How I did my Rogue game

The purpose of this is to give some insight into how I wrote my Rogue game, and to give some hints to those of you unsure about what you are doing. This is not the only way to write this game, and it is not required that you write it this way. For many of you some parts of this will be too late to make your life easier, that's ok – you learned more by coming up with a solution on your own, perhaps with some questions for the TAs, and the point of this entire exercise is to learn.

Parsing the XML file

XML files have start and end elements. A start element has the form <ElementName *args*> and an end element has the form </ElementName>. The *args* are optional and not all start elements have them.

When a start element is encountered, the startElement function in your parser class is called. Various functions, demonstrated in the example code, can be used to determine the ElementName string and to get the various *args*. The example also shows how to retrieve information that is available when the end element for some element is reached, indicated by endElement being called by the parser.

XML elements in our file are logically of two types: those that create objects, and those that define fields of an object. Room is an example of an element that creates an object, and posX is an example of an element that defines a field of an object. Objects created in our dungeon are in six categories:

The dungeon holds global properties of the dungeon, such as the playing area and rooms in the dungeon.

Rooms hold information about the room (its position and size) and things (Items and Creatures) that are contained in the room. Items and Creatures that are defined in the room are associated with the Room, i.e., the room has containers that holds all of the Creatures and Items that are in the room.

Passage ways connect rooms. In our dungeon, passage ways never hold things, but there is no reason why they couldn't.

Creatures, like rooms, hold information about themselves (hit points, for example), Items the creature has, and actions associated with the creature. A list holds all of the items a creature owns. As well, an attribute holds the weapon and sword, if any, that the creature wears or wields. Actions associated with creatures are divided into categories:

1. actions that occur with death, and
2. actions that occur when hit.

These are stored in two different containers, one involving hit actions and the other death actions. The Player is a special creature that we can move.

Items (such as armor, scrolls and swords) have various properties defined and contain ItemActions. References to ItemActions associated with an item are kept in a container that is a field or attribute of the Item object. (We'll use reference to mean a handle for an object, not necessarily a C++ or Java reference – it can be a reference, a pointer, or an actual object, depending on the language and circumstances).

Actions will always belong to either the item or creature being parsed, and will always be defined between the start and end element of an Item or a Creature. If it is a CreatureAction it will belong to the creature being parsed, and the creature reference should be non-null. If it is an ItemAction it will belong to the Item being parsed, and the item reference should be non-null.

How to associate an object with its owner.

When we encounter the start element of an object that can contain other objects (a dungeon, room, creature or Item) we should create that object, and have a field in our parser class that references, or points to, that object

When we encounter an object that belongs to a dungeon, a room, a creature or an item we should check which pointer to a dungeon, room, creature or item is non-null. Because rooms are nested within dungeons, and creatures are nested within rooms and items are nested within creatures or rooms, and actions are nested within either an item or a creature, we can examine our references to determine what is being parsed.

Actions will always belong to either the item or creature being parsed. If it is a CreatureAction it will belong to the creature being parsed, and the creature reference should be non-null. If it is an ItemAction it will belong to the Item being parsed, and the item reference should be non-null. The action can be added to the Item or Creature by calling a setter on the appropriate object.

Creatures only belong to rooms, so if a creature is found then the room reference should be non-null and the creature can be added to the room.

Items can belong to either rooms or creatures. Since creatures are nested within rooms, if the creature reference is non-null then we are parsing a creature and the item belongs to the creature the creature reference points to. If the creature reference is null, then the room reference should be non-null, and the item belongs to the room, and should be added to the room.

Rooms belong to the dungeon, and are added to the dungeon, using the reference to a dungeon.

When the end element for the room, creature, item or action is reached, its reference is set to null. This is not done for the dungeon reference because it contains references for the rooms, which in turn contain everything else, and so the dungeon reference is the root of a tree that defines the parsed objects.

Displaying our dungeon and its contents

There are two major data structures used in the display.

The first is what I'll call the physical display, and is what is observed by someone playing the game. For C++ this is done using ncurses, and for Java this is done using asciiPanel. I'll call this the *terminal* in the following discussion. The terminal receives and displays ascii characters that are passed to it by the *objectGrid* in a window on a physical screen.

The second is a two dimensional of stacks of type Displayable which I will call an objectGrid. Note that anything that can be displayed inherits from Displayable. The objectGrid also has a top and bottom area for displaying various information about the game. Rooms floors and walls, passage ways and passage junctions, creatures and items are all displayable. As well, I used a Char class that extends Displayable

whose purpose is to hold a character to be displayed, and is used for putting messages into the message areas of the objectGrid.

The overall size of the objectGrid and the terminal are the same.

We use an array of stacks, not an array of Displayable references (same usage of reference as in parsing) because a position of the dungeon may have several items. Thus, a point in a room could have the room floor, one or more items, and a Creature standing on the item(s). The topmost (last in to the stack) thing in the stack is what is displayed.

When a thing is placed in the objectGrid, the terminal is updated with the character representing the thing. When a thing moves on the objectGrid, the terminal is updated with the current top character on the position the thing moved from, and the terminal is updated with the character for the thing that moved in its new position. Note that in our dungeon the only thing that moves is the Player, but Items can be picked up by the Player.

There are classes that derive from Displayable to represent room walls, floors, passage ways and a Char that is used for messages and holds a character to be displayed. I also use a container of non-traversable locations that holds all of the walls and empty spaces in the dungeon, making it easy to check for whether or not a location can be moved across. You could also check the type of the thing at the point being moved to to determine this.

Combat, movement and death

When the player attempts to move into a square occupied by a monster, the player will hit the monster. That the player is attempting to move into a square occupied by the monster can be determined by looking at what is at the top of the objectGrid stack at the location the player is trying to move to. If it is a creature, then the move will cause the player to try to hit the creature.

The damage done to the creature will be a random integer between 0 and the maximum hits that the player can inflict, inclusive, added to the value of a sword that is *wielded*. Swords in the pack but that are not wielded do not affect the damage done (see the commands below to see how a sword is wielded, and how I represent that). There is a public function associated with all creatures that can be called by an attacker to indicate to the creature object that it has been hit, and how many hit points of damage there are, and the location of the attacker. When a monster is attacked, it will strike the player, with hit points again being a random integer between 0 and the maximum hits that the player can inflict, inclusive.

When a victim creature, either the player or a monster, is hit, all hitActions associated with the victim creature are performed by that creature. This is done by traversing the container of hitActions and performing each one.

When a creature, either the player or a monster, dies, all deathActions associated with the creature are performed by the dying creature. When the player dies the game is over. The screen should be displayed as it is after the player's deathActions are performed.

Player movement serves two purposes in the game. Movement obviously has the effect of either moving the player or causing the player to hit a monster. Movement also serves as a measure of time. Every so many moves the player gains one hit point added to its health. The number of moves that does

this is the hpMoves element in the .xml file description of the player. The Hallucinate action also lasts for a given number of moves that is given by the actionIntValue element in the .xml file. So it is necessary for movement to be communicated to both the object that controls player movement (the Player object, in my game) and to the Hallucinate action. In my game I use the Observer pattern to do this, with the Observers being the Hallucinate action and Player, and the Subject being the object getting character input, which is the objectGrid object in my game.

Commands

The following commands are implemented in my game:

Change, or take off armor 'c': armor that is being worn is taken off and placed it in the pack. I have a reference in Creature that points to the armor being worn, and this reference is copied into the pack container and set to null. If no armor is being worn a message should be shown in the *info* area of the game display.

Drop 'd' <integer>: drop item <integer> from the pack, where <integer>-1 is an index into the pack container. If the index does not refer to an item in the pack an informational message is printed on the game display in the *info* area. If the item is dropped it should no longer be shown as in the pack until it is picked up again. If the item dropped is a wielded sword or worn armor, the sword or armor is no longer wielded or worn. The dropped item should be beneath the player, and the player should still be displayed after dropping the item. When the player moves from the location where the item was dropped the dropped item will be displayed.

End game 'E' <Y | y>: end the game. I print a message in the info area using Char objects added to the objectGrid to make it clear why the game ended.

Help: '?': show the different commands in the *info* section of the display. This is the bottom message display area. Char objects are added to the objectGrid to display the message.

Help 'H' <command>: give more detailed information about the specified command in the *info* section of the display. Char objects are added to the objectGrid to display the message.

Show or display the inventory 'i': show the contents of the pack, printing the *name* for each item in the pack. For swords and armor print out the value of the sword or armor. For scrolls print out what the scroll name Each item is preceded by an integer 1 … *max items in the pack* that is used to index into the pack container when removing or dropping items in the pack. If an item in the inventory is a sword that is wielded, put "(w)" after the name of the sword. If the item in the inventory is armor that is worn, put "(a)" after the name of the armor.

Pick up an item from the dungeon floor 'p': pick up the visible item on the dungeon floor location that the player is standing on and add it to the pack container. Room and passage way floors cannot be picked up. If multiple items are in the location, only the top item is picked up.

Read an item 'r' <integer>: the item specified by the <integer>-1 must be a scroll that is in the pack. Reading a scroll causes the ItemActions associated with the scroll to be performed. The scroll is removed from the pack when it is read, and basically vanishes from the game.

Take out a weapon 'T' <integer>: take the sword identified by <integer>-1 in the pack container and have the player wield it. I do this by having a reference in my Player that shows the weapon being wielded. If the identified item is not a sword, or no such item exists, show a message in the *info* area of the game display. The sword remains in the pack when wielded – this makes it easy to drop sword that is wielded. If we no longer want the player to wield the sword it needs to be dropped and picked up.

Wear item 'w' <integer>: take the armor identified by <integer> from the pack container and make it the armor being worn. In my game, there is a reference to Armor that indicates if armor is being worn, and is null if no armor is being worn. If the identified item is not armor, or no such item exists, show a message in the *info* area of the game display. Armor is taken off with the 'c' command, described above.

Creature Actions

Creatures can have the following actions associated with them. All actions implement an abstract performAction( ) function that allows them be invoked without worrying about the specific kind of action.

Print: this is mentioned in the RogueProjectInstructions.pdf file but we will not be implementing it.

ChangeDisplayType: changes the character that represents the creature. Used primarily in a death action for the Player to change what a dead player looks like.

Remove: removes the creature from the display. Primarily used to remove a dead monster from the objectGrid and the terminal.

Teleport: causes the creature to go to a random place in the dungeon that is legal for a creature to be, i.e., within a room or a passageway between rooms. A creature should not end up outside of the traversable part of the dungeon. I use a while loop to randomly generate coordinates until a valid coordinate is found. A more efficient, but harder to program technique might be to pick a room or passage way at random, and then pick a location within the room or passage way at random.

UpdateDisplay: causes the creature to update the display of itself. I do this by essentially removing the creature from its position in the objectGrid and then putting it back into that position. The objectGrid then updates the terminal.

If the creature is the player, the hitpoints displayed in the top message area should also be refreshed.

YouWin: This is typically executed by the creature that is killed. We will print out the message given in the actionMessage element of the XML definition of the action.

Players have the following additional actions:

DropPack: drop a item in position 0 of the pack container. The action of dropping is the same as the drop command above, under Commands, and in my implementation this just calls the drop( ) command.

EmptyPack: the same as DropPack, except items are dropped repeatedly until the pack is empty.

EndGame: typically executed when the player dies, it causes the game to be ended and all further input to be ignored. I do this by setting a static flag in the Dungeon that signals the end of the game.

ItemActions

Bless/curse item: This action is associated with scroll. It reduces the effectiveness of a sword or armor that is being worn or wielded. A sword or weapon in the pack is not affected. The actionCharValue in the XML definition of the action will be 'a' if it affects the armor being worn, and a 'w' if it affects the sword, or weapon being wielded. If the item to be affected is not being worn or wielded, the scroll has no effect. Print a message to the bottom message area indicating that the scroll was blessed or cursed. I print the message:

scroll of cursing does nothing because " + name + " not being used when the object is not worn or wielded, and *name + " cursed! " + intValue + " taken from its effectiveness*, where *name* is the name of the item.

Hallucinate: it registers with the observer and when invoked causes each item within the dungeon to display a different character with each move of the player. Displayed characters should be among the set of characters used to represent room floors, room walls, passage ways, passage junctions, items and creatures. This lasts for a limited number of moves specified in the actionIntValue element for the action in the .xml file.

Hallucinate implements the Observer interface of the Observer pattern, and registers with the objectGrid, which receives characters, in my dungeon. It is notified of each input command, and the hallucinate action keeps track of the number of moves so that hallucination stops after the specified number of moves.

When it is invoked it will print out a message in the info area of the game display that hallucinations will continue for some number of moves.

I implemented this by having a static flag set in my Displayable base class that says that hallucinations are occurring. Displayable has a getChar( ) function that returns the character for the displayable – when the hallucinate flag is set, the getChar( ) returns a random character, and otherwise returns the normal character. When the specified number of moves have occurred, the Hallucinate action resets the flag to end the hallucinations.