Amaze

Author: chaered@gmail.com

Version: 1.25

Date: 2015-05-03

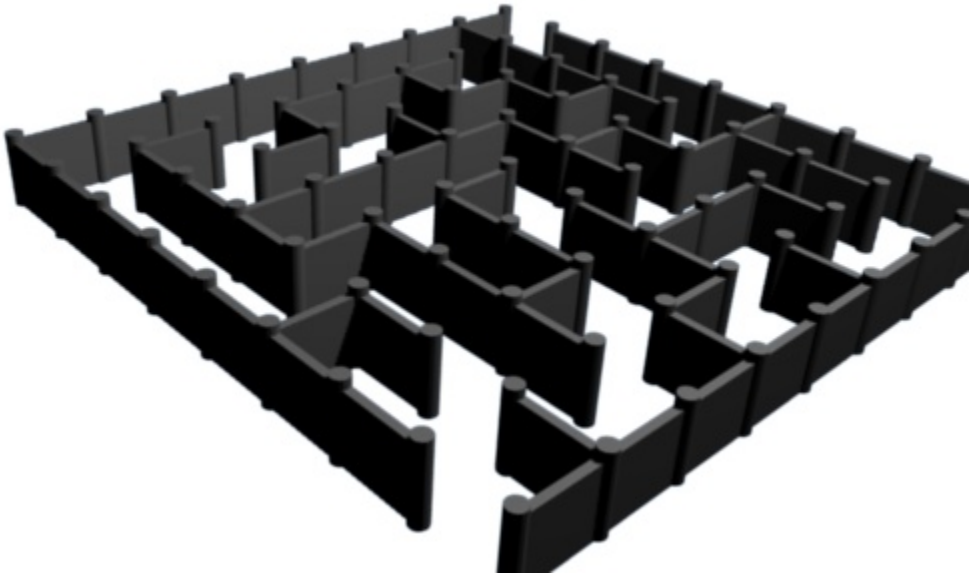


Table of Contents

1. Introduction

This document contains the development documentation for *amaze*, an open-source program to generate pictures of simple mazes. It is likely to change much over time, as the program evolves.

The current Amaze version is 1.2-1.

1.1. Name

The application itself was originally called “Amaze”. Since there was already a project my that name, the SourceForge project is called “qtamaze”. This name seemed to be globally unique when I picked it, at least it didn't have other hits when I googled it. There is now another project, created at *http://code.google.com/p/qtamaze/* three months later, an MP3 downloader, that bears no relation to this maze program. There are a bunch more applications that have plain “amaze” as their name; see the SourceForge qtamaze project page for some links.

2. Project

The *amaze* project is being developed by me, as user *tinco* of SourceForge.net, and hosted under *http:://qtamaze.sourceforge.net/*.

Version 1.1-1 through 1.1-6 were all written over the same period in the first quarter of 2010. Then the project languished for about a year, until February 2011. Then I updated the libraries to Ubuntu 10.10 (Maverick), and did more extensive code updates and extensions, starting with 1.1-7, which featured the new “cuts” option and improved enter/leave point setting. This continued up to 1.1-22 with incremental improvements in maze quality, the new octagon tile shape, maze shape masking, mazegrams, and more localization.

After early 2011, the project then languished, until I picked it up again 2015 Q2, four years later. The target platforms have moved on: Win XP is more or less dead, Win 7 the new minimum; Ubuntu begat Mint and others, Qt moved to Qt5; the online help was based on a hosted wiki that SourceForge has dropped in the meantime; the Blender export uses an API that is not longer supported. The initial goal is a refresh of the code, to build for current platforms, without significant changes in functionality. To signal the shift, the versions will jump to 1.2-*x*.

The effort invested in this project has been mainly about the Qt framework, installers, localization, and export formats. The initial version of the actual maze generation code, at the heart of the application, represents only a few hours of work. The MSI installer, in contrast, cost a least a man-week. I think this is about typical for any program development (outside of throw-away personal code) where you climb the first-time learning curve on some of the tools; packaging, documentation, localization, version upgrades, porting, etc. are >90% of the effort. The difference with professional development is mainly in freedom from specifications, formal design, status reports, and support and migration issues.

2.1. Feedback on SF

Up to 1.1-6, the review section on the *qtamaze* project page showed 18 reviews, all positive, with just one containing text (“great!”). Nobody posted anything in the forums or tracker except myself so far.

2.2. Feedback in reviews

Reviews are a great source of feedback, both the initial review and the comments on it. I've seen about half a dozen review so far, some for the 2010 version, some more current. The focus of Amaze is on first-time, casual users We can classify reviewers as probably being first-time but somewhat less casual; this is important, because if a reviewer misses something or is confused by the UI, then the majority of the (more casual) other users will be even more likely to encounter the same difficulties.

Some very useful feedback came from *snapfiles.com* (thanks!) for their review of version 1.19, as a short series of recommendations on the “SnapFiles Developer Admin Center”:

1. Should offer size presets based on common paper sizes (for printing).
2. Limited documentation available.
3. Size of maze is limited to the size of the screen.
4. Option to create or save color schemes for the maze design.
5. Option to save a maze project for future modifications.
6. Print preview.

2.2.1. Paperwork

Some of these points reflect the fact that, although we specifically bill the program as geared towards printing simple mazes, the program is not actually all that well-equipped for a paper-oriented workflow. The original intent was to keep the learning curve very low by not requiring the user to learn anything new about sizes and scaling: the screen is the content, the whole content, and nothing but the content. The maze on screen maps pixel-to-pixel to what is exported or printed. The paper coming out of printer is a whole separate world, and we need a way to bridge that gap without losing the simplicity of the interface.

2.2.2. Saving

As we get more tweaks and options for the user, it is possible for the user to invest more time into setting things up, and at some point it becomes a bother to not be able to reuse that effort, or to share it with someone else, or take it to a new environment. Also, there is now no way to go back to a maze later, and e.g. tweak the colors. So yes, we need to start thinking about the best way to save, load and export that information. As of 1.1-9, the save state contains the following (sample from Ubuntu):

```
[General]
version=1.1

[main]
splash=true
size=@Size(553 394)
position=@Point(31 25)
```

```
[maze]
wide=20
high=20
path=1
wall=1
stay=1
show=true
shape=2
color\back=@Variant(\0\0\0\x43\x2\xff\xff\x88\xb8\x3\x3\xff\xff\0\0)
color\wall=@Variant(\0\0\0\x43\x2\xff\xff\x13\x88\xff\xffUU\0\0)
color\path=@Variant(\0\0\0\x43\x1\xff\xff\xff\xff\0\0\0\0\0\0)
cuts=109
moat=1
guide=false
mask=
```

Most of this should be pretty straightforward. Items that are not currently saved, but that really should be, include:

- The enter and leave points, normalized to their 0-100 range. See option –**xenter** and clicking on the maze.
- The “imath” formula, if any. See option –**mask [imath]**.
- The directories chosen in the export dialogs, or in the File>OpenOutline menu item.
- The creep speed (see option **–creep**, and the View>FasterCreep menu item).
- The message text font attributes.

Beyond saving information that reflects the current, complete collection of settings and selections, we can consider saving *parts* of that information, e.g. color schemes like SnapFiles suggested. See also ticket 3190883 (“Add color-combo picker, like *agave*(1) on Ubuntu”). The hard part may be to decide which bits of state to group as a savable entity. Maybe we should start with the color schemes, to get some experience with how well we can handle those. One consequence of saving and loading state is compatibility: once you let users invest effort into creating projects, make sure to stay backward compatible in future versions.

2.2.3. Documentation

The text shown for the Help>Help menu item is very summary, and on Windows (where most of the downloads end up) the *amaze*(1) man-page is not installed. We do have an online version of the man-page (linked through *qtamaze.sf.net*) and some screenshot in the wiki, plus this development document, but no real first-time user document. Maybe put a video tutorial on *vimeo.com*?

3. Versioning

The sources at SourceForge are kept under CVS, and (with the exception of binary files) have a "$Revision" keyword expanded to their individual version. We're not using any branches at this point, so all file versions should be of the form **1.***x*.

We keep track of a tripartite overall product version number *major.minor-release (e.g "1.2-3")*, used to identify the downloadable binary installations. It is maintained on the last line of two files: *VERSION* and *RESEQNO*. All the places where the version numbers are used are derived from this. To change the numbers, update those two files, then run *amaze/tools/upversion.sh*, then do “cvs commit”.

The uploaded binaries are named e.g. *amaze-1.2-3.msi*, but all releases for a version are also saved under CVS in *ububin* and *winbin* using the same name without the release number, e.g. *amaze-1.2.msi*.

Uploading the web pages and binaries to SourceForge is still a manual process, cf. *web/README*.

4. Features

The focus of the application itself is on first-time, casual users that want to make a few mazes with minimal effort, or play around with it a bit. The feature set is mostly geared towards a low learning curve and getting the mazes into a printable or otherwise usable form with the minimum number of steps. One thing that makes life easier for the implementors is that everybody already knows what a maze is, so we don't need to explain the concepts to the users.

4.1. Image masking

The use of masking (a.k.a outline) images is pretty intuitive. What is still a hindrance is the inability of *amaze* to accept an image URL and download by itself. Also, we now require a perfectly monochrome “transparent” part; maybe we need an alternative, looser filter.

4.2. Procedural masking

In 1.1-17 we started adding procedural masking, with the “-math” option. The difference with image masking is that we use a formula instead of a scaled image. See *Canvas::maskWithImath*(). The user provides a boolean expression, taking an (*x*,*y*) input, which gets evaluated for all non-rim cells, with coordinates mapped to a normalized [-1,1] range; when false the cell is hidden by the mask; when true, is remains visible. The expression syntax is very ad-hack now and still evolving, and there is no GUI access to the formula. Obviously this is not so much for the casual user, but it is fun to put in; this is more for myself.

The 1.1-17 syntax follows below. All names and operators are 1 character, except numeric constants; spaces are ignored (outside the constants). All values are floating-point or boolean; the top-level expression must be boolean. The expression is purely functional; no assignment or binding. Operators are dyadic infix or monadic prefix. No precedence, except that all monadic ops take precedence of dyadic; all evaluation is left-to-right for dyadics. Monadic ops are “-” (means “not” for boolean), “+” (absolute value), “_” (floor), “^” (ceiling), “/” (reciprocal), “S” “C” “T” “L” “E” (sin cos tan log exp). Dyadic ops are “-”, “+”, “*”, “/”, “_” (min), “^” (max), “<”, “>”, “=”, “#” (unequal), “&” (boolean-and), “|” (boolean-or). Numeric constants are decimal digits, optional dot and more decimals, and optional factor “^” (degrees to radians) or “%” (divide by 100). Variables are just “x” and “y”. Examples:

- `(x*x)+(y*y)<1` – makes a circle shape.
- `x<y` – makes a triangle

- `y-0.2<(90%*C(3*x))` – looks like underwear

I'd like extend this a bit before offering it through the GUI. In the implementation, it should change from re-parsing the formula for each call to parsing once and keeping it in-care as a syntax tree. Here's the proposed extended syntax.

- Names: single lower-case letter, or single upper-case letter followed by sequence of lower-case. Examples: “`x`”, “`Sin`”, “`Ln`”.
- Operators: add some multi-char ones (“`<=`”, “`>=`”), add C-style conditional (“`a?b:c`”).
- Add comparison chains, in same direction: “`a<b<c`”.
- Add precedence levels at least for the dyadic operators: highest power/log, then mul/div, then add/sub, then comparison, and finally boolean operators.
- Allow Unicode math characters: “¼” (quarter, alternative to `“(1/4)”`), “½”, “¬” (boolean-not), “º” (degree, instead of “^”), “²”, “√”, “≠”, etc.
- Allow elision of “*”, juxtaposition to mean multiplication in most cases, and power in constant as 2nd operand, so “`2x`” = “`2*x`”, and “`x2`” = “`x**2`”, and “`3xy2`” = “`2*x*(y**2)`”.
- Add the rest of the usual math functions: atan, sinh, etc.; also, some random number generation, and a way to avoid repeating subsequences.
- Add some variable that the user can control through a slider.

4.3. Printing

Although the program advertises itself as being meant to print mazes, it currently (as of release 1.1-22) really just has a “Print” command to dump the maze (scaled to page size) to a printer device, but does not have any features to help with laying out the maze on the page, or decorating the page otherwise. Although complex lay-out and decoration is best done by exporting the maze as an image and then preparing a page in a dedicated DTP program, basic printing from within Amaze should at least allow some control over the scaling and positioning of the maze on the page. Also, until we have decent SVG output, exported images on a page can show ugly upscaling effects because the rasterized exported maze images are drawn to screen resolution.

5. Implementation

The implementation is split between three major classes:

- *CellGrid* represents a single maze and its solution.
- *Canvas* can paint a maze from the information in *CellGrid*.
- *Amaze* is the main GUI class, and ties together a *Canvas* with controls to manipulate it.

When anything changes that would change the display of the maze, *Canvas* is asked to repaint it. It currently does this without any buffering or caching. When any maze content parameter changes (size, tile shape, cuts, etc.), the *CellGrid* is asked to generate a new maze. The maze generation entry point is

CellGrid::generate(). All tile shapes share the same basic code for generation, with some tweaks. So for triangle tiles *generate()* calls *generateTriangleMaze()*, which carries out the rim cell adjustments and enter/exit point adjustments specific to triangle mazes, and which then calls the shared *walk()* function to build maze walls. The *walk()* algorithm is the classic maze build method:

- Mark all cells as unvisited.
- Build walls between every cell.
- Start on the entry point cell.
- Mark this cell as visited. If it is the exit point cell, mark it as part of the solution path. Pick each neighboring cell in a random order. If we can visit it (not visited yet), break down the wall, go to that cell, and recursively repeat this rule. If, after that, the neighbor turns out to be part of the solution, mark this cell as part of the solution path too.

There are a few modifications when the tile shape is not a square, to ensure there are no walls in the middle of a tile, and that paths are fully visible. We can vary the algorithm a bit by tweaking the order in which neighbors are visited; increasing the “stay” value favors continuing a path in a direction over a purely random continuation. The result of this algorithm is a valid maze, but not a very good one by some measures. For one thing, the solution path tends to be subjectively too long; for the other, the maze will have no loops, but be purely a folded tree structure. We apply two improvements: path loop reduction, and cross-cuts. There are optional, and applied to the generated maze in *generate()* as a last step.

Loop reduction is done in *reducePath()*. If two neighboring cells are both on the solution path, and not already consecutive, then the whole section of the path between them is a loop then we can cut out to shorten the path. We breach the wall between the two cells, and build a new one between the first and its original follower on the path. This procedure typically reduces a path by 60-90% in length.

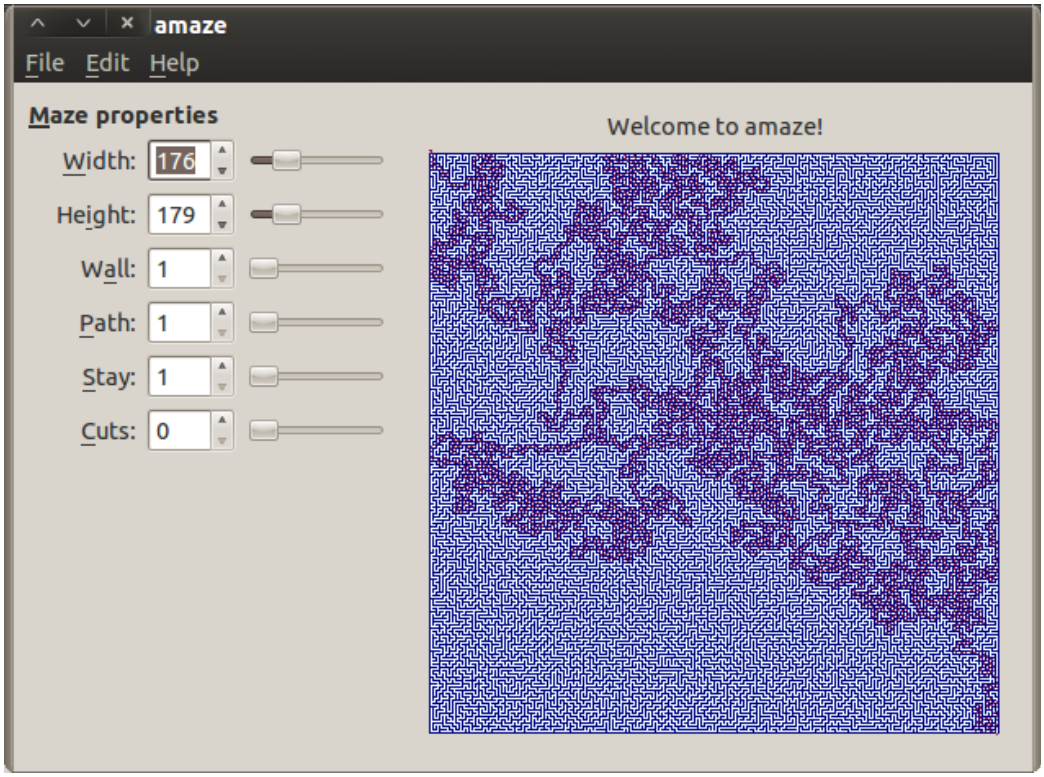
Cross-cuts, made by *crossCut()*, breach walls between cells not on the solution path, without introducing new possible solution paths in the maze, and without creating very small isolated wall sections. This involves dividing the maze into non-connected zones, cutting only between cells in the same zone, and walking the wall structures on each side of a potential cut to see if the result would still be complex enough (using some heuristics) to not look too fragmented. The resulting maze leads to more wandering around, and looks more interesting.

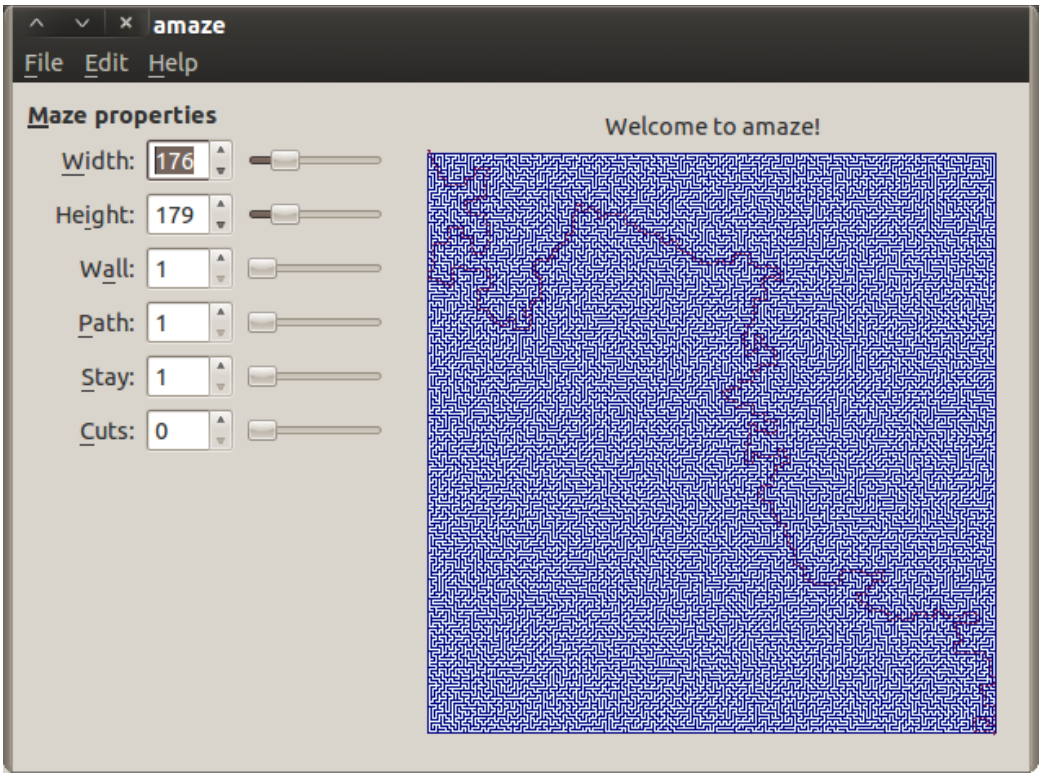
5.1. Path reduction details

To reduce the path, we iterate over the cells of the solution path in order. When we are at the *n*th point in the path, we remember the previous (*n*-1)th path. From the *n*th point, using its (*x*,*y*) coordinates, we look at the neighboring cells. If a neighbor is valid, and has the “path” flag set (marking it as part of the solution, and it is not the previous point, then we can assume it is part of the remainder of the path. It cannot be part of the preceding path, because then this algorithm would have already have made a shortcut from that neighbor to this cell, and it would have to be the immediate predecessor in the path, which, as we have just checked, it is not, QED. We look forward along the remainder of the path points to see which one matches the neighbor coordinates. Since it has been marked as on the path, it will be found as the *m*th. The length *m*-*n* is the length of the sub-path to eliminate., If *m*-*n*=1 then *m* is the next step on the path, hence 0 steps to eliminate, and we ignore it.

The algorithm is applied to a “perfect” maze, i.e. a maze in which there is exactly one path between any pair of cells. The result will still be a perfect maze, as we will see. When we have found a sub-path to eliminate, we breach the wall between the *n* and *m* point. We know there will be a wall, because we already know *m* is not the previous or next point on the path, and in a perfect maze there cannot we both a direct path between the two as well as the sub-path we found. The breach would introduce a second path, so we build a new wall between *n* and *n*+1. This will not leave *n*+1 through *m*-1 isolated, since we know they are still connected (uniquely) to *m*, and *m* is connected (directly now) to *n*. As a minor complication, the tile shape restricts where we can place walls, so we may actually build the wall after *n*+1 or *n*+2; the same proof of connection remains true. We then visit all points *n*+1 through *m*-1 and reset the corresponding cells' “path” flag, so they are no longer marked as on the path; then, we shorten the list of path points by moving down points *m* and higher to *n*+1. Finally, we continue the iteration.

Note this algorithm works well for square and triangle tile shapes, because each tile corresponds directly to a single cell. For hexagon and octagon mazes, we do not make all visually possible reductions, because a path segment passing horizontally through a row of tiles may be 2 or 3 cell rows vertically removed from a similar horizontal line of cells on the path. Here are two screen shots, with the “before” and “after” pictures for this algorithm on the same maze:





5.2. Cross-cuts details

The cross-cut algorithm takes as input a perfect maze (strictly one path between any pair of cells), and produces a path that may contain multiple paths between two cells, as long as neither of those cells lies on the solution path. The reason for applying it is that it make mazes more interesting to solve than plain perfect mazes, since the presence of loops between cells may defeat some common search strategies. The algorithm is based on the observation that each cell off the path immediately connected to one on the path is a unique gate between the path and all other cells connected to that gate; this is a consequence of having a perfect maze. Hence, creating any extra connections involving only cells connected to the gate will not create new connections between cells on the path, because those cells are only connected to other non-path cells through their respective gates. The cells connected to a gate form a "zone", and zones are by definition not connected other than through the solution path, because the gates are already connected through the path, and by definition there cannot be multiple connections in a perfect maze.

The algorithm runs in two phases: zone assignment (*CellGrid::zone*()), and cut attempts. Each cell in the grit has a "zone" bitfield (*cell_t.zone*). This stores a value of 0 (no zone known), 1 (exempt), 2 (being processed), or 3 and higher (assigned zone). For zone assignment, we iterate over the entire grid. When we hit an unzoned cell that is not on the path (*cell_t.path*=false), we flood-fill from that cell, temporarily assigning zone=2 to bound the flood, collecting all connected cells that have path=false, and keeping score of the set of zones assigned to non-connected neighbors. Then, we pick a zone that no neighbor is using, and set all the collected cells to that zone. If a zone is too small (threshold now set to 8) and hence unlikely to lead to good cuts, or there is no zone code left to use, we assign zone=1, which is exempt from cuts.

Next, we make a number of attempts at making up to *n* cuts, where *n* is set by the user with the "cuts" slider. For each try, we pick a random cell and direction; if the neighbor in that direction is in the same zone, the zone is valid (3 or up), and there is still a wall between them, then we measure the "niceness" (see *CellGrid::niceCut*()) of the two wall complexes emanating from the two sides of the prospective wall breach, using a heuristic formula based on total connected walls, minimum continuous segments, and number of bends. If the result is nice enough, we remove the wall. The purpose of the niceness criterion is to avoid creating tiny, "floating" wall segments that are too obviously off the path.

Note the niceness code is recursive, but it only dives deep enough to measure that the required minimum has been met, hence there is no potential stack explosion. The other code is non-recursive and uses memory and time at most O(*N*) = linear with the number of cells. One possible optimization for predictability with more cuts in smaller mazes would be to pre-compute a list of cells and directions with valid zones, excluding only the niceness computation, and picking from them at random, instead of picking entirely random cells to try. One critique for the result of the algorithm is that it favors cuts directly between the border wall and the inside, as the border is always "nice". A critique of the implementation could be that the niceness formula is essentially arbitrary, and has not been subjected to any systematic aesthetic testing.

5.3. Factoring

Since a lot of the reactions and working of the program depend on the tile shape (triangle, square, etc.), it may seem logical to split the *Canvas* and *CellGrid* classes into a *TriangleCanvas*, *SquareCanvas*, etc. However, they share almost 100% of their configuration settings and other state, and the code is very similar with small differences embedded in the logic. The user can switch shapes and expect all other settings to remain unchanged. Splitting off the state would either lead to duplicate data and a lot of resynchronization infrastructure, or factoring out all the settings into a separate configuration class, and leaving the per-shape classes almost stateless. Retaining all the shared logic would be hard because many inner-loop differences should not involve call-outs between classes for reasons of speed, so we would have code duplication, which in turn would be bad for maintainability. The compromise so far has been to split off some functions that are very shape dependent (e.g. *paintTriangleMaze*, *paintSquareMaze*) and use switches on shape elsewhere. If we ever add support for mazes not based on a shared underlying cell grid structure (not currently planned), we might have to revisit the whole factoring approach.

5.4. Mazecode

Since several of the export formats entailed producing a file that consisted of a bunch of boilerplate and a heap of generated and very repetitive code, some of them share the same implementation approach: The output is mostly boilerplate, with a single method that takes a string that encodes a maze structure; the last line of the output is a call to that method. The string is generated by *CellGrid::mazecode*(), and follows a simple format. The model is a state machine with a "current" *x* and *y* position, and an "original *x*" position. The mazecode commands up to 1.1-16 are simple, and cannot deal with octagon tiles yet, nor do they include the solution path. The object produced should be *walls* and *poles.* Poles are junctions between walls, they may be invisible or rendered as something separate.

char	*mazecode action*
+	move 1 step right, plant a pole
-	plant horizontal wall
|	plant straight vertical wall
/	plant vertical wall veering left
\	plant vertical wall veering right
space	move 1 step down, go to original *x* (end of square tile row)
;	move part-step down, go to original *x* (end of tri/hex tile row)
.	move part-step right (for triangle, hexagon tiles)

Plans are to revise this for a new mazecode variant. Since the mazecode string and the boilerplate function that it is fed into are always emitted together, there should not be any of the usual concerns about version incompatibility. The new state machine has a counter *n*, a border-wall flag *b*, and knows the tile shape. Provisional command list:

char	*mazecode action*
+	move *n* steps right, plant a pole
-	plant horizontal wall, *n* tiles long
_	plant horizontal wall with odd vertical offset (octagon only)
|	plant straight vertical wall, *n* tiles long
/	plant vertical wall veering left, *n* tiles long
\	plant vertical wall veering right, *n* tiles long
space	(ignore)
;	move *n* steps down, go to original *x*, set *n*=0
.	move *n* steps right
,	start odd row
digit	set *n*
*	next wall is a border wall (set *b*)

In the new state machine, all commands that use *n* reset it to 1 afterwards, and all wall commands reset *b*. **Update:** Implemented new mazecode in 1.1-19.

6. Open issues

Open issues, which can be either bugs or feature requests, are sometimes also tracked in tickets in the *qtamaze* tracker system of SourceForge, as lines in the *amaze*(1) man-page, in the *TODO* list in CVS, and in “XXX” marked comments in the sources. This document is for a more detailed or comprehen-

sive look at open issues.

6.1. General

1. We could add a *regenerate* button, as an alternative to Ctrl+N. There is room under the sliders for some buttons. Another place would be to add a toolbar, but that starts looking a little daunting for such a simple program. Same for adding a button to cycle between the tile shapes.
2. *File>Open*, *File>Save* don't work yet. They do not really make sense as long as this is not in any way an editor. Maybe we should get rid of them.
3. Maybe add “actual value” labels after wall/path sliders, to reflect how the canvas scaled down the requested value to fit the maze of the screen; cf. *Canvas::paintMaze*.
4. Allow running without GUI, just generate and export. Backport the *option.cpp* from *gent*(1) so we can mix options and file arguments for this. We'd need to have all the settings as options, including display area dimensions and colors, export image type, and for export formats the optional gamma and quality values. **Update:** Backport done, non-option args not used yet.
5. The enter/leave position markers are drawn the same way, maybe they should be different shapes.
6. I don't like how colors are saved in the settings file; change to use explicit RGBA numbers?
7. Add option on Windows to use INI file instead for registry for saving settings between runs?
8. On Windows, the installer reports amaze as being from an “unknown publisher”. Should change to Morgul, or give a reference to the SourceForge project.
9. We put a lot of effort into minimizing maze painting time, partially because when the animated solution is creeping through the maze we get a repaint event every second. However, it might be more worthwhile to always first paint to a background buffer without showing the path, and if the maze has not changed (not the colors, or path width, etc.) just copy the buffer and paint a path over it. This trades off extra memory consumption for less drawing computation. **Update:** Done, as of 1.1-11. Reduces overhead on my system for a 100x200 octagon display with creeping solution from 25% (on 4-CPU system) to <1%.

6.2. Maze quality

The whole purpose of this tool is to generate mazes, so we need to consider what makes a maze “good”, and try to generate mazes that measure up to that standard.

1. The algorithm in *niceCut*() to determine eligibility for a cross-cut is kind of arbitrary, and could be improved. It currently measures a remaining path for having at least a given length (see *CrossParts*) or a given number of bends (see *CrossBends*) in sequence. It should not count the initial bend, or bends from triangle vertical to point and back to same vertical (but does now). Maybe it should add lengths of branches, or adapt to average wall length in maze, or change for selected tile shape, or allow user to tweak the values. Also, it currently tries *n* times in random locations; if we have a lot of cuts, it might be more efficient to do a single pass over the maze to collect eligible cuts and then randomly choose among them.

2. The cross-cut algorithm currently only applies to non-solution cells. At the same time, we tend to generate overly long solution paths (subjectively speaking). Maybe we could use the same cross-cut approach to tweak the solution path by cutting short loops in it. This needs to be a little different, since we do not want to introduce a bypass path, so a solution cut needs to close off one of the original connections. **Update:** Done, see *reducePath()*.
3. There seems to be a bug in either *crossCut* or *reducePath* in octagon mode; I've seen instances of bypass, i.e. a non-solution path connecting two parts of the solution path. **Update:** Fixed in 1.1-8; see *CellGrid::zone()*. We now divide the maze into self-contained zones at the start of cross-cutting, and only cut within a zone. This gets rid of bypass, and as a bonus this filters out undersized zones so we don't waste time trying to cross-cut in them.

6.3. Editability

Any sufficiently configurable program becomes an editor for the configuration settings. At some point of tweaking we would start to change *amaze* from a configurable maze generator into a maze editor with generation features. Depending on how far we want to take that, a fresh redesign may be in order. Here we discuss some intermediate steps.

4. Let user tweak the maze generation algorithm. Currently we just have the “stay” value (not working for triangle tiles). At each tile the algorithm basically has one choice: which tile to try next to add to the path. We could capture this in an expression, and let the user parametrize that.
5. Let user pick departure and destination tiles. If we only allow rim tiles, we could have a single in arrow and an out arrow to drag along the maze border. Good practice for Qt drag-and-drop support. Not sure how they should react to resizing; maybe treat current position as preferred percentage of width/height until user does explicit drag. **Update:** amaze 1.1-5 added a right-click menu to pick rim tiles. **Update:** amaze 1.1-7 adds visible markers, but not draggable yet, only through context menu (right-click). **Update:** amaze 1.1-10 adds (temporary, undocumented) options “-xenter”,, “-yenter”, “-xleave”, “-yleave” taking an argument in range 0—100 for a percentage of the width or height of the cell grid.
6. Currently all tiles in the grid are used for a maze. We could let the user mask out some of them, to get a different rim shape, and maybe even allow holes in the shape. Maybe allow toggling individual tiles, or masking from a background image. **Update:** New option “-mask” and menu action File > Open Outline do this as of 1.1-11 (still in alpha), has many rough edges.
7. Let user save/load a maze. If we add user-modifiable state to the grid, the user should be able to save and reload it. Maybe also add auto-save, in that case, and undo/redo.

6.4. Export

1. Could add "description" (text box) to options part of export dialog, for use with the formats that support it.
2. For the "gamma" option I don't provide a slider yet, because though there is a floating-point spinbox, *QSlider* is integer only. Need to figure out how to best map this.
3. Output maze to ODT? To Povray?

4. Add export to image sequence, for later animation; cf. Ubuntu *gifsicle*(1).
5. We could export as animated GIF, with a subsection of the solution winding its way through the maze. It does not seem *QImageWriter* will let you write an animated GIF, though. We could port part of *gifsicle* for that.
6. Should we have a way to switch off the "confirm overwrite" dialog used in export?
7. When user selects existing file name for export, we now suppress appending the default suffix, even if the name did not appear in the filtered list, so if a user literally types "x" and a file "x" does exist, we will overwrite (after confirm), whereas if "x" does not exist we extend to (and check for overwrite) e.g. "x.png". Is that the best way to handle it?
8. In export, if the user manually types in a name with the "wrong" suffix, we do not warn; should we?

6.4.1. To Blender

We only export to Blender 2.49 now, and only as a Python script that the user needs to manually load into the scene (in a text editor panel) and then execute (Alt-P). Alternatively, the user can run “blender -P *script*”, delete the standard cube, then save-as from within Blender. I would not mind putting in the time to improve this, if anybody seemed to be (interested in) using the feature.

1. The generated wall and pole objects should have a common parent object. **Update:** Done.
2. The tiles should have floor planes. **Update:** As of 1.1-7 we add a rectangular ground plane, only sized properly for square mazes. We also pass in (but do not use) the path color. **Update:** Fixed ground plane size in 1.1-19.
3. The *mazecode* string should be split to avoid really long lines. **Update:** Prepared for this, see new “sep” argument of *CellGrid::mazecode*().
4. The solution path is not reflected in the *mazecode* or Blender script.
5. Maybe generate ".blend" file directly using “blender -b/-P” run. Think *Qprocess*.

Update: Exported script now targets the Blender 2.73 Python API; this is incompatible with Blender 2.49, and vice versa.

6.4.2. To Maya

1. Export to MEL loses the colors. We should probably just generate a single Lambert shader with the wall color and defaults for specularity etc.
2. We only export to MEL now, but Maya also supports Python (both native and *PyMEL*). Add export for those formats?

6.4.3. To X3D

1. X3D/VRML export has mostly the same shortcomings as the Blender one.
2. The colors and shapes are now fully written out for each pole and wall object. Not sufficiently

familiar with X3D yet, but there has to be a better way.

3. The initial viewpoint (“departure”) does not look at the maze. Needs different position and/or rotation.
4. The maze has no tile plane, and no solution path indication.

6.5. Coding

I've tried to resist my impulse to create too much glue code initially in the *CellGrid* class. Now that the code is growing in size, we should refactor some of the *gmat* access and direct coordinate calculations into macros and inline functions.

1. In "Canvas" class, split into separate subclass per tile shape.
2. Move printing job to separate thread, with progress bar.

Code size has not ballooned too much, I think. As of 1.1-19 we have the following C++ stats in lines of code:

file	*v1.1-10*	*v1.1-14*	*v1.1-19*
src/amaze.cpp	1,297	1,608	1,848
inc/amaze.h	122	179	190
src/canvas.cpp	1,313	2,327	2,732
inc/canvas.h	148	273	308
src/cgrid.cpp	1,740	1,862	1,948
inc/cgrid.h	296	356	411
src/expdia.cpp	319	319	333
inc/expdia.h	67	67	67
src/main.cpp	536	617	633
src/mexpr.cpp	-	-	501
inc/mexpr.h	-	-	83
src/option.cpp	132	132	132
inc/option.h	31	31	31
src/update.cpp	249	249	896
inc/update.h	101	101	221
inc/version.h	15	15	15
total	6,366	8,136	10,693

6.6. Glitches

1. Printing tall maze in hexagon tile mode does not work properly yet, falls off page.

2. In PostScript print, the line caps stick out a bit.
3. Triangles mode sucks for stay>1, because diagonal is not considered straight at the cell grid level. Likewise, octagon mode inconsistently counts the diagonal tile connections as horizontal, because in the cell grid they are at (*x+/-*1, *y).*
4. Large triangles should have tile height at sin(60) (= approx. 87%) of width to look equilateral, not just isosceles. **Update:** Done as of 1.1-11, for triangles of sufficient size; same for hexagons and octagons.
5. On Windows, the Ctrl+Q shortcut does not quit amaze. On Ubuntu it does. Ditto for Ctrl+Space to toggle tile shapes. Maybe reserved by Windows? If so, pick some other shortcuts. **Update:** It turns out Ctrl+Space is the default shortcut for switching input methods (e.g. pinyin) on many systems; changed to Ctrl+“.” (non-numpad period).
6. For hexagon tiles, the animated path takes two steps to do one vertical tile, because it equals two cells). That is visually inconsistent.

6.7. Eye candy

1. Add a background image. Either to composite into image, or to drive something in the masking and maze generation.
2. Add cute sound effects.
3. Let user select margin color for export (now hard-coded as alpha).
4. Let amaze executable appear as “amaze” icon in thumbnail view under Ubuntu. **Update:** We do have this working in the WinXP version now. However, we might want to provide a 64x64 and 48x48 version in the ICO file, currently it only has 32x32 (original) and 16x16 (scaled down).

6.8. Install and ports

So far I've done installers for Ubuntu and Win XP. The debian installer for Ubuntu was fairly straightforward, although I needed some samples to get the build script running properly. The Win XP installer took a *lot* longer to get running, WiX is a bit of black magic, and the whole experience of getting a semi-working MSI was somewhat daunting. (Please don't even mention localized WiX Uis, auto-uninstall in *msiexec*, and other things that haunt my bad nights.)

1. Do Mac OS/X port? Solaris?
2. Wrap it up as an RPM package. Note: Started trying to do this, but to test with OpenSuSE or Fedora I need to run *VirtualBox*, and that requires switching on AMD-V, which my current (old) BIOS version does not let me do... More work. **Update:** Started on it, see *qtamaze/amaze/rpm/**.
3. Find copy of EUPL v1.1 to copy into sources. Also add link to it.

Localization of installers would also imply diving into MS language codes and code-pages, rather than plain Unicode strings. Microsoft language codes are 6 bits area + 10 bits language apparently, but not

entirely consistent or complete[1]. Some of the language codes involved:

lang#	*locale*	*description*
0406	da	Danish
0407	de	German
0809	en_GB	English (GB)
0409	en_US	English (US)
----	eo	Esperanto
0C0A	es	Spanish
040C	fr	French
0411	ja	Japanese
0413	nl	Dutch
041D	sv	Swedish
0804	zh_CN	Chinese (simp.)
0404	zh_TW	Chinese (trad.)

6.8.1. NSIS

After release 1.1-21 I got installer bug reports that boiled down to group “BUILTIN\Administrator” not being found on non-English Windows. Turns out the group name is localized, the non-localized name is SID “S-1-5-32-544” (wonderful mnemonic), and there is no way to use the SID in WiX without yet another plug-in, requiring running a Visual-C compiler next to the MinGW compiler, etc. After this final straw, I decided to experiment with a switch to NSIS, which I had not used before. After half a day I have an installer with some rough edges. Current open issues:

1. When clicking "Cancel", the confirmation dialog Yes/No buttons are not localized. Internal NSIS bug? Missing switch on our side?
2. The Esperanto translation uses the x-convention (e.g. “cx” for “ĉ”), is there a proper ISO-8859-3 one?
3. Need to get the license text localized with MUI2. Luckily we can just reuse the same RTF files already done for the WiX version.
4. MUI2 allows “SimpChinese” and “TradChinese”, but they don't show up at runtime in the initial language selection dialog. Not really acceptable to omit, since Chinese is the biggest locale in the Amaze download statistics.
5. NSIS command *MinimalTargetOS* does not work yet... When is NSIS 2.47 coming out? There is a separate, non-official Unicode port on another site, but I don't want to rely on non-official

1 For example, Spanish has a dozen different “area” part, with 2 separate ones for Spain (differing only in sorting order), a generic one for “Spanish in Latin America”, several for specific countries e.g. “Spanish in Argentina”, a separate “Spanish in US” (but not for Canada) and “Spanish in Puerto Rico”. In contrast, Esperanto has no assigned code. English has a defined code-page (decimal 1252), Chinese does not.

versions or unstable builds.

6. The uninstaller UI does not reflect the language selected during installation.
7. Get colors & images to theme NSIS installer.
8. According to the NSIS documentation it should be possible to cross-compile and then build the Windows installer from Ubuntu, so we could build both OS targets without changing environment. Need to figure out if that works. **Update:** Now cross-compiling on Linux, thanks to Debian *nsis* 2.4.6 package and MXE (http://mxe.cc). The "mxe.cc" website has a Unicode capable 3.0 alpha version, but this may not be ready for prime time. See the new *mxebuild.sh* script.

6.9. Localization

1. Localization in *amaze* prior to 1.1-7 did not work, because the "qm" files were not packaged. It did work in the development environment. We need to either install them as separate files in the package, or include them as compiled-in resources. I'd normally favor separate files (easier to debug and add, no runtime memory overhead for unused locales), but the hassle of making them separate files in the Windows MSI makes me lean towards inclusion as a resource. **Update:** Separate file after all, leading to a lot bigger WiX file. Using a Qt lib package dependency instead on Ubuntu.
2. Check the localized files for shortcut/accelerator conflicts.
3. Find out why *QtLinguist* keeps resetting the language in *amaze/trans/amaze_eo.ts* to English.
4. Get somebody to proofread the Spanish localization. Ditto for Danish.
5. Add languages: [fr], [zh]? Stats on 2011-02-09 show total 3,333 downloads of 1.1-6.msi, of which 73% from China, so zh_CN is probably the most worthwhile locale. With a non-Latin based menu language, I'm not sure how to define and show accelerators in Qt. Some widget buttons in Qt have a label format like "颜色(C)" in Chinese, where the English one would have "Color", and the German one "Farbe". **Update:** Added accelerator keys to the Chinese strings, using the initial of the pinyin transcription.
6. Localize the Windows installer. I think this requires separate ".wxs" files.
7. Translate the man-page. **Update:** As of 1.1-20, we have a [de] and [nl] version, besides the [en_US] one. The 1.2-1 adds [es], needs proof-reading (badly!), and may add [eo]. I'm limiting a few parts to the English version only, specifically the Qt/X11 generic options (too technical for casual users, and hard to translate) and the ToDo/Bugs lists (change too often).
8. The *Help>Help* text should be localized for more locales, not just [de] and [nl]. Note the locale does not work here for now, need to find out why. See the *amaze/doc/help_*.htx* files. The locale resolution in *Amaze::help()* does not seems to work; we always get the non-locale-specific version (*amaze/doc/help.htx*). **Update 2011-02-03:** We now get warnings from *rcc*(1) about "potential duplicate alias detected", tracked to "http://bugreports.qt.nokia.com/browse/QTBUG-7812", may be related. For now, forget localized resource aliases; instead, use localization to get name of localized resource, then use that.
9. Loading the *qm* file for the Qt widgets themselves requires (a) having it available, and (b)

knowing where it resides. I've got a problem with (a) for [nl], and for (b) on Windows. On Ubuntu, the available translations seem to appear at "`/usr/share/qt4/translations`"; not sure if there is a less hard-coded way to get at them. Don't know about Windows. In the "xpbuild.bat" they would be "`%Qt4%\qt\translations\qt_da.qm`" etc. Do we just copy them and ship as part of the app? Would add about 200-300kB per locale. **Update:** Shipping in WinXP MSI as of release 1.1-8. Note there is no `qt_eo.qm` (Esperanto) or `qt_nl.qm` (Dutch) in the WinXP version of Qt4.

For the Qt-supported "tr(...)" localization in the sources, the Qt tools (*lupdate, linguist, lrelease*) work well. The main problem I've had is having to redo translations when a message in the source text changes.

For the help text files and especially for the man-pages, I have no good work-flow yet. I did the Dutch translation of "amaze.1" by hand; did some global substitutes to start, then ended up mostly retyping all text. For the German translation I tried going through Google translate; had to first strip most *troff* mark-up, then re-type markup in the translated result, and repair the unwanted translations of option names etc. Found out some issues: *groff* needs extra "-k" option for UTF-8 support, hyphenation command is not entirely standard (".hla" versus ".hylang"). The auto-translated German was really bad, maybe because the input is very abbreviated and technical. Given that only advanced users will need the man-page (I hope!), and that those are more likely to be able to use the English (or Dutch, or German) version, I'm giving further man-page translations a low priority, unless somebody else volunteers.

6.9.1. Dynamic locale

So far, the locale is determined during command line option processing, and fixed before the GUI comes up. I've considered adding a "Language" menu so the users could switch to a different locale from the GUI. See the *Amaze:: langMenu* item, currently disabled. The problem is that it requires handling a translator change for all widgets that use or indirectly contain "tr(...)" calls.

- One way would be to add a handler just to the containers (like *Amaze* and *Canvas*), but that would have to duplicate almost all of the "tr(...)" calls a seconds time, making for bloated "*.ts" files and a maintenance nightmare.
- The alternative is to write our own "extended" version of *QAction*, *QMenu*, etc. classes that retain the untranslated string inside, and that do the "tr(...)" calls on them both on first creation and on a locale change-over event. I'm not sure how to use this with *QtLinguist* though, as it seems to look for the "tr" calls in the code to find the strings to process.
- One hitherto unexplored option would be to do the following on a locale change: collect current settings, delete all GUI widgets except the outermost window, then rebuild everything with new locale, and finally restore transient settings. Could be fragile around the setting save/restore part. **Update:** Implemented this in 1.1-22, works okay with some rough edges; see *Talk* menu. Not entirely happy with the menu texts.

6.9.2. Organization

Ultimately, localization is the part of any small software project such as this, where one person cannot really hope to do a decent job; there are just too many languages. The real learning experience for this

should then involve dealing with organizational issues, such as:

- How do you find volunteers help with localization? Note SourceForge has a “help wanted” ads section.
- How do you coordinate localization? In a volunteer network, you may not be able to get deadlines adhered to.
- How do you assess the quality of a particular set of translations? How do you register who provided a particular translation string?
- How do you deal with disputes on the “proper” translation for items?
- If you have multiple people working on the same locale, how do you ensure consistency of vocabulary and style?
- If some translations are out of date, should they be retained without change, marked internally, annotated visibly, or removed? Is having no translation better than having an obsolete one? What should be the policy for adding (or removing) entire locales?
- How do you know which parts of a translation are up-to-date?

7. Update log

Notes here from update started 2011-02-03. The Ubuntu 10.10 version for Qt4 is now 4.7.

1. The README file does not describe how to call *qmake* etc. for development.
2. The color picker is kind of weird, the value is determined by the slider to the right of the colorful HSV square, so selecting a color in the square only determines the H and S of the pick. This means if selection started with black, you first need to change the slider, else it will stay black. This is a feature, but may be confusing. Put in “Help”?
3. Added options: -cuts, -tweak (“no-reduce”, “debug-grid”, “debug-walk”), -path-color, -tile-color, -wall-color, -mask, -mask-heart (= “`-mask [heart]`”), -full, -mini, -no-show, -text, -text-chars, -text-fill, -text-color/-tc. Aliased “-p” as “-path”, “-s” as “-do-show”, “-path-color” as “-pc”, “-tile-color” as “-bc” (“-tc” = “-text-color”), “-wall-color” as “-wc”.
4. Added floor plane in Blender export, not well tested.
5. Added a bit more localized text. I completed the set of “doc/help_*.htx” files, using Google translation with manual fixing up for Spanish, Danish and Chinese. The Chinese version is probably really bad at the moment... **Update:** Received offer from SF user *openyan* to help with [zh] localization. Added [en_GB] locale with “colour” instead of “color”.
6. WinXP *amaze.exe* now has built-in icon. Also added as separate icon file in install. Fixed WiX description to set “allow all users” file permission.
7. Need to test with non-opaque color. Users can select them in the dialog, but the results may be messy, since the *paintXxxMaze()* functions make no effort to not overwrite areas. Use right alpha blending mode!
8. I've started on grid-mask support in 1.1-10. This should properly be done through image files

for the masks, and include using those masks to drive the path generation (like “stay”, but on a per-cell basis). As a special Valentine's day hack I added “-tweak mask-heart” to only mask off the grid shape to a love heart, so you can generate a card from it with a text like “show me the path to your heart” or similar. **Update:** See new “-mask” option in 1.1-11, needs more polish.

9. Added a “demo” mode in 1.1-14, to cycle between tile shapes and path display modes. The demo steps are not currently configurable from the command line or GUI, it's a hard-coded sequence. However, the sequence itself is a command list fed into a little state machine, so it is easy to change and extend at the source code level.

We may need to add more files for localization, if we do not turn them into compiled-in resources (which would inflate the executable size a lot). The installed file list in Debian for 1.1-6 is:

Path	**Use**
`/usr/bin/amaze`	The application executable
`/usr/share/applications/amaze.desktop`	Desktop menu etc.
`/usr/share/app-install/desktop/amaze.desktop`	?
`/usr/share/doc/amaze/changelog.Debian.gz`	Copy of `debian/changelog`
`/usr/share/doc/amaze/changelog.gz`	?
`/usr/share/doc/amaze/copyright`	Copyright notice, = `debian/copyright`
`/usr/share/man/man1/amaze.1.gz`	compressed man-page

I assume the translation files should be installed as “`/usr/share/amaze/qt_de.qm`” etc. **Update:** Since 1.1-8 we added the 16x16 etc. icons for the Ubuntu install (no SVG yet; try Inkscape?).

8. Future Directions

Amaze version 1.1 has progressed from release 1.1-1 through 1.1-10, gradually growing the code-base and minor improvements. However, the slow accumulation of code has lead to a code-base that is showing the limits of the original design, and getting harder to maintain. We should spend a little time reorganizing some of the code before adding any more feature work. Here are a provisional wish-list, target, plan and more for an upcoming Amaze 1.2. **Update:** The 2015 reboot of amaze will use version 1.2 for the platform refresh.

One bridge I don't want to cross yet is introducing the amount of configuration state and actions that would move Amaze into the conceptual realm of a “maze editor”, because that implies whole slew of infrastructure (save, open, close, undo/redo, cut, copy, paste, find, auto-restore, versioning, format specifications, etc.) I do not want to dive into right now – maybe in an Amaze version 1.3 or 2.0 if that ever comes out. I'm starting to collect some ideas.

In my opinion, a lot of open-source software errs in not paying enough attention to the experience provided to a casual, first-time user, who is likely to have only a mild and transient interest in using the software to either pass some time, or accomplish a very specific and simple task. Essentially, *anything* that comes between the user finding a reference to the product and finishing the task (or enjoying the

activity) is bad, and easily too much. Things in that category:

- Having to *look* around to *find* the appropriate download button. (In fact, having to *download at all* is a bother, which is why web-based apps are increasingly popular).
- Having to do so in a *different language.* (Localization begins at the download page.)
- Not seeing *confirmation* about *what* the software does, to reassure the user that the program is likely to fulfill the purpose of the user. (Description in ten words or less, screenshot, example output, etc.)
- Having to consider whether to *trust* the download. (Getting there via endorsement site is better. Place for comments is good.)
- Being made to select the appropriate *version*, and *click* on the button. (Should auto-detect OS, and maybe start auto-download.)
- Having to figure out *how to install* the software. (On Windows, anything beyond an MSI is asking too much. On Ubuntu, it should be a prepared ".deb" package. Tarballs etc. lose all casual users.)
- Having to *make choices* during installation. The user just wants to "rip it out of the box". (Sensible defaults matter, configurations should be done later, and nobody reads EULA texts anyway.)
- Having to *make choices* anywhere. Choice is only welcome when the *user* asks for it, when it is clear they can change their mind *afterwards*, the *implications* are clear, and the options are *limited*. If a proposed setting is "the default, usual, conventional thing", either just do it, or make sure the user knows that, so they don't confuse a message of "hi, just letting you know I put this in a good state" (low attention, *relax*) with "you have to stop your train of thought and make an effort to understand and not screw up here" (distraction, annoyance, *stress*).
- Having to *read*. (Users do not read manuals. Or README files. Or dialog text over half a dozen words. Really.)
- Once installed, having to *hunt* for how to start the program. (Using a distinctive, recognizable icon helps. Produce both a shortcut and a desktop menu item, including tooltips etc.)
- The *empty first screen* syndrome: opening the program for the first time, and having no idea how to proceed.

This list could continue for a while, and I'm sure *qtamaze* is not perfect on the score. However, we need to keep focused on getting better in this respect, and making certain we're not regressing.

8.1. Masking

This is planning for Amaze 1.2-*x*.

From the users' perspective, the focus of the new version will be on support for mazes in arbitrary shapes, using a mask image. A peek at things to come was in the 1.1-10 release using option "-tweak heart-shape", also known as the Valentine option, allowing the creation of heart-shaped mazes. From a

development perspective, the idea is to get rid of a lot of special-purpose rim case handling, use image buffers to cut down on the amount of repainting, and clean up the whole handling of enter/leave markers. In general, we want to cleanly separate the “tile” and “cell” concepts, the UI should only deal with the former.

Update: I decided to do interim releases 1.1-11 through 1.1-20, incorporating some of the stuff planned for 1.2, because it was taking a long time.

1. Change the “width” and “height” user controls on the left to refer to the visible maze size measured in tiles, not cells. The use of cells should be a purely internal implementation detail. **Update:** Done in 1.1-15, in sliders, saved preferences, and CLI options.
2. Don't bother to construct grid rectangles that contain odd top/right fringes, like hexagon mazes with the bottom row offset from the top row. Odd fringes require a lot of special-case code, tend to produce uglier mazes, and bring no real benefits. Once we have support for arbitrary maze shapes, a user could still create odd cases if really desired, by masking out a final row or column. **Update:** No special code anymore in 1.1-11 for fringes. Also, since 1.1-15, avoid the ugly “dangling octweens” case for octagons by forcing an odd cell row count for that shape.
3. Change the background painting to use a cached image buffer. This should speed up the continuous repaints for creep mode. We could also consider using less optimized code for the background drawing as a trade-off, when the drawing is not needed all the time. Maybe use a bitmap plane, so background color changes do not require repainting the background buffer. **Update:** In 1.1-11 we cache the entire maze pixmap, prior to painting solution path and markers. Reduces overhead of path creep mode a *lot*.
4. Use a buffer image when drawing hexagon, octagon or triangle backgrounds, to avoid recomputing. Alternatively, use a single paint path and reuse that. Not sure which is better. With masking introduced for maze shapes, we can no longer just draw the edges and then block in the middle as a single rectangle. **Update:** In 1.1-11 see *findSolid()*, we try to optimize for a single best-fit rectangle, paint the rest per tile. Not so bad, since background is cached now.
5. Add color-set selection, similar to Ubuntu's *agave*, to pick matched color triple (back, path, wall) using color theory rules. **Update:** Release 1.1-15 added a “Spin colors” action, which is remotely related to this.
6. Instead of showing a green blank canvas with "no maze" written in the middle, be more user-friendly; show the maze area grid, and a more usable message. This would also let the user move the enter/leave points if those are causing the problem. **Update:** In 1.1-19, we add a more specific message saying *why* there is no maze.
7. Have a grid overlay to paint over the background (if any) but under the walls, to aid the user in picking tiles e.g. for enter/leave points, or showing how maze masks would map to tiles. **Update:** Done in 1.1-11, action Ctrl-H.
8. Let user specify an image to use as the maze mask. Stretch maze to cover the canvas (maybe with an option to lock the aspect ratio). **Update:** Done with no locking, no solution for islands.
9. Add extra flags in cell to distinguish border walls from regular ones. **Update:** Done (*vbor, hbor*). Note we are not drawing moat differently yet.

10. Use the same to mark a non-wall as an enter/leave point. Allow user to set wall thickness and color of border wall separately. Offset extra-thick border walls to not intrude on in-maze space. **Update:** Added Moat (analogous to Wall) control, but not used yet. Commented out in 1.1-11.
11. Generate draw-path separate from solution-path list, so we can have a single step per tile in creep mode, and maybe for mazecode export too.
12. Include the solution path in mazecode output. Change to start off with tile shape indicator. In Maya code, add ground plane. In Blender and Maya, add path display. Maybe add mode to call Blender as subprocess and export the ".blend" file, maybe even add turntable generation. Add marker position and direction info to mazecode.
13. Add option to generate pictures of creeping path as series of images. Include factor for only every Nth frame, or single image for background + walls and separate overlay frames containing only the creeping path with rest transparent.
14. Keep enter/leave marker size independent of tile size (but not smaller), so we can sill see the markers in a big maze. Allow users to suppress them in the printed and exported images. Maybe change the whole concept of how we place enter/leave points: should tie them to a specific border wall instead of a cell location, because that way we implicitly select the direction as well. **Update:** Now have separate size, with min and max, computed in *TileScale.*
15. Maybe allow custom wall elements for drawing the maze. This could be based on either wall elements or crossings (where walls come together) or both. Could be done by letting user supply an image for each, or giving an image and dragging boxes to pick the elements. May be too involved for now, gets close to editing.
16. Have a list of mazes for the user to cycle through and add or delete, instead of just a single one. Show list as thumbnails that include the solution path.
17. Improve the update check to optionally run automatically on start-up, maybe limited to weekly check. Offer to download new version if available so the user does not need to go and fetch it manually. **Update:** Dialog now lets user download and (attempt to) install. MSI still needs manual uninstall, so limited usefulness on Win XP for now. No automatic periodic check.
18. Allow the user to switch locales via a menu action. Generate localized installers, using the MSI localized WiX files. Put in accelerators for all [zh_CN] menu and button items. Add a [zh_TW] locale. Get all locales proofread. **Update:** Locale switch added in 1.1-22. Experimental NSIS installer on Windows is localized (WiX/MSI is not).
19. Find somebody with a Mac willing to do a Mac version.
20. Add new maze types. One would be a “squares with twists”, where a block of 4 cells could do a cross-over (X shape), but otherwise a regular square maze mode. Another option is a “message maze” where we put letters in the maze so the solution reads off a message, and generate garbage letters for the other paths, using a language-specific statistical distribution.
21. We could add a “Mail to” export option to go directly to email with a maze. Especially useful in combination with the letter maze. **Update:** No mail-to yet, but 1.1-20 adds letter mazes (“mazegrams”). In 1.1-21 we will add font selection and scaling for the mazegram letters.

8.1.1. Network files

Two seemingly disconnected improvements are a little more involved and related than I originally anticipated:

1. The current “check update” dialog tells the user whether or not the version is out of date, and even provides the URL of the directory on SF to download it from, but the user then has to put in a lot of effort:
   (a) copy the URL manually;
   (b) open up a browser, and create a new tab;
   (c) paste the URL as the address, and hit enter;
   (d) find the download file link on the page;
   (e) click on the link;
   (f) tell the browser where to save the image;
   (g) quit the Amaze program;
   (h) manually remove the current installation of Amaze (Wind XP: Control panel, Add/remove programs; Ubuntu: Synaptic, Local packages, or “`dpkg -remove`”);
   (i) open up a finder to the download directory;
   (j) double-click the installer file;
   (k) click through the installer buttons;
   (l) re-start Amaze from the Start/Application menu or desktop shortcut.
2. The user can drag-and-drop local image files onto the canvas as outlines, but for non-local file references (URLs dragged in from browser links), the user needs to do “save to file” from the browser, then open up the download location in a finder, and drag the downloaded file onto the canvas.

Both the first part of the update, and most of the non-local image procedure, concern the conversion of a URL to a local file reference. In general, the user does not really care about the local file; the need for its existence is internal to the program. We run into many of the same issues as a browser programmer.

1. It can take a long and somewhat unpredictable time to copy the remote data. Because of this, we generally do not want to block the UI main loop while waiting for the data, but move the network operation to a separate thread.
2. The user may decide to cancel the ongoing operation, so we need to offer a dialog that lets the user issue a cancel.
3. The user needs some information about long operations, like a progress bar, so we need to estimate the size in advance, and provide some feedback metric while ongoing. This may involve yet another thread, to provide timing ticks.
4. During the download, we can either make the download dialog modal (no UI actions available to the user except “cancel”) or allow the user to continue with operations that do not depend on

the result of the download (the typical browser approach). In the latter case, we need to decide how to continue when the download is ready. Should download failure be modal?

5. We could combine the modal and non-modal approaches too, depending on the context: upgrade downloads could be non-modal, and mask image downloads modal.
6. Should an upgrade download in progress be suspended during a later mask image download, for better interactive response and lower bandwidth demands?

The next set of issues concern caching of remote files. The user does not care much about these internal files, so asking them to pick a name and location for them is bothering the user with internal bookkeeping details, a bad design. Instead, we need to manage a local file copy cache somewhere. **Update:** As of 1.1-18 we have download and installation in the update dialog.; no mask URLs yet. Progress bar is useless because SF's Apache config does not put the file size in an HTTP header, so we don't know how big the file is until we've downloaded it.

8.2. Editor mode

This is planning for Amaze 1.3-*x*. Very preliminary. We want to give the advanced user more control over the look and composition of the generated mazes, and that means basically acting as an editor more than a pure generator. The trick will be to add facilities without reducing the appeal to the majority of users who will be first-time, casual users.

1. We could have “draw the border wall” mode to let the user create a mask by drawing instead of masking, but that may be too much like an editor mode for version 1.2.
2. We could get more information from the masking image, and pass some into the mazecode for export. Examples would be color information for walls, or terrain height for 3D export.
3. Let user switch locale through a menu.
4. Have built-in code to generate an animated GIF, MNG or APNG. All of these have some issues: GIF is primitive, MNG is badly supported, APNG is contentious. Looked into APNG, does not look too hard in principle, but not looking forward to writing my own LZW-compression code; there are libs, but not all cross-platform. **Update:** I started on some code to produce animated SVG, see *svg*() in *output.cpp*; however, it turns out you cannot easily animate a path, because it is not a single attribute transformation. I could make a marker run along the path quite easily, though. We could maybe break up the solution path into small segments, and animate each one of those separately so they fade in/out at the right moment. Note that browser support for animated SVG is not universal yet, for example *Firefox* 3.6.13 does not show the animation.

8.3. Circle maze

I'd like to add another maze tile type that is a little different from the ones done so far: a maze of concentric rings, because of the aesthetic appeal. I'll call this a circle maze for short. Technically the cells are wedge-shaped segments of the concentric rings, and rings further from the center have more cells than inner rings. After some deliberation, it seems we could largely treat this as a special variant of the plain square maze, mapping the rings to concentric square rings of cells in the cell grid. The corner cells of those square rings need special treatment because they don't easily map to anything visible in

the round rings, so we make them invisible, and apply special rules to generate the visual walls between the neighboring cells of these invisible corner cells. Masking and canvas-to-cell coordinate mapping (needed for the context menu, and setting the departure cell by mouse click) differ from the other maze tile shapes in needing circle-based projection instead of straight lines, but are not fundamentally different.

Some terminology first. When the maze is *n* cells wide, we horizontally divide it into *n*/2 rings, and number them from the innermost outward, with the innermost (one if odd, two if even) called “ring 0”, then going outward ring 1, 2, through *n*/2-1. We always mask out ring 0 visually so we have an empty space at the center, and put the destination point there.

The hardest part of providing circle mazes may be to find the right mix of optical tricks and maze grid constraints to produce a visually pleasing result. The non-circle maze types all have tiles with perfectly aligned and uniform wall/path sizes, but for circle wedge cells the radial paths (i.e. between rings) are not always as neat. If we limit ourselves to *n*-by-*n* circle mazes (same width as height) with even *n* (odd *n* discussed later), then we have eight continuous radial walls from inside out: 2 horizontal, 2 vertical, and 4 diagonal at a 45° angle. We can call each of the ring segments between two continuous walls an octant. All other radial walls are staggered, because the number of cells goes up by 1 per octant per ring as we go outward. So ring 0 has 0 cells per octant, ring 1 has 1, ring 2 has 2, etc. The total number of grid cells for ring *n* is 8*n*+4, with 4 being the number of invisible corner cells per ring. The staggering happens because for each next ring, the walls must “move over” a bit to accommodate the extra cell.

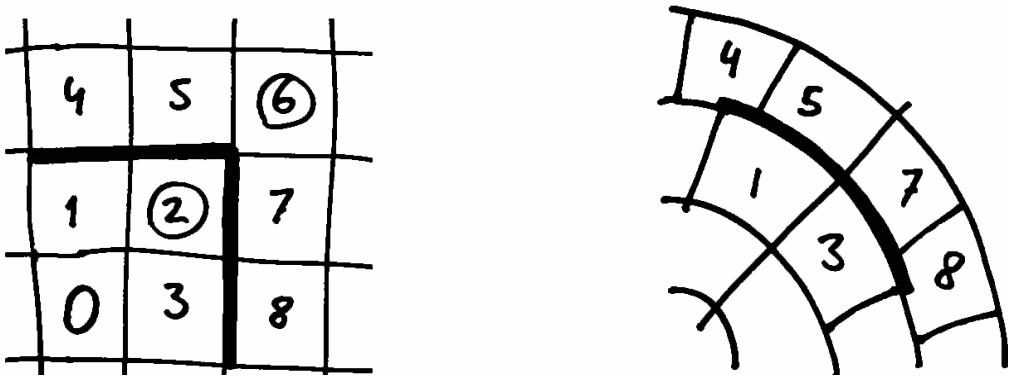


Illustration 1: Mapping cell grid ring corner to visual maze

In the illustration, we show the mapping of a ring corner in the cell grid to the visual representation of the same set of cells as circle wedge tiles. Note how the corner cells (2 and 6) have no visual equivalent. The thicker line, representing a circumferential line, becomes a smooth circle segment. Note how the wall between 1 and 4 is much shorter than the wall from 1 to 5, because of the staggering; this becomes more pronounced as we get further from the center. Also note there is no direct wall between 1 and 5 in the cell grid; what is show as a wall between tiles 1 and 5 on the right corresponds to the combination of the cell grid walls from 1 to 2, and 2 to 5: if both are open, then 1 is connected to 5, else they are walled off. Likewise, the grid has no direct connection between 5 and 7; the tiles are connected if either both 5-6 and 6-7 are open, or both 5-2 and 2-7.

Update: Circle mazes need some more thought. I've got some inactive partial code in 1.1-16. Experi-

menting on paper, it's hard to make circle mazes “look right” using purely automated rules.

9. Localization

Here is a table of some Amaze jargon and the word used systematically as a translation in the localization strings. Sometimes a word has multiple possible translations in the target language, we try to be consistent.

Original	**[da]**	**[de]**	**[eo]**	**[es]**	**[nl]**	**[zh]**
maze	labyrint	Labyrinth	labirinto	laberinto	doolhof	迷宫
message		Nachricht	mesaĝo	mensaje	boodschap	音信
outline	omrids	Grundriß	ujo	silueta	omtrek	外形
path (solution)	vej	Pfad	vojo	ruta	spoor	路
tile	tegl	Fliese	ero	losa	tegel	瓦
wall	væg	Wand	muro	pared	muur	墙

10. References

About providing an RPM package:

- http://www.gurulabs.com/downloads/GURULABS-RPM-LAB/GURULABS-RPM-GUIDE-v1.0.PDF

About using box drawing:

- http://en.wikipedia.org/wiki/Box-drawing_characters – used in *CellGrid::uniBox()*.