STPPC2x Manual

STPPC2x is a collection of puzzle games for the GP2X handheld games console. It is a port and extension of the puzzles contained in Simon Tatham's Portable Puzzle Collection[1] that is customised for the GP2X and includes new games not in the usual collection, for example Slide and Sokoban from the "unfinished" section of Simon Tatham's Portable Puzzle Collection, Maze 3D from Edward Macnaghten[2] and some versions of my own[3] creation from puzzles I've seen, such as Mosco.

Terminology

STPPC2x refers to the GP2X port of Simon Tatham's Portable Puzzle Collection only.

The term "collection" refers to the entirety of all of the games (i.e. STPPC2x as a whole).

The term "game" refers to a particular type of puzzle (e.g. Blackbox, Bridges, etc.)

The term "puzzle" refers to a particular instance of a game (e.g. a particular Bridges game).

Boxes like this indicate technical information that you don't need to know.

Downloading and Installing

The main website for the collection is http://www.ledow.org.uk/gp2x/. However, it is also posted on the GP2X file archive (http://archive.gp2x.de). Simply download the .zip file for the latest version. You may or may not want to download the Music Pack too, which adds some nice background music to the collection. The source code should also be available at the main website.

To install the collection, simply unzip the downloaded file and place the contents onto your GP2X's SD card, either using an SD card reader or the GP2X link cable. I suggest that you install it into an STPPC2x folder, so that it doesn't interfere with your other software.

You should have the following approximate directory structure:

```
/stppc2x
        /<License files>
        /<README's>
        /credits.txt
        /help/<help text files>
        /images/<preview and miscellaenous PNG files>
        /music/<MP3 files, may be empty on a new installation>
        /screenshots/<screenshot files, may be empty>
        /stppc2x.gpe
```

---

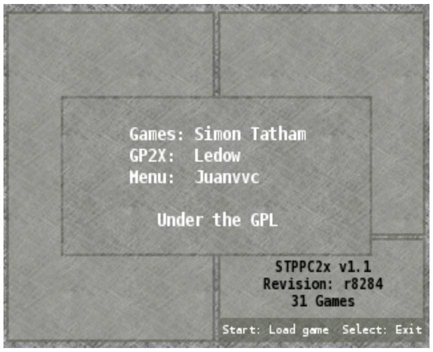
[1] http://www.chiark.greenend.org.uk/~sgtatham/puzzles/

[2] http://eddy.edlsystems.com/maze3d/

[3] Ledow, the main author of STPPC2x, http://www.ledow.org.uk/gp2x/

Starting the collection

To start the collection, turn on your GP2X, go to Games, SD Card and find the stppc2x folder that you copied there. Inside, you should see an entry for “stppc2x”. Start this with the X button.



Games: Simon Tatham
GP2X: Ledow
Menu: Juanvvc

Under the GPL

STPPC2x v1.1
Revision: r8284
31 Games

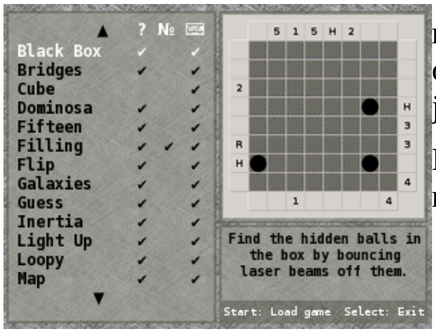
Illustration 1: The startup screen

You should see the startup screen (Illustration 3: An example of a Netslide puzzle). This contains the version number of STPPC2x and the version of the original collection that it is most closely based upon. Because many updates in Simon Tatham's collection do not affect the GP2X collection, this won't always be the latest, but it will give you a hint as to whether a new feature or game should be in STPPC2x or not.

Underneath that, it tells you how many games are in this version. Some of these games may be from different authors or otherwise not in the main puzzle collection but have been included because they meet the criteria of “Playable, fun, no major bugs”.

Press the Start button to move onto the main menu, or Select to exit the collection entirely.

Starting a Game



▲	?	№	⌨
Black Box	✓		✓
Bridges	✓		✓
Cube			✓
Dominosa	✓		✓
Fifteen	✓		✓
Filling	✓	✓	✓
Flip	✓		✓
Galaxies	✓		✓
Guess	✓		✓
Inertia	✓		✓
Light Up	✓		✓
Loopy	✓		✓
Map	✓		✓
▼			

Find the hidden balls in the box by bouncing laser beams off them.

Illustration 2: The main menu screen

You should now be at the main menu screen (Illustration 2: The main menu screen). This has a list of the games available within the collection and a lot of information about them. You can use the joystick/joypad to move up and down the list.

Each line tells you about one game. The columns next to the game names indicate:

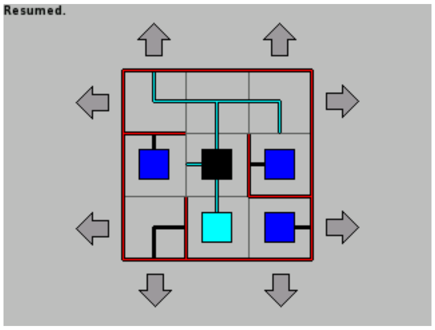
- Whether it has a built-in solver (underneath the ? column)
- Whether it requires you to input numbers (underneath the No. column)
- Whether the game can be controlled with the “cursor keys” mode of the collection (underneath the keyboard icon ⌨)

Additionally, when you highlight a game using Up and Down, the right-hand side will show you a short preview picture of the game and a brief description of it.

Use up and down to select the game you wish to play and press Start to load it. If you want to quit the entire collection, press Select.

Playing a game

Once you have selected a game and pressed Start, a new puzzle will start straight away. If the puzzle takes a long time to load, it may be because you have chosen a very difficult setting for that puzzle or merely that it takes a long time to start that particular type of puzzle (Slide currently takes longer than any other game to start but if you set the puzzle options too high, any puzzle can take a long time).



Resumed.

Illustration 3: An example of a Netslide puzzle

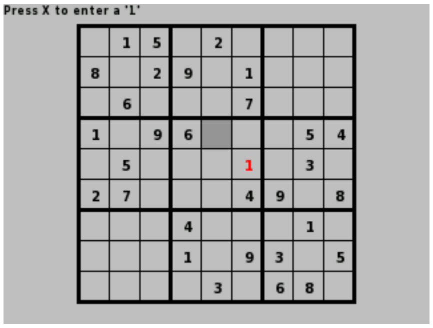
The individual games are all very different and there is a brief rundown on them all below. Generally, though, there are a lot of common things across all puzzles.

At the top, is the status line. This is a small line of text that informs you of certain things. This may be that you've paused or resumed the game, that you've just loaded a game, that you've completed a particular puzzle, that you've made a mistake or that a certain time has elapsed.

The majority of the screen is devoted to the puzzle area, the grey part. This is where the puzzle is drawn and where you can interact with it to try to solve it.

To interact with the puzzle, you use the GP2X's joystick/joypad and the A, B, Y, X keys. Each type of game is different and so you can “click” in different areas and different buttons do different things. However, generally, the A button will do most of what you need to solve a puzzle. A, Y and B usually act as left-, middle- and right-mouse clicks would act if you were playing with a mouse. Some games (e.g. Untangle, Map, Guess, etc.) require you to “drag-and-drop” elements. This works as you would expect... Move the cursor to the item to be dragged, press and hold the A button (to left-drag) and then move the cursor to where the item is to be dropped and release the A button.

Some games require you to enter numbers to solve them (those with a tick-mark under the “No.” column on the main menu). On a PC, you would type these at the keyboard but the GP2X doesn't have a keyboard. Instead, there is a little control system to help you.



Press X to enter a '1'

	1	5		2				
8		2	9		1			
	6				7			
1		9	6				5	4
	5				1		3	
2	7				4	9		8
			4				1	
			1		9	3		5
				3		6	8	

Illustration 4: An example game of Solo

The GP2X will remember a number for you if your game requires you to use numbers. Pressing the L button will decrease that number by one. Pressing the R button will increase that number by 1 The number's current value will be shown on the screen in the status line (Illustration 4: An example game of Solo).

When you are ready to “type” that number in, just press the X button and it will be “typed”. Some games require you to “click” (for example, with the A button) in a particular place first, so that they know where to put the number, and then press X to put the number there.

In this example, you can see the number 1 has been entered (incorrectly) into the grid. Depending on the type of game and the size of the particular puzzle, you may not be able to enter a particular number. For instance, if you are playing Unequal on a 4x4 grid, the largest number you can ever enter is 4 and the smallest 1. STPPC2x knows this and will “loop” around, so if you have 4 as the currently selected number and press R, it will jump back to 1. If you have 1 as the currently selected number and press L, it will jump forward to 4.

Similarly, for larger puzzles, STPPC2x automatically knows that you can use the digits up to 9, for instance, and for some very large puzzles, or for certain game types, it will let you use the digits 0 and A-Z in the same way (e.g. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, etc.).

There are also two control systems, but we will get to those later.

Many of the games feature built-in solvers. These allow you to let the computer try and solve the puzzle for you. This is helpful to learn the game, to help you find those elusive solutions and to see if the puzzle ***can*** be solved at all. To invoke the built-in solver for games that support it (those with a tick-mark underneath the “?” column on the main menu), press the Vol- and Vol+ buttons simultaneously.

Undo / Redo

Often, you make one silly mistake and just want to take it back. There is a facility for this also. You can undo every move made in a puzzle, one at a time, by holding the L key and pressing Vol-. Additionally, you can “redo” an undone move by holding the L key and pressing Vol+. Note, however, that some puzzles will let you undo/redo moves but they won't forget if you “died” in between. For example, if you click on a mine in the Mines game (which means that you have lost), it will remember that, even if you Undo it and click on another square! Even if you Undo and save the game, and then reload, Mines will remember that you died!

You can also pause and quit a game. To quit a game and return to the main menu, hold down L and press R.

Pausing the game

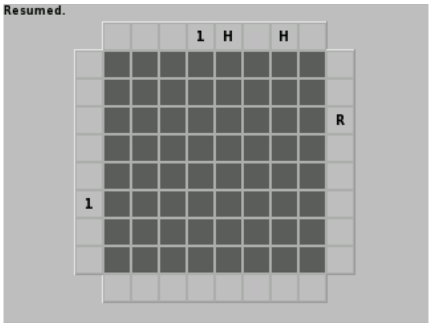
To pause a game at any time, press the Start button. This will take you into the in-game menu (Illustration 12: The main in-game menu) and also pause the game. You will see the version of STPPC2x and the name of the currently selected game. To unpause the game, press Start again or select “Return to game” from the in-game menu.

The Games

Below there is a section for each of the games in the collection. Hopefully, you can use the information given to learn and play through each of the games on your own. This list is current as of STPPC2x version 1.1

Black Box

Objective - Find the hidden balls in the “Black Box”.



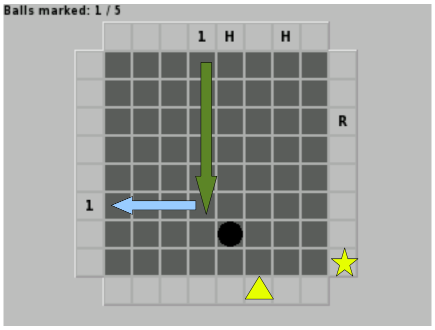
Resumed.

Illustration 5: Black Box

Elements – The “Black Box” is the central dark grey area. The squares outside that area are the “lasers”.

Method - You can only find out where the balls are by bouncing “laser beams” off them. To fire a laser, click on one of the light-grey squares. It will reveal a number or letter.

If it reveals a number, this is a “beam number” and there will be another of the same number somewhere else on the grid. This is where that particular “shot” left the black box. A laser beam travels in a straight line from the outer edge, however its path is modified by the balls hidden within the black box.



Balls marked: 1 / 5

Illustration 6: The first ball revealed

If a beam encounters a ball in front and to the left, it will then turn 90 degrees to the right and vice versa. For example, if we look at beam “1” in the illustration, we “fired” it from the top number “1” on the grid. From there it entered the black box heading straight down. However, it left on the left-hand edge, therefore it must have encountered a ball somewhere inside the black box. In this case, the ball is where we have placed it.

How do we know this? Because when the laser beam headed “down” (green), it would have had to turn “right” (actually left on the grid, because the beam was moving down) in order to leave where it did. So there must have been a ball in front and to the left (actually right on the grid).

Similarly, we would expect a beam fired from the point marked with a yellow star to travel left (because it has been fired into the grid from the right-hand edge), “bounce” left and leave by the point marked with a yellow triangle. This is, of course, assuming that no other balls exist inside the black box.

However, the status line on the top left informs us that we have marked only 1 of 5 balls that are actually in the black box!

What are the other symbols that we have on the grid? Well, we've “fired” three other lasers in our efforts to find all the balls. We've been shown an “H” (which means that the beam encountered a ball head-on and was absorbed) – absorbed balls never leave the grid and so there is no exit for it... the other H is ***another*** laser that we've fired that also hit a ball head on and was absorbed.

The R is another laser that was fired but whose beam ended up doing one of two things: Either it left by the same square which it entered (a reflection) or it was "deflected" by a ball before it even entered the grid (so this option would indicate that there was a ball ON that edge to one side of the R). Only further investigation will tell us which of these options are possible and so reveal every ball's location.

You can fire as many lasers as you want (you can even fire every laser possible) and not be penalised. However, the idea is to have as few lasers fired as possible when you work out the location of all the balls.

When you think you have found the locations of all the balls and marked them all with a black circle, a green button will appear. Clicking this button will test whether you are right and either show you the solution if you were (i.e. you win) or show you where there's a contradiction by highlighting it in red (i.e. go back and try again). You can only click this green button when you have exactly the right number of ball locations highlighted (5 in this example).

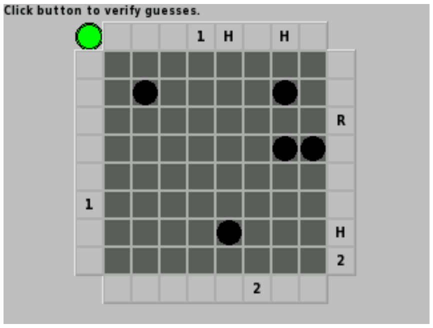

Click button to verify guesses.

Illustration 8: Are we right?

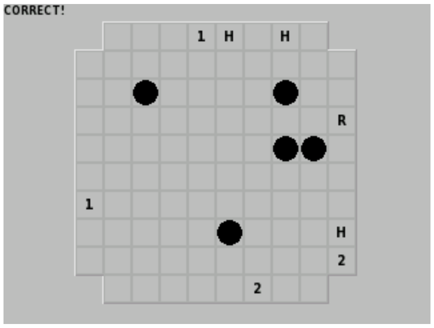

CORRECT!

Illustration 7: Yes we are!

Controls:

	Mouse Control System	**Cursor Key Control System**
Joystick/joypad	Move mouse cursor	Move cursor
A	Click on a square	Select a square
B	Mark square/row/column as empty	Mark square/row/column as empty

A-Click a light-grey square on the outer edges to fire a laser.

A-Click a dark-grey inner square to mark a possible ball.

A-Click the green circle to check your guess.

B-Click to mark squares, rows or columns empty.

Bridges

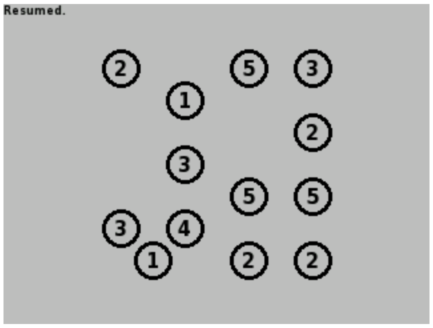


Illustration 9: Bridges

Objective – Join each of the islands with the correct number of bridges. Each island should end up with no more and no less than the number of bridges written inside it, all the islands should be connected (so you could get from any island to any other island just by “walking over” the bridges).

Elements – The circles are the islands. The lines that you draw between them are the bridges. The number inside the circle tells you how many bridges that island has connected to it in total.

Method – Start with the easy islands. By default, an island can only have a maximum of 2 bridges in each direction – up, down, left, right (never diagonally). Thus, if we find an island marked “2” which is only in the same row or column as one other island, then we know that both of those bridges are to that island. Similarly, if we have an island marked “4” and it could only connect to two other islands, we know that they both have two bridges (e.g. the “4” in Illustration 9: Bridges can only connect to the “3” above it and the “3” to the left of it, so there must be two bridges to each island from “4”).

When you have eliminated the easy islands, start working on a logical basis. For instance, with the “3” on the left of the screen in Illustration 9: Bridges we ***know*** it has two bridges connecting to the right (we just saw that because of the “4”). That leaves one bridge and only one place for it to go... up to the “2” island.

By a process of elimination, you will be able to find all the bridges.

Controls:

	Mouse Control System	**Cursor Key Control System**
Joystick/joypad	Move mouse cursor	Move cursor
A	Lock / Unlock an island, draw a bridge	Lock / Unlock an island, draw a bridge
B	Lock / Unlock an island, draw a bridge	Lock / Unlock an island, draw a bridge

A-Drag between two islands to add a bridge between them.

Keep A-Dragging to add more bridges or remove all bridges in that direction.

B-Drag from an island to place a marker when you know that there can't be a bridge in a certain direction.

Cube

Objective – Move the cube over the painted surface to “pick up” all the painted squares onto the cube.

Elements – The cube (or other shape) can move over the surface. The blue squares are the painted squares. The grey squares are unpainted squares.

Method – When you move the cube onto a square, it will either pick up the blue paint (so the bottom face of the cube becomes blue and the square on the floor because grey), put down blue paint (so the square on the floor becomes grey and the face of the cube becomes blue), or neither (for instance, if a painted face touched a painted square on the floor, both stay painted).

You have to move the cube about so that it becomes completely painted in blue and no blue squares remain on the floor. Generally the best tactic is to try to form a "chain" of painted squares that you can come to later on and, in one set of movements, pick up every one of painted squares.

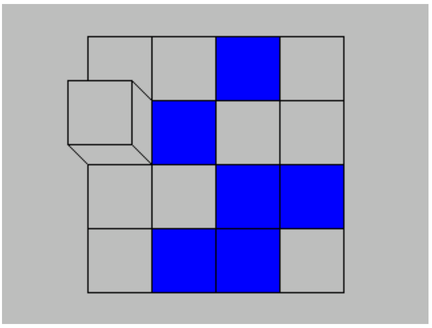


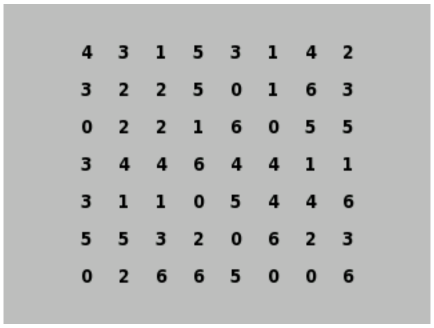
Illustration 10: Cube

Controls:

	Mouse Control System	**Cursor Key Control System**
Joystick/joypad	Move mouse cursor	Move in the direction pressed
A	Move in the direction of the cursor	N/A

Dominosa

Objective – Mark the positions of the complete set of dominoes which could be place horizontally or vertically anywhere in the grid.



4	3	1	5	3	1	4	2
3	2	2	5	0	1	6	3
0	2	2	1	6	0	5	5
3	4	4	6	4	4	1	1
3	1	1	0	5	4	4	6
5	5	3	2	0	6	2	3
0	2	6	6	5	0	0	6

Illustration 11: Dominosa

Method – The initial grid is just a grid of numbers. However, it has been made by fitting together a complete set of dominoes 0-0 to 6-6, it's just that we can't see if, say, the top right corner of Illustration 11: Dominosa is a 4-6 and a 2-3, or a 4-2 and a 6-3 or even some more complicated arrangement.

The easiest (but most tedious) way to solve this game is to hunt through the grid for each domino in turn. First, look for everywhere where a 0-0 could be. In this example, there's only one possible position, at the right-hand-side of the bottom row. Because there's only one possible place, we can mark it for certain – this ***must*** be where the 0-0 is hiding.

This will leave us with a "6" on its own on the right hand side. This must also be part of a domino but it can't be a 0-6 because we know the 0 is already used. Thus, it must be a 3-6 placed vertically. Knowing that, we can mark it as definite.

When we get stuck trying to make more logical deductions from our first move, we can then move on to look for a 0-1, then a 0-2, then a 0-3, 0-4, 0-5, 0-6. If we find multiple places that ***could*** be the domino in question (e.g. there are spaces that could be a 0-2 in the bottom-left, bottom-right, upper-left and bottom-middle!), then we just move on because we can't know for sure which one is actually that domino.

When using this method, take note that a set of dominoes consists of only 28 dominoes. A 1-0 and a 0-1 are the same thing and only one of them exists in the grid. So we should only look for:

0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 0-6,

1-1, 1-2, 1-3, 1-4, 1-5, 1-6,

2-2, 2-3, 2-4, 2-5, 2-6,

3-3, 3-4, 3-5, 3-6,

4-4, 4-5, -4-6,

5-5, 5-6,

6-6

What if we get all the way to 6-6 and the grid isn't complete? Start again at 0-0 and carry on, making sure not to mark the same domino twice (there can only be one of each domino on the grid!). By the time you got to 6-6, you would probably have made at least one change to grid that will let you advance further the next time a certain number comes around. As we mark each domino that we're sure of, the job becomes easier.

If we get stuck or don't wish to search the entire grid each time, there's a shortcut we can use. When we marked the 0-0, we automatically eliminated other possibilities for the dominoes near it. So we ***know*** that the left-hand 0 can't be part of a 0-5 or 0-6 and that the right-hand 0 can't be part of a 0-2 or 0-6 there. We can use this to shortcut a little because with that single move we eliminated 2 possible positions for 0-6, one for 0-5 and one for 0-2... so it might be an idea to try looking for them next as they will have less places to hide now!

Controls:

	Mouse Control System	**Cursor Key Control System**
Joystick/joypad	Move mouse cursor	Move in the direction pressed, and highlight a pair of numbers
A	Mark the position of a domino	Mark the position of a domino
B	Add a marker when you ***know*** that a domino can't lie in that direction	Add a marker when you ***know*** that a domino can't lie in that direction

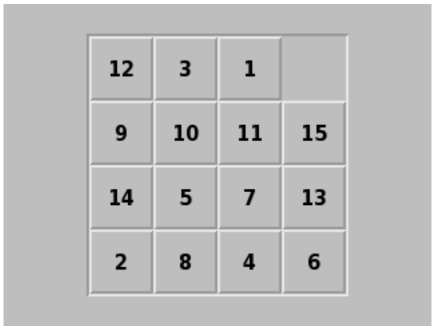
A-click between two numbers to mark them as a domino.

A-click them between again to remove the mark.

B-click between two numbers to mark when you know there ***can't*** be a domino lying that way.

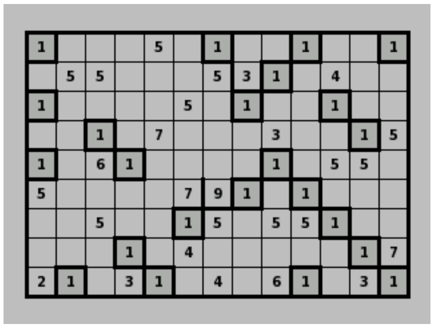
B-click between them again to remove that mark.

- Fifteen



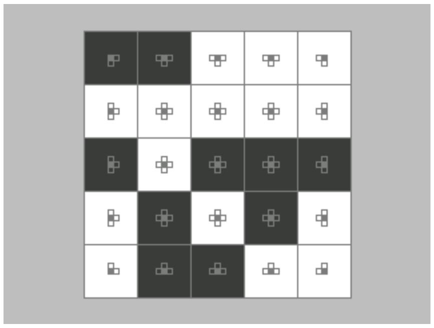
12	3	1	
9	10	11	15
14	5	7	13
2	8	4	6

-
- Filling

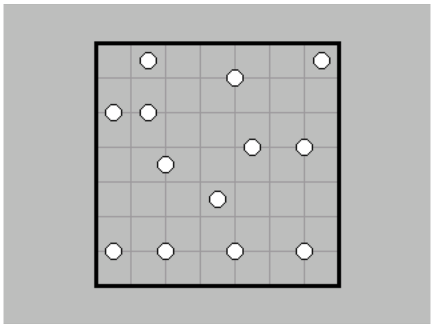


1				5		1			1			1
	5	5				5	3	1		4		
1					5		1			1		
		1		7				3			1	5
1		6	1					1		5	5	
5					7	9	1		1			
		5			1	5		5	5	1		
			1		4						1	7
2	1		3	1		4		6	1		3	1

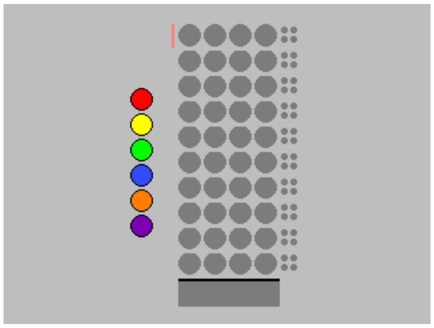
-
- Flip



-
- Galaxies

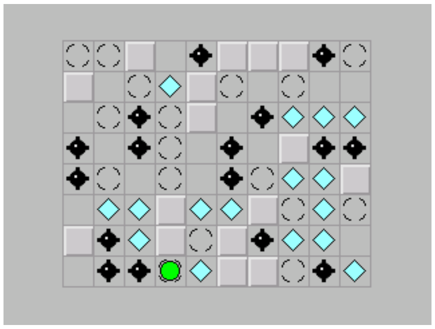


-
- Guess

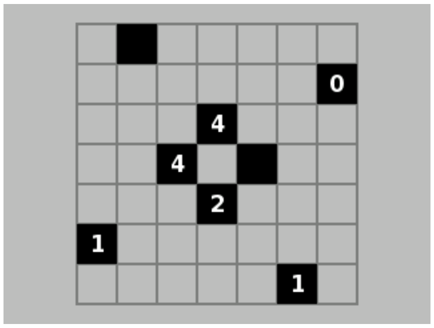


-

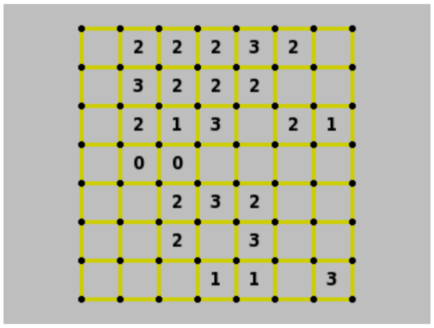
- Inertia



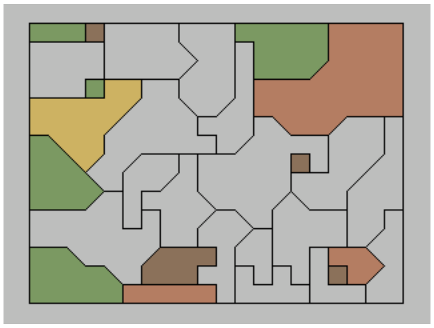
-
- Light Up



-
- Loopy

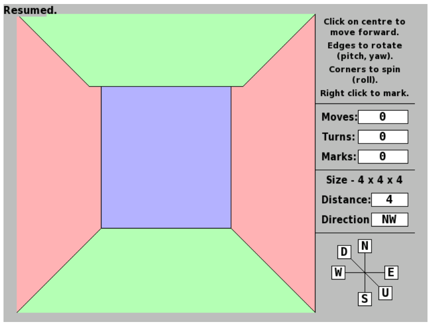


-
- Map



-
- Maze 3D

-



Resumed.

Click on centre to move forward. Edges to rotate (pitch, yaw). Corners to spin (roll). Right click to mark.

Moves: 0

Turns: 0

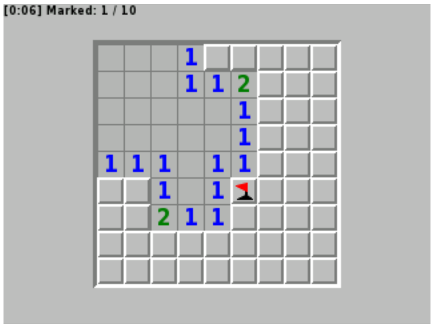
Marks: 0

Size - 4 x 4 x 4

Distance: 4

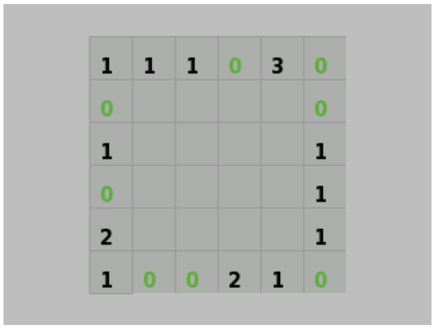
Direction NW

- Mines



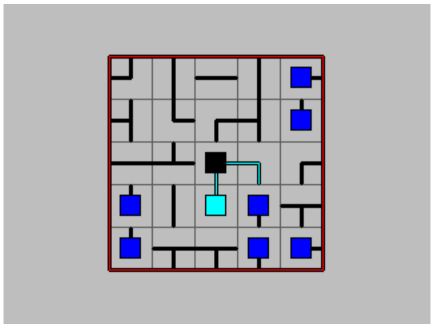
[0:06] Marked: 1 / 10

-
- Mosco

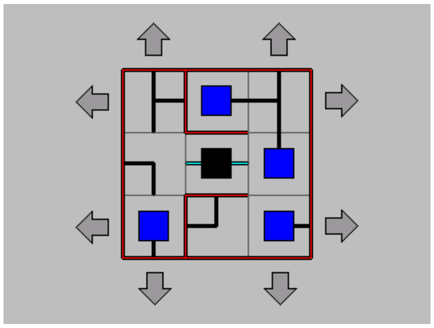


1	1	1	0	3	0
0					0
1					1
0					1
2					1
1	0	0	2	1	0

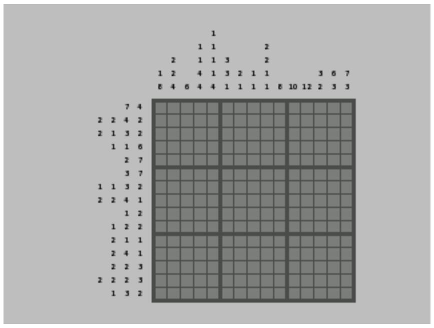
-
- Net



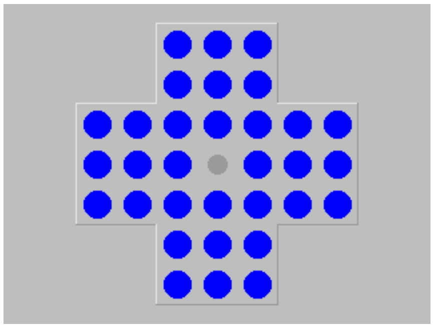
-
- Netslide



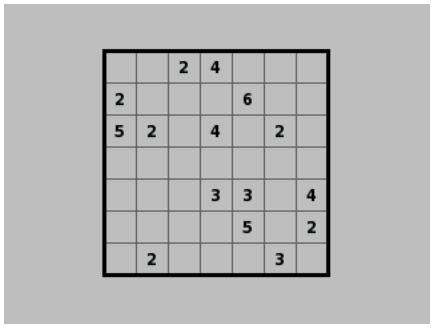
-
- Pattern



-
- Pegs

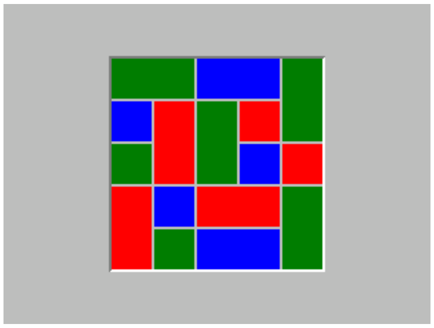


-
- Rectangles

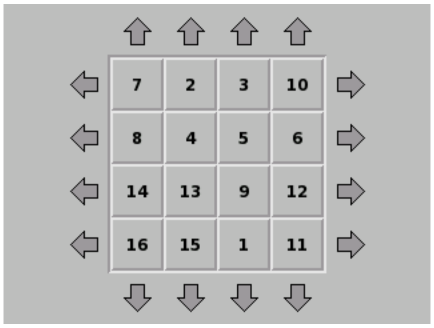


		2	4			
2				6		
5	2		4		2	
			3	3		4
				5		2
	2				3	

-
- Same Game

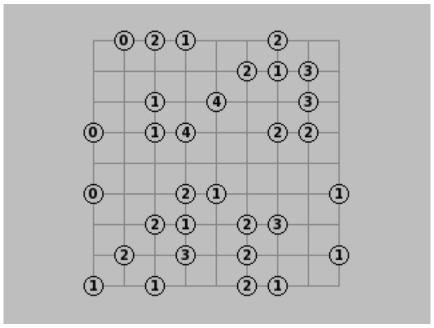


-
- Sixteen



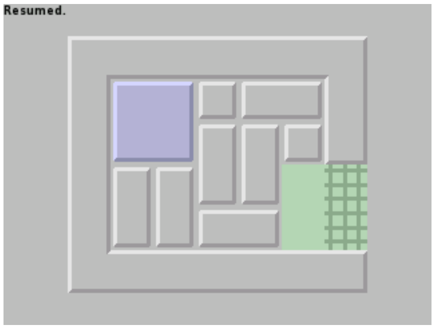
7	2	3	10
8	4	5	6
14	13	9	12
16	15	1	11

-
- Slant



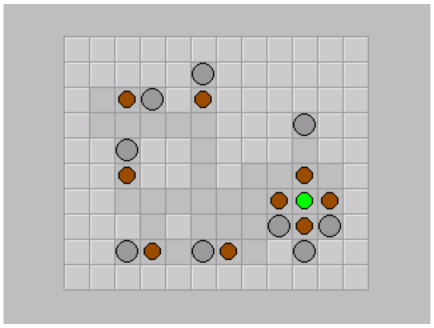
	0	2	1			2		
					2	1	3	
		1		4			3	
0		1	4			2	2	
0			2	1				1
		2	1		2	3		
	2		3		2			1
1		1			2	1		

-
- Slide

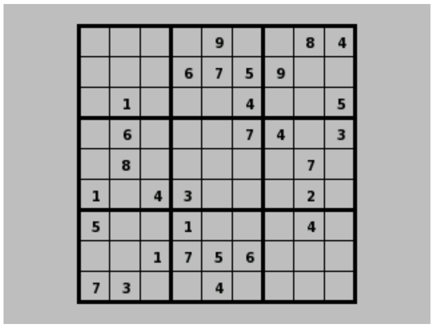


Resumed.

-
- Sokoban

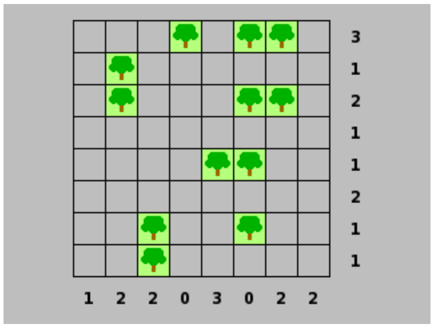


-
- Solo



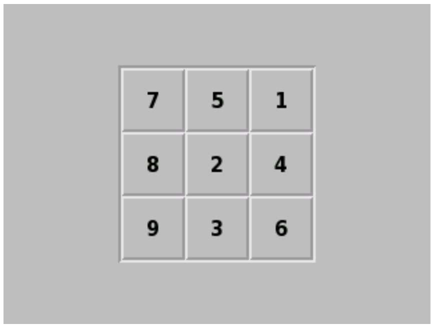
				9			8	4
			6	7	5	9		
	1				4			5
	6				7	4		3
	8						7	
1		4	3				2	
5			1				4	
		1	7	5	6			
7	3			4				

-
- Tents



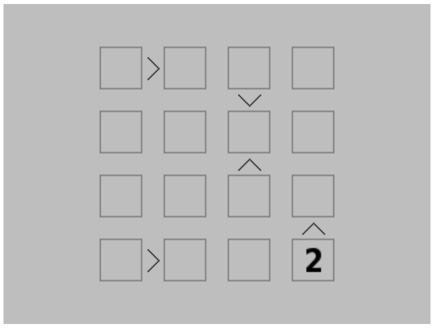
			🌳		🌳	🌳		3
	🌳							1
	🌳				🌳	🌳		2
								1
				🌳	🌳			1
								2
		🌳			🌳			1
		🌳						1
1	2	2	0	3	0	2	2	

-
- Twiddle



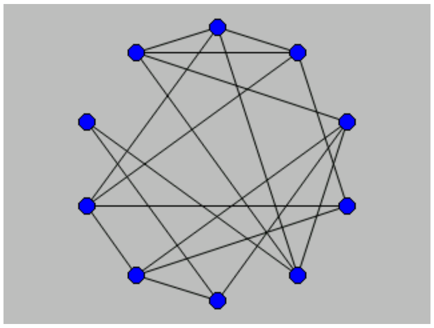
7	5	1
8	2	4
9	3	6

-
- Unequal



-

- Untangle



-

Navigating the in-game menu

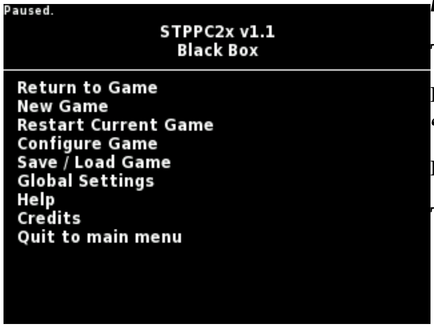


Illustration 12: The main in-game menu

To select options within the in-game menu, use the joystick/joypad to move the cursor onto an option and then press A to “left-click” or B to “right-click”. For most items, both buttons do the same but for some menus, they will increase or decrease a number respectively.

The options on the main menu are:

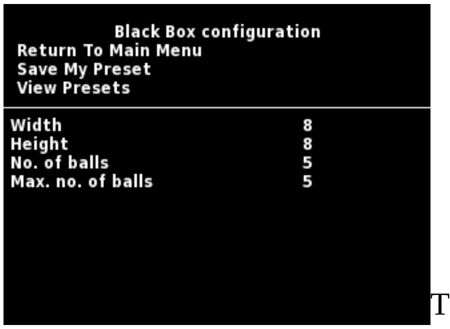
- Return to Game – unpauses the game and goes back to the game screen.
- New Game – starts a new puzzle of the current game. Your current game will be lost.
- Restart Current Game – clears the game and restarts the exact same puzzle you were playing. Useful if you realise that you've made a mistake and want to start on the same puzzle again. Your current progress will be lost.
- Configure Game – lets you change options for this particular puzzle or game.
- Save / Load Game – allows you to save your progress on this game, or load in a previous save game.
- Global Settings – change settings that affect games of all types.
- Help – get help on how to play the current game, or which buttons do what.
- Credits – see information on people who contributed to the collection in some way.
- Quit to Main Menu – go back to the main menu, so you can select another game. Your current game will be lost.

Configuring a game

A game may have several ways to make it more or less difficult, or a way to turn it into an entirely different type of game entirely. Some games have many configuration options and some have almost none. The game configuration menu will show you all of the options available for the current game.

On the configuration screen, the menu options above the line allow you to do certain things:

- Return to main menu – takes you back to the in-game menu, any settings you have changed on the configuration screen will take effect on the next “New Game”.
- Save My Preset – saves the current options as a “preset”, so that you can save your favourite type of puzzle instead of having to set the options each time. The “My Preset” is loaded every time you start a game from the main menu.
- View Presets – this lets you look at any presets that the game itself has. Most games have several presets of varying difficulty to give you a range of challenges.



Width	8
Height	8
No. of balls	5
Max. no. of balls	5

Illustration 13: A configuration screen for Black Box

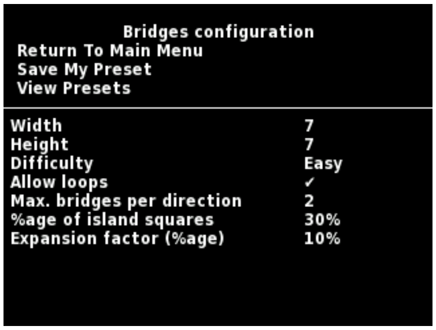
The menu options below the line are the actual configuration that will be used when a New Game is next started. If you haven't changed anything, they are the same options as the puzzle that you are currently playing.

In this example (Illustration 13: A configuration screen for Black Box), we see the configuration screen for the game Black Box. This particular game lets you adjust the Width and Height of the puzzle as well as the minimum and maximum number of balls inside the “black box”.

You can adjust these options using the cursor and the A and B buttons to suit your preference. However, be careful of setting some options too high as some games are capable of bringing even a top-of-the-range computer to a halt for many hours, and the GP2X is only a slow computer in comparison to a desktop PC!

The game's presets will give you an idea of how high you can go on some options. See the Presets section below.

As well as numerical options, some games have options that let you select from a list of types or turn certain options on and off. This example (Illustration 14: A configuration screen for Bridges) shows such options for “Difficulty” (which, when clicked on, cycles through Easy, Medium and Hard settings) and “Allow loops” which shows a tickmark when enabled and a cross when disabled.



Bridges configuration
Return To Main Menu
Save My Preset
View Presets

Width	7
Height	7
Difficulty	Easy
Allow loops	✓
Max. bridges per direction	2
%age of island squares	30%
Expansion factor (%age)	10%

Illustration 14: A configuration screen for Bridges

Note that whenever you change an option, it is remembered for the current session but may not take effect until you start a “New Game” from the in-game menu. However, unless you save your options as a preset, they will not be remembered once you have left the collection.

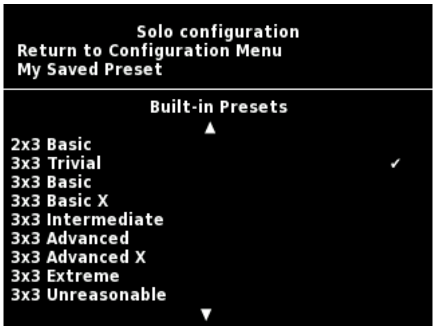
Using configuration Presets

Presets are sets of configuration options for a particular game that are “saved” onto the SD card on the GP2X. They are useful for storing your favourite settings and many games come with some of their own presets (so they act as “difficulty levels”).

A preset can be saved at any time by going into the configuration screen for a game, changing the options to the desired values and then pressing the “Save My Preset” option. Whenever you start a new game, your preset is used to get the settings for the new game, so be careful not to save a preset which takes a long time to generate!

You can access a game's built-in presets and load your preset manually using the game Presets menu, by selecting View Presets.

This screen (Illustration 15: The Presets menu for Solo) lists all of that game's built-in presets below a line. If there are many presets, arrows will appear that will allow you to scroll up and down the list of presets by clicking on them.



Solo configuration
Return to Configuration Menu
My Saved Preset

Built-in Presets
▲
2x3 Basic
3x3 Trivial ✓
3x3 Basic
3x3 Basic X
3x3 Intermediate
3x3 Advanced
3x3 Advanced X
3x3 Extreme
3x3 Unreasonable
▼

Illustration 15: The Presets menu for Solo

To load a preset, click on the preset name (e.g. 7x7 easy) and a new puzzle will be generated with the options contained in that preset. Be warned that your current game will be lost.

If a preset has a tick-mark beside it, it means that the current options are the same as that preset (so either you are already playing a game with that preset, or you managed to duplicate that preset with your current settings).

Selecting the option “My Saved Preset” will load any preset that you may have saved for the current game.

Presets are saved into the main STPPC2x folder and have the filename “<gamename>.ini”, e.g. solo.ini for Solo. They should be easily readable as they just follow the standard .INI file format but please don't try to edit them outside of STPPC2x.

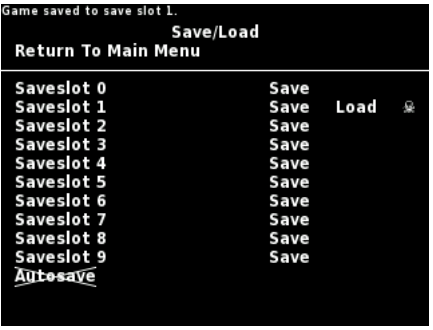
If a Preset file corrupts, it may stop you loading that particular game. If this happens, you may need to remove the preset file from the SD card. As an emergency measure, holding down L or R when pressing Start to load a game from the main menu will make STPPC2x ignore any saved Presets or Autosaves for that game. You can then save a new “My Preset” from within STPPC2x's Presets menu.

Saving your progress

You can save your current puzzle at any time by selecting Save / Load game from the in-game menu. This will show you the Save/Load menu (Illustration 16: The Save / Load menu).

To save the current puzzle, click on the word “Save” next to a saveslot. There are ten saveslots (0-9) available for each game. Please note that clicking Save will overwrite anything already in that saveslot.

To load that game again in the future, click on the word Load. Your current puzzle will be lost and the puzzle contained in the saveslot will be loaded.



Game saved to save slot 1.

Illustration 16: The Save / Load menu

To delete a savegame, click on the skull & crossbones icon (☠) next to the saveslot. STPPC2x will ask you to click the icon again if you really wish to delete the saveslot. If you are sure that you want to delete the game, click the skull icon again, otherwise click anywhere else.

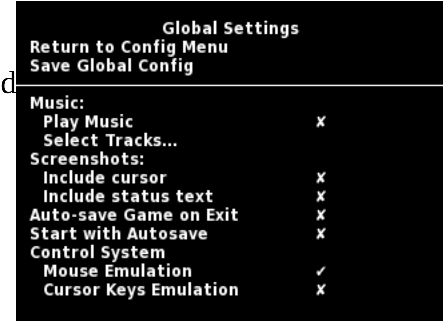
At the bottom of the list of saveslots, there is a special saveslot Autosave. If this is crossed (as in the above illustration), then Autosave is currently disabled. Autosave is a special option to save your puzzle every time you leave the colleciton or change games. If Autosave is enabled, this special slot will be used to save the game into each time. If it exists, you can load the autosave game using the Load button which appears next to it and delete it with the skull and crossbones icon, whether or not Autosave is currently enabled. However, only the Autosave facility can save to this slot.

Games are saved to the STPPC2x folder on the SD card in the same format as all Simon Tatham's Portable Puzzle Collection ports – a plaintext file which can be transferred to any system and played. In STPPC2x, the saveslot files are named “<gamename>[0-9].sav”, e.g. solo0.sav or blackbox7.sav. You can bring in STPPC savegames from other systems if you wish, so long as the game is available on STPPC2x and you put them in the right folder with the right filename. Autosaves are named “<gamename.autosave>”, e.g. dominosa.autosave, however it is recommended that you don't try to edit or overwrite Autosaves manually.

If an Autosave corrupts, it may stop you loading that particular game. If this happens, you may need to remove the autosave file from the SD card. As an emergency measure, holding down L or R when pressing Start to load a game from the main menu will make STPPC2x ignore any saved Presets or Autosaves for that game. You can then delete the Autosave game from within STPPC2x's Save / Load menu.

Global Settings

Global settings affect every game in the collection. There are several settings in this menu:



Global Settings
Return to Config Menu
Save Global Config

Music:
Play Music ✗
Select Tracks...
Screenshots:
Include cursor ✗
Include status text ✗
Auto-save Game on Exit ✗
Start with Autosave ✗
Control System
Mouse Emulation ✓
Cursor Keys Emulation ✗

Illustration 17: Global Settings Menu

- Play Music – If ticked, then music will play in the background while you play the puzzles.
- Select Tracks... - Takes you to the Track Selection menu which allows you to select which music tracks will play.
- Screenshots include cursor – If enabled, screenshots will include the cursor that you see on the screen.
- Screenshots include status text - If enabled, screenshots will include any text in the status bar.
- Auto-save Game on Exit – If enabled, every time you exit a game and return to the main menu, the game will be saved. This allows you to carry on where you left off each time without having to manually save the game. This is useful if you often need to switch off your GP2X in a hurry and don't want to lose your progress but see the warning in Appendix A: Writing to the SD Card.
- Start with Autosave – If enabled, every time you start a game, STPPC2x will look for an Autosave file for that game and, if it finds it, will automatically load it. If this option is disabled, the Autosave game will not be loaded automatically when you start a game but will still be available from the Save/Load menu.
- Mouse Emulation Control System – If enabled, the GP2X's joystick and buttons will act as though they were a mouse. The joystick will move the mouse cursor and the A, Y and B buttons will act as left-, middle- and right-click buttons. This is the default and all games can be played like this.
- Cursor Keys Emulation Control System – If enabled, the GP2x's joystick and buttons will act as though they are the cursor keys on a keyboard. The joystick's movements will correspond to Up, Down, Left and Right and the A and B buttons will act as a primary and secondary button. Typically, this translates to moving the cursor to the box or element above the currently selected one inside a game. This simplifies the controls in many games and doesn't need such accuracy as the mouse mode but not all games support it and not all games are easy to play with it.

Two control systems exist because the GP2X's controls do not always map nicely to those that the game expects to be played with. This is a problem with many ports of the collection because, for example, touchscreen devices, mice and keyboards can be used to play many of the games but not all of them. Some games are designed to be played with a keyboard, some can only be played with a mouse and some can be played with either system but one or the other is usually more comfortable.

As examples of the Control System, “Untangle” can only be played with a mouse (indicated by the lack of a tick in the ⌨ “keyboard” column on the main menu), Sokoban greatly benefits from Cursor Key control but can be played with a mouse, Net can also be played with either system but it's much easier to play with Cursor Key control if you are on a crowded train and Loopy, although both control systems are supported, is much harder to play with Cursor Key control if certain options are turned on (non-square grids).

You can tell what sort of control system you are using by looking in the top-right corner of the screen when the game is paused (Illustration 18: The keyboard icon showing in the top-right). If you see a little keyboard icon (⌨), then you are using the Cursor Key control system, otherwise you are using the Mouse control system.

Any changes to the Global Options take effect immediately, however, unless you select “Save Global Config”, they will reset to defaults the next time you load STPPC2x. Save Global Config saves the current settings as the default for all future games.

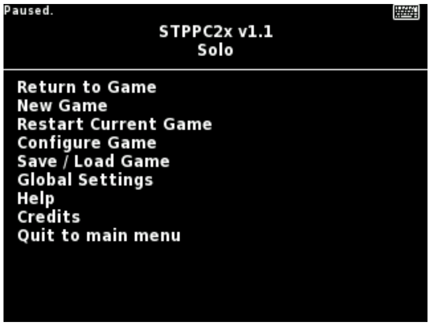


Illustration 18: The keyboard icon showing in the top-right

> The Global Config is saved in the main STPPC2x folder in a file named “stppc2x.ini”. It is not recommended that you edit this file manually. If this file becomes corrupt, STPPC2x may refuse to load. To remedy this, delete the stppc2x.ini file.

Help

The help menu can provide you with help so that you don't have to refer to this manual when you are actually playing the game. There are three options:

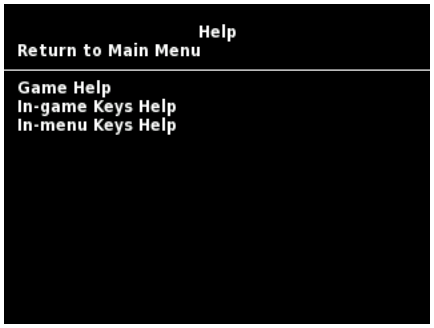


Illustration 19: The Help Menu

- Game Help provides a quick run-down of the rules of the game as well as any keys you might need to use. This isn't a full explanation of every rule or puzzle but should be enough to remind you how to play or to start learning.
- In-Game Keys Help is a list of all the buttons that you can press while in a game and what they do. Again, this is only a summary.
- In-menu Keys Help is a list of all the buttons that you can press while the game is paused and in the menus. This includes a couple of shortcuts for saving games, configurations and other similar keys.

- Added code to replace wrapper script - now any command line argument will stop the program respawning the menu at exit, otherwise it will always run /usr/bin/gp2xmenu.

Appendices

Appendix A: Writing to the SD Card

Some options in STPPC2x write data to the SD card. Although this is usually perfectly fine, if a program writes to the SD card a lot, it can "wear" it out over time. This usually only happens if an awful lot of data is written very frequently onto an old SD card that has been used for a long time. Also, the GP2X is a battery-powered device and batteries can die at awkward moments. If a file is being written while the battery dies, it could corrupt that file or the entire SD card.

The chances of either thing happening are slim - however, STPPC2x takes no chances and writes to your SD card as little as possible. It will ***only*** write to the card when you:

- Save a game
- Save My Preset
- Save the global settings
- Enable Auto-save

STPPC2x is also very careful to write only small amounts of data (usually around 1-2 Kb per save game) and to do it in one single operation and then close the file quickly. This should mean that even if the battery dies, as little damage as possible will be done and only to the actual file being written (i.e. it might corrupt a savegame or configuration file if it was in the middle of writing it, but everything else should work just fine).

Out of these operations, the Auto-save option is possibly the most potentially dangerous (hence, this option is not turned on by default) as every time you exit a game back to the main menu, it saves the entire game to the SD card. This way, whenever you leave a puzzle and then come back to it later (even if you have turned off the GP2X in the meantime) you should be exactly where you left that puzzle.

I don't believe that anybody will have a problem with this, but it only takes one report from someone who trashed their SD card by turning on all these options and playing thousands of games to get a reputation that the program "breaks SD cards". As the main developer, I have played thousands of games of every type in STPPC2x, including heavily-testing the save functionality in a variety of ways, and have played STPPC2x for longer and with more unfinished code than anybody else and all the time I've used the same cheap 2Gb SD card and not had any problems. But be warned – it's a possibility.

Appendix B: Development History

STPPC2x started because I really wanted a GP2X port of the puzzle collection. I initially tried to garner support for someone else to write the program on the GP32X forums (http://www.gp32x.com). My C was rusty to say the least and I knew someone else could do a much better job than I. I posted a request for someone to port it and this resulted in a few reponses but nobody seemed eager to take on the project.

In the end, I resigned myself to having to re-learn C, which I had used many years before and forgotten most of. My experience with programming harks back to the days when compiling and linking were entirely seperate steps done by entirely seperate programs and manually invoked by the programmer. “Makefiles” didn't exist and a lot of the time you fought with the compiler for hours because it just returned a vague error that you were responsible for tracking down. Also, using libraries was a bit of a nightmare because it required a perfect environment setup.

However, as an experiment, I read up on GP2X programming and downloaded a couple of development environments, most of them based on GNU GCC with which I'd had some horrible previous experiences. One of them, the Open2x toolchain, seemed to be exactly what I was after. In fact, an initial compile of the unmodified code almost worked out of the box. If Open2x hadn't existed in the state it was in, I never would have bothered with this port.

The problem was that the collection was designed to run under a certain “frontend”. Frontend's existed for GTK, Windows, Palm, and eventually Java but none of them would suit the GP2X's embedded-Linux setup. Most people were using SDL on the GP2X and I'd used SDL in the past. It seemed that writing an SDL frontend would be the thing to do – after all, it was a “portable” puzzle collection, so it should be quite easy to “port”, right? Bloody hell.

Porting to SDL involved writing an SDL frontend. Basically, the puzzles know nothing of the computer they run on and simply ask the frontend to “draw a circle”, “draw a rectangle”, “tell me the background colour”, “wake me up in ten milliseconds”, etc. The frontend is responsible for actually creating the window that these shapes draw on, drawing each of them when asked and all the other functions. Each of these functions needed to be rewritten to use SDL routines.

At first, I took the GTK code (because that compiled straight away and ran without problems on my Linux development machine – a 600MHz IBM Thinkpad) and alongside a lot of the GTK statements, I inserted SDL code that did similar things. This worked marvellously well and allowed me to build an executable which, when run on a Linux PC, created a GTK window and an SDL window at the same time and tried to draw the same objects on each.

There were a lot of problems. Firstly, I still had the GTK-dependency in the code for a long time. Also, there were ***lots*** of places that called GTK functions and I couldn't duplicate them all – I needed to know that this was going to work with some small tests first before I committed myself to it. And I discovered that I absolutely ***hate*** Makefiles. These are small “scripts” that are supposed to make compiling and linking several C source code files into objects and then turning them all into one lovely program. The problem is that Makefiles are so inherently clever that they can be incredibly complex and obscure to read. And when you want to not only plug GTK code and libraries into a collection of 30 games but also SDL code and libraries, things quickly get out of hand.

I solved this problem by reverting to my old-fashioned way of thinking. I made a script “make_net.sh” that built just the “net” game and nothing else. It contained two commands – one to compile the SDL frontend, and one to link it into an executable. Because I had run the unedited Makefile, I had all the files that it needed to “link” to already, including the net game itself and all the GTK objects and the SDL libraries were already installed on my system. This worked marvellously well and I was able to just run this script every time I needed to test my new SDL code without worrying about Makefiles. At one point, I had 28 of those files, one for each game!

Now that I had at least one game that I could test with, I started complementing GTK functions with SDL ones one-by-one. The standard SDL package can only draw rectangles unless you want to sit and write your own “circle” routines etc. Happily, though, I found SDL_gfx which is a library to do just want I wanted... fast, simple circles and other shapes with a single command. I also noticed that it was already included in Open2x, so the decision to use it was really quite easy and I started adding to the GTK functions with the ones in SDL_gfx.

This isn't as easy as it sounds because there was a lot of other GTK code and some of it wasn't relevant at all to the GP2X (e.g. clipboard and file saving) and the stuff that was relevant wasn't always as simple as just replacing the GTK "draw_circle" routine with SDL's "draw_circle". For instance, the games often want to draw a polygon... to do this, they give GTK a list of points and a colour and the GTK functions go off and draw and fill the polygon. SDL_gfx's equivalent didn't want a list of points in the same order, though. Instead of "x1, y1, x2, y2, x3, y3, ..." for a list of the x- and y-coordinates of the points, SDL_gfx wanted "x1, x2, x3, … , y1, y2, y3, … ". A simple difference to account for but there were quite a few things like this that cropped up.

The rectangle routines used unclear definitions... if I ask for a square that starts at coords 1,1 and is 3 wide, does that mean it ends at 4, 4, or 3,3, or what? What if it includes a border (so you have a white square with a black border)... does that add one pixel all around the edge or do the routines take account of that when they draw it? It turned out that a lot of things weren't stated explicitly and it took a lot of trial and error to find out what the actual truth was. It wasn't helped by the fact that the puzzles, GTK and SDL all had their own ideas about what should happen.

This showed up time and again. The puzzles wanted a filled circle (so a circle of one colour drawn over a smaller circle of another colour to give a border). GTK apparently has a command especially for this, but with SDL, I had to draw two seperate circles manually. But does the larger sit inside the coordinates specified or outside?

Within a few hours, I had a buggy version of Net that loaded up two windows, one GTK and one SDL, and tried to draw the same things on both. It crashed, it had streaks, stray lines, disjoint lines, incorrect colours, but it was clearly Net.

Net was tricky, though, in that it used quite a lot of the functions and I wasn't too sure I was even going to be able to remove the GTK stuff for a while. A simpler target appeared – Untangle. This used a handful of functions (draw_line, draw_circle) and because the circle and line routines were so simple, it displayed almost perfectly without me having to do any more.

The next step was to get some input. Although I could control the puzzles from the GTK interface, the SDL interface didn't even know what a mouse-press was. After a few more hours, I had an SDL "event loop" that duplicated the GTK functionality. It wasn't long before a complete game of Untangle could be played from start to finish using the SDL functionality alone.

At this point, all the code was still being written in the file "gtk.c", to save me having to re-write the Makefiles, but, by commenting out an awful lot of vital GTK code, I was able to compile the whole thing to a single, massive executable that could run on the GP2X and didn't need GTK at all.

I tried it out and, after a few false starts, it worked on the GP2X. Getting input to the GP2X was more tricky because it didn't know what a mouse was either yet, so I had to dig out an old joystick (to SDL, the GP2X buttons look like joystick buttons) with lots of buttons and I used that to rewrite the input layer so that it could be controlled with joystick buttons. The problem was that this was a bit of a bodge just so that something worked and I ended up rewriting this "pretend these joystick buttons are a mouse" code several times before I got it right.

After removing or commenting out an awful lot of apparently vital GTK code, I released Untangle on its own as a pure-SDL game and I got hundreds of downloads, several words of encouragement and at least one person re-wrote their own version of Untangle because mine didn't work on the touchscreen GP2X's (not really surprising as I didn't have one to test it with!).

Bouyed up by the wave of enthusiasm, I carried on and ended up coming back to Net. It seemed to use a good enough complement of all the functions necessary that if I could get Net working on its own, I could get almost all the other games working. This is (nearly) what happened, but already there were cracks. The boxes on Net didn't line up properly and the timer routine necessary to animate the turn of the boxes could only be done with GTK or SDL but not both simultaneously (otherwise one “click” would show up as two!).

Eventually, the SDL timer code caught up with the GTK code and I was able to completely switch it over to use the SDL code. I didn't realise at the time but my timer code was disgusting and doing just about everything wrong but it worked in virtually all the games without a hiccup.

With timers and most of the drawing functions workly well enough, I tried the other games (which I hadn't yet touched) and to my surprise a lot of them just started working. Some of the more complex ones didn't want to do anything useful and there were lots of interesting bugs... Mines crashed as soon as you clicked on anything, anything that involved drag and drop didn't work at all except for Map which seemed to work almost perfectly and a lot of games weren't displaying in the right colours.

The colour problem was quickly solved when I noticed that I had incorrectly assumed that GTK colours and SDL colours were similar. One is stored as a floating point number from 0 to 65535 for Red, Green and Blue and one is stored as a set of three numbers from 0 to 255. To complicate matters, the puzzles used a floating point number from 0 to 1 for each of RGB. Somewhere along the way, I had completely messed up the conversion between them and it had screwed up all the colours. I managed to fix this quite quickly, though.

I tried to fix the display problems several times but the problem was that there was so much code interacting it was difficult to find the cause. I didn't have access to a nice debugger and using GNU GDB was still a bit of a mystic art to me, so I relied on the programmer's fallback debugger – a lot of printf()'s manually inserted into the code at relevant points in the code to let the programmer know what is going on when it hits them. Tedious, labour-intensive, inelegant but they worked.

I ended up fixing a myriad of problems that had no effect on the actual puzzles yet but which obviously had something wrong with them. Even a simple puzzle like Net can not only draw to the screen, but only redraw certain parts of it, or copy parts of the screen to memory and paste them back later, or ask a timer to do things for them, or react to the user's input, so it was incredibly difficult to find out why a few simple boxes weren't lining up.

In the end, I managed to remove the worst of the drawing problems and cracked on with getting the rest of the games working. I was able to release a set of the games where 90% of them worked enough to be playable soon after. Unfortunately, I was still compiling them one at a time and so they were about 2Mb each and each one included a complete copy of SDL inside it! This was also popular and it wasn't long before I found myself putting out regular releases as I fixed more and more bugs.

Drag-and-drop – a nightmare to simulate with a “virtual” mouse like I was creating using the GP2X's buttons but eventually Untangle worked well enough to play seriously. However, Pegs was far, far too slow when something was dragged. I later found this to be caused by an old bit of code that I was still using that was redrawing the entire screen every time something happened. Removing that made Pegs work again.

Timers – they had to fire on time, every time, and often the games did strange things like set timers and then set them again with a different interval. This threw my simple routines into all sorts of confusion but eventually I fixed the timers to take account of this and things like animations started to work properly. The timer in Mines never worked though and, because Mines was still crashing at almost the same point as the timer ***should*** have started, I ended up trying a lot of debugging on my timer routines to see if they were the problem.

Copy buffers – the games can copy bits of the screen around memory and use them later. A lot of the time this is used to “remember” what was under the mouse cursor and every time the mouse moves to redraw what was underneath it so that dragging appeared as if an object was really moving. This caused some problems because it wasn't entirely clear how large an area of screen the games wanted to copy and sometimes it went off the edge of the puzzle or overlapped with something else. If this happened, the games often crashed. This meant that a lot of checks had to be put in so that I didn't accidentally start writing to memory that wasn't on the screen at all. And having the “source” and “destination” the wrong way around for several weeks didn't help either! Eventually, this started making Map and Guess look correct when someone dragged-and-dropped a colour.

Colours – The GP2X has a rather strange quirk that I didn't notice until after several releases – I was quite happily drawing white and grey on the screen at the same time and happened to notice that in Net, I couldn't “lock”an island. Well, I could, but it wouldn't show up as locked. On the PC with the same code it worked absolutely fine. For a while I ignored it (and people complained in the forums about it at the time but I didn't connect the two) but by chance one day, I was watching my brother play the games and noticed that, from a certain angle, I could see the white outline that I thought was missing. It was just that the LCD on the GP2X greatly decreases the contrast between colours so similar colours look identical. It's only when you tilt the screen that you notice that they are drawing as two different colours. A few tweaks to some colour numbers and I had them visible on both PC and GP2X.

Numbers – Quite quickly, I realised that a lot of the puzzles needed numbers to be typed in. This wasn't necessarily a problem – I was already emulating a mouse using the GP2X's joystick, so emulating a primitive “keyboard” was easy too. When you press R, the number goes up, when you press L, the number goes down and when you press X, the number gets “typed” into the games. It worked amazingly well for just a few short lines of code and it's one of the few parts that's remained relatively unchanged.

Eventually I got almost all of the games working. By this time, I was focused on getting to a magic v1.0 release where they all worked and then I would stop. There was a fly in the ointment, though. That damn Mines game. Everything else worked well enough to play but Mines resolutely refused to do anything but crash. It wasn't an overly complicated game, it didn't do anything that the other games didn't do but it still refused to even let you start a game properly.

I spent a lot of time trying to hunt down the problem but I hadn't really looked inside a game before – they were mostly mysterious collections of graph theory, trees, proofs and solvers and they were almost impossible to unravel. I usually had the games telling me what they wanted (“draw this here”, “draw that there,”, etc.) - I'd never had to look inside the games before. I tried several times to trace my way through Mines up to the point that it crashed and I could never quite keep track of what it was supposed to be doing. All I know is that it kept hitting an assertion – a piece of code that the original programmer puts in when the code reaches somewhere it shouldn't be able to. Removing the assertion just made Mines crash even more, so there was obviously something wrong.

I couldn't just stop releasing the games, though, so I turned my attention to other things, often returning to Mines and that damn problem. I started using button-combinations to call other functions that were already part of the puzzles, such as “Solve”. I added in proper font display, using SDL_ttf, FreeType and the DejaVu fonts, so that Bridges could be played. Once I had font display working, I was able to introduce the “status bar”, which is just one function that gets called with text which the frontend then has to show to the user – a few lines of code and that was working too. I decided to misuse the statusbar and use it for things which weren't technically part of the game... my “keyboard input”, certain messages and so on.

I worked on memory management because I'd been sorely neglecting to do anything properly with the GP2X's memory – I was lazy and it was powerful enough to just take bad memory use in its stride, but there were problems such as I was loading a font into memory every time I printed a line of text... eventually it would have crashed but you could play a game quite happily for ages before it would have crashed, so nobody really noticed. I cleaned up the memory management code a lot, though, because all sorts of nasty things can happen if you mess up the memory.

I started updating the games to match newer versions in the original collection. I re-worked the whole mouse emulation code a dozen times trying to get it to play ball nicely on both joystick-button-based GP2X's and those with touchscreens (It's still not perfect but I'm sick of that damn code almost as much as I was sick of the Mines problems). I started documenting the code, adding lots of error checks for if files were missing, or things went wrong.

I started adding the smaller tweaks – pixel-perfect positioning of text, a proper monospaced font (which, although the collection says puzzles can use, none of them currently do – but my port had the facility just waiting for them to use it!), larger screen support (the GP2X can “fake” a 640x480 screen on its 320x240 LCD and some games really needed to pretend they were drawing on a larger screen to work correctly), adding help text files so people would know how to play the games (which also included a lot of file access routines that would later become....), saving and loading of the games to files, saving and loading the configuration (both thanks to the INIParser library that saved me an awful lot of reinventing the wheel), allowing people to change the game's parameters, adding new games (I found Eddy McNaughten's Maze3D and tied it into the collection and also put a couple of “working but rubbish” games from the official collections' “unfinished” folder into STPPC2x).

Someone (juanvvc) wrote a menu so that the games could be run one after another without having to return to the GP2X's menu. The code was simple enough and the graphics nice enough that it ended up as a permanent part of the port. I eventually managed to rewrite the entire Makefile so that I could join all of the games into one huge executable that could be run... Juan's menu would become a vital component in this and the collection soon became one 2Mb program that could run all the games one after the other. I even got a bug report because someone assumed I must have messed up... before I was distributing 28 seperate 2Mb executables in one zip file and now it was just one... they thought they had a corrupt download!

I had a small accident and lost a day's worth of code but I was still undettered and recreated all my work within hours and carried on coding. But still I couldn't get Mines working. I added a Loading... screen because “Slide” took so long to generate a puzzle that people thought it had crashed. I started my first “hack” on the inside of a game and made Map display the regions that you couldn't change in a more obvious manner. I added presets, even rewrote the entire Makefile now that most of it was no longer relevant, I added menu after menu of options, I passed the code through Valgrind and caught dozens of silly bugs that I'd created and even a few in things like INIParser, I even added some background music (Thanks DrLou!). But still I couldn't solve the Mines conundrum.

I was annoyed at the still imperfect display, and eventualyl found a stupid display bug that was making Inertia look ugly (I was picking a random colour from a random memory location instead of the real colour) and finally fixed the stray lines that were showing up in puzzles (it was all caused by a +1 in the wrong place!). Still, though Mines evaded me!

And then I got annoyed and emailled the original author (Simon Tatham) with a long emailling detailling my woes and the things that I had tried. There may even have been an accusatory tone that the “portable” puzzle collection wasn't as portable as it claimed to be... Mines worked fine on Linux and Windows PC's but every time I ran it on the ARM processor inside the GP2X, it went mad. He reluctantly agreed to take a look and within hours he had found it... the line was:

memset(<memory area>, < -99 >, <number>)

which fills the memory area with <number> lots of the number -99. It looks so innocent. It should be so innocent. But it wasn't. There's technically nothing wrong with that line. The games use memset for all sorts of numbers from -1 to 0 to 255 and none of them exhibit problems. But for some reason, on the PC the above line would happily do what it was ordered to and on the GP2X, it would just make things crash.

It turned out that the Open2x library that I was using had a faulty version of memset. It worked fine for everything, unless you asked it to do a negative number less than -1. Then it went absolutely mad and did incredibly stupid things like filling the memory with strange patterns instead of filling it with -99's.

It wasn't a problem with my code. It wasn't a problem with the puzzle's code. It wasn't a problem with the compiler (I'd even tried several versions). It was a silly bug inside an internal library. A one line test program shows it up straight away but because the Mines code was so complex, the actual memset was ***miles*** away from where the problem occurred and it wasn't even obvious that anything was amiss – memset is such an integral part of the code that you just assume it would work. I didn't want to change my libraries, so I changed my code instead and found a couple of more instances where memset was throwing things out... in all the other cases it never really mattered what memory was filled with and so nobody noticed, but in Mines is just caused everything to crash. The biggest, most annoying problem of the whole port was solved and now all the games worked.

I geared up for a v1.0 release. I added a new game of my own creation - Mosco (I mean the code - the game idea was from a puzzle I found in an Ivan Moscovich mathematics book, hence the name). I perfected the menus, I fixed bugs, I cleaned it up, I checked every avenue, dotted every I, crossed every T... I released it. Thousands of downloads went out.

Then I noticed the bug in Mosco that meant you could complete it without having actually finished the puzzle.

Bugger.

I'm still working on new things to go into STPPC2x, but I consider it “complete enough”. I play it every day on the train to work for hours and just love it. I'm so glad I decided to port it myself. It's been great fun programming again and it's been a great project – lots of things that can be tweaked, lots of ways to change things and customise them, plenty of scope for improvement, more than my fair share of stupid, self-inflicted and incredibly annoying bugs, and plenty of gratitude from the community.

Ledow