U61 - another block-based game

Christian Mauduit

v1.1.0

Contents

Chapter 1

Introduction

1.1 U61 in a nutshell

U61 is "another block-based game". Its main features are:

- Network play. Gameplay should be fast even with poor bandwidth. Local multiplayer game is also available.
- Theme support. The graphics and sounds are fully configurable.
- Rules customization. As it uses Lua scripts, the game rules can be changed by any player, without having to modify any complex C++ code.
- Cross-platform support. Since it is based on ClanLib, U61 runs on any platform supported by ClanLib (GNU/Linux, Windows...).

1.2 Basic rules

The basic rules in U61 are:

- The player has a map, and blocks fall int it. Blocks are made of colored squares.
- The player may control the block as it falls, for instance rotate or translate it.
- When the block lands at the "bottom" of the map, ie when it can not fall any more, it is merged with the map.
- When a given pattern is matched in the map, for instance, squares of the same color, then these squares explode and disappear.
- There's always a special square on the map, and when this square explodes, a "curse" is activated. Curses can be anything from bonus to maledictions.

These rules are not very precise, and this is what makes U61 different from other block-based games: it is possible to create one's own rules and play with them. This is achieved by creating scripts, using the Lua language.

1.3 Hey, but this is a Tetris clone!

No! No and No!

U61 is *not* a Tetris(c) clone. Its rules are much more general.

It's true that when I first started the project (in 1999), its name was "Tetr61s" and it was supposed to look much like "EITTris" with network support. Since then, I've had to deal with the so-called "Tetris Company", and realized that cloning Tetris(c) is not possible anymore. So I decided to start a more general project, which name is U61. Of course it's possible to configure U61 so that it looks exaclty like the genuine Tetris(c). But such a configuration is not provided in the standard distribution (inded the "classic" set of rules includes many enhancements).

So Tetris(c) can in a way be considered as a subset of U61, but this does not mean at all that U61 is a Tetris(c) clone.

But if you liked Tetris(c), Columns(c), or Puyo-Puyo(c), then you should like U61, so try it!

1.4 License

U61 is distributed under the GNU General Public License (GPL).

This means U61 is free software. Free as in speech. You can use it, modify it, and distribute it freely as long as your modified product still respects the GPL. For more information about the GPL, see:

- The "COPYING" file which should be present in every distribution of U61. It contains the GPL.
- http://www.gnu.org. The GNU (Gnu's Not Unix) is the result of the work of the FSF (Free Software Fundation). It is an attempt to make a complete and free UNIX like OS.

Please note that even if U61 can be considered as Open Source (http://www.opensource.org), I consider it as Free Software. So if you ever mention U61, please refer to it as being Free Software, and not plain Open Source.

1.5 Authors

As of today, I (Christian Mauduit) have written most of the code in U61. My coordinates are:


```
Christian Mauduit

E-mail: ufoot@ufoot.org
Web site: http://www.ufoot.org

GnuPG public key: FD409E94 - http://www.ufoot.org/gnupg.pub
GnuPG fingerprint: 4762 1EBA 5FA3 E62F 299C  B0BB DE3F 2BCD FD40 9E94

Snail mail: 32 rue Jean Moulin  95100 Argenteuil  FRANCE
```


Feel free to contact me, I always appreciate feedback.

An important contributer is Ursula Adler (U-Woman), who gracefully prepared the "default" theme for me. Please keep in mind that the graphics she prepared for me can be (and they are) released under the GPL, but her graphics are usually protected by other license agreements. Check her web site (http://absorbed.org) for more information. Please do not rip her graphics without asking.

Jimmy Kaplowitz wrote the "unstable" curse, and is currently works on Debian support for U61 - which is not an easy task since U61 has a bunch of dependencies 8-)

Chapter 2

Installation

2.1 Download instructions

U61 is freely available on the internet. As it is Free Software, you might also find it on a CD-Rom - provided that someone has taken a time to prepare and burn/press such a CD 8-)

Wether you get U61 from a web or ftp site, from a CVS repository or if you find it on CD-Rom, you should always get it under terms of the GNU General Public License. This is very important.

If you do not have U61 and want to get it, try the following URLs:

- http://www.ufoot.org/u61
- http://savannah.gnu.org/projects/u61
- http://u61.sunsite.dk

You may also get U61 from CVS, using the following commands:

```
cvs -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/u61 login
cvs -z3 -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/u61 co u61
```

When you are prompted with a password, just press "ENTER".

Today the downloadable packages are pretty. I'm working on packaging U61 in a better way. For instance have a "mini" package that would fit on a floppy and have some "extra" packages with all the themes and other stuff which are nice to have but can be skipped when one just wants to try the game out. But this requires work - and time - to be done the right way.

2.2 Required tools and libraries

2.2.1 Libraries

These libraries are required only if you want to compile and install U61 from the source. The GNU/Linux and Windows binaries are statically linked against all these libs so that you don't need to install them to run the game.

- zlib 1.1.3 : http://www.info-zip.org/pub/infozip/zlib/ (a compression library)
- Hermes 1.3.2 : http://www.clanlib.org/ (provides conversion between various image formats)
- Lua 4.0 : http://www.lua.org/ (the scripting language used by U61)
- libMikMod 3.1.8 : http://www.mikmod.org/ (allows the game to play .mod, .s3m, .xm, .it... files)
- ClanLib 0.6.1 : http://www.clanlib.org/ (a general purpose game Library)‘

Note that with the latest releases of U61 you need Lua to be installed, but you do not need Tolua nor clanLua. U61 uses Lua directly and does not use the ClanLib Lua bindings. This was not true for U61-1.0.0 which required Tolua and clanLua.

Since ClanLib is rather modular, you'll need to compile several modules, some of which are optional in ClanLib but mandatory for U61. The required modules are:

- clanCore
- clanSignals
- clanApp
- clanDisplay
- clanSound
- clanMikMod

You might also compile clanLua, but U61 won't use it...

BTW, recent releases of U61 do not use clanNetwork any more. The reason is that U61 used to use clanNetwork1, and clanNetwork1 has some flaws which caused boggus segfaults. It's true that a brand new clanNetwork2 is available, but the API has changed since clanNetwork1 and I preferred switching to the plain standard POSIX/Winsock API than migrating to clanNetwork2. As a result, you might compile clanNetwork if you like, but U61 won't use it.

2.2.2 Tools

These tools are required if you try to compile U61, if you got a binary distribution, you do not need them.

- C++ compiler : gcc under GNU/Linux, Visual C++ 6.0 under Windows. Other compilers might work as well, but you'll probably need to edit the makefiles.
- Python 1.5.2 : http://www.python.org/ (needed to build the themes and compile the docs)
- LaTeX : http://www.latex-project.org/ (optional, needed to compile the docs in PostScript format)
- dvips : http://www.radicaleye.com/dvips.html (optional, needed to compile the docs in PostScript format)
- PDFLaTeX : http://www.tug.org/applications/pdftex/ (optional, needed to compile the docs in PDF format).
- makeinfo : http://www.gnu.org/software/texinfo/texinfo.html (optional, needed to compile the docs in GNU Info format).

2.3 Source install under GNU/Linux

In theory, the best way to install U61 is to install it from the sources. However, on my machine, it can take me up to 3 hours to compile all the required libs and then the game. And keep in mind that I'm quite used to compiling U61 8-) So it might be a wise choice to download the static binary, which is about 1Mb bigger, but will probably save you more than 2 hours, unless you have already a working ClanLib install and a fast CPU.

Here are some reasons you might wish to install U61 from the source:

- You want to use a specific ClanLib target, such as ggi for instance, since the static binary uses the default X11 target.
- You want to install U61 in another location that /usr/local, because you are not root on your machine or just don't like /usr/local.
- You don't have a Pentium or for some strange reason the static binary segfaults on your machine.
- You never install binaries on your computer. Just like me 8-)

The source tarball should be named:

```
u61-?.?.?.tar.gz
```

U61 uses GNU autoconf. The install scripts I've prepared aren't perfect and do not check everything, but most of the problems should be detected. Let's say for instance that you are installing version 0.3.0. Once you got u61-0.3.0.tar.gz, type:

```
gunzip u61-0.3.0.tar.gz
tar xf u61-0.3.0.tar
cd u61-0.3.0
./configure
make
```

Now log as root and type:

```
make install
exit
u61
```

And if everything is all right, you should see the game running...

2.4 Static binary install under GNU/Linux

Since release 0.4.1, a static binary for GNU/Linux is available. Since the executable is statically linked against ClanLib and every library including the GNU libc, it should run on any GNU/Linux box which has an up to date kernel and X-Server running. In fact, if this binary does not work, installing from the source will probably not help much, but at least it might give you an idea of what's going wrong.

Note that the files are named "...-i386-binary.tar.gz" but there might be some problems running them on a plain 80386 computer. In fact, they are "i386" files as opposed to SPARC or ALPHA files, but

they probably require a Pentium to run. If you want to run the game on a true 386, you can still try and compile the source but the game will be very very slooooow.

The static binary archive should be named:

```
u61-?.?.?-i386-pc-linux-gnu.tgz
```

Let's imagine you are installing version 1.0.1. Once you have downloaded the file, log as root and type:

```
tar xzfP  u61-1.0.1-i386-pc-linux-gnu.tgz
```

Now log as a single user under X11 and type "u61" in an X-Term, and the game should run.

Note the "P" option in the tar command line, which tells tar to preserve full path names. This means the files will be installed directly in /usr/local, that's why you need to log as root. More precisely, files are installed in the following directories:

- /usr/local/games : binary files
- /usr/local/bin : links to binary files
- /usr/local/share/games/u61 : theme and script files
- /usr/local/share/man/man6 : documentation as a man page
- /usr/local/share/info : documentation in GNU Info format
- /usr/local/share/doc/u61 : documentation in various formats
- /usr/local/share/pixmaps : icon for Gnome, KDE, WMaker...

2.5 RPM install under GNU/Linux

Since release 1.0.1, I provide RPMs for U61. They should work pretty much like any other RPM but keep in mind that since U61 has a fair number of dependencies, installing the static binary as described above might be even easier than installing all the RPMs for ClanLib, MikMod, Lua,... and all possibly all the other packages required by U61.

Besides, I'm not an RPM expert, and the RPM I build have very basic and crude dependency informations, so they are far from being bug-free.

As a conclusion, use RPMs if you are familiar with them, otherwise, go for the static binary, it's certainly the easiest solution.

2.6 Source install under Windows

If you want to install U61 form the source under Windows, you'll need Microsoft Visual C++ 6 and a fair amount of free time 8-) The MSVC workspace is included in the GNU/Linux source tarball, so this is what you need to do:

- Get the GNU/Linux source tarball, and unzip/untar it.

- Get a working ClanLib developper install. Try to compile the ClanLib examples first.
- Tweak the U61 project file so that it matches your personnal install. You'll probably have to change some include and/or library directories.
- Pray that MSVC does not segfault when compiling 8-)
- Create a directory structure which matches the one I use in the binary releases. The most important point is that u61.exe must be able to find the theme and script files.

If this succeeds, you'll probably end up with exactly the same content as the default binary release, so at this time you might wonder: "why the hell did I spent a whole afternoon doing this?" 8-}

BTW, there's no easy way - at least I do not provide it - to compile the datafiles under Windows. What I personnally do when I release windows binaries is that I import the compiled datafiles from my GNU/Linux environment. As I almost never run Microsoft Windows, and since this poor system does not have a decent batch support, I'm not ready to implement and maintain such a fonctionnality.

2.7 Binary install under Windows

All you have to do is unzip the file in a directory, for instance "C:\Program Files\".

WinZip - or whatever unzipper you use - should create a "U61" subfolder, which contains everything, including the "U61.EXE" file you have to launch to play the game.

Chapter 3

Quick start

3.1 Play alone

First you must run the game 8-) that's to say:

- Double-click on u61.exe under windows.
- Type u61 in an Xterm under GNU/Linux.

Then the main menu should appear on the screen. To navigate in the menus, you'll have to use:

- The keyboard arrows: up/down to change the selected menu item, and left/right to change some settings such as sound volume.
- The "ESC" key to go back to the previous menu. Also usefull to quit the game...
- The "ENTER" key to validate your choice.
- "F10" will close the game immediately, without any warning. Usefull if you don't want to go all the way to the "Exit" menu.

There's no mouse support in U61 yet.

So to play alone, just select the "Play alone" menu item by simply pressing "ENTER", and the game starts. You can control the falling blocks with the keyboard arrows.

If you don't like the key settings or their repeat rates, please keep in mind that U61 is highly customizable. Indeed the "Player options" -> "Set up player n" -> "Key settings" -> "Advanced key settings" enables you to set up different repeat rates for each key.

BTW, the "Play alone" option is here only for training since playing U61 alone does not really make any sense. You *need* to find other gamers to play with, for U61 has been designed to be a multiplayer game, and you will completely miss the fun if you only play alone.

3.2 Start local multiplayer game

This is definitely more fun than playing alone! Up to 4 players can play on a single computer, however, until I add joystick support to U61, I can't really figure out how 4 players could fit comfortably - it's possible but not really easy - on a single 102 keys keyboard.

So here's what you'll have to do:

- First you'll have to set up keys for the different players. By default U61 maps the same keys for all the players. This is because it's impossible to set key options which will fit for every situation (1,2,3 or 4 players). So you need to go to "Player Options" -> "Set up player n" -> "Key settings" menu, and set redefine the keys. To define a key, just select the action to bind with the up/down arrows, and press "ENTER", then the key you want to use for this action.
- Once you have set up the keys, go to "Player Options" -> "Set active players" or "Start new game" -> "Choose players" and there you can enable/disable players.
- Now go the "Start new game" and select "Local multiplayer game" and you're ready to play.

Now whenever you see a weird black and white square, with a "?" in it, just try to make it dissappear. And see what your opponents think of it 8-) BTW, the game never ends, whenever a player looses, he restarts with an empty field. This is what some would call a "deathmatch" mode.

3.3 Start a network game

Probably the trickiest thing to do, but also one of the most powerfull features of U61. U61 offers much more than a single one-to-one network game. Currently, up to 5 computers can be connected together, and 2 players can play on each computer. So this makes a total of 10 simultaneous players...

U61 uses a client/server architecture, so a computer will act as a server, and then others will act as clients and connect to it.

- To start the server, simply select "Start new game" -> "Create network game".
- To start the client, select "Start new game" -> "Join network game". By default, the client tries to connect on "127.0.0.1" which is the loopback network interface. You *need* to set the name or the IP address of the server to the correct value (which is of course the address of your server). To know what is the address of the server, type "ipconfig" (under Windows) or "ifconfig" (under GNU/Linux, must be run as root) on the server. Once you have set up a correct IP address, (or again the name of the server, but I personnally prefer using IP addresses directly) you can select "Join".

If everything is OK, the client player(s) should appear on the server. Oh, BTW, the client should better hurry connecting to the server, for players on the server do not wait for the clients to start playing. This is the kind of methods used in games like Quake by ID Software for instance.

By default, U61 will use port 8061 to communicate. Many firewalls won't let IP packets pass them if one uses this port. For instance, it might be impossible to play over the internet at work, if the only connexion to the internet you have is an access with a proxy on port 80 - which is what many people have. This is not U61's fault anyway... You can also have problems if the server is started on a machine which uses network address translation (NAT). If you encounter serious problems setting up an internet

game, my advice would be to first try to set up a network game on a LAN, and then when you know how to make it work locally, try and launch it over the internet again.

3.4 Configure the game

Well, I keep on saying that U61 is "highly customizable", so I guess some of you will be eager to know how to do it 8-)

There are basically 4 ways to configure U61:

- Use the GUI/menus to change options such as key settings, sound volume...
- Write Lua scripts to create new sets of rules. This is definitely what makes U61 different from any other block-based games.
- Create or tweak the existing themes to change the look and feel of the game.
- Modify the C++ core engine. You can do it since the game is GPL'ed. However, I do not recommend it since this kind of practice is likely to introduce severe - and not easy to deal with - compatibility issues.

How to change the theme and/or change the rules is described in other sections, so here I'll only explain how to change the options using the menus. Many settings are quite self-explanatory, so I do not detail them, but some still require some explanations. Here's a list of usefull options and what they do:

- "Game options" -> "Speed" -> "Curse delay" : the time (in seconds) after which the special "curse square" will automatically be moved.
- "Game options" -> "Graphic" -> "Max frames per sec" : the max frame rate. Having a maximum frame rate avoids U61 taking 100% of the CPU. Increase this value to have a smoother display on high-end systems.
- "Player options" -> "Set active players" : here you'll choose which players will be activated. Note than in a network, game, only 2 players will be allowed per machine.
- "Player options" -> "Set up player n" -> "Advanced key settings" -> "[some key]delay" : this is the time (in 1/100 sec) before the key starts repeating. This is a per-player settings so players can have different key sensitivity even if they play at the same time on the same machine.
- "Player options" -> "Set up player n" -> "Advanced key settings" -> "[some key]repeat" : this is the time (in 1/100 sec) used in repeating mode. It should be lower than the equivalent "[some key]repeat]" option.
- "Player options" -> "Set up player n" -> "Anticipation mode" : allows the player to view where his block will land. Also called "ghost mode" in some other block-based games.

Chapter 4

Troubleshooting

4.1 I can't compile U61, help!

4.1.1 Problem

U61 configuration scripts are not perfect, and compilation might fail for various reasons since U61 depends on a fair amount off libraries, some of them not being - yet - available on every GNU/Linux distribution (ClanLib, Hermes, Lua...).

4.1.2 Solution

- Read the INSTALL file in the root directory.
- Check that all the libs needed by U61 are correctly installed. For instance, Lua is quite tricky to install since you have to run ld manually to create .so files.
- Try and install a binary release. The GNU/Linux static binary does not require any extra library to run, so it might save you lots of time.
- If you're still in trouble, you can still ask for help on the mailing-list.

4.2 When I run U61, I get protection faults 8-(

4.2.1 Problem

This often happens when data and/or configuration files are not installed correctly. It may also happen if you have installed a new release of U61 over an older one.

4.2.2 Solution

Under GNU/Linux, remove the following files/directories:

- "/usr/local/share/u61" (or "/usr/share/u61"). This is the directory where all the themes and scripts are stored.
- "~/.u61rc". This is a useless file used by previous releases of U61.
- "~/.u61". This is the directory where the configuration file is stored. It can also contain custom themes and scripts.
- "/etc/clanlib.conf". This is ClanLib's configuration file, used mainly to choose which display target should be used.

Once you have removed these, reinstall the game and and try and launch it again.

Chapter 5

Mailing lists

5.1 u61-user

5.1.1 Description

This list is for general discussions about U61. Here you can make suggestions, submit bug reports, ask for help, find players, etc... Basically, any question or remark which concerns the game is welcomed on this list.

5.1.2 Practical informations

You can't send messages to the list without subscribing. The only reason for this is that it's one of the only way to block spam efficiently. However, I insist on the fact that anyone can subscribe, and the subscription to the list is not moderated. So if you are a human being and not a stupid spam robot, you're welcome on the list 8-)

Here's a list of usefull URLs:

- To (un)subscribe: http://mail.nongnu.org/mailman/listinfo/u61-user
- To consult archives: http://mail.nongnu.org/pipermail/u61-user/
- To post on the list: u61-user@nongnu.org

Chapter 6

Themes

6.1 About themes

6.1.1 What is a theme?

Think of themes as "skins" which can be applied to the game. Technically a theme is a datafile or a directory which contains:

- Graphics
- Sounds
- Musics

The scripts are *not* contained in the dafafile, so one can use any theme with any set set of rules, since both are indenpendants.

The theme can be chosen in the "Options/Theme" menu. Just move to the theme you want to use with the keyboard arrows and press ENTER.

6.1.2 Copyright and artwork

It's very difficult to find artwork which can be GPL'ed. Most of the free resources on the web are available for free as long as one does not make money from them. However this kind of restriction is not compatible with the GPL, so I have created almost everything myself, from scratch. This way *all* the artwork in U61 is available under the terms of the GPL. This means that:

- All the graphics are GPL'ed. I have drawn most some of them myself, and the pictures come from my personnal collection. Only some of the graphics in the default theme have been contributed by Ursula Adler, and she agreed that I placed her contributed work under the terms of the GPL.
- All the sounds are GPL'ed. Some of them I found on a CD with domain public sounds, and the others I have recorded myself, with a microphone.

- All the musics are GPL'ed. I insist on this point since copyrights on .mod, .it or .xm files are not always very clear. So these mods are not public domain. They are placed under the terms of the GNU General Public License. The samples are also GPL'ed. I have recorded the drums years ago with a dusty SB16 and a friend, and the brass samples where recorded at home with my own instruments and my good old microphone 8-)

If you want to contribute to U61 by creating new themes, keep in mind that your work must be available under the terms of the GPL. This means that I can not accept themes which use graphics from a proprietary game for instance. So if you want to make a "Diablo 2 theme" you can do it but I won't distribute - and you should not distribute it either since it would be a plain GPL violation, besides being also a copyright infringement of Blizzard's game.

6.1.3 Difference between .dat and .scr themes

As U61 is based on ClanLib, it is able to read the datafiles in 2 different ways:

- .dat files are WAD-like files. This mean all the resources are stored in a single opaque .dat file. I used this format to ship the default theme since this format is less problem-prone than .scr files. Basically, if you have a successfully compiled .dat file, there are few chances that the games fails when it runs, since everything has been checked before. All the resources which are required to build the default.dat file are of course available in the source package.

- .scr files are plain text files which contain links to the resources files. In U61, a "mytheme.scr" file will basically contain pointers to graphics, sounds and music files located in the "mytheme" directory. You just have to modify the .pcx, .wav or .xm files - which are "standard" file formats. The only drawback of this system is that you might end up in breaking up the theme and in this case the game simply won't run.

By default the makefiles in source package produce both .dat and .scr versions of all the available themes. Then the install scripts chose to install the default theme as a .dat file and other themes as .scr files, but you are free to change this behavior.

6.1.4 Directory issues

U61 reads datafiles from 2 different directories:

- The builtin theme directory. It should be something like "/usr/local/share/u61/theme" under GNU/Linux or "C:\Program Files\U61\data\builtin\theme" under Windows. This is where all the files coming with U61 are installed. Even if it's OK to modify these files directly, you should not modify them directly, since after you won't be able to get the genuine themes back easily.

- The user theme directory. It should be something like "~/.u61/theme" under GNU/Linux or "C:\Program Files\U61\data\user\theme" under Windows. These directories are empty when you first install the game. This is where you should put your home-made themes, so that there's no conflict with the builtin themes which are originaly shipped with the game.

6.2 Building new themes

6.2.1 About the makefiles

To build .dat and .scr themes, I use an IMHO quite complex makefile system, involving Python scripts which generate makfiles automatically. You might wonder why I did that since with ClanLib's support of .scr files I could have simply copied the source directory into the target install directory? Well, here's the answer:

- On one hand, I like the idea, when I create themes, to share files between themes. For instance the default and the fanfare themes use the same sprites for squares. It would be a nonsense to duplicate all those files in developpement. If I change these sprites, I do not want to make the changes twice.

- On the other hand, when the themes are installed. People might to add, delete, and share their installed themes if they have tweaked them. And in this case, if themes have been installed as .scr files, you never know what to delete and/or change. For instance, should you want to delete the fanfare theme, you would have to look at every file it uses, and check that it is not used by another theme. And if you imported a friend's theme which uses patched versions of the genuine graphics, then all your already installed themes would be affected.

Therefore, the makefiles build .scr files from .theme files. The .theme file is the developper file, with references to files in the square or sound directories, and the .scr file is the final file which will be read by U61, and it contains references to files in a directory which is automatically created and contains a copy of all the required files. This way, each theme is stored in a independant directory.

However, if you want to tweak an existing theme, you can simply take the installed .scr and associated directory and start tweaking theme right away without bothering with all this makefile stuff. But keep in mind that if you want your theme to become part of the "official" U61 distribution, creating it with the same makefile system I used will make things much easier in the long run.

6.2.2 Getting started

Let's say you want to build a new "foo2" theme from an existing "foo" theme. What you should first to is open the foo.scr file with a text editor, and look what's in it. It is a plain and quite self-explanatory file, so it should not take ages to understand its global structure.

So just copy the existirg foo.scr theme and its associated directory from the builtin theme directory ("/usr/local/share/u61/theme" under GNU/Linux or "C:\Program Files\U61\data\builtin\theme" under Windows) to the user theme directory ("~/.u61/theme" under GNU/Linux or "C:\Program Files\U61\data\user\theme under Windows) and when you run U61, you should see the theme foo appear twice in the list of available themes. One is the builtin theme and one is the theme you just copied.

Rename the foo.scr file you have copied to foo2.scr, (only the foo.scr should be renamed for now but not the foo directory, since this freshly copied foo.scr file still has references to a directory which name is foo). Now if you restart u61, foo (the builtin theme) should be available along with foo2 (the one you have just copied/renamed).

You are now ready to tweak foo2.

6.2.3 Naming conventions

All the sections follow some basic naming conventions:

- a "_w" suffix indicates that the value is a width
- a "_h" suffix indicates that the value is a height
- a "_x" suffix indicates that the value is an x coordinate. 0 is associated to the left of the screen.
- a "_y" suffix indicates that the value is an y coordinate. 0 is associated to the top of the screen.
- an "offset_" prefix indicates that the value is an offset which will be added to the top-left corner of the zone the item is associated to.
- a "_big" suffix indicates that the value is used when drawing a map for a local player. Usually twice bigger than its _small equivalent.
- a _small suffix indicates that the value is used when drawing a map for a remote player. Usually twice smaller than its _big equivalent.

6.2.4 The property section

This section contains integer values which are used to define the size of the theme (640x480, 800x600...) and the position of various items on the screen. All dimensions are given in pixels.

Here is the list of all the values:

- screen_w, screen_h: the size of the window in windowed mode, or the resolution in full-screen mode.
- player_w, player_h: the size of the zone which is associated to a player. This includes the map and the info zone. Those zones will be tiled and/or centered automatically.
- offset_map_x_big, offset_map_y_big, offset_map_x_small, offset_map_y_small: the position of the map background bitmap. This bitmap is the main background image a player sees on his playfield. The offset is added to the position of the player's zone, which is calculated at run-time.
- offset_info_x_big, offset_info_y_big, offset_info_x_small, offset_info_y_small: the position of the info background bitmap. This bitmap is the background image of the info zone, where score and other information are displayed. The offset is added to the position of the player's zone, which is calculated at run-time.
- offset_squares_x_big, offset_squares_y_big, offset_squares_x_small, offset_squares_y_small: the position of the top-left square. Indeed, the background bitmap might be bigger than the actual playfield, so ususally this value is the size of the "border". The offset is added to the position of the map background image, which is not necessary the same the the global position of the player.
- offset_name_x_big, offset_name_y_big, offset_name_x_small, offset_name_y_small: where the name of the player should be displayed. The offset is added to the position of the info background image.
- offset_score_x_big, offset_score_y_big, offset_score_x_small, offset_score_y_small: where the score of the player should be displayed. The offset is added to the position of the info background image.

- offset_target_x_big, offset_target_y_big, offset_target_x_small, offset_target_y_small: where the name of the target should be displayed. The target is the player's next victim, who will receive a curse whenever the player gets an occasion to send one. The offset is added to the position of the info background image.
- offset_preview_x_big, offset_preview_y_big, offset_preview_x_small, offset_preview_y_small: where the next block preview should be displayed. The offset is added to the position of the info background image.
- offset_curses_x_big, offset_curses_y_big, offset_curses_x_small, offset_curses_y_small: where the list of curses which affect the player should be displayed. The offset is added to the position of the info background image.
- offset_antidotes_x_big, offset_antidotes_y_big, offset_antidotes_x_small, offset_antidotes_y_small: where the number of available antidotes should be displayed. The offset is added to the position of the info background image.

6.2.5 The font section

This section contains all the fonts used in the game, plus some special fonts used to display curses and antidotes.

There two kinds of fonts in ClanLib:

- The first kind is the one I use but it's deprecated. You have to make a 256 color indexed bitmap, and tell which color is the transparent color. One also needs to draw rectangles with colors 253,254 and 255 to tell the font engine where the letters are. Check the existing fonts in U61 to have more information on how to make this kind of fonts.
- Alpha based font can be built from .tga files. Check out ClanLib's doc to know more about this kind of fonts.

Here is the list of all the values:

- menu: font used to display the menus. Must contain all the common ascii characters.
- info_big, info_small: font used to display the informations in the info zone, such as scores and players names.
- symbol_big, sumbol_small: special font used to display the curses and antidotes. The "-" character represents the negative curses, the "+" character represents the positive curses, and the "a" character represents the antidotes.

6.2.6 The sound section

This section contains all the sounds, except the musics.

Here is the list of all the values:

- game_start: played when the program is launched.
- menu_move: played when the players moves up or down in the menus.

- menu_move: played when the players moves validates a choice in the menus.
- block_touch: played when a block lands at the bottom of the map.
- block_pattern: played when the player completes a pattern, for instance when a line disappears.
- player_start: played when a remote player joins the game.
- curse_received: played when a player receives a curse.
- player_loose: played when a player looses and restarts with a blank map.

6.2.7 The music section

This section contains all the musics, in .mod, .xm and in a more general manner in any format supported by MikMod.

Here is the list of all the values:

- number: the number of mods available. It's OK to make a theme with no music at all, in this case number must be 0.
- mod_n: the actual module files. In "mod_n" the "n" should be replaced by an index number, ranging from 0 to (the number of mods available)-1.

6.2.8 The square section

This section contains all the sprites used for the squares.

You need to provide 16 set of sprites, because there are 2 sizes (small and big) and 8 colors. For instance "small_3" is the set of sprites associated to color 3 used to dislay the squares for network players.

The bitmap files must contain 16 different pictures for each sprite, which will be used to animate the sprite. If you don't want to use the square animation, simply provide a sprite with 16 times the same picture.

Please check ClanLib's documentation to know more about sprites. IMHO a good way to understand how it works is to try and understand how the builtin squares in U61 have been created.

Here is the list of all the values:

- big_n, small_n: the standard squares.
- big_curse, small_curse: the special square which blinks and indicates where the current curse square is. BTW, the sprite does not need to handle the blinking, simply draw a standard 16 step sprite, and the game engine will handle the blink automatically.
- big_explosion, small_explosion: the sprite used to represent the explosion of a square before it disappears.

6.2.9 The map section

This section contains all the background pictures used for the player's playfield.

Here is the list of all the values:

- number: the number of maps available. There must be at least one map so number can not be 0.
- big_n, small_n: the maps, where one should be able to put 10x25 squares. In "big_n" and "small_n" the "n" should be replaced by an index number, ranging from 0 to (the number of maps available)-1.

6.2.10 The back section

This section contains general background pictures.

Here is the list of all the values:

- main: the main background image. The one which is displayed when one launches the game, and is used as a background in any situtation.
- shade: this bitmap is blitted on the screen in the in-game menus. By default I use a plain bitmap which makes everything look darker.

6.2.11 The info section

This section contains the background pictures used for the "info" zone, where the score and other parameters are displayed.

Here is the list of all the values:

- big, small: the picture used to fill the "info" zone.

Chapter 7

Script tutorial

7.1 Disclaimer

Before you start, please note that this document is:

- *not* a Lua tutorial (Lua is the language use for scripting in U61). Of course at the end of this document I hope you'll be able to do some basic Lua scripting since Lua is quite quick to learn, but if you want to do advanced Lua coding, go and check the official sources. I'm myself not a great Lua programmer so I won't be able to help you a lot anyway.
- *not* a complete reference of what is possible to do with U61. It only shows how to get started with script writing, but if you really want to do complicated things, please read the rest of this site.

However, it should help you write your first script lines, and I have good reasons to think these are the hardest to code.

You do not need any special programming skills before reading this document. If you are curious and wish to learn how to toy with u61 it should be enough, provided that you know at least how to move, copy and edit files on your system. Of course it makes things easier to know about programming, but honestly Lua is not a difficult language to start with.

7.2 Step 1: create a script file

Well, let's not reinvent the wheel for instance, we'll just copy an existing file. Writing a complete file from scratch might indeed be too hard for a start.

If you got an "official" U61 package (from http://www.ufoot.org for instance) you already have some scripts that came with the game. By the time I write this document, only the UNIX platform is supported, so the script files should be in "/usr/share/script/u61" or "/usr/local/share/script/u61" or in the directory where you installed U61.

Now copy "/usr/local/share/u61/script/classic.lua" into "~/.u61/script/test.lua".

If now you launch U61, and go to "Game options / Rules" you should see "test" appear in the list, use

the arrows to move the selection on it, and the start a new game. You should have the impression to play with the "classic" rules, but in fact you are using the "test.lua" script.

7.3 Step 2: create an error

OK, now scripting allows lots of liberty in programming. One of the basic consequences for U61 is that you can code almost anything in a script, you are pretty sure you won't get a protection fault. In fact the worst thing that can happen (and it does happen) is an infinite loop. The result is that the game hangs and you have to kill it.

So what I propose is to deliberately create an error in the script and see what happens. This will help you deal with errors later. So just edit your brand new test.lua script file with your favorite editor (vim and GNU Emacs seem to work good but I would never mean to influence you in the choice of the editor) and go to, say line 93. There you just add the following text:

```
\""56zboub!
```

Now relaunch U61, which should have kept test.lua as your default script, and you should get nothing on the player(s) map(s) but instead a message on the console (provided that you have launched U61 from a terminal) that says:

```
lua error: unexpected token;
  last token read: `\' at line 93 in file `/home/ufoot/.u61/test.lua'
Lua error reading script, code is 1.
```

And this is it. You have the line number of the error and a short description of it. This should help debugging.

It is not necessary to quit and re-run completely u61 before trying out a new script. Indeed the script is reloaded from the disk each time you start a new game (with "Quick start" for instance). So all you need to do is:

- Launch U61
- Select your script with "Game options / Rules"
- Start a game with "Quick start", view what happens.
- Edit your file, save it.
- Re-start the game with "Escape / Stop this game / Quick start" and view how your modifications influence the gameplay.

As a start, just remove the faulty text I just told you to add from the script, same it and re-start a new game without quitting U61. The error should disappear.

7.4 Step 3: learn Lua basics

As I said before, this document is not a Lua reference, and it won't replace the reading of the official manual, which is available on this site anyway. Still, you do not need to be a Lua wizzard to tweak

U61's scripts, as I'm not myself a very bright Lua programmer. So this short passage is just to give you a quick start.

In U61, I essentially use Lua functions, there are other aspects of Lua programming than functions, but I think functions are easy to understand, so I use them. To declare a function, do like this:

```
function my_function(par1,par2)
    ...
end
```

As you see, it's easy to declare a function, you might use this function elsewhere by calling:

```
my_function("hello","world")
```

Within a function, you can declare local values. Local values have a scope which is limited to the function where they are declared. Any function may return a value, here's an example:

```
function calculate(a,b)
    local temp
    temp=a+b
    return temp
end
```

As you may have guessed, this function performs an addition, you can call:

```
c=calculate(1,2)
```

and the value of "c" will be 3.

Lua has some common controls like "if...then" or "while...do". Here is a short example:

```
if val==0 then
    result="zero"
elseif val<0 then
    result="negative"
else
    result="positive"
end
```

And another one:

```
while i>0 do
    result=result*2
    i=i-1
end
```

To compare numeric values, you can use the operators:

- == (equals)
- ~= (different)

- >(greater)
- <(lower)
- >= (greater or equal)
- <= (lower or equal)

Now if you have already programmed in another language like C, Pascal or Basic you know that almost as much as I do about Lua. I can tell you it's enough to start hacking.

7.5 Step 4: Try to understand the existing scripts

OK, now we know how to modify a script randomly and generate errors. Great, but you might want to do more than that! Let's start by a - very - short analysys of our test.lua script file.

"user_rotate_left" is called when the rotate left key is pressed. In U61 0.2.2 the code for user_rotate_left for the classic.lua script is:

```
function user_rotate_left()
    rotate_block_left()
end
```

This means the action to perform when "user_rotate_left" is called is to call "rotate_block_left". So what's the difference between "user_rotate_left" and "rotate_block_left"? Well, they do exactly the same but "user_rotate_left" is a reserved name, this function being what I call a "user callback", since it's called by the C++ engine and it's the responsibility of the scripter to code it correctly. "rotate_block_left" is totally unknown from the C++ engine, it's just a function I created because I found it convenient to have a function that rotates a block already coded and usable elsewhere in the script. If we look at the code of "rotate_block_left", we find:

```
function rotate_block_left()
    local x
    local y
    local i

    i = u61_get_nb_items()-1
    while i>=0 do
        x = u61_get_item_x(i)
        y = u61_get_item_y(i)
        x,y = y,-x
        u61_set_item_x(i,x)
        u61_set_item_y(i,y)
        i = i-1
    end
end
```

Well, we won't detail the way it works, but we'll still notice that it uses the functions with a name like "u61_fonction_name". These functions belong to what I call the "system api". It's a set of functions which allow the user to query and modify U61's world. For instance "u61_get_nb_items" returns the size

of the current block, which with the classic rules is generally 4. But beware, it could be more if you had a nasty curse, and in a general manner, it's good to write scripts that are usable in various situations.

7.6 Step 5: Toy arround

Now, we'll as a first example change our "test.lua" file so that when the user rotates a block, the block goes down of one square. This will forbid abusive rotation of the block and make the game harder.

Just edit test.lua and find the code for "user_rotate_left" and "user_rotate_right". It should look like:

```
function user_rotate_left()
    rotate_block_left()
end

function user_rotate_right()
    rotate_block_right()
end
```

Just replace this by:

```
function user_rotate_left()
    rotate_block_left()
    u61_set_block_y(u61_get_block_y()+1)
end

function user_rotate_right()
    rotate_block_right()
    translate_block_y(1)
end
```

And it should work! You'll notice that the 2 functions are coded differently. With "user_rotate_left" I used the U61 system API directly, whereas with "user_rotate_right" I used an Lua function which encapsulates the calls to the system API. I personnally find it a good idea to make very small functions but you are free to do whatever you want. But I believe some functions such as the user callbacks should be as small as possible, for in the long run it should make the sharing and merging of script files easier.

7.7 To be continued...

Well, that's all for now, I hope to enrich this document soon.

Chapter 8

Script basics

8.1 Introduction

This section describes how scripting works within U61. Reading this should help you a lot writing new scripts for U61. However, it assumes that you have some basic programming skills. Anyways, if you are not a programmer yourself, you might find out that hacking U61 scripts is not that hard, by having a look on the script tutorial I made.

8.2 Different categories of functions

8.2.1 Overview

When scripting within U61, one has to use functions. There's not really any object-oriented approach in the scripts I wrote and I do not really see where such a thing as OO whould make things easier for beginners.

So you'll basically have to use functions, functions and functions, divided in 3 categories:

8.2.2 User callbacks

These functions *must* be defined in every script you make. They are called by the C++ engine. A typical user callback is "user_block_shape" which is the function that defines the shape of blocks. It receives an integer and must create the block associated to this code. You may change this functions (it's even recommended!) but keep in mind that they should always do something in the same spirit than the genuine ones. I mean that a "user_block_move_left" function should not try to change the map radically. However, there are protections to avoid this, as we'll see later, but still keep in mind that those functions should do something that corresponds in some way to the name they have. For instance, replacing "user_block_rotate_left" by a function that changes a block in some weird way is ok, as long as the game stays playable.

8.2.3 System API

These functions are available in every script you write. In fact, it's just like I added a library to standard lua fonctions, and you can use it. A typical system api function is "u61_map_add_score" which takes an integer and adds it to the current score. You can not modify these functions, as they are coded in C++. They are the only tool you have to read or modify a player's parameters.

8.2.4 User library

User library. Well, of course you can create your own set of Lua functions. In fact, those functions will be called within user callbacks and will contain calls to system API functions or to other user functions which form what I call the user library. For instance, there is a "column_down" function in the classic.lua script, which moves a whole column down by one square. This function might interest you in another script, so it can be viewed as belonging to a library but it's very different from a system API function since you can modify it.

8.3 Context handling

When a user callback function is called, it's executed within a context which is associated to a player. You will never find in a system API function a parameter like 'player_id'. Indeed all script functions in U61 are designed to operate on one single player. You can safely assume that there won't be any unwanted interaction between players.

But there's a big pitfall (which I believe is not so easy to avoid that it seems) : you can *not* store anything in a global lua value. Well of course you can do it within a function - but in this case what is the advantage over a local value??? - but you can be 100% sure that what you stored there won't be available next time the function is called. Worse, it could be overwritten by some other call to the same function but for another player. Basically, using global values to communicate between different players is a *very* bad idea and is bound to fail.

However, I tried to provide a decent API to store persistent values. So you may use "u61_map_set_global" or "u61_map_get_global" to store and retrieve integers. Everything you store in there will be backuped and restored each time a user callback is called. This way you may for instance put here a flag which means "this player has the szwing-szwang-bouloulou malediction on him and he can not move his blocks whenever they are yellow or pink". You could also store a custom counter and when this counter reaches 0 you just mess up everything in the map. So these things are possible but beware, you may never store anything as a global lua value, for this would mean your script would not fit for network play, which is in my opinion very sad.

8.4 Access rights

Now, these functions may read or write info on 2 different "objects", which are the map and the block, these objects are described later.

And sometimes, U61 needs some functions *not* to modify one of these objects, are not to be able to get informations about it. This is very important. For instance, the "user_block_move_right" function must not change the map. But you might think: "why should one impose such a limitation?", for I have, indeed, decided that it would be so.

In fact, the issue is about performance. Indeed the "move_..." functions may be called very often, and they need to backup everything they might possibly change, because this way if the move is not possible we can perform a sort of "rollback". So authorizing those functions to modify the map would mean a backup of the map each time you move a square, and I decided that this could be too time consumming for a simple basic move.

So every user callback and every system API function has rights associated to it. This rights are:

- block read
- block write
- map read
- map write

Basically, when a user callback is called, all the functions it calls inherits the rights it has. For instance, within "user_block_rotate_right" which grants the "block_read" and "block_write" rights, you can call "u61_block_get_x" which requires the right "block_read", but you can not call "u61_map_set_score" which requires the "map_write" right. That's all.

8.5 Squares

As U61 is a block-based game, everything is about squares. The next section describes the block and map objects, which are both made out of squares: the block is a set of squares associated to positions and a map is an array of squares.

The most important attribute of a square is its color. Today, there are 8 square colors. Some might think this is a limitation but I had to choose a number and keep myself to it since it makes it really easier to make game logic and graphics work well together. Basically it's possible to make U61 work with 64 colors, but you'll need to draw a lot of new squares and hack the C++ code. Wether it's worth it is up to you, in any case the code is GPL'ed so you can modify it if you wish, but I think 8 colors is enough to have fun and is quick to visualize.

When I say "color" it does not mean there are blue, yellow, red squares. You might want to draw different squares which do not look the same but have the same color. Color is just a way to differenciate squares from a logical point, wether the players sees different colors depends on the artist.

Very important: the convetion in U61 is to assume that when the color of a square is negative, the square is in an "empty" or "transparent" state. For instance, if you want to define a tetramino you need to define a block with 4 squares with colors greater than 0, and 96 squares with a color of -1. In the same spirit, a map is by default initialized with squares having color -1.

8.6 The block object

Well, I said there was no OO programming and I talk about a "block object". OK, let's be clear, the "block object" is not an object from the programming point of view, but it's true that from a theorical point it's an object with fields (stored by the C++ engine) and methods (prefixed by "u61_block_"). However, things such as inheritance have no sense in U61's context, so forget about OO programming and just think about the block object as some data you can access trough a pre-defined API.

So this block object represents the "falling block". It is basically composed of squares. It can have from 1 to 100 squares, and can be of any shape. Each square has the following parameters:

- x : the x coordinate. The greater it is the more the square will be on the right.
- y : the y coordinate. The greater it is the more the sqare will be close to the bottom of the map.
- color : the color of the square. The color can range from 0 to 7. A negative value indicates that the square is disactivated.

The block also has its own x and y parameters. These parameters tell in fact the global position of the square within the map. Concretely, when a block square is drawn on the map, the coordinates used are (block_x+square_x,block_y+square_y). So for instance, to move a whole block down, one just has to increase its y value.

The block object also has some interesting functions such as "block_center", please check the script API to get a complete description of them.

8.7 The map object

The map object contains... everything but the block! Indeed, you need write access to the map object to send curses, to update your curse, and of course to modify the map.

One could categoruze the map attributes and methods like this:

- Square related functions, such as "u61_set_square_color". These are probably the most usefull functions since they allow you to change the aspect of the map. Basically, the map can be considered as an array of 10x25 squares, each square having a color (0-7 are regular colors and -1 means there's no square at all).
- Curse related functions, such as "u61_get_curse_y". With these you can move the special block associated to the next player's curse, you can send a curse to another player, or use an antidote.
- Global parameters handling. Score for instance can be accessed "u61_add_score".
- Custom values storage. If you write advanced scripts, you might wish to store some data and get it back later in another function. This is *not* possible with global Lua values, because of multiplayer issues, but some functions such as "u61_set_global" make it possible to store global numbers.

Chapter 9

Script callbacks reference

9.1 Introduction

User callbacks are lua functions which are called from the C++ engine. U61 requires all these functions to be declared in a script, even if they do nothing. In these user callbacks, you may use functions from the U61 system API, but be carefull, you can not use any system API functions in any user callback, since there are some access right constraints.

9.2 user_do_curse

9.2.1 Declaration

`user_do_curse(num,sent)`

- Arguments: a code previously returned by "user_new_curse", a boolean saying if the curse is local or comes from another player.
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.2.2 Description

This function can be called int 2 cases:

- When a square explosion has ended, and this square's location was the location of the current curse. If you are playing with the default theme, it means that the square with the blinking "?" has exploded.
- When another player has called the "u61_send_curse" function, and the current player was his default victim.

Basically, one could separate the curses into 2 categories:

- Those who affect the player which launches them. They usually have a positive effect. For instance, it can be a goodie which slows your blocks down so it's easier for you to play. These curses are executed directly by "user_do_curse", the parameter "sent" being set to 0 (false).

- Those who affect the launcher's victim. They usually have a negative effect. For instance, it can be a nasty curse which speeds up the blocks, so that they become uncontrollable. These curses are also executed by "user_do_curse" (with "sent" set to 0), but one has to call "u61_send_curse" manually to send the curse to the other player (the victim). This other player will receive the curse, but this time the "sent" parameter will be 1 (true). So the "sent" parameter is fundamental since it allows you to identify if the curse is to be sent to the victim or to be executed.

You might wonder why I did not simply create 2 categories of curses, and let the C++ engine handle all this "send to victim" business. Well, I liked the idea of letting the scripter control what's happening. For instance, you may create a special curse which sends severa curses (all the ones you currently have for instance) to your victim. Basically, what I described with the "sent" parameter determining if one has to send the curse or not is only a suggestion. You may use the system the way you want, for instance, it's possible to create a special curse which forces your victim to send a curse to its own victim...

Of course one of the big strenghs of U61 is that the "user_do_curse" and "u61_send_curse" work pretty well, even in a slow networked environment - at least I hope so 8-) -.

9.2.3 Example

The following example is a very simplified version of "user_do_curse". This function can indeed grow very big and it's a wise decision to make it call smaller functions. But still this purpose of this example is to remain simple. It has the 2 main types of curses:

- "ID_CURSE_PLUS" is a goodie which is executed locally and increases the player's score.

- "ID_CURSE_MINUS" is a nasty curse which reduces the score of the player's victim.

```
ID_CURSE_PLUS=3
ID_CURSE_MINUS=4

function user_do_curse(num,sent)
    if (num==ID_CURSE_MINUS) then
        if (sent==0) then
            u61_send_curse(num)
        else
            u61_add_score(-1000)
        end
    elseif (num==ID_CURSE_PLUS) then
        u61_add_score(1000)
    end
end
```

9.3 user_do_shape

9.3.1 Declaration

`user_do_shape(num)`

- Arguments: a number previously given by user_new_shape
- Return values: none
- Rights: block_read, block_write, map_read

9.3.2 Description

This function shapes the block before it appears on the screen. You should not include any random behavior in this function. In fact it is a bad idea to include random behaviors in U61 in general, but I mention it here for it might be very tempting to ignore the "num" parameter and do a random number generator oneself. But this is bound to fail in a network game for you can not garantee the remote players will see the right shape. In fact the "num" parameter "is" what U61 sents to other players, so it's only by using it that you can garantee game consistency.

A typical "user_do_shape" only needs to call the "u61_add_item" function. Indeed this function allows you to add squares to a block and this is what you generally need to do since at the very start of the function the block is empty, and you wish to end with a fully defined block. You should add least one square in evrey block, or the game might freeze.

Of course you can make the shape generation depend on a curse flag or something, but I feel that these kind of things should be done in the "u61_new_shape" instead. For instance, if you want to block the shape 0 for a while, it's IMHO better not to generate a 0 output in "u61_new_shape" than modify temporarly the creation of shape 0 in "u61_do_shape".

One important thing is to know that after "user_do_shape" has been executed, U61 always executes the "u61_center" function, which might alter the coordinates you have set. So you should not try to recognize a block later in the script by asking the coordinates of his squares, unless you are sure that "u61_center" can not alter them, or if you know in what way they have been changed. You may wonder why I chose this behavior, for it can seem anoying. Well, I just personnally estimate that an automatic "u61_center" is a comfortable thing to have, and it avoids badly centered blocks.

9.3.3 Example

This set of functions generates:

- A triangle with color 0 if num is 0
- A square with color 2 if num is 1
- A circle with color 5 if num is 2

```
function user_do_shape(num)
    if num==0 then
        triangle(0)
```

```
    elseif num==1 then
        square(2)
    else
        circle(5)
    end
end

function triangle(color)
    u61_add_item(0,0,color)
    u61_add_item(0,1,color)
    u61_add_item(1,1,color)
    u61_add_item(0,2,color)
    u61_add_item(2,2,color)
    u61_add_item(0,3,color)
    u61_add_item(1,3,color)
    u61_add_item(2,3,color)
    u61_add_item(3,3,color)
end

function square(color)
    u61_add_item(0,0,color)
    u61_add_item(1,0,color)
    u61_add_item(2,0,color)
    u61_add_item(3,0,color)
    u61_add_item(0,1,color)
    u61_add_item(3,1,color)
    u61_add_item(0,2,color)
    u61_add_item(3,2,color)
    u61_add_item(0,3,color)
    u61_add_item(1,3,color)
    u61_add_item(2,3,color)
    u61_add_item(3,3,color)
end

function circle(color)
    u61_add_item(1,0,color)
    u61_add_item(2,0,color)
    u61_add_item(0,1,color)
    u61_add_item(3,1,color)
    u61_add_item(0,2,color)
    u61_add_item(3,2,color)
    u61_add_item(1,3,color)
    u61_add_item(2,3,color)
end
```

9.4 user_get_curse_name

9.4.1 Declaration

`user_get_curse_name(num)`

- Arguments: a code previously returned by "user_new_curse"
- Return values: the name of the curse, which should be shown to the player
- Rights: map_read

9.4.2 Description

This function has a very simple goal: provide the user with a name for each curse. This name will be disaplyed in the status zone, under his map. Of course the game will run correctly if this functions returns a blank string, bug it will certainly not help the players.

The names should not be too long. At the time I write these lines, there's a limit of 10 characters, so if you exceed this limit the name will be truncated. And be carefull if your name uses lots of "W" and "M", for these letters are usually wider then plain "I"s.

9.4.3 Example

The following sample defines 2 curses names. A default name ("surprise") has been defined in case the curse is unknown, which should, it's true, never arrive if the code is consistent.

```
function user_get_curse_name(num)
    local name

    name="surprise"

    if (num==ID_CURSE_MALEDICTION_OF_GOOLOO) then
        name="gooloo"
    elseif (num==ID_CURSE_YOULLDIE) then
        name="you'll die"
    end

    return name
end
```

9.5 user_get_program

9.5.1 Declaration

`user_get_program()`

- Arguments: none

- Return values: the name of the program ("u61")
- Rights: none

9.5.2 Description

This function must be present so that the game engine makes sure the script is a valid U61 script. Of course the fact that this function does not exist or does not return the good value does not mean the script is actually error free, but it provides a good way to detect obvious errors such as using a totally wrecked file.

If this function is not present or does not return the correct value, U61 will display an error message each time the script is read.

9.5.3 Example

The code for this function should always be the same and look like this:

```
function user_get_program()
    return "u61"
end
```

9.6 user_get_version

9.6.1 Declaration

```
user_get_version()
```

- Arguments: none
- Return values: the version of the program
- Rights: none

9.6.2 Description

This function allows the game engine to know which version the script was designed for. This is very important, since the script API can slightly change between 2 different releases - of course the less it changes the better it is, but it's good to have a way to know the version anyway.

If this function is not present or does not return the correct value (the version of the U61 binary which should be used with the script), U61 will display an error message each time the script is read.

9.6.3 Example

The code for this function should look like this:

```
function user_get_version()
    return "1.0.0"
end
```

9.7 user_land

9.7.1 Declaration

`user_land()`

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read

9.7.2 Description

This function is called when it's not possible to move the block down any more. It's role is to modify the shape of the block before it is merged with the map. Most of the time, this function will do nothing, but you may wish - in some cases - to change the shape of the block just as it lands.

Let's take an example: you might consider that your blocks are not "solid" blocks, ie they break themselves into parts as they land. This is a very common trick in block-based games, and that's why it's available in U61.

You could also wish to send a curse as the block lands, or modify the map. This is not allowed because of performances issues, which lead me not to give the "map_write" right to this function. Indeed this function is very important for the "anticipation" mode. In this mode, the user can view in real-time where the block will land. This feature is achieved by calling the "user_move_down" function until "user_block_land" is required. And I have decided that such operations (as they are called very very often) should not require a full backup of the map, which can be quite CPU-consuming since the map structure can be quite big. It isn't that big now but I expect it to grow as I will add features to U61. So basically, just remember that there's a good reason not to modify the map in this function. However, it does not mean you have no callback at all when a block lands. Indeed, it is possible to set up a callback on a pattern match by using "user_match_pattern". This should - I hope - be enough in most cases.

9.7.3 Example

This example shows a function that makes blocks move up of one row as they land. This should make them hang in the air as they land. I can't figure out in what kind of context it could be usefull, but it's only a theorical example:

```
function user_land()
    u61_set_block_y(u61_get_block_y()-1)
end
```

9.8 user_match_pattern

9.8.1 Declaration

```
user_match_pattern(match_count)
```

- Arguments: the number of times "user_match_pattern" has already been called during last matching session.
- Return values: 1 if some matching shapes and/or colors have been found, 0 otherwise.
- Rights: map_read, map_write

9.8.2 Description

This is a fundamental function in U61. It is responsible for saying which squares should explode when a block lands. It is called after the block has landed, ie "user_land" has already been called when "user_match_pattern" is called. It is also responsible for incrementing the score of the player.

You'll notice that this function has no access rights on the block. This is because there's no more block when this function is called. Indeed, all the squares of the falled block are merged with the map. This means that if there were 10 squares in the map and 3 squares in the block, when this function is called you get an "all in one"map with 13 squares. By default, the block squares are put "as is" on the map, but if you want a more accurate control, you may tweak the "user_land" function.

The pattern to be matched can be anything, based on the position and the color of the map squares. You may search a completed horizontal line (as in the "classic") script, some sort of alignments, a shape, a sequence of colors, well, anything! I believe the "user_match_pattern" function is what makes a given set of rule realy different from the others.

Once the pattern is matched, it is a good idea to make the squares which should disappear explode. Technically, you could remove them right away (I've not tried it still), but I just think it's more user friendly for the player to see the block explode. The common method is to try and find a pattern, and when the first pattern is found, stop the search and make the founded pattern explode. Then, the Lua function is exited, and the game keeps on going. It's take a little time for squares to explode. While there's at least one square exploding on the map, U61 does not make a new block appear. Instead, it waits for the explosion to end, and when the explosion is ended, it runs the "user_match_pattern" function again. This way, it's possible to have cascading effects, and the player can understand why 12 of his squares disappeared while a pattern is usually made of 3 blocks.

This leads to a very important point: the "match_count" parameter. This parameter is equal to 0 when "user_match_pattern" is called for the first time. Then logically, "user_match_pattern" should make some squares explode (if needed of course) and let the squares alone while they blow up. When the explosion is finished, "user_match_pattern" is called again, but this time "match_count" is 1. Next time it will be 2, etc... When there's no more pattern to match, then U61's engine gives the player a new block, and resets the internal value corresponding to "match_count". This is a usefull feature for it allows an accurate control of the score, for instance, you might make something that gives:

- 100 points if there's only one pattern matched
- 300 points if there are two pattern matched

- 700 points if there are three pattern matched
- etc...

9.8.3 Example

The following example finds any horizontal line which is filled with squares, or has only one square missing. It's very close from the pattern match used in the "classic" script. Note that the code does not lie directly in user_match_pattern but in an external function which is more general and can be used in another set of rules if needed.

```
function user_match_pattern(match_count)
    return match_holed_line(match_count,1)
end

function match_holed_line(match_count,max_hole)
    local x
    local y
    local width
    local height
    local holes
    local lines

    width=u61_get_width()
    height=u61_get_height()

    lines=0
    y=0
    while y<height and lines==0 do
        holes=0
        x=0
        while x<width do
            if u61_get_square_color(x,y)<0 then
                holes=holes+1
            end
            if not (u61_is_square_exploding(x,y)==0) then
                holes=holes+1
            end
            x=x+1
        end
        if holes<max_hole then
            delete_line(y)
            lines=lines+1
        end
        y=y+1
    end

    if lines>0 then
        u61_add_score((match_count+1)*1000)
```

```
        if (match_count>=3) then
            u61_add_antidote()
        end
    end

    return lines
end
```

9.9 user_move_down

9.9.1 Declaration

`user_move_down()`

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read

9.9.2 Description

This function looks like "user_move_left" and "user_move_right" at first sight, but it is quite different. Indeed the "user_move_down" function must imperatively move the block down in some way. The reason is that this function is used in the following cases:

- The user presses the "move down" key (that's to say he presses the key which he has specified in his settings to be the "move down" key).
- A certain amount of time has ellapsed, and U61 decides to make the block move down. How long this delay is depends on the global speed of the game, which can be set in the game options, and generally increases while the game is running. And this is why it is important that one of the actions performed by "user_move_down" is actually to shift the block down. Indeed, one of U61 basic rules - the ones you can not change within the script -, is that "the block falls, and when it lands on other blocks, it freezes". If "user_move_down" does not move the block down, you could get a situation where your block never freezes and you never get any new block. Basically, a new block is sent to the user when a call to "user_move_down" has created a conflict between the block's squares and the map's squares. It is only in this case that the block will be freezed and that "user_land" and "user_match_pattern" will be called.
- The user presses the "drop" key, in this case "user_move_down" is called as often as possible until the block is stuck on the map. Again, if you ever make a "user_move_down" that does not lead to an incoherent position after a certain amount of calls, this function will loop for ever...

Please note that in this function you do not need to check wether the operation you want to perform is possible or not. As with other "user_move_..." or "user_rotate_..." functions, the C++ engine automatically checks if something is wrong, and if there's a conflict it rollbacks the whole operation.

9.9.3 Example

The following function moves the block down, and increments the value of a global value at the same time:

```
MOVE_DOWN_COUNTER=123

function user_move_down()
    u61_set_block_y(u61_get_block_y()+1)
    u61_set_global(MOVE_DOWN_COUNTER,u61_get_global(MOVE_DOWN_COUNTER)+1)
end
```

9.10 user_move_left

9.10.1 Declaration

```
user_move_left()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read

9.10.2 Description

This function can generally be called in 2 cases:

- The user has pressed the "Move left" key. (which key it is physcically depends on the user's settings). This is the most common case, and you had certainly thought of it.
- The "Rotate left" or "Rotate right" key has been pressed, and the "user_rotate_..." corresponding callback has put the block in such a state that there's a conflict between a square of the block and a square of the map. Therefore U61 automatically simulates a "Move left" or "Move right" key press, and then retries to execute the "user_rotate_...". This is an interesting feature since it enables the user to rotate his blocks even if the block is stuck to a wall.

Apart from this last point the "user_move_left" function is very similar to "user_rotate_left". One could certainly imagine scripts vhere "user_move_left" would not translate the block on the left. And even if in your script "user_move_left" perfor;s a left translation of the block, it is possible in this function to test if a given curse is active and change the function's behavior. This is what the "goofy" curse (in U61 0.2.2) does: it inverts the effects of "user_move_left" and "user_move_right", which makes the game a little... harder to control!

It's important to note that substracting 1 to the x coordinate of each square of the block won't make the block move to the left. Indeed after each call to "user_move_left" U61 centers the block (similar to a call to u61_center), so the only way to make a block translate is to use the "u61_set_block_x" function (or any associated function).

9.10.3 Example

This example function makes the block move left and down at the same time:

```
function user_move_left()
    u61_set_block_x(u61_get_block_x()-1)
    u61_set_block_y(u61_get_block_y()+1)
end
```

9.11 user_move_right

9.11.1 Declaration

`user_move_right()`

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read

9.11.2 Description

Similar to "user_move_left" but associated to the "Move right" key.

Like with the "user_rotate_..." functions, "user_move_right" does not have to be the exact contrary of "user_move_left". You are just free to imagine the weirdest behaviors.

9.11.3 Example

In this example, the block is shifted of 2 blocks right:

```
function user_move_right()
    u61_set_block_x(u61_get_block_x()+2)
end
```

9.12 user_new_curse

9.12.1 Declaration

`user_new_curse(num)`

- Arguments: a random value generated by the C++ engine
- Return values: the code of the curse that will be triggered if the current curse block explodes
- Rights: block_read, map_read

9.12.2 Description

This function is very similar to "user_new_shape". It's goal is to control the nature of the next curse which will be given to the player. Its only purpose is to return an integer which will be interpreted by "user_do_curse" later.

This function is called long before the player actually matches a pattern. Indeed, it is required to do so since the name of the next curse must be displayed in the player's information board/zone.

Of course, this "next curse" generation can be influenced by curses which affect the current player. For instance, you might decide that when a player is victim of the "THOU_SHALL_DIE" curse, then he can only get poor curses which have almost no effects on the opponents, and leave him in a hopeless position, bound to lose very soon. This is not fair practice but it's theorically possible 8-P

9.12.3 Example

The following example returns a curse code between 0 and 9 if the "ID_ZOUBIDA_MALEDICTION" curse is activated, and a code between 0 an 19 if not.

```
ID_ZOUBIDA_MALEDICTION=5

function user_new_curse(num)
    if u61_get_curse_age(ID_ZOUBIDA_MALEDICTION)<0 then
        num=mod(num,20)
    else
        num=mod(num,10)
    end

    return num
end
```

9.13 user_new_shape

9.13.1 Declaration

```
user_new_shape(num)
```

- Arguments: a random number between 0 and 1 000 000 000
- Return values: the code that should be use for the next call to "user_do_shape"
- Rights: block_read, map_read

9.13.2 Description

This function is usefull for you to control the codes used for shape generation. Basically, a set of scripts might require the shape number to be between 0 and 6. So in this case you will return a value between 0 and 6. The "num" argument is here to provide a randomly generated number. You are free to use

this value or not, but keep in mind that if you don't use it, it will be too late in the "user_do_shape" function to use some random numbers. You can consider this function as an intermediate between the random function and the function that generates the shape. Indeed the number it returns is used by the C++ engine to generate an internal event which will be sent to all remote players.

Tweaking this function can be very interesting if you want for instance to set up a mode where the player only receives blocks with shapes 34 an 125.

9.13.3 Example

This function returns a shape code between 0 and 6. More precisely, it returns:

- 40% of 0
- 20% of 1
- 10% of 2,3,4
- 5% of 5,6

```
user_new_shape(num)
    local result

    num=mod(num,100)

    if num>=60 then
        result=0
    elseif num>=50 then
        result=1
    elseif num>=30 then
        result=2
    elseif num>=20 then
        result=3
    elseif num>=10 then
        result=4
    elseif num>=5 then
        result=5
    else
        result=6
    end

    return result
end
```

9.14 user_rotate_left

9.14.1 Declaration

```
user_rotate_left()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read

9.14.2 Description

This function is called when the user presses the "Rotate left" key (what key exactly it is depends his key settings).

It may or may not perform a geometrical rotation. This is the case with the "classic.lua" script, but if you play with the "vertical.lua" script, you'll find out that rotation is in fact a "color shift". Basically, rotation should be any player launched function that alters your block but is not a plain translation.

If after executing the code of "user_rotate_left" there's a confict between a block square and a map square, I mean if they have the same absolute position, then the operation is automatically cancelled. This is one of the reasons you do not get "map_write" access in this function, indeed, the rollback of a map, especially the act of making a copy of a whole map in case the operation fails would in my opinion be exagerately time consuming, and I like the idea that "user_rotate_left" only affects the block.

This function is granted a "map_read" access, therefore you check if a curse is activated and modify the behavior of "user_rotate_left" dynamically.

9.14.3 Example

In order to show that "user_rotate_left" does not need to be a geometrical correction, this sample function performs a left/right flip of the block:

```
function user_rotate_left()
    local size
    local i

    size=u61_get_nb_items()
    i=0
    while i<size do
        u61_set_item_x(i,-u61_get_item_x(i))
        i=i+1
    end
end
```

9.15 user_rotate_right

9.15.1 Declaration

```
user_rotate_right()
```

- Arguments: none
- Return values: none

- Rights: block_read, block_write, map_read

9.15.2 Description

Exactly the same than "user_rotate_left". It's important to note that "user_rotate_left" does not need to be the contraryof "user_rotate_right". You may for instance decide that for a given script the "rotate left" and "rotate right" keys will correspond to 2 "special actions" which are by no way linked to each other. In fact the reason these functions are call "user_rotate_..." for is that in most scripts this name makes sense, and it would be IMHO a waste of time to redefine for each key and each script what label should be associated to keys.

9.15.3 Example

The following function is a function that increments the color of all the squares of the block:

```
function user_rotate_right()
    local size
    local i

    size=u61_get_nb_items()
    i=0
    while i<size do
        u61_set_item_color(i,mod(u61_get_item_color(i)+1,8))
        i=i+1
    end
end
```

9.16 user_square_blown_up

9.16.1 Declaration

```
user_square_blown_up(x,y)
```

- Arguments: coordinates of the square which blew up
- Return values: none
- Rights: map_read, map_write

9.16.2 Description

This function is called when a square has exploded. Explosions are usually started after a pattern match (see "user_match_pattern"), and then the C++ engine handles the explosion by himself, without calling any script. This takes a little time - during which the game keeps running - and when the explosion is finished, after say approximately 1/2 second, "user_square_blown_up" is called.

If the square that exploded had the same location that the current curse, then the "user_do_curse" function is launched. You do not have to do anything about this, it's just done automatically by the C++ engine.

Note that a single call to "user_match_pattern" can cause several squares to explode, so there will be several calls to "user_square_blown_up" - one per square to be precise -.

It's the responsibility of "user_square_blown_up" to change the map's structure after the square's explosion. If for instance the blown up square is to be replaced by the squares which are above it, one has to shift "manually" the squares down. Of course, you can decide that in some circumstances, the end of the explosion should cause the start of a new explosion elsewhere...

When this function is called, the square is still present on the map, and it still has its color. I mean that you can make the difference between a blue and a red blown up squares. This can be usefull if you want to set up cascading chains of explosions. Such effects can also be obtained by tweaking "user_match_pattern", but you may prefer to place your code in "user_square_blown_up", especially if your chain of explosion really depends on where the previous explosion was. The important thing to remember is that when there are no exploding squares and no pattern to match left, then the player gets a new block.

9.16.3 Example

The following function shifts down all the squares that are above the blown up square, and changes their color to that of the disappeared square. Note that this function moves the "curse" square if needed.

```
function user_square_blown_up(x,y)
    shift_column_down_and_color_it(x,y)
end

function shift_column_down_and_color_it(x_col,y_bottom)
    local y
    local color

    color=u61_get_square_color(x_col,y_bottom)
    y=y_bottom
    while u61_get_square_color(x_col,y-1)>=0 and y>0 do
        u61_set_square_color(x_col,y,color)
        y=y-1
    end
    u61_set_square_color(x_col,y,-1)

    if u61_get_curse_x()==x_col and u61_get_curse_y()<y_bottom then
        u61_set_curse_y(u61_get_curse_y()+1)
    end
end
```

9.17 user_start

9.17.1 Declaration

```
user_start()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.17.2 Description

This function is called at each "fresh start". A "fresh start" is when the player either enters the game or has just lost so his map and all its parameters but the score have been cleared.

A typical use is if you want the game to start with "some blocks" already placed so that it's harder to play.

It's also a good place to define global parameters such as map width and height.

9.17.3 Example

In this example, whenever a player starts a game, he's granted an antidote, and has the "STARTER" curse on him for 30 seconds.

```
ID_CURSE_STARTER=123

function user_start()
    u61_add_antidote()
    u61_register_curse(ID_CURSE_STARTER,3000,0)
end
```

9.18 user_time_callback_1

9.18.1 Declaration

```
user_time_callback_1()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.18.2 Description

This function is called once per second. It has all rights on the map and block, so it's possible to do anything with it. By default, it increments the score of the player by 1 point, since the longer you play, the better you are...

Another typical use of this function would be to check if a curse is active and execute a special function if the curse is set on.

9.18.3 Example

In this example, if the "ID_BAD_LUCK" curse is active, then the player's score is not incremented any more.

```
ID_BAD_LUCK=13

function user_time_callback_1()
    if (u61_get_curse_age(ID_BAD_LUCK)<0) then
        u61_add_score(1)
    end
end
```

9.19 user_time_callback_10

9.19.1 Declaration

```
user_time_callback_10()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.19.2 Description

This function is called 10 times per second. It has all rights on the map and block, so it's possible to do anything with it.

This is a very good place to code most of the curse effects. Indeed, doing something 10 times per second makes things look fast and smooth enough in many cases, and it's not *too* time consuming (at least not as much as placing the code in user_time_callback_100).

9.19.3 Example

In this example, if the "ID_SHIFT_LEFT" curse is active, then the block is moved to the left.

```
ID_SHIFT_LEFT=33

function user_time_callback_10()
    if (u61_get_curse_age(ID_SHIFT_LEFT)<0) then
        u61_set_block_x(u61_get_block_x()-1)
    end
end
```

9.20 user_time_callback_100

9.20.1 Declaration

```
user_time_callback_100()
```

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.20.2 Description

This function is called 100 times per second. It has all rights on the map and block, so it's possible to do anything with it.

Please take care not to put any complex code here, since it's executed very often. For instance, if there are 5 players in a game, it will be called 500 times per second, so if it takes 1 millisecond to be executed, then this function will eat up 50% of the CPU, so there won't be much left for other functions (which include all other Lua functions, C++ core engine, other tasks ran by the OS....). Basically, this function is here so that people who want a very accurate control of what's happening have this control, but in most cases one should try to use user_time_callback_10 or user_time_callback_1.

9.20.3 Example

In this example, if the "ID_SHIFT_RIGHT" curse is active, then the block is moved to the right. One could have - as in most cases - used the user_time_callback_10 function instead, only the block would have moved slower.

```
ID_SHIFT_RIGHT=66

function user_time_callback_100()
    if (u61_get_curse_age(ID_SHIFT_RIGHT)<0) then
        u61_set_block_x(u61_get_block_x()+1)
    end
end
```

9.21 user_use_antidote

9.21.1 Declaration

`user_use_antidote()`

- Arguments: none
- Return values: none
- Rights: block_read, block_write, map_read, map_write

9.21.2 Description

This function is called when the player presses the key associated to the "use antidote" function. It is a user callback since there can not be a generic function for this. Indeed, what has to be done for such an action really depends on how all the curses have been coded.

Here are a few examples of what one could code in this function:

- Remove the oldest curse. That's an obvious solution, when you use an antidote, it cures your oldest illness.
- Remove the curses in a special order, based on the severity of the curse. This way one could imagine that an antidote always cures the most annoying curse.
- Change the effect of curses. One could imagine special curses which the antidote can't really kill, but can make them less annoying, as if the desease were half cured.
- Invert the effect of the antidote. Indeed, a nice curse would be to put the player in such a state that the use of an antidote has the opposite effect of what he expects.

So as there are so many possibilities, I think it's a good thing to let the scripter code whatever he wants for this function. In fact, the one thing to remember about antidotes is that they appear in the status box of the player, and it's supposed to help the player.

9.21.3 Example

In this example, the antidote cures the oldest curse which affects the player, except for the "ID_ROTTEN_CURSE" curse. This makes the "rotten" curse an uncancellable curse, from an antidote point of view. Of course you can imagine that there are other ways to cure it, such as matching a special pattern for instance.

```
ID_ROTTEN_CURSE=66

function user_use_antidote()
    local oldest

    oldest=u61_get_oldest_curse(0)
    if oldest~=ID_ROTTEN_CURSE then
        u61_cancel_curse(oldest)
    end
end
```

Chapter 10

Script API reference

10.1 Introduction

The API functions are functions you can call within Lua, when you are scripting, to get information from the core C++ engine, and also modify what's happening in the game.

Technically speaking, these functions are just plain Lua functions mapped over C functions in the core engine, and these C functions call C++ methods off the "U61_Map" and "U61_Block" C++ objects. In this document, I will later say that the C++ engine calls some API fonctions automatically. In fact, it does not call the Lua functions but the C++ methods directly. But the result is exaclty the same, only the C++ engine does not make as many checks as the API functions do.

Those functions are quite "crash proof". I mean that various checks are triggered when you call them. For instance, calling a function which requires an array index with a bad, inexisting index will simply result in returning a default value, but there should not be (such a bug should be reported) a system crash of the core C++ engine.

The design of this API is not perfect, so maybe I'll create more functions later, since the functions I provide here are very low-level functions. However, I'll try and do my best not to remove any functions, so that in the long run scripts can be re-used easyly, and remain compatible with each other.

10.2 u61_add_antidote

10.2.1 Declaration

`u61_add_antidote()`

- Arguments: none.
- Return values: none.
- Rights: map_write.

10.2.2 Description

Gives an antidote to the current player.

The given antidote should appear in the status zone (a little heart in the default theme). The player is not forced to use it right away. In fact, as long as there's at least one antidote available, any press on the "use antidote" key will run the "user_use_antidote" function.

Note that even if the antidote does not appear in the status zone for there are already too many antidotes available, the "u61_add_antidote" will function correctly, and "u61_get_nb_antidotes" should always return the exact value.

10.2.3 Example

In this example, we give an antidote and a score bonus to the player.

```
function nice_present()
    u61_add_score(1000)
    u61_add_antidote()
end
```

10.3 u61_add_item

10.3.1 Declaration

```
u61_add_item(x,y,color)
```

- Arguments: coordinates and color of the square to add.
- Return values: none.
- Rights: block_write

10.3.2 Description

This function adds a new square to the current block. It is the function which should be called in the "user_do_shape" callback, to build new blocks before they start falling.

You can consider this function as an "append" method on the block object. In a way, the block is a vector of squares, which grows as you call "u61_add_item", and can be emptied with "u61_clear_block". If you had a block with 3 squares, after a call to this function, it will have 4 squares. The added square has an index of [n-1], if the size of the block is n.

Of course, the square properties which are set up by this function (position and color), can be changed later in the game.

10.3.3 Example

In this example, we add a square with a parameterd color at the right of the square which is already at the right extremity of the block.

```
function expand_right(color)
    local max_x
    local x
    local y
    local i

    max_x=0
    y=0

    i = u61_get_nb_items()-1
    while i>=0 do
        x=u61_get_item_x(i)
        if x>max_x then
            max_x=x
            y=u61_get_item_y(i)
        end
        i=i-1
    end

    u61_add_item(x,y,color)
end
```

10.4 u61_add_score

10.4.1 Declaration

```
u61_add_score(score_diff)
```

- Arguments: the value which should be added to the score.
- Return values: none.
- Rights: map_write.

10.4.2 Description

This function adds some points to the current score.

You can also use this function with negative values, if you want to decrease the score of a player. In fact, this function is really equivalent to a manual call to "u61_get_score" and then "u61_set_score". It is there to avoid you the pain of calling 2 functions where 1 is enough in most of the cases.

10.4.3 Example

The following function adds some points to the player only if the "NO_POINTS" curse is not activated.

```
ID_NO_POINTS=33
```

```
function tweaked_add_score(score_diff)
    if u61_get_curse_age(ID_NO_POINTS)<0 then
        u61_add_score(score_diff)
    end
end
```

10.5 u61_blow_up_square

10.5.1 Declaration

```
u61_blow_up_square(x,y)
```

- Arguments: the coordinates of a block in the map.
- Return values: none.
- Rights: map_write.

10.5.2 Description

This function starts an explosion at the given location, it should be used in the "user_match_pattern" function.

Of course, in the "user_match_pattenr" function, you could directly remove squares from the map, but you can get a nicer effect by calling the "u61_blow_up_square" function. It will indeed start an explosion on a given square, and at the end of the explosion - that's to say a few game cycles after the explosion has started -, the C++ core engine will call your "user_square_blown_up" callback. This way you may start an explosion, let the game run while the explosion is displayed to the player, and then get some control of what's happening at the end of the explosion with the "user_square_blown_up" function.

This function is very usefull if you want to create some cascading effects. In fact, I don't know about any other way to do it yet...

10.5.3 Example

This example blows up the bottom line of the map. What has to be done when the explosion is over is not defined here, for it is defined i the "user_square_blown_up" function.

```
function blow_bottom()
    local x
    local y

    y=u61_get_height()-1
    x=u61_get_width()-1
    while x>=0 do
        if u61_get_square_color(x,y)>=0 then
            u61_blow_up_square(x,y)
        end
```

```
        x=x-1
    end
end
```

10.6 u61_cancel_curse

10.6.1 Declaration

```
u61_cancel_curse(curse)
```

- Arguments: the id of a curse.
- Return values: none.
- Rights: map_write.

10.6.2 Description

This function cancels a persistent curse which had been set up with "u61_register_curse".

This function cures the illness caused by a curse by disactivating it. It can be used in the "user_use_antidote" function for instance. Of course any persistent curse is automatically removed with time, but this function is very usefull in the case of a pseudo-permanent curse which has a very long delay.

10.6.3 Example

In this example, we disable the "ID_CURSE_HUNGRY" curse manually.

```
ID_CURSE_HUNGRY=10

function feed_the_beast()
    u61_cancel_curse(ID_CURSE_HUNGRY)
end
```

10.7 u61_center_block

10.7.1 Declaration

```
u61_center_block()
```

- Arguments: None
- Return values: None
- Rights: block_write

10.7.2 Description

This function centers the block, this means that after it has been called, the gravity center of the block (approximately), will be located at (x,y), x and y being the global coordinates of the block.

This is very usefull when you want to code something like a rotation or a symetry. You can indeed code things in a simple manner using "x=-x" like algorithms, and then call "u61_center_block" to center the block. If you had not this function, there could be situations where after a sequence of rotations/symetries, a block could have moved to the left and/or right. Worse, a rotation could cause the block to move down, which is bad in a block-based game. Please note that this function does note change the global (x,y) values of the block (access by "u61_get_block_x" for instance). Its role is only to make (x,y) be the actual center of the block.

This function is automatically called by the script engine after the following user callbacks:

- user_rotate_left
- user_rotate_right
- user_move_left
- user_move_right
- user_move_down

10.7.3 Example

The following example leaves the block in a unchanged state. Indeed, the squares are translated, but "u61_center_block" cancels the translation.

```
function leave_block_unchanged()
    local i

    i = u61_get_nb_items()-1
    while i>=0 do
        u61_set_item_x(i,u61_get_item_x(i)+1)
        i=i-1
    end
    u61_center_block()
end
```

10.8 u61_clear_block

10.8.1 Declaration

```
u61_clear_block()
```

- Arguments: None
- Return values: None
- Rights: block_write

10.8.2 Description

This function clears the current block, this means it removes all the squares from it.

Beware not to leave a cleared, empty block falling in the players map, because it might cause some problems, since the C++ engine will not be able to find out when the block is supposed to have landed.

10.8.3 Example

In this example, we imagine that we want to change the shape of the block as it falls. Since "user_do_shape()" is always called on an empty block, the functions which prepare shapes usually do not clear the block first. So we need to force the block clear in we want to change completely the shape of the block outside the "user_do_shape" function.

```
function change_shape_to_tetramino_bar()
    u61_clear_block()
    tetramino_bar()
end
```

10.9 u61_clear_map

10.9.1 Declaration

```
u61_clear_map()
```

- Arguments: none.
- Return values: none.
- Rights: map_write.

10.9.2 Description

Clears the map, ie removes all the squares.

Keep in mind that this function clears all squares but does not perform a "total reset" since some parameters - such as active curses and score - still keep their values after the map has been cleared.

10.9.3 Example

This function puts a square in the middle of the map, after clearing it:

```
function square_in_the_middle()
    u61_clear_map()
    u61_set_square_color(u61_get_width()/2,u61_get_height()/2,1)
end
```

10.10 u61_delete_antidote

10.10.1 Declaration

```
u61_delete_antidote()
```

- Arguments: none.
- Return values: none.
- Rights: map_write.

10.10.2 Description

Removes an antidote from the player available antidotes.

Note that it's not necessary to call this function within the "user_use_antidote" function, since the number of antidotes is automatically decreased by the C++ core engine. However, you might want to use this function if you want to remove manually some antidotes.

If you call this function too many times, the number of antidotes will remain 0, it should never get negative, since the C++ engine will not allow it.

10.10.3 Example

This function clears all the antidotes of a player.

```
function clear_antidote()
    while u61_get_nb_antidotes()>0 do
        u61_delete_antidote()
    end
end
```

10.11 u61_get_anticipation_state

10.11.1 Declaration

```
u61_get_anticipation_state()
```

- Arguments: none.
- Return values: true (1) if anticipation mode is on, or false (0).
- Rights: map_read.

10.11.2 Description

This function returns the state of the anticipation mode, that means wether the player should see where the block will land if he presses the "drop" key right away.

You may have noticed that this parameter seems available in the player options menu, and the player might choose to have this information or not. However, the "u61_get_anticipation_state" function does not return the state of the menu item. Indeed, the "anticipation frame" is drawn only if the menu option has been set to "on" *and* the "u61_get_anticipation_state" returns true. So if you set the option to false in the menu, you will never see it, no matter what happens in the game.

10.11.3 Example

In this example, we return true if both anticipation and preview mode are set to true.

```
function is_game_easy()
    local result

    if u61_get_anticipation_state() and u61_get_preview_state() then
        result=1
    else
        result=0
    end

    return result
end
```

10.12 u61_get_block_x

10.12.1 Declaration

`u61_get_block_x()`

- Arguments: None
- Return values: The x coordinate of the block.
- Rights: block_read

10.12.2 Description

This function returns the x coordinate of the block. This value should approximately correspond to the center of gravity of the block.

10.12.3 Example

The following function returns the real position of a square in a block, ie the position this square has on the map.

```
function get_absolute_x(i)
    return u61_get_block_x()+u61_get_item_x(i)
end
```

10.13 u61_get_block_y

Exactly the same as "u61_get_block_x", but x becomes y...

10.14 u61_get_curse_age

10.14.1 Declaration

`u61_get_curse_age(curse)`

- Arguments: the id of a curse.
- Return values: the age of the curse.
- Rights: map_read.

10.14.2 Description

This function returns the time ellapsed since the last call to "u61_register_curse" with the same curse id. If the curse is not active at all, then the function returns -1.

The time is counted in game cycles. It takes 100 game cycles to make one second. It's in fact the same unit used in "u61_get_time()".

This function's primary goal is to allow the scripter to know wether a curse has been registered with "u61_register_curse". These registered curses appear in the player's status zone, and are representent by little skulls in the default theme. A typical use of this function would be to test if its return value is greater or equal to 0, and decide to disable and/or enable features in the game.

10.14.3 Example

In this example, we don't allow the user to use the rotate left key if he's doomed with the "ID_CURSE_NO_ROTATE" curse. It supposes there's a "low_level_rotate_left" function that does the dirty job of rotating the block.

```
ID_CURSE_NO_ROTATE=20

function my_rotate_left()
    if u61_get_curse_age(ID_CURSE_NO_ROTATE)<0 then
        low_level_rotate_left()
    end
end
```

10.15 u61_get_curse_state

10.15.1 Declaration

```
u61_get_curse_state()
```

- Arguments: none.
- Return values: 1 if the curse mode is on, 0 if not.
- Rights: map_read.

10.15.2 Description

Returns true if there should be a special "curse square" on the map.

This function is very different from "u61_is_curse_available". Indeed, it returns true if the map is in such a mode that a curse square could exists. But this does not garantee there's a curse square available. A trivial example is the empty map. If there are no squares at all on the map, you can oviously not have a special square anywhere. In fact, this function is merely to return the value set by "u61_set_curse_state" but in most of the cases what you'll need to call is "u61_is_curse_available".

10.15.3 Example

In this example, the player has his score freezed when he can't have any curse square on his map. This is a very unfair script since he is already disadvantaged by having no weapon at hand.

```
function add_score_unfair(points)
    if not (u61_get_curse_state()==0) then
        u61_add_score(points)
    end
end
```

10.16 u61_get_curse_x

10.16.1 Declaration

```
u61_get_curse_x()
```

- Arguments: none.
- Return values: the x coordinate of the "curse square" in the map.
- Rights: map_read.

10.16.2 Description

This function gives the x position of the "curse square". This special square is the black and white "?" in the default theme.

When this special square disappears (after an explosion), then the "user_do_curse" function is called. Of course you can start curses on other events, such as a special pattern match, but the explosion of this special square should remain the most common method to issue a curse - at least this is the way I view things, you may agree or not with me on this point.

This special square historically comes from EITtris. Since U61 is quite extensible I could have honestly imagined a more original and flexible way to launch curses but I was too lazy and I just like this idea since it reminds me of all the nights spent playing EITtris with 3 friends on a single computer in a small student room =8-)

10.16.3 Example

In this example, the player gets a score bonus if the curse square was located in the right or left columns of the map.

```
function bonus_if_curse_on_sides()
    local x

    x=u61_get_curse_x()

    if x==0 or x==(u61_get_width()-1) then
        u61_add_score(1000)
    end
end
```

10.17 u61_get_curse_y

Exactly the same as "u61_get_curse_x", but x becomes y...

10.18 u61_get_global

10.18.1 Declaration

```
u61_get_global(i)
```

- Arguments: the index of the global value.
- Return values: the value of the global.
- Rights: map_read.

10.18.2 Description

This function returns the global value associated to a given index (between 0 and 99).

This function is by no way related to the luac "getglobal" function. For more information about how to use it, see the documentation of "u61_set_global".

10.18.3 Example

This function returns the value of the ID_COUNTER value. Note that the lua object ID_COUNTER is a global lua value. However, you can use it safely since it is a constant. It's just a way to write things in a cleanier way. Without such constants, the code would rapidly become ununderstandable. But the value returned by the function is not a global lua value, it's a local value which has been initialized with a value stored internally by the C++ core engine.

```
ID_COUNTER=8

function get_counter()
    return u61_get_global(ID_COUNTER)
end
```

10.19 u61_get_height

10.19.1 Declaration

```
u61_get_height()
```

- Arguments: none.
- Return values: the height of the map.
- Rights: map_read.

10.19.2 Description

Returns the height of the map.

This function returns a value between 5 and 25. It is pretty much like the "u61_get_width" function, so please read the documentation of "u61_get_width" to understand why you should use this function.

10.19.3 Example

The following example colors the whole map, using the example function "colorize_row".

```
function colorize_map(color)
    local y

    y=u61_get_height()-1
```

```
    while y>=0 do
        colorize_row(y,color)
        y=y-1
    end
end
```

10.20 u61_get_item_color

10.20.1 Declaration

```
u61_get_item_color(i)
```

- Arguments: the index of the square.
- Return values: the color of the square.
- Rights: block_read

10.20.2 Description

Returns the color of a square in the block.

The color should normally always be a regular color (ie between 0 and 7), since having an empty square in a block does not really make any sense.

10.20.3 Example

The following function returns true if a given color is present in the block.

```
function is_color_present(color)
    local i
    local present

    present=0

    i = u61_get_nb_items()-1
    while i>=0 do
        if u61_get_item_color(i)==color then
            present=1
        end
        i=i-1
    end
end
```

10.21 u61_get_item_x

10.21.1 Declaration

```
u61_get_item_x(i)
```

- Arguments: the index of the square.
- Return values: the x coordinate of the square.
- Rights: block_read

10.21.2 Description

This function returns the x coordinate of a square in the block.

Note that the returned coordinate is not absolute. This means that if you want to have the real position of this square in the map, you'll have to call the "u61_get_block_x" function too and add the results.

10.21.3 Example

This function returns the width of a block, by getting the min and max x values.

```
function get_block_width()
    local i
    local min_x
    local max_x

    min_x=1
    max_x=-1

    i = u61_get_nb_items()-1
    while i>=0 do
        x=u61_get_item_x(i)
        min_x=min(min_x,x)
        max_x=max(max_x,x)
        i=i-1
    end

    return max_x-min_x
end
```

10.22 u61_get_item_y

Exactly the same as "u61_get_item_x", but x becomes y...

10.23 u61_get_nb_antidotes

10.23.1 Declaration

```
u61_get_nb_antidotes()
```

- Arguments: none.
- Return values: the number of antidotes available.
- Rights: map_read.

10.23.2 Description

This function returns the number of antidotes available. It is very similar to "u61_get_nb_curses".

The number of antidotes available is not updated by the C++ core engine. You need to give antidotes to players explicitly, by calling the "u61_add_antidote" function.

You can easily view the value this function would return by counting on the screen the number of little hearts (assuming you are playing with the default theme) in the status zone. Still, if there are too many antidotes available, only a limited number of them is displayed. Howvever, the "u61_get_nb_antidotes" will always return the exact value, even if it's too great to be displayed accurately.

10.23.3 Example

In this example, we increase the score of the player if he has more than 5 antidotes.

```
function update_score()
    if u61_get_nb_antidotes()>=5 then
        u61_add_score(10)
    end
end
```

10.24 u61_get_nb_curses

10.24.1 Declaration

```
u61_get_nb_curses(good)
```

- Arguments: the type of curse to get information about.
- Return values: the number of persistent curses currently active.
- Rights: map_read.

10.24.2 Description

This function returns the number of persistent curses which have been set up with "u61_register_curse".

Calling it with a value of 0 will return you the number of "bad" curses, and calling it with a value of 1 will return the number of "good" curses. For more informations on bad and good curses, see "u61_register_curse".

You can easily view the value this function would return by counting on the screen the number of little skulls (assuming you are playing with the default theme) in the status zone. Still, if there are too many curses activated, only a limited number of them is displayed. Howvever, the "u61_get_nb_curses" will always return the exact value, even if it's too great to be displayed accurately.

10.24.3 Example

In this example, we decrease the score of the player if he is affected by more than 5 persistent curses.

```
function update_score()
    if u61_get_nb_curses(0)>=5 then
        u61_add_score(-1)
    end
end
```

10.25 u61_get_nb_items

10.25.1 Declaration

```
u61_get_nb_items()
```

- Arguments: none
- Return values: the number of squares in the current block.
- Rights: block_read

10.25.2 Description

Returns the number of squares in the current block. It is very usefull if you want to program a loop on all the squares of the block.

If this function returns 4, it means that there are 4 squares available, which can be accessed with indexes 0,1,2 and 3. Any call outside this range to function like "u61_set_block_x" will simply do nothing.

10.25.3 Example

In the following example, the x and y coordinate of each square are exchanged.

```
function flip_xy()
```

```
    local i
    local x
    local y

    i = u61_get_nb_items()-1
    while i>=0 do
        x=u61_get_item_x(i)
        y=u61_get_item_y(i)
        x,y=y,x
        u61_set_item_x(i,x)
        u61_set_item_y(i,y)
        i=i-1
    end
end
```

10.26 u61_get_oldest_curse

10.26.1 Declaration

```
u61_get_oldest_curse(good)
```

- Arguments: the type of curse to get information about.
- Return values: the id of the oldest curse.
- Rights: map_read.

10.26.2 Description

This function returns the id of the oldest curse which has been set with "u61_register_curse".

Calling it with a value of 0 will return the oldest "bad" curse, and calling it with a value of 1 will return the oldest "good" curse. For more informations on bad and good curses, see "u61_register_curse".

It's important to be able to know rapidly which of the active persistent curses is the oldest one. A very simple example is a basic "user_use_antidote" where you want to remove a curse. You need a way to choose which one to delete. A possibility is to remove the oldest one, and this way the list of curse behaves like a FIFO (first in/first out) queue. And there you need the "u61_get_oldest_curse" function, or you would have to poll manually each curse to know his age and compare them...

The id returned by this function could typically be used with "u61_cancel_curse".

10.26.3 Example

In this example, we remove the oldest curse, except if it is the "ID_CURSE_BAD_LUCK" curse.

```
ID_CURSE_BAD_LUCK=13

function special_antidote()
```

```
    local curse

    curse=u61_get_oldest_curse(0)

    if curse~=ID_CURSE_BAD_LUCK then
        u61_cancel_curse(curse)
    end
end
```

10.27 u61_get_preview_state

10.27.1 Declaration

```
u61_get_preview_state()
```

- Arguments: none.
- Return values: true (1) if the preview should be displayed, of false(0).
- Rights: map_read.

10.27.2 Description

Returns the preview state of the player. The preview is what shows the player what kind of block he will get next time a block falls.

Unlike the anticipation state, the preview state can not be controlled from the menus. By default, it is set to true at the beginning of the game, and may be changed by calls to "u61_set_preview_state".

10.27.3 Example

This function gives more points to the player if preview state is off.

```
function tweaked_add_score(points)
    if u61_get_preview_state()==1 then
        u61_add_score(2*points)
    else
        u61_add_score(points)
    end
end
```

10.28 u61_get_score

10.28.1 Declaration

```
u61_get_score()
```

- Arguments: none.
- Return values: the current score.
- Rights: map_read.

10.28.2 Description

This function returns the current score.

One important thing about the score is that it is set to 0 each time the game restarts. The core C++ engine knows what is the player's highest score, but it does not inform the script about it on purpose. In fact, different high scores might be displayed for the same player on different machines. This occurs when someone has been playing for hours and has luckily got a high score of 999999. Then a remote player decides to connect himself to the game. In this case, he will not be informed that there's someone with a 999999 high score. I don't consider it a bug since it has been designed like this. High scores and frags are counted differently on every machine. Indeed, each machine calculates the high scores according to what it sees. What has happened before does not count at all. No matter if the guy playing on the server has a high score of 999999, what you see if how much he managed to score playing with you.

10.28.3 Example

This example returns true if the score is greater than a given value.

```
function is_score_enough(limit)
    local result

    result=0

    if u61_get_score()>=limit then
        result=1
    end
end
```

10.29 u61_get_square_color

10.29.1 Declaration

```
u61_get_square_color(x,y)
```

- Arguments: the coordinates of a square in the map.
- Return values: the color of the square.
- Rights: map_read.

10.29.2 Description

This function returns the color of a square in the map. The possible return values are:

- -1 if the square is not activated, ie there's no colored square at the given coordinates.
- 0-7 if there's a colored the square. There are 8 colors, ranging from 0 to 7.

This function is very important in the game, and you'll probably end up in using it all the time if you create scripts for U61. It is the best way to know if there's a square at a given position. You just have to call it and if it returns something greater than 0, then it means there's something.

It is important to note that it is quite different from "u61_get_item_color". Indeed, "u61_get_item_color" takes an index as a single argument, since the block is a vector of squares, whereas "u61_get_block_color" takes the coordinates of the square as an argument. This is because the map and block objects have fundamentaly different structures.

10.29.3 Example

In this example, we return 1 if there's a square at a given location, and 0 if there's none. This way we have a true/false function which tells us if the place is free.

```
function is_there_a_square(x,y)
    local exists

    exists=0

    if u61_get_square_color(x,y)>=0 then
        exists=1
    end

    return exists
end
```

10.30 u61_get_time

10.30.1 Declaration

```
u61_get_time()
```

- Arguments: none.
- Return values: the current system time.
- Rights: map_read.

10.30.2 Description

This function returns the system time associated to the map. The unit is 0.01 sec, this means that 100 equals one second.

The time can be very usefull, you can for instance use it as a pseudo-random value. This is very important, since the use of any other random value in a script could raise serious problems. Indeed, in U61, all the computers make all the calculus about all the players. So if a script generates internally a random number, then the game may not behave the same on 2 computers which are linked during a network game. By using the time value, you are sure that your pseudo-random number will be exactly the same on any computer, so the game will behave correctly. To sum up, the only pseudo-random values that should be use in U61 are:

- The value returned by "u61_get_time".
- The value returned by "u61_get_score".
- The argument of "user_new_block".
- The argument of "user_new_shape".

The time is always positive, but you should not assume that games start at time 0. Indeed, when a player looses, the game ends, and another game starts, but there's no time reset. This is for the core C++ engine to handle messages correctly. Therefore, the time value can get very great.

10.30.3 Example

In this example, we use the time as a pseudo-random value. The result is a random number between 0 and range-1.

```
function pseudo_random(range)
    return mod(u61_get_time(),range)
end
```

10.31 u61_get_width

10.31.1 Declaration

```
u61_get_width()
```

- Arguments: none.
- Return values: the width of the map.
- Rights: map_read

10.31.2 Description

Returns the width of the map.

This function returns a value between 2 and 10. In previous versions of U61 it used to return 10 all the time, since map size was not configurable. This is not the case anymore, and script programmers should not assume that map width is always 10 (the default value).

You might argue that since the map width is controlled by a user script (see u61_set_width), it's useless and cumbersome to call u61_get_width() when one knows that width is for instance of 8 because one called u61_set_width(8) before. My opinion is that it's much cleaner and less error prone to use u61_get_width() in all scripts.

10.31.3 Example

This function modify the color of a whole line in the map. Only the active colored squares are affected.

```
function colorize_row(y,color)
    local x

    x=u61_get_width()-1
    while x>=0 do
        if u61_get_square_color(x,y)>=0 then
            u61_set_square_color(x,y,color)
        end
        x=x-1
    end
end
```

10.32 u61_is_block_ok

10.32.1 Declaration

```
u61_is_block_ok()
```

- Arguments: none.
- Return values: 1 if the block is correctly placed, 0 if there's a conflict with the map.
- Rights: map_read, block_read.

10.32.2 Description

Returns true if the block is correctly placed, and has a correct shape. By "correct" we mean that:

- there are no squares in the block which are in conflict with the map squares, that's to say we could freeze the block as is without superposing a map square and a block square.

- there are no squares in the block that are outside the map, that's to say too much on the right, left or too low. If the block is too high it does not matter (think of when the block just start falling for instance, and you'll understand why it's obviously possible).

You may use this function when you want to move the block in a complex manner, or completely change it, and really need to keep an accurate control on what's happening. Indeed, this function is called very often by the C++ core engine, and for instance it's perfectly useless to call it in functions like "user_move_down" for instance, since the check is already performed, and you'll only slow down the game by calling it twice.

But a good example of when "u61_is_block_ok" is usefull is a script that needs to make the block cross the entire map, or move it for say at least 10 squares. Then the C++ core engine won't be able to detect if there are squares between the initial and final position. All what the C++ core engine is aware of is the initial and final positions. So this way the block might (from the player point of view) go through a solid wall, which is not what the scripter wants. The scripter wants the block to stop if there's a wall. So what he'll have to do is move the block by steps of 1 square and call "u61_is_block_ok" each time and stop if there's a problem.

It's usually not a problem if at the end of the script function you leave the block in an incorret state, with a conflict. Since the C++ core engine performs himself a check, he will move the block up or sideways so that it fits, or sometimes perform a plain rollback and cancel all your script functions. But if you want to control exactly what's happening, then you'll have to do it yourself in your script.

10.32.3 Example

In this example, the block is moved in both directions x and y, only if there are no squares on its path. The block is left in an incorrect state but this is not a problem since the C++ core engine will automatically cure the problem.

```
function translation_yx_safe(n)
    local i

    i=n
    while i>0 do
        u61_translate_x(1)
u61_translate_y(1)
        if u61_is_block_ok()==0 then
            i=0
        end
        i=i-1
    end
end
```

10.33 u61_is_curse_available

10.33.1 Declaration

```
u61_is_curse_available()
```

- Arguments: none.
- Return values: 1 if there is actually a special curse square on the map,0 if there is none.
- Rights: map_read.

10.33.2 Description

Returns true if there's a special curse square available, that's to say if the "curse state" parameter is set to true *and* the curse square is located on an active square in the map.

This function is very different from "u61_get_curse_state". Indeed, if the special curse square is badly placed, or if it is impossible to place it correctly, for instance when there are no squares at all, then you might get a situation where "u61_get_curse_state" returns true and "u61_is_square_available" returns false. For instance, this is often the case when a new game starts, since there are no squares on the map, and by default the "curse state" parameter is set to true.

10.33.3 Example

In this example, we return true if we the special curse square is badly placed, ie when "u61_get_curse_state" and "u61_is_curse_available" do not return the same value.

```
function is_curse_badly_placed()
    local result

    result=0

    if u61_get_curse_state()==1 and u61_is_curse_available()==0 then
        result=1
    end

    return result
end
```

10.34 u61_is_square_exploding

10.34.1 Declaration

```
u61_is_square_exploding(x,y)
```

- Arguments: the coordinates of a square in the map.
- Return values: 1 if the square is exploding, 0 if not.
- Rights: map_read.

10.34.2 Description

This function tells if the square is exploding, that's to say if the player can see an explosion at this location.

When this function is called, the square is "still there". Im mean that it is still possible to test its color for instance. Basically, a square is in the exploding state after "u61_blow_up_square" and before "u61_square_blown_up" have been called.

10.34.3 Example

This function counts all the exploding squares from the map.

```
function count_explosions()
    local x
    local y
    local count

    count=0

    y=u61_get_height()-1
    while y>=0 do
        x=u61_get_width()-1
        while x>=0 do
            if not (u61_is_square_exploding(x,y)==0) then
                count=count+1
            end
            x=x-1
        end
        y=y-1
    end

    return count
end
```

10.35 u61_register_curse

10.35.1 Declaration

```
u61_register_curse(curse,time,good)
```

- Arguments: the id of a curse, the time it should last, and the kind of curse it is.
- Return values: none.
- Rights: map_write.

10.35.2 Description

This function registers the curse, ie it adds it to the list of active curses. The curse will be active for a limited time, counted in 1/100 of seconds, but a value of 0 should make it last forever - or at least most player won't notice the difference since if you use 0 the curse will last for about 10 hours...

Used with the "u61_get_curse_age", this function allows you program "persistent" curses. By persistent curse I mean a curse which is displayed in the player's status zone (represented with a little skull in the default theme) and can be cancelled by an antidote. A typical non-persistent curse is a curse that fills your map with grabage. A typical persistent curse is the curse that forbids you to use some of your keyboard keys.

Curses set up with "u61_register_curse" can be cancelled manually with "u61_cancel_curse". Otherwise they will disappear automatically after some time.

The third parameter allows you to choose wether the curse is a real bad curse or if it's a "good" curse. Basically, it's up to you to decide which curses are good and which aren't. Basically a curse that accels the player is a bad one, and one that slows the player is a good one, since it makes the game easier. So in fact a "good" curse is an "anti-curse", it's a benediction opposed to a malediction. For U61's C++ engine, the only difference between good curses and bad curses is that the icons used to represent them on the screen are not the same.

10.35.3 Example

In this example, we have defined a function that handles remote curses (see the definition of "u61_send_curse"). And if the curse id is greater than 50, then we assume the curse is a persistent one, so we set it up for a time of 6000, that's to say about 1 minute.

```
function my_handle_remote_curse(curse)
    if curse>50 then
        u61_register_curse(curse,6000,0)
    end
end
```

10.36 u61_send_curse

10.36.1 Declaration

`u61_send_curse(curse)`

- Arguments: the id of a curse.
- Return values: none.
- Rights: map_write.

10.36.2 Description

This function sends a curse to the default victim of the current player.

It is probably one the most interesting function in U61. At least it is definitely *the* function which makes U61 a cool (?) game. Basically, this function sends a message over the network - if needed of course - to a remote player, and on the remote machine, the "user_do_curse" function is called. This allows a player to alter the map of another player.

A mojor pitfall is to get confused with the difference between what I call a local curse and a remote curse:

- A local curse is issued when the special "curse square" explodes for instance. The consequence is a call to "user_do_curse" in the map where the square disappeared with the second parameter set to 0. In most of the cases, a local curse will do something positive on the player's map, such as clearing it.

- A remote curse is issued after an explicit call to "u61_send_curse". The consequence is a call to "user_do_curse" in the map of the default victim (the name of the victim is displayed in the player's status zone), with the second parameter set to 1. Any script code that has a "map_write" access may call "u61_send_curse". This way, remote curses can be launched in many situations. In most of the cases, a remote curse will do something negative on the player'smap, such as filling it with garbage.

10.36.3 Example

In this example, the player sends a "SPECIAL_CURSE_1" and a "SPECIAL_CURSE_2" to its default victim. This function might be called within "user_do_curse", but it'snot necessary, you could also call it when the player has matched a very rare pattern.

```
SPECIAL_CURSE_1=51
SPECIAL_CURSE_2=52

function send_special_package()
    u61_send_curse(SPECIAL_CURSE_1)
    u61_send_curse(SPECIAL_CURSE_2)
end
```

10.37 u61_set_anticipation_state

10.37.1 Declaration

```
u61_set_anticipation_state(state)
```

- Arguments: a boolean (1 or 0) telling if this mode should be on or off.

- Return values: one.

- Rights: map_write.

10.37.2 Description

This function sets the state of the anticipation mode, that means wether the player should see where the block will land if he presses the "drop" key right away.

As mentionned in the doc of "u61_get_anticipation_state", this parameter is mixed with the value entered by the player in the menus, and the preview of where the block will land is drawn only if both are true. Basically, it should be easier to play with this mode set to true, but some players (me for instance) find that in the long run it's not so great to have it set on. So in your scripts, you might decide that a curse disables this mode (to make the game harder) and enables it back when the curse is over. But for players like me who do not like the anticipation mode, it would be annoying to see this mode appearing when I have said in my options that I do not want to have it. So it's OK to call this function with false if you want to make things a little harder, but just remember that it will do nothing on players who have the option set to false anyway.

10.37.3 Example

In this example, we set the mode to false or true according to a curse state.

```
ID_HALF_BLIND=8

function update_half_blind()
    if u61_get_curse_age(ID_HALF_BLIND)>=0 then
        u61_set_anticipation_state(0)
    else
        u61_set_anticipation_state(1)
    end
end
```

10.38 u61_set_block_x

10.38.1 Declaration

```
u61_set_block_x(x)
```

- Arguments: The x coordinate of the block.
- Return values: None
- Rights: block_write

10.38.2 Description

This function changes the global x coordinate of the block. Basically, it allows translation.

This function is very important, since it's not possible to perform a translation by calling "u61_set_item_x". Indeed, the "u61_center_block", which is called quite often, modifies the individual coordinates of each squares in a block so that it's centered on (x,y), x and y being set by "u61_set_block_x" and "u61_set_block_y". So you *need* to call this function if you want to translate the block horizontally.

10.38.3 Example

In this example, we move the block to the middle of the map.

```
function to_middle()
    u61_set_block_x(5)
end
```

10.39 u61_set_block_y

Exactly the same as "u61_set_block_x", but x becomes y...

10.40 u61_set_curse_state

10.40.1 Declaration

```
u61_set_curse_state(state)
```

- Arguments: 1 if the special curse square is to be activated, 0 if not.
- Return values: none.
- Rights: map_write.

10.40.2 Description

Sets the current "curse state", ie tells if there should be a special curse square available on the map.

As I'm writting this document, the "curse_state" parameter is set to true on a regular basis, so if you really want to remove the curse permanently, you have to call "u61_set_curse_state" continuously. By default, the curse will be disable for a given time (corresponding to a curse period, that's to say the delay until the curse moves and/or changes).

10.40.3 Example

In this example, we disable the special curse square when the user has more than 100000 points. This should prevent scores from getting too high...

```
function limit_score_with_curse()
    if u61_get_score()>100000 then
        u61_set_curse_state(0)
    end
end
```

10.41 u61_set_curse_x

10.41.1 Declaration

```
u61_set_curse_x(x)
```

- Arguments: the x coordinate of the special curse square.
- Return values: none.
- Rights: map_write.

10.41.2 Description

Sets the x position of the special curse square - that's to say the square which launches curses when it explodes, in the default theme it is the black and white "?".

Note that if you call this function with an invalid parameter, ie a position which is outside the map or where there's no active square, then the special curse square won't be visible. In fact, you'll end up in a situation where "u61_get_curse_state" returns true and "u61_is_curse_available" returns false, even if you have set the curse square as active with "u61_set_curse_state".

You should also keep in mind that it's not really necessary to call this function to move the curse square on a regular basis, since this is already done by the C++ core engine. In fact, this function is usefull when you mess up the whole map with calls to "u61_set_square_color", remove accidentally the square which was at the special square's location, and still want the special square to be available. In this case, you have to move the special curse square manually.

10.41.3 Example

In this example, we put the curse square on the first active square founded. The search starts at the bottom of the map.

```
function put_curse_on first()
    local x
    local y

    y=u61_get_height()-1
    while y>=0 do
        x=u61_get_width()-1
        while x>=0 do
            if u61_get_square_color(x,y)>=0 then
                u61_set_curse_x(x)
                u61_set_curse_y(y)
                x=-1
                y=-1
            end
            x=x-1
        end
        y=y-1
```

```
    end
end
```

10.42 u61_set_curse_y

Exactly the same as "u61_set_curse_x", but x becomes y...

10.43 u61_set_global

10.43.1 Declaration

`u61_set_global(i,value)`

- Arguments: an index and a value.
- Return values: none.
- Rights: map_write.

10.43.2 Description

This function updates the global value associated to a given index (between 0 and 99).

This function is by no way related to the luac "setglobal" function. Indeed, in U61, you must *not* use any global lua variable to store parameters. Lua globals can only be used to store constants. This is due to the fact that if you use global lua variables:

- They will be shared by all the players, which might not be what you want.
- They won't be shared accross the network, since the core engine does not transmit lua globals between computers.

Therefore, the chances that you end up with a wrecked game are important. Please do not try to do it, for this kind of bug never appears when you test your script with a single player, but will always occur during a thrilling internet play. See Murphy's theory.

So "u61_get_global" and "u61_set_global" are here to help you in case you want to have a persistent value stored. The values accessed with these functions are different for each player, and do not generate any inconsistency when used in a network game. Basically thay consist in a array of numbers, where you can store your own values, for instance parameters linked to a curse. These numbers are set to 0 each time a new game starts, ie each time the map is filled with squares and the player looses.

As of today, numbers are the only lua type which is usable with this function. It's actually a severe limitation, since being possible to store any lua object would make the writting of scripts a lot easier. However, I haven't yet managed to find a nice solution to this problem. It's true that lua offers the possibility to get a reference on any object, but I fear such a method would lead to memory problems - at least until I or someone else finds a way to handle references in a clean manner.

Please keep in mind that if you want to use a global value such as a boolean telling if the player has the right to do some special action, then "u61_register_curse" might be a better choice than "u61_set_global".

10.43.3 Example

The following function decrements the value of a global counter.

```
ID_COUNTER=8

function decrement_counter()
    u61_set_global(ID_COUNTER,u61_get_global()-1)
end
```

10.44 u61_set_height

10.44.1 Declaration

```
u61_set_height(h)
```

- Arguments: the new map height.
- Return values: none.
- Rights: map_write

10.44.2 Description

Changes the height of the map.

This function can safely be called "on the fly" while a player is playing and moving his block arround. All present squares will automatically be shifted so that the bottom squares stay at the bottom after resizing.

The height can range from 5 to 25. Note that depending on the block size, resizing the map to a "too small" size might make it simply impossible to play at all. Use with caution!

10.44.3 Example

The following function divides the height by 2:

```
function half_height()
    u61_set_height(u61_get_height()/2)
end
```

10.45 u61_set_item_color

10.45.1 Declaration

```
u61_set_item_color(i,color)
```

- Arguments: the index and color of the square.
- Return values: none.
- Rights: block_write

10.45.2 Description

Modifies the color of a square in the block.

The color should normally always be a regular color (ie between 0 and 7), since having an empty square in a block does not really make any sense.

10.45.3 Example

The following example sets all the squares of the block to the given color.

```
function set_color(color)
    local i

    i = u61_get_nb_items()-1
    while i>=0 do
        u61_set_item_color(i,color)
        i=i-1
    end
end
```

10.46 u61_set_item_x

10.46.1 Declaration

```
u61_set_item_x(i,x)
```

- Arguments: the index of the square, and its x coordinate.
- Return values: none.
- Rights: block_write

10.46.2 Description

This function modifies the x coordinate of a square in the block.

Note that the returned coordinate is not absolute. This means that if you want to set the real position of this square in the map, you'll need to call the "u61_get_block_x" function too.

10.46.3 Example

This function sets the absolute position of a square in the block, ie its position in the map. So wherever the block may be, calling "set_absolute_x(0,0)" will place the first square of the block in the leftmost column of the map.

```
function set_absolute_x(i,x)
    u61_set_item_x(i,x-u61_get_block_x())
end
```

10.47 u61_set_item_y

Exactly the same as "u61_set_item_x", but x becomes y...

10.48 u61_set_preview_state

10.48.1 Declaration

```
u61_set_preview_state(state)
```

- Arguments: the state for the preview mode (1=on, 0=off).
- Return values: none.
- Rights:

10.48.2 Description

Sets the preview state of the player. The preview is what shows the player what kind of block he will get next time a block falls.

This function would typically used in a curse, to remove the great help provided by the preview, and put the player in a difficult position.

10.48.3 Example

This example sets the preview mode to false if the score is greater than 100000. It assumes that any player capable of getting such a high score is good enough to play without a preview.

```
function disable_preview_for_good_players()
    if u61_get_score()>100000 then
        u61_set_preview_state(0)
    end
end
```

10.49 u61_set_score

10.49.1 Declaration

```
u61_set_score(score)
```

- Arguments: the new score.
- Return values: none.
- Rights: map_write.

10.49.2 Description

This function updates the current score.

You'll have to remember that scores should not exceed 999.999, so that they can fit on 6 digits. However, if you exceed this limit, the value will be truncated to 999.999 anyway.

10.49.3 Example

The following example doubles the score of the player.

```
function double_score()
    u61_set_score(u61_get_score()*2)
end
```

10.50 u61_set_square_color

10.50.1 Declaration

```
u61_set_square_color(x,y,color)
```

- Arguments: the coordinates of a square, and its color.
- Return values: none.
- Rights: map_write.

10.50.2 Description

This function sets the color of a square in the map. The possible values for color are:

- -1 if the square is to be disabled.
- 0-7 if one whishes to put a colored square in this location.

Any call with an invalid color (lower than 0 or greater than 7) will result in setting the color to -1. The question is: "why should the game be limited to 8 colors?". well, in fact, I had to fix a limited number of colors, since theme creation would have been too complex with a flexible number of colors. Indeed, you would have risked to get a 16-color based script with an 8-color based theme. And then you can't play since you can not distinguish all types of squares. So I had to set up a limit. I chose 8 and think it's a good compromise between rich gameplay and easyness of theme creation.

In the scripts and themes I have created, I have tried to respect the following rules:

- Colors 0-6 are standard colors for blocks.
- Color 7 is a special color, used for squares that are generated by curses and which in a general manner don't come with standard blocks. In the "classic" theme, theses are the grey squares.

However, this is just a convention, and you can bypass it. Still, players might find the game more consistent if you respect it,

10.50.3 Example

This example moves all the squares from the current block to the map. This is what is done before the "user_match_pattern" function is called. However, you should never need such a function - it's only an example - since this work is automatically done by the C++ core engine.

```
function block_to_map()
    local i
    local x
    local y
    local color

    i = u61_get_nb_items()-1
    while i>=0 do
        x=u61_get_item_x(i)+u61_get_block_x(i)
        y=u61_get_item_y(i)+u61_get_block_y(i)
        color=u61_get_item_color(i)
        u61_set_square_color(x,y,color)
        i=i-1
    end
    u61_clear_block()
end
```

10.51 u61_set_width

10.51.1 Declaration

```
u61_set_width(w)
```

- Arguments: the new map width.
- Return values: none.

- Rights: map_write

10.51.2 Description

Changes the width of the map.

This function can safely be called "on the fly" while a player is playing and moving his block arround. All present squares will automatically be shifted and/or deleted so that they are globally "in the middle" of the resized map.

The width can range from 2 to 10. Note that depending on the block size, resizing the map to a "too small" size might make it simply impossible to play at all. Use with caution!

10.51.3 Example

The following function doubles the width of the map:

```
function double_width()
    u61_set_width(u61_get_width()*2)
end
```

10.52 u61_shift_map

10.52.1 Declaration

```
u61_shift_map(x,y)
```

- Arguments: 2 integers which define how squares will be translated.
- Return values: none.
- Rights: map_write

10.52.2 Description

Shifts/translates all the map squares.

This function can be usefull when you've just resized a map and/or want to move all the squares in the map at once. Note that it's automatically called whenever the map is resized, still the default behaviour might not be the one you expect.

Squares which are outside the map limits after being translated are simply deleted/ignored.

10.52.3 Example

The following function shifts the map down:

```
function shift_down()
    u61_shift_map(0,1)
end
```

Chapter 11

Script library reference

11.1 Introduction

The library functions are Lua functions I have created and they are used by the scripts which are distributed with U61.

Basically, you could create your own set of rules from scratch, by filling user callbacks with pure calls to the core API. I honestly think it's too bad not to re-use some of the work I've already done. So a more reasonnable approach would be, in my opinion, to take an existing script and modify it until it fits your need. And this section's goal is to provide a documentation on what I have programmed my own scripts.

Basically, this section has 2 goals:

- Give an overview of what alreayd exists, and where to find it. For instance, if you want to modify a curse in the "classic" set of rules, you might find here in which file this curse is coded, and what resources it uses.

- Set up some coding guidelines. Of course since U61 is GPL'ed, you can modify the code the way you want. However, experience shows that no project can survive without a minimum of rules. Most of the rules I usein scripting for U61 are based on common sense and should not surprise anyone.

If you look in the source package, you'll find 2 subdirectories: "library" and "package":

- "library" contains most of the code, which is split in several modules, each ".lua" file being one of these modules.

- "package" defines the user callbacks. Each package requires functions from the library. The idea is that you can create multiples packages which rely on the same library modules. This way, when a library module gets some bug-fix or improvement, all the packages that uses it can benefit from the update.

11.2 Coding guidelines

11.2.1 Why?

If you code new scripts for U61, and want them included in the main U61 release, I'd really appreciate that you respect some basic rules, which should make the code easier to maintain. This is not to forbid you anything and should not endanger your freedom (remember the game is GPL'ed) but without a minimum of organization, I'm pretty sure it will be impossible to maintain an important database of scripts. And I'd just love it if zillions of Lua scripts for U61 could be available on the net.

11.2.2 Files

I never build complete U61 rules within a single file. The method I use is to create many little files and then cat them together. Of course if you are trying new features, it might be usefull to take an existing script, for instance "classic.lua", and edit it directly. And if you want to share this script with other people, you can put it on ftp for download without even asking me. BTW, if you do that, a good idea would be to rename the script. Indeed, in a network game, clients can automatically save the server script on their hard drive, and they might get confused if there are 33 different "classic.lua" scripts. So "booby-classic5.lua" would certainly be a much better file name.

But if you write some script and want me to include it in U61's main distribution, then it will truely make things easier for me if you use the method I used. If you want to look how I have set up my scripts, please look in the "script" folder of the source package.

As of today, if I ever get enough contributed scripts, this is what I'll do:

- Original and well done scripts will be included in the main distribution, that's to say they'll be added to the current scripts, and will use the "library-package" structure.
- Any other other script, as long as it works (no Lua errors, game launchable), will go in a "u61-contrib" package which will contain raw scripts, saved as is.

Of course, as I already mentionned, this is not limitative, if you want to make your own U61 distribution you can do it, as long as it's still GPL'ed. However I'm ready to collect scripts and coordinate the whole project, so you're free to accept my offer or not 8-)

11.2.3 Functions

Functions should always be prefixed with the name of the file they are in. This applies to all functions which are defined in the "library" directory. This way, there should never be any conflict between function names. Basically:

- Script callback functions are prefixed with "user_".
- Script API functions are prefixed with "u61_".
- Script library functions are prefixed with "filename_". So a function which is defined in "supercurse.lua" could be name "supercurse_hello_world" for instance. BTW, this implies there should not be any "_" in file names, so that it easier to figure out where the filename ends.

11.2.4 Constants

Global constants should always be prefixed with "ID_". As of today, there are 2 types of constants in U61:

- Curse indexes. They are prefixed by "ID_CURSE_".
- Global indexes. They are prefixed by "ID_GLOBAL_".

11.3 Modules

11.3.1 What's this?

Since I do not have the time to document all the script library functions in detail, I have decided to provide (at least) a list of the different modules available, and give a short description of what they do. If you want to know more about them, read the code!

11.3.2 antedote.lua

Contains the code for the "ANTEDOTE" curse, which gives an antedote to the player.

11.3.3 bigx.lua

Contains the code for the "BUGX" curse. The curse replaces the current block by a big X shape.

11.3.4 boost.lua

Contains the code for the "BOOST" curse, which makes the blocks fall very fast for a few seconds.

11.3.5 clear.lua

Contains the code for the "CLEAR" curse. This curse clears the map, ie removes all the squares.

11.3.6 color.lua

Contains function to match color patterns, such as aligned squares of the same color.

11.3.7 colored.lua

Contains code to generate colored bars, as in the "vertical" set of rules.

11.3.8 connected.lua

Contains a powerfull - but yet the code is ugly 8-} - match function, which search any tetramino, pentamino, hexamino. Well, in fact it just recognizes any sequence of adjacent squares of the same color.

11.3.9 drop.lua

Contains the code for the "DROP" curse, which forces the block to be dropped.

11.3.10 cycle.lua

Contains some functions to shift the colors of the block. This can be an alternative to geometrical rotations when the rules are heavily based on colors.

11.3.11 fiftytwo.lua

Contains the code for the "FIFTYTWO" curse. When affected by this curse, the user will only get "5" and "2" like tetraminos.

11.3.12 goofy.lua

Contains the code for the "GOOFY" curse. This curse inverts the left and right commands.

11.3.13 haha.lua

Contains the code for the "HAHA" curse. This curse writes "HA HA" in the map.

11.3.14 holedline.lua

Contains the code for the "HOLEDLINE" curse. This curse adds a line with a missing square at the bottom of the map.

11.3.15 line.lua

Contains the code to match a line pattern. This means that it detects whenever there's a completed horizontal line of squares, of any color.

11.3.16 metamorphose.lua

Contains the code for the "METAMORPHOSE" curse. It changes the current block to another block.

11.3.17 mixedline.lua

Contains the code for the "MIXEDLINE" curse. It adds a colored line at the bottom of the map.

11.3.18 morecolors.lua

Contains the code for the "MORECOLORS" curse. This curse causes some functions to use 7 colors instead of 6, which can sometimes make the game harder.

11.3.19 nohands.lua

Contains the code for the "NOHANDS" curse. This curse disables the control of the block, like if keyboard was disabled.

11.3.20 onlybars.lua

Contains the code for the "ONLYBARS" curse. This curse is a friendly one: it gives the player vertical bars for a given time, which makes the game easier.

11.3.21 rotation.lua

Contains some functions to perform geometrical rotations.

11.3.22 shuffle.lua

Contains the code for the "SHUFFLE" curse. This curse shuffles the map so that evrything gets messy.

11.3.23 sides.lua

Contains the code for the "SIDES" curse. This curse moves the block to the right or left of the map.

11.3.24 split.lua

Contains the code to "split" a block as it lands. It is used for instance in the "user_land" callback in the "couple" set of rules.

11.3.25 stairs.lua

Contains the code for the "STAIRS" curse. This curse draws stairs on the map.

11.3.26 tetramino.lua

Contains the code to generate tetraminos. Tetraminos are blocked formed of 4 squares.

11.3.27 topdown.lua

Contains the code for the "TOPDOWN" curse, which inverts the map, ie puts the blocks that were at the bottom on top.

11.3.28 topholedline.lua

Contains the code for the "TOPHOLEDLINE" curse, which puts 2 lines with holes in them at the top of the map.

11.3.29 translation.lua

Contains some functions to perform translations of the block without calling 2 API functions (a "get" and a "set").

11.3.30 unstable.lua

Contains the code for the "UNSTABLE" curse which makes the map unstable. Contributed by Jimmy Kaplowitz.

11.3.31 utils.lua

Contains various utility fonctions, which are not linked to any peculiar curse or set of rules.

11.3.32 wind.lua

Contains the code for the "WIND" curse. This curse causes the block to move on the left on a regular basis.

11.4 Constants

11.4.1 What's this?

This section describes the constants used in the scripts. As you can see, these constants are all defined in capitals, and I used the "ID_" prefix for all of them. I suggest you do the same if you add some, it should keep things simpler 8-)

I won't describe how these constants work in detail, I just give the name of the constants and their value. Please do not assume that, for instance, ID_CURSE_GOOFY is equal to 1, for these values may change some day. The right method when you need a new constant is to pickup a free number, assign it to a variable with a readable name, and then always use this variable.

11.4.2 Curses

- 0 : ID_CURSE_ANTEDOTE

- 1 : ID_CURSE_HOLEDLINE
- 2 : ID_CURSE_GOOFY
- 3 : ID_CURSE_CLEAR
- 4 : ID_CURSE_SHUFFLE
- 5 : ID_CURSE_WIND
- 6 : ID_CURSE_FIFTYONE
- 7 : ID_CURSE_BIGX
- 8 : ID_CURSE_MIXEDLINE
- 9 : ID_CURSE_MORECOLORS
- 10 : ID_CURSE_BOOST
- 11 : ID_CURSE_DROP
- 12 : ID_CURSE_TOPDOWN
- 13 : ID_CURSE_TOPHOLEDLINE
- 14 : ID_CURSE_STAIRS
- 15 : ID_CURSE_HAHA
- 16 : ID_CURSE_NOHANDS
- 17 : ID_CURSE_METAMORPHOSE
- 18 : ID_CURSE_SIDES
- 19 : ID_CURSE_ONLYBARS
- 20 : ID_CURSE_UNSTABLE

11.4.3 Globals

- 0 : ID_GLOBAL_WIND_COUNTER
- 1 : ID_GLOBAL_WIND_BLOCK_X

Chapter 12

Technical notes

12.1 Global architecture

12.1.1 About C++ and Lua

U61 is a (rather big) monolithic C++ program, which embeds an Lua interpreter. This interpreter reads and runs Lua scripts which can be tweaked by players to adjust rules to their taste.

There's much more C++ code than Lua scripts. Indeed Lua is used almost nowhere in the game but for rules definition.

As I get more and more experience in programming I feel that this is not especially a very good design. Indeed there's plenty of inelegant and poorly written C++ code in U61, which would have been clearly feasible in any common scripting language, be it Lua, Python, Perl, Scheme, whatever. There are several reasons which led me to chose not to use Lua more often in the game:

- I wanted to clearly separate the C++ game core from the Lua user scripts, although I did not really realize that this separation could have been done even if I had used Lua in the game core. I would certainly have needed to set up different environments for the Lua interpreters, but it was possible.
- I wanted the game core to run as fast as possible, and did not want to waste CPU time by using interpreted code in it. Reality shows that ugly akward C++ code rarely runs fast and U61 is no exception. It's slow for what it does. Mark my words, I did not say it would not run fast on your computer, only it could drop down from 30% CPU used to something like maybe 3% if it had been written in a smarter way.
- When I first coded U61, C++ was still a language I estimated, and I really thought I'd make great things with it. Experience showed me that me and C++ are definitely not made for each other. Now I really hate this language, find it ugly, hard to program, consider that its object-oriented aspect is very complicated to master, especially since I've used scripting languages like Python, where OO programming does make sense, and does not look like an ugly pile of hacks and tricks.
- Lack of experience. When I started U61, I did not really know much about scripting languages like Python (http://www.python.org) and ignored cool tools like SWIG (http://www.swig.org/). With these tools I would have ended up with a much cleaner design than the current monster

C++ core. I would still have chosen Lua as an extension language for it perfectly fits, but for instance all the GUI (menus) would have been better written in Python.

12.1.2 ClanLib

U61 clearly depends on ClanLib for most of its tasks, including graphics and sounds.

However, if you look at the code you'll notice there are many features present in ClanLib, that I do not use in U61. Worse, I have my own - and often limited and buggy - implementation of a bunch of things, for instance:

- network communication
- menu handling

There's a good reason for this: U61 is a "long term project" for me, and has been arround for already 3 years. In 3 years, ClanLib has moved from version 0.2.x to 0.7.x, and a large part of its API has changed.

ClanLib, as of version 0.2.x, was almost all I needed for a game library. My only real problem with it was that it was not completely finished, it lacked a few minor features, and some stability, but basically things were there.

Point is that over the years the ClanLib folks have added an impressive number of new features, also fixed a bunch of design flaws, but doing this they have broken compatibility several times, with honestly very few concrete improvements from my egocentric "focused on U61" point of view.

As I write these lines ClanLib seems to aim at being a complete game developpement environment with support of fancy hardware acceleration and so on. But I do not need this at all for U61. U61 has very basic graphical needs, won't really gain anything with OpenGL, has its GUI already hard-coded. All it needs is just plain old ClanLib 0.2.x with a few patch applied - I'm a bit exagerating, but not that much.

So for instance, concerning network code, I've simply moved away from ClanLib. When Magnus Norddahl (ClanLib's main developper/project leader) came with his ClanNetwork 2 implementation, I did not have the courage to make the migration from ClanNetwork 1 to ClanNetwork 2. No matter how many fancy options ClanNetwork 2 has, what I needed was to be 100% sure that I would never ever need to rewrite my network code when changing to ClanNetwork 3, 4, 5 etc. Added to this, I know a bit about POSIX socket programming, so changing form ClanNetwork 1 to POSIX sockets was even simpler for me than changing to ClanNetwork 2. Now U61 does not need ClanNetwork any more, it has its own low-level network API wrapper, and even if it's poorly written it compiles and performs good enough for U61 needs.

As long as ClanLib 0.6.x is out and that there's a reasonnably easy way to compile U61 "as is" with some ClanLib version, U61 will still use ClanLib. However, should ClanLib's display and/or sound API change, I might as well completely switch away from ClanLib to some more "API-stable" library, maybe SDL (http://ww.libsdl.org) or Allegro (http://alleg.sourceforge.net). AFAIK ClanLib O.7.x has a different way to handle graphics than ClanLib 0.6.x. What I will do will depend on how tricky it will be to switch from 0.6.x to 0.7.x.

This is sad for ClanLib is a remarkable piece of software, is well written and feature rich, but the fact that its API changes so often makes it rather complicated for me to develop with it, especially when my projects (like U61) take several years to be completed, and are supposed to be also avaiblable several years after they've been completed.

12.2 Security

12.2.1 Buffer overflows

Let's be clear: U61 is a game and it's no heavily secured enterprise bullet proof application.

However I tried and did my best to provide a secured enough environment. For instance, all string manipulations are done using snprintf-like functions, instead of the default sprintf-like functions, which are known as being very likely to create vulnerabilities.

Still, there might be some security hole in U61 - as in any program BTW - so use it at your own risks, and do not run it as root or any super user account!

12.2.2 Remote Lua code execution

One of the main features of U61 is that its rules are customizable. A consequence of this is that in network games, all the clients download the game rules from the server, which is an Lua source code, and execute this script.

So U61 does execute remote code within its embedded Lua interpreter, and does not perform any advanced checking before executing it. From a theorical point of view, this is a very good place to put some sort of virus or trojan horse.

Still, this code is executed in a restricted environment, it should have no access to I/O functions which could allow it to retrieve files from the hard drive, communicate using the network interface and so on. So in practice, I've taken the necessary steps to secure this aspect of the program and no trouble should come from it.

However, you should be warned that U61 gets files from the network, executes them and stores them on your hard drive.

BTW this is exactly what a web browser does when you view a page containing scripts. It gets the HTML file from the network, executes the JavaScript code in it, and stores it on your hard drive in some cache folder. Experience shows that most browsers have had security holes - most of them being fixed rapidly - so I'd be very lucky if U61 had no potential security hole at all 8-)

12.2.3 Passwords

The network password is stored as plain text on your hard drive, in the "config.lua" file, so anyone capable of reading files on your account can technically read this password. So it's wise to use a "weak" password like "hello".

However, passwords are not sent as plain text on the network, instead U61 sends an MD5 checksum of the password, so it's not possible to use a sniffer to capture the password when you join a network game.

12.3 Network model

12.3.1 U61 is different

Most modern network games use a client/server model, where the server maintains a "game state". The clients send informations describing what the player did, the server processes these informations, updates the "game state" and sends informations back to the client.

This model has several advantages, but one of its big drawbacks is that when a client has a slow connection with the server the game becomes "laggy", and is usually no more fun to play.

U61 does not use this model. In fact, both server and client maintain their own "game state", and the only information which is normally sent on the network is messages like "player X pressed key Y".

12.3.2 Network lag

The reason I chose this design in the first place is that it allows each client/server to handle its own players in real-time, without waiting for informations coming from the server.

This way, players on a slow-linked client never really "feel" the lag with the server. From their point of view everything is done locally, when they press a key, the effect of this key is immediate and they do not need to wait until the information "I want this block to be dropped" is sent to the server, processed, and re-sent back to the client.

Of course, this is only true for local players, there can still be some lag with remote players. U61 solves this problem by "anticipating" the remote players map states. Before showing remote players maps, it calculates the map state "as it would be now if the player had pressed no key at all". Whenever it receives a key stroke event, it performs a sort of "rollback", and corrects the map so that it takes the key stroke in account.

This explains that when you play over a slow link on the network, you might see little quirks in remote players maps behaviours, with some impossible moves. However my opinion is that this is a lesser problem than true real lag which forbids smooth play.

12.3.3 Synchronization

Another important issue is synchronization between clients. Indeed, since each client maintains its own game state for all players, a bug in the implementation could easily lead to a total desynchronization of the game. By desynchronization I mean that after some times the maps for a given player could be completely different on different computers.

Since only key strokes are transmitted, this is theoricall possible, the implementation bug being "a key stroke does not have the same effect on 2 different computers".

This explains why it's impossible to use the builtin system random number generator while writing Lua scripts for U61. Indeed the generated random number wouldn't be the same on each computer (or one would need to "seed" it all the time, and it would not be really random anymore...).

12.3.4 Cheating

Another nice aspect of this network model is that it's practically impossible to cheat. Indeed a player can't really "fake" his map, since its map state is calculated from key strokes on every networked computer.

Running a modified server wouldn't even really allow someone to cheat since clients would continue to process key strokes in a normal way and give the "right" map states to players. The only thing a wicked server could do is to mess up key strokes and send wrong informations, this is somewhat equivalent to make a bot play instead of a human players. Until someone writes an AI capable of playing U61 better than a human - given the fact that rules are changeable - this is not likely to be a real problem.

Additionnaly, U61 clients send and expect "checksums" of maps on a regular basis, and if these checksums are wrong are missing, it dumps error messages in the log. This is usefull to track down bugs, and also allows you to figure if someone's trying to cheat. Again, it's still possible for a wicked server to send the right checksum but maintain a wrong map state, but what good it be good for, since this server would virtually be playing another game.

Finally, you could technically write a U61 client and/or server that would display "Player A has won with 1000000 points", the problem being that all other connected clients would see the real pathetic score of 3 points 8-)

12.3.5 Possible improvements

One major improvement I'm thinking about would be to completely remove the client/server aspect, and make all clients potential servers. This way node A could start a game, node B could join it, then C and D could connect on B, A could leave the game and E would still be able to connect on C.

This could lead to endless - and hopefully fun - games where players would connect / disconnect as they wish, with no need for a permanent server.

Chapter 13

Known bugs

13.1 Major bugs

13.1.1 Game slows down and freezes

This is due to some imperfections in event handling. What happens is that the game core spends more time displaying things than processing events. Therefore there's a growing time gap between what's displayed and what's being calculated internally, at an event level. And the situation gets worse and worse as the display gets slower since the engine has to anticipate many frames. So the game chokes to death... I'm trying to find a compromise between real-time constraints and performance issues.

Still, keep in mind that with "recent" computers, running at more than 1GHz, there's almost no chance that you ever hear about this bug. I used to run (and develop BTW) U61 on a K6-200, and I had no problems with it. However, you need to know that it "can" happen, especially if your machine is old and/or its CPU is heavily used by some other programs.

13.2 Minor bugs

13.2.1 Win32 event dispatcher received WM_QUIT

You might get this message if you run U61 in windowed mode under Windows and close the application by clicking the standard "cross" which is in the top-right corner of the window.

13.2.2 Unable to type dots and other weird characters

Due to some limitations in key handling in ClanLib, you might get trouble trying to type characters such as "." or "-". This is very annoying since you need them to type ip addresses and/or machine names. Still, there's a workaround for ".", which is usually the most annoying character: simply press the space bar on your keyboard and it will enter "." instead of " " if you are editing a server name.

13.2.3 The numpad keys won't work

Again, this is due to the fact that ClanLib is still under heavy developpement. I'll try and do my best so that this bug is fixed some day, since it is very important for U61 that all the keys work correctly.

Chapter 14

Frequently Asked Questions

14.1 Why is the game called U61?

In fact, when I first started this project, I had called it "Tetr61s". This was because the game was at first supposed to be quite close to Tetris(c) and there's a letter in Russian which is pronounced "i" (like in "tetris") and when you write it down it looks like you have written 61.

Since the Tetris Company (http://www.tetris.com) claims to have a look-and-feel copyright on Tetris(c), and sues independant game programmers who program Tetris(c) like games, I could not call the game Tetr61s, nor make a plain Tetris(c) clone. So I decided to start a new project, more general, and change the name. So "U" is because my nickname is "U-Foot" and "61" is because it is the successor of "Tetr61s".

14.2 Can I copy the game?

Yes, since the game is Free Software (free as in speech), protected by the GPL (GNU General Public License). You should have received a copy of the GPL with the game. For more informations on the GPL, check http://www.gnu.org.

14.3 Why did you use ClanLib instead of SDL?

SDL starts being a quite mature library, it's widespread, present in most GNU/Linux distributions, and used by professionnals such as Loki. Therefore, one might wonder why I chose ClanLib instead, knowing that it's much harder to install and also less common library than SDL.

I chose ClanLib because is a high-level programming library, and provides much more than SDL. Its main drawback is that it's still under developpement and therefore API changes occur a bit too often for me. But its big advantage is that it provides tons of usefull functions, and has a very clean design. I also like the idea that ClanLib is developped by independant programmers, just like me.

14.4 Why did you code yet another Tetris(c) clone?

There are 2 answers to this question:

- If you really think U61 is a Tetris(c) clone, then you probably have not really played U61. Try and play with different set of rules using the "Game options" -> "Rules" menu, and play with friends - 3 or 4 players starts being fun... And don't miss the "special square" which is represented by a black and white "?" in the default theme.
- Of course I could have decided to tweak an existing game such as Tetrinet or Quadra instead of starting a new developpement from scratch, but I believe the final result would have been less flexible, and... ...I really enjoy programming 8-)

14.5 Source package is hudge, why?

It's true that the source package for U61 is very big, even once it's been compressed. It's even bigger than the binaries...

The reason is relatively simple: I think - and this is also the FSF point of view - that the source package must reflect my developpement environment, and offer users who download it all the code, tools and ressources I use to compile the game. Therefore the source package is simply my whole developpement tree, so it contains *everything* and it's big.

You could argue that I could separate the C++ source code from graphics and such things but it would force me to have several source packages, and then I'd need to have some scripts to interract between these packages and it would make things more complicated. I did try it - previous versions of U61 used to have an u61 and an u61-data package - and it sucks.

You'll also notice that there are some unused extra graphics, sounds or scripts in the source tree. These are ressources I'm "likely to use" in further developpements, are which have been done and abandonned in the past, but are not bad enough to be thrown away. Again, I put *all* my U61 stuff in the source package.

Another good reason for doing that - besides making all my work freely available - is that everything I put in the source package is stored on a distant CVS server, this means that if my house burns and I loose all material things my work is not completely lost 8-)

Chapter 15

Copying

U61 is another block based game.