>Remember

By Hugo Labrande
Issue #10 : Saving and checkpointing

If you write a game using an engine such as the Z-Machine, the Quill, or PAWS, you usually don't have to think about saving and loading/restoring, as these engines typically handle such functionalities. But as it turns out, there are several tricky issues related to saving and restoring, which sometimes intersect with technical limitations and game design. So I thought it would be an interesting topic for this month's issue!

And, as usual, I'll start with what I know best, which is the Z-Machine - but don't worry, we'll broaden the scope to tape-based games too!

How the Z-Machine handles save games

Here is a high-level explanation of how Infocom’s Z-Machine handles saved games. The Z-Machine has several memory sections: static memory and dynamic memory. Static memory typically contains immutable things such as the game's strings and code; dynamic memory contains things like variables and the current object tree. The Z-Machine interpreter also has a stack (with temporary variables, function calls, etc.) and the Program Counter (pointing to the next instruction that needs to be executed). When you save a game, what it does is essentially save the dynamic memory, the stack, and the Program Counter to a file; when you restore, all of these are imported to populate the interpreter’s memory. It kind of is the equivalent of "save states" in an emulator, where the state of the stack, registers, etc, is saved, and restoring is made by initializing all the memory zones with the saved data, then letting emulation take it from there.

This is conceptually simple (you are emulating a machine and cryo-freezing the state of the machine) and provides a general solution that works across every game; but it also creates some problems. The main problem is that the saves are tied to a particular game file. Well, you could say, that's ok, I don't expect my Arkanoid save to work for my Krakout disk image, right? Of course, but it's actually worse than that: if you fix a bug or a typo in a game, this creates a different game file (it can change the internal addresses of strings, for instance), and all previous saves need to be discarded. This is a problem when writing and testing your own game, but also a problem for players in this day and age: most games nowadays, especially ones distributed on digital platforms, are routinely and silently patched, without destroying the player's saves. The market has changed, and the player's expectations too, which means you likely need to implement a different saving system. (And unfortunately, this cannot be done reliably in Z-Machine games; you might be able to figure something out on some machines, but not all interpreters will support it, especially not Infocom's, as this was all added to the Z-Machine standard later.)

But there's good reasons, in general, in an adventure game, why saving across several versions of a game is a complicated problem. If you're fixing a typo, and otherwise not affecting the puzzle structure, there shouldn't be any problem. (But even if you don’t change the puzzle structure, there could be issues: what if you change the score associated with a sub-goal? What if you add "achievements"? What if you add new "flags" tracking if a player has done something or not? In all of these situations, if the player saved past the part you have changed, there will be some inconsistencies or missed content.) But if you're fixing a bug, like the player was not supposed to have grabbed a particular object so easily, how do you reconcile this with saves where the player has that object? Changes and fixes to the game can make some player states incoherent with the new game, which is a very tricky issue to solve and probably can only be solved on a case-by-case basis.

If you want to dig deeper into that, the following links are interesting:
  https://intfiction.org/t/savegame-backwards-compatibility/8914
  https://eblong.com/zarf/glk/save-files-break.html

What about checkpointing?

Another thing I havent mentioned in the previous part is that the fact that a save game is game-specific means you can't carry saves over to a new game, or make a two-part game where the save is used to initialise the second part. Carrying a save between a game and its sequel hasn't been done very often (a recent example that comes to mind is *Mass Effect* and its sequel *Mass Effect 2*), but it is a question that comes back from time to time on forums. But there are quite a few two-part tape-based text adventures from the 1980s and 1990s that actually implemented that: at the end of part 1, please insert a tape so your data can be saved, then your data (including your score so far) will be imported at the beginning of part 2. Examples of this are *Dragon Slayer* by Martin Freemantle, *The Fisher King* by Dennis Francombe, and *Jester's Jaunt* by June Rowe and Payl Cardin (where the game actually lets you look around for a while before loading your tape data), and many more. Let’s look at the PAWS manual (more precisely, the Notebook) on how this works:
  https://www.mocagh.org/miscgame/paws-alt-notebook.pdf
It’s on page 23, and the mechanism is explained: the flags are carried between both games; you need the same number of locations in both parts, and the same number of objects, and the objects that can be carried forward between parts need to have the same description. If you abide by these rules, when arriving at the final location of part 1, the player is prompted to save their position, then part 2 needs to be loaded, and when this is done the player character is transported to the new location 1. This system is still a bit rigid, but here you see an advantage of the PAWS’s database format (where objects and flags and such are stored in databases) versus Infocom’s Z-Machine format (with addresses, a stack, etc.): the latter is more flexible, but it means it is hard to find a common structure between two games. (Hard, but maybe not impossible – but Infocom never looked into it, presumably because the Z-Machine essentially requires disks and never ran on tapes.) In any case, even if this solution was doable in PAWS, it sounds like saving your position to tape was seen as a time-consuming inconvenience by players...

An alternative that was popular at the time for multi-part games was passwords: at the end of part 1, you get a password, and you need to enter it to unlock part 2. There were a lot of Spectrum games using this system (from Gareth Pitchford's *Twilight Inventory*: *Diarmid*, *Dreamare*, *The Black Knight*, etc. - but there were dozens more), and the system was also used more recently in Davide Bucci's games like *The Queen's Footsteps*. (How to set it up in PAWS is explained on page 22 of the PAWS Notebook linked above.) This requires, however, to design the game a certain way, and have "bottleneck points": points in the scenario where the story must always be the same, and the inventory is the same, so a password can in essence teleport you to that immutable point of the story. (This often implies giving a justification for why your inventory is changed or pared down to a few specific objects: a different day, player has been captured and jailed, etc.) This is a good design compromise, since most games follow the model of a linear story with defined story beats (like most point-and-click adventure games too, really), but wouldn't work for other games, such as Infocom's *Deadline*, which cannot be broken down in several parts. Note that such a multi-part design also helps with the update problem mentioned above: a message like "sorry, the game has been updated since you last played, you have to restart to the beginning of the section you were in" might be a good compromise for modern players.

I wanted to finish this article by mentioning a hybrid system, which to my knowledge doesn’t seem to have been implemented in very many text adventures, but was actually very common in other games. The hybrid system would be of a game with several parts that are password-gated, but with several passwords possible, each giving a different starting situation. An example of this was *The Black Tower*, by Diane Rice, where the password given at the end of part 1 could be different depending on the objects you are carrying at the time; also *Nether Regions*, by Gareth Pitchford, and possibly more. This system requires a system to generate passwords from the data, and maybe add a bit of obfuscation so players cannot guess the passwords; it could be as simple as chaining conditional statements to determine which state the player is in, and give a custom password in each of these cases. But note that with the right encoding, any information can be carried in these kinds of passwords: flags (like ones indicating whether an object is in inventory or whether a task has been performed by the player), but also the number of turns, the score, the number of coins, health and mana points, etc. Such password schemes were actually extremely common in early NES games, where no battery saves were included in the cartridge; and it's actually a very interesting topic if you enjoy encoding, reverse-engineering, and obfuscation. As a good example, the following video explains how passwords are generated in Castlevania II, and hold information from the items unlocked to the number of garlics carried:
https://www.youtube.com/watch?v=_3ve0YEQEMw

I think this basically concludes this month's article! We've seen a few models involved in saving player progress in a text adventures, and the implications of each of these. I hope you found the article interesting! And if you know of any more saving schemes, or examples of games that I would have missed, don't hesitate to let me know!