

Creating a new command

Adding a new command to the game generally requires two steps: add the command to `CommandParser` and implement its functionality in one of the events modules. Typically you would also want to add information about its syntax and maybe even a help file. Note that to add an emote you would follow this same procedure, because emotes are just commands right now.

Let's say you want to add a *jump* command which can cause a character to jump over something.

CommandParser

At the top of the `CommandParser` module there are several class variables pointing to sets of commands. This is to make lookup slightly faster. Each set is typically a bunch of related commands so they can be separated into different methods for parsing. Add your new command to the appropriate set.

For our example, *jump* would be a movement command so we'll add it to the `@movement` variable like so:

```
@movement = Set.new(['go',
  'east',
  'e',
  :
  'pose',
  'jump'    #Here we've added it
])
```

Next we need to add the parsing information for the command and convert it into an event. Parsing of commands is split into different functions for efficiency, but they are all basically the same. They take in the input command and either return an `Event` object or `nil` if the command does not parse correctly. Continuing our *jump* example, we'll head down to `parse_movement` method and add in our new *jump* command which looks like `JUMP OVER [OBJECT]`.

```
def parse_movement(input)
  event = Event.new(:Movement, :action => :move)
  case input
  when /^go\s+(.*)$/i
    event[:direction] = $1.downcase
  when /^jump\s+on\s+(.*)$/i    #Here's where we've add it
    event[:action] = :jump
    event[:object] = $1
```

Now we know what our *jump* event will look like. It basically just contains the object we wish to jump over. Note that our regex is case insensitive and allows any number of spaces between words. This is a good practice to make commands a little more forgiving.

Event Module

When we specify the type of an event and the `:action`, we have told the EventHandler which module and method to use to run that event. In our example, the module is Movement and the method which will be called is `jump`. All methods which handle events take the same set of parameters: the event, the player, and the room the player is in. Actually, player and room are just for convenience, because you almost always need to have those variables.

The methods tend to have a similar structure as well. The first step is generally to check if the object of the event actually exists. If it does not, output a message to the player and return from the method. Check any other parameters of the event which need to be correct. If everything checks out, make the necessary changes. The final step is almost always to then output the event to the room so that other objects can react to it.

Let's go through a simplified version of our *jump* example:

```
def jump(event, player, room)
  #Find the object
  object = room.find(event[:object])

  #Check the object
  if object.nil?
    player.output "What are you trying to jump over?"
    return
  elsif not object.jumpable?
    player.output "You cannot jump over #{object.name}."
    return
  end

  #Setup the output
  event[:target] = object
  event[:to_player] = "You jump over #{object.name}."
  event[:to_target] = "#{player.name} jumps over you."
  event[:to_other] = "#{player.name} jumps over #{object.name}."

  #Send out the event, which will show the above messages
  #to the proper people.
  room.out_event event
end
```

First, we check if there is an object in the room matching the input. If there isn't, we let the player know and return. Then we check if the object can be jumped over. If it cannot, we let the player know and return. Then we modify the original event to include output strings. Note that we also set the target to be the object the action targeted. This will make sure the correct output goes to the correct place, but also allows other objects to react appropriately.

At the very end, we send the event to the room, where everything will get taken care of properly. Of course, in this case, we have done little but create an emote. Events can often be much more complicated.

Syntax

If you would like to add a helpful syntax reminder for a command, head over to the Syntax module. This basically contains a big hash table called Reference which points from commands to the command syntax. Add your command in anywhere.

Here's the *jump* example:

```
Reference = {
"acexit",
"ACEXIT [DIRECTION] [EXIT_ROOM]",
:
"go",
"Go where?",
"jump",                #Here is our simple addition
"JUMP OVER [OBJECT]",
"lock",
"What would you like to lock?",
:
}
```

Loading your changes

If you were making all these changes which the server was running, you can have them dynamically loaded using the ARELOAD command:

```
|> areload components/commandparser
Reloaded components/commandparser: true
|> areload events/movement
Reloaded events/movement: true
|> areload help/syntax
Reloaded help/syntax: true
```

If everything works out, then you have added a brand new command to the game!