PunyInform Game Author's Guide

Written by Fredrik Ramsberg and Hugo Labrande

Contents

Chapter 1

Introduction

This document is intended to provide some guidance as well as various tips and tricks for anyone writing a PunyInform game.

The chapters *Testing & Debugging* and *Before Release* are essential reading for anyone who intends to release a PunyInform game.

The *Optimizations* chapter is for when you see the need to make the game smaller and/or faster, typically because you want to make the game fit z3 limitations and/or because you want to provide the best possible user experience.

Chapter 2

Testing & Debugging

Here are some advice on finding and fixing problems in your game.

2.1 Use the debug commands

PunyInform has a nifty set of commands to be used when debugging. Read the docs on these commands at

https://github.com/johanberntsson/PunyInform/wiki/Manual#debugging

and make sure you try them out and understand how to use them. They can be used to teleport to other locations, moving objects to your inventory, checking what's in scope and more. Whenever you're having trouble getting your code to run in `before`, `after` etc, you can use *Actions* and/or *Routines* to figure out which actions are triggered and which user-supplied routines are executed.

2.2 Enable all error checking

Inform 6 has the ability to check for a number of problems at runtime, using Strict error checking mode. Strict error checking is enabled by default when compiling to z5 or z8, but is not available at all for z3. Since it makes Inform code both bigger and slower, we usually recommend game authors to disable it, and this is done on the first line of the game template minimal.inf ("!% -~S"). When you're looking for problems in your code, it's often useful to enable strict error checking (comment out that line) and compile as z5 or z8.

Also, PunyInform has a constant called RUNTIME_ERRORS. whenever you're looking for problems in your code, set this to 2. This enables all checks for errors and prints full information when a problem is detected. If you don't define it, it defaults to 2 when DEBUG is defined.

2.3 Create a command file

Consider saving a list of commands needed to play the game from start to finish. When playing on a modern interpreter for a modern OS, you can type *recording on* to start saving all commands to a file, and *recording off* to stop. To read a command file and execute all commands in it, type *replay*. Note that you need to define `OPTIONAL_EXTENDED_METAVERBS` in your game code to enable these commands.

Having a command file like this makes it easy to check that it's still possible to win the game, whenever you have made changes. You can also save a transcript of the game played with the command file, and then compare a playthrough made at a later date to the original transcript to see what has changed - if you broke something, this should make it fairly easy to spot. On Unix-like OSs you can use `diff` to compare. On Windows you may want to install a specialized program such as WinMerge.

2.4 Testers

Don't think it's enough that you test the game yourself. Good beta-testers are invaluable in the process of producing a good game. They will try things you never thought of, and help you find lots of little things and big things that need to be fixed. You will also notice where they get stuck, so you can decide if you want to provide more hints for the solution to some problem, provide an alternate solution, or somehow make it easier.

Ask testers to provide transcripts of playing sessions so that you can easily see if they interact with the game world as you expect. You can also see opportunities to add or improve responses to non-essential actions.

If you don't have any volunteers for testing, you can ask for help in some forum, such as the one at https://intfiction.org/ .

And of course, make sure you give credit to your testers, as well as others who have somehow helped out with your game.

Chapter 3

Before release

These are some tips you may find helpful when your game can be played from beginning to end, and you feel it's soon ready to be released to the public.

3.1   Create an IFID

There's a standard for identifying text adventures, and it's part of The Treaty of Babel (See https://babel.ifarchive.org/). Each game gets an IFID - a unique identifier which can be used to look up data about the game. It's a good idea to include an IFID in your PunyInform game. Somewhere in your source code, you write a section like this:

```
Array UUID_ARRAY string "UUID://XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX//";
#Ifdef UUID_ARRAY;#Endif;
```

Instead of all the Xs you put your unqique identifier consisting of the characters 0-9 and A-F, which you obtain from https://www.tads.org/ifidgen/ifidgen .

The IFID remains the same when you release updated versions of your game. If the game is ported to a new system (say from PunyInform to Twine or Inform 7), it gets a new IFID. Also, if the game is translated to another language (say French), it gets a new IFID as well.

3.2   Set Release and Serial

When you start developing a game, you don't need to set the Release number and Serial number. They will get reasonable defaults. As you release a game, you want to have control over these constants, as this helps identifying which version of the game is available at a certain web site, or which version a certain player is running when they encounter a bug. So, you add something like this near the top of your source code file:

```
Release 1;
Serial "210131";
```

The recommended way to use the release number is to set it to 1 for the initial release, then increase it by one for each new release you make. The serial number is typically set to the date when the release is made, in the format YYMMDD.

3.3   Check limits

PunyInform has a number of limits which have been set to reasonable values, but some games will need to raise some of these limits. Read about these limits at

https://github.com/johanberntsson/PunyInform/wiki/Manual#parameters

The limits which you have to be particulary careful with are:

MAX_TIMERS (default 32): If there is any chance that there could be more than 32 timers or daemons active at the same time, raise this limit. If, on the other hand, you know that you only have say 3 daemons and no timers in your game, you can set MAX_TIMERS to 3 to save a bit of dynamic memory. Also search through any extensions you use to check that they're not using timers or daemons before you start lowering this limit.

MAX_SCOPE (default 50): How many objects can be in scope at the same time. Imagine the player picking up all movable objects and placing them in the location with the most static objects. Add any actors, their possessions, the player's body parts if any, etc. MAX_SCOPE should be higher than this number of objects. If there's a situation in the game where MAX_SCOPE is too low for all objects that should be in scope, some objects will quietly be ignored, meaning some objects can't be referenced and if they have an each_turn routine it won't be executed. If the game has been compiled in debug mode, an error message is printed when this happens.

3.4   Check articles

PunyInform has a simple mechanism for printing the indefinite article of an object:

1. If the object has the `proper` attribute, its name is a proper name (like "John") and it has no article.
2. If the object has the `pluralname` attribute, the article is "some".
3. Otherwise, the article is "a".

This works well for most objects. However, sometimes you want to use a different article, such as "an" or "a bunch of" (Of course "a bunch of" isn't an article, strictly speaking, but we can use it as one in PunyInform.). To do this, you add the `article` property to the object, and give it a string or routine as its value.

7

As object names are more often printed with their definite article, it's easy to miss that some objects may have the wrong indefinite article. Before you release your game, make sure you go through all your objects and check the articles. In particular:

- Check that objects which start with a vowel sound (like airplane, egg and umbrella but not unicorn) have article "an".

- Check that plural objects either sound fine with the article "some", or have another article specified. I.e. trousers may have article "a pair of", bees may have article "a swarm of" etc.

If you want to see the article of an object in action, compile the game in debug mode, *purloin* the object and check your inventory.

3.5   Turn off DEBUG

While the DEBUG mode is invaluable during development, make sure you turn it off when compiling a game for release, or it will allow players to cheat, plus it looks like a rather sloppy release. Note that when the game is compiled in DEBUG mode, a "D" is printed after the library verion when the game starts, like "PunyInform v4.4 D".

Additionally, it's generally advicable to set `RUNTIME_ERRORS` to 0 when making a proper release. This means the library will keep quiet about problems it spots which don't make the game crash. This makes the game file smaller and stops the library from breaking player immersion with complaints about programming errors.

3.6   Convert your Zcode file to HTML

The Parchment HTML Converter at https://iplayif.com/api/sitegen allows you to convert Zcode to HTML, allowing players to play your game in a web browser. This process can be automated using `curl` and its REST API from command line, like this:

```
curl -o output.html -F "story_file=@mygame.z5" \
https://iplayif.com/api/sitegen
```

Chapter 4

Optimizations

PunyInform leaves you with about 100k bytes to write your game (if you're using the z3 format), but sometimes it's not enough. Maybe your ambitious game "almost fits" in the z3 format; maybe you'd like your grand epic to be playable on a single 1541 disk for the C64; maybe you could fit it all on a 130kb disk for the Atari 8bit. But even if you don't get in a situation where you have to make optimizations, you may want to anyway - a shorter game will play smoother on a machine with little memory, like most 8-bit computers. In any case, we're here to help! This chapter will give you some helpful tips and tricks to make your game smaller. Depending on how much time and energy you can spend, you could save up to an extra 10kb on your game file size!

4.1 Things that save hundreds of bytes

4.1.1 Use abbreviations properly

Abbreviations are fixed strings that get replaced by a 10-bit long code in order to save space in the text. You just need to declare them, and the compiler will apply them whenever possible, if you compile with the `-e` switch. A topic of particular interest is which abbreviations to use.

Infocom used the full 96 abbreviations one can declare in the Z-machine. Puny-Inform ships with 64 abbreviations, which have been picked based on the text in the library files. This saves some space, but these generic abbreviations will not capture the fact that your protagonist's name, "Eyjafjallajökull", could be declared as abbreviated text. A good set of abbreviations is uniquely tailored to your game. Inform's compiler has a switch, `-u`, that looks at your game's text and finds 64 custom abbreviations; it usually gives better results. To have Inform come up with the best set of abbreviations, compile the game with the `-u` switch, and redirect output to a text file, like this:

```
inform6 +lib mygame.inf -v3u > abbreviations.txt
```

Then open the produced text file and scroll to the bottom. Copy all the lines at the end of the file beginning with `Abbreviate`, and paste them at the beginning of your source code file, right after the lines at the top with compiler directives. As an alternative, you can put them in a separate file which you `Include` in your main source code file. Also, make sure you have the line `Constant CUSTOM_ABBREVIATIONS;` in your source, before including globals.h, or your new abbreviations won't be used.

Very recently, interest in the algorithmics of the problem have led to the creation of tools to compute even better abbreviations. Henrik Åsman's program and Matthew Russotto's program aim to compute 96 abbreviations with efficient implementations; a slightly less efficient program in Python is the one written by Hugo Labrande. Both Åsman's and Labrande's programs can output any number of abbreviations in Inform's expected format and can handle Inform 6's newest gametext format, introduced in version 6.35. Currently PunyInform requires version 6.36, so these program are useful for all PunyInform developers.

You can now declare up to 96 abbreviations (if you don't declare any "low strings" - that is, set the compiler switches MAX_ABBREVS to 96 and MAX_DYNAMIC_STRINGS to 0), which saves even more space. You could expect savings of up to 7kb on a 128kb file!

4.1.2   Omit unused routines

The Inform 6 compiler switch $OMIT_UNUSED_ROUTINES is off by default, but can be turned on by putting it in the compilation flags. This frees up memory at no cost, and can save several hundred, maybe even thousands, of bytes! This switch is set at the start of the file `minimal.inf` so if you base you game on that file, you have this covered.

4.1.3   Turn off strict error checking

By default, the Inform compiler adds code to every z5 or z8 game to check for a number of problems in your code at runtime. This is a useful and nice feature, but it makes the game slower and ~10 KB larger, so you probably want to turn it off when targeting 8-bit computers. You do this with `-~S`. This is done in the first line in `minimal.inf`.

Just keep in mind that this mechanism exists, and if you get weird errors or crashes you may want to enable it for testing. Note that you'll need to compile as z5 or z8 for it to work.

4.2 Things that save dozens of bytes

4.2.1 Use string constants

If you have a string of over 10 characters repeated somewhere in your code, you could declare that string to be a constant, then point every instance of it to the constant instead. Something like

```
Constant MSG_LOOKS_DANGEROUS = "Going in that direction looks dangerous.";
Constant MSG_HAM " braised ham with mashed potatoes and green beans";
```

```
Object Pub "Pub"
  with
    description [;
        "You're in the pub. Dark doorways lead north and west.
            On the menu today:", (string) MSG_HAM, ".";
    ],
    before [;
        OrderFood:
            "You decide to order the", (string) MSG_HAM,
                ". Yummy, that was delicious!";
    ],
    n_to MSG_LOOKS_DANGEROUS,
    w_to MSG_LOOKS_DANGEROUS,
    s_to Street,
  has light;
```

Depending on the size of the text fragment that's repeated, you could save anywhere from a few bytes to a few hundred bytes. Using this technique repeatedly can yield savings of a few kilobytes in a long game, at the expense of making your code a bit less readable; just make sure you use explicit constant names.

One useful trick to identify such fragments quickly is to export the game's text into a file (`-r` switch in I6's compiler), then sort the lines alphabetically. You might then be able to identify identical strings, or strings whose beginning have a lot in common (which is great: all things being equal, replacing a prefix or a suffix by a constant is slightly better than replacing text in the middle of a string, since that's 2 `print` opcodes vs 3). You can also tweak your text so very similar sentences end up being the same ("There is no power on the island" vs "There isn't any electricity on the island").

4.2.2 Replace a switch with an array

If you have a large conditional switch statement for which the consequences are of the same format (they're all a print, or adding something to the same variable,

etc), you can turn this into a simple table lookup. Construct an array with the changing values, and use a–>var to access them. So instead of:

```
switch(i){
  1: print "We are the champions";
  2: print "We will rock you";
```

use this:

```
Array songs "We are the champions" "We will rock you" ...
print (string) songs-->i;
```

This is the opposite of the advice under **2b** in section 45 of the DM4, but that example is for when you want to save on readable memory (which can be no more than 64 KB) by not declaring too many arrays, and you're willing to pay the cost to transform it into a routine.

4.2.3   Compare a value to multiple values

When writing complicated conditions featuring comparing one variable to multiple things (dictionary word, in the case of a parse_name, for instance), always group these comparisons using 'or', as follows:

```
if w=='sea' or 'ocean' or 'atlantic' or ....
```

The Z-machine has an opcode to perform such comparisons by groups of three, which the Inform compiler utilizes to generate shorter code.

4.2.4   Use Simple Doors

If you're using more than four doors, you can save space by using `OPTIONAL_SIMPLE_DOORS`. As a bonus, the code gets shorter and more legible. Read more at https://github.com/johanberntsson/PunyInform/wiki/Manual#doors .

4.2.5   Set RUNTIME_ERRORS to 0

RUNTIME_ERRORS has three settings:

- 0: Perform a bare minimum of error checking. If there's a problem, just print the error number.
- 1: Perform full error checks. If there's a problem, just print the error number.
- 2: Perform full error checks. If there's a problem, print a suitable error message.

When compiling with DEBUG enabled, setting 2 is the default. When DEBUG isn't enabled, setting 1 is the default.

In a production build, when the code has been thoroughly tested, you may want to set RUNTIME_ERRORS to 0. This helps make the game file smaller, and

the reduced checks also make it faster. If you still want all error checks, but skip the explanatory error messages, you can set it to 1.

4.3 Things that save several bytes

4.3.1 Avoid conditions in mathematical expressions

Inform has support for evaluating conditions as part of a mathematical expression, as in `return (Lamp has light);` or `danger = (child(elevator) ~= 0)`. However, this requires Inform to generate some complex code, so try not to use it - Write explicit `if` statements instead.

```
return (player == werewolf);
```

will not be as space-efficient as

```
if (player == werewolf) {
    return true;
} else {
    return false;
}
```

4.3.2 Compare to zero

The Z-Machine has an opcode for "test if zero" or "test if non-zero". If you know that a variable is either true (== 1) or false (== 0), it's faster and shorter to compare the variable to false than to true. So instead of:

```
if(x == true) print "The squirrel is happy".;
```

use:

```
if(x ~= false) print "The squirrel is happy".;
```

or just:

```
if(x) print "The squirrel is happy".;
```

4.3.3 Use the random() statement

If you want to print a text at random, or return a value at random, don't forget that the random() statement can take any number of arguments and return one of them with equal probability. No need for a switch or an if to simulate a weighted dice: random(1,2,3,4,6,6) will give you the value you need.

4.3.4 [Z3-specific] Use the low dictionary resolution to your advantage

The z3 format has a resolution of 6 characters; that is, every single word is identified by its first 6 characters. The compiler will replace every word with

the suitable dictionary value, which means

```
if(w == 'insect' or 'insects')
```

literally tests the same thing twice. Remove any useless test to save a few bytes every time. This could also help with fitting everything in a "name" property, instead of having to write a "parse_name" routine, which is costly.

4.4   Other optimizations

4.4.1   Use Manual Scope

This is an optimization for speed only. "Scope" means which objects the player, or another actor, can refer to. By default, the PunyInform library will assume that what's in scope changes whenever a user-supplied routine is called, and this may happen a lot. This causes the library to recalculate the scope quite often, and this makes the game slower, particularly in situations where a lot of objects are in scope.

If you want to improve on this situation, you can define the constant `OPTIONAL_MANUAL_SCOPE`. This means you take responsibility for telling the library when you have done something that might affect the scope. As a general rule, set `scope_modified` to `true` whenever you use `move` or `remove` or you change any of the attributes `open`, `transparent` or `light`. However, if the object affected is nowhere near the player you don't need to set `scope_modified`. All library routines that move the player or move or modify objects, like `OpenSub()` and `PlayerTo()`, already set `scope_modified` as needed.

If you don't want to worry about manual scope while programming, you can just wait until you're ready to release the game, add `Constant OPTIONAL_MANUAL_SCOPE;` at the beginning of your program, then search for `move`, `remove` and `give` in your code and add `scope_modified = true;` as appropriate.

Example of code that needs `scope_modified = true`:

```
Constant OPTIONAL_MANUAL_SCOPE;

Object Button "button"
  with
    name 'button',
    after [;
      Push:
        move Puppy to location;
        remove self;
        scope_modified = true;
        "A loud click is heard, a puppy comes running into the room,
          and the button sinks into the table, becoming invisible.";
```

14

```
    ],
  has static;
```

4.4.2   Consider using Manual Scope Boost

This is another optimization for speed. In short, you define `OPTIONAL_MANUAL_SCOPE_BOOST` and the library tries to avoid looking for `react_before`, `react_after` and `each_turn` routines to run, when there weren't any routines of a certain type last turn, and what's in scope hasn't changed since then. See the full documentation for this feature at:

https://github.com/johanberntsson/PunyInform/wiki/Manual#manual-scope-boost

4.4.3   Use manual setting of reactive attribute

This is an optimization you can perform to make your game start faster. Unless you have done this, PunyInform will look through all your objects when the game starts, and set the `reactive` attribute on all objects that provide `react_before`, `react_after` or `each_turn`. When the game is running, only objects that have this attribute are considered when checking for these properties, for reasons of speed. While letting the library set this attributes automatically works well, it means there's an extra pause as the game starts. For a large game, this could take a few seconds on an 8-bit computer.

These are the steps you need to take to set the attribute manually instead:

1. Define the constant `OPTIONAL_MANUAL_REACTIVE`
2. Compile the game in Debug mode, start it and type the command "DEBUG REACTIVE"
3. For every object which is reported as needing the `reactive` attribute, add the attribute in the source code for that object. Typically, you can skip it for the player object, unless you have added an each_turn routine to it.

Note: If you use `parse_name` quite sparingly (up to about 10% of all objects the player can refer to), you probably want to define `OPTIONAL_REACTIVE_PARSE_NAME` as well. If you do this, and you define `OPTIONAL_MANUAL_REACTIVE`, you will also need to set the `reactive` attribute for all objects that provide the `parse_name` property.

4.4.4   Move arrays to static memory

If you have arrays whose contents never change, you can place them in static memory, like this:

```
Array my_array static --> 1 2 3 4 5 "String1" "String2";
```

This makes dynamic memory smaller, which means save and restore get faster. Since static memory can also be swapped out, it means gameplay can be smoother

in sections of the game where the arrays aren't used.