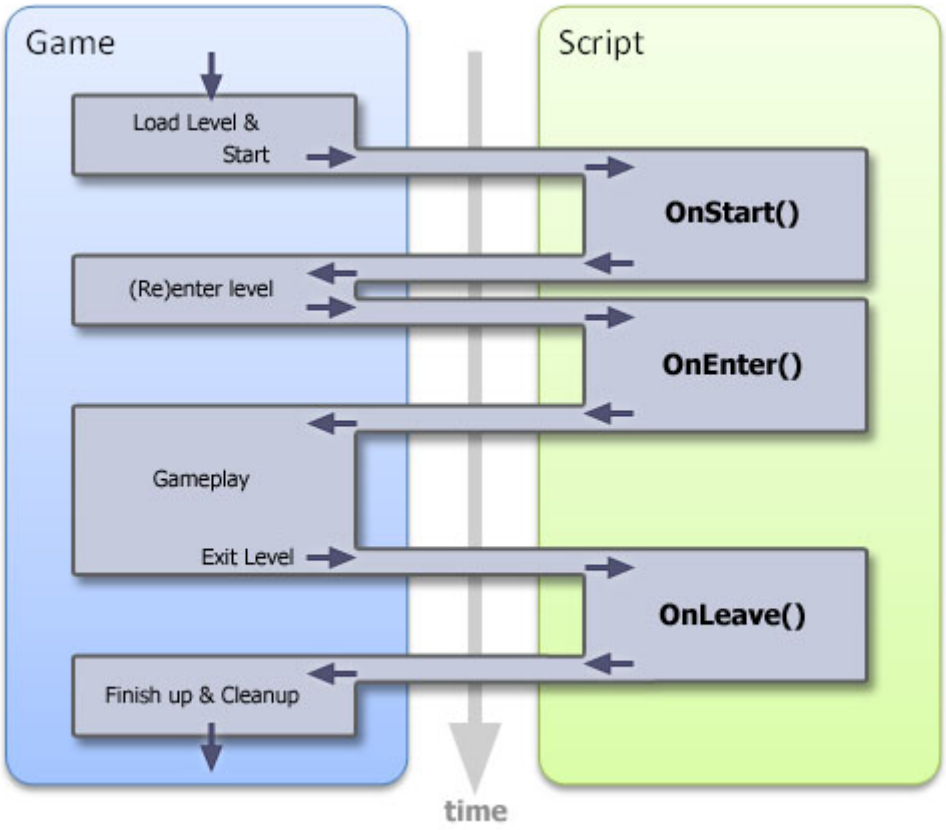
By default, there are only 3 events where this happens. These are listed below:

- level start – this happens when the level is first loaded,
- level enter – this occurs whenever the player (re)enters the level (the player might be allowed to backtrack),
- level leave – this occurs on level transition, when the player leaves one level to enter another.

The HPL2 engine exposes 3 script functions which correspond to these events. These are (respectively):

- OnStart()
- OnEnter()
- OnLeave()

A well-formed script file must have at least one of these functions implemented, although they can be empty (i.e. they do nothing). As the game runs, and a map is loaded, on these level events the control is handed over to the corresponding functions, and they are executed (see the image). By default, these 3 events are the only points in time where this happens.



However, the scripter can use these predefined functions to tell the engine that some other, user-defined functions need to be called at other points in time. This is achieved by using timers, or by telling the game to call specific user-defined functions when certain in-game events occur, like collisions or player-entity interactions. These additional, user-specified functions that the engine is supposed to call when something happens are known as *callbacks*.

For example, inside OnEnter() you can add a timer, and specify (or “hook-up”) your callback function, which defines what should happen when the timer's period is over. The game engine will then wait for the specified time span to elapse, before handing over the execution flow to your callback function.

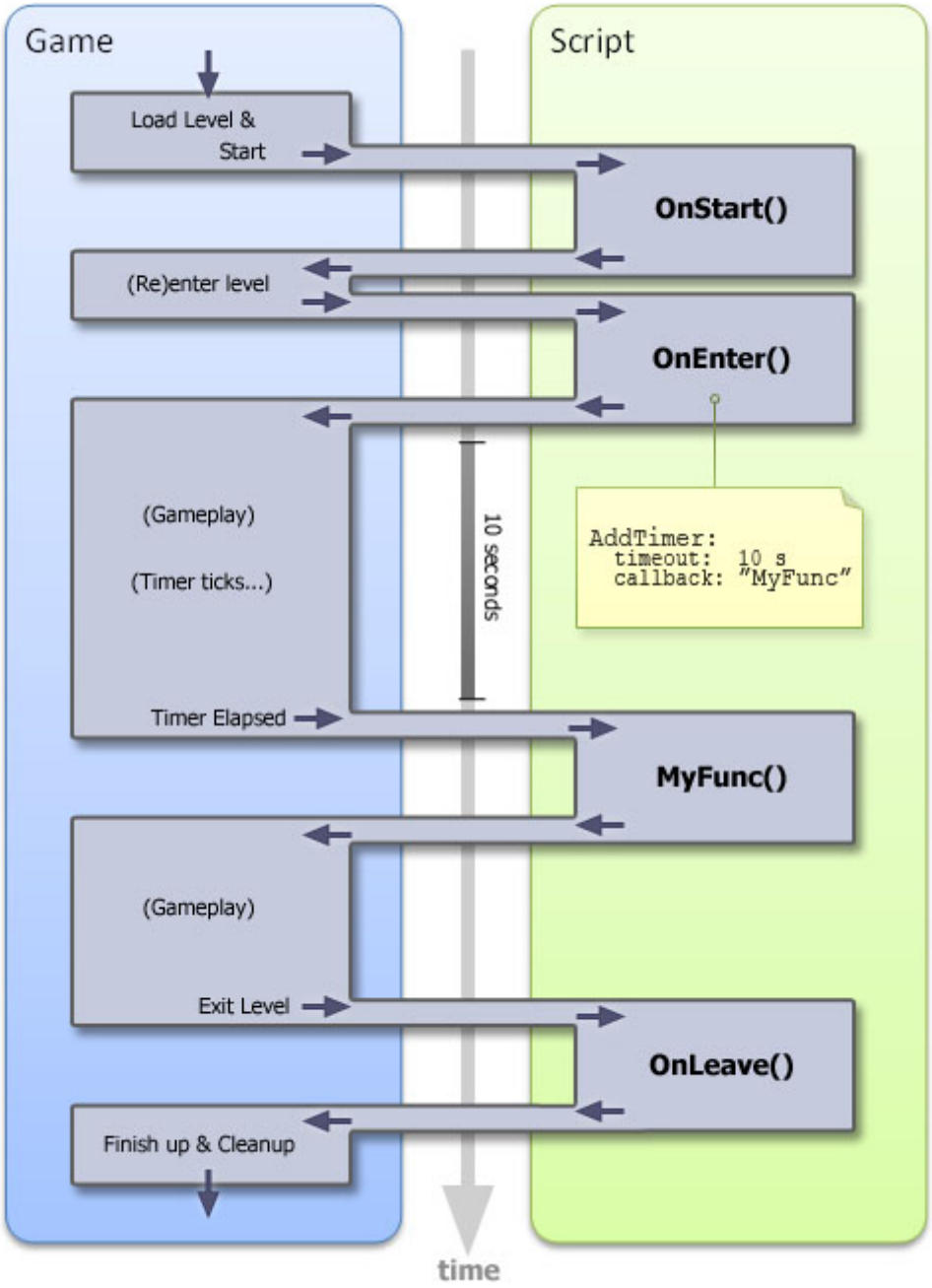
```
void OnEnter()
{
    AddTimer("internal.timer.id", 10.0f, "MyFunc");
}

void MyFunc(string &in timerID)
{
    FadeOut(5.0f); // fades the screen to black
}
```

At that point, the callback will execute, allowing for the script code inside the function to do it's job. When the callback function reaches it's end, the game takes over again.

In the example code above, when the level is entered, OnEnter() gets called. Inside it, the AddTimer() finction adds a 10 second timer to the game, and sets the callback function to “MyFunc”, which is defined bellow. Then the game takes over. After 10 seconds, the game makes the call to MyFunc() script function, passing in the timer id as the timerID parameter (here “internal.timer.id”), so that it

can be checked inside the callback if required (but it is not used here).



Inside the MuFunc() callback, a single call is made to the FadeOut() function, which fades the screen to black in 5 seconds. Note that this function just signals to the game that it should fade out the screen; the FadeOut() function returns immediately, not after 5 seconds, allowing the game engine to take the control back as soon as possible. (Otherwise, the gameplay might be blocked for those 5 seconds!). So, after a quick call to FadeOut(), which signals the game engine what should happen with the screen, the end of MyFunc() is reached, and the game takes over immediately.

Event callbacks can be added at any location in the script that is going to be executed. For example, with timers you often want the timer callback to be called more than once (say, every 5 seconds). A common pattern is to add the first timer from within OnEnter(), and then to assign the same callback function at the end of the callback itself, like this:

```
void OnEnter()
```

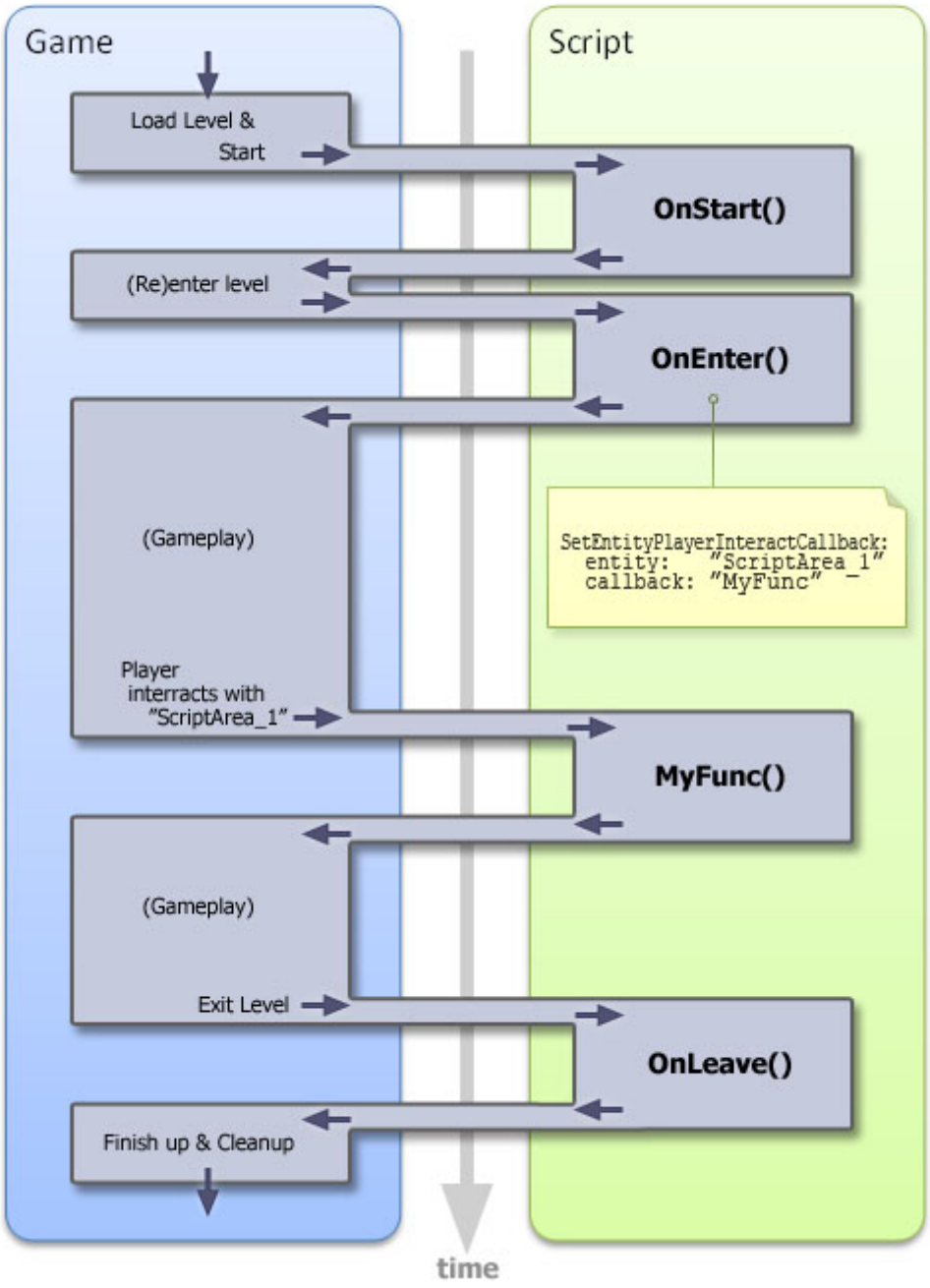
```
{
    AddTimer("internal.timer.id", 5.0f, "MyFunc");
}

void MyFunc(string &in timerID)
{
    // do something...

    // Call this function again in 5 seconds
    AddTimer("internal.timer.id", 5.0f, "MyFunc");
}
```

Similarly, callbacks for other types of events may be defined. You can assign callback functions for collisions, player interaction events, item use events, player “look-at” events, lantern lit event, button/lever operation events, respawn events, etc.

The diagram below depicts the execution flow for a script which adds a player interact callback.



Some callback functions may be hooked up directly from the the level editor, by setting the appropriate properties on an entity. The game will check if the specified functions exist in the script file, and if they do, a call to each will be made when the conditions are met.

- For more information about the script functions exposed by the engine, see Engine Scripts.
- To learn more about the editors and other tools, go here.
- For information on setting up Amnesia for modding and scripting, see Setting Up Developement Environment.