Game Author's Guide

1 Introduction

This document is intended to provide some guidance for anyone aiming to release a PunyInform game. While you can read it any time, it's probably the most useful when the game is nearing completion.

2 Before releasing a PunyInform game

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.

2.1 General

Here's a list of things you should always check before releasing a game.

2.1.1 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 line 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.

2.1.2 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.

2.1.3 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. 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.

2.1.4 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 v1.9 D".

2.1.5 Set RUNTIME_ERRORS

In a production build, when the code has been thoroughly tested, you may want to set RUNTIME_ERRORS to 0. This means the library will only check for a bare minimum of error conditions, and if an error occurs only the error code will be printed, not an explantion of what the error means. This helps make the game file smaller, and the reduced checks also makes it faster. If you still want all error checks, but skip the explanatory error messages, you can set it to 1. This is the default when building without the DEBUG mode. When in DEBUG mode, the default setting is 2, meaning that full error messages are also printed.

2.2 Testing

Here are some advice on testing your game.

2.2.1 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.

2.2.2 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*.

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.

2.2.3 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 find 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.

2.2.4 Check limits

PunyInform has a number of limits which have been set to reasonable values, but some games will need to raise certain limits. Read the paragraph on Parameters under 'Customizing the library' at

https://github.com/johanberntsson/PunyInform/wiki/Manual#customizing-the-library

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 32): 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. If the game has been compiled in debug mode, an error message will instead be printed.

2.3 Optimizations

As your game grows, you may find that you have to make optimizations or it won't fit in the desired Z-code version size constraint, or it won't fit on a disk

for a certain 8-bit computer that you want to release it for. But even if you don't get in the situation that 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.

2.3.1 Abbreviations

Inform has two means of text compression built-in. The first is that each character in a strings is reduced to one or more five-bit codes, and three codes are put into two bytes. Lowercase letters and space use a single five-bit code each, so for the most part, three characters fit into two bytes. This kind of compression is always in use.

The second means of text compression is abbreviations. You can define up to 64 abbreviations to be used in a story file. When you compile, you add the flag `-e` to say that you want "economy mode", which means enable the use of abbreviations. Each abbreviation takes up two five-bit codes, regardless of its length.

PunyInform has a set of abbreviations which provide a good starting point, but they are only based on the text in the library. As you add text to your game, these abbreviations will be less and less relevant. To have Inform come up with the best set of abbreviations, compile the game with the -u flag, 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 the last 64 lines of the file, 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.

To get even better abbreviations, you can use Hugo Labrande's Python script, available at https://github.com/hlabrand/retro-scripts .

Using Inform's mechanism to find the best abbreviations can typically make the story file about 10% smaller. Using Labrande's script can increase that by about 1%, so if you managed to save 10% with Inform's abbreviations you can save ~11% with Labrande's abbreviations.

2.3.2 Reusing string values

If you write the same string in two different places in the source code, it will be created in two different places in the final story file, and use twice as much memory as if you had only written it once. If you use the same string several times in the source code, you can instead put it in a constant. If you print the value, you need to put `(string)` before it. If you would otherwise write it as a

string in code, meaning “print this string, print a newline and then return true”, you must add `print_ret (string)` before it. Example:

```
Object Hallway "Moor"
  with
    description "You're on a moor.",
    n_to "It would be foolish to wander off in that direction.",
    w_to "It would be foolish to wander off in that direction.",
    e_to [;
      if(self hasnt general) {
        print "It would be foolish to wander off in that direction.";
        give self general;
        " You hesitate.";
      }
      return OutsideCottage;
    ],
    s_to [;
      if(PaperMap in player)
        return Castle;
      "It would be foolish to wander off in that direction.";
    ],
  has light;
```

can be transformed into this:

```
Constant REPLY_FOOLISH_WANDER =
  "It would be foolish to wander off in that direction.";

Object Hallway "Moor"
  with
    description "You're on a moor.",
    n_to REPLY_FOOLISH_WANDER,
    w_to REPLY_FOOLISH_WANDER,
    e_to [;
      if(self hasnt general) {
        print (string) REPLY_FOOLISH_WANDER;
        give self general;
        " You hesitate.";
      }
      return OutsideCottage;
    ],
    s_to [;
      if(PaperMap in player)
        return Castle;
      print_ret (string) REPLY_FOOLISH_WANDER;
    ],
  has light;
```

2.3.3 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 .

2.3.4 Manual scope

This is not an optimization for size, but for speed. "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, particulary 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 give or remove any of the attributes `open`, `transparent` or `light`.

Example:

```
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.";
    ],
  has static;
```