Setting Up A Monster

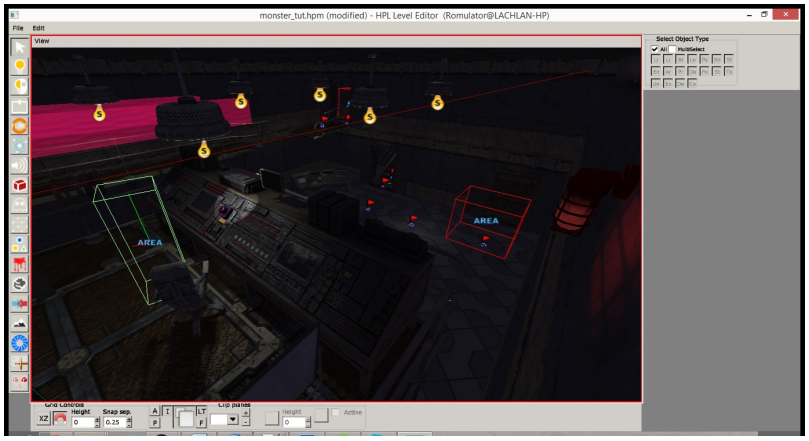
Monsters are your primary threat in SOMA. They are there to prompt the player to think on their toes and re-evaluate their situation to manage the situation with better strategy. As such, strategizing how a Player can or will react to the monster is the trick to creating an effective monster. Giving the Player the chance to hide, but also assuring the monster is a threat is the key to a perfect monster. In this tutorial, we will discuss how to place a monster and different scripts we can apply to make the monster threatening to the Player.

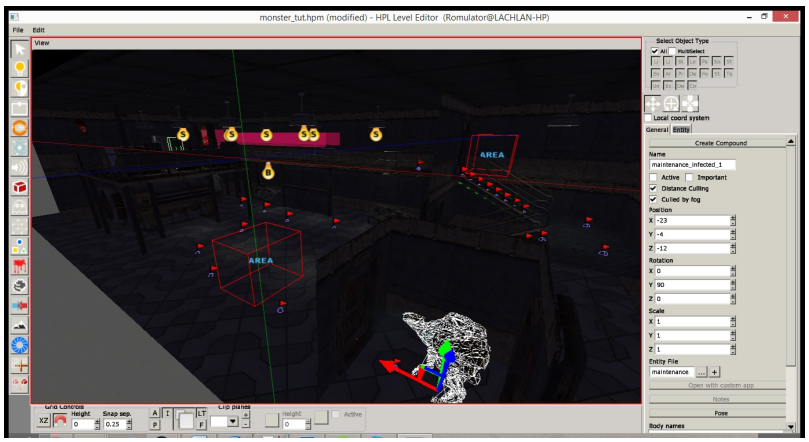
Which monster?

All the monsters in the game behave differently. Which one you use is purely dependant on you, but the “how will it act?”, why **this** monster and why **here** are the questions you should be asking. Since I'll be covering the basics, I will use the maintenance_robot, which if you are unfamiliar, is the first hostile enemy you can encounter in the game, located in Upsilon. It is the most basic of enemies to configure, encounter and set up. There are others which you can use, they may not be covered here, but if you can figure out something I cannot, please add it here!

Making the monster work

In order to start this tutorial, I needed a map and needed to lay out a few obstacles to help me get away from the monster. I decided to make a map like this, which has a nice basic layout for maintenance_robot_1 (the hostile robot from Upsilon):





The monster starts off as inactive - you can see this in the second image. I did not necessarily have an objective, but if I pressed a button, the power would turn off, and to turn it back on, I needed to go downstairs. Upon walking through a ScriptArea, the monster would spawn and enter the room.

Once it spawned, it followed a series of pathnodes – walking in a circular motion - then it spotted me, so I evaded it with amazing skills.

And it looped this particular route forever, since that was how I scripted it to behave. So how did I do it?

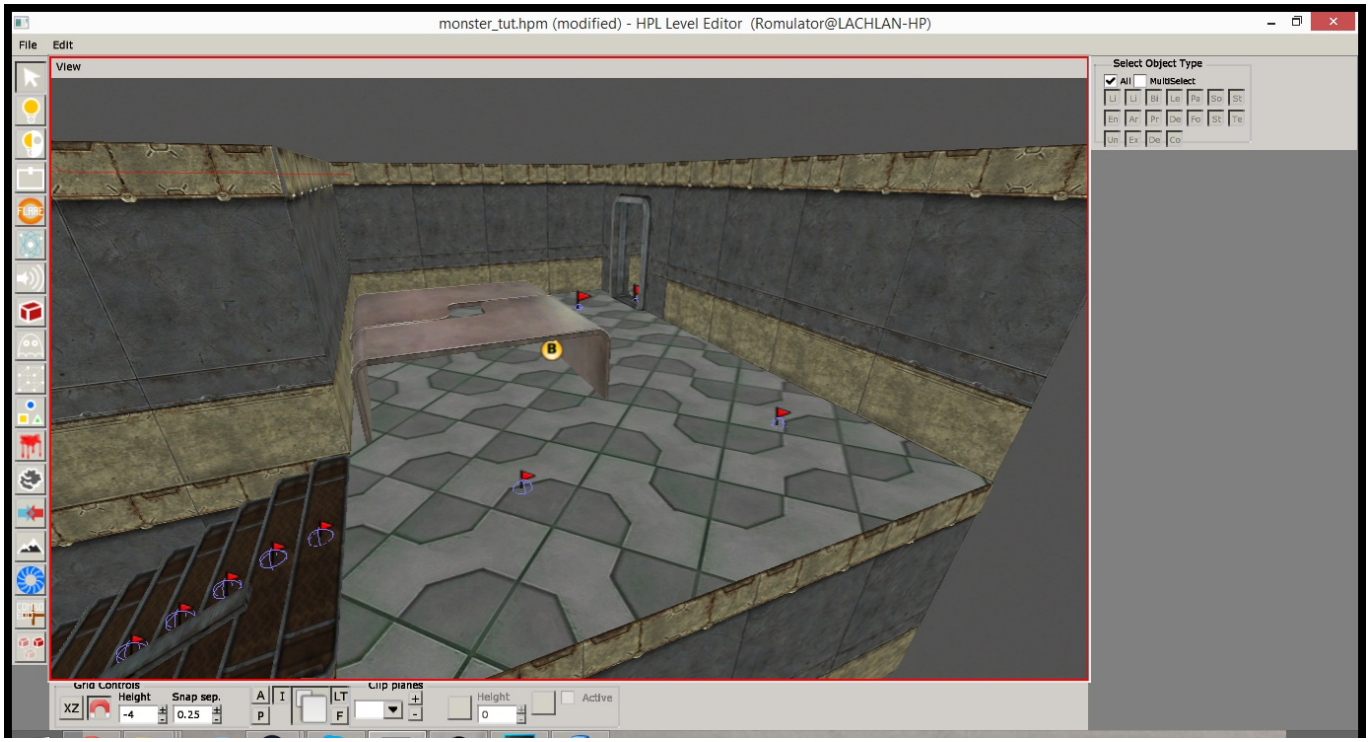
Pathnodes. What are they?

Pathnodes are a type of Area where you define the path that an actor can follow. Generally, pathnodes are applied to enemies to define where they can go, what patrol route they should follow and a few other technical things. A monster will usually follow the pathnodes to reach you if it cannot find a direct route to you (for example, if there is a wall between you and the monster) and will also follow such nodes when you have lead it into an area it is unfamiliar with.

The amount of pathnodes you place in your map is entirely up to you. However, your aim is to make sure the monster can go everywhere it needs to and not get stuck (I had this problem in my map). However, the rules of pathnodes are pretty much this:

- If an enemy needs to go into multiple rooms, there should be a pathnode in the doorway.
- If there is something blocking the path (like a table), you need to construct a path around it.
- If the enemy needs to go up or down stairs, there needs to be a pathnode on each step.

The following image has all three rules in it. Since the enemy may need some assistance getting around the table, we have a pathnode on the right side.



In Script

Before I begin, I recommend you go set up CodeLite. It has all the necessary syntax help you need if you set it up correctly. You can follow the official method here, or you can view acommunity member's tutorial which has pictures. The following has been written assuming you have some experience with HPL2 (Amnesia) custom story development. It's not required, but will help immensely here.

Part two of our enemy is to script how it moves. Without pathnodes to follow, the general enemy will despawn if it is not within the sight of the player (as they will when their assigned path has ended). This is how my script for this map looks like:

```
class cScrMap : iScrMap
	{
	void OnStart()
		{
		//Anything that should happen on Map Load.
		Entity_AddCollideCallback("Player", "opendoor", "Open_Door");
		}

	bool Open_Door(const tString &in asParent, const tString &in asChild,
int alState)
		{
		//Activate Enemy. Give it pathnodes. Starts walking. The door
opens so it can reach pathnode_2.
		Entity_SetActive("maintenance_infected_1", true);
		for (int i = 1; i &lt;12; i++)
			{
			Pathfinder_Track_Add("maintenance_infected_1",
```

```
"PathNodeArea_"+i, 0.0f, 0.5f, "", false);
                }
          Pathfinder_Track_Start("maintenance_infected_1", true, 1.0f, "");
          SlideDoor_SetClosed("slidedoor_theta_tunnels_2", false);
          return true;
          }
     //More Code Here...      }
```

Now, this may look complicated, but let's have a read of it step by step:

```
class cScrMap : iScrMap
     {
     //Code
     }
```

Now, I'm not entirely sure what this does, but my whole code errored without it. If you are familiar with HPL2, this is different, since you need to include this line of code at the beginning and pretty much the rest of your voids, bools, and other subroutines occur within the braces.

```
void OnStart()
    {
    //Anything that should happen on Map Load.
    Entity_AddCollideCallback("Player", "opendoor", "Open_Door");
    Entity_AddCollideCallback("maintenance_robot_1", "close_door",
"Close_Door");
    }
```

void OnStart() is an event where you can declare anything which you require to begin as you start the map. OnStart() only occurs in the first occurence of the map loading - therefore, if you backtrack to this level, the code here will not be called upon return. If you need such to occur, use OnEnter(). (Learn about Execution Flow here for more information).

Entity_AddCollideCallback(); is a callback. Adding a callback declares that you want a particular number of events to occur which you will define later on in your code. The syntax of Entity_AddCollideCallback(); is the following:

```
Entity_AddCollideCallback(const tString &in asParentName, const tString &in
asChildName, const tString &in asFunction);
```

const tString &in asParentName: The name of the parent entity as declared in the Level Editor which collides with the child. Usually the “moving” entity. Can be “Player” to mean the player.
const tString &in asChildName: The name of the child entity to collide with - usually not moving. In this case, I have made it a Trigger Area called opendoor.
const tString &in asFunction: The name of the function as named in the code. I called it “Open_Door”, since that is what my function will do.

Furthermore, because we have a Callback, we need to know the **Callback Syntax** in order for the engine to find the sequence of events to run through. Entity_AddCollideCallback's Callback Syntax is this

```
bool asFunction(const tString &in asParent, const tString &in asChild, int
```

```
alState)
	{

	}
```

Change “asFunction” to the name of your function as you declared above. In my case, it is called “Open_Door”. The rest of the line can stay the same.

Now let's see the sequence of events in the Open_Door callback routine.

```
bool Open_Door(const tString &in asParent, const tString &in asChild, int
alState)
	{
	//Activate Enemy. Give it pathnodes. Starts walking. The door opens so
it can reach pathnode_2.
	Entity_SetActive("maintenance_infected_1", true);
	for (int i = 1; i &lt;12; i++)
		{
		Pathfinder_Track_Add("maintenance_infected_1", "PathNodeArea_"+i,
0.0f, 0.5f, "", false);
		}
	Pathfinder_Track_Start("maintenance_infected_1", true, 1.0f, "");
	SlideDoor_SetClosed("slidedoor_theta_tunnels_2", false);
	return true;
	}
```

As you can see in my comment, the enemy is activated, assigned pathnodes, then instructed to walk. While this is happening, a door opens. Once again, let's go through each line.

```
Entity_SetActive(const tString &in asName, bool abActive);
```

const tString &in asName: The name of the entity to active or deactivate. If it is deactivated, it becomes invisible in game.
bool abActive: Whether the entity is visible or invisible. true = visible, false = invisible

```
for (int i = 1; i <12; i++)
	{
	Pathfinder_Track_Add("maintenance_infected_1", "PathNodeArea_"+i, 0.0f,
0.5f, "", false);
	}
Pathfinder_Track_Start("maintenance_infected_1", true, 1.0f, "");
```

This is a For-loop. Now this is a slightly more advanced form of code. To understand For loops, and loops in general, I recommend you check out these two links:

- "For" Loops
- Loops (A bit more advanced)

```
Pathfinder_Track_Add(const tString&in asEntityName, const tString&in
asNodeName, float afMinWaitTime, float afMaxWaitTime, const tString&in
```

```
asAnimName, bool abLoopAnim);
```

const tString&in asEntityName: The name of the entity to assign the pathnodes to.
const tString&in asNodeName: The name of the pathnode to add to the sequence of pathnodes available for the entity.
float afMinWaitTime: The minimum amount of time to wait at the pathnode before continuing.
float afMaxWaitTime: The maximum amount of time to wait at the pathnode before continuing.
const tString&in asAnimName: The name of the animation the entity will play upon reaching the pathnode.
bool abLoopAnim: If the animation should be looped each time the entity arrives at that particular pathnode.

```
Pathfinder_Track_Start(const tString &in asEntityName, bool abLoop, float
afUpdateFreq, const tString &in asEndOfTrackCallback);
```

const tString&in asEntityName: The entity in which will start patrolling once assigned pathnodes through Pathfinder_Track_Add();
bool abLoop: If, when the sequence of pathnodes has ended, the entity follows the same path again.
float afUpdateFreq: How often the path is recalculated. Default is 1.0f (recalculated every second). Should be enough.
const tString &in asEndOfTrackCallback: A callback to call once the track has ended. Not needed here, so it will not be explained further.

```
SlideDoor_SetClosed("slidedoor_theta_tunnels_2", false);
```

This quite simply opens the SlideDoor. I included it as part of my code to test if I could open doors with triggers, and so that the monster spawns behind a door, rather than in direct line of sight (immersion breaking).

We also keep the **return true;** line because the callback is of boolean type. When we collide with it, we return “true” to the collision in order to trigger it.

As you can see, it is a bit complicated. Scripting is not easy to start with, but when you begin to recognise the patterns, then you will get the hang of it. Never copy and paste code unless you understand what is actually happening, else you won't learn and will struggle with much more advanced forms of coding.

How can I learn more?

Well, for starters, you can explore my map if you would like! You can download it from this dropbox link.

Other ways of learning could be to go through SOMA's maps to see if you can figure out where and how they did a particular sequence of events. Most of the code above was done using this very method.

The other way is to experiment. Set up CodeLite and change the Code Completion to 1, so when you start typing, you see every keyword. See if you can recognise what particular keywords, functions, effects and other kinds of callbacks do, how you can set them up and make them work well!

But hey! If you have any questions, please, feel free to start a thread in SOMA's Development support forum.

From:
https://wiki.frictionalgames.com/ - **Frictional Game Wiki**

Permanent link:
https://wiki.frictionalgames.com/hpl3/community/other/monsters?rev=1444163600

Last update: **2015/10/06 21:33**