How to Fit a Large Program in a Small Machine

How to Fit a Large Program into a Small Machine

or

How to fit the Great Underground Empire on your desk-top

Marc S. Blank and S. W. Galley

[Panel]

As a rule, "sophisticated" progrmming is pretty boring - optimizing the unnecessary to speed up the uninteresting. Here, however, is an incredibly sophisticated package intended for fun and games. The more you know about software, the more astounded you will become as you read this.

Imagine yourself sitting down at your favorite personal computer, inserting a diskette, turning on the power, and seeing what follows. Your typed replies are in capital letters.

```
Zork: The Great Underground Empire
part 1, release 1
(c) Copyright 1980 Infocom, Inc. All
rights reserved. Zork is a trademark
of Infocom, Inc.

West of House
     You are standing in an open field
west of a white house, with a boarded
front door.
     A rubber mat saying 'Welcome to
Zork!' lies by the door.
     There is a small mailbox here.
>OPEN THE SMALL MAILBOX
    Opening the mailbox reveals a
leaflet.
>READ THE LEAFLET
(Taken)
  Welcome to Zork: The Great Under-
ground Empire
    Zork: The Great Underground Em-
pire is a game of adventvre, danger,
and low cunning. In it you will explore
some of the most amazing territory
ever seen by mortals.
    No computer should be without
one!
    The original ZORK was created by
Tim Anderson. Marc Blank, Bruce
Daniels, and Dave Lebling. It was
inspired by the ADVENTURE game of
Crowther and Woods. This version was
created by Marc Blank, Dave Lebling,
Joel Berez, and Scott Cutler.
>WALK AROUND THE HOUSE
North of House
    You are facing the north side of a
white house. There is no door here,
and all the windows are boarded up. To
the north a narrow path winds through
the trees.
>AGAIN
Behind House
    You are behind the white house. A
path leads into the forest to the east. In
one corner of the house there is a small
window which is slightly ajar.
>LOOK IN THE WINDOW
    You can see what appears to be a
kitchen.
>GO IN
    The kitchen window is closed.
>OPEN IT
    With great effort, you open the
window far enough to allow entry.
>GO IN
Kitchen
      You are in the kitchen of the white
house. A table seems to have been
used recently for the preparation of
food. A passage leads to the west and a
dark staircase can be seen leading
upward. A dark chimney leads down
and to the east is a small window which
is open.
      On the table is an elongated brown
sack, smelling of hot peppers. A bottle
is sitting on the table.
     The glass bottle contains:
A quantity of water
>PICK UP
pick up what?
>ALL BUT THE SACK
kitchen table: An interesting idea . . .
glass bottle: Taken.
quantity of water: I can't reach that.
>GO WEST
Living Room
     You are in the living room. There is
a door to the east, a wooden door with
strange gothic lettering to the west,
which appears to be nailed shut, and a
large oriental rug in the center of the
room.
     There is a trophy case here. A
battery-powered brass lantern is on
the trophy case. Above the trophy case
hangs an elvish sword of great
antiquity.
```

This transcript shows the beginning of an adventure into a land of Zork. Zork was originally a single "computerized fantasy simulation" game in which the object is to explore a labyrinth, called the Great Underground Empire, comprising a variety of interesting caves or rooms, and to find hidden treasures and return safely with them. We now say that Zork is a computer-game environment in which different games, with different scenarios and different puzzles to solve, can be written. A Zork player converses with a Zork program by typing commands in a kind of restricted English and reading the program's English responses. A longer description of the original game and program can be found in "Zork: A Computerized Fantasy Simulation Game" by Lebling, Blank, and Anderson (IEEE Computer, April 1979, pp. 51-59).

The original Zork game was implemented on a DECsystem-10 at the MIT Laboratory for Computer Science in a local Lisp-like language called MDL. This Zork game was later translated into a Fortran version for DEC PSP-11 Computers and made available through the DECUS program library. In both versions the program is large: it occupies most of a process's virtual storage on a 10, and it requires a large disk for secondary storage on an 11. In converting Zork to run on personal computers, the designers needed some way to shrink it in order to fit it into the relatively small available storage.

One shrinking tactic was to remove the features of MDL that are not needed in Zork, such as coroutines, associative storage, and fancy input/output. The stripped-down version of MDL that resulted was named Zork Implementation Language (ZIL). However, that was not enough: a straightforward compilation of a ZIL version of the original Zork game into the machine language of any known personal computer would still have produced an executable program too large to fit.

The solution was to invent a "virtual machine," specifically designed to execute Zork programs; the virtual "Z-machine" has a machine language called "Z-code", in which Zork programs can be expressed very compactly. Then all that was needed was a Zork Interpretive Program (ZIP), written in the machine language of any given target personal computer, that would imitate a Z-machine in carrying out the Z-code operations. (A compiler that translates from ZIL to Z-code is also needed, of course, but the highly-structured nature of MDL, and hence ZIL, makes that a relatively simple task.) A good benchmark for the storage saved by rewriting Zork in ZIL is the Zork parser, which analyzes a player's English input: the parser for the PDP-10 occupies 1OK 36-bit words, while the Z-code parser, which is actually better functionally, occupies only 3K 8-bit bytes.

This Z-code approach is similar to that of compiling a Pascal program into "P-code," (although there are now P-code machines, like Western Digital's Pascal Microengine^{TM}, that are real and not just virtual). In effect, Z-code is like P-code: a string of subprogram calls, with the bodies of the subprograms executed by a Z-machine or ZIP. Any often-used sequence of pperations in Zork programs could, in principle, be compressed into a Z-code instruction, thereby moving the sequence of operations into the Z-machine or ZIP, where it needs to appear only once. The Z-machine designer just has to be judicious in choosing Z-code bit patterns and subprogram parametrizations to get the most benefit from this virtual-machine method.

Besides compressing the space needed by Zork programs, the Z-code approach also makes conversion to another (real) computer easier, because, assuming that the design of Z-code is reasonably machine-independent, all one needs to do is to implement ZIP on the new machine.

Z-code obiects:

Z-code is an object-oriented language (as are Lisp and MDL and ZIL). In this section the various kinds of objects and the possible operations on them are described. Excerpts from a transcript of a game are used to illustrate the uses of these objects.

All Z-code objects occupy one or two bytes in storage, and exactly two bytes while they are being processed. Like MDL, ZIL uses "type codes" to distinguish among the different types of objects, but Z-code does not, to save space: the ZIL compiler checks for proper use of types, but ZIP doesn't bother. A Z-code operation that yields a truth-value (integer 0 or 1) is called (as in Lisp) a "predicate"; the Z-code operation-codes for all predicates include a bit for inverting the sense of the test, another space-saving measure.

```
Dam Lobby
    This room appears to have been
the waiting room for groups touring
the dam. There are exits here to the
north and east marked 'Private,'
though the doors are open, and an exit
```

```
to the south.
    Some guidebooks entitled 'Flood
Control Dam #3' are on the reception
desk. There is a matchbook whose
cover says 'Visit Beautiful FCD#3' here.
>COUNT MATCHES
You have 5 matches.
>COUNT NOSES
I don't know the word 'noses.'
```

Integer:

An integer, such as the number of matches left in a matchbook, is stored in two bytes, according to the normal bit-level representation used by the hardware. Operations on integers include the four normal arithmetic functions, remainder or modular reduction, and generation of a random integer in the range 1-N. (Modular reduction - calculating the remainder in a division - is useful for stepping through a set of English responses cyclically. Random-integer generation is useful for choosing a response at random from a set of similar responses.) Predicates test for one integer being less than, greater than, or equal to another integer, and for an integer being zero. (Since the sense of a test can be reversed using one of the bits in its operation code, this means implicitly that there are also predicates to test for greater-than-or-equal, less-than-or-equal, not-equal, and not-zero.) Testing for zero may seem redundant, given an equality test, but it is used often enough that the cost of using another operation code is outweighed by the value of eliminating the byte that would be used to hold a zero in every instance. Naturally, there is an operation to "print&uot; an integer in the output stream - though, of course most personal computers display characters on a CRT rather than printing them on paper.

In some situations an integer is treated as a string of 16 independent bits; for this case there are operations for Boolean "and" and "or" and "not," and for testing individual bits. The characters in the player's input are also stored as integers, using the ASCII code; for this case there is an output operation to print a single character, as shown in the last response above.

```
The Troll Room
    This is a small room with passages
to the east and south and a forbidding
hole leading west. Bloodstains and
deep scratches (perhaps made by an
axe) mar the walls.
    A nasty-looking troll, brandishing
a bloody axe, blocks all passages out
of the room.
    Your sword has begun to glow
very brightly.
>KILL TROLL WITH KNIFE
    The blow lands, making a shallow
gash in the troll's arm! The troll swings
his axe, and it nicks your arm as you
dodge.
>AGAIN
    The quickness of your thrust
knocks the troll back, stunned. The
troll slowly regains his feet.
>AGAIN
    A quick stroke, but the troll is on
guard. The troll swings his axe, but it
misses.
>AGAIN
    A good slash, but it misses the troll
by a mile. The axe crashes against the
rock, throwing sparks!
>AGAIN
```

```
    The troll is disarmed by a subtle
feint past his guard. The troll, now
worried about this encounter, recovers
his bloody axe.
```

String:

As you can see, Zork programs tend to be wordy, so strings of characters need to be stored as compactly as possible. Three characters can fit in two bytes, if each character uses only five bits. But five bits can encode only 32 characters directly, and that is obviously not enough. The solutlon is to use different "contexts" and to reserve one or more "characters" for switching among contexts. (This technique is similar to the five-bit Baudot code, which was used by early Teletypes before ASCII was invented.) Z-code strings use three contexts - lower case, upper case, and digits/punctuation - and several characters for switching among the contexts, either "permanently" or only for the next character, the latter in order to capitalize a word or use a single punctuation mark. Because of the extra characters used to switch contexts, Z-code strings work out to about five and a half bits per visible character, which is still significantly more compact than eight bits.

The only operation on strings is to print them, i.e., show them to the player. There is no need to manipulate them, except to print strings sequentially so that they form sentences for the player. (Actually, simply because it occurs so often, there is also an operation for printing a string and then returning from a function call.) Most strings are stored without "new-line" characters, and ZlP takes care of "folding" the output into lines of a size convenient for the particular display being used; a few rigidly-formatted strings do use new-lines in order to draw a crude picture with the characters.

```
>SWIM
I don't really see how.
>SWIM
I think that swimming is best performed in water.
>SWIM
Perhaps it is your head that is
swimming.
...
>TAKE HOUSE
What a concept!
>AGAIN
A valiant attempt.
>AGAIN
You can't be serious.
>AGAIN
Not bloody likely.
>AGAIN
An Interesting idea...
```

Table:

A table is used to keep a set of related objects (like the responses above) together, as a list is used in Lisp or a record is used in Pascal. A table is stored as a number of two-byte (or sometimes one-byte) objects one after another. The length of a table is sometimes stored in its first elemcnt - to be used by parts of a program thst need to step through all the elements - and sometimes not - if the program itself knows how long the table is. Operations on a table can get a word (two bytes) out of it and put a word into it; get a single byte out, and put one in; and move the pointer along to another element (like CDR in Lisp) or backwards to a previous one (like BACK in MDL); (Actually; since a pointer to a table is just a plain address, ordinary addition and subtraction are used to move a pointer forward and backward in a table.)

Thing:

"Things" are probably the most interesting objects, since they represent the player, rooms, enemies, weapons, treasures and so on - the stuff of which a game is made. To avoid confusion, "Thing" will always be written with a capital "T" when it refers to this kind of Z-code object. Since there are relatively few Things used in a Zork game, but they are used very frequently, each Thing is designated by a one-byte number rather than, say, its two-byte address in storage. This design decision limits the number of possible Things in one game to 255, with the number zero reserved to mean "no Thing." (There are also "pseudo-Things", whose names the parser will recognize as significant but which have no interesting properties.)

A Thing's number can be easily translated into its address in storage where its parts are found: status bits, contents, and properties. These parts are stored sequentially, as in a table, in a very strict format: four bytes for status, three bytes for contents, and two bytes holding the address of the property table - so ZlP needs to know only the address of the first Thing in order to translate a Thing number into its address.

Each Thing has 32 status bits, which can be turned off or on or tested individually by Z-code operations. Status bits represent qualities of a Thing, both permanent (edible, burnable, fightable "room," etc.) and temporary ("lit," "open," etc.).

```
>TAKE SACK
Taken.
>LOOK IN SACK
The brown sack is closed.
>OPEN SACK
Opening the brown sack reveals a
lunch, and a clove of garlic.
>TAKE LUNCH OUT
Taken.
>TAKE CLOVER THEN PUT
BOTTLE IN SACK
Taken.
Done.
>LOOK IN SACK
The brown sack contains:
A glass bottle
The glass bottle contains:
  A quantity of water
```

A Thing's *contents* part relates it spatially to other Things in three ways: as a parent ("container") as a chlld ("containee"), and as a sibling ("inmate"). For example, if the knife and the bottle are in the kitchen, then the kitchen's "child" slot would hold the number of the knife, the knife's "sibling" slot would hold the number of the bottle, and the knife's (and bottle's) "parent" slot would hold the number of the kitchen. (Of course, depending on how they got there, the knife and the bottle might be interchanged in this data structure.) Such a data structure is commonly called a list: Things can be added to or removed from a list simply by moving the appropriate numbers into the slots. Manipulating list structures of this kind - the reason Lisp was invented - is essential in Zork games, as Things get moved here and there by the player. For convenience, "contents" has a more general meaning in ZIL: for example, the player's baggage is "contained in" the player Thing. The operations on contents are

- "move X into Y,"
- "remove X from everything" (e.g., if it is eaten or otherwise destroyed),
- "is X in Y?" and "what is X in (if anything)?" (using X's parent slot),
- "what is the first Thing (if any) in X?" (using X's child slot), and
- "what is the next Thing (if any) in the same Thing that X is in?" (using X's sibling slot).

```
     This is the attic. The only exit is a
stairwey leading down. On a table is a
nasty-looking knife. A large coil of
rope is lying in the corner.
>TAKE NASTY KNIFE
Taken.
...
Maze
     This is part ot a maze of twisty little
passages, all alike.
     A skeleton, probably the remains
of a luckless adventurer, lies here.
Beside the skeleton is a rusty knife.
     The deceased adventurers use-
less lantern is here. There is a skeleton
key here. An old leather bag, bulging
with coins, is here.
>TAKE RUSTY KNIFE
     As you pick up the rusty knife,
your sword gives a single pulse of
blinding blue light.
```

Finally, each Thing has a table (in the format described previously) of its properties, such as name, size, capacity, score value, verbose description, north neighbor, synonyms, and action routines; the last can be seen in the transcript above involving the two knives. (The name of a Thing identifies it uniquely to the player throughout the whole game, e.g., "kitchen," "bottle," "thief." There is a special Z-code operation for printing a Thing's name.) Since there are a limited number of properties a Thing can have, and not all of them require the same smount of storage, a special format is used to store them in this table: the first byte of each property has a five-bit property number (allowing 32 different properties) and a three-bit count of the number of immediately following bytes used to store the property. The operations on properties are "get property Y of Thing X," "store Z as property Y of Thing X," "get the storage address of property Y of Thing X," and "get the next property of Thing X following this property."

```
>SCORE
    Your score would be 10 (total of
375 points), in 9 moves. This score
gives you the rank of Beginner.
```

Variable:

Variables are used to keep track of the situation all through the game. (Parts of Things are used also.) For example, the player's current score is a variable, called a "global" variable because it is the same no matter what functions have been called. Like Things, variables are identified by a one-byte number. A special number is used to identify the top of the "stack," and fifteen more numbers identify "local" variables used by the function currently executing. (The stack is described later.) The remaining 240 numbers identify global variables. Operations on variables store and retrieve the current value, increment or decrement an integer value by one and optionally test it for crossing a given threshold, push a value onto the stack, pop a value of the stack, and print the value.

Other operations:

The remaining Z-code operations do not deal directly with objects. These operations include the null operation, go-to (jump, branch), call a function, return a value to the calling function, and return "true" or "false" or whatever is on top of the stack to the calling function. Also there are special operations to input a line from the player and find the significant words in it using a vocabulary table, save the game situation on a disk or tape, restore a situation saved previously, quit playing or start afresh.

Z-code format:

Each Z-code instruction begins with a byte containing the operation code, which includes bits describing how the operands (or arguments) are addressed. Some operations always use a certain number of operands, and some operations use an unpredictable number of operands, from zero to four. Each operand that is used can be either a "small" (8-bit) integer, a "large" (16-bit) integer, or a variable (8 bits) . (A string is an operand for only one operation code, the print routine.) Following the operation-code byte are the operands. An operation that returns a value has another byte following the operands that tells in what variable to store the value. A predicate operation has another byte or two that tells where to go to, relative to this current instruction, if the predicate is true.

A sub-program is called with a special "call" operation code, whose operands are the address of the sub-program and the (zero to three) arguments or parameters for this call.

Z-code uses a single stack to store both temporary values and return addresses for pending function calls. Z-code doesn't distinguish between addresses and other kinds of data but - because ZIL is a "structured" programming language - the ZIL compiler produces Z-code that uses the stack in a disciplined way, "pushing" and "popping" words in the correct order and not confusing one datum for another.

Storage management:

The address space of a personal computer running ZIP is divided into three parts: impure storage, resident pure storage, and non-resident pure storage. "Impure" means that the data in storage can be modified; impure storage is where all the Things, variables, stack, and so on live. To "save" the game situation just requires writing this part of storage onto a disk or tape, and to "restore" is the opposite; all other parts of the program are pure and unchanging, so they don't need to be saved.

Resident pure storage holds ZIP itself. "Resident" means that the code is loaded from disk at the start of execution and resides in primary memory throughout the game. Non-resident pure storage is paged (physical storage divided into page frames and Z-code program divided into pages, typically with 256 bytes each) and swapped (program pages are kept on disk until needed and then read in to a suitable page frame for interpretation). It may be necessary to swap in a page to read the next sequential byte (if the next byte lies on the next page) or during a go-to, call, return, or print-string Z-code instruction (if the target address lies on another page). Fortunately. the first situation is easy to test for, since the first byte of any page always has zeros for the least significant bits of the address; the second situation occurs infrequently and is easily handled by the parts of ZIP that interpret those instructions. ZIP keeps track of which pages are currently in which frames by using a page table, which holds the first virtual address of each page that is currently swapped in. ZIP must also translate addresses accordlng to which frame currently holds which page.

All the objects used by a Z-code program are set up in their proper locations and formats by the ZIL compiler. There is no need to create new objects while the Z-code program runs, and so there is no need to reclaim storage used by old objects with a Lisp-like garbage collector.

With ZIP needs to know to start running a game are the first address of the three parts of the address space; the first address of Thing storage, global-variable storage, and the vocabulary table used by the "input" Z-code instruction (whence it can find the number and size of vocabulary entries); and the address of the first executable instruction in the game.

Conclusion:

We have described how a large program can be squeezed into a small machine by using a "virtual-machine" approach to encoding the program. Essentially, the designers identified sub-programs within the large program (and within a whole series of similar programs) and chose ways to encode invocations of those sub-programs as compactly as possible. Then duplicate copies of those sub-programs couid be removed, and the whole program could be encoded in the "machine language" of a virtual Z-machine. This technique follows a long line of tradition in computer programming that encourages identifying sub-programs and making them modular and easy to use.

Using this technique, versions of Zork games for personal computers such as Radio Shack TRS-80 and Apple have been developed by Infocom, Inc., P.O. Box 120, Kendall Station, Cambridge, MA 02142. For information about distribution and sales, contact Personal Software, Inc., 1330 Bordeaux Drive, Sunnyvale, CA 94086.

Richard A. Bartle (richard@mud.co.uk)
21st January 1999: htflpism.htm