Entity Components

Overview

What follows here is an overview of all the different entity components that are present. Entity components are used to extend the functionality of any derived `iLuxEntity` class.

To add a module simply call the creation function in the SetupAfterLoad() call in the entity script file. It is often required to save a handle to the returned class as well.

CharMover

Creation function: `cLuxCharMover@ cLux_CreateEntityComponent_CharMover(iLuxEntity @apEntity, iCharacterBody @apCharBody)`

General

This is used to more easily move around a character body. It controls movement forward, rotation and very simple avoidance behaviors.

PathFinder

Creation function: `cLuxPathFinder@ cLux_CreateEntityComponent_PathFinder(iLuxEntity @apEntity)`

General

This is used to add pathfinding capabilities. It relies on Path nodes being placed in the map and usually works together with CharMover (but this is not needed).

StateMachine

Creation function: `cLuxPathFinder@ cLux_CreateEntityComponent_StateMachine(iLuxEntity @apEntity)`

General

This adds a state machine to the entity. It can be used for pretty much every thing but is mainly used for AI purposes.

It is important to note that there can only be ONE statemachine in a class!

To use the statemachine, you need to create states right after having created the module. It is very important that all states are always created in the same order for all entities using the same script file (basically meaning you should not dynamically create states during the update loop or similar). For instance:

```
@mpStateMachine = cLux_CreateEntityComponent_StateMachine(mBaseObj);
mpStateMachine.AddState("Idle", eState_Idle);
mpStateMachine.AddState("Move", eState_Move);
mpStateMachine.AddState("Stop", eState_Stop);
```

Every state has a couple of actions (methods really) that called in response to different events. These are:

Enter()
Called when the state is started.
Leave()
Called when the state is over. This is called before Enter of the new state!
Update(float afTimeStep)
Called every update.
SubStateOver(int alSubStateId)
Called when a sub state is over. More on substates below.
Message(int alMessageId)
This is any entity message that is intercepted. Can also be a custom message.
TimerUp(int alTimerId)
When a timer is over.

For each state, each of these functions get a name based on the syntax like this:
`void State_[StateName]_[Action]`
For instance the `Enter()` action in the state “Idle” is:

```
void State_Idle_Enter()
```

Note that you only have to define the actions that you want, so for instance a Stop state might only be something like:

```
void State_Stop_Enter() {
 //Stop the character
}
void State_Stop_Leave() {
 //Make the character move again
}
```

And skip Update, etc. (This is actually the best practice too!)

Sub State

A sub state is just like a state that runs along side the normal state. What makes it different is that they only last as long as the state that started it. Also sub states cannot change the normal state, only what the next substate will be. It can be sort of seen a smaller state machine inside a state. The sub states are created just like normal states but using the `AddSubState(tString, int)` method instead. Also note that substates can be shared between all of the states. They only belong to the a state in the sense that a sub state started in state X will only run while state X is active.

The function syntax for sub states is:
`void SubState_[StateName]_[Action]` For example:

```
void SubState_ThrowObject_Enter()
```

When a sub state is over, the state that started it has the action `SubStateOver(int alSubStateId)` called, where `alSubStateId` contains the id of the sub state.

Sub state as all the same actions as the normal state except for SubStateOver(int alSubStateId)

Another important feature of sub states is that there not has to be one set. So it is okay to do:

```
 mpStateMachine.ChangeSubState(-1)
```

which set no state at all as sub state. This is actually the default setting when ever a new normal state is started.

Default State

The default state is a state that can be used by both normal states and sub states. To add it just write a state function with the name “Default” instead of a state name. (This means a state cannot be named “Default”.) Then this function will be called if the current state does not contain it. So for instance if you have something like:

```
void State_Default_Update(float afTimeStep){
...
}

void State_State1_Update(float afTimeStep){
...
}
```

If the state machine is currently in `State1` then the `Update` for `State1` will be called. However, if the current state is `State2` and it does not have an implemention of `Update`, the `Default` version of `Update` will be called.

There is a special case for `State_Default_Message`. This is also called if the state specific function returns false.

Timers

Timers are used to check if a certain amount of time has passed. They are only valid for the state that started it, so if a timer is active when state changes, then it becomes removed. Also, state and sub state do not share timers. So if sub state “ThrowObject” starts a timer, then `TimerUp` will only be called in the sub state, and not the normal state. Timers are started with `StartTimer(int alId, float afTime)` in the state machine and can be stopped with `StopTimer(int alId)`.

SoundListener

Creation function: `cLuxSoundListener @ cLux_CreateEntityComponent_SoundListener (iLuxEntity @apEntity)`

General

This simply makes the entity into a sound listener and it will now receive the event message eLuxEntityMessage_SoundHeard whenever a sound of a specific type (specfied in “sounds/ai_reaction_sounds.dat”).

HeadTracker

Creation function: `cLuxHeadTracker @ cLux_CreateEntityComponent_HeadTracker(iLuxEntity @apEntity)`

General

This will make the entitie's head follow a certain target. What is the head is set b setting up the bones that makes up the neck and how much influence each should have. This is done using Setup(...) or by using the variables in the .ent file (use LoadFromVariables from the script to load the variables).

ForceEmitter

Creation function: `cLuxForceEmitter @ cLux_CreateEntityComponent_ForceEmitter(iLuxEntity @apEntity)`

General

This will give the entity a force field (cForceField) which makes certain things in the game world (such as undergrowth) animate and react to its presence . If MaxForceSpeed is higher than 0 it will also vary

in strength depending on the speed of the MainBody (or character if set).

BarkMachine

Creation function: `cLuxBarkMachine @ cLux_CreateEntityComponent_BarkMachine (iLuxEntity @apEntity)`

General

This creates a component that can have one or more states with random sounds or voice quips (barks). It is meant to be used with entities that require different sounds/voices to be randomly played depending on which state it is in.

BackboneTail

Creation function: `cLuxBackboneTail @ cLux_CreateEntityComponent_BackboneTail (iLuxEntity @apEntity)`

General

When this is added, the backbone of the entity (as defined in the ent file) will be bent as the entity moves along. It is meant to be used by things like fishes, etc to get more natural like movement, but can of course be used for whatever. Note that all setup should be done in the ent file, and have the component load the type variables.