GRL language

A.T. Hofkamp

Compiled at December 30, 2007

Contents

Chapter 1

Introduction

1.1 Goal

The goal of the GRL language is to allow specification of user-defined extensions to the Transport Tycoon[1] games in a human-readable form. Another language used for this purpose is the NFO[2] language.

A specification in the NFO language is a numbered list of real sprites and pseudo sprites, each defined by a length (in bytes), and a sequence of bytes, words, and double values. The languages also has support for strings, and Unicode characters, and it has a number of extra escape sequences, for example for specifying tests. While the abstraction level of the language is quite low, it is extremely flexible. You can express anything that can be encoded as a sequence of bytes.

The GRL language has a different starting point, it assumes that a user (a graphic artist) should specify what he/she wants in a (to him/her) logical and compact way, and the computer should encode that information into bytes, and make it understandable to the tycoon application.

Consequences of this idea are

- 'Trivial' computations such as lengths, distances, and counts are much better left to a computer (this is why NFOrenum exists).
- The user should decide on the order of the (pseudo) sprites in the file, and specify values for fields in them.
- Since humans are much better at text processing, fields and information should be specified in text-form when possible.
- If you need to specify a numeric value, you may either use decimal or hexadecimal numbers, whatever is most appropriate to you.
- The computer should do what it does best, namely collecting the field information, checking whether it makes sense (or else report something to the user), and encode the data to correct (pseudo) sprite(s).

The current GRL language is a small step towards that goal. The main difference is that a (pseudo) sprite is defined not as a sequence of bytes, but as a collection of key-value pairs. The

[1] Programmed by Chris Sawyer, released by MicroProse.
[2] Developed by Josef Drexler, see http://www.ttdpatch.net/grfcodec/.

keys of the pairs are the names of the fields that need to be given a value in the (pseudo) sprite, and their allowed values are allowed values of the fields.

To provide an easy way for extensions, a key-value pair always accepts a number as its value, even if a name can be used. This improves the chance that you will be able to specify what you intend, even if the `GRL` program is not ready for it. In addition, the program is liberal in accepting values. It will complain when it doesn't understand something, but it will continue producing an output file, thus allowing for generating files with features that the program does not entirely understands. The final fall-back mechanism is the `grl-bytes` primitive, which basically drops you back to the `NFO` language. With this primitive you can specify anything, although you probably never want to use it (and if all is well, you will never have to).

1.2 The `GRL` language

`GRL` stands for *GRF Language*, it is a way of specifying how your sprites and settings should be used in the transport tycoon games.

The entire language is built on the idea of sets of key-value pairs, for example,

```
sprite {
    filename = "c:\myfile.pcx", x-pos = 50, y-pos = 50,
    x-size = 18, y-size = 10, x-rel = -1, y-rel = 5,
    compression = color-0-transparent
}
```

Above, eight '*key* = *value*' pairs (*fields*) specify all data of a real sprite. This is also indicated by the word '`sprite`' at the first line. The curly brackets around the key-value pairs indicate that these pairs all belong together to form the specification of one real sprite. Such a set of key-value pairs, together with the first word and the curly brackets is called a *node*.

In general, the order of the fields is not relevant. You can use any order that you like, as long as you specify all data. In the same way, layout of a node is also not important. You can put the opening curly bracket at left of the next line, or put the entire sprite definition on one line.

You can add a line comment by means of the '`//`' sequence, also available in the NFO language. All text after the sequence, up to the end of the line, is considered to be comment, and skipped.

You can make a list of nodes by separating them with a comma (','). Also, they can be nested in each other and they can be labeled, for example

```
sprite-sets {
    feature = cargo,

    mail-cargo:
    collection {
        sprite { ..... },
        sprite { ..... }
    },

    passenger-cargo:
    collection {
        sprite { ..... },
        sprite { ..... }
    }
}
```

Note the comma after the closing bracket of both first `sprite` nodes, and the first `collection` node.

The above is a NFO Action1, where you can define a list of collections of sprites. At the innermost level, you see the `sprite` nodes (without the key-value pairs for brevity of the example). Several sprites are packed together into a `collection`. In front of each collection is a label (the words `mail-cargo` and `passengers-cargo`). The colon after the word indicates that it is a label rather than the start of another node. Unlike the key-value fields, the order of the nodes does matter here. `grl` orders the node from top to bottom, that is, the collection labeled `mail-cargo` is the first collection (with offset 0).

Both collections, and a 'feature' key-value pair are put in a `sprite-sets` node, which represents aNFO Action1 primitive together with the real sprites that should be directly below it.

The labels are used to refer to the labeled node from elsewhere in the `GRL` file, for example, the `mail-cargo` label can be used in a cargo NFO Action2 shown below

```
cargo-graphics {
    cargo-id = 75,
    use-collection = mail-cargo
}
```

Other uses of the labels are referring to NFO Action2 primitives from NFO Action3 primitives, and as destination label for conditional jumps (NFO Action7 and NFO Action9).

A `GRL` specification at the outermost level consists of a comma-separated list of nodes (with labels for conditional jump destinations).

1.3 Values

The most basic value in `GRL` is the unsigned number, a sequence of digits, like '`124`'. If you want to enter a hexadecimal number, use '0x' as prefix, as in '`0xa0`'. For negative values, you can add a dash '-' in front of an unsigned number. You can do this however only with keys that expect signed numbers.

For a string (a sequence of characters, terminated with a NULL-byte), you can use double quotes. The percent ('%') character is used as escape character, since it is not used often, and it allows use of both forward slashes and back-slashes (most often used in file-names) without escaping. A few examples are shown in the table below

String	**Codes (hexadecimal)**
`"abc/de" "f\ghi"`	61 62 63 2f 64 65 66 5c 67 68 69 00
`"%{percent}, %{quote}, %{0x0d} %{10}"`	25 2c 20 22 2c 20 0d 0a 00

The first example demonstrates that multiple strings after each other are silently concatenated to form one big string first. This is useful for nicely formatting of long strings. Also, it shows that both forward slashes and back-slashes need not be escaped. The second example shows the two forms of escape sequences that exist in `GRL` strings. After the percent sign, there is either a word or a number, surrounded by curly brackets. The shown escape sequences will always work, at some points, the `GRL` encoder program allows some additional sequences (in particular, while reporting error messages with the '`message`' primitive NFO ActionB). Last but not least, note that the terminating NULL-byte is always automatically added when you use a string.

For a value that is in fact an encoded ASCII character, you can use single quotes around the character (demonstrated below in the example).

The last category of values are values longer than four bytes, or values that are more a sequence than a single large value, for example the identification key-value pair in `grl-id` (NFO Action8). In such cases, you can use a byte sequence to state its value, for example

```
grl-id { identification = [ 'A', 'H', 1, 3 ],
         name = "A GRL example name",
         description = "%{0xXX} 2007, A.T. Hofkamp%{0x0d}%{0x0a}"
}
```

In a limited number of cases, you can also state a value by means of a name, for example '`feature = train`'. The word '`train`' here represents the value 0. These names are not hard-coded in the program but loaded from tables (currently all collected in a separate `tables.grl` file).

Unfortunately, support of this feature is currently limited to simple cases. For a value that is in fact a collection of flags (such as the '`compression`' key-value pair in a real sprite), you can specify a combination of words by separating them with a '+' sign, as in

```
compression = color-0-transparent + store-compressed
```

Like the use of words to express values, support for the use of a combination of words to express a number of separate flags is also quite limited currently.

Chapter 2

Reference

In this chapter, a more detailed explanation is given of all the `GRL` primitives. Each section explains one primitive. There is a close relation with the NFO actions, and you are advised to also read the corresponding NFO documentation. Often a `.html` page of the `NewGraphicsSpec` is listed in the section as a reference.

Below is a 'reverse' table where you can find how each NFO action is available in terms of `GRL` primitives.

NFO action	`GRL` **primitive**	**Description**
Action0	define-properties	Set various properties of objects, see Section 2.3
Action1	sprite-sets	Collections of real sprites, see Section 2.4
Action2	vehicle-graphics	Train, road, ship, and plane graphics, see Section 2.5
	station-graphics	Station graphics, see Section 2.7
	building-graphics	House and industry-tile graphics, see Section 2.6
	industry-callback	Handle production callback, see Section 2.8
	cargo-graphics	Cargo graphics, see Section 2.9
	select-graphics	Variational Action2, see Section 2.10
Action3	link-graphics	Link graphics to game objects, see Section 2.11
Action4	define-text	Set generic strings, see Section 2.12
	define-vehicle-name	Set the name of vehicles, see Section 2.12
Action5	-	Currently not implemented
Action6	-	Currently not implemented
Action7	cond	Conditional jump, see Section 2.2
Action8	grl-id	Identification, see Section 2.1
Action9	init-cond	Initial conditional jump, see Section 2.2
ActionA	-	Currently not implemented
ActionB	message	Generate a message, see Section 2.13
ActionC	-	Not useful
ActionD	-	Currently not implemented
ActionE	-	Currently not implemented
ActionF	-	Currently not implemented
Action10	-	Currently not implemented
Action11	-	Currently not implemented
Action12	-	Currently not implemented
Action13	-	Currently not implemented
-	grl-bytes	A sequence of bytes as pseudo-sprite, see Section 2.14

2.1 Identification

The identification of the GRF file is done with the grl-id primitive. An example

```
grl-id {
    version = 6,
    id = [ 'A', 'H', 3, 1 ],
    name = "Short name",
    description = "Longer description with author name(s)%{0x0d}%{0x0a}"
}
```

See NewGraphicsSpec:Action8.html for further details.

2.2 Conditional jumps

Conditional jumps or exits (jumps to the end of the file) in the GRF file are defined using the init-cond (conditional jump during initialization), or the cond ('normal' conditional jump) primitives. An example

```
cond {
    variable = climate, test = equal, value = toy-land,
    var-size = 1, destination = dest-label
}
```

In the condition_variable_table table, the available variable names are listed. The allowed names for the tests are listed in the condition_test_table table, and in the condition_value_table, you can find the list of available values. If you use a name as variable rather than its equivalent number, the size of the value is also available, so you do not need to give var-size = 1. Finally, dest-label is the destination that is jumped to, if the test holds. It should be a label of a sprite at the outermost level further down the file. If you want to skip the remainder of the file, use the special label exit. For further details on conditional jumping, see NewGraphicsSpec:Action7.html.

2.3 Define properties

With the define-properties command, you can set many of the properties of vehicles, etc. For example,

```
define-properties {
    feature = industry,
    id-values {
        id = 0,
        new-value { property = substitute-industry-type, value = 0x01 },
        new-value { property = industry-type-override, value = 0x1b }
    }
}
```

The feature field defines for which feature you are setting the properties. Each id-values node defines new values for one (vehicle) id. You can add more id-values nodes, each time the id field should be incremented by one, and the same properties should be set.

Inside a new-value node, the property field sets which property should be changed, the value field defines the new value. Its actual value depends on which property you are changing.

Finally, the `size` field defines how large the value is in bytes. The latter is not shown here, since in many cases, the value of the `size` field can be deduced from the `property` field, for example, when you set the introduction year, it is known that the size is one byte. In those cases, you can omit the `size` field. See also NewGraphicsSpec:Action0.html for more details.

2.4 Defining sprite collections

With the `sprite-sets` primitive, you can define a number of collections containing (the same number of) sprites. An example with one collection containing one real sprite is

```
sprite-sets {
    feature = cargo,
    mail-cargo: collection {
        sprite {
            filename = sprites/cargo.pcx",
            compression = color-0-transparent,
            x-pos = 322, y-pos = 8,
            y-size = 55, x-size = 64, x-rel = -31, y-rel = -24
        }
    }
}
```

You can have more than one sprite in one collection. Just add another one, separating both with a comma. Also, you can have more than one collection of sprites (again, just add another one, separating them from each other with a comma). All collections must have the same number of sprites. The names allowed at the `compression` key-value pair are listed in the compression_table table. The *mail-cargo* label is not really necessary, except that you probably want to refer to this collection in a later `cargo-graphics` primitive. See also NewGraphicsSpec:Action1.html for more details.

2.5 Vehicle graphics

Defining graphics for vehicles (train, road, ship, plane) is done with the `vehicle-graphics` primitive. For example,

```
vehicle-graphics {
    feature = train, cargo-id = 5,
    move-types = [ move-label ],
    load-types = [ loading-empty, loading-full ]
}
```

This defines the vehicle graphics for a train (with ID being 5). The `move-types` and `load-types` fields contain a list of labels that refer to sprite collections from the last `sprite-sets`. See also NewGraphicsSpec:Action2Vehicles.html for more details.

2.6 Building graphics

The graphics for houses and industry-tiles are defined using the `building-graphics` primitive. For example

```
building-graphics {
    feature = industry-tile,
    cargo-id = 3,
    ground { .... },
    building { .... }
}
```

The `feature` field defines for which kind of building you are defining graphics. Correct values are `house` (number 7), and `industry-tile` (number 9). The `cargo-id` field defines which ID this action has. Next is a `ground` node, and (in this case) one `building` node. For the early building stages, you may want to omit the `building` node. Also, you can have several `building` nodes which together form the building.

2.6.1 Ground and building nodes

The `ground` node looks like

```
ground {
    use-sprite = ground-sprite,
    draw-type = normal,
    // color-translation = concrete,    // if 'recolor' is used
    // sprite-from-action-1 // normally implied by use of a label
}
```

A `ground` node contains information about the sprite to draw for the ground. The `use-sprite` value `ground-sprite` is a label that refers to a previously defined collection of sprites in a `sprite-sets` node. Alternatively, you may use a number to refer to one of the built-in sprites. The `draw-type` defines how to draw the sprite. Besides the value '`normal`', you can also use '`transparent`', and ' `recolor`'. If you set `draw-type = recolor`, you can set the color translation table with the `color-translation` field. You can either use a numeric value, or use a name. The program does a default action when you do not specify a color translation. See NewGraphicsSpec:Action2HousesIndustryTiles.html for more details. The `sprite-from-action-1` field is a boolean flag denoting that the sprite specified in the `use-sprite` field is a sprite from this file rather than a built-in sprite. Normally this is set implicitly by using a label in the `use-sprite` value.

`building` nodes are similar to `ground` nodes. See below for an example

```
building {
    use-sprite = building-sprite,
    draw-type = normal,
    // color-translation = yellow,    // if 'recolor' is used
    // always-draw-normal,
    // sprite-from-action-1, // normally implied by label
    x-extent = 16, y-extent = 16, z-extent = 25,
    x-offset = 0, y-offset = 0, z-offset = 0
    // use-bounding-box-building
    // relative-offset
}
```

The first extension is the boolean flag `always-draw-normal`. You can set this flag to prevent the program from drawing the sprite in transparent mode. See NewGraphicsSpec:Action2HousesIndustryTiles.html

for more details. The second extension is that you have to specify a bounding box and a relative position of the building sprite. The `x-extent`, `y-extent`, and `z-extent` fields define the bounding box, and the offset is specified with the `x-offset`, `y-offset`, and `z-offset` fields.

For the second and further building nodes, the offsets are all relative to the ground sprite. If you want them relative to the previous building node, add the `relative-offset` flag. The `use-bounding-box-building` flag is for forcing that a building sprite with a new bounding box is used rather than more compact form that contains only the x and y offsets relative to the previous building.

2.7 Station graphics

To define graphics for stations, you should use the `station-graphics` primitive, for example

```
station-graphics {
    cargo-id = 6,
    little-sets = [ little-cargo ],
    lots-sets = [ many-cargo ]
}
```

This defines graphics for a station. Its ID number is 6, there is one sprite collection for drawing small amounts of cargo namely the collection that `little-cargo` refers to, and one sprite collection for rendering large amounts of cargo namely the sprite collection that the `many-cargo` label refers to. If you leave the `little-lots-treshold` at 0 (or set it to 0), you can leave out the `little-sets` key-value pair. See also the NewGraphicsSpec:Action2Stations.html for more details.

2.8 Industry callback

`industry-callback` is currently not implemented!

With an industry callback, you can define how the industry process its goods. There are two versions available, '`version = 0`' with fixed numbers, and '`version = 1`' that uses registers. An example of an industry callback with numbers is shown below

```
industry-callback {
    id = 18, version = 0,
    subtract-in-1 = 100,
    subtract-in-2 = 200,
    subtract-in-3 = 300,
    add-out-1 = 50,
    add-out-2 = 75,
    again = 1
}
```

The version with registers has the same fields, but instead of the fields `subtract-in-1`, `subtract-in-2`, `subtract-in-3`, `add-out-1`, `add-out-2`, and `again` getting a numeric constant of amounts of cargo, they state the number of a register. See also NewGraphicsSpec:Action2Industries.html.

2.9 Cargo graphics

The graphics used for drawing cargo are set with something like

```
cargo-graphics {
    cargo-id = 38, collection = mail-cargo
}
```

The ID is here 38, and the sprite collection to use is labeled `mail-cargo`. Since there is always exactly one collection needed, there are no list brackets around the label. See NewGraphicsSpec:Action2Cargos.html for more details.

2.10 Select graphics

For a dynamical way of setting the graphics, you can use the `select-graphics` command. It allows you to select between several alternatives, depending on the value of a variable. For example

```
select-graphics {
    feature = train,
    cargo-id = 29,
    type = object-byte,
    variable {
        name = year, // parnum = num,
        shift-right = 0, and = 0xffff
    },
    case { minimal = 0, maximal = 30, use-cargo-id = graph1 },
    case { minimal = 31, maximal = 80, use-cargo-id = graph2 },
    default-case = graph3
}
```

The `select-graphics` starts with the `feature` field for which you define the graphics. The `cargo-id` field define the ID number for this node. The `type` field defines the type of data. The name of its value consists of two parts. The first part is either `object` or `related`, the second part states the size, and is `byte`, `word`, or `double`.

Next comes the variable that you use to base your decisions on. The simple case is shown above. In the `variable` node, you give the name of the variable (depends both on the value of the `feature` field and the value of the `type` field). If the variable is in the `60+x` range, an additional parameter must be given with the `parnum` field. The values of the `right-shift` and `and` fields manipulate the variable (by shifting, and and-ing with a mask). In addition, you may add a value (specified with for example `add = 21`) together with either a `divide` or a `modulo` field.

More advanced computations are also possible, see Section 2.10.1 for more details.

The selection which graphics to use is made by zero or more `case` nodes. Each `case` node has a `minimal` and a `maximal` field to define the range, and a `use-cargo-id` field to state which graphics to use. Its value is a label referring to another graphics node defined previously in the file.

Finally, the `default-case` field must always be specified. It is selected when none of the cases matched (that is, the value obtained was outside the ranges specified by the cases (in the above example, when the year was larger than 80). The value of the `default-case` field is also a label referring to a previous graphics node.

2.10.1 Advanced computations

Rather than just querying a single variable, and performing some simple changes on it, you can perform more advanced computations with multiple variables. You define a tree of computations, where the left operand is either another computation or a variable, and right operand is a variable. For example, to subtract 30 from the year, write

```
compute {
    variable { name = year, right-shift = 0, and = 0xffff },
    operation = subtract,
    variable { name = all-bits-1, right-shift = 0, and = 30 }
}
```

2.11 Link graphics

To link the new graphics to the game, you use a `link-graphics` primitive, for example

```
// first define some graphics to use ....
st-mail: station-graphics { .... },
st-passengers: station-graphics { .... },
st-normal: station-graphics { .... },

link-graphics {
    feature = station,
    id = 0, id = 5,
    cargo { cargo-type = mail, cargo-id = st-mail },
    cargo { cargo-type = passengers, cargo-id = st-passengers },
    default-cargo-id = st-normal
}
```

The `feature` field defines for which feature you are linking graphics. The list vehicle or station ID's (starting with number 0 within each feature) is defined with a number of *id* fields. You may omit the list ID's for a generic feature-specific definition. Next comes a list of `cargo` nodes. For each cargo type (specified with a `cargo-type` field, you can state which graphics should be used with a `cargo-id` field, which is either an unsigned number, or a label to a `vehicle-graphics`, `building-graphics`, `station-graphics`, or `select-graphics`. This list may also be empty. Finally, you must define a default graphics to use. For more information, see NewGraphicsSpec:Action3.html.

There is an additional field `livery-override` to state that you want to override the normal graphics. See NewGraphicsSpec:Action3LiveryOverride.html for more information.

2.12 Set names of strings and vehicles

To set the names of objects, use `define-vehicle-name` for vehicles, and `define-text` for generic strings. Below is an example copied from oilpowerw.grf by *Born Acorn* and *DaleStan*.

```
define-text {
    language = german, feature = industry,
    string { number = 0x4803, text = "Kohlekraftwerk" }
}
```

It specifies the name of the coal-based power station in German. The meaning of the value of the `language` field depends on the version that you define in the `grl-id` primitive. If it is less than seven, the value of the `language` field is a bit set of the languages 'american', 'english', 'german', 'french', and 'spanish' (use a '+' sign between the identifiers to give several languages). If the version in `grl-id` is at least seven, it is a single language. In both cases, you can also use the value '`unknown`' as language. The `feature` field states for which feature the strings are defined. Finally, you should define a new value for one or more strings (with incrementing numbers). In the example, only one string changed. For more details, see NewGraphicsSpec:Action4.html.

2.13 Messages

With a `message` primitive, you can output a (error) message to the user. In general, messages are preceded by a `cond` or `init-cond` primitive that tests a condition (and if it holds, jumps over the message).

An example of an error message

```
message {
    severity = error, language = english,
    text = "%{file} is for the %{data} version of TTD.",
    data = "DOS"
}
```

The `severity` field defines how serious the error is. Allowed values are 'notice', 'warning', 'error', and 'fatal'. The `language` field is the same as with the language field in the '`define-text`' primitive. The `text` field can be any text, as long as there is a '`%{file}`' and a '`%{data}`' escape sequence in it (in that order). In addition, you may have zero, one, or two '`%{parnum}`' escape sequences after the '`%{data}`' sequence. The string of the `data` key-value pair gets inserted in the '`%{data}`' sequence. If you use the '`%{parnum}`' escape sequences, you must also provide numeric values for these by means of the `parnum1` (for the first occurrence), and `parnum2` (for the second occurrence) key-value pairs.

The following values for the `text` field are special in the sense, that the string is available in the Tycoon program rather than being stored in the `GRF` file:

```
"%{file} requires at least TTDPatch version %{data}"
"%{file} is for the %{data} version of TTD."
"%{file} is designed to be used with %{data}"
"Invalid parameter for %{file}:  parameter %{data} (%{parm})"
"%{file} must be loaded before %{data}."
"%{file} must be loaded after %{data}."
```

For more information, see NewGraphicsSpec:ActionB.html.

2.14 Grl bytes

With the `grl-bytes` primitive, you can specify a pseudo sprite as a sequence of bytes.

```
grl-bytes { data = [ 0x0c, 'H', 'e', 'l', 'l', 'o' ] }
```

Since you can specify anything in the data block, you can specify everything that the `NFO` language allows (except that the additional escape sequences, strings, etc cannot be used). At the same

time, it is unlikely that you ever want to use this primitive. It is here mainly as a back-up option when the `GRL` language is not supporting some primitive. By using the `grl-bytes` you can at least continue while the `GRL` language gets fixed.