wedged

Claude Heiland-Allen

2013–2019

Contents

1 .gitignore

```
dist
dist-newstyle
```

2 LICENSE.md

```
# Free Art License 1.3 (FAL 1.3)

## Preamble

The Free Art License grants the right to freely copy, distribute, and
transform creative works without infringing the author's rights.

The Free Art License recognizes and protects these rights. Their
implementation has been reformulated in order to allow everyone to use
creations of the human mind in a creative manner, regardless of their
types and ways of expression.

While the public's access to creations of the human mind usually is
restricted by the implementation of copyright law, it is favoured by the
Free Art License. This license intends to allow the use of a work's
resources; to establish new conditions for creating in order to increase
creation opportunities. The Free Art License grants the right to use a
work, and acknowledges the right holder's and the user's rights and
responsibility.

The invention and development of digital technologies, Internet and Free
Software have changed creation methods: creations of the human mind can
obviously be distributed, exchanged, and transformed. They allow to
produce common works to which everyone can contribute to the benefit of
all.

The main rationale for this Free Art License is to promote and protect
these creations of the human mind according to the principles of
copyleft: freedom to use, copy, distribute, transform, and prohibition
of exclusive appropriation.
```

Definitions

work either means the initial work, the subsequent works or the common work as defined hereafter:

common work means a work composed of the initial work and all subsequent contributions to it (originals and copies). The initial author is the one who, by choosing this license, defines the conditions under which contributions are made.

Initial work means the work created by the initiator of the common work (as defined above), the copies of which can be modified by whoever wants to

Subsequent works means the contributions made by authors who participate in the evolution of the common work by exercising the rights to reproduce, distribute, and modify that are granted by the license.

Originals (sources or resources of the work) means all copies of either the initial work or any subsequent work mentioning a date and used by their author(s) as references for any subsequent updates, interpretations, copies or reproductions.

Copy means any reproduction of an original as defined by this license.

1. OBJECT

The aim of this license is to define the conditions under which one can use this work freely.

2. SCOPE

This work is subject to copyright law. Through this license its author specifies the extent to which you can copy, distribute, and modify it.

2.1 FREEDOM TO COPY (OR TO MAKE REPRODUCTIONS)

You have the right to copy this work for yourself, your friends or any other person, whatever the technique used.

2.2 FREEDOM TO DISTRIBUTE, TO PERFORM IN PUBLIC

You have the right to distribute copies of this work; whether modified or not, whatever the medium and the place, with or without any charge, provided that you:

- attach this license without any modification to the copies of this work or indicate precisely where the license can be found,
- specify to the recipient the names of the author(s) of the originals, including yours if you have modified the work,
- specify to the recipient where to access the originals (either initial or subsequent).

The authors of the originals may, if they wish to, give you the right to distribute the originals under the same conditions as the copies.

2.3 FREEDOM TO MODIFY

You have the right to modify copies of the originals (whether initial or subsequent) provided you comply with the following conditions:

- all conditions in article 2.2 above, if you distribute modified copies;
- indicate that the work has been modified and, if it is possible, what kind of modifications have been made;
- distribute the subsequent work under the same license or any compatible license.

The author(s) of the original work may give you the right to modify it under the same conditions as the copies.

3. RELATED RIGHTS

Activities giving rise to author's rights and related rights shall not challenge the rights granted by this license.
For example, this is the reason why performances must be subject to the same license or a compatible license. Similarly, integrating the work in a database, a compilation or an anthology shall not prevent anyone from using the work under the same conditions as those defined in this license.

4. INCORPORATION OF THE WORK

Incorporating this work into a larger work that is not subject to the Free Art License shall not challenge the rights granted by this license.
If the work can no longer be accessed apart from the larger work in which it is incorporated, then incorporation shall only be allowed under the condition that the larger work is subject either to the Free Art License or a compatible license.

5. COMPATIBILITY

A license is compatible with the Free Art License provided:

- it gives the right to copy, distribute, and modify copies of the work including for commercial purposes and without any other restrictions than those required by the respect of the other compatibility criteria;
- it ensures proper attribution of the work to its authors and access to previous versions of the work when possible;
- it recognizes the Free Art License as compatible (reciprocity);
- it requires that changes made to the work be subject to the same license or to a license which also meets these compatibility criteria.

6. YOUR INTELLECTUAL RIGHTS

This license does not aim at denying your author's rights in your contribution or any related right. By choosing to contribute to the development of this common work, you only agree to grant others the same rights with regard to your contribution as those you were granted by this license. Conferring these rights does not mean you have to give up your intellectual rights.

7. YOUR RESPONSIBILITIES

The freedom to use the work as defined by the Free Art License (right to copy, distribute, modify) implies that everyone is responsible for their own actions.

8. DURATION OF THE LICENSE

This license takes effect as of your acceptance of its terms. The act of copying, distributing, or modifying the work constitutes a tacit agreement. This license will remain in effect for as long as the copyright which is attached to the work. If you do not respect the terms of this license, you automatically lose the rights that it confers.
If the legal status or legislation to which you are subject makes it impossible for you to respect the terms of this license, you may not make use of the rights which it confers.

9. VARIOUS VERSIONS OF THE LICENSE

This license may undergo periodic modifications to incorporate improvements by its authors (instigators of the Copyleft Attitude movement) by way of new, numbered versions.
You will always have the choice of accepting the terms contained in the version under which the copy of the work was distributed to you, or alternatively, to use the provisions of one of the subsequent versions.

10. SUB-LICENSING

Sub-licenses are not authorized by this license. Any person wishing to make use of the rights that it confers will be directly bound to the authors of the common work.

11. LEGAL FRAMEWORK

This license is written with respect to both French law and the Berne Convention for the Protection of Literary and Artistic Works.

USER GUIDE

How to use the Free Art License?

To benefit from the Free Art License, you only need to mention the following elements on your work:

> [Name of the author, title, date of the work. When applicable, names of authors of the common work and, if possible, where to find the originals].
>
> Copyleft: This is a free work, you can copy, distribute, and modify it under the terms of the Free Art License
> <http://artlibre.org/licence/lal/en/>

Why to use the Free Art License?

1. To give the greatest number of people access to your work.
2. To allow it to be distributed freely.

```
3. To allow it to evolve by allowing its copy, distribution, and
   transformation by others.
4. So that you benefit from the resources of a work when it is under the
   Free Art License: to be able to copy, distribute or transform it
   freely.
5. But also, because the Free Art License offers a legal framework to
   disallow any misappropriation. It is forbidden to take hold of your work
   and bypass the creative process for one's exclusive possession.

### When to use the Free Art License?

Any time you want to benefit and make others benefit from the right to
copy, distribute and transform creative works without any exclusive
appropriation, you should use the Free Art License. You can for example
use it for scientific, artistic or educational projects.

### What kinds of works can be subject to the Free Art License?

The Free Art License can be applied to digital as well as physical
works.
You can choose to apply the Free Art License on any text, picture,
sound, gesture, or whatever sort of stuff on which you have sufficient
author's rights.

### Historical background of this license:

It is the result of observing, using and creating digital technologies,
free software, the Internet and art. It arose from the Copyleft
Attitude meetings which took place in Paris in 2000. For the first
time, these meetings brought together members of the Free Software
community, artists, and members of the art world. The goal was to adapt
the principles of Copyleft and free software to all sorts of creations.

------------------------------------------------------------------------

<http://www.artlibre.org>
Copyleft Attitude, 2007.

You can make reproductions and distribute this license verbatim
(without any changes).

Translation: Jonathan Clarke, Benjamin Jean, Griselda Jung, Fanny Mourguet,
Antoine Pitrou. Thanks to <http://framalang.org>
```

3 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

4 wedged.cabal

```
name:               wedged
version:            3
synopsis:           Wedged postcard generator.
license:            OtherLicense
license-file:       LICENSE.md
author:             Claude Heiland-Allen
```

```
maintainer:         claude@mathr.co.uk
copyright:          (C) 2013,2015,2016,2018 Claude Heiland-Allen
homepage:           https://mathr.co.uk/wedged
category:           Demo
build-type:         Simple
cabal-version:      >=1.10

description:
    Wedged (C) 2013,2015,2016,2018 Claude Heiland-Allen.
    .
    Copyleft: This is a free work, you can copy, distribute, and
    modify it under the terms of the Free Art License
    <http://artlibre.org/licence/lal/en/>.
    .
    Usage:
    .
    > mkdir 7x5
    > cd 7x5
    > wedged 13 9 0.5 72
    > cd ..
    .
    > mkdir 12x8
    > cd 12x8
    > wedged 14 9 0.8 72
    > cd ..
    .
    Output:
    .
    189 EPS files in the 7x5 dir, totalling 25 MB, runtime 3m15s.
    .
    115 EPS files in the 12x8 dir, totalling 17 MB, runtime 6m25s.
    .
    Run time measured using a single core of a 4.3GHz AMD Ryzen 7 2700X
    Eight-Core Processor.
    .
    Information:
    .
    Version 0 worked with GHC 7.8 and Diagrams 1.2 with the Cairo backend.
    .
    Version 1 was updated to work with GHC 8.0 and Diagrams 1.3 with the
    Cairo backend.
    .
    Version 2 was updated to work with GHC 8.4 and Diagrams 1.4 with the
    Rasterific backend.
    .
    Version 3 is updated to work with GHC 8.6 and Diagrams 1.4 with the
    Postscript backend.

executable wedged
  main-is:          Wedged.hs
  build-depends:    base >=4.7 && <4.14,
                    MonadRandom >=0.1 && <0.6,
                    array >=0.5 && <0.6,
                    containers >=0.5 && <0.7,
                    strict >=0.3 && <0.4,
                    colour >=2.3 && <2.4,
                    diagrams-lib >=1.4 && <1.5,
```

```
                    diagrams-postscript >=1.4 && <1.5
  default-language: Haskell2010
  other-extensions: FlexibleContexts

source-repository head
  type: git
  location: https://code.mathr.co.uk/wedged.git

source-repository this
  type: git
  location: https://code.mathr.co.uk/wedged.git
  tag: v3
```

5 Wedged.hs

```
-- Wedged (c) 2013,2015,2018 Claude Heiland-Allen <claude@mathr.co.uk> <https://mathr.co.uk>
-- Copyleft: This is a free work, you can copy, distribute, and modify it under
-- the terms of the Free Art License <http://artlibre.org/licence/lal/en/>

{-# LANGUAGE FlexibleContexts #-}

module Main (main) where

import           Control.Monad          (guard, liftM2)
import           Control.Monad.Random   (MonadRandom, runRand, getRandomR, newStdGen, StdGen)
import           Data.Complex           (Complex((:+)), magnitude, mkPolar)
import           Data.Function          (on)
import           Data.List              (group, groupBy, sortBy, nub, nubBy)
import           Data.Maybe             (mapMaybe, fromJust, listToMaybe)
import           Data.Ord               (comparing)
import           Data.Strict.Tuple      (Pair((:!:)))
import           System.Environment     (getArgs, withArgs)
import           System.Exit            (exitFailure)
import           System.IO              (hPutStrLn, stderr)
import           Data.Array.Unboxed     (UArray, bounds, inRange, ixmap, indices)
import qualified Data.Array.Unboxed     as U
import           Data.Map.Strict        (Map)
import qualified Data.Map.Strict        as M
import           Diagrams.Prelude
  hiding (inside, magnitude, appends, clamp, Colour, translate, place, render, e, D, N, P, unP, Empty, normalize)
import qualified Diagrams.Prelude       as D
import           Data.Colour.SRGB       (sRGB24)
import           Diagrams.Backend.Postscript.CmdLine (B, defaultMain)

type N = Int
type R = Double
type C = Complex R

data Colour = Red | Yellow | Green | Cyan | Magenta
  deriving (Eq, Ord, Show, Read)

type Label = Int
type Depth = Int
type Size  = Pair Int Int
```

```
40  type Coord = Pair Int Int
    type Grid  = UArray Size Int

    grid :: [[Cell]] -> Grid
    grid css = U.array ((0:!:0),(h1:!:w1))
45      [ ((y :!: x),munge c)
        | (y,cs) <- [0..h1] `zip` css
        , (x,c ) <- [0..w1] `zip` cs
        ]
      where
50      w1 = length (head css) - 1
        h1 = length css        - 1

    elems :: Grid -> [Cell]
    elems = map unmunge . U.elems
55
    (!) :: Grid -> Coord -> Cell
    a ! i = unmunge (a U.! i)

    (//) :: Grid -> [(Coord, Cell)] -> Grid
60  (//) a = (U.//) a . map (fmap munge)

    assocs :: Grid -> [(Coord, Cell)]
    assocs = map (fmap unmunge) . U.assocs

65  data Cell  = Empty | Blocked | Filled !Label !Colour
      deriving (Eq, Ord, Show)

    munge :: Cell -> Label
    munge Empty = -1
70  munge Blocked = -2
    munge (Filled l Red) = 2 + 16 * l
    munge (Filled l Yellow) = 3 + 16 * l
    munge (Filled l Green) = 4 + 16 * l
    munge (Filled l Cyan) = 5 + 16 * l
75  munge (Filled l Magenta) = 6 + 16 * l

    unmunge :: Label -> Cell
    unmunge (-1) = Empty
    unmunge (-2) = Blocked
80  unmunge n = case n `divMod` 16 of
      (1, 2) -> Filled l Red
      (1, 3) -> Filled l Yellow
      (1, 4) -> Filled l Green
      (1, 5) -> Filled l Cyan
85    (1, 6) -> Filled l Magenta
      x -> error $ "unmunge: " ++ show (n, x)

    isEmpty :: Cell -> Bool
    isEmpty Empty = True
90  isEmpty _ = False

    isBlocked :: Cell -> Bool
    isBlocked Blocked = True
    isBlocked _ = False
95
    isFilled :: Cell -> Bool
```

```
isFilled Filled{} = True
isFilled _ = False

colour :: Cell -> Maybe Colour
colour (Filled _ c) = Just c
colour _ = Nothing

label :: Cell -> Maybe Label
label (Filled l _) = Just l
label _ = Nothing

unsafeColour :: Cell -> Colour
unsafeColour (Filled _ c) = c
unsafeColour _ = error "unsafeColour"

data Piece = P{ pid :: !Int, unP :: !Grid } deriving (Show)
instance Eq Piece where p == q = pid p == pid q
instance Ord Piece where p `compare` q = pid p `compare` pid q

pieceColour :: Piece -> Colour
pieceColour = unsafeColour . (! (0 :!: 0)) . unP

colours :: [Colour]
colours = [Red, Yellow, Magenta, Green, Cyan]

rawPieces :: [Piece]
rawPieces
  = mapMaybe (fmap snd . normalize isFilled . P 0 . grid)
   . zipWith ccells colours . paras . lines $ pieceData

ccells :: Colour -> [String] -> [[Cell]]
ccells c hss = map (map (cell c)) hss

pieceData :: String
pieceData = "**\n**\n\n*--\n***\n\n-*-\n***\n\n-**\n**-\n\n****\n"

cell :: Colour -> Char -> Cell
cell c '*' = Filled 0 c
cell _ '-' = Empty
cell _ _ = error "cell"

paras :: [String] -> [[String]]
paras [] = []
paras ls = case break null ls of
  (p, ls') -> p : paras (drop 1 ls')

orientations :: [Piece -> Piece]
orientations =
    [ id
    , reverse' . transpose'
    , mapReverse' . transpose'
    , reverse' . mapReverse'
    , reverse'
    , mapReverse'
    , transpose'
    , reverse' . mapReverse' . transpose'
    ]
```

```
onP :: (Grid -> Grid) -> Piece -> Piece
onP f (P i g) = P i (f g)

reverse' :: Piece -> Piece
reverse' = onP vflip

mapReverse' :: Piece -> Piece
mapReverse' = onP hflip

transpose' :: Piece -> Piece
transpose' = onP dflip

vflip :: Grid -> Grid
vflip g =
  let bs@((y0:!:_),(h1:!:_)) = bounds g
      f (y :!: x) = (h1 - (y - y0) :!: x)
  in  ixmap bs f g

hflip :: Grid -> Grid
hflip g =
  let bs@((_ :!: x0),(_ :!: w1)) = bounds g
      f (y :!: x) = (y :!: w1 - (x - x0))
  in  ixmap bs f g

dflip :: Grid -> Grid
dflip g =
  let ((y0 :!: x0),(h1 :!: w1)) = bounds g
      f (y :!: x) = (x :!: y)
  in  ixmap ((x0 :!: y0),(w1 :!:h1)) f g

pieces :: [Piece]
pieces = zipWith P [0..] . nub . map unP . liftM2 o rawPieces $ orientations
  where o q@(P _ _) f = snd . fromJust . normalize isFilled $ f q

data Board = B
  { unB :: !Grid
  , topLeft_isEmpty :: !(Maybe Coord)
  , colour_counts :: !(Map Colour Int)
  }
  deriving (Eq, Ord, Show)

mkB :: Grid -> Board
mkB g = B
  { unB = g
  , topLeft_isEmpty = topLeft isEmpty g
  , colour_counts = M.fromList (colours `zip` repeat 0)
  }

rectangle :: Size -> Board
rectangle (h :!: w) = mkB $ U.listArray ((0 :!: 0),(h-1 :!: w-1)) (repeat (-1))

place :: Coord -> Label -> Piece -> Board -> [Board]
place yx l piece board
  | fits yx piece board = [blit yx l piece board]
  | otherwise = []
```

```
(==>) :: Bool -> Bool -> Bool
x ==> y = if x then y else True
infix 1 ==>

(=/>) :: Bool -> Bool -> Bool
x =/> y = if x then y else False
infix 1 =/>

surround :: Piece -> [Coord]
surround = (surrounds M.!)

surrounds :: Map Piece [Coord]
surrounds = M.fromList [(p, surround' p) | p <- pieces]

surround' :: Piece -> [Coord]
surround' (P _ piece) = nub
  [ vu
  | yx@(y :!: x) <- indices piece
  , isFilled (piece ! yx)
  , vu <- [(y-1 :!: x),(y+1 :!: x), (y :!: x-1), (y :!: x+1)]
  , inRange (bounds piece) vu ==> isEmpty (piece ! vu)
  ]

fits :: Coord -> Piece -> Board -> Bool
fits (y :!: x) p@(P _ piece) (B board _ cc)
  = inside bp bb &&
    cc M.! pc < hi &&
    and [ isEmpty (board ! (v+y :!: u+x))
        | vu@(v :!: u) <- indices piece
        , isFilled (piece ! vu) ] &&
    all distinct
        [ board ! yx
        | (v :!: u) <- surround p
        , let yx = (v+y :!: u+x)
        , inRange bb yx ] &&
    (pc == Cyan ==> case bp of
      ((0:!:0),(3:!:0)) -> not (blocked (y - 1 :!: x) || blocked (y + 4 :!: x))
      ((0:!:0),(0:!:3)) -> not (blocked (y :!: x - 1) || blocked (y :!: x + 4))
      _ -> error "fits")
  where
    bb@((y0 :!: x0), (h1 :!: w1)) = bounds board
    bp = bounds piece
    h = h1 - y0 + 1
    w = w1 - x0 + 1
    n :: Double
    n = fromIntegral (h * (w - 1)) / fromIntegral (4 * length colours)
    md = 4 * round n
    hi = md + 4
    pc = pieceColour p
    distinct = (Just pc /=) . colour
    blocked yx = inRange bb yx =/> isBlocked (board ! yx)
    inside ((ly :!: lx),(hy :!: hx)) ((lv :!: lu),(hv :!: hu))
      = lv <= (ly+y) && (hy+y) <= hv && lu <= (lx+x) && (hx+x) <= hu

blit :: Coord -> Label -> Piece -> Board -> Board
blit (y :!: x) l p@(P _ piece) (B board (Just (ty :!: tx)) cc) =
    B board' (topLeftFrom ty tx isEmpty board') cc'
```

```
  where
    cc' = M.adjust (4 +) (pieceColour p) cc
    board' = board // [ (yx, blit1 l (piece ! vu) (board ! yx))
                      | vu@(v :!: u) <- indices piece, let yx = (y + v :!: x + u
                        ↪ )
                      ]
blit _ _ _ _ = error "blit"

blit1 :: Label -> Cell -> Cell -> Cell
blit1 l (Filled _ c) Empty = Filled l c
blit1 _ Empty x = x
blit1 _ x y = error $ "blit1" ++ show (x, y)

topLeft :: (Cell -> Bool) -> Grid -> Maybe Coord
topLeft p a = listToMaybe [ i | i <- indices a, p $ a ! i ]

topLeftFrom :: Int -> Int -> (Cell -> Bool) -> Grid -> Maybe Coord
topLeftFrom ty tx p a = go ty tx
  where
    ((_ :!: x0),(h0 :!: w0)) = bounds a
    go y x
      | y > h0 = Nothing
      | x > w0 = go (y + 1) x0
      | p (a ! yx) = Just yx
      | otherwise = go y (x + 1)
      where yx = (y :!: x)

normalize :: (Cell -> Bool) -> Piece -> Maybe (Coord, Piece)
normalize p (P i piece) = do
  (y :!: x) <- topLeft p piece
  return ((y :!: x), translate (-y :!: -x) (P i piece))

translate :: Coord -> Piece -> Piece
translate (y :!: x) (P i g) = P i (ixmap bs (\(v :!: u) -> (v - y :!: u - x)) g)
  where
    ((y0 :!: x0),(h1 :!: w1)) = bounds g
    bs = ((y0 + y :!: x0 + x), (h1 + y :!: w1 + x))

fill :: Depth -> [Piece] -> Board -> [Board]
fill 0 _ board = do
  guard $ colourCounts board
  guard $ lineLengths board
  return board
fill d piecesm board = do
  Just yx <- return $ topLeft_isEmpty board
  piece <- piecesm
  board' <- place yx (d - 1) piece board
  guard $ diverse board'
  fill (d - 1) piecesm board'

colourCounts :: Board -> Bool
colourCounts b = all (lo <=) cs && any (== md) cs
  where
    cs = M.elems (colour_counts b)
    ((y0 :!: x0), (h1 :!: w1)) = bounds (unB b)
    h = h1 - y0 + 1
    w = w1 - x0 + 1
```

```
    n :: Double
    n = fromIntegral (h * (w - 1)) / fromIntegral (4 * length colours)
    lo = md - 4
    md = 4 * round n

lineLengths :: Board -> Bool
lineLengths (B g _ _) = all (<= l) . concatMap (map length . group) $ hs ++ vs
  where
    hs = [ [ g ! (y :!: x) == g ! (y+1 :!: x) | x <- [x0..w1] ] | y <- [y0 .. h1↙
        ↪ -1] ]
    vs = [ [ g ! (y :!: x) == g ! (y :!: x+1) | y <- [y0..h1] ] | x <- [x0 .. w1↙
        ↪ -1] ]
    ((y0 :!: x0), (h1 :!: w1)) = bounds g
    w = w1 - x0 + 1
    l = w - 2

depth :: Board -> Maybe Depth
depth g
  | 0 == n `mod` 4 = Just (n `div` 4)
  | otherwise = Nothing
  where
    n = length . filter isEmpty . elems . unB $ g

packings :: [Piece] -> Board -> [Board]
packings piecesm board = maybe [] (\d -> fill d piecesm board) (depth board)

blockings :: Board -> [Board]
blockings (B board _ _) =
    blockings' (x0 - 200) (x0 - 100) y0 m0 board
  where
    ((y0 :!: x0), (h1 :!: w1)) = bounds board
    h = h1 - y0 + 1
    w = w1 - x0 + 1
    m0 = M.fromList [ (x, n) | x <- [x0 .. w1] ]
    n = ((h - 1) `div` w) + 1
    blockings' x'' x' y m b
      | y > h1 = if all (< n) (M.elems m) then return (mkB b) else []
      | otherwise = do
          let a x = abs (x - x') > 2 && abs (x - x'') > 2
          x <- M.keys $ M.filterWithKey (\x n' -> a x && n' > 0) m
          let b' = b // [((y:!:x), Blocked)]
              m' = M.adjust (subtract 1) x m
          blockings' x' x (y + 1) m' b'

diverse :: Board -> Bool
diverse (B b k _) = case k of
    Nothing -> d (row h1) && all d cols
    Just (ty :!: _) | ty > y0 -> d (row (ty - 1))
    _ -> True
  where
    row y1 = [ colour $ b ! (y1 :!: x) | x <- [x0 .. w1] ]
    cols   = [ [ colour $ b ! (y :!: x) | y <- [y0 .. h1] ] | x <- [x0 .. w1] ]
    d = (5 <=) . length . nub
    ((y0 :!: x0), (h1 :!: w1)) = bounds b

main :: IO ()
main = do
```

```
  args <- getArgs
  case args of
    [sh,sw,ss,sd] -> do
      h <- readIO sh
      w <- readIO sw
      s <- readIO ss
      d <- readIO sd
      main' (s * d) (h :!: w)
    _ -> hPutStrLn stderr "usage: /path/to/wedged heightInCells widthInCells ↙
      ↳ cellSizeInches dotsPerInch" >> exitFailure

main' :: Double -> Size -> IO ()
main' cellSize s@(y :!: x)
  = mapM_ (uncurry (putDiagram w h)) . zip [0..] . map unB
  . concatMap (nubBy (equivalentBy ((==) `on` colour)) . packings pieces)
  . nubBy equivalent . blockings . rectangle
  $ s
  where
    w = round $ fromIntegral (x + 1) * cellSize
    h = round $ fromIntegral (y + 1) * cellSize

equivalent :: Board -> Board -> Bool
equivalent = equivalentBy (==)

equivalentBy :: (Cell -> Cell -> Bool) -> Board -> Board -> Bool
equivalentBy ceq (B a _ _) (B b _ _) =
    a `eq` b || a `eq` vflip b || a `eq` hflip b || a `eq` hflip (vflip b)
  where
    eq p q = bounds p == bounds q && and (zipWith ceq (elems p) (elems q))

putDiagram :: Int -> Int -> Int -> Grid -> IO ()
putDiagram w h n g = do
  withArgs ["-w", show w, "-h", show h, "-o", show3 n ++ ".eps"] $ do
    defaultMain . fst . render g =<< newStdGen
  where
    show3 i
      | i < 0 = show i
      | i < 10 = "00" ++ show i
      | i < 100 = "0" ++ show i
      | otherwise =      show i

render :: Grid -> StdGen -> (Diagram B, StdGen)
render g = runRand $ do
  cs <- mapM renderCells $ pieceCells g
  return $ withEnvelope' e (mconcat cs `atop` (e # lc white # fc white)) # ↙
    ↳ centerXY
  where
    e = fromVertices [ p2(fromIntegral $ xlo-1,fromIntegral $ ylo-1), p2(↙
      ↳ fromIntegral $ xlo-1,fromIntegral $ yhi+1), p2(fromIntegral $ xhi+1,↙
      ↳ fromIntegral $ yhi+1), p2(fromIntegral $ xhi+1,fromIntegral $ ylo-1) ]↙
      ↳  # closeTrail # (`at` p2(fromIntegral$xlo-1,fromIntegral$ylo-1)) # ↙
      ↳ stroke
    withEnvelope' a b = withEnvelope (a `asTypeOf` b) b
    ((ylo:!:xlo),(yhi:!:xhi)) = bounds g

pieceCells :: Grid -> [[(Coord, Cell)]]
pieceCells
```

```
430     = map (sortBy (comparing fst))
        . groupBy ((==) `on` (label . snd))
        . sortBy (comparing (label . snd))
        . assocs

435 renderCells :: (Functor m, MonadRandom m) => [(Coord, Cell)] -> m (Diagram B)
    renderCells ((j :!: i, Filled _ Red):_) =
        (draw True (2^wdepth) (rgb Red) . (:[])) `fmap` appendsM [ w a b, w b c, w c↙
            ↳ d, w d a ]
      where
        wdepth :: N
440     wdepth = 4
        w = wobble wdepth
        a = fromIntegral i :+ fromIntegral j
        b = fromIntegral i :+ fromIntegral (j + 1)
        c = fromIntegral (i + 1) :+ fromIntegral (j + 1)
445     d = fromIntegral (i + 1) :+ fromIntegral j
    renderCells [(j0:!:i0, Filled _ Yellow),(j1:!:i1,_),(j2:!:i2,_),(j3:!:i3,_)] =
        (draw False (2^wdepth) (rgb Yellow) . (:[])) `fmap` appendsM ws
      where
        wdepth :: N
450     wdepth = 4
        w = wobble wdepth
        a = fromIntegral i0 :+ fromIntegral j0
        b = fromIntegral i1 :+ fromIntegral j1
        c = fromIntegral i2 :+ fromIntegral j2
455     d = fromIntegral i3 :+ fromIntegral j3
        ws = case (j1 - j0, i1 - i0, j2 - j0, i2 - i0, j3 - j0, i3 - i0) of
          (0, 1, 0, 2, 1, 2) -> {-   --,  -} [ w a b, w b c, w c d ]
          (1, 0, 2,-1, 2, 0) -> {-   ,|   -} [ w a b, w b d, w d c ]
          (1, 0, 1, 1, 1, 2) -> {-   '--  -} [ w a b, w b c, w c d ]
460       (0, 1, 1, 0, 2, 0) -> {-   |'   -} [ w b a, w a c, w c d ]
          (0, 1, 0, 2, 1, 0) -> {-   ,--  -} [ w d a, w a b, w b c ]
          (0, 1, 1, 1, 2, 1) -> {-   '|   -} [ w a b, w b c, w c d ]
          (1,-2, 1,-1, 1, 0) -> {- --'    -} [ w b c, w c d, w d a ]
          (1, 0, 2, 0, 2, 1) -> {-   |,   -} [ w a b, w b c, w c d ]
465       x -> error $ "yellow" ++ show x
    renderCells [(j0:!:i0, Filled _ Green),(j1:!:i1,_),(j2:!:i2,_),(j3:!:i3,_)] =
        (draw False (2^wdepth) (rgb Green) . (:[])) `fmap` appendsM ws
      where
        wdepth :: N
470     wdepth = 4
        w = wobble wdepth
        a = fromIntegral i0 :+ fromIntegral j0
        b = fromIntegral i1 :+ fromIntegral j1
        c = fromIntegral i2 :+ fromIntegral j2
475     d = fromIntegral i3 :+ fromIntegral j3
        ws = case (j1 - j0, i1 - i0, j2 - j0, i2 - i0, j3 - j0, i3 - i0) of
          (0, 1, 1,-1, 1, 0) -> {-  _|'   -} [ w c d, w d a, w a b ]
          (0, 1, 1, 1, 1, 2) -> {-   '|_  -} [ w a b, w b c, w c d ]
          (1, 0, 1, 1, 2, 1) -> {-   ',   -} [ w a b, w b c, w c d ]
480       (1,-1, 1, 0, 2,-1) -> {-   ,'   -} [ w a c, w c b, w b d ]
          x -> error $ "green" ++ show x
    renderCells [(j0:!:i0, Filled _ Cyan),(j1:!:i1,_),(j2:!:i2,_),(j3:!:i3,_)] =
        (draw False (2^wdepth) (rgb Cyan) . (:[])) `fmap` appendsM [ w a b, w b c, w↙
            ↳ c d ]
      where
```

```
485       wdepth :: N
          wdepth = 4
          w = wobble wdepth
          a = fromIntegral i0 :+ fromIntegral j0
          b = fromIntegral i1 :+ fromIntegral j1
490       c = fromIntegral i2 :+ fromIntegral j2
          d = fromIntegral i3 :+ fromIntegral j3
      renderCells [(j0 :!: i0, Filled _ Magenta),(j1 :!: i1, _),(j2 :!: i2, _),(j3 :!: i3, _)] =
          draw False (2^wdepth) (rgb Magenta) `fmap` mapM appendsM wss
        where
495       wdepth :: N
          wdepth = 4
          w = wobble wdepth
          a = fromIntegral i0 :+ fromIntegral j0
          b = fromIntegral i1 :+ fromIntegral j1
500       c = fromIntegral i2 :+ fromIntegral j2
          d = fromIntegral i3 :+ fromIntegral j3
          wss = case (j1 - j0, i1 - i0, j2 - j0, i2 - i0, j3 - j0, i3 - i0) of
            (1,-1, 1, 0, 1, 1) ->  {-  _|_   -} [ [ w a c ], [ w b c, w c d ] ]
            (0, 1, 0, 2, 1, 1) ->  {-   -|-  -} [ [ w b d ], [ w a b, w b c ] ]
505         (1, 0, 1, 1, 2, 0) ->  {-   |-   -} [ [ w b c ], [ w a b, w b d ] ]
            (1,-1, 1, 0, 2, 0) ->  {-  -|    -} [ [ w b c ], [ w a c, w c d ] ]
            x -> error $ "magenta" ++ show x
      renderCells _ = return mempty

510   perturbMidpoint :: MonadRandom m => C -> C -> m C
      perturbMidpoint p q = do
        let m0 = (p + q) / 2
            r1 = magnitude (p - q) / 16
        t <- getRandomR (-pi, pi)
515     r <- getRandomR (0, r1)
        return $! m0 + mkPolar r t

      append :: (R -> t) -> (R -> t) -> R -> t
      append f g t
520     | t < 0.5   = f $! 2 * t
        | otherwise = g $! 2 * t - 1

      appends :: [(R -> t)] -> R -> t
      appends fs t = fs !! ti $ tx
525     where
          l = length fs
          t' = t * fromIntegral l
          ti = clamp (floor t') 0 (l - 1)
          tx = t' - fromIntegral ti
530
      appendsM :: (Functor m, Monad m) => [m (R -> t)] -> m (R -> t)
      appendsM fs = appends `fmap` sequence fs

      wobble :: MonadRandom m => N -> C -> C -> m (R -> C)
535   wobble 0 p q = return $ lint p q
      wobble n p q = do
        r <- perturbMidpoint p q
        pr <- wobble (n - 1) p r
        rq <- wobble (n - 1) r q
540     return $ pr `append` rq
```

```
lint :: C -> C -> R -> C
lint p q t = c (1 - t) * p + c t * q where c r = r :+ 0

clamp :: Ord t => t -> t -> t -> t
clamp x lo hi = lo `max` x `min` hi

draw :: Bool -> N -> D.Colour R -> [(R -> C)] -> Diagram B
draw cl m c fs = (plot 0.2 # lc c `atop` plot 0.3 # lc black) # lineCap ↙
    ↳ LineCapRound # lineJoin LineJoinRound
  where m' :: R
        m' = 1 / fromIntegral m
        ps :: [Path V2 R]
        ps = [ cubicSpline cl
          [ p2(x,y)
          | i <- [0 .. if cl then m - 1 else m]
          , let t = fromIntegral i * m'
          , let x:+y = f t
          ] | f <- fs ]
        plot k = strokePath (mconcat ps) # lwL k

rgb :: Colour -> D.Colour R
rgb Red     = sRGB24 205  63 125
rgb Yellow  = sRGB24 213 135  54
rgb Green   = sRGB24  58 110  70
rgb Cyan    = sRGB24  56 138 170
rgb Magenta = sRGB24 100  70 124
```