Griddle: Ghu's Region Internal Database Description LanguagE

July 23, 1987

This is the definition of the formats for Griddle. There are two formats. One is nominally the input format, which we call "griddle" format. The other is a compressed output format, which we are calling "raw" format. These are considerably simplified from the older "riddle" and "genesis" formats. The new formats should greatly simplify tools programming, as well as reducing the bandwidth required to transmit region definitions around.

"GRIDDLE" FORMAT:

A Griddle file consists of one or more Griddle statements. There are five types of statements: include, assignment, macro-assignment, object-use and object-definition.

The include statement has the form:

```
include "<filename>"
```

where <filename> is the name of a file to include, expressed as a string constant. It does what you expect.

The assignment statement has the form:

```
<name> = <expression>
```

while the macro-assignment statement has the form:

```
<name> @= <expression>
```

The two are similar in that they both assign a value to an identifier, <name>. However, the nature of the value is different: The ordinary assignment statement computes the value of <expression> and makes that computed value the value of <name>. Further references to <name> result in this value. The macro-assignment statement, on the other hand, merely assigns the expression <expression> itself to be the value of name, and each time <name> is referenced, <expression> is recomputed and the resulting value used. In other words, macro-assignment defers the computation of <expression> and always results in the value that <expression> would have if it were substituted directly for <name>. If the values of any of the terms in <expression> change between references to <name>, then the corresponding value that results from a reference to <name> will also change.

The object-use statement describes an instance of an object which is to be created and placed in the database. Its form is:

```
use <className> [ <tag> ] [ = <globalId> ] {
        <properties>
}
```

The interpretation of this is: create an object of class <className>. If expression <globalId> is present, use its value as the new object's global identifier, otherwise generate an identifier internally. If <tag> (an identifier) is present, set its value to be the new object's identifier number.

<properties> consists of zero or more statements of the form:

```
        <fieldName>: <data>
```

where <fieldName> is a symbol that is the name of one of the fields of class <className>, and <data> is one or more expressions, separated by commas, giving values for the named field of the new object.

The object-definition statement has the form:

```
        define <classNumber> "<className>"
                <fields>
        enddefine
```

This says to define class number <classNumber> to have name <className> and the designated fields. <fields> consists of zero or more statements of the form:

```
        <fieldName> [ ( <dimension> ) ] : <type>
```

where <fieldName> is a symbol that is to be the name of the field. <dimension>, if given, is a number greater than 0 that is the array dimension of an array field (if omitted, it defaults to 1 -- i.e., not an array). <type> is the data type of the field. The following types are recognized:

bin15	two-byte integers
integer	synonym for bin15
bin31	four-byte integers
long	synonym for bin31
byte	one-byte integers
character	characters stored in single bytes
words	characters stored in two-byte words
varstring	pl1-style variable length string stored as a two-byte length followed by the string itself; dimension indicates maximum possible length of string
bit	single bits
avaid	(four-byte) avatar global ids
objid	(four-byte) object global ids
regid	(four-byte) region global ids
entity	entity specifiers: 2-byte type followed by 4-byte global id; type values 0, 1 and 2 correspond to regions, avatars and objects respectively
fatword	a two-byte integer stored as pairs of two-byte words, the first containing the low-order byte and the second the high-order byte

Expressions can take the following forms:

<name>	the value of identifier <name>
<number>	a numeric constant
<string>	a string constant
<bitstring>	a bitstring constant
(<expression>)	the value of the expression
~ <expression>	bitwise complement
- <expression>	arithmetic negation
a <expression>	create avatar id
o <expression>	create object id
r <expression>	create region id
<expression> + <expression>	addition
<expression> - <expression>	subtraction
<expression> * <expression>	multiplication
<expression> / <expression>	integer division

```
<expression> % <expression>            integer modulus
<expression> & <expression>            bitwise AND
<expression> | <expression>            bitwise OR
<expression> ^ <expression>            bitwise XOR
```

A <name> can be any string of alphabetic characters, digits, and '_', that does not start with a digit.

A <number> can be any of the standard number formats we use: a decimal number (any sequence of decimal digits that does not start with 0), an octal number (any sequence of octal digits preceded by '0'), a hex number (any sequence of hexadecimal digits preceded by '0x' or '0X'), a binary number (any sequence of 0 and 1 preceded by '0b' or '0B'), a quarter (any sequence of base-4 digits preceded by '0q' or '0Q') or an ascii character enclosed in single quotes (with the usual \ escape sequences).

A <string> is any sequence of characters enclosed in double quotes, with the usual \ escape sequences.

A <bitstring> is a PL/1 style bit string: a series of 1's and 0's enclosed in apostrophes and followed by 'b'. For example:

```
'100101010011'b
```

Also, of course, whitespace may be freely used for formatting, and comments, bracketed by "/*" and "*/", may be inserted at almost any point.

"RAW" FORMAT:

A "raw" file consists of 1 or more lines. Each line has the same format, a decimal class number, followed by a space, followed by a hex string (an even number of hexadecimal digits, each of which represents a byte of data). A number of physical lines may be combined into a single logical line by ending all but the last with a backslash ("\") character as a continuation marker.

Each line specifies the data to fill the fields of a database descriptor of an object, region or avatar. Which of these (and what class, if an object) can be determined by examining the class number. The hex string is an ASCII representation of the binary form of an in-core database record of the given type.

Certain fields in the hex data will be global database identifiers. These should be interpreted by the program reading the data relative to the class definitions (as laid out by Ghu 'define' statements). A positive value P in one of these fields indicates that the value P should be used for the identifier. A negative value N, however, is interpreted. If N is in the range -1 to -1000 (inclusive), just use the value N itself. However, if N is less than -1000, use the identifier assigned to the thing defined by the -(N+1000)th line of the file (e.g., -1001 is the first item in the file, -1002 is the second, and so on).

THE GRIDDLE AND FRED PROGRAMS:

Griddle and Fred are the two incarnations of a program that reads and writes region descriptions in the above formats. Griddle is primarily a translation tool while Fred is an interactive editor. Griddle reads some files and then writes some files, as directed by the command line options. Fred allows you to interactively edit one region at a time using a Commodore 64 and Fastlink.

These are the command line options:

filename	Read the object descriptions (in either of the above formats) contained in the given file (Griddle only). You can specify multiple input files. They will be read in the order specified.
-R	Assign relative ids to the regions generated, even if the are specified with absolute ids in the input.
-c filename	Read a contents vector file (Griddle only). Contents vector files are an obsolete representation, but we still have old data that we may want to get at.
-v filename	Write a contents vector file (Griddle only). The input should contain exactly one region plus contained objects.
-g filename	Output a griddle format file (Griddle only) containing all the objects in the various inputs.
-r filename	Output a raw format file (Griddle only) containing all the objects in the various inputs.
-i filename	Run Griddle in indirect mode. In indirect mode, Griddle reads "indirect lines" from the specified file, each of which specifies a file to read together with connectivity and parameter information. These indirect control files are the output of Plex.
-m number	Set the maximum number of objects that this run of Griddle can deal with. The default value is 256, which is fine for Fred, but large realms need to have more room allocated.
-t	Run Fred in "test mode". In test mode, Fred doesn't use the Fastlink port and none of the visual editing aids that use the C64 are available.

Any and all of the output file options may be used at the same time. I.e., you can generate both 'raw' and 'griddle' format output with one command.

In addition to the command options, the programs also read a few environment variables.

GHUDEFINES is the name of a file containing a series of 'define' statements. This file is read in first before any other processing takes place to define all the basic object classes. By default it is "/u0/habitat/defines.ghu".

CLASSINFO is the name of a "class.dat" file produced by Muddle. It is used to obtain information about class representation in the C64. By default it is "/u0/habitat/class.dat".

FREDPATH is the name of a default path for Fred to use when reading and writing files. By default it is "./", i.e., the current working directory. Fred also looks for a file named ".fredpaths" in the user's home directory for a list of other paths to choose from.

FASTPORT is the conventional variable that holds information about what Fastlink port to connect to the C64 with.

MORE ABOUT FRED:

Fred is an interactive utility that works with a C64 over Fastlink. It loads Aric's Reno program into the C64 from the file "/u0/habitat/reno.out" and then provides a screen and keystroke oriented front-end.

These are the key commands that Fred understands:

b Moves the current object into the background (i.e., clears the high-bit of the object's Y-coordinate).

B Tells the C64 to suppress display of the background objects, thus highlighting the foreground objects. The display remains thus until refreshed by the RETURN command or some other operation.

^B Toggles the border display bit on the currently selected object (which should be a trapezoid).

c Creates a new object. You are prompted for the class. You can enter a number or a standard class name, or take the default (which is displayed) by hitting RETURN.

C Tells the C64 to display the contents of the currently selected object (which ought to be a container of some sort). This display remains until the screen is refreshed.

d Displays the state information about a particular object. The information is shown on the terminal screen. You are prompted for the noid. This object becomes the active object.

D Drops the current object into the container under the cursor. The object under the cursor really should be a container of some sort.

^D Displays the current working path list on the terminal screen. The current path is highlighted.

e Allows you to edit the current object's state information. The information is displayed on the terminal screen with the first field highlighted. RETURN advances to the next field, BACKSPACE takes you back to the previous field. Simply by Typing in a new value change the field. Hitting the DEL key or advancing past the last field exits edit mode. If you have made any changes the revised object will be sent to the C64 for display.

E Allows you to edit the corners of the trapezoid that is the current object (it should be one). In trapezoid edit mode, pressing 'L', 'l', 'R' or 'r' places the joystick in control of the upper left, lower left, upper right or lower right corners, respectively, of the trapezoid. Moving the joystick moves the corner around. Pressing 'x' exits trapezoid edit mode.

f Moves the current object into the foreground (i.e., sets the high-bit of the object's Y-coordinate).

F Cycles the current object's flat type among GROUND, WALL, SKY and IMPASSABLE. The object should, of course, be a flat or a trapezoid.

g Saves the current region in a file in Griddle format. You will be prompted for the file name.

G Tells the C64 to render objects with patterns that emphasize their flat types. Display remains this way until refreshed.

h Gives help info -- a one-screen summary of these commands.

H Nudges the current object's container offset to the left a notch. Only affects the C64.

i Loads Reno into the C64.

I Changes how the current object will be held by an Avatar. Only affects the C64.

J Nudges the current object's container offset up a notch. Only affects the C64.

K Nudges the current object's container offset down a notch. Only

affects the C64.

L Nudges the current object's container offset to the right a notch. Only affects the C64.

n Tell the C64 to render the region in night mode. Display remains this way until refreshed.

o Displays on the terminal screen a list of the noids and classes of all the objects in the region.

O Flips the orientation of the current object (i.e., toggles the low-bit of the orientation byte).

p Increments the current object's color or pattern.

P Decrements the current object's color or pattern.

^P Allows you to edit the current working path list. The paths are displayed on the terminal screen with the current path highlighted. RETURN advances the current path to the next one on the list. BACKSPACE backs up to the previous one. You can change a path simply by typing in a new value. Hitting the DEL key exits the edit mode, and whatever path was selected at the time becomes the working path.

q Quits Fred.

r Reads a region from a file and displays it on the C64. You will be prompted for the file name. The file can be in either raw or griddle format.

s Sets the current object's graphic state to 0.

S Increments the current object's graphic state.

t Touches the object under the cursor, i.e., makes it the current object. The object's state information is displayed on the terminal screen.

T Twins the current object, i.e., creates a new object that is an exact duplicate of the current object.

u Undeletes the last object that was deleted, if any.

w Causes the Avatar in the region, if there is one, to walk to the current object location.

x Deletes the current object.

z Saves the current region in a file in raw format. You will be prompted for the file name.

+ Increments the current object noid.

- Decrements the current object noid.

! Allows you to execute a Unix command. You will prompted for the command string.

. Increments the current object's X-coordinate (actually increments it by 4, since object positions must be byte aligned anyhow).

, Decrements the current object's X-coordinate (actually decrements it by 4, since object positions must be byte aligned anyhow).

? Increments the current object's Y-coordinate.

/ Decrements the current object's Y-coordinate.

> Increments the current object's Y-coordinate by 10.

< Decrements the current object's Y-coordinate by 10.

RETURN Refreshes both the terminal screen and the C64 display.

^Z Pauses Fred