Custom Actors

Sometimes the base game doesn't offer exactly what you need for your level. That's when you should turn to custom actors!

> This guide provides a basic overview of how to create custom actors. A decent understanding of C++ along with some prior NSMB code modding knowledge are recommended for this tutorial.

Section 1 - The Boiler Plate

This is code you will be writing basically every time you go to create a custom actor. It is standard practice to split your actor into two files `myActor.cpp` and `myActor.hpp`

myActor.cpp

```
#include "nsmb.hpp"
#include "myActor.hpp"

ncp_over(0x020c560c, 0) const ObjectInfo objectInfo = MyActor::objectInfo; //Stage Object ID 44 (use this in the editor)
ncp_over(0x02039a34) static constexpr const ActorProfile* profile = &MyActor::profile; //objectID 46

s32 MyActor::onCreate(){
	return true;
}

bool MyActor::loadResources() {
    return true;
}

s32 MyActor::onDestroy(){
	return true;
}

s32 MyActor::onRender(){
	return true;
```

```
}

bool MyActor::updateMain(){
	return true;
}
```

This file is where you program your actor.

At the beginning of the file is where you define:

- Where in the object bank table your actor's object info is stored. The array index is what you place in the editor to "place" your object. In the sample above, it is replacing ID 44. The beginning of this array is located at `0x020c529c`
- Where in the main process table your actor's profile is stored. The beginning of this array is located at `0x0203997c`

Here's an overview of what these functions are usually used for:

Function	Purpose
`onCreate()`	This function is ran when your actor is first created. This function is used for initalizing variables and anything that should be done once.
`loadResources()`	This function is called when the game is setting up your actor. [Todo]: Find the normal use case for this function.
`onDestroy()`	This function is called when your Actor is being removed from the stage. This function is used for cleaning up after the actor (i.e. freeing resources).
`onRender()`	This function is called every game tick while your actor is in view of the camera. This is typically used for graphics related code (for example, changing what texture an NSBTX is currently drawing).
`updateMain()`	This function is called every game tick while your actor is loaded. This function is used for the main functionality of your code (i.e. calculating positions, updating variables, etc).

> TODO: Expand this table with more functions you can override in a `StageEntity`

myActor.hpp

```
#pragma once

#include "nsmb.hpp"
```

```
class MyActor: public StageEntity {
public:
    virtual s32 onCreate() override;

    static bool loadResources();

    virtual bool updateMain() override;

    virtual s32 onRender() override;

    virtual s32 onDestroy() override;

    static constexpr u16 objectID = 46;

    static constexpr ObjectInfo objectInfo = {
        0, 0,
        0, 0,
        0, 0,
        0, 0,
        CollisionSwitch::None,
    };

    static constexpr u16 updatePriority = objectID;
    static constexpr u16 renderPriority = objectID;
    static constexpr ActorProfile profile = {&constructObject<MyActor>, updatePriority, renderPriority,
loadResources};
};
```

This file is where you store your instance variables and declare your functions.

Section 2 - State Machines

While not all Actors *need* to have a state machine, it can often times greatly improve the readability and reliability of your code (state machines are also a strategy Nintendo used when writing actors for the game).

Defining the State Machine

In your `.hpp`, you need to add the following:

```
class MyActor: public StageEntity {
    /* ... */

    // Functions for the state machine. Add more as needed.
    void exampleState();
    void anotherExampleState();

    void (*updateFunc)(MyActor*);
    s8 updateStep;

    void switchState(void (MyActor::*updateFunc)());

    /* ... */
}
```

In your `.cpp`, here's what you need:

```
s32 MyActor::onCreate(){
  /* ... */

  // Initalize the current state
  switchState(&MyActor::exampleState);

  /* ... */
}

bool MyActor::updateMain(){
  /* ... */

    // Make sure to call the update function every tick
  updateFunc(this);

  /* ... */
}

bool MyActor::exampleState(){
    if (updateStep == Func::Init) {
        updateStep++;
        return;
    }
```

```
    if (updateStep == Func::Exit) {
        return;
    }
}

bool MyActor::anotherExampleState(){
    if (updateStep == Func::Init) {
        updateStep++;
        return;
    }

    if (updateStep == Func::Exit) {
        return;
    }
}

void MyActor::switchState(void (MyActor::*updateFunc)()) {
    auto updateFuncRaw = ptmf_cast(updateFunc);

    if (this->updateFunc != updateFuncRaw) {
        if (this->updateFunc) {
            this->updateStep = Func::Exit;
            this->updateFunc(this);
        }

        this->updateFunc = updateFuncRaw;

        this->updateStep = Func::Init;
        this->updateFunc(this);
    }
}
```

Using the State Machine

While the code may look intimidating, state machines are very intuitive once you start working with them.

Function/Variable	Purpose

`updateStep`	This variable keeps track of what update step the current state is in. 1 (or `Func::Init`) is used for when a state function is entered for the first time. Useful for setting variables related to the current state. -1 (or `Func::Exit`) is used for when a state function is being exited. Used for cleaning up any state specific code before the code changes to the next state. All other update step values can be used inside a step to create "sub steps" via conditional code.
`switchState()`	This function will swap the current state to the function passed as a parameter. Call this function when you want to change what state you are in.

> TODO: Expand this page with more actor components (for example, colliders)

Revision #5
Created 19 February 2025 01:43:56 by Ndymario
Updated 14 April 2025 16:11:02 by keeper