BurrTools

Andreas Röver, Ronald Kint-Bruynseels

Email: roever@users.sf.net

Table of contents

Prologue

to my mother (1950-2005)

What are BurrTools? BurrTools contain two main parts. On the one hand there is a program that assembles and disassembles burr-type puzzles. That program contains a graphical user interface (GUI) which allows creation and editing puzzle definitions, solving the puzzle and the display and animation of the found solutions. This is probably the most interesting part for most people. On the other hand there is a C++ library that may help with the search for and design of new puzzles. This library contains all the necessary tools to write programs that do the things that the graphical interface does (and more).

The first part of this document describes the graphical program. It should contain descriptions of all concepts and explain how to use them in the GUI program. The second section will contain a description of the library and some internals. Also, some of the used algorithms are explained. This section is probably only interesting for people wanting to use the library for their own puzzle design explorations.

But first a little bit of history of this program. There are already two programs that do the same that BurrTools can do. One is BCPBox/Genda written by Bill Cutler. Cutler's programs are very versatile, they even can handle space grids different from cubes. The other one is PuzzleSolver3D by André van Kammen. I had bought this program a while ago and have generally been quite satisfied with it. I have taken over quite some ideas from the GUI that André developed. So why another program you might ask. Here are a few reasons:

1. The available programs are not for Linux, which is my operating system of choice,
2. the available programs are binary only programs and hence it is quite hard to do more interesting things like burr growing in an automated way,
3. the programs do cost money,
4. PuzzleSolver3D seems to be abandoned. There hasn't been any update for quite a while, and
5. PuzzleSolver3D has some nasty limits to the shape sizes and the number of possible placements.

Anyway, I was not completely satisfied with the available software. Then in summer 2003 a German computer magazine, started a competition to write a program that counts the number of solutions to a merchandising puzzle of them as fast as possible. My program wasn't the fastest but it was the starting point for the BurrTools.

As there are many people out there that are a lot more creative than I am and that could use a program like this to design nice puzzles I decided to make it public and free[1].

I added a GUI that can work on many operating systems, including Linux and Windows. This has the disadvantage that the GUI looks a bit different from what the normal Windows user is used to, so stay calm if things look a bit unusual, they behave in fact quite similar to how a normal Windows-program behaves.

Lately 2 people played important roles in the development of the program. These 2 are Ronald Kint-Bruynseels and Derek Bosch. Ronald has rewritten the first part of this manual and has generally contributed lots of well organized suggestions. Derek is responsible for the OSX port of the program. Without him there would be no binary for this operating system available.

1. Free as in free speech and as in free beer (see `http://www.gnu.org`)

I want to thank both of them for their work. I also want to thank all the other people that have sent in bug reports, suggestions and praise. Their input is very welcome and crucial to the further development of the program.

All this work has taken nearly 3 years to reach the current state, I hope it was worth it and you have a lot of fun with the program.

Andreas Röver

Part I

User Guide

Chapter 1
Getting Started

1.1 Introduction

The first part of this user guide is written from a *procedural* approach. Rather than sequentially describing the elements of the GUI and their functions, this manual guides you through the program the way you should create your first design. Terms may be briefly repeated in several places of the text. Although too much redundancy has been avoided.

Throughout the text the following fonts and notations are used:

Roman	This font is used for the main text.
Roman Italics	Italics are used to emphasis words or sentences in the main text.
Roman Bold	Used for titles, subtitles and *concepts*.
Sans Serif	Used for elements of the GUI. The same wording is used as in the GUI.
Sans Serif Bold	Used for elements of the GUI and indicating that an in-depth explanation follows.
Small Capitals	Used for program names and puzzle names.
`Typewriter`	Used for file names, directories, URL's and code examples.
`[Typewriter]`	Between square brackets. Denotes a keyboard command.
▷	Followed by a number. A reference to another part in the text that provides more detailed information on the subject. To be read as '*see (also) ...*'.

1.2 Installing BurrTools

1.2.1 Downloading BurrTools

BurrTools is an *Open Source* software project. The most recent release of the program is always available for *free* download at the BurrTools website:

`http://burrtools.sourceforge.net`

At the bottom of that page you can select the proper download for your operation system. This will bring you to the download page where you have to select a mirror site to start the downloading. It's highly recommended to select the mirror site on the server nearest to your location.

Microsoft Windows users can either download the `Windows Binary` (a zipped file which needs manual extraction and installation) or the self-extracting `Windows Installer`. Unless you have a slow connection to the internet downloading the installer is probably the best option. To use BurrTools on a Linux platform you can either download provided pre compiled version or the `Source` files and compile the program on your system (see installation guidelines below).

1.2.2 Installation of BurrTools

1.2.2.1 Microsoft Windows

If you downloaded the `Widows Binary`, just extract the file into the directory of your choice and the GUI is ready to be used. When you opted for the `Windows Installer`, start the executable and follow the instructions on your screen.

1.2.2.2 Mac OS X

For detailed installation instructions please refer to the manual or help files of your operating system.

1.2.2.3 Linux / Unix

For LINUX users BURRTOOLS comes in two versions: a pre compiled binary and source code.

The binary is provided in the hope that it is working on many variants of the LINUX OS. It is compiled for Intel processors and requires a more or less modern LINUX system. As distinct versions of UNIXES differ widely it is likely that the binary will not work for your system. In that case you need to compile BURRTOOLS yourself.

Using the Pre compiled Binary

If you want to try the binary, just download the archive with the current version. Decompress the archive into a directory of your choice and start `burrGui` within that directory. It either works or it doesn't.

If it doesn't make sure you have at least the following libraries installed on your system: `zlib`, `libpng`, `libxml2` and `libxslt`. Of course you also need a working X windowing system. If the program still doesn't work call `ldd burrGui` from the console within the path where you decompressed the files. This will list all libraries required by the binary and where the system could find them. If one of the listed binaries is not available try to install that. If all that doesn't work you should consider compiling on your own or mail me.

Compiling from Source

These installation instructions just contain some hints for the compilation of BURRTOOLS. As BURRTOOLS requires a few not so widespread libraries it is not the easiest task to do this.

To install BURRTOOLS for UNIX you first need to make sure you have the following libraries installed: `zlib`, `libpng`, `libxml2` and `libxslt`. These libraries are usually installed on every LINUX system. You just have to make sure that you have installed the development packages, otherwise it is not possible to compile a program that use these libraries, but just start programs that use them.

Additionally the following libraries are required: `flkt` and `xmlwrapp`.

`Fltk` is the library used for the GUI of BURRTOOLS. It may be included in your LINUX distribution or it may not.

The problem is that we need a version of this library that is not compiled with the default switches. This library must be compiled with C++ exceptions enabled. If you don't do this the program will simply shut down when an internal error occurs instead of displaying an error message and making an emergency save. To compile `flkt` with exceptions enabled you have to do the following:

- Download and decompress as usual
- Run `configure` just as usual
- Remove `-fno-exceptions` from the file `makeinclude`
- Finish normally be calling `make` and `make install`

It is of course possible to use a normal version of the `fltk` library, you just don't get the emergency save feature if there is a bug in the GUI of BURRTOOLS. But as the number of bugs is hopefully quite small right now that should not be such a big problem.

The last library, `xmlwrapp`, can be hard to find, so here a link

`xmlwrapp.sf.net`

This library is compiled and installed in the usual Unix way, read their installation documentation.

Now BURRTOOLS can be compiled and installed the usual way with `configure`, `make`, `make install`.

1.2.3 Files and Folders

After installing BURRTOOLS the following files should be on your system:

`burrGui.exe`	The graphical user interface (GUI) to create puzzle files for BurrTools.
`UserGuide.pdf`	This user guide.
`COPYING`	A text file containing the *GNU General Public Licence*. This file may be deleted to save on disk space, but should always be included when sharing the program. Read it carefully before sharing or modifying the program.
`AUTHORS`	A text file containing information about the contributors to the development of BurrTools. This file may be deleted to save on disk space.
`ChangeLog`	An automatically created text file containing an overview of the changes made to the program since version 0.0.6. This file may be deleted to save on disk space.
`NEWS`	A more readable version of `ChangeLog`. Here all (more or less important) changes to the different versions are collected in a comprehensive list. This is probably the place to look for what changed when downloading a new version. This file may be deleted to save on disk space.
`uninstall.exe`	The uninstall program will only be added after installing BurrTools with the `Windows Installer`.

Also a new folder, `examples`, is created. This subdirectory contains a few examples of existing puzzles that illustrate the capabilities and functions of BurrTools. A brief overview of the examples is presented in Appendix A.

1.3 Concepts and Definitions

Before we start describing the functions of BurrTools, let's synchronise our use of vocabulary and explain a few concepts that are crucial to the way BurrTools works.

1.3.1 Definitions

Voxel. A voxel *(volume pixel)* is a space unit in the 3-D space. The shape of the voxels is defined by the space grid type. Currently BurrTools only supports cubic voxels. Each voxel has one of the following three states: *Empty*, *Fixed* (*Filled*) and *Variable* (▷ 1.3.2). Additional to that each voxel can also contain supplementary information in the form of colours that are attached to the whole or parts of that voxel. Currently BurrTools can only attach one single colour to the voxel as a whole.

Spacegrid. The spacegrid defines the shape and orientation and arrangement of the voxels. Right now there are 3 space grids available in BurrTools: cubes, prisms with a equilateral triangle as base and tightly packed spheres. Spacegrids are always fixed and periodic. That means that a voxel in a certain position will always have the same shape and orientation. So a spacegrid defining, for example, all Penrose patters is not possible because this is neither a fixed nor a periodic pattern.

Shape. This is a definition of a 3-dimensional object. Shapes are assembled out of voxels.

Piece. A piece is a shape that is used as a part of the puzzle.

Multipiece. Some pieces may have the same shape. BurrTools requires you to tell it that two or more pieces do have the same shape, otherwise it will find all solutions with all permutations. So a multipiece is a piece that's used more than once in the problem.

Group (also Piece Group). A collection of pieces (and/or multipieces) than can move with respect to each other, but cannot be separated from one another. Denoted with {}.

Result. This is the shape that the pieces of the puzzle are supposed to assume once the puzzle is assembled.

Problem. A problem in BurrTools consists of a list of pieces and/or multipieces, a result shape and possibly some constraints. You can have more than one problem in a file as it may be possible to have more than a single task with the same set of pieces (e.g. Piet Hein's Soma Cube). In other words, a problem is a statement about *what to do* with the pieces.

Puzzle. A puzzle is either a single problem or a collection of problems.

Identifier. A unique code to identify a shape, colour or problem. This consists of an automatically assigned prefix to which a custom name may be added. The prefix is already unique. It is a letter followed by a number. The letter is different for all items that required identifiers, e.g. it is S for shapes, P for problems and C for colours.

Assembly. An assembly is a physically possible (meaning the pieces do not overlap in space) arrangement of pieces so that the resulting shape is formed. It is not guaranteed that it is actually possible to get the pieces into the positions of the assembly without using advanced technologies like Star Trek beaming.

Solution. A Solution is an assembly with instructions how to assemble/disassemble the pieces.

Assembler. The part of the program or algorithm that tries to assemble the puzzle.

Disassembler. The part of the program that tries to find out how the pieces must be moved to assemble the puzzle. It does this by trying to disassemble an assembly. Some puzzles like Pentominoes don't require checking for disassemblability, they are always constructible. That's why these two tasks are separated.

Solver. A short name to refer to the assembler and disassembler as a unit or just one of these without specifying which one.

1.3.2 Concepts

As described above BurrTools works with shapes which are merely a collection of voxels that each can have either one of three different states: *empty*, *fixed* or *variable*. Particularly the difference between fixed and variable voxels has a great impact on the way the solver works and which assemblies are considered to be valid and which are not. Besides that, the validity of solutions can be further restricted by imposing colour constraints.

1.3.2.1 Voxel States

Empty. The empty state is rather superfluous as it can also be regarded as the absence of any voxel. It is just used in the result shape to indicate the spots that can't be filled at all (holes).

Fixed (or Normal). The normal or fixed voxels *need to be filled* in the final result, otherwise it is not considered to be a valid assembly.

Variable. The variable state is used to instruct the program that for a particular voxel it is unknown whether it will be filled or empty in the final assembly. This is required for puzzles that have holes in *undetermined* places (like all the higher level six-piece burrs). All voxels that *might* be empty *must* have the variable state in the result shape. Right now the variable state can only be used in result shapes and the solver will pop up an error message whenever it encounters a variable state in a normal piece.
Later on variable voxels might be used in piece shapes as well to define voxels in the shape that the program might alter to create interesting puzzles.

The question now is: *why not always use variable voxels* in the result shape? This is a matter of speed. When the program tries to find assemblies and encounters a voxel that it is unable to fill with the available shapes it can immediately abort, if that voxel has the filled state in the result shape as the algorithm is instructed that it *must* fill this particular voxel but it cannot do so, so something is wrong. On the other hand, if the state of that voxel is variable the algorithm knows nothing and has to carry on.

1.3.2.2 Colour Constraints

Colours allow you to add constraints to the possible placement of pieces. This is done by assigning a colour to one or more voxels of the piece(s) and the result shape (▷ 3.5). Then you can set some colour placement conditions for each problem (▷ 4.3). The program will place pieces only at positions that fulfil the colour conditions defined.

These colour conditions currently allow the definition of what coloured voxels of the pieces may go into what coloured voxels in the result shape. The *neutral colour* is different, since it always fits. Voxels in a piece that are in the neutral colour fit everywhere and neutral coloured voxels in the result shape can accommodate for every piece voxel, independent of its custom colour.

Currently the assigned colour is used just like painting the whole voxel with this colour, but in the future more advanced possibilities for colouring and conditions may be added.

1.4 Notes for PuzzleSolver3D Users

BurrTools was initially very much based on PuzzleSolver3D by André van Kammen but diverged quite a bit from that. We strongly advise you to read this user guide since there are some features in BurrTools that work somewhat different as their counterparts in PuzzleSolver3D and there are also some functions that PuzzleSolver3D doesn't have. Below are the most prominent differences that need your attention:

1. BurrTools doesn't handle holes automatically as PuzzleSolver3D does. This may at first sound like a disadvantage but in fact it isn't. Unless you select *'Outer limits of result must be filled'* on the solve tab, PuzzleSolver3D treats all cubes of the target shape as cubes that might be filled but don't need to be. Cubes that *must* be filled however speed up the search process. The more there are of these (as compared to the total number of cubes), the faster the solver will run as fewer possibilities are left to test. BurrTools requires you to specify exactly which cubes in the result shape must be filled and which ones may be empty.
2. The BurrTools solver doesn't automatically detect multiple identical pieces. You need to specify, if a piece is used more than once. If you just copy them the way you do in PuzzleSolver3D the program will find way too many solutions. For example, with Bruce Love's Lovely 18 Piece Burr it will find nearly 40,000,000 times as many solutions as there really are. So be careful.
3. BurrTools allows you to define multiple problems in a single session. So you can, for example, save all the Soma Cube (Piet Hein) problems within one single file.
4. BurrTools has no limits to the number and size of pieces. You can have as many pieces as you want and they are not confined to a grid of $24 \times 24 \times 24$.
5. There is no limit to the number of possible positions for the pieces. So it won't happen that BurrTools complains about too many placements. As long as your computer has sufficient memory the program will merrily continue working – even if it would take longer than the universe exists – to complete the search.
6. BurrTools supports another gridspace besides the cube space supported by PuzzleSolver3D. This allows the design and analysis of completely new puzzles

1.4.1 Importing PuzzleSolver3D files

BurrTools also has capabilities for *importing* PuzzleSolver3D files. So there's no need to redo your designs from scratch, although some postediting may be required because of the differences in handling duplicates of pieces and holes in the puzzle.

There are 2 possibilities for the holes. Depending on whether the option "Fill outer Cubes" is enabled or not when you solve the puzzle with PuzzleSolver3D you must either make the inner cubes of the result shape or the whole shape variable when you want to get the same results with BurrTools. This can be done with the tools described in section 3.8.1. With these tools you can make inner and outer cubes of a shape variable.

The duplicate pieces are handled automatically. BurrTools adds all shapes to the new puzzle but does not add duplicates to the problem instead the counter for the original is increased. The unused shapes are marked as unused and can be deleted when they are not required.

Chapter 2
The BurrTools Interface

When BurrTools is started for the very first time the GUI will look like Figure 2.1 which shows the main window. Although some small variations may occur depending on your operating system, screen resolution and display preferences settings. The GUI has four major parts. On top there is a *menu bar* that allows handling of files and offers extra functionality as well as some preferences settings for the program. At the bottom there is a traditional *status line* presenting relevant information about the task at hand. In between there is a *tools section* on the left and a *3-D viewport* on the right.

2.1 The BurrTools Menus

Below is a brief overview of the main menu entries with references to the places in the text where a more detailed explanation is provided.

File. This menu holds the procedures for handling files within BurrTools and for exiting the program (▷ 2.1.1).

Toggle 3D. Swaps the 2-D and the 3-D grids for the Entities tab (▷ 3.4.1).

Export. Contains a submenu with 2 entries. One allows you to export the contents of the 3-D viewer that can be used to create high quality solution sheets (▷ 6.2). The other allows you to create STL files for 3-D printers (▷ 6.3)

Grid Parameters. This menu entry will allow you to change parameters for the currently used space grid. These parameters include things like scaling of axes or skew. Not all space grids support parameters.

Status. This opens up a window containing lots of maybe useful information about the shapes of the puzzles (▷ 3.10).

Edit Comment. Allows appending textual information to the puzzle file (▷ 6.1).

Config. This menu item provides some preferences settings (▷ 2.1.2).

About. Shows a window with some information about the program.

2.1.1 The File Menu

The **File** menu has all the traditional entries for handling files. Many of these are well known from other software and need not much explanation. Some of the items also have keyboard short cuts as indicated in the menus. Prior to executing most of these commands a warning (and option to cancel) is given whenever changes to the current design haven't been saved yet.

New. Starts a new design after removing all the information of the current one. The first thing that happens when you start a new puzzle is that you will be asked which spacegrid to use. When BurrTools is started it always starts with a puzzle that uses the cubes spacegrid, so when you want to use another grid you need to use this menu.

Load. Opens a BurrTools `*.xmpuzzle` file. A notification will pop up when a *partially* solved design is loaded. Short cut: `[F3]`.

Import. This entry opens a traditional file dialogue that allows importing PuzzleSolver3D files (`*.puz`) into BurrTools. Although these imported designs often can be subjected to the solver right away, some postediting may be required because of the differences in the way BurrTools handles holes in the result and uses duplicates of pieces. BurrTools will import *all* the pieces from the `*.puz` file and assign them to the shapes S1 to Sn-1. Accordingly, the *result* from the PuzzleSolver3D file will be assigned to the last shape (Sn). Also a problem definition is automatically created (▷ Chapter 4).
Since all imported shapes consist only of fixed voxels, the result shape may need some editing (puzzles that have internal holes or pieces not filling the outskirts of the result shape) to make the solver run. Also duplicated pieces are preferably deleted from the Shapes list (▷ 3.2) but certainly from the Piece Assignment list (▷ 4.1.2), otherwise BurrTools will find way too many solutions as it will differentiate between all the possible permutations of these identical pieces.

Save. Saves your work into a `*.xmpuzzle` file. If the design had not been saved before (indicated with '`Unknown`' in the BurrTools windows title bar) the Save As command will be activated. Short cut: `[F2]`.

Save As. Allows you to save any changes to a new file and thus keeping the original design the way it was.

Quit. Shuts down BurrTools.

Except when the solver is actually *running*, saving your work is always possible. This means that after stopping (pausing) the solver it is possible to save the results found thus far. Later on these partially solved puzzles can be loaded again and the solving process may be resumed. This allows you to subject 'huge' problems (e.g. 25 Y-pentominoes in a $5 \times 5 \times 5$ cube assembly) to BurrTools and have them solved in several sessions overnight or whenever you don't need your computer for other tasks.

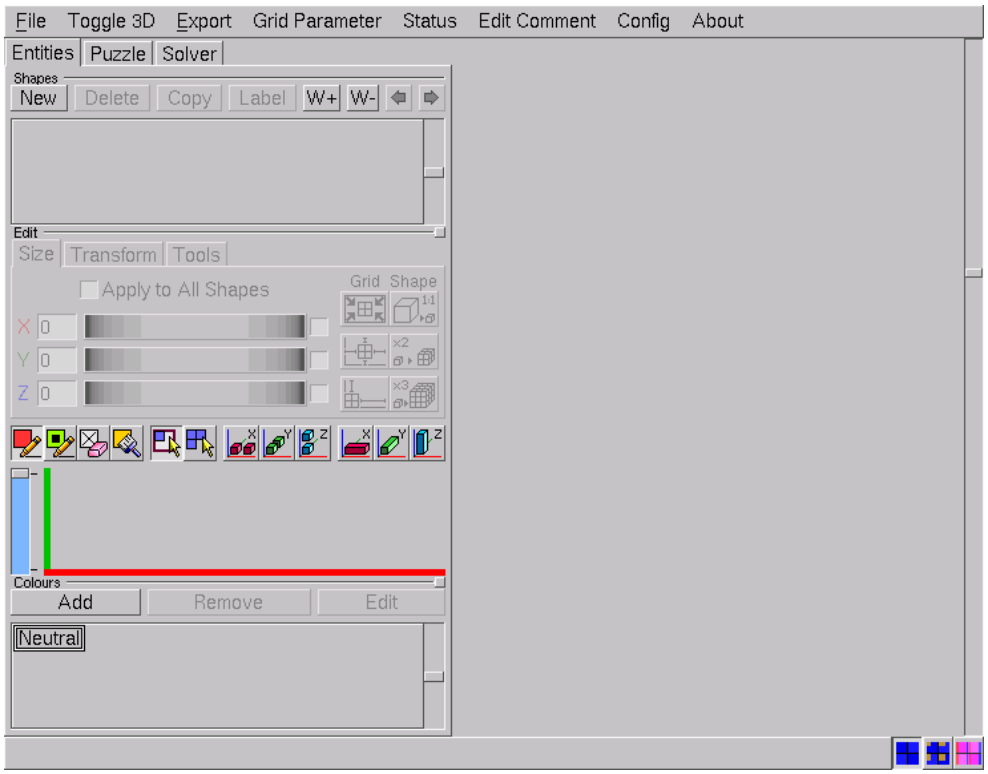


Figure 2.1. The main window on start-up

2.1.2 The Configuration Menu

The **Config** item on the menu bar opens a new window (Figure 2.2) to set some options for the GUI. These settings will be stored in a file that is either in your home directory (Unix) or in your profile (Windows). The program will use these settings each time it is started.

Fade Out Pieces. This option affects the way pieces that *become separated* from the rest are depicted. Hence, the effects are only visible after running the solver (▷ 5.4.3).

Use Lights in 3D View. This option toggles the use of a spotlight in the 3-D viewer. When disabled the items in the 3-D viewport get a uniform (high) illumination, whereas enabling this option provides a more rendered appearance of the objects by adding a spotlight in the upper right corner of the 3-D viewport and shading the faces of the objects. However, on some systems this may result in a relatively dark left bottom corner that can hamper a clear view on the objects.

Use Tooltips. By default BurrTools shows tooltips for most of its controls, but to the more experienced user these become soon very annoying. This option allows you to switch these tooltips off.

2.2 The Status Line

The status line has two parts. On the left information about the task at hand is given and on the right are some tools to alter the 3-D view. Currently you can select there *how* the 3-D view show the shapes. You have the choice between the normal view where each piece is drawn with its neutral colour, a view where each piece is drawn with its colour constraint colour (if it has one assigned ▷ 3.5). The third option is an anaglyph called mode (see figure 2.3). In this mode the pieces are drawn using the red-cyan method to display real 3-D. You can view these with a red-green, red-blue or red-cyan glasses. The *r*ed glass must be in front of your *r*ight eye.

2.3 The Tools Section

In between the menu bar and the status line is the most important part of BurrTools. The section that allows you to submit existing puzzles to the solver, but more even important lets you create and test your own designs.

2.3.1 The Puzzle People

The tools section has three major tabs that somewhat can be compared with real people in the world of mechanical puzzles. First there is the Entities tab, which can be seen as the craftsman who *creates* different *shapes* but hasn't to bother about the purpose of these (▷ Chapter 3). As long as his saw blade is sharp he's the happiest man in the whole wide world. Next, we have the Puzzle tab. This is the weirdo who thinks it's fun to come up with completely insane *problems* to be solved with the otherwise very innocent objects of our craftsman (▷ Chapter 4). However, his contribution to the preservation of our planet is considerable... by saving a lot of wood scraps from the incinerator. And last we have the Solver, the poor guy who spends not only a great deal of his money on these finely crafted puzzles but almost all of his leisure time on *solving* them (▷ Chapter 5), only to feel very euphoric when he finally succeeds. But scientists are still breaking their heads over the question whether this is caused by the sweet smell of success, or is merely due to severe sleep deprivation.

Figure 2.2. The configuration window

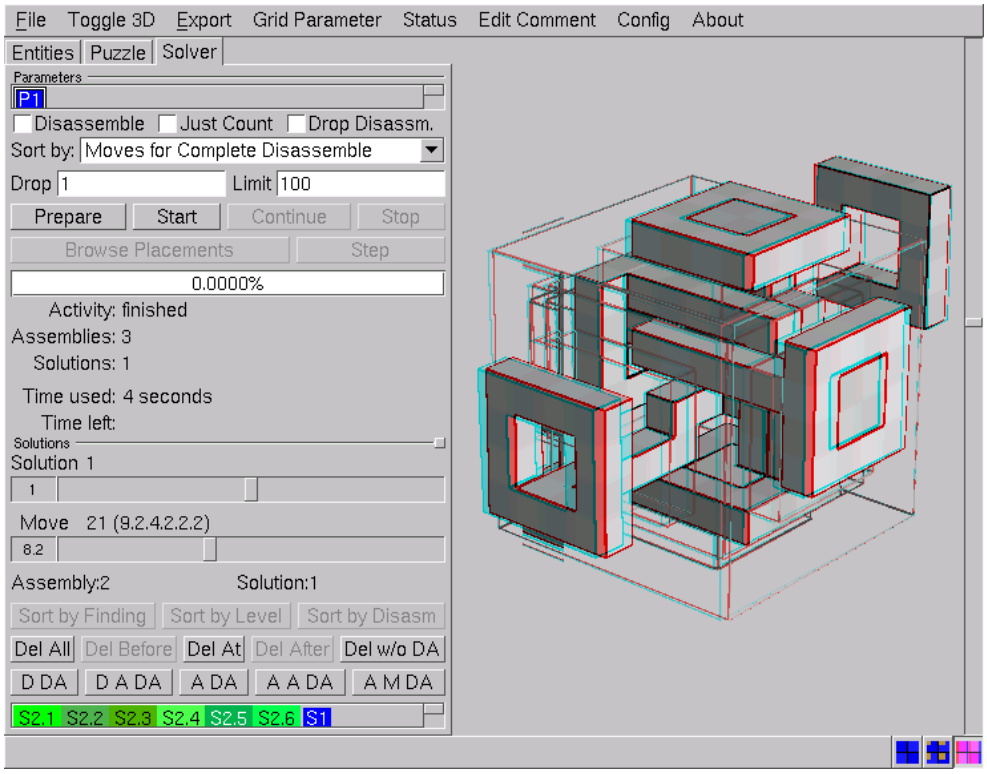


Figure 2.3. Disassembler in Anaglyph Mode

2.3.2 Resizing the Elements

Although the layout of the GUI is designed to suit the needs of most users, it sometimes may be useful to resize some elements to enhance the comfort in using BurrTools. Besides the traditional resizing of the main window, BurrTools has a couple of features to alter the relative importance of its controls.

First, the tools tabs can be made wider or narrower (thus making the 3-D viewport more or less important) by dragging the right edge of the tools section. Hovering your mouse pointer over that edge will make it change into a left-right arrow, indicating that you can start dragging it.

Second, within each of the three main tabs some sections (panels) can be resized as well. For example, if you have a design with many different shapes but no colour constraints at all, reducing the size of all colour related controls and maximising those concerning shapes could be very advantageous. The panels on the tool tabs are separated by so called resize handles (Figure 2.4). The separators that allow resizing are easily recognised by a little bevelled square on their right end. Hover your mouse pointer over the lines until it changes into an up-down arrow, indicating that you can drag the separator up or down to resize the panel.

Note that each section has a minimum size. It is not possible to make it smaller than that minimum size.

2.4 The 3-D Viewer

Normally the biggest part of the GUI is reserved for the 3-D viewport. In fact this 3-D viewer is threefold and has different properties for each of the tabs of the tools section. For the Entities tab the 3-D viewport shows the currently selected shape and reflects all editing operations performed on that shape. Also the x-, y- and z-axes are shown to assist navigating in space. With the Puzzle tab activated an overview of the current problem is presented: the result shape (double sized) on top and a single instance of each shape used as pieces below it. Finally, for the Solver tab, the 3-D viewer can be used to browse all found assemblies and/or show an animation of the moves involved in the disassembly of the puzzle.

Figure 2.4. Resize handles

Any object in the 3-D view can be *rotated* by simply dragging it and the scrollbar on the right allows *zooming* in or out on that object by respectively moving the slider down or up. Note that the zoom settings are independent for each of the three tools tabs.

Extra options for the 3-D viewer are available in the Config menu (▷ 2.1.2).

Chapter 3
Creating Shapes

The key concept of BurrTools is *shapes*. A shape is simply a definition of an object in 3-D space and consists of a collection of voxels (space units). These voxels in turn may have their own characteristics such as *state* and *colour*. Note that this definition also includes shapes made out of voxels that are only attached to each other by a single edge, just a corner or even are completely separated in space. The solver certainly won't bother... but how these shapes could be crafted in the workshop is beyond the scope of the program.

All functions and tools for creating and editing shapes - once the grid type is set - are located on the **Entities** tab (shapes are the *physical entities* that can make a puzzle when subjected to certain rules) which has - from top to bottom - three main sections (Figure 3.1):

The Shapes panel. This section is mainly a list of the available shapes and has the tools for creating and managing the shapes. Shapes to be edited can be selected in this list (▷ 3.2).

The Edit panel. This section provides the tools to build or edit the currently selected shape. This panel contains a series of subtabs with several tools for adjusting the Size of the shapes, Transform them in 3-D space and some extra editing Tools. Below these subtabs there's a toolbar with the devices for actually constructing the shapes in the 2-D grid at the bottom of the panel (▷ 3.4).

The Colours panel. This panel contains - besides a list of the available colours - the tools to create and edit custom colours which can be assigned to the voxels of the shapes. These colours can be merely ornamental or can have a serious impact on the way the solver will work by imposing restrictions on the possible placements of the pieces (▷ 3.5).

3.1 Spacegrids

Currently BurrTools handles cubic grids, grids that use prisms with a base shape that is a equilateral triangle and tightly packed spheres. The spacegrid is used for all shapes that are used within a puzzle so you can not have one shape made out of cubes and one using another grid. The spacegrid needs to be set *before* you start with the puzzle. It can not be changed later on. The gridtype is selected when you use the **New** option. Some gridtypes support it to set some parameters of the grid, like scaling or skew. These parameter can be used to suppress certain orientations for shapes but not to create new puzzle shapes. E.g the sphere grid might some time support a switch to turn it into a space of rhombic dodecahedra. This space is very similar except that some orientations that are possible with spheres can not be done with the dodecahedra.

Same for cubes: there might be a parameter that scales the cubes in y-direction. If that values differs from the x-direction value it will be only possible to turn the cubes by 180° when rotated around the x-axis.

3.2 Creating Shapes

The very first step is to initialise the shapes that can be used in your puzzle design. All the tools to do so are just below the **Shapes** caption (Figure 3.1). Clicking the **New** button starts a completely new one with an empty grid, while **Copy** allows you to edit a previously entered shape without destroying the first. Obsolete and redundant shapes can be discarded with the **Delete** button.

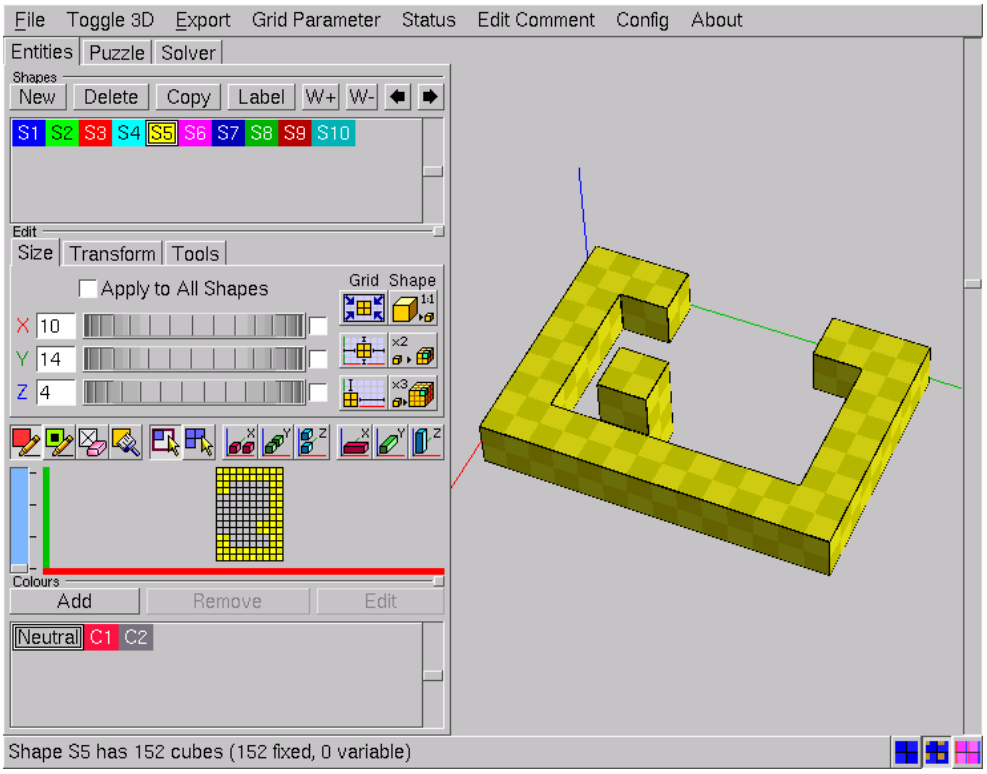


Figure 3.1. Creating shapes on the Entities tab

All shapes are identified with an '**S*x***' prefix. This prefix serves as a unique identifier for the shape throughout the GUI and cannot be removed or altered, but **Label** allows you to add a more meaningful name. Note that on the status line the shapes will only be referred to by their prefixes.

By clicking an identifier in the list the shape becomes selected and ready to be edited. Also a short description of that shape appears on the status line. The currently selected shape is indicated with a white border around its identifier in the shapes list.

The buttons with the arrows pointing left and right allow you to change the position of the shape in the list. The first one moves the selected shape toward the front of the list, whereas the other button moves the shape toward the end of the list. Note that rearranging shapes will cause to change their prefix but not the additional name.

Unlike the pieces in PuzzleSolver3D shapes don't need to be part of the puzzle. This means that you can build a file that contains a vast number of shapes, e.g. all 59 notchable six-piece burr pieces, of which you assign only 6 to the pieces of your puzzle design.

Finally the shapes have an additional parameter: the weight. This value is used when constructing the disassembly animations. When the disassembler has found 2 groups of pieces that can be moved against each other it needs to decide which group to actually move and which to keep where it is. This decision can be influenced with the weight. The program searches the maximum weight in both groups and the one group that has the bigger maximum weight will be kept in place and the other group will be moved. If both groups have the same maximum weight the group with the smaller number of pieces will be used.

3.3 Grid Functions

Since shapes are defined as objects in 3-D space and theoretically 3-D space is unlimited in size, it's convenient somehow to be able to define a more feasible subspace to work with. This, and some more advanced scalings of the shapes, can be accomplished with the functions on the **Size** subtab (Figure 3.2) of the **Edit** panel.

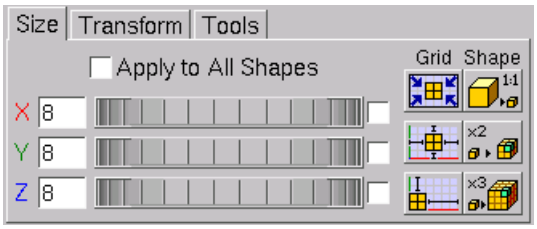


Figure 3.2. Grid and scaling functions

Note that the tab might look slightly different for different gridtypes. For example the sphere grid doesn't have the shape buttons as those are useless with this grid.

3.3.1 Adjusting the Grid Size

When the very first shape is initialised it has a default grid size of $6 \times 6 \times 6$, but all other new shapes will inherit the grid size of the currently selected shape. This feature can be very useful in creating a series of shapes that are restricted with respect to certain dimensions (e.g. all pentacubes that fit in a $3 \times 3 \times 3$ grid). Selecting the proper shape before creating a new one often can save a considerable amount of time by avoiding otherwise necessary grid adjustments.

Adjusting the grid size to your needs can be done either by entering values in the input boxes next to the axis labels or by dragging the spin wheels. When you enter values the grid will be updated as soon as you select one of the other input boxes (either by a mouse click or by the `[Tab]` key) or when you press the `[Return]` key. Note that the grid is also updated by simply clicking in or next to the 2-D grid. To avoid unexpected results it's recommended always to confirm the entered values with the `[Return]` key. Increasing any grid dimension is completely harmless, but decreasing them needs some caution since it can destroy parts of the shape.

The checkboxes for *linking adjustments* - to the right of the spin wheels - allow you to adjust two or all dimensions simultaneously. All linked dimensions will increase or decrease by the same *absolute* amount. However, none of the dimensions can be made smaller than 1 unit. Linked dimensioning is very useful in creating bigger and complex shapes such as the result shape of No Nukes! (Ronald Kint-Bruynseels), which is easily done by first creating the central burr in a $6 \times 6 \times 6$ grid and adding the extensions after resizing the grid to $14 \times 14 \times 14$ and centring the 'core' in that enlarged grid.

3.3.2 Advanced Grid and Scaling Functions

BurrTools has some powerful time saving functions to manipulate the position of the shape in its grid or to rescale a shape together with the grid. These features are grouped below the captions **Grid** and **Shape** on right side of the Size subtab. The first set of three will only affect the grid and/or the position of the shape in the grid, the other procedures however will have an impact on the shape as such by scaling it up or down.

Below is an overview of these functions, explaining what they precisely do and with an indication of the purpose they were introduced in BurrTools. No doubt you'll soon find other situations in which these tools can prove to be valuable.

Grid tools. Most of these tools are somewhat extended versions of the more general transformation tools (▷ 3.7) and have the advantage that they can act on all shapes at once (▷ 3.3.3).

Minimise the grid - This function will minimise the grid to fit the dimensions of the shape it contains. Use it to reduce the disk space occupied by your puzzle files. Note that the result of this function is strictly based on the contents of the grid and will have no effect whatsoever on empty grids.

Centre the shape in the grid - This function centres the shape in the surrounding grid thus allowing you to edit all sides of the shape. In some cases this will also *increase* one or more dimensions of the grid by a single unit to provide truly centring. The function is most useful in editing symmetrical shapes in combination with the compound drawing methods (▷ 3.4.4).

Align the shape to the origin - This function brings the shape as close as possible to the origin of the grid. It can very useful if you want to make a descending series of rectangular blocks by copying the shape and manually adjusting the grid dimensions.

Shape tools. Use the following functions wisely because unnecessary and extreme scaling up of the shapes will bear a heavy load on your system resources and can increase solving time dramatically. Also, trying to undo such 'ridiculous' upscalings with the 1:1 tool can take a considerably long time. So, *think twice, click once...*

These tools only make sense for spacegrids where a group of voxels can be group to make a upscaled shape that looks like a voxel of the grid, e.g. a group of 2x2x2 cubes looks like a bigger cube. As this is not working with spheres, those tools are not available there.

Minimise the size of the shape (1:1 tool) - This function tries to make the shape as small as possible without any loss of information and at the same time scales down the grid by the same factor. Use this function to check the design for oversized shapes which would slow down the solver. Note that although this function can undo the effects of both the next scaling functions, the result cannot be guaranteed since the algorithm may scale down beyond the initial size.

Double the scale - This function will double the scale of the shape (and its grid). In other words, it will replace every voxel in the shape with a group of voxels that all have the same characteristics (state and colour) as the original voxel. This can be very useful to introduce half-unit notches or colouring into the design without having to redraw the shape(s).

Triple the scale - This function is similar to doubling the scale. Only now a scaling factor of 3 is used and hence every voxel in the shape will be replaced by 27 identical voxels. This can be very useful if you want to introduce *'pins and holes'* into your design.

3.3.3 Adjusting All Shapes

A last, but certainly not least, item to mention is the **Apply to All Shapes** checkbox. When checked *all* shapes, whether they are selected or not, will be affected by the settings and procedures on the Size subtab. This is very useful and time saving when a certain adaptation needs to be done to all the shapes, e.g. transforming a six-piece burr with length 6 into one with length 8.

However, some precautions are build in to prevent unnoticed destroying of shapes. Manually reducing any grid dimension will still only be performed on the currently selected shape, whereas increasing (which is completely harmless to the shapes) will affect all grids. On the other hand, minimising the grids will be applied to all shapes since it is content related. The 1:1 tool won't affect any shape unless *all* shapes can be scaled down *by the same factor*. This to prevent ending up with an unintended mixture of differently scaled shapes.

3.4 Building and Editing Shapes

Once a shape has been initialised the 2-D grid wherein it can be build becomes accessible on the **Edit** panel. Basically one needs only three tools to create any shape, but some more features are added to make life easy. All these are on the toolbar right above the 2-D grid (Figure 3.3). The

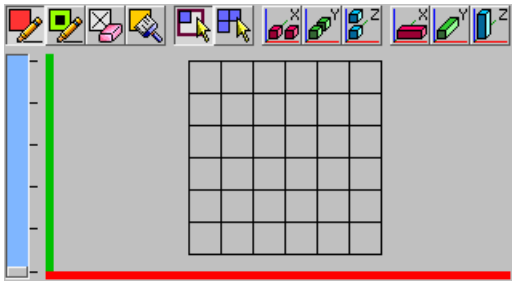


Figure 3.3. Toolbar and 2-D grid

first four buttons are the *basic drawing tools* and *colouring tool*. These are all toggle buttons, meaning that enabling one will disable the others. They affect the presence and/or the state and colour of the voxels by clicking in, or dragging over the cells in the 2-D grid.

Next come two toggle buttons that allow you to select the *drawing style*. This is the way the basic drawing tools will respond to dragging the mouse over the grid cells. Finally, a series of *compound drawing tools* follows. These extend the range of the basic drawing tools and can all be cumulatively added to them.

3.4.1 Navigating in 2-D and 3-D

Building and editing takes almost exclusively place in the 2-D grid to which the 3-D viewport only acts as a visual aid. Both have their corresponding axes in the same colour: *red* for the x-axis, *green* for the y-axis and *blue* for the z-axis. For the 2-D grid, which actually can show only a single layer at a time, the z-axis is represented with a scrollbar (Figure 3.3). By default every new shape starts on the bottom layer and the scrollbar allows you to move up and down through the different layers along the z-axis (the number of z-layers is always indicated with the proper number of ticks along the scrollbar). Another way to navigate these z-layers is by pressing `[+]` (moves up one layer) or `[-]` (moves down one layer) on the keyboard.

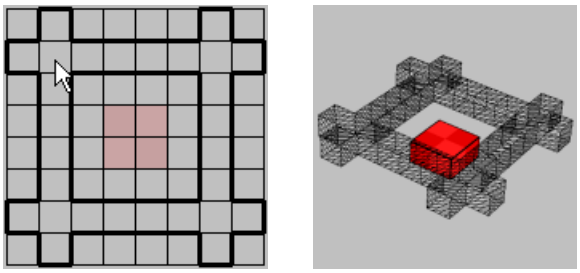


Figure 3.4. Selections of grid cells in 2-D an 3-D

Moving the mouse cursor over the 2-D grid gives an indication of the cell(s) - depending on the state of the compound drawing tools - that will be affected by clicking. These indications are also reflected in the 3-D viewer. Furthermore, to facilitate positioning on different layers every non-empty voxel on the 2-D layer just below the current one 'shines through' in a very light shade of the neutral colour associated with that shape (Figure 3.4). This makes building shapes from bottom to top very easy.

With larger grid sizes the cells of the 2-D grid can become very small, even when the available area for the grid on the Entities tab is maximised. To overcome this inconvenience the 2-D grid and the 3-D viewport can be exchanged. To do so, click the **Toggle 3D** item on the menu bar or press [F4]. Note that this only affects the position of the 3-D viewport for the Entities tab.

3.4.2 Basic Drawing Tools

The basic drawing tools affect only the presence and/or the state of a particular voxel in the shape. In fact they're - together with the brush tool (▷ 3.5.3) - all that's needed to create any shape in BurrTools. The following is a description of these tools. Note that each is also accessible through a keyboard short cut.

Fixed pen - Use this tool to draw *normal* or *fixed* voxels. Fixed voxels are represented by completely filled cells in both the 2-D and the 3-D grid (▷ 3.6). Remember that these fixed voxels *must be filled* in the final result. Keyboard short cut: [F5].

Variable pen - This tool allows you to draw *variable* voxels. In the 2-D grid these variable voxels do not completely fill the cells, but have a narrow border showing the background of the grid. In the 3-D viewport the variable voxels have a black inset (▷ 3.6). Variable voxels instruct the solver that these particular places may be *either filled or empty* in the final result. So variable voxels are only allowed in result shapes and the solver will give a warning whenever it encounters any variable voxels in a shape used as a piece. Short cut: [F6].

Eraser - The eraser will remove voxels from the shape. Note that clicking or dragging with the right mouse button has the same effect of erasing voxels. The eraser tool however proves its use in minute adaptations of shapes. Short cut: [F7].

3.4.3 Drawing Styles

BurrTools has two different drawing styles. These styles affect the way voxels are drawn/erased or colours are added by *dragging* with the mouse. In drawing shapes by simply clicking 'cell-by-cell' both are equivalent.

Rectangular dragging style ('rubber band') - On dragging over the 2-D grid with the mouse just a *rectangular selection* of cells will be made. This is shown with a heavy border around the selected cells and the voxels will only be altered on releasing the mouse button. Releasing the mouse button outside the actual grid however will make the whole operation void and can serve as some kind of 'undo'. This style not only proves its use in drawing rectangular shapes or parts, but is extremely useful for adding colour to (large areas of) the shape.

Free dragging style - All drawing and colouring operations will be performed on a single cell basis and *as soon as* the mouse cursor is dragged over that particular cell. This drawing style is very useful for creating complex and irregular shapes and colour patterns.

The status of these drawing styles is remembered by BurrTools so that it always defaults to the drawing style that was active on the last shut down of the program.

3.4.4 Compound Drawing Tools

Although the basic drawing tools are all that is needed for creating shapes, some compound drawing tools are added to speed up the process. The compound drawing tools can be added *cumulatively* to the basic drawing tools and only extend the range of action for the latter ones.

Note that these tools always go along the 3 orthogonal axes, so they are very useful for cubes but might needs a bit getting use to for the other spaces as they might behave differently along the 3 axes. The triangular prisms for for example are stacked along the z-axis, side by side along the x-axis and tip by tip along the y-axis.

Symmetrical drawing methods - For every voxel drawn, erased or coloured its symmetrically placed counterpart (with respect to the centre of the grid and along one of the space axes) will be affected as well. Activating only one of these options will double the number of edited cells, whereas activating two or all three will affect respectively four times and eight times as many cells simultaneously. These options are not only useful for drawing symmetrical shapes, but they are also very well suited for finding the centre of the grid and (temporarily) setting the extends of a shape.

Column drawing methods - These options - possibly combined with the symmetrical drawing tools - can really speed up drawing shapes as they will affect *all* voxels that are in the same row or column along one of the space axes. The number of voxels that will be affected depends on the size settings of the grid. Hence, to take fully advantage of these functions the grid should be first adjusted to the proper dimensions.

3.5 Adding Colour

There are basically two reasons for using colours in your puzzle designs. The first is merely *aesthetically* and colours are only used to explore the looks of the puzzle. This can help you selecting the proper species of woods or stains before taking your design to the workshop. The second however is far more important as it uses colours to force c.q. prevent certain positions of particular pieces in the assembly. These *constraining* techniques can be very useful to pursue a unique solution for a puzzle design. Of course one can try to achieve both the aesthetic and constraining goals at the same time. Figure 3.5 shows an example of Dracula's Dental Disaster (Ronald Kint-

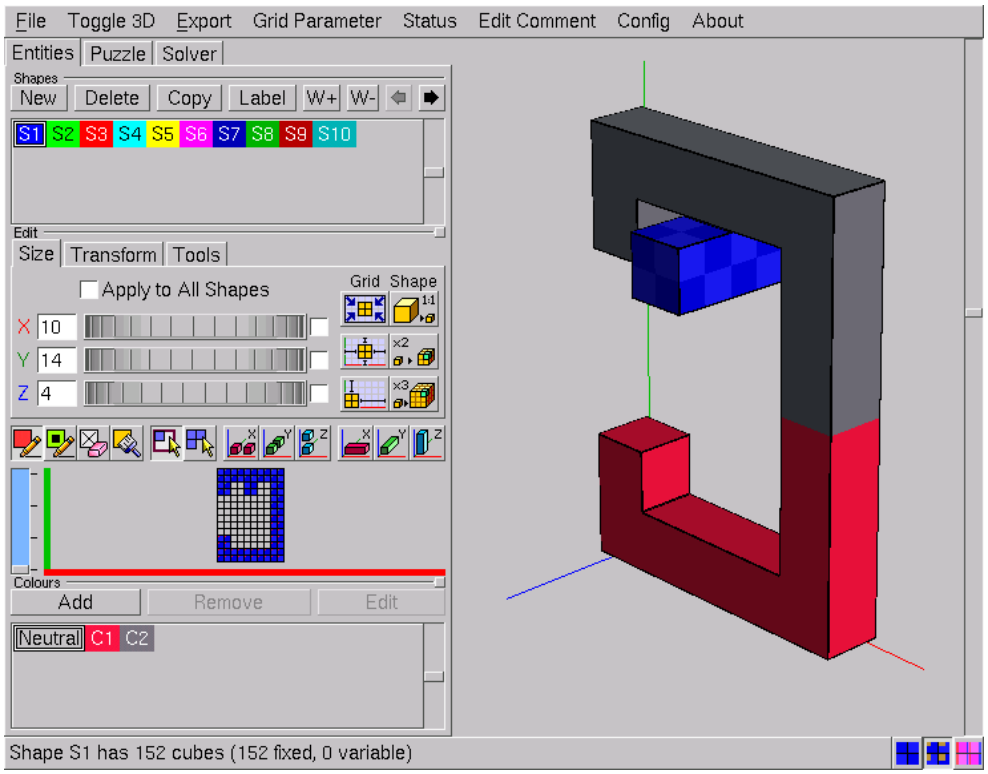


Figure 3.5. A shape with custom colours

Bruynseels) in which colours serve both. The red and black voxels are meant to impose constraints on the placements of the pieces, whereas the white colour of the parts on the inside of the pieces is only used to make them look nice.

3.5.1 The Neutral Colour and Custom Colours

Even when no 'special' colours at all are used, all created shapes do look different with respect to their 'colour'. This is the so called *neutral colour* and is only there to *distinguish* the shapes from one another. These neutral colours are standard for each newly created shape (the first one in the shapes list is always blue, the second one green, the third one red, etc...) and cannot be altered.

As far as the solver is concerned, the neutral colour doesn't even exist as all appearances of it are fully interchangeable. So any voxel in the pieces that has only the neutral colour can go into any voxel of the result shape and every voxel in the result that has no other colour than the neutral can accommodate for any voxel of the pieces, independent of its colour.

Independent from their neutral colour, voxels can have customised colours as extra attributes. To avoid confusion, it's recommended to have these colours well distinguishable from the neutral colours in use, since a custom colour that is identical to one of the neutral colours will have a completely different effect on the way the solver behaves. Almost without exceptions custom colours need some constraint settings (▷ 4.3) to make the solver run.

3.5.2 Creating and Editing Custom Colours

The tools for creating and editing colours are located on the **Colours** panel of the Entities tab. This panel also has a list in which the colours can be selected to be used in the design or to become edited. The **New** button allows you to create a custom colour. A dialogue will pop up and present you the necessary tools to create the colour you need. Accordingly the **Edit** button allows you to transform an already existing colour using a similar dialogue. This dialogue also shows the currently selected colour for comparison (unless the neutral colour is selected, which makes the dialogue to show the default medium grey). Note that the neutral colour can be neither removed or changed. It's important to realise that the BurrTools engine only discriminates custom colours by number as indicated in their prefix '**C*x***' and not by the actual colours themselves. Hence it is possible to create identical colours that nevertheless will be treated as different colours. So, it's strongly advised to introduce only colours for which the difference can easily be discerned. Otherwise, finding out why a puzzle has no solutions can be very hard. The **Remove** button will not only discard the colour from the list, but will also remove it from any voxel that has it as an attribute by replacing it with the neutral colour.

When you add a colour BurrTools automatically add a constraint rule that pieces of this colour can be placed into result voxels of this colour. This is done so because this is the most often used usage case of colours. If you don't want this you have to explicitly remove the rules (see ▷ 4.3). Also when a new problem is created BurrTools automatically adds one rule for each colour that will allow placement.

3.5.3 Applying Colours

Colours can be applied while drawing the shape. Just select a colour and it will become an extra attribute of the fixed pen or the variable pen. Additional colouring can be done by using the *Brush* tool.

Brush tool - This is a *'colouring only'* device and merely adds the selected colour to the voxels without altering their state. The brush tool can also be activated by pressing `[F8]` on the keyboard.

The behaviour of this brush tool is similar to that of the drawing pens. So it obeys the drag styles and can be extended with the compound drawing tools. Note that the right mouse button will still completely erase the voxel.

3.6 Representations

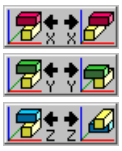
Voxels can either be fixed or variable and each of these can come with or without an additional custom colour. In BurrTools all of these have their own specific representations in the 2-D grid as well as in the 3-D viewport. Figure 3.6 shows an overview of these. In this picture the neutral colour is red (= shape S3) and the custom colour is green (RGB = 0.600, 0.753, 0).

Fixed voxels always fill the cell completely in the 2-D grid as well as in the 3-D grid. In all the pictures of Figure 3.6 the voxels on the left are fixed voxels. Variable voxels only fill the cell partially in 2-D and have a black inset in 3-D (the voxels on the right in Figure 3.6).

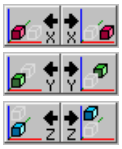
Voxels that have a custom colour added (the yellow voxels in Figure 3.6) show this colour as an inset in the 2-D grid, whereas in the 3-D viewer they are completely painted with this colour (provided that the **Colour 3D View** on the status line is checked, otherwise they will be painted in the neutral colour). Note that in both grids the neutral colours also have a slightly checkered pattern which can assist navigating in space (except for the spheres, they have no checkering).

3.7 Transformation Tools

Editing complex shapes can be very cumbersome and requires often a lot of navigating through the 2-D grid. So, properly positioning and/or orientating the shape in the 2-D grid can save a lot of time. BurrTools comes with a set of functions that help you adjust the position and orientation of the shapes. These functions are grouped on the **Transform** subtab of the **Edit** panel (Figure 3.7). The first thing to see is that the transform tab looks quite different for all 3 available gridtypes. On the top of the figure you see the tab for cubes, blow for the triangles and at the bottom for spheres.

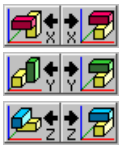


Flip - These 'three' functions are merely *one single mirroring tool* and the only difference is the orientation of the mirrored shape they provide. The first will mirror the shape along the x-axis (or in a plane through the centre of the grid and parallel to the YZ-plane). The other do perform the same task, but along the y-axis (XZ-plane) or the z-axis (XY-plane) respectively. Note that each button can either undo its own action as well as the actions of the other buttons since the result of each function can be obtained by simply rotating the outcome of any other. However, there are three buttons to provide some control over the orientation of the mirrored shape in the grid space, which can have a time saving effect if the shape needs further editing.



and more

Nudge - These functions provide *translations* (along the x-axis, y-axis or z-axis for the cubes or along different axes for other gridtypes) of the shapes in their surrounding grids. These buttons have two parts, of which the left part will shift the shape towards the origin of the grid and the right part will move it away from the origin. Note that shifting a shape beyond the boundaries of the grid will (partially) destroy it. So these nudging operations can also be used to erase unwanted parts on the outer limits of the shapes.



Rotate - These functions allow you to *rotate* the shapes around an axis parallel to the x-axis, y-axis or the z-axis. Again, these buttons have two parts, of which the left rotates the shape 90° anti-clockwise (viewed towards the origin) and the right button turns the shape 90° clockwise. To avoid destroying shapes by rotating them the grid may become rotated as well.
The triangle space has only one rotation button for the x and y-axis because it is only possible to rotate by 180° around these axes.

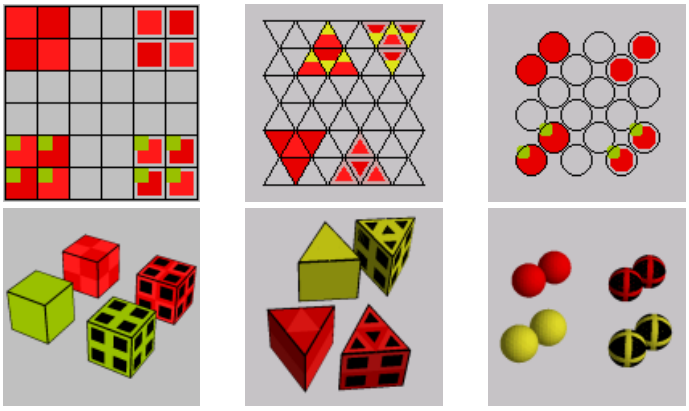


Figure 3.6. Representations in 2-D and 3-D

3.8 Miscellaneous Editing Tools

The **Tools** subtab (Figure 3.8) offers extra editing tools. Currently only some constraint related tools are available.

3.8.1 Constraining Tools

These tools are *mass editing* tools that somehow have an impact on the possible placements of the pieces in the final result. They act either on the inside or the outside of the shape. Voxels that

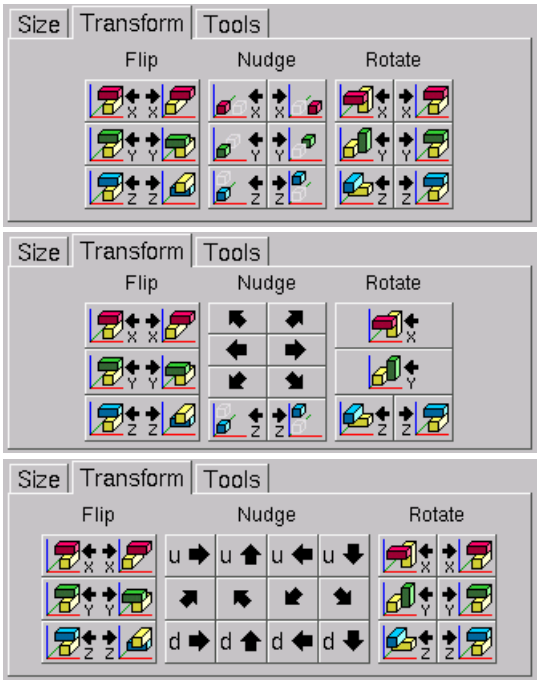


Figure 3.7. Transformation tools

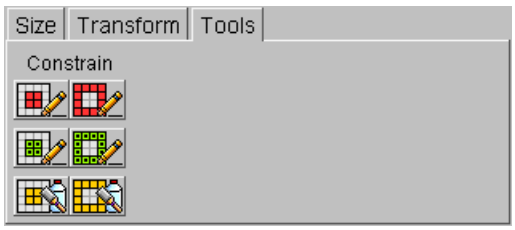


Figure 3.8. Extra editing tools

are considered to be on the inside are voxels that have another voxel adjacent to *all* of their faces. Consequently, outside voxels have at least one empty voxel neighbouring.

Fixed Inside/Outside - These functions allow you to change the state of the voxels that are either on the inside (left button) or on the outside (right button) of the shape into fixed voxels. Although one can think of situations in which these can be useful as such, they are mostly used to undo the effects of the next two functions.

Variable Inside/Outside - These functions will respectively make all the voxels on the inside or the outside of the shape variable. Making the inside variable is very useful for puzzles with internal holes in undetermined places. On the other hand making the outside variable can prove its use in a lot of design situations (e.g. adding extensions to the pieces). Clicking both buttons will make the shape completely build out of variable voxels. Use these wisely as the more variable voxels there are, the slower the solver will run.

Colour Remover - These buttons will remove any custom colours from the voxels that are either on the inside or the outside of the shape and replaces them with the neutral colour. Removing the colour from the inside can prevent having to apply complex colouring to the result shape in situations were the colour constraints are only relevant to the overall appearance of the puzzle.

3.9 Managing Shapes and Colours

Currently only the shapes can be rearranged with the left and right arrow buttons of the Shapes section, but more advanced managing procedures will be added in the future.

3.10 Shape Information

When using the main menu entry **Status** a window (Figure 3.9) like the one above opens and displays all kinds of information about all the shapes available inside the puzzle. The table columns have the following meanings:

Units Normal. Contains the number of voxels inside the shape that have the state fixed.

Units Variable. Contains the number of voxel inside the shape that have the state variable.

Units Sum. Contains the number of voxels inside the shape that are either fixed or variable.

Identical. If the shape is identical to another shape with smaller number the first one of these number is displayed, so if shape 3, 4 and 5 are identical shape 4 and 5 will point to shape 3 but shape 3 will show none. So the table only points to a shape above.

Identical Mirror. A shape is entered, if the shapes can somehow be transformed into the other including the mirror transformation

Identical Shape. A shape is entered, if the shape is identical without including mirrored shapes.

Identical Complete. In this case shapes must be completely identical including colours and not only the appearance of the shape.

Connectivity. This part of the table shows if the shape is completely connected and doesn't contain any separate voxels

Connectivity Face. This part is marked with an X when all parts of the shape are connected via the faces of the voxels

Connectivity Edge. This part is marked with an X when all parts of the shape are connected via an edge or a face of the voxel

Connectivity Corner. This part is marked with an X when all parts of the shape are connected via a corner, an edge or a face

Holes. This part of the table contains information about possible holes inside the shapes.

Holes 2D. A 2D hole is a hole in the shape, if the shape would be 2 dimensional. So the o-octomino has a 2D hole.

Holes 3D. A 3D hole is a completely surrounded region inside a shape.

Sym. This is a column that is mainly there for my help. BurrTools needs to know about all kinds of symmetries a shape can have. If a shape turns up that has a kind of symmetry yet unknown to the program it can not solve puzzles with this shape. So here is a tool to check beforehand and without the need to create a problem. If you ever see a coloured mark in the last column send me the shapes where it turns up. As long as this last column contains only numbers without a color mark everything is fine.

Because calculating all this information can take a considerable amount of time BurrTools pops up a window when it is working on accumulating this table. The window contains a progress bar to guess how much longer it will take. There is also a **Cancel** button at the bottom that lets you abort this calculation and view the already gathered results.

3.11 Tips and Tricks

Below are some tips and tricks that can be useful to simplify your designs, speed up the designing and/or solving process, or can be used as workarounds for some limitations of BurrTools. We encourage the reader to share his own tips and tricks with us so that we can incorporate them in a future update of this document.

3.11.1 Voxel State and Size Tips

State and Solver Speed. The more variable voxels (as compared to the total number of voxels) there are in the result shape the slower the solver will run. Also the number of pieces has an impact on the solving time. Hence, replacing variable voxels with empty spaces for determined holes in the puzzle is to be considered. Also leaving out a piece in complex packing puzzles (and making its position in the result empty) can reduce the solving time considerably.

Size and Solver Speed. Also the size of the shapes has an effect on the solving speed, since bigger shapes inevitably lead to more possibilities: for a $1 \times 1 \times 1$ cube there's only one possible placement in a $2 \times 2 \times 2$ grid (excluding symmetries), but for a $2 \times 2 \times 2$ cube there are four of them in a $4 \times 4 \times 4$ grid. So trying to minimise *all* shapes with the 1:1 tool before taking the puzzle to the solver is highly recommended for complex designs.

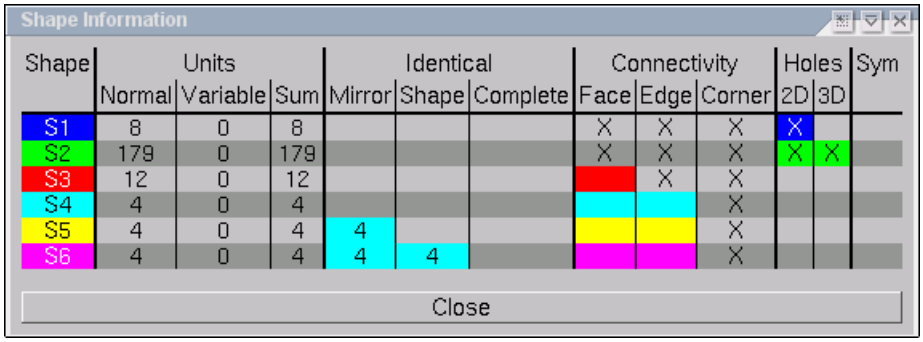


Figure 3.9. The Status window

Complete sets. Often complete sets of pieces (e.g. the hexacubes in Haubrich's Cube) can be easily made by repeatedly copying the current shape and editing it with the properties of left and right clicking.

Symmetry. A detailed treatment of some symmetry issues will be added to the next update of this document.

3.11.2 Colouring Tips

Colouring Shapes. Colouring shapes as a whole is easily done with the brush tool in combination with the rectangle dragging style and z-columns switched on.

Aesthetic Colours. When colours are solely used for aesthetic reasons make sure that the *result* shape has only the neutral colour. This will prevent having to set a lot of constraint conditions.

One-Sided Polyominoes. Polyominoes can be made one-sided by having them two layers high and adding different constraint colours to both the layers. The constraint settings (▷ 4.3) should simply be a 'one-to-one' relationship.

Hiding Pieces. For puzzles in which the goal is to hide a certain piece on the inside of the assembly (e.g. Trevor Wood's Woodworm) *two* constraint colours should be used. One for the exterior and one for the voxels on the inside of the result shape. Also colour the piece that must be hidden with this 'inside' colour and apply the 'outside' colour to all other pieces. The constraint settings (▷ 4.3) must then be such that the piece to be hidden is only allowed to go into the 'inside' colour and the other pieces may go into either colour.

3.12 Simulating non-cubic spacegrids

It is possible to emulate spacegrids different from cubes by just using cubes. This way BurrTools can solve different kind of puzzles. This section will give hints of how to things. It will not contain obvious emulation possibilities like hexagons with 6 triangles or x by y rectangles using several squares, but rather the more complicated possibilities. The chapter can not be complete but it rather wants to show what can be done and give you some initial ideas. If you come up with a cool idea you are welcome to send it to me and I will include it in here.

Generally this emulation requires to use more cubes for one basic unit. This will probably result in a slowdown of the solving process. But this slowdown is not always that grave. BurrTools knows how to merge voxels that are always occupied by the same piece into one, so if there is for example a puzzle that uses hexagonal pieces made out of the triangular prisms and these hexagons are always within a hexagonal grid BurrTools will merge the 6 triangle together and work with the resulting shapes. This only takes some time at the initialisation phase. On the other hand there might be many placements of pieces that fit the underlying cube to triangle grid that are not proper placements and that need to be sorted out first. This can take a long time. Major Chaos by Kevin Holmes for example has a lot of illegal placements for pieces that need to be sorted out. That takes a very long time, but once that is done the solving is actually very fast.

3.12.1 Two-Sided Pieces

If you have pieces that have a top and a bottom there are several possibilities to model that in BurrTools. One possibility is to use colours. Make the piece and the result 2 layers thick. The bottom layer of both will get a special colour.

Another possibility is to add an additional layer that has voxels only in certain places as seen in the picture. The additional voxel prevents the rotation of the shape. But you have to make sure that the allowed rotations are still possible, e.g. if you place the notches in different places rotation around the z-axis is also no longer possible. An example can be seen in Figure 3.10

3.12.2 Diagonally Cut Cubes and Squares

Cubes can be cut in many different ways, the cut that results in shapes such as given in Figure

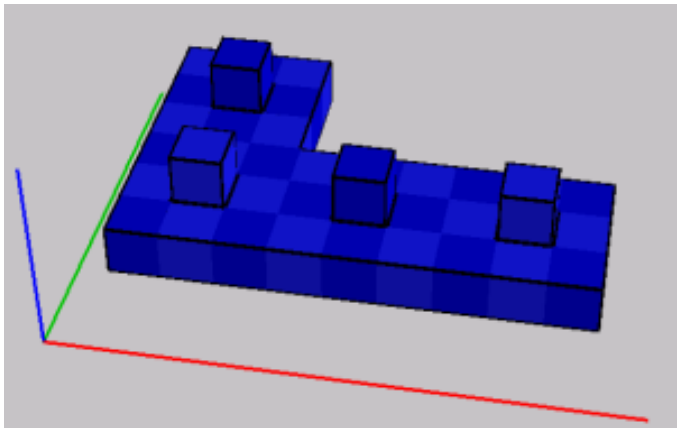


Figure 3.10. Emulate 2 Sided Piece

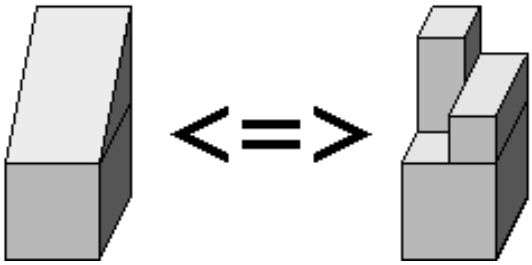


Figure 3.11. Diagonally cut cube

3.11 can be emulated using cubes as seen in the image.

It is, of course, also possible to simulate diagonally cut squares this way. The squares need to be 2 layers thick.

3.12.3 Cairos

Cairos are pentagons but luckily they have only 4 rotations, so it is possible to emulate them using squares. Figure 3.12 demonstrates how that can be done.

3.12.4 Squares with Cuts of Slope 0.5

3.12.5 Edge Matching

Sometimes it is possible to emulate edge matching problems by using notches and dents at the outside of the shapes.

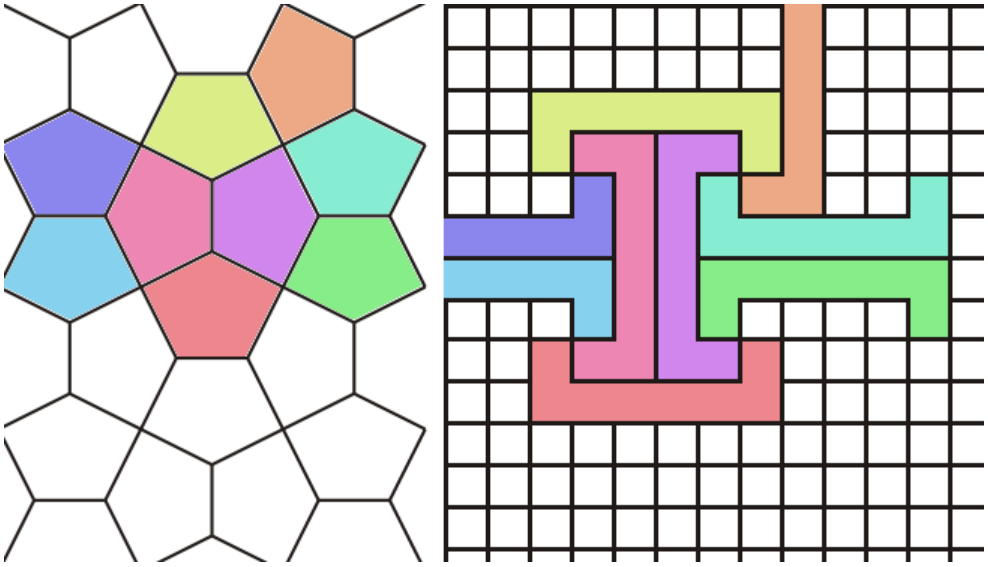


Figure 3.12. Emulation of Cairos using squares

3.12.6 Other possibilities

There are many other shapes that can be emulated. As one example I will show 2 ways to emulate William Waites Knit Pagoda (see Figure 3.13). Additionally to the shape the pieces have an

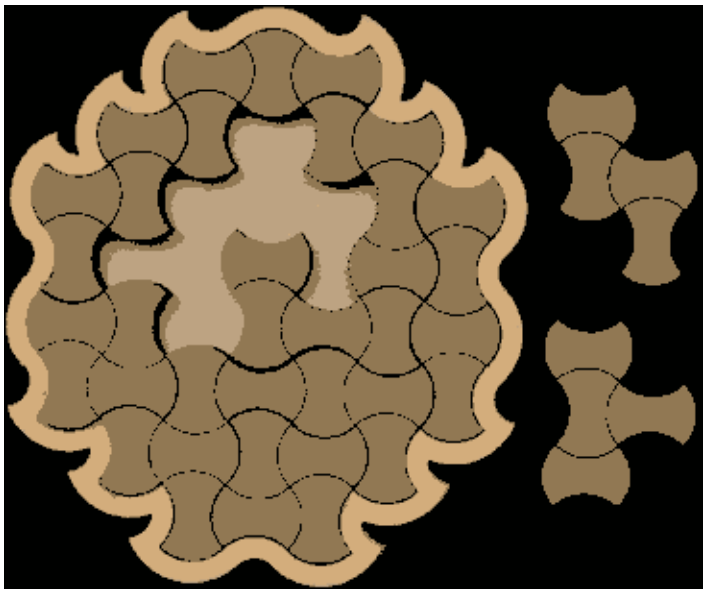


Figure 3.13. The Knit Pagoda

upside and a bottom. Figure 3.14 shows 2 possible ways to emulate these pieces. Both shapes emulate the T-shaped piece seen on the right bottom.

It is quite easy to see that the pink shape working. It is constructed starting with a 3x3x1 square and adding a cube at the centre of one shape if that side is bulged outward and removing one cube, when the side is bulging inwards. Finally add a cube at the centre of the 3x3 square to make it unflippable.

The second is quite a bit more complicated to understand. Here the starting point is a 2x2 square. A cube is added or removed for the bulges just as in the other case but those cubes can not be in the middle. They are at one side so that the cube from an outer bulge can go into a gap created by an inner bulge. The resulting shape for one unit contains 4 cubes along a zig-zag line.

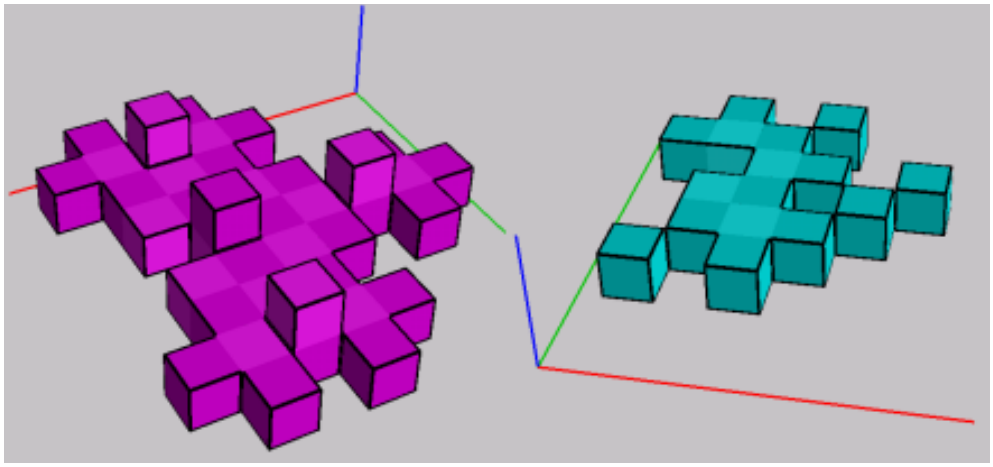


Figure 3.14. Emulation for one of the Knit Pagoda Pieces

You can see it by looking for the lighter cubes in the turquoise shape above. This ways has the additional advantage of avoiding flips because when the piece is flipped over the orientation of the bulges changes and the cubes do not mesh.

Chapter 4
Defining Puzzles

Typically a puzzle problem in BurrTools consists of a collection of pieces (shapes) and a goal, say another shape that the pieces should form when correctly assembled. This is what we call a *simple problem definition*. Note that it may well be not that 'simple' to solve it in real life. More elaborated or *complex puzzle problems* contain also colour constraints and/or grouped pieces.

As stated before, a puzzle can be a collection of problems, either simple, complex or a mixture of both. The **Puzzle** tab (Figure 4.1) provides all the tools needed to build a variety of puzzle problems that are suited for the Solver.

4.1 Defining Simple Problems

As defined above, a simple puzzle problem consists only of a collection of pieces and a result shape that can be assembled (and preferably also be disassembled) with these pieces (Figure 4.2). Bear in mind that a simple problem also implicates that *all* the pieces can be separated from one another. It takes only two steps (which are also required for complex problems) to create such a problem: *initialising* the problem and *assigning* shapes to the pieces and the result.

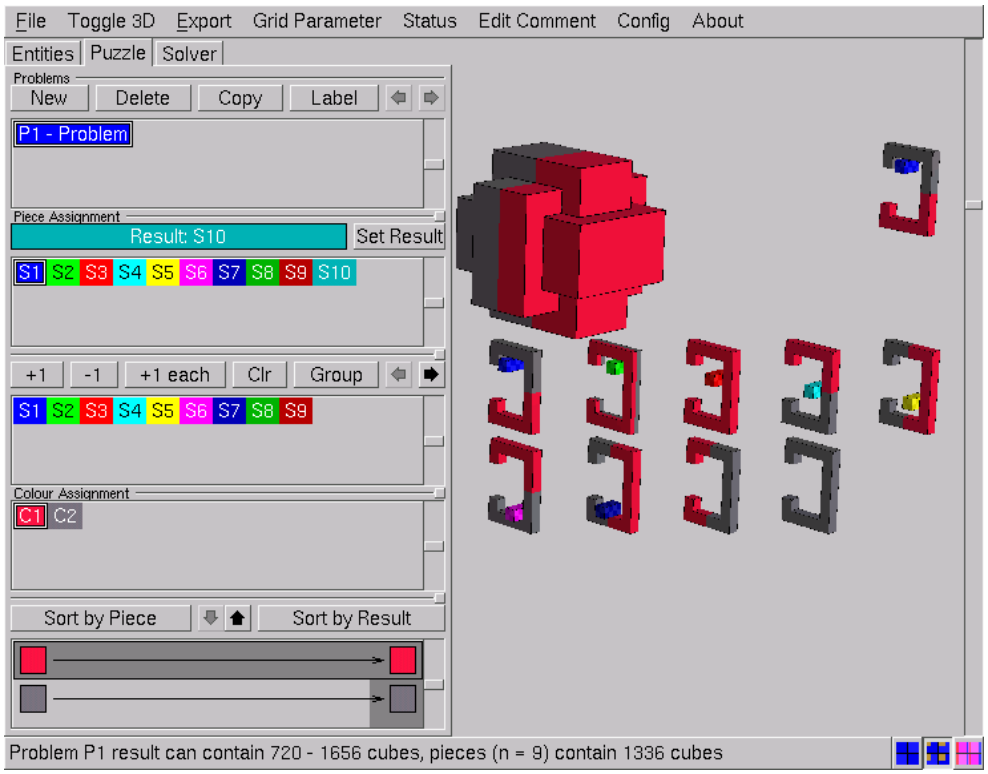


Figure 4.1. Defining problems on the Puzzle tab

4.1.1 Initialising Problems

The first step is to *initialise* the problem(s). All the tools to do so are just below the **Problems** caption. Just like with shapes this can be done by clicking the **New** button to start a completely new one, or by using **Copy** to edit a previously created problem definition without destroying the first. Accordingly, problems can be removed with the **Delete** button. All problems find their place in the problems list below these buttons and are identified with a '**P*x***' prefix to which a more meaningful description can be added by clicking the **Label** button. Also the methods for selecting and rearranging problems are similar to their counterparts on the Entities tab and need no further explanation here.

4.1.2 Piece Assignment

Until now we dealt with shapes as rather abstract concepts. Only by *assigning* these shapes to the pieces or the goal of a puzzle they become meaningful. All available shapes are presented in the top list of the **Piece Assignment** panel in which they can be selected and be given their purpose in the puzzle. Since a strict distinction is made between shapes and pieces, it's not necessary that all shapes are used in a single problem or in any problem at all.

Although not mandatory, it's probably best to assign the result shape first: select the appropriate shape and click **Set Result**. The result shape is then depicted in the top left part of the 3-D viewport (which also shows a smaller example of the currently selected shape) and the status line shows some information about the problem at hand. Next, any other shape can be assigned to the pieces of the puzzle by selecting it and clicking **+1**. This adds a single copy of the shape to the second list which holds all the shapes used as pieces. If multipieces are involved, just add as many instances of the shape as required by the same means. In the list of pieces any multipiece has an instance counter added - between brackets - to its identifier. A single instance of every shape used in the puzzle is shown in the lower part of the 3-D viewer. To make corrections, pieces can be removed from the puzzle by selecting them (they also can be selected by clicking them in the pieces list) and clicking **-1**. Again, this only removes a single instance and needs to be repeated for removing multipieces.

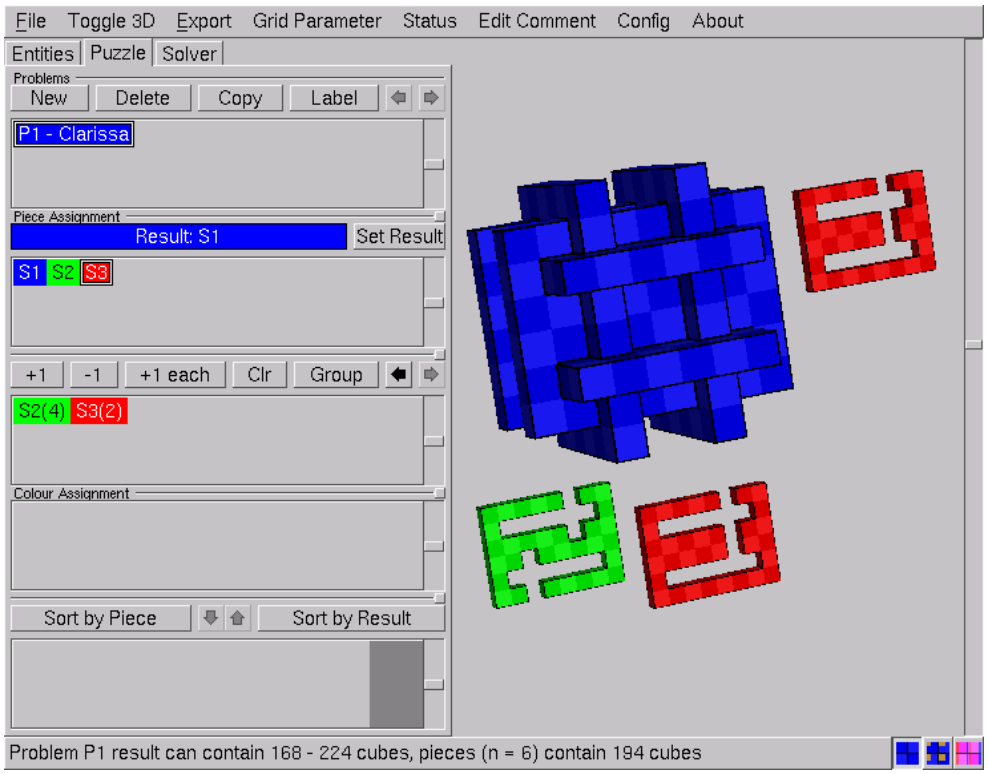


Figure 4.2. A simple puzzle problem with multipieces

Most of the time it is necessary to add one instance of all defined shapes to the puzzle. If there are a lot of them this can take while. This is what the **+1 each** button is for. It increases the piece counter for each shape (except the one assigned for the result) by one. Or it adds a first instance of the shape to the problem. The **Clr** button removes all pieces from the problem.

Since it doesn't make sense to have a certain shape to be result and piece at the same time, the shape set as result cannot be added to the list of pieces. Consequently, assigning a shape that's already in the list of pieces to the result will remove it from the list.

Whenever the total number of cubes in the pieces is within the boundaries set by the result shape (which can be inspected on the status line) this kind of simple puzzle problems can be taken to the solver. Note that the solver won't run when one or more pieces contain any variable voxels.

4.2 Grouping Pieces

Something we deliberately haven't mentioned in the description above is the fact that the solver will halt whenever it is unable to separate some pieces from each other. In other words, the solver will attempt to separate *all* the pieces from each other and reports that no solution exists when it fails to do so. This is just what is required for most puzzles as you need to have single pieces as a starting point. But there are a few puzzles for which you have groups of pieces that are *movable* but *not separable*. Here the piece groups come in handy. Probably everyone familiar with PuzzleSolver3D ever experienced the futile attempts of that program trying to solve such designs by nearly endlessly shifting the entangled pieces back and forth. Not so with BurrTools as piece groups allow you to tell the disassembler that it is OK when it cannot separate a few pieces from one another.

4.2.1 Concept

When the disassembler finds two or more pieces that cannot be taken apart it checks whether all of the involved pieces are in the same group. If that's the case it rests assured and continues. If the pieces are *not* in the same group the disassembler aborts its work and reports that the assembly can not be disassembled. This is the basic idea, but there is a bit more to it.

4.2.1.1 Complete Disassembly

A special case is *'Group-0'*. All pieces in this group *need to be separated* from each other. This group is included so that it is not required to place all the pieces into their own group, when you want to completely disassemble the puzzle. Pieces automatically go into Group-0, so you don't need to take care of that. As a matter of fact you won't even find any reference to that Group-0 in the GUI.

4.2.1.2 Basic Piece Grouping

On the other hand, when dealing with puzzles of which is known that certain pieces (say Sa and Sb) can't be separated from each other, grouping these pieces will cause the solver to report a valid disassembly for which the grouped pieces are treated as a single piece. Be it not a rigid piece since the parts can freely (within certain boundaries) move with respect to each other.

$$\{\mathrm{S}a,\ \mathrm{S}b\}$$
$$\text{Group-1} \rightarrow \mathrm{S}a{+}\mathrm{S}b$$

Of course this technique can also be used (in a truly designing situation) for pieces that *may* be entangled. If these pieces are indeed inseparable the solver will report so, but if they can be separated the solver may report the complete disassembly as well:

$$\{\mathrm{S}a,\ \mathrm{S}b\}\ ?$$
$$\text{Group-1} \rightarrow \mathrm{S}a{+}\mathrm{S}b$$
$$\text{Result: } \{\mathrm{S}a,\ \mathrm{S}b\} \text{ and/or } \mathrm{S}a,\ \mathrm{S}b$$

Now for the hard part: *pieces can be in more than one group*. If you have e.g. a puzzle for which you know that piece Sa either interlocks with piece Sb or piece Sc and cannot be separated from it, but you don't know which of those (Sb or Sc) piece Sa is attached to, you can assign Group-1 to Sa+Sb and Group-2 to Sa+Sc:

$$\{Sa, Sb\} \text{ or } \{Sa, Sc\}$$
$$\text{Group-1} \rightarrow Sa{+}Sb$$
$$\text{Group-2} \rightarrow Sa{+}Sc$$

This way the disassembler detects that both pieces are in Group-1 when Sa and Sb are inseparable and it finds that both pieces are in Group-2 when Sa and Sc cannot let go from each other. In both cases the solver will report a valid disassembly. However, if Sb and Sc are entangled the solver is not able to find a valid disassembly.

4.2.1.3 Grouping Multipieces

All instances of a multipiece need to have the same group assignment, but you can instruct how many of these may be in a group *maximally*. That means you can make statements like 'not more than 3 pieces of Sn *may* be in Group-1':

$$Sa_1, Sa_2, \ldots Sa_n$$
$$\text{Group-1} \rightarrow Sa_1{+}Sa_2{+}Sa_3$$

Now how does it all come together? The disassembler starts to do its work. For each subproblem (a subproblem is a few pieces that it somehow has to get apart) it first checks if there is a unique group assignment for all involved pieces - i.e. all pieces have exactly one group assigned and that group is the same for all of them - it doesn't even attempt to disassemble that subproblem.

If this is not the case it tries to disassemble. In case of a failure it adds the pieces that are in this subproblem to a table of lists of pieces. This is an array and each entry contains a list of pieces. Once done with the disassembler the program comes back to this table and tries to assign a group to each of the lists of pieces in the array. It just checks all possibilities by comparing the entries of the table with the group assignments made by the user. Whenever the sum of pieces (of a certain shape Sx) in such a 'problematic' table entry is bigger than the value the user designated to that particular piece, no valid group assignment can be made. If the program can find a valid assignment the puzzle is disassembled, if it can not the puzzle is assumed to be not disassemblable.

4.2.1.4 Example

Assume we have a puzzle that contains (among others) 5 pieces of shape Sa. Three of them might go into Group-1 and another 2 into Group-2. There is also a piece Sb that might go into Group-1:

$$\text{Group-1} \rightarrow Sa_1{+}Sa_2{+}Sa_3{+}Sb$$
$$\text{Group-2} \rightarrow Sa_1{+}Sa_2$$

After the disassembler ran we have the following lists of pieces in the table:

1. Sa, Sa
2. Sa, Sa, Sb

Now the program has to assign Group-2 to the first set of pieces and Group-1 to the second set of pieces. Because otherwise piece Sb would be in the wrong group, it can only be in Group-1. If there would be another piece Sa in the first set it would not be possible to assign groups because we can only have two pieces Sa in Group-2. But it would be possible to have another piece Sa in the second set.

We have no idea how useful this might be with puzzles as most of the currently available puzzles require a complete disassembly. But who knows, maybe this feature will help in the design of lots of puzzles new and crazy ideas.

4.2.2 Creating Piece Groups

Although the above may sound complicated, implementing piece groups is actually very simple. All actions take place in the **Group Editor** (Figure 4.3) which becomes activated by clicking the **Group** button. Initially the Group Editor shows a tabulated overview of the pieces used in the problem. The first column (Shape) lists the pieces by their prefix and name, the second (n) enumerates the instances of each. Note that it is possible to add or remove instances by changing these n-values.

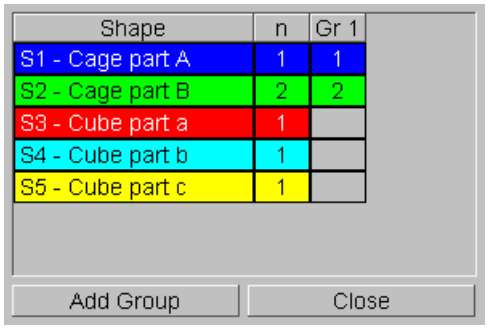


Figure 4.3. The Group Editor

Creating piece groups is straightforward as the **Add Group** button simply adds a new group to the problem. Each new group gets its own column (Gr 1, Gr 2, etc...) in which one can specify the *maximum* number of instances of a certain piece that can go in that particular group. Just click on a cell and it will become an input box. Cells that contain a value > 0 will receive the neutral colour of the corresponding shape, cells with zero are grey and no number is shown. Any group that has no values at all in its column will be deleted on closing the Group Editor. Hence, deleting all the values of a previously made group will remove the group even if its column stays present in the Group Editor.

4.3 Setting Colour Constraints

BurrTools automatically adds the probably most used rules for colour constrains when you add a new colour or when a new problem is created. That rule is that each colour can be placed into itself, e.g. a piece with colour Cx can go into a Result of colour Cx. If you don't want that or if you need additional placement possibilities you can change the colour constraint rules in the colour assignment section.

The **Colour Assignment** panel (Figure 4.4) also has two lists. The first one shows all the available custom colours and allows selecting a certain colour for which then some relations can be set. These relations simply indicate which colour(s) in the result can accommodate for which colour(s) in the pieces. By allowing certain combinations (which is in fact prohibiting all other combinations) constraints are imposed on the theoretically possible placements of the pieces. These relationships are shown and constructed in the second list. This list has three columns of which the first shows the 'piece colours', the last shows the 'result colours' and the one in between clearly depicts the relationships by a series of arrows pointing from the piece colours to the result colours. The list is either sorted by the piece colours or by the result colours. The buttons **Sort by Piece** and **Sort by Result** switch between these two views.

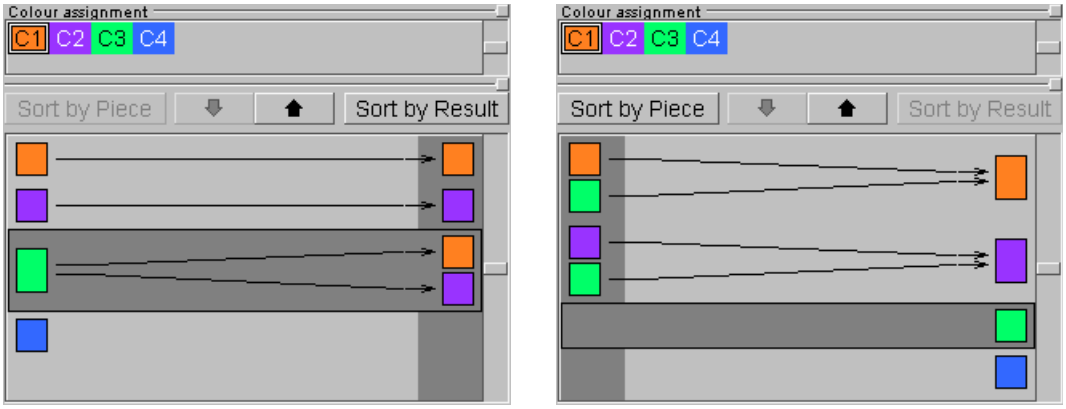


Figure 4.4. Colour assignment

When *sorted by piece* (the left part of Figure 4.4), the bottom list is showing you that every voxel of the pieces with colour Cx can go into every voxel of the result that has one of the colours on the end points of the arrows starting from Cx. When *sorted by result* (on the right in Figure 4.4), the list shows which piece colours will be allowed to go in a particular colour of the result.

To set these relationships, first click the piece colour (or result colour, depending on the sorting method) for which you want to set the constraints. This will activate the 'relations line' for that particular colour which is indicated with a dark surrounding box (note that clicking anywhere on this relations line has the same effect). Next, the down and up pointing arrows will respectively add or remove the colour selected in the top list to or from the constraint settings.

4.4 Managing Problems

Currently puzzle problems can only be rearranged with the left and right arrow buttons of the Problems section, but more advanced managing procedures may be added in the future.

4.5 Tips and Tricks

Some tricks and tips will be added to the next update of the user guide.

4.5.1 Grouping Tips

4.5.2 Constraint Tips

Chapter 5
Solving Puzzles

Solving puzzles is what BurrTools is really about. Without its solving engine the program would be nothing more than a simple tool for drawing a very specific kind of 3-D objects... a task a lot of other software is no doubt even better suited for.

Solving puzzles is very straightforward with BurrTools even if the **Solver** tab (Figure 5.1) has quite a some controls. On top there is the Parameters panel, that contains a list allowing you to select a specific problem to be solved, provides option settings for the solver and has a series of buttons to direct the solving process. Finally, some information of the ongoing solving process is presented.

A second panel (Solutions) has the tools to browse the different solutions found, animate the moves to disassemble the puzzle to *inspect* the solutions in detail and to organize found solutions.

5.1 Solver Settings

In order to make the solver run a problem must be selected first. A list of all previously defined problems is available right below the **Parameters** caption. Selecting problems to be solved is similar to selecting shapes, colours or problems on the other tabs. Note that only the selected problem will be solved and that solving one problem will preserve the results of any already solved or partially

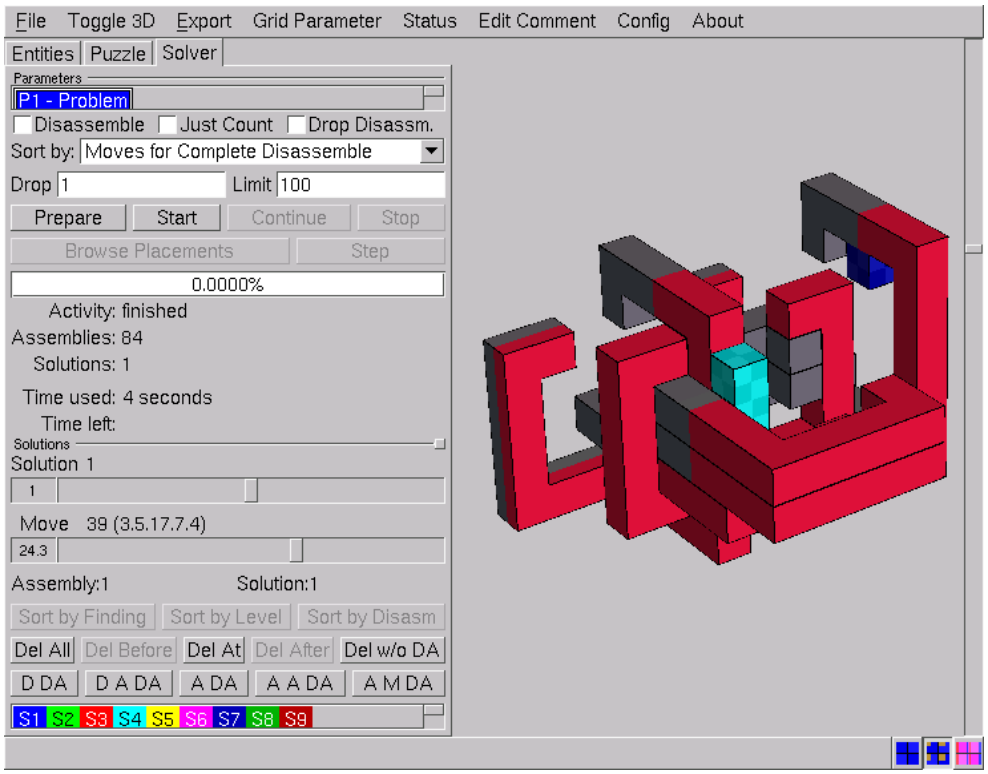


Figure 5.1. Solving puzzles

solved problem. Currently there are the following options for the solver. All deal with the kind of information the solver will report.

Solve Disassembly. When checked the solver will also try to disassemble the assemblies found and only those that indeed can be disassembled will be added to the list of solutions. If this option is left unchecked, the solver will merely search for all *theoretically* possible assemblies, i.e. assemblies for which the pieces do not overlap. Since solving disassemblies takes time (and often far more than assembling), it's recommended to leave this option unchecked for puzzles that always can be disassembled (e.g. PENTOMINO problems and other packing problems). For that kind of puzzles running the disassembler would only slow down the process without any gain in information. Also saving and loading the disassembly instructions takes a lot of time and memory, so if they are not really needed they are just a waste of time.

Just Count. When checked the solver will only count the number of solutions it will drop the found solutions right after they were found. Check this option if you're only interested in the number of solutions and not in the solutions themselves.

Drop Disassm. When checked the program checks, if the found assembly is disassembable and discards the solution if it is not disassembable. But the disassembly is *not* stored, only the assembly and some information *about* the disassembly (like its level). This is useful if you have a problem that has many solutions and you want to find the most interesting solutions. Disassemblies take up a lot of memory within the computer so it is useful to just save some information while solving the puzzle and then later on, when everything is finished recalculate the disassemblies for the interesting solutions.

Sort by. This option lets you choose in which way the found solutions are ordered. There are 3 possibilities:

1. Unsorted: The solutions are sorted into the list in the order in they are found.
2. by Level: The solutions are sorted by the level. First the number of moves to remove the first piece, if that is identical then by the moves for the second piece, and so on.
3. by number of moves to disassemble: The solutions are sorted by the sum of all moves required to completely disassemble the puzzle.

Drop. If a puzzle has very many solutions it might not be possible or even necessary to save all of them. E.g for polyomino-like puzzles it might be nice to keep just every 1000 of the millions of solutions to have a profile of the possible solutions. Here you can specify every how many-th solution you want to keep. A 1 means you keep every solution, a 100 means you keep the first and the 101st and the 201st and so on.

Limit. Limits the number of solutions to be saved. There will never be more than the specified amount of solutions in the list. When the list is full the program has 2 choices:

1. Solutions are sorted: The programs throws away the solutions at the end. So low level solutions are removed
2. Solutions are unsorted: The program starts to throw away every second solution. So when you started with a drop-value of one and the list is full the program starts to drop every 2nd solution is finds and only adds every 2nd solution to the list. But for each added solution it also removes every 2nd solution that already has been added to the list. After a while the list contains only every 2nd solution then the program only adds every fourth solution and removes again every 2nd solution in the list which result in only every fourth solution ending in the list. This sounds complicated but what is does is that is makes sure you have an nice crossection of all the solutions found until then and not just the first or last.

5.2 Solving Puzzles

Next to the solver options are some buttons to direct the solving process. Problems can be solved either in an automatic way or in a (manually) step-by-step manner.

5.2.1 Automatic Solving

An automatic search will proceed until all solutions, i.e. assemblies and disassemblies (when requested) are found. To begin an automated search click the **Start** button. Typically the solving process consists of a preparation phase followed by several cycles of assembling and disassembling. The latter one is of course omitted when the Solve Disassembly option is left unchecked.

The automatic solving process can also be interrupted by clicking **Stop**, but often the solver needs to finish some tasks first before it can actually halt (▷ 5.2.1.2). Any interrupted solving process can be saved to the puzzle file and be resumed in another session with BurrTools. In fact, on loading such a partially solved puzzle BurrTools will inform you about the possibility to continue with the search for solutions. When the solver is interrupted the shapes (▷ Chapter 3) and/or the problems (▷ Chapter 4) can be edited. If no editing whatsoever of these has been done the solving process can be simply resumed (**Continue**), otherwise you need to start all over again. But keep in mind the this saving of the internal state of the solver is very version dependent. So it is likely that a new version of BurrTools can not resume solving a puzzle saved with an older version. So it is good practice to finish solving jobs with one version of BurrTools before updating to the next.

When the solver is running it provides a lot of information about its current state (what it is doing) and an estimate of the time it will need to finish the search. All this information is presented on six lines immediately below the solver control buttons (Figure 5.2).

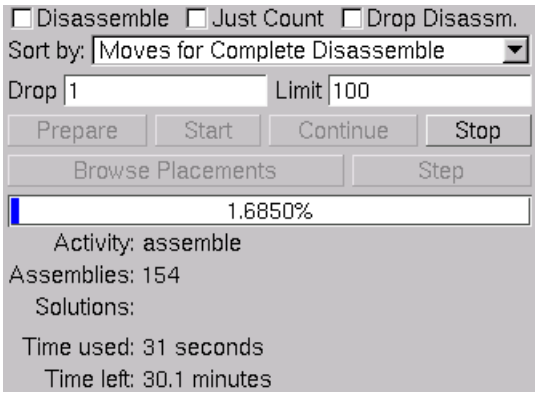


Figure 5.2. The solver information

5.2.1.1 Solver Progress Information

The first line of the solver information is a progress bar indicating the percentage of work it has done. The fifth (**Time Used**) and the sixth (**Time Left**) line respectively show the time already spend on the search and an estimate of the time still needed to finish the solving process. Note that the latter one and also the information about the percentage done are *very rough estimates* since these are based on the possible placements of the pieces already tested and still to test. However, the possible placements to be tested are constantly fluctuating as they are determined by the positions of previously placed pieces (▷ 5.3).

5.2.1.2 Solver State Information

Probably most important is the **Activity** and result information provided by the solver. The Activity line not only tells you what the solver is currently doing, but it also whether the solver can be interrupted or not. The following is an overview of the activities of the solver:

nothing. This indicates that the solver is ready to be started (provided a valid problem is selected) and that no information is available about earlier attempts to solve the selected problem.

prepare. The solver is creating the internal data structure for the assembler. This structure is more or less a listing of all the possible places that all the pieces can go to.

optimize piece *n*. In this second stage of the preparation the placements for each piece are tested for plausibility. Some placements are just nonsense in a way that they result in unfillable holes or prevent the placement of other pieces. These placements are removed from the data structure (▷ 5.3).

assemble. The program is currently searching for assemblies.

disassemble. An assembly was found and is now tested for disasembability.

pause. A search was started and interrupted.

finished. The search was completed, all found solutions, ordered by the set up sorting criterium, can be inspected (▷ 5.4).

please wait. The user wanted to stop the search, but the program still has to finish what it is doing right now. Only the assembler is interruptible. The preparation and optimisation stages need to be finished. The disassembly search also has to be finish first.

error. Something is wrong with the puzzle and an error message, providing more specific information on the error, is usually displayed.

Finally the solver gives information about the thus far found **Assemblies** (i.e. assemblies for which the pieces do not overlap in 3-D space) and **Solutions** or disassemblies (i.e. assemblies that also can be constructed in real life using the particular pieces of the puzzle). Note that the Solutions are only reported (and in fact tested) when the Solve Disassembly option is enabled.

5.2.2 Manually Solving

Besides the automated search BurrTools allows you to run the solver step-by-step. Note that this feature is still under construction and that it has a lot of shortcomings. For instance, it won't add the found solutions to the list or update the solver information. So it certainly needs a lot of improvements in a future release of the program. For the time being it is only useful to check the assembly process when something went wrong with the automated search.

A manual search needs the initial preparation phase as well as an automatic search. This can be accomplished by clicking **Prepare**. The solver will halt after this initial phase and the subsequent steps of the assembler can be seen in the 3-D viewer by clicking the **Step** button.

5.3 Browsing Placements

The **Browse Placements** button opens a window (Figure 5.3) that lets you examine the positions for each piece that will by tried by the assembler. The placements displayed in this window are the possible positions left in the current state of the assembler. So if the assembler has placed a piece Sa and this prevents placing another piece Sb at some positions, these positions of piece Sb will *not* be visible in the list. If you want to see every placement tried you either have to initialise a manual search (click the Prepare button), stop the assembler before is starts to do anything (click Stop while in preparation or optimisation stage) or you have to wait until the assembler has finished its work.

The **Placement Browser** window (Figure 5.3) has a very simple layout and consist mainly of a 3-D viewer and some additional scrollbars. This 3-D viewport, that shows the outline of the result shape and therein the shape for which the possible positions are to be analysed, behaves similar to the one of the main window. Drag the piece to rotate it in space and use the scrollbar on the right to zoom in or out.

Each piece in the problem (note that each instance of a multipiece is available) can be selected with the scrollbar on top of the window. The left scrollbar allows browsing all the different placements for the selected piece. Both these scrollbars can also be controlled with the cursor keys on the keyboard: `[Up]` and `[Down]` for the left scrollbar and `[Left]` and `[Right]` to select the piece. Be careful though, the first stroke on the keyboard that doesn't fit the current scrollbar will just select the other one and the following keystroke will start to move the slider.

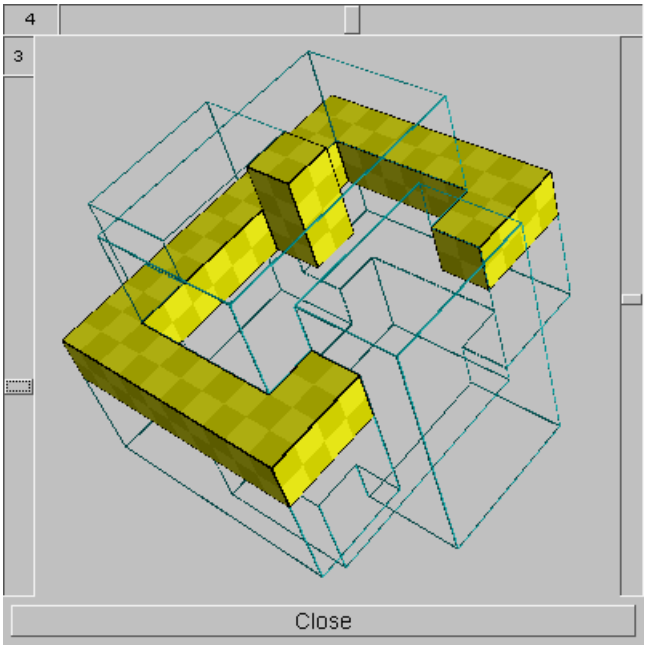


Figure 5.3. Placement browser

5.4 Inspecting Results

As soon as any result is found the solutions list becomes available on the **Solutions** panel and the 3-D viewer shows the first solution in the list. Note that subsequent solutions are simply added to that list and that they only can get sorted by the total number of moves (in case disassembly was requested) after the search is completed. Already found solutions can at any time be inspected and this does not interfere with the ongoing solving process, but bear in mind that on completing the search resetting the scrollbar for browsing the solutions may be needed to show the solutions properly ordered.

This panel has four components: a scrollbar (**Solution**) to browse the different solutions, a second scrollbar (**Move**) to view the moves involved in the disassembly, an array of buttons with very short labels to organize the solution list and a list of all instances of the pieces in the puzzle problem, which allows you to alter the visibility of particular pieces in the solution(s).

5.4.1 Selecting Solutions and Animating Disassemblies

By moving the slider of the top scrollbar (**Solution**) any solution from the list can be selected as is indicated by its number in the text box left of the it. Above the scrollbar there is an indication of the total number of solutions in the list. When the scrollbar is active it can also be controlled by the `[Left]` and `[Right]` cursor keys. Keep in mind that the number of solutions in the list may be different from the real number of solutions. The correct number of solutions for the problem is shown in the solver progress section.

The second scrollbar (**Move**) also has a text box on the left, this time reflecting the stage of disassembly (i.e. the number of moves executed in the disassembling process) of the currently selected solution. Moving the slider to the right will animate the disassembly, moving it to the left will reassemble the pieces in the 3-D viewer. Again, when activated the scrollbar can be controlled by the `[Left]` and `[Right]` cursor keys. Above this scrollbar the *total* number of moves required for the disassembly is shown followed by the level(s) of the selected solution. Note that this scrollbar is only visible for solutions which have disassembly instructions available.

The position of the Move scrollbar isn't affected by selecting any other solution and thus allows easily comparing the different solutions at a particular stage in the disassembly process.

Below the Move scrollbar are 2 fields that show you 2 numbers associated with the currently selected solution. The first is the assembly number and the second is the solution number. Both numbers define when a solution was found. The first found assembly gets assembly number one. But that one might not be disassembable so it gets thrown away. The second found assembly gets assembly number two and if it is also disassembable it gets solution number 1. So you will see assembly 2 and solution 1 in these 2 fields for the given example.

5.4.2 Handling Solutions

The big button group below the Solution selector and animator lets you modify the solutions. They are only activated when no solver is running.

With the buttons in the first row you can resort the found solutions by the same criteria as you can select for the solver. You can sort them in the order they were found (unsorted) or by level or by sum of moves to completely disassemble.

The second row buttons allows the deletion of certain solutions from the list.

Del All. removes all solutions

Del Before. removes all solution before the currently selected solution. The selected solution is the first one in the list that is not removed

Del At. removes the currently selected solution

Del After. removes all solutions behind the currently selected one. The selected on is the last one that is kept

Del w/o DA. remove all solutions that have no disassembly

The last row of buttons allow the addition or removal of disassemblies to the list of puzzles.

D DA. deletes the disassembly of the currently selected solution. The disassembly is replaced by a something containing only information about the disassembly, so you can still sort the solutions

D A DA. deletes all disassemblies

A DA. adds the disassembly to the currently selected solution

A A DA. add the disassembly to all solutions. Already existing disassemblies are thrown away

A M DA. add the disassembly to all solutions that do not have one. Solutions that already have a disassembly are left unchanged

5.4.3 Visibility of Pieces

In the list at the bottom of the Solutions panel all pieces used in the problem are represented by their identifier. Instances of multipieces have a counter added to their prefix which now takes the form '**Sx.n**' and their neutral colour may be slightly modified to tell them apart.

By clicking an identifier the visibility state of that particular piece is altered in the 3-D viewer. Each piece can have three states: *visible, outlined* or *invisible.* Clicking an identifier repeatedly just cycles through these states and also alters the way the identifiers are depicted in the list. These features are very useful in designs for which the pieces are packed in a box, since the box would hide most of the action that is going on inside (e.g. Al Packino, ▷ Appendix A). Also they are very useful for inspecting the interaction of a few pieces and allow comparison between different solutions as the visibility states remain invariant in selecting solutions.

By default the pieces that become separated from the rest gradually fade out during the final move. Sometimes this is unwanted as it may hinder a clear view on what's going on. This can be avoided by unchecking **Fade Out Pieces** on the options window (activated through **Config** on the menu bar).

Chapter 6
Reporting with BurrTools

BurrTools comes with some extra features to assist you in making puzzle solution sheets, either for your personal archives or to be issued with your exchange puzzles and commercially produced puzzles. Currently, these capabilities are very basic and need to be improved in a future update of the program. So, don't expect too much from them right now, but rather consider them to be merely a preview or a teaser to stick to BurrTools.

6.1 Adding Comments

The **Edit Comment** entry on the menu bar opens a new window that allows you to add textual information to the puzzle file. It can be used to append extra information to the puzzle such as the name of the designer, or a 'to do' list for your own designs.

6.2 Exporting Images

The **Export - Images** entry on the menu opens a window that allows you to export a portion of the current puzzle in to (a list of) images (see Figure 6.1). The window has a 3D view on the right and input elements that control what is being created on the left. On the very bottom of these controls you can select what you want to create images of. Depending on what is present in the puzzle, the following things can be exported:

Shape. An image of a single shape is created. You can select which shape with the shape selector below.

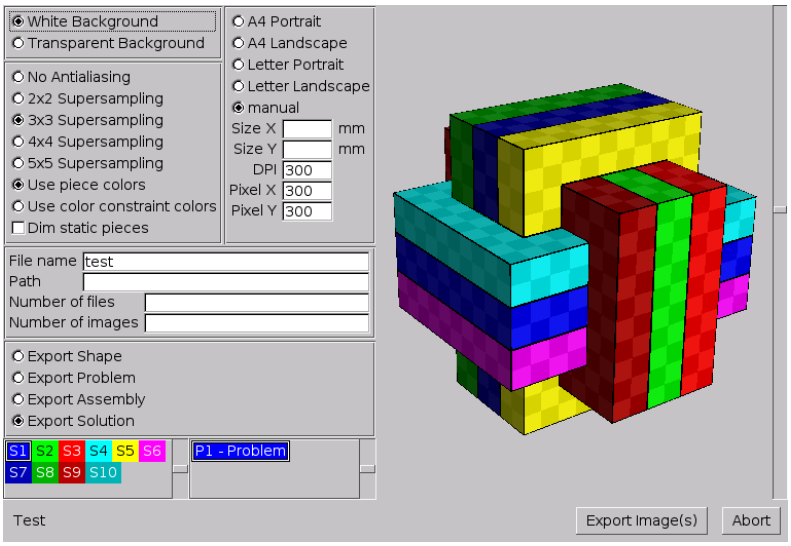


Figure 6.1. The image export window

Problem. An image containing all shapes that are used for a problem is created. Again you will find the problem selector below that is used to select which problem you are going to create images of.

Assembly. An image showing the positions of the pieces in an assembly is created. You can select the problem. Of that problem the first assembly is exported.

Solution. An image containing all steps necessary to disassemble a problem is created. In this case you also select the problem with the selector below. The images will be created for the last solution of that problem.

This is the first thing that you have to select. Naturally only the choices are available to which the puzzle has data. So if you have not run the solver on the current puzzle it is impossible to export solutions or assemblies.

Above these selector you find the file output parameters. First the name and the path to where the images are supposed to be created. If you give no path the images are put into the working directory of the program. The file name is just a prefix, so if you keep 'test' as file name you get files of the form 'test000.png', 'test001.png', ...

Finally you can say how many images you intend to create. BurrTools will try to do so, but might use less. If you only have one assembly to export, only one page can be created.

The Number of images entry is ignored by the software for the time being, it will be used later on.

Above these input elements you find the last section that defines how you want to output, what you output. You can define the quality and some additional parameters that influence how the images look, but not what is to be seen.

In the top left corner you find the definition of the background of the image. You can choose between transparent or white. Transparent is useful if you want to have a background with patterns or want to further edit the images.

Below you find the settings for the oversampling factor. The higher that is the smoother the images will look, but the more memory and calculation time is required.

Below you can select if you want to use the constraint colours for the output or rather the neutral colour of the shapes.

The checkbox Dim static pieces makes BurrTools draw pieces that are not involved in the current move in a lighter colour, so that the actually moving pieces are easier to spot. This, of course, only works when exporting solutions.

Finally there are the parameters for the image size that the program creates. You have 2 possibilities. Either define the pixel size directly, or define the size of the image in millimetres and the DPI printer resolution. If you want to create A4 or letter sized images for printing you can use the predefined sizes.

To position the shapes in the output images you can use the 3D view at the right of the export window. All images exported will use the same settings for angle and zoom as in that 3D view. If the shapes reach above or below the 3D view they will be cut. Left and right is different. The width of the images to generate is not fixed. So the program will make them quite a bit wider to accommodate the horizontal spread of the pieces.

If you have finished with all settings press Export Image(s). You will see a flurry of images in the 3-D view. The program draws the shapes there and grabs the content from the display. This may take a while. First the size of the images is determined then the images are drawn in the required high resolution for the output. The progress can be seen on the left besides the 2 buttons. You will see how many images are finished and how many there are overall.

Hint: If you get unexpected results and broken images try to do nothing while the images are exported. On Linux it is forbidden to change the virtual desktop because then nothing is drawn.

The export is far from what we want it to be, many important features are missing, so you can expect some progress in later versions of BurrTools.

6.3 Exporting to STL

STL, which stands for Standard Triangulation Language or Standard Tesselation Language is a file format used by stereolithography software. STL-Files describe the surface of 3-dimensional objects.

BurrTools can export single shapes into STL files so that 3D printer can quickly fabricate prototypes of them.

The main menu entry **Export - STL** opens the window seen in Figure 6.2. The window has shape selector, a 3-D view of the selected shape and some parameters that control the created shapes.

Filename and **Path** control the name and position of the generated file. **Cube Size** controls the base length of the created cubes. **Bevel** controls the size of the bevel and **Shrink** allows to have a gap between different pieces, so that it is actually possible to assemble them. If the shapes were make to correct sizes they would touch and movement impossible.

The STL-Export does right now only work for cubes. Triangles and spheres are not working. Also the shapes to export must not contain any variable voxels.

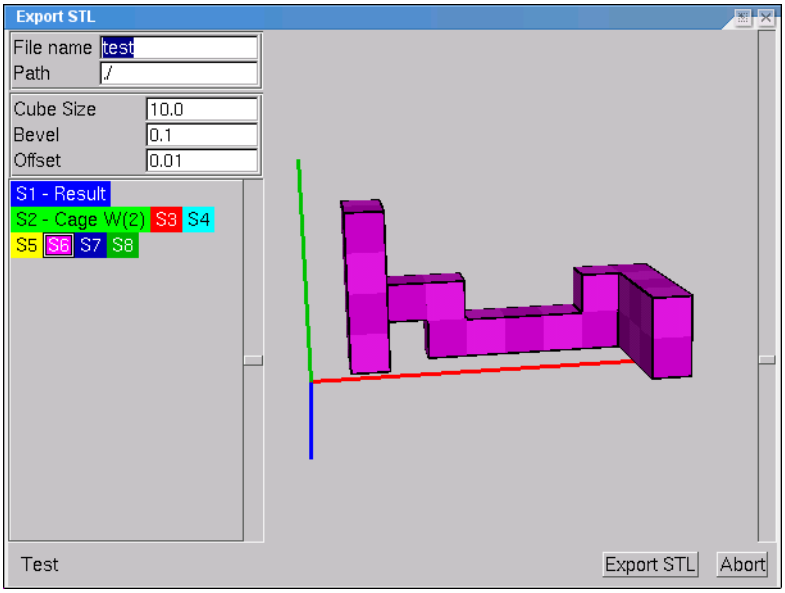


Figure 6.2. The STL-Export window

Chapter 7
Future Plans

So, what are our future plans? There are a lot of things still missing (or in need of improvements) from the current program. A list of things that might be interesting to implement are the following:

- Add some special algorithms that are faster for certain kind of puzzles. The current algorithm is quite good for nearly all puzzles, but it's not *the* fastest.
- Add more colour constraint possibilities, e.g. edge matching, ...
- Add different more space grids, add parameters to some grids (lengths and angles).
- Add rotation checks to the disassembler.
- Add a shape generator: create all piece shapes that fulfil certain rules (shape, colours, union of two shapes, ...)
- Libraries of shapes to import pieces from.
- Add tools for puzzle design (see below).
- Make it possible to divide problems so that they can be solved parallel on several computers and then the solutions are merged back together in one file.
- Improve multi threading so that multi-core CPUs are better used.
- Improve assembler to cope with ranges of piece numbers (e.g. 1-5 of piece x) and doesn't need to place all pieces. So that is is possible to solve piece sets and also to create puzzles by defining a set of pieces and let the program find out which of them results in a nice puzzle.
- Better tool for colourization of a piece. E.g. checkering, but it needs to be more general than just checkering.
- Create a debug window to make it possible to find out why there is no assembly or why an assembly can not be taken apart.
- Speed improvements.
- "Unificator" a tool that makes it easy to check the results of adding color constraints. For example: suppose you have designed a puzzle with one interesting solutions and many uninteresting and you want now, by adding color constrains, make that one solution unique. It can be a labor intensive task to do that. The unificator would help here by quickly showing the resulty of adding color here or making a piece that color, ...

We would be very happy to get contributions from other people. After all there are quite a few people out there that have their own puzzle solving programs, maybe they have some nice additions. There is one important thing to keep in mind: the additions have to run on Linux. So you can not use any proprietary library that is not available for Linux.

7.0.1 Burr Design Tools

The following paragraphs are written as if the features were already implemented, but this is only done so that the text can be copied into the real book without having to rewrite a lot of it.

There are 3 possible design methods implemented in BurrTools

1. BurrGrowing after Dic Sonnevelds ideas

2. Constructing, the natural approach
3. Destructing, the inverse way, take the assembled puzzle and try to assign cubes to one of the pieces

The following sections will describe these methods

7.0.1.1 Burr Constructing

The idea behind Constructing is to create new puzzles out of a set of pieces, try all possibilities and select the best found. To give the designer a great number of possibilities there are loooots of options here beginning with the design of the pieces ending with the method of how to solve the generated puzzle and how to save them.

The basis for the Burr Construction is a normal puzzle file containing some shapes. These shapes are then taken by the constructor and made into many puzzles that are solved.

So lets start with the piece generation. Each piece for the puzzle that needs to be generated may be assembled out of the following possibilities: a fixed piece, a list of pieces, a merger of 2 or more pieces, a piece containing variable cubes. The whole possibilities can be stacked on one another, so you can specify a list of 2 pieces where is piece is the merger of 3 pieces containing variable cubes... . All this can result in many possibilities, so be careful if you want a full analysis this side of eternity. Because of the complexity the program also might encounter the same puzzle several times. It will also be possible to let the program select puzzles out of the definition space by chance instead of doing a full analysis.

So what do the possibilities mean.

fixed piece. a shape containing no variable cubes. This shape is directly used

variable piece. a shape containing n (n>0) variable cubes. All shapes are used that have one of the 2^n possible conditions for the variable cubes are used

list of pieces. the pieces in the list are taken one after the other

merger of n pieces. a new piece is constructed containing the union of both pieces, where the union is set, if one of more of the shapes to merge is set and the others are not set and variable is at least one is variable.

At the end of the process it is possible to define the type of connection that must exists inside the shapes, shapes that do not fulfil this requirement are dropped

All these possibilities may lead to a huge number of shapes, so be careful.

Now it is possible to select the way the puzzle is solved. This includes disassembly (if or if not), also reduction and parameters for reduction can be set

Finally it is possible to select the way the created puzzles are saved.

- All / only Solvable / only uniquely
- keep best with least number of solutions
- keep best with highest disassembly level
- keep best with biggest disassembly tree (most branches on the way out)
- keep best with highest number of not disassembable solutions

Save puzzles with solution(s) or without to save space

The puzzles are all saved into single directory, that must be selected

It would be nice to be able to stop the search process and continue later on, the parameters for the constructor should be saved into the source puzzle file (including the current state)

7.0.1.2 Destruction

Destruction is in some way the inverse process of construction. Here you start with the finished assembly and you assign the outer voxels to certain pieces. Now the search process starts by assigning the not yet assigned cubes to pieces or to voids. All possibilities are tried and the best are kept.

Additionally it is possible to pose certain requirements on the piece shapes. You can say in which way the pieces must be connected (by faces, edges, corners), if the pieces need to be machine makable.

Also it is possible to do the whole process randomly instead of completely

7.0.1.3 Burr Growing

This method has been pioneered by Dic Sonneveld. It is suitable to create extremely high level burrs. The algorithm works by adding cubes to pieces to prevent certain moves and hope that the puzzle will still be disassembable in a different way.

Part II

Advanced User Guide

Chapter 8
The Internals

This chapter explains some of the internals. It is still quite incomplete, probably out of date and might even be wrong...

8.1 The Puzzle File Format

For those people that want to do things that the GUI is not supporting the exact file format of the files used by the GUI and the library may be of interest.

The format is actually a gzip compressed XML-File. The program can read both, compressed and uncompressed files transparently so you don't need to zip them before loading into the program. The GUI always writes compressed files so if you want to change something in them you first need to decompress it.

I wont describe all the elements of the XML-File, it's easier if you enter something similar to what you need in the GUI and look in which way the program saves these information.

8.1.1 Voxel Space

Because this is probably the most complicated part of the format here is a description of how the voxel spaces are saved. The size of the space is saved inside the attributes of the node and the contents of the node is saved in text of the node.

The 3 voxel states are saved with 3 characters:

- '`_`' for empty voxels
- '`#`' for filled voxels
- '`+`' for variable voxels.

Colours can not be attached to empty voxels but to voxels with the other 2 states. Currently colours are just a number (up to 2 digits) that are simply written as a decimal number and are appended to the voxel state. If the colour number is 0 (which is the neutral colour) nothing is appended.

8.2 The Library

The library is available for all people who want to do an analysis that would be too much work to do by hand with the GUI. A bit of C++ programming experience is necessary to handle the task.

There are 4 important classes in the library. The class `voxel_°°°c` handles a 3 dimensional array. Each position inside the array corresponds with one cube inside the piece. The class `puzzle_c` is responsible for the whole puzzle containing a set of pieces and a solution. The classes `assembler_x_c` and `disassembler_x_c` (where `x` is a number which may be available to select different algorithms that do the same task) are responsible to find assemblies and to disassemble the found assemblies. The important aspects of these classes will be explained in the next sections.

8.2.1 Class Voxel

This class contains functions to organise, modify, transform 3-dimensional arrays of cubes. Each entry inside the array contains 2 values:

- The type of voxel (is it empty `VX_EMPTY`, filled `VX_FILLED` or a variable cube `VX_VARIABLE`

- The colour constraint colour. Here values between 0 and 64 are possible. 0 is the neutral colour.

The class provides a set of functions to rotate, translate, mirror, resize and minimise the shape. The `transform` function allows to generate all possible rotations — also including mirroring, if wished. The function `selfSymmetries` calculates which of these transformations result in the same shape. `Connected` finds out the all the cubes in the shape are connected in one big piece (neither the assembler nor the disassembler requests that this is the case).

If all this is not enough then there are functions that return the value of the different cubes inside the shape and also to set the value of the cubes. These functions exists in different versions. One requires the x, y and z coordinate of the cube requested. The other just takes one number. For this function all the cubes are in one long row. This function is efficient to use if all cubes are traversed and an action is done that is independent of the exact position of this cube inside the shape. Finally there is a set of get functions that also work with coordinates outside the box of the shape. These function always return `VX_EMPTY` for cubes outside the bounding box.

Then there is a bounding box that encloses all non empty voxels. This box is used by the selfSymmetry function. It only transforms the part inside of the box and then compares. There are 2 comparison functions: one compares the voxel space one by one the other one compares the space inside the bounding box, so the content may be shifted and still they are considered identical.

8.2.2 Class Puzzle

This class contains all the information of the puzzle including the shapes, the result shape and piece shapes and number, the colour constraints, the solutions, the grouping information and some statistics. This class contains all the information that gets saved in hard disc.

The class contains a huge amount of functions that allow you to set and get the contained information.

8.2.3 Class assembler

As already explained this class tries to find assemblies for a puzzle. It uses the dancing link algorithm explained later.

The caller is informed about found solution via a class that the caller has to provide. This class contains a function. This function is called for each found assembly with the found solution as parameter.

The caller can then do whatever he pleases. He can just count the number of solutions by increasing a counter. He can save the found solutions. He can analyse, if the found solution is disassembable. If the caller is not interested in the solution he has to delete it.

8.2.4 Class Disassembler

The disassembler tries to find out if an assembly can be taken apart. And if it can be taken apart it will return a shortest disassembly sequence. The class contains some datastructures to make it possible to quickly check multiple assemblies of the *same* problem. So it is possible to chreate one instance of this class and disassemble a whole set of puzzles and then destroy it.

8.2.5 Class Assembly

This class contains an assembly of a puzzle. The assembly is always connected to a specific problem of a puzzle because it takes reference to the piece numbers defined in the problem and also to the shapes of the pieces defined within the puzzle.

The assembly itself contains just a list of positions and transformations. What shape is behind that must be asked from the puzzle class

Assemblies can be transformed. This changes to placement and transformation of all the included pieces so that the resulting piece arrangement is rotated.

Assemblies can also be compared. This is required for the rotation avoiding technique describes below for the assembler.

8.2.6 Class Disassembly

This class contains all the information to completely (or with piece grouping not completely) disassemble the puzzle. It contains a tree. On each branch of the tree the puzzle separates into 2 parts. If one part can not be further assembled (e.g only one piece is in that part or the grouping makes is not necessary to disassemble that part) the pointer to the subtree is `NULL`. Each node of the tree contains a list of piecepositions that are the steps to take the problem apart.

8.2.7 Example

A very simple example can be found within the source code of the project. Check the burrTxt sources. They just check a few command line options, load the puzzle and then solve it, no fuzz with user interface, multi threaded application, ...

8.3 The Algorithms

There are only two algorithms of interest inside this program. One is the assembly algorithm. This one is based on the "Dancing Link" algorithm from D.E.Knuth. I needed to update the algorithm in 2 ways:

1. We require cubes that *may be filled* as well as cubes that *must be filled*. The original algorithm only provides the 2nd type of cubes.
2. We need to do something about multiple identical pieces. The original algorithm will find $\prod_{s \in \text{shapes}} \text{num}(s)!$ as many solutions as there really are.

The 2nd interesting algorithm is the disassembler. This is mainly a breadth first tree search over all possible placements of the pieces.

8.3.1 Assembly

As already said this algorithm is based on the Dancing link algorithm. This algorithm is mainly a very efficient and elegant backtracking method that stops much more early than many other algorithms. It stops when is finds that a piece can not be placed any more. It stops when it finds that a cube of the solution shape can not be filled any more. These recursion stops don't need to be implemented separately, but they are part of the algorithm. But bevore we go on describing the details, there is one mayor problem that needs to be solved: avoid finding solutions multiple times.

8.3.1.1 How to Avoid Finding Multiple Assemblies

Now this is a complicated problem. There is the naïve approach which would be to save all found assemblies and check new found assemblies against this list. This has major problems. You need to save all assemblies and there can be many. You need to check against all those save assemblies and that can get slow. If you want to make a break and later on continue you need to save all those solutions on harddisc and load them again. An of course the worst problem is that you waste a lot of time. If it just would be possible to not find those solutions in the first place.

To solve this problem let us first analyze what kind of double solutions exist

Identical assemblies. These are solutions that do look completely identical (they are not even rotated). There are 2 possible reasons for this to happen:

1. Two or more identical pieces that are exchanged
2. One piece has symmetries and the (invisible) difference between the 2 found solutions is that this piece is rotated
3. A bug in the code that makes the program find *really* identical assemblies. Lets assume that this is a rare event.

Rotated assemblies. These are solutions that are identical but need to be rotated first to find that out.

The first kind of assemblies can be avoided relatively easily by removing rotations from pieces that result in the same piece. And by being careful with identical pieces and avoid finding the permutations of these pieces. With these precausions it can be assured that *no* identical looking assemblies are found.

The second kind is very hard. The recursive part of the program will find them. It is possible to avoid finding a few of the rotations and in some puzzles is even possible to avoid finding any of them but there are puzzles where the program *will* find some or even all possible rotation, so a solution needs to be found that can detect rotations when they are found.

But first let's see how we can avoid as many of the possible rotations as possible. This is done by selecting one piece and dropping a few of the rotations that are possible with this piece. As this piece can not only be inside the solution in certain positions all solutions that would require that piece to be rotated will not be found. If we can be sure that all solutions that are dropped also exist as a rotation inside the solutions that we find we are lucky. But which piece to select? And what rotations to drop?

To find out which piece it helps to think of the perfect piece. Lets assume our target is a cube and it has only one solution. A cube has 24 symmetries so we would normally find 24 solutions (maybe even more, due to mirror solutions, but let's forget about this for a while). With each rotation that we drop from our selected piece one of the possible rotations for the solutions wont be found until we have only one possible rotation left for the selected piece and so we find only that solution where this piece is in that left over rotation. All other rotations would require the piece to be in another rotation, which is not in our list to try. But this only works, if the piece really has 24 differen rotations from which we can drop 23. If the piece is symmetric in one way or another it will not have that many different rotations as a few of them will result in the same piece and thus can not be considered. So the best choice is always a piece with no symmetries. What to do if there is none such piece? Select one has has the least overlap with the symmetries of the result shape.

Before we make clear what that means we have to see which rotations need to be dropped. We need to drop those rotations that might result in a rotated solution. A rotated solution is one that has the same exterior appearance. So the possible rotations result from the shape of the result. If all these rotations do exist in the selected piece we can supress the rotations from the solutions by dropping them.

And now back to the clean and general solution. Here Bill Cutler came to my help. He told me what he did and that is something very ingenious.

The first thing to do it to be able to compare two assemblies that are the same but one is a rotation of the other and be able to say assembly a_1 is smaller or larger or equal to assembly a_2. This comparison can be implemented by comparing piece positions and transformations. It can be completely arbitrary. It just must be assured that the rotation suppression with the pivot piece does not remove the one transformation that is the smallest when compared with the comparison.

Now the following is done for each assembly found. At first all rotions of this assembly are generated that result in the same shape for the assembled shape. These assemblies are compared with the found one. If there is one that is smaller than the found one drop the found assembly and go on searching. If the found assembly is the smallest one do whatever needs to be done with it.

There are 2 left open the question. What to do if the found assembly is the smallest but there is another assembly just as small? And how can be assured that the rotation selected by the comparions function is not removed by the rotation avoiding method.

First to the first question. When does this happen? This happens then when solution itself (not only the shape of the result but the also the construction) has some symmetry. That means that there are 2 indetical looking solutions that differ in exchanged pieces ore a rotation of a piece that does result in an identical looking piece. This kind of identical solution has already been successfully avoided, so there is no need to take special precautions, that case is ignored. If the found assembly is one of the smallest it is taken, if there is one ore more smaller assembly, it is dropped.

Now on to the 2nd problem. Here we need to make sure that the rotation avoiding method knowns about the comparison function and makes sure that the smallest of the assemblies is kept. Here is one possibility:

If the comparison function looks like this:

```
for (p = 0 up to number of pieces the assembly) {
```

```
  if (rotation of piece p in assembly 1 < rotation of piece p in assembly 2)
    return assembly 1 is smaller
  elseif (rotation of piece p in assembly 1 > rotation of piece p in assembly 2)
    return assembly 2 is smaller
  elseif (pos x of piece p in assembly 1 < pos x of piece p in assembly2)
    return assembly 1 is smaller
  and so on
}
```

For this function an assembly with a piece 1 with a smaller rotation number is always smaller that one with a bigger rotation number.

So if we chose a rotation avoiding technique that always selects piece one as pivod piece and always removed the bigger rotation number, we should be on the save side.

8.3.1.2 The Dancing Link Algorithm

I will describe the only the basics for the original dancing link algorithm. For further information read the document available on Mr. Knuths web page (`http://www-cs-faculty.stanford.edu/~knuth/musings.html`).

The algorithm represents the puzzle as a matrix. In this matrix the first columns represent the pieces and the last columns represent one voxel of the result shape each.

Each line of the matrix corresponds to one possible placement of one piece inside the result. The column of the piece and the columns the represent the places inside the solution that the piece occupies with the placement are 1 inside the matrix. All the other cells are 0.

The search itself runs on this matrix. It searches for a set so that all the lines in this set taken together contain exactly one 1 in each column. This means that each piece must be used and each cube in the result must be filled.

The algorithm does 2 operations on the matrix:

1. Cover column n and uncover column n. This means that the column is removed from the matrix and no longer taken into account for the search. When a column is removed all the rows that contain a 1 in this column will also be removed.

2. Cover and uncover row n. This means that we select this row for the set of rows that we search. The row covering also removes and re-includes all the columns that contain a 1 in this row. On these columns operation 1 is performed.

The 2nd operation can be interpreted as. Taking one piece and putting it inside the result at one possible place. This results in the fact that a few cubes of the result don't need to be observed any longer and all placements of all other pieces that collide with this placement don't need to be checked further.

The cover and uncover operations are the inversion of one another. If we first cover something and then uncover it again the matrix is in exactly the same state.

The algorithm is now recursively trying all possibilities. It selects one column and then tries covers all rows that contain a 1 in this columns and then calls itself.

It finished when there are either no more columns left. Then we have found a solution or there is one column with no rows. Then we have found a dead end and backtrack.

This algorithm is per se not dependent on square cubes it is not dependent on any shape. You only need to transfer your puzzle into the matrix. Even William Waites[8.1] puzzles should be possible. But as the square and cubes are most common I have for now only implemented this transformation.

Now to the changes that I have done to this basic algorithm. There is first the matter with the 2 types of cubes. This is easily solved by removing the columns of the cubes that *may be filled* from the list of columns that need to be covered. They are still in the matrix, they just don't *need* to be covered to find a solution.

8.1. see `www.puzzlemist.com`

The 2nd problem was much harder. How handle multiple identical pieces? The solution that I finally implemented is to enforce an order. All pieces get a number and all the placements get a number. If we now have 2 identical pieces a and b with $a < b$ I force that the placement of a, $p(a)$ is also smaller than the placement of b so $p(a) < p(b)$. This is done by always placing all identical pieces in one go. The moment the algorithm decides to place one of the pieces that occur multiple times it will also place all the others and always check that these have larger placement numbers.

8.3.2 Disassembly

The disassembly algorithm is a breadth first tree search. In this tree every node represents one possible relative position of the pieces. To find out what can be moved in this node the algorithm Bill Cutler used for his 6 Piece Burr analysis is used. His algorithm anaylzes for 2 pieces how far the first piece piece can be moved in the positive direction of each of the 3 axis if the other piece is fixed. This results in 3 matrixes each square with as many rows and columns as there are pieces. The values for negative directions can be taken from transposed matrixes. To make these matrixes useful they need to contain not pairwise information but for the whole state. To get this information the following property is used:

> If piece A can be moved x units when B is fixed and piece C can be moved y units whan piece C is fixed then piece C can not be moved more than x+y units when B is fixed.

With this property the 3 matrixes are treated again and again until all values have reached a stable value. The resulting values tell you exactly how far each piece can be moved when some other pieces are fixed.

Now all possible new states are generated with the aid of these calculated values.

This worked nice but it has been quite slow. Slower than PuzzleSolver3D at least. So I started to optimize. The slowest part has been te pairwise analysis of all piece pairs. Initially I implemented more and more complicated schemes that were supposed to speed up thing. But the code got more and more complicated and due to the usage of preprocessor macors utterly undebuggable. And it was still slower than PuzzleSolver3D.

Finally I came up with a new scheme that solved the speed issues: a cache. This cache contains the values calculated for the movement possibilities of 2 pieces. Once they are calculated they are put into the cache and used from there later on. The cache contains the 3 calculated values. The key is calculated from the piece numbers, their relative positions and their transformations. The incorporation of the transformations made it possible to used the cache over the whole process of a puzzle analysis and not to restart it for each assembly. This has a mayor impact: the number of cache hits is for some puzzles way over 90%.

This cache also has another nice property. It is possible to remove information for a certain piece from it. This comes in handy when burrgrowing is used, as the information for changed pieces can be removed from the cache but the rest is still intact and useful information.

8.4 Adding to the Library

There is currently one useful thing besides the normal improvements that might be added to the library: Other puzzle types. The assembly algorithms is so abstract that it can cope with many different types of assembly puzzles, as long as they have some kind of pattern. Currently the assembler only supports puzzles made out of cubes but there is nothing that prevents solving puzzle where the base unit is a hexagon. Of course the disassembler can not do work with this kind of puzzles.

To add other geometries the assembler is split into 2 parts. The dancing link algorithm and the algorithm that prepares the matrix for the dancing link algorithm. This preparation part is called the front end.

Part III

Appendices

Appendix A
Examples

BurrTools comes with some examples that illustrate the capabilities and functions of the program. We'd like to thank the designers for allowing us to include their designs in the BurrTools package.

A.1 Al Packino

Design. Ronald Kint-Bruynseels, 2003, Belgium.

File. `AlPackino.xmpuzzle`

Remarks. This puzzle shows how to properly make packing puzzles. You always should include the box as a piece so that the program can also check if the pieces can be moved into or out of the box. You can also see how to handle multipieces. When looking at the solution it is useful to display the box as a wire frame. This can be done by clicking at the blue rectangle at the lower end of the tools. The rectangle with the text "S1-Box" in it.

A.2 Ball Room

Design. Stewart Coffin, #197-A, USA

File. `BallRoom.xmpuzzle`

Remarks. This puzzle shows off the sphere gridspace. It also demonstrates that is is possible and useful to include more than one problem within one file.

A.3 Bermuda

Design. Bill Cutler, 1992, USA

File. `Bermuda.xmpuzzle`

Remarks. This puzzle demonstrates the triangle space grid. You can see that you can stack many layers on top of each other.

A.4 MINE's CUBE in CAGE

Design. Mineyuki Uyematsu, 2002, Japan.

File. `CubeInCage.xmpuzzle`

Remarks. This file contains MINE's CUBE in CAGE 333, cube g. This puzzle demonstrates how to use the grouping capabilities. The puzzle contains 3 interlocked pieces that construct a cage. These pieces move but can not be taken apart. It needs to be told to the program that this is intentional. So here you have an example of how to do that.

A.5 Dracula's Dental Disaster

Design. Ronald Kint-Bruynseels, 2003, Belgium.

File. `DraculasDentalDisaster.xmpuzzle`

Remarks. This puzzle demonstrates the use of colour contraints. Halve of the result must be red and the other halve black. You can see the colours if you enable the checkbox in the status line at the bottom right.

A.6 Level 98 Burr 'The Pelican'

Design. Dic Sonneveld, 2000, The Netherlands.

File. `PelikanBurr.xmpuzzle`

Remarks. This is a *very* high level burr. It takes 98 moves to get the first piece out of the box. This is just a demonstration of what is possible.