Presented at Born Digital and Cultural Heritage Conference, ACMI, Melbourne, Australia, June 19-20 2014.

There and Back Again: A Case History of Writing "The Hobbit "

Abstract

In 1981, two Melbourne University students were hired part-time to write a text adventure game. The result was the game "The Hobbit", based on Tolkien's book, which became one of the most successful text adventure games ever. "The Hobbit" was innovative in its use of non-deterministic gameplay, a full-sentence parser, the addition of graphics to a text adventure game and finally "emergent characters" – characters exhibiting apparent intelligence arising out of simple behaviours and actions – with whom the player had to interact in order to "solve" some of the game's puzzles.

This paper is a case history of the development of "The Hobbit." Little has been written about the development of the first generation of text-based computer games; this case history provides insight into this developmental period in computer game history. As co-developer, I offer my recollections of the development process, the internal design, and the genesis of the ideas that made "The Hobbit" unique. I compare the development environment and the resulting game to the state-of-the-art in text adventure games of the time. Lastly, I discuss the legacy and recent revival of interest in the game.

"Let us not follow where the path may lead.

Let us go instead where there is no path,

And leave a trail."

Japanese Proverb

The Tenor of the Times

It was early 1981. I was a Bachelor of Science student at Melbourne University, majoring in Computer Science (CS) and just starting my last year. These were the early days of Computer Science education, and the curricula required today for undergraduate Computer Science students had not yet been developed. In our classes we were studying topics like sort algorithms and data structures and operating systems such as BSD Unix. Another class focused on calculating rounding and truncation errors occurring as a result of a series of digital calculations. We were taught software development using a systems analysis method called HIPO[1] – Hierarchical Input-Process-Output, the best practice in structured programming – and that documenting our code was a good practice. Object-oriented programming was still in the future.

During our first couple of years in the CS program, programming projects were written using "mark sense cards", which we marked up with pencils and fed into card readers after waiting in a long queue of students – sometimes for an hour or two to get a single run. You had to get the program running within a certain number of runs or the card reader would redistribute the lead across the cards, making them illegible.

By the time we reached the last year of the Bachelor's degree, in our CS classes we were actually allowed to log onto a Unix machine in the lab and work there, if we could get access to a terminal (which often meant waiting for hours, or booking a timeslot, or waiting till late in the evening). We programmed in Pascal, Fortran, Assembler, C (our favorite), and Lisp. Our favorite editor was, universally, Vi. I remember programming a PDP8 in Assembler to run a toy train around a set of tracks, switching the tracks as instructed; we hand-assembled the program, typed it in and debugged it using a hexadecimal keypad.

By this time I'd built my own PC, from a project in an electronics hobbyist magazine. I'd purchased the mother board, which came as a peg-board with a printed circuit on it, minus any components or cross-wiring. I would go to the electronics parts store with my list of chips, resistors, capacitors and diodes, and solder for my soldering iron. In the store they'd say, "tell your boyfriend we don't have these" – it was not even considered possible that I might be

[1] https://en.wikipedia.org/wiki/HIPO

the person purchasing them. The system had a small number of bytes – around 128 bytes, I believe (that is not a misprint) – of free memory, and used a black and white TV as a monitor. You wrote programs in a simple Assembler, hand-assembled it and typed it in using a hexadecimal keypad. There was no save function, so whenever the system restarted you had to re-type in the program. It was actually impressive to see the programs you could develop in that amount of space.

I was used to being one of around 2-4 women in my university classes, whether it was a smaller class of 30 students or one of the massive Physics classes holding perhaps two or three hundred. Sexism was alive and kicking. The norm for women – for most of the fellow students at my all-girl high school, MacRobertson – was to become secretaries or nurses (although my closest friend for many of those years became a lawyer, traveling to the 'Stans to negotiate for oil companies, and is now chairman of the board). One fellow student (luckily, I don't remember who) gave me the ultimate compliment: "you're bright, for a girl!" In self-defense, I partnered with another woman – Kerryn – for any pair projects. Whenever we had 4-person group projects we joined with another frequent pair, Phil Mitchell and Ray, who were amongst the few men willing to partner with us; these group experiences later led to me recruiting the other three to work at Melbourne House.

My game-playing experience was very limited. There was a Space Invaders arcade game in the lobby of the student union at the university that I sometimes played. For a while there was a game of Pong there, too. The Unix system featured an adventure game we called "Adventure" – "Colossal Cave", also often referred to as "Classic Adventure". In our last year I played it obsessively for some time, mapping out the "maze of twisty little passages", until I had made it to through the game once. At that point it instantly lost interest for me, and I don't believe I ever played it again. I was not aware of any other computer games.

State-of-the-art PC games were a very new thing – PC's were a very new thing – and at the time were written in Interpretive Basic by hobbyists. Sometimes the games were printed in magazines, taking maybe a page or two at most, and you could type them into any computer that had a Basic interpreter and play them. The code was generally written as a long list of if-then-else statements, and every action and the words to invoke that action was hard-coded. The game-play was pre-determined and static. Even if you purchased the game and loaded it (from the radio-cassette that it was shipped on), you could generally solve the puzzles by reading the code. The rare games that were shipped as compiled Basic could still be solved by dumping memory and reading the messages from the dump.

Getting the Job

I was working early Sunday mornings as a part-time computer operator, but wanted a job with more flexibility. On a notice board I found a small advertisement looking for students to do some programming, and called. I met Alfred (Fred) Milgrom, who had recently started a company he called "Melbourne House", and he hired me on the spot to write a game for him. Fred was a bit of a visionary in thinking that hiring students with Computer Science background could perhaps do a better job than the general state-of-the-art of self-taught hobbyists.

Fred's specifications to me were: "Write the best adventure game ever." Period.

I told Phil Mitchell about the job, as I thought he had the right skills. I brought him along to talk to Fred, who hired him to work on the game with me. Kerryn and Ray joined us at Melbourne House later that year to write short games in Basic for publication in the books. These books featured a series of games, most of them about a page or two in length. The books were often sold along with a radio-cassette from which you could load the game rather than having to type it in yourself. Ray only stayed briefly, but Kerryn I think stayed for most of the year, and wrote many games. She'd sit at the keyboard and chuckle as she developed a new idea or played a game she'd just written.

Software Design, Cro-Magnon Style

So, what would "the best adventure game ever" look like? I started with the only adventure game I'd ever played: Classic Adventure. What did I not like about it? Well, once you'd figured out the map and solved the puzzles, it was instantly boring. It played the same way every time. Each Non-Player Character (NPC) was tied to a single location, and always did the same thing. Lastly, you had to figure out exactly the incantation the game expected; if the game expected "kill troll", then any other command – "attack the troll", for example – would get an error message. You could spend a long time trying to figure out what command the game developer intended you to issue; as a result, most adventure games tended to have the same actions, paired with the same vocabulary.

Phil and I split the game cleanly down the middle, with clearly defined interfaces between the two halves. I took what today we would call the game engine and data structures. Phil took the interface and language portion. I don't remember who had the original idea of a much more developed language than the standard "kill troll" style of language used by other text adventures of the time; my thinking stopped at the level of having synonyms available for the commands. I had almost no involvement in the parser; I remember overhearing conversations between Fred and Phil as the complexity of what they were aiming at increased. For a time, Stuart Richie was brought in to provide language expertise. However, his thinking was not well suited to what was possible to develop in Assembler in the space and time available, so, according to what Phil told me at the time, none of his design was used – although I suspect that being exposed to his thinking helped Phil crystallize what eventually became Inglish. No matter what the user entered – "take the sharp sword and excitedly hack at the evil troll", say, he'd convert it to a simple (action, target) pair to hand off to me: "kill troll", or perhaps, "kill troll with sword". Compound sentences would become a sequence of actions, so "take the hammer and hit Gandalf with it" would come to me as two actions: "pick up hammer", followed by a next turn of "hit Gandalf with hammer".

I put together the overall design for a game that would remove the non-language-related limitations within a couple of hours on my first day on the job. I knew I wanted to use generalized, abstracted data structures, with general routines that processed that structure and with exits for "special cases", rather than the usual practice of the time of hard-coding the game-play. My intent was that you could develop a new game by replacing the content of the data structures and the custom routines – a "game engine" concept I did not hear described until decades later. We even talked about developing a "game editor" that would allow gamers to develop their own adventure games by entering items into the data structures via an interface, but I believe it was never developed. I very early on decided that I wanted randomness to be a key feature of the game – recognizing that that meant the game could not always be solved, and accepting that constraint.

I envisaged three data structures to be used to support the game: a location database, a database of objects and a database of "characters". The location "database" (actually, just a collection of records with a given structure) was pretty straightforward, containing a description of the location and, for each direction, a pointer to the location reached. There could also be an override routine to be called when going in a direction. The override allowed features or game problems to be added to the game map: for example, a door of limited size (so you could not pass through it while carrying too many items) or a trap to be navigated once specific constraints had been met. There's a location (the Goblin's Dungeon) that uses this mechanism to create a dynamic map, rather than having fixed connections to other locations: for each direction, an override routine is called that randomly picks a "next location" for the character to arrive in from a given list of possible locations. Another innovation in the location database occurred when Phil added pictures to specific locations, and drew them when the player entered one of those locations.

Similarly, I conceived of an object database with a set of abstract characteristics and possible overrides, rather than hard-coding a list of possible player interactions with specific objects. Each object had characteristics and constraints that allowed me treat them generically: weight, size, and so on – in effect, a simple (by today's standards) physics engine. An object could have the capability to act as a container, and a container could be transparent or opaque; a transparent container's contents could be seen without having to open it first. There were generic routines that could be applied to all objects: for example, any object could be picked up by something bigger and stronger than it, or put into a bigger container (if there was enough room left in it). Some routines could be applied to any object that matched some set of characteristics; an object could also have a list of "special" routines associated with it that overrode the general routines. There was a general "turn on" routine that applied to lamps, for example, that could also be overridden for a magic lamp by a different, more complex "turn on" routine.

Each non-player character (NPC) was also an object that began in an "alive" state, but could, due to events in the game, stop being alive – which allowed a player to, for example, use a dead dwarf as a weapon, in the absence of any other weapon). However, the physics engine caused "kill troll with sword" to inflict more damage than "kill troll with (dead) dwarf".

In addition to regular object characteristics, each NPC had a "character", stored in the third database. I conceived of an NPC's character as being a set of actions that the NPC might perform, a sequence in which they generally performed them and a frequency of repetition. The individual actions were simple and were generally the same actions that a player could do (run in a given direction, attack another character, and so on); but again, these routines could be overridden for a specific character. The sequence could be fixed or flexible: an action could branch to a different part of the sequence and continue from there, or even jump to a random location in the sequence. The

apparent complexity of the character comes from the length and flexibility of its action sequence; the character "emerges" as a result. For example, Gandalf's short attention span and kleptomania were represented by a sequence like: "[go] <random direction>. [Pick up] <random object> [Say, "what's this?"]. [Go] <random direction>. [Put down] <random object>."

The division between inanimate object and NPC was left intentionally a little blurry, giving extra flexibility. For example, the object overrides could also be used to modify character behavior. I actually coded an override where, if the player typed "turn on the angry dwarf", he turned into a "randy dwarf" and followed the player around propositioning him. If he was later turned off, he'd return to being the angry dwarf and start trying to kill any live character. Fred and Phil made me take that routine out.

In order to develop each character, I went through the book and tried to identify common sequences of behavior that I could represent through a sequence of actions that would capture the "texture" of the character. Some characters were easy; for a troll, "{If no alive object in current location} [go] <random direction> {else} [kill] <random object with status 'alive'>" was pretty much the whole list. Others were harder, such as characterizing Thorin; and yes, I did write the now-classic phrase, "Thorin sits down and starts singing about gold." (I hereby apologize for how frequently he said that; short character-action list, you see.) An action could invoke a general routine which was the same for all NPCs – like, choose a random direction and run, or choose a live object in the location and kill it; or, it could be an action specific only to this NPC, as with Thorin's persistent singing (as seen in Figure 1). For Gandalf, the generic "pick up" routine was used under the covers, but overridden for Gandalf to utter "what's this".

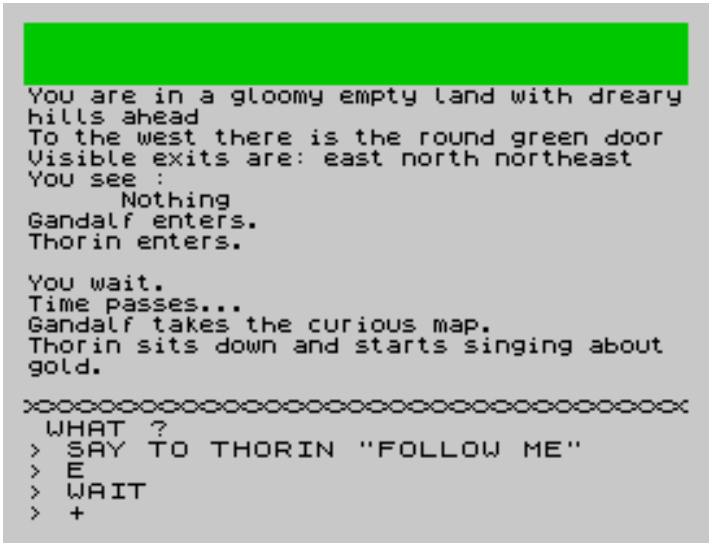

Figure 1. Gandalf and Thorin exhibit classic behavior. Courtesy Winterdrake.

Sometimes an alternate behavior list could be chosen based on events, as can be seen in Figure 2. For example, the friendly dwarf would become violent once he'd been attacked (or picked up). For a while, we had terrible trouble with all the NPCs showing up in one location and then killing each other before the player had the chance to work his way through the game, before I got the character profiles better adjusted. Some character would attack another, and once a battle was in progress any (otherwise friendly) character entering that location would be attacked and end up joining in. The same mechanism was used to allow the player to request longer-running actions from NPCs, such as asking a character to follow you when you needed them to help solve a puzzle in a (sometimes far) different location from where they were when you found them. In general the NPCs were programmed to interact with "another", and did not differentiate whether the "other" was the player or not unless there was a game-related reason for doing so. The NPCs exhibited "emergent behavior"; they just "played" the game themselves according to their character profile, including interacting with each other. In essence, the NPCs would do to each other almost anything that they could do to or with the player.

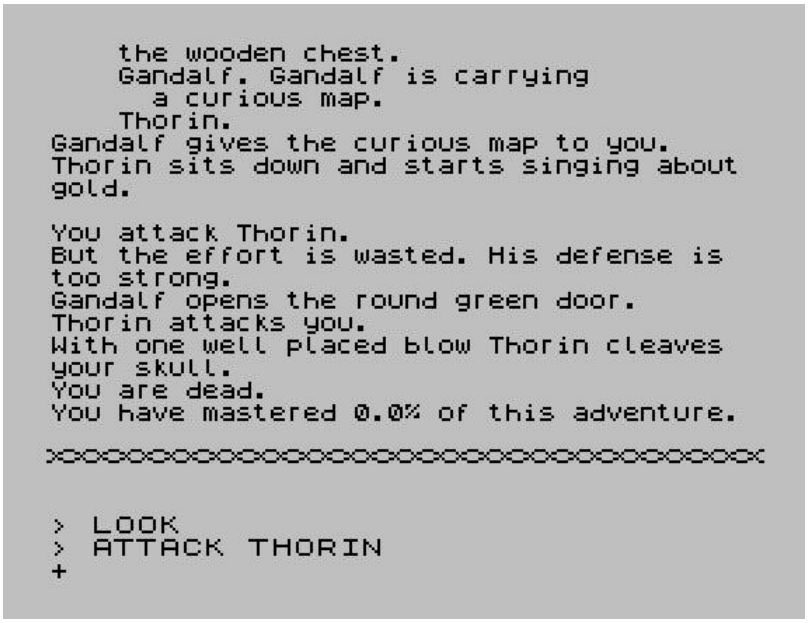

Figure 2. The player modifies Thorin's default behavior – to the player's cost.

Phil programmed the interface to accept input from the player, and after each turn he would hand control to the NPC system, which would allow each (remaining) alive character to take a turn, as can be seen in Figures 1 and 2. For the time, this design was revolutionary; the model then was to have a single, non-mobile NPC in a single location, with only a couple of specific actions that were invoked once the player entered that location, and behaving the same way each time you played the game. Even in the arcade games of the time, we were able to identify that each object the player interacted with behaved the same way each time, and they did not interact with each other at all.

At the beginning of the game, we would generate, for each NPC, a random starting point in that NPC's action list, giving the game much of its random nature. This combination of factors led to the "emergent characters"; or, seen another way, "a bunch of other characters just smart enough to be profoundly, infuriatingly stupid" (Maher 2012).

I quickly transitioned to the concept of the player merely being another character, with a self-generated action list. At some point I experienced the emergent nature of the characters while trying to debug and was joking about the fact that the characters could play the game without the player being there; that discussion led naturally to the famous "time passes" function, where, if the player took too long in taking his next action (or, chose to "wait", as in Figure 1), the characters would each take another turn. This feature, which Melbourne House trademarked as "Animaction" (Addison-Wesley Publishing Company, Inc. 1985), was another innovation not seen in prior text adventures, where game-play depended wholly on the player's actions. (It is also noteworthy how many of the game's innovations began as jokes. I now believe this to be true of much innovation; certainly it has been, for the innovations I've been involved in.)

The next, seemingly obvious step to me was to allow – or even require – the player to ask the NPCs to perform certain tasks for him (as seen in Figure 3), and to set up puzzles that required this kind of interaction in order to solve them. This addition added another layer of complexity to the game. As commented by one fan, "As most veteran Hobbit players know, a good way to avoid starvation in the game is to issue the command "CARRY ELROND" whilst in Rivendell. In the game Elrond is a caterer whose primary function is to give you lunch and if you carry him then he will continue to supply you with food throughout the game."[2] Another had a less tolerant view: "Sometimes they do what you ask, but sometimes they're feeling petulant. Perhaps the seminal *Hobbit* moment comes when you scream at Brand to kill the dragon that's about to engulf you both in flames, and he answers, "No." After spending some time with this collection of half-wits, even the most patient player is guaranteed to start poking at them with her sword at some point."[3]

[2] http://solearther.tumblr.com/post/38456362341/thorin-sits-down-and-starts-singing-about-gold
[3] http://www.filfre.net/2012/11/the-hobbit/

The non-determinism of the overall game meant that it was not, in general, possible to write down a solution to the game. There were specific puzzles in the game, however, and solutions to these puzzles could be written down and shared. However, people also found other ways to solve them than I'd anticipated. For example: "A friend of mine has discovered that you can get and carry both Elrond and Bard. Carrying Elrond with you can by quite useful as he continuously distributes free lunches. And, to be honest, carrying Bard is the only way I've found of getting him to the Lonely Mountain. There must be a better way." ("Letters: Gollum's Riddle" 1984) As commented by a retrospective, "And actually, therein sort of lies the secret to enjoying the game, and the root of its appeal in its time. It can be kind of fascinating to run around these stage sets with all of these other crazy characters just to see what can happen — and what you can make happen." (Maher 2012)

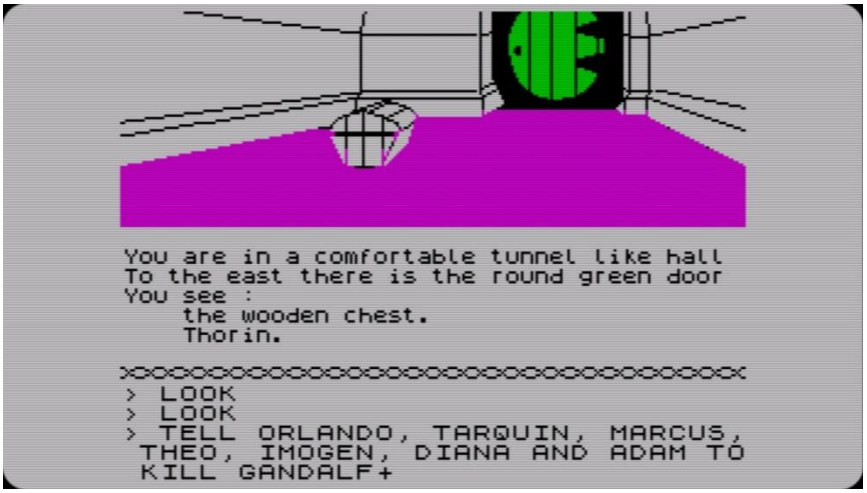

Figure 3. "The Hobbit" starting location, and a player action that I never thought of.

Inglish

While I worked on the game, Phil designed, developed and wrote the language interpreter, later dubbed Inglish. I had little interest in linguistics, so I generally tuned out the long discussions that Fred and Phil had about it – and was supported in doing so by the encapsulation and simple interface between the two "halves" of the game, which prevented me needing to know any more.

Every word was stored in the dictionary, and since only 5 bits are used to represent the English alphabet in lower-case ASCII, the other 3 bits were used by Phil to encode other information about speech parts (verb, adjective, adverb, noun), valid word usages, what pattern to use when pluralizing, and so on. I've seen screen images from versions of the game in other languages (e.g., Figure 4), but I do not know how the translations were done or how the design worked with these other languages.

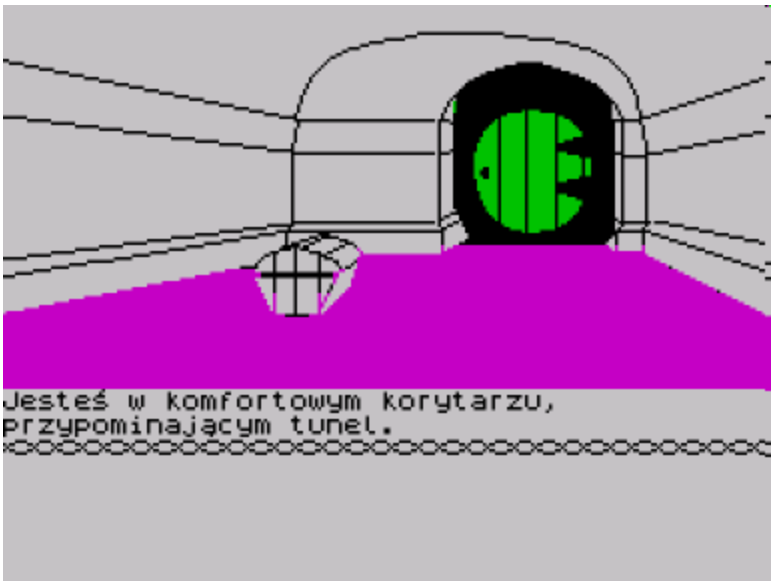

Figure 4. Opening scene from one of many foreign language versions.

Phil translated player commands into simple "verb object" commands to hand to me, with some allowed variations to allow for different action results. For example, I seem to remember that "viciously kill" would launch a more fierce attack, and use up more strength as a result, than just "kill". Rather than a set of hard-coded messages (as was the norm), we generated the messages "on the fly" from the dictionary and a set of sentence templates. At the end of some action routine, I would have a pointer to a message template for that action. The template would contain indicators for where the variable parts of the message should be placed. I would then pass the message, the subject and object to the language engine. The engine would then generate the message, using, once again, spare bits for further customization. To take a simple example, "Gandalf gives the curious map to you" used the same template as, say, "Thorin gives the axe to the angry dwarf".

We were so limited by memory that we would adjust the size of the dictionary to fit the game into the desired memory size; so the number of synonyms available would sometimes decrease if a bug fix required more lines of code. It was a constant trade-off between game functionality and language richness. As a result of all the encoding, dumping memory – a common method of solving puzzles in other text adventures – provided no information for "The Hobbit".

Software Development, Cro-Magnon-Style

Our initial development environment was a Dick Smith TRS80 look-alike, with 5 inch floppy drives. Initially I believe we used a 16k machine, then a 32k, and towards the end a 48k or perhaps 64k machine. Our target machine for the game was initially a 32k TRS80. During development, the Spectrum 64 was announced, and that became our new target. Game storage was on a cassette tape, played on a regular radio-cassette player. As the other systems became available we continued using the TRS80 platform as the development environment, and Phil took on the question of how to ports the game to other platforms.

We had a choice of two languages to use for development: Basic, or Assembler. We chose Assembler.

During initial development, the only development tool available was a simple Notepad-like text editor, and the majority of code was written that way. Later I believe a Vi-like editor became available; even later, I have faint memories of a very early IDE that allowed us to edit, assemble the code and step through it.

We initially worked with the system's existing random number generator, but realized that its pseudo-random nature made the game play the same way each time – against what I hoped to achieve. Phil then spent some time writing a "true" random number generator, experimenting with many sources of seed values before he was successful. He tried using the contents of various registers, but discovered that these were often the same values each time. He tried using the time, but the TRS80 did not have a built-in battery or time, and most people did not set the time each time they started the system – so again, if someone turned the machine on and loaded the game, we would get the same

results each time. After some experimentation he finally succeeded, and the game – for better or worse, and sometimes for both – became truly random.

Debugging was a nightmare. Firstly, we were debugging machine code, initially without the advantage of an IDE; we ran the program, and when it crashed we tried to read the memory dumps. In Assembler, especially when pushing the memory limit of the system, the Basic programmer's technique of inserting "print" statements to find out what is happening is not available. We had characters interacting with each other in distant parts of the game, and only actions in the current location were printed on the game player's console. In one of several cases where a game feature was originally developed for other reasons, we initially wrote the "save" mechanism to help us debug parts of the game without having to start from the beginning each time. It then became part of the delivered version, allowing players to take advantage of the same function.

At some point, the idea of adding graphics came up, I think from Phil. Fred commissioned Kent Rees to draw the pictures, and Phil figured out how to draw them on the various systems; I do know that he adapted the pictures from the originals Kent provided in order to make them easier to draw. The first version of his code always drew the picture when you entered a location that had one; however, it was so slow and annoyed us (me) so much that Phil quickly added a switch to turn them off.

Sidelines

In between coding "The Hobbit", we occasionally took time to work on other games. Fred would give us $20 to go and play arcade games, sometimes as often as each week, to see what other folk were doing and what the state of the art was in that industry. Someone in our group of four wrote a version of Pac-Man. We spent hours with one person playing Pac-Man, trying to get up to higher levels in the game, while the others leant over the arcade machine trying to figure out the algorithms that caused prizes to appear and how the behavior changed across the game levels. We didn't see it as piracy, as arcade games and home computers were at that time seen as being completely unrelated industries – it was more in the spirit of gaining ideas from another industry for application into ours.

Another game that we wrote was Penetrator; Phil was the clear lead on that game while I worked on some pieces of it, and I think Kerryn may have worked on it a bit too. It was a copy of the arcade game "Scramble". Because of the speed (or lack thereof) of the processors at the time, we had to ensure that each separate path through the game took the same amount of time; even a difference of one "tstate" (processor state) between one path of an "if-then-else" to another would interfere with smooth motion, so we spent significant time calculating (by hand) the time taken by each path and choosing different Assembler instructions that would compensate for the differences (and given that "NO-op" took 2 tstates, it was not always easy). Another difficulty was getting the radars to turn smoothly, while handling the variable number of other activities taking place in the game. It took forever to get it "right".

At the beginning we drew the screen bitmaps for all the landscapes on graph paper and then hand-calculated the hexadecimal representations of each byte for the screen buffer, but that became so tedious so quickly that Phil wrote an editor that we could use to create the landscapes. In the end the landscape editor was packaged with the game, as a feature.

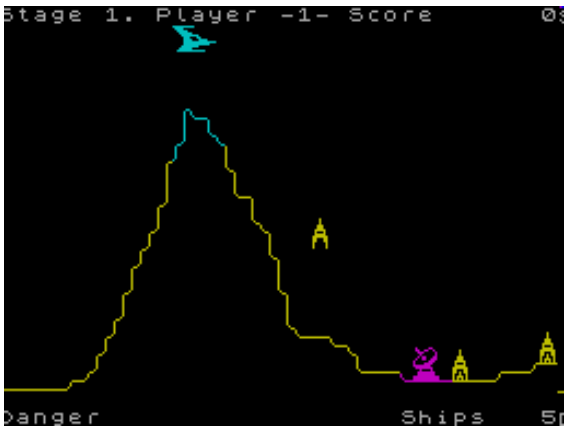

Figure 5. Screen shot from the game Penetrator

Another "pressing" issue for shooter games of the time was that of keyboard debounce. At the time a computer keyboard consisted of an electrical grid, and when a key was pressed the corresponding horizontal and vertical lines would register a "high". You checked the grid at regular intervals, and if any lines were registering high you used a map of the keyboard layout to identify the key that had been pressed. However, you had to stall for just the right amount of time before re-reading the keyboard; if you waited too long, the game seemed unresponsive, but if you read too quickly, you would read several key presses for each key press that the player intended. While it was possible to use the drivers that came with the keyboard, they did not respond quickly enough to use for interactive games. "Getting it right" was a tedious matter of spending hours fiddling with timings and testing.

Perhaps A Little Too Random

In addition to all the other randomness it exhibited, "The Hobbit" was also known to crash seemingly randomly. There were a number of reasons for this. Firstly, "The Hobbit" was a tough game to test. It was a much bigger game than many others of the time. Unlike the other games, it was approximately 40k of hand-coded Assembler[4], as opposed to the commonly-used interpreted Basic (a few more advanced games were shipped in compiled Basic). It was written without the benefit of formalized testing practices or automated test suites. The assembly and linking programs we used were also relatively new, and during development, we would find bugs in them. I remember spending hours debugging one time only to discover that the assembler had optimized away a necessary register increment, causing an infinite loop; I had terrible trouble trying to invent a different coding sequence that prevented the assembler from removing the required increment. Altogether, I took away lessons about not letting your application get too far ahead of the ability of your infrastructure to support it.

Secondly, the game was non-deterministic; it was different every time it was played. It exhibited its own manifestation of chaos theory: small changes in starting conditions (initial game settings, all generated by the random number generator) would lead to large differences in how the game proceeded. Due to the "emergent characters", we constantly had NPCs interacting in ways that had never been explicitly progammed and tested, or even envisioned. The game could crash because of something that happened in another location that was not visible to the player or to person testing the game, and we might never be able to identify or recreate the sequence of actions that led to it.

It was possible to have an instance of the game that was insoluble, if a key character required to solve a specific puzzle did not survive until needed (often due to having run into a dwarf on the rampage); this was a constraint I was happy to accept, though it frustrated some players. The ability to tell the NPCs what to do also meant that people told them things to do that we hadn't accounted for. The very generality of the game engine – the physics, the language engine, and the ability for the player to tell characters what to do – led players to interact with the game in ways I'd never thought of, and that were certainly never tested. In some cases, they were things I didn't realize the game was capable of.

Epilogue

"The Hobbit" was released in 1982 in Australia and the U.K. Figure 6 shows a typical packaging. It was an instant hit; amongst other awards, it won the Golden Joystick Award for Strategy Game of the Year in 1983, and came second for Best Game of the Year, after "Jet-Pac". Penetrator came second in the Golden Joystick Best Arcade Game category, and Melbourne House came second for their Best Software House of the Year, after Jet-Pac's publishers ("Golden Joystick Awards" 1984). A couple of revisions were published with some improvements, including better graphics. Due to licensing issues it was some time before a U.S. release followed.

[4] An analysis by the Wilderland project ("Wilderland: A Hobbit Environment" 2012) shows the following code breakdown: game engine and game, 36%; text-engine for input and output, the dictionary, the graphics-engine, and the parser 22%, graphics data 25%; character set (3%), buffers (8%), and 6% as yet unidentified.

Figure 6. Game release package.

At the end of 1981, I finished my Bachelor's degree. We were beginning to discuss using the Sherlock Holmes mysteries as a next games project; I was not sure that the adventure game engine I'd developed was a good fit for the Sherlock style of puzzle solving, although there were definitely aspects that would translate across. However, I was also ready to start something new after a year of coding and debugging in Assembler. I'd proved that my ideas could work, and believed that the result Phil and I had produced was the desired one – an adventure game that solved all my frustrations with Classic Adventure, and in my mind (if not yet in other people's) met Fred's target of "the best adventure game ever".

I interviewed with several major IT vendors, and took a job at IBM, as did Ray. Kerryn took a job in a mining company in Western Australia. Phil stayed on at Melbourne House (later Beam Software), the only member of our university programming team to continue on in the games industry. We eventually all lost touch.

During this time, I was unaware that the game had become a worldwide hit. Immersed in my new career, I lost touch with the nascent games industry. At IBM, I started at the same level as other graduates who had no experience with computers or programming; developing a game in Assembler was not considered professional or relevant experience. Initially I became an expert in the VM operating system (the inspiration and progenitor for VMWare, I've heard), which I still admire for the vision, simplicity and coherence of its design, before moving into other technical and consulting position. In late 1991 I left Australia to travel the world. I eventually stopped in Portland, Oregon, with a plan to return to Australia after 2 years – a plan that has been much delayed.

A 3-year stint in a global Digital Media business growth role for IBM U.S. in the early 2000's brought me back in contact with games developers just as the movie and games industries were moving from proprietary to open-standards based hardware and infrastructure. The differences in development environments, with large teams and sophisticated supporting graphics and physics packages, brought home to me how far the games industry had come. But while I appreciate the physics engines and the quality of graphics that today can fool the eye into believing they are real, the basis of a good game has not changed: simple, compelling ideas still captivate and enchant people, as can be seen in the success of, for example, Angry Birds. I also believe that the constraints of limitations – such as small memories and slow processors – can lead to a level of innovation that less limited resources does not.

And Back Again

As the Internet era developed, I started receiving letters from fans of "The Hobbit". The first person I recall tracking me down emailed me with an interview request for his Italian adventure fan-site in 2001, after what he said was a long, long search. The subsequent years made it easier to locate people on the Internet, and the emails became more frequent. At times I get an email a week from people telling me the impact the game had on the course of their lives.

In 2006, the Australian Centre for the Moving Image (ACMI) held an exhibition entitled "Hits of the 80s: Aussie games that rocked the world" (Australian Centre for the Moving Image 2007), featuring "The Hobbit". It felt a little like having a museum retrospective while still alive: a moment of truth of how much things have changed, and at the same time how little. The games lab curator, Helen Stuckey, has since written a research paper about the challenge of collecting and exhibiting videogames for a museum audience, using "The Hobbit" as an example (Stuckey).

In late 2009 I took an education leave of absence from IBM US to study for a Masters/PhD in Computer Science at Portland State University. (IBM and I have since parted company.) When I arrived one of the PhD students, who had played "The Hobbit" in Mexico as a boy, recognized my name and asked me to present on it. While searching the Internet for graphics for the presentation, I discovered screen shots in many different languages and only then began to realize the worldwide distribution and impact the game had had. Being in a degree program while describing work I'd done during my previous university degree decades before caused many conflicting emotions. I was also amazed at the attendance and interest from the faculty and other students.

In 2012, the 30-year anniversary of the release, several Internet sites and magazines published retrospectives; a couple contacted me for interviews, while others worked solely from published sources. The same year I was contacted by a fan who had been inspired by a bug ("this room is too full for you to enter") to spend time over the intervening decades in reverse-engineering the machine code into a "game debugger" of the kind I wish we'd had when we originally developed it: Wilderland ("Wilderland: A Hobbit Environment" 2012). It runs the original game code in a Spectrum emulator, while displaying the position and state of objects and NPCs throughout the game. His eventual conclusion was that the location is left over from testing (and I even have a very vague memory of that testing). That a game I spent a year writing part-time could cause such extended devotion is humbling.

In retrospect, I think we came far closer to Fred's goal of "the best adventure game ever" than we ever imagined we would. The game sold in many countries over many years, and by the late 1980's had sold over a million copies (DeMaria 2002) – vastly outselling most other games of the time. During one interview, the interviewer told me that in his opinion, "The Hobbit" transformed the genre of text adventure games, and that it was the last major development of the genre: later games merely refined the advances made. Certainly Beam Software's games after "The Hobbit" did not repeat its success.

While many of the publications, particularly at the time of release, focused on the Inglish parser, it is the characters and the richness of the gameplay that most people that contact me focus on. I believe that just as the game would have been less rich without Inglish, putting the Inglish parser on any other adventure game of the time would in no way have resembled the experience of playing "The Hobbit", nor would it have had the same impact on the industry or on individuals.

In 2013, the Internet Archive added "The Hobbit" to its Historical Software Collection[5] – which, in keeping with many other Hobbit-related events, I discovered via a colleague's email. Late that year, ACMI contacted me to invite me to join the upcoming Play It Again project, a game history and preservation project focused on ANZ-written digital games in the 1980s. That contact led to this paper.

As I complete this retrospective – and my PhD – I am struck again by the power a few simple ideas can have, especially when combined with each other. It's my favorite form of innovation. In the words of one fan, written 30 years after the game's release, "I can see what Megler was striving toward: a truly living, dynamic story where anything can happen and where you have to deal with circumstances as they come, on the fly. It's a staggeringly ambitious, visionary thing to be attempting."(Maher 2012) A game that's a fitting metaphor for life.

Disclaimer

This paper is written about events 30 years ago, as accurately as I can remember. With that gap in time, necessarily some errors will have crept in; I take full responsibility for them.

Note that all screenshots shown were found on the Internet as I was writing this paper.

References

Addison-Wesley Publishing Company, Inc. 1985. "The Hobbit: Guide to Middle-Earth." http://playitagainproject.org/wp-content/uploads/hobbit-manual-addison-wesley.pdf.

Australian Centre for the Moving Image. 2007. "Hits of the 80s: Aussie Games That Rocked the World." May. http://www.acmi.net.au/hits_80s_home.aspx.

[5] https://archive.org/details/The_Hobbit_v1.0_1982_Melbourne_House

DeMaria, Rusel, Wilson, Johnny L. 2002. *High Score!: The Illustrated History of Electronic Games*. Berkeley, Cal.: McGraw-Hill/Osborne.
"Golden Joystick Awards." 1984. *Computer and Video Games*, March.
"Letters: Gollum's Riddle." 1984. *Micro Adventurer*, March.
Maher, Jimmy. 2012. "The Hobbit." *The Digital Antiquarian*. November. http://www.filfre.net/2012/11/the-hobbit/.
Stuckey, Helen. "Exhibiting The Hobbit: A Tale of Memories and Microcomputers."
"Wilderland: A Hobbit Environment." 2012. http://members.aon.at/~ehesch1/wl/wl.htm.

Veronika M. Megler very recently completed her PhD in Computer Science at Portland State University, working with Dr. David Maier in the emerging field of "Smarter Planet" and big data. She received her M.Sc. in Computer Science at Portland State in 2012. V.M. Megler's dissertation research enables Information-Retrieval-style search over scientific data archives. Her most recent industry position was as Executive IT Architect at IBM USA. She has published more than 20 industry technical papers and 10 research papers on applications of emerging technologies to industry problems, and holds two patents, including one on her dissertation research. Her research interests include applications of emerging technologies, scientific information management and spatio-temporal databases. Ms. Megler was in the last year of her B.Sc. studies at Melbourne University when she co-wrote "The Hobbit." She currently lives in Portland, Oregon, and can be reached at vmegler@gmail.com.