gruff

Claude Heiland-Allen

2011–2015

Contents

1 data/colourize.frag . . . 3
2 data/icon.png . . . 5
3 data/merge.frag . . . 5
4 data/minimal.frag . . . 5
5 extra/cache.frag . . . 5
6 extra/CacheView.hs . . . 7
7 extra/combine.frag . . . 13
8 extra/compute.cc . . . 13
9 extra/expblend.frag . . . 17
10 extra/Exponential2.hs . . . 17
11 extra/Exponential.hs . . . 21
12 extra/feature-database.hs . . . 25
13 extra/FixedPrecision.hs . . . 27
14 extra/gruff-cache-stats.sh . . . 28
15 extra/gruff-tile-score.c . . . 28
16 extra/Makefile . . . 28
17 extra/mp_real.h . . . 28
18 extra/MuAtom.hs . . . 30
19 extra/Number.hs . . . 34
20 extra/Poincare.hs . . . 34
21 extra/rts.c . . . 36
22 extra/SoftFloat.hs . . . 36
23 extra/test.hs . . . 39
24 extra/TileFeatures.hs . . . 41
25 extra/UnComplex.hs . . . 42
26 .gitignore . . . 44
27 gruff.cabal . . . 44
28 LICENSE . . . 46
29 Setup.hs . . . 52
30 src/Browser.hs . . . 52
31 src/Compute.hs . . . 66
32 src/Fractal/GRUFF.hs . . . 70
33 src/GLUTGtk.hs . . . 73
34 src/gruff.hs . . . 74
35 src/Interact.hs . . . 83
36 src/Logger.hs . . . 84
37 src/Progress.hs . . . 85
38 src/QuadTree.hs . . . 87
39 src/Shader.hs . . . 90
40 src/Snapshot.hs . . . 90
41 src/StatusDialog.hs . . . 91
42 src/Tile.hs . . . 92

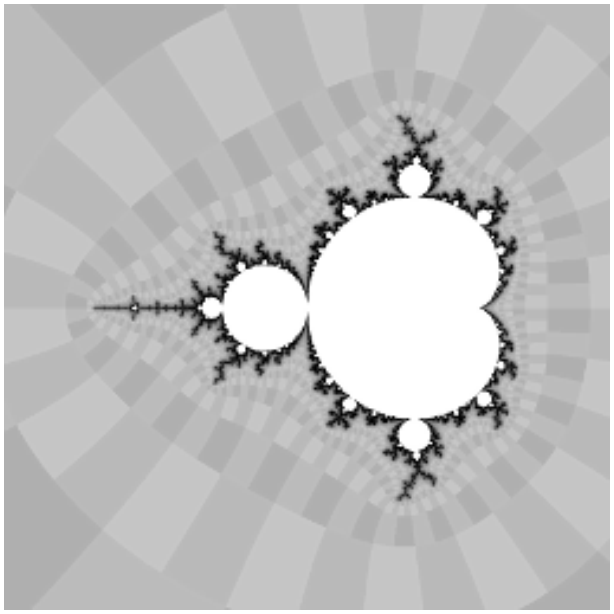
43 src/TileShake.hs 95
44 src/Utils.hs 97
45 src/View.hs 97
46 TODO 100

1 data/colourize.frag

```
uniform sampler2D de;
uniform sampler2D it;
uniform sampler2D tt;

uniform float hshift;
uniform float hscale;

float lin_interp(float x, float domain_low, float domain_hi, float range_low, ↙
    ↳ float range_hi) {
  if ((x >= domain_low) && (x <= domain_hi)) {
    x = (x - domain_low) / (domain_hi - domain_low);
    x = range_low + x * (range_hi - range_low);
  }
  return x;
}

float pvp_adjust_3(float x) {
  // red
  x = lin_interp(x, 0.00, 0.125, -0.050, 0.090);
  // orange
  x = lin_interp(x, 0.125, 0.25,  0.090, 0.167);
  // yellow
  x = lin_interp(x, 0.25, 0.375,  0.167, 0.253);
  // chartreuse
  x = lin_interp(x, 0.375, 0.50,  0.253, 0.383);
  // green
  x = lin_interp(x, 0.50, 0.625,  0.383, 0.500);
  // teal
  x = lin_interp(x, 0.625, 0.75,  0.500, 0.667);
  // blue
  x = lin_interp(x, 0.75, 0.875,  0.667, 0.800);
  // purple
  x = lin_interp(x, 0.875, 1.00,  0.800, 0.950);
  return(x);
}

vec3 hsv2rgb(float h, float s, float v)
{
  float i, f, p, q, t, r, g, b;
  if (s == 0.0) {
    // Ignore hue
    r = v;
    g = v;
    b = v;
  } else {
    /* Apply physiological mapping: red, yellow, green and blue should
       be equidistant. */
    h = pvp_adjust_3(h);
    h = h - floor(h);
```

```
    h = h * 6.0;
    i = floor(h);
    f = h - i;
    p = v*(1.0 - s);  // The "low-flat" curve
    q = v*(1.0 - (s*f));  // The "falling" curve
    t = v*(1.0 - (s*(1.0 - f)));  // The "rising" curve
    if (i < 1.0) { r = v; g = t; b = p; } else
    if (i < 2.0) { r = q; g = v; b = p; } else
    if (i < 3.0) { r = p; g = v; b = t; } else
    if (i < 4.0) { r = p; g = q; b = v; } else
    if (i < 5.0) { r = t; g = p; b = v; } else
                 { r = v; g = p; b = q; }
  }
  return vec3(r, g, b);
}

void main(void) {
  float d = texture2D(de, gl_TexCoord[0].st).x;
  float i = texture2D(it, gl_TexCoord[0].st).x;
  float t = texture2D(tt, gl_TexCoord[0].st).x;
  vec4 c = vec4(1.0, 0.0, 0.0, 1.0);
  if (i > 0.0) {
/*
    float h = hscale * log(i) + hshift;
    h -= floor(h);
    float s;
    float v;
    if (t           > 0.0) { s = 0.25; } else { s = 0.15; }
    i /= 2.0;
    if (i - floor(i) < 0.5) { v = 0.95; } else { v = 1.00; }
    c.rgb = hsv2rgb(h, s, v);
    c.rgb *= clamp(0.5 + 0.25 * log(d), 0.0, 1.0);
*/
    c.rgb = vec3(clamp(0.5 + 0.25 * log(d), 0.0, 1.0));
  }
  gl_FragColor = c;
}
```

2 data/icon.png



3 data/merge.frag

```
uniform sampler2D sheet0;
uniform sampler2D sheet1;
uniform float blend;

void main() {
  vec2 p = gl_TexCoord[0].xy;
  gl_FragColor = mix(texture2D(sheet1, p), texture2D(sheet0, p), blend);
}
```

4 data/minimal.frag

```
uniform sampler2D de;

uniform vec3 interior;
uniform vec3 border;
uniform vec3 exterior;

void main(void) {
  float d = texture2D(de, gl_TexCoord[0].st).x;
  vec4 c = vec4(interior, 1.0);
  if (d > 0.0) {
    float k = pow(clamp(0.5 + 0.25 * log2(4.0 * d), 0.0, 1.0), 2.0);
    c.rgb = mix(border, exterior, k);
  }
  gl_FragColor = c;
}
```

5 extra/cache.frag

```
uniform sampler2D tex;

float lin_interp(float x, float domain_low, float domain_hi, float range_low, ↙
    ↳ float range_hi) {
  if ((x >= domain_low) && (x <= domain_hi)) {
    x = (x - domain_low) / (domain_hi - domain_low);
    x = range_low + x * (range_hi - range_low);
  }
  return x;
}

float pvp_adjust_3(float x) {
  // red
  x = lin_interp(x, 0.00, 0.125, -0.050, 0.090);
  // orange
  x = lin_interp(x, 0.125, 0.25,  0.090, 0.167);
  // yellow
  x = lin_interp(x, 0.25, 0.375,  0.167, 0.253);
  // chartreuse
  x = lin_interp(x, 0.375, 0.50,  0.253, 0.383);
  // green
  x = lin_interp(x, 0.50, 0.625,  0.383, 0.500);
  // teal
  x = lin_interp(x, 0.625, 0.75,  0.500, 0.667);
  // blue
  x = lin_interp(x, 0.75, 0.875,  0.667, 0.800);
  // purple
  x = lin_interp(x, 0.875, 1.00,  0.800, 0.950);
  return(x);
}

vec3 hsv2rgb(float h, float s, float v)
{
  float i, f, p, q, t, r, g, b;
  if (s == 0.0) {
    // Ignore hue
    r = v;
    g = v;
    b = v;
  } else {
    /* Apply physiological mapping: red, yellow, green and blue should
       be equidistant. */
    h = pvp_adjust_3(h);
    h = h - floor(h);
    h = h * 6.0;
    i = floor(h);
    f = h - i;
    p = v*(1.0 - s);  // The "low-flat" curve
    q = v*(1.0 - (s*f));  // The "falling" curve
    t = v*(1.0 - (s*(1.0 - f)));  // The "rising" curve
    if (i < 1.0) { r = v; g = t; b = p; } else
    if (i < 2.0) { r = q; g = v; b = p; } else
    if (i < 3.0) { r = p; g = v; b = t; } else
    if (i < 4.0) { r = p; g = q; b = v; } else
    if (i < 5.0) { r = t; g = p; b = v; } else
                 { r = v; g = p; b = q; }
  }
```

```
  return vec3(r, g, b);
}

void main(void) {
  float n = texture2D(tex, gl_TexCoord[0].st).x;
  float h = n;
  float s = 1.0;
  float v = clamp(0.0, 1.0, 0.25 + log(h + 1.0));
  gl_FragColor = vec4(hsv2rgb(h, s, v), 1.0);
}
```

6 extra/CacheView.hs

```
module CacheView (cInitialize) where

import Control.Concurrent (forkIO)
import Control.Monad (forM_, liftM2, when)
import Data.IORef (IORef, newIORef, readIORef, atomicModifyIORef)
import qualified Data.Map as M
import Data.Map (Map)
import Data.Maybe (catMaybes, mapMaybe)
import System.FilePath ((</>), dropExtension)

import Foreign (alloca, peek, nullPtr)
import Graphics.Rendering.OpenGL hiding (Position, Size)
import qualified Graphics.Rendering.OpenGL as GL
import Graphics.Rendering.OpenGL.Raw
  ( glGenFramebuffers, glBindFramebuffer, glFramebufferTexture2D
  , gl_FRAMEBUFFER, gl_COLOR_ATTACHMENT0, gl_TEXTURE_2D
  , glTexImage2D, gl_R32F, gl_LUMINANCE, gl_FLOAT
  , gl_FALSE, glClampColor, gl_CLAMP_VERTEX_COLOR, gl_CLAMP_READ_COLOR
  , gl_CLAMP_FRAGMENT_COLOR )
import Graphics.UI.Gtk hiding (Region, Size)

import Number
import GLUTGtk
import Shader (shader)
import QuadTree (Child(..), unsafeName, Quad, Square(..), square, child, root, ↙
    ↳ outside, Region(..))
--import Tile (rootSquare)
import Utils (getFilesRecursive)

import Paths_gruff (getDataFileName)

rootSquare :: Square
rootSquare = Square{ squareSize = 8, squareWest = -4, squareNorth = -4 }

names :: Map Char [Child]
names = M.fromList names1 `M.union` M.fromList names2
  where
    names1 = [ (unsafeName [i], [i]) | i <- [minBound..maxBound] ]
    names2 = [ (unsafeName [i,j], [i,j]) | i <- [minBound..maxBound], j <- [↙
        ↳ minBound..maxBound] ]

fromPath :: ([FilePath], FilePath) -> Maybe [Child]
fromPath (ps, f) =
  let s = concat ps ++ dropExtension f
```

```
      p = concat $ mapMaybe (`M.lookup` names) s
  in  if all (`M.member` names) s then Just p else Nothing

cDisplay :: IORef GruffCache -> IO ()
cDisplay cR = do
  c <- readIORef cR
  let Size w h = cSize c
      TextureObject tex = cTex c
  when (cRecalc c) $ do
    viewport $= (GL.Position 0 0, GL.Size (fromIntegral w) (fromIntegral h))
    loadIdentity
    ortho2D 0 (fromIntegral w) (fromIntegral h) 0
    withFBO (cFBO c) tex $ do
      clearColor $= Color4 0 0 0 1
      clear [ColorBuffer]
      let toPixel' = toPixel c
          qs = cQuads c
          zm = fromIntegral . snd . head $ qs
      withBlend (One, One) $ renderPrimitive Quads $ forM_ qs $ \(q, z) -> do
        let sq = square rootSquare q
            s = squareSize sq `max` cSquareSize c
            (x0, y0) = toPixel' (squareWest sq) (squareNorth sq)
            (x1, y1) = toPixel' (squareWest sq + s) (squareNorth sq + s)
            v x y = vertex $ Vertex2 (fromRational x :: GLdouble) (fromRational y :: GLdouble)
            k = fromIntegral z / zm :: GLdouble
        color $ Color3 k k k
        v x0 y1 >> v x0 y0 >> v x1 y0 >> v x1 y1
    atomicModifyIORef cR $ \c' -> (c'{ cRecalc = False }, ())
  -- draw texture
  loadIdentity
  ortho2D 0 1 1 0
  textureBinding Texture2D $= Just (cTex c)
  currentProgram $= Just (cProg c)
  lt <- GL.get $ uniformLocation (cProg c) "tex"
  uniform lt $= TexCoord1 (0 :: GLint)
  renderPrimitive Quads $ do
    let Size tw th = cTexSize c
    let tx = fromIntegral w / fromIntegral tw
        ty = fromIntegral h / fromIntegral th
        v :: GLdouble -> GLdouble -> IO ()
        v x y = do
          texCoord $ TexCoord2 (tx * x) (ty * (1-y))
          vertex $ Vertex2 x y
    v 0 1 >> v 0 0 >> v 1 0 >> v 1 1
  currentProgram $= Nothing
  textureBinding Texture2D $= Nothing

withFBO :: GLuint -> GLuint -> IO () -> IO ()
withFBO fbo tex act = do
  glBindFramebuffer gl_FRAMEBUFFER fbo
  glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D tex 0
  act
  glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D 0 0
  glBindFramebuffer gl_FRAMEBUFFER 0

withBlend :: (BlendingFactor, BlendingFactor) -> IO a -> IO a
```

```
withBlend bf act = do
  ob <- GL.get blend
  obf <- GL.get blendFunc
  blend $= Enabled
  blendFunc $= bf
  r <- act
  blendFunc $= obf
  blend $= ob
  return r

data GruffCache = GruffCache
  { cSize       :: Size
  , cCenter     :: (Rational, Rational)
  , cLevel      :: Int
  , cCacheDir   :: FilePath
  , cGL         :: GLUTGtk
  , cProg       :: Program
  , cTex        :: TextureObject
  , cTexSize    :: Size
  , cFBO        :: GLuint
  , cRecalc     :: Bool
  , cQTree      :: QTree Integer
  }

fromPixel :: GruffCache -> Int -> Int -> (Rational, Rational)
fromPixel c x y = (x', y')
  where
    Size w h = cSize c
    a = fromIntegral w / fromIntegral h
    (cx, cy) = cCenter c
    r = 8 / 2 ^ cLevel c * fromIntegral h / (2 * fromIntegral tileSize)
    x' = cx + r * (fromIntegral x / fromIntegral w - 0.5) * a
    y' = cy + r * (fromIntegral y / fromIntegral h - 0.5)

toPixel :: GruffCache -> Rational -> Rational -> (Rational, Rational)
toPixel c = f
  where
    f x y = {-# SCC "toPixel'" #-} (x', y')
      where
        x' = ((x - cx) / r' + 0.5) * w'
        y' = ((y - cy) / r  + 0.5) * h'
    Size w h = cSize c
    w' = fromIntegral w
    h' = fromIntegral h
    a = w' / h'
    (cx, cy) = cCenter c
    r = 8 / 2 ^ cLevel c * h' / (2 * fromIntegral tileSize)
    r' = a * r

tileSize :: Int
tileSize = 256

cRealize :: IORef GruffCache -> IO ()
cRealize cR = do
  f <- getDataFileName "cache.frag"
  prog <- shader Nothing (Just f)
  drawBuffer $= BackBuffers
```

```
    glClampColor gl_CLAMP_VERTEX_COLOR gl_FALSE
    glClampColor gl_CLAMP_READ_COLOR gl_FALSE
    glClampColor gl_CLAMP_FRAGMENT_COLOR gl_FALSE
    [tex] <- genObjectNames 1
    texture Texture2D $= Enabled
    textureBinding Texture2D $= Just tex
    textureFilter Texture2D $= ((Nearest, Nothing), Nearest)
    textureWrapMode Texture2D S $= (Repeated, ClampToEdge)
    textureWrapMode Texture2D T $= (Repeated, ClampToEdge)
    textureBinding Texture2D $= Nothing
    texture Texture2D $= Disabled
    fbo <- alloca $ \p -> glGenFramebuffers 1 p >> peek p
    atomicModifyIORef cR $ \c -> (c{ cProg = prog, cFBO = fbo, cTex = tex }, ())
    c <- readIORef cR
    cReshape cR (cSize c)

cSquareSize :: GruffCache -> Rational
cSquareSize c = squareSize rootSquare / 2 ^ cLevel c / (2 * fromIntegral ↙
    ↳ tileSize)

cInitialize :: GLUTGtk -> Pixbuf -> FilePath
            -> IO ( Window
                  , Maybe R -> Maybe R -> Maybe R -> IO ()
                  , IO () -> IO ()
                  )
cInitialize gl' icon cacheDir' = do
  -- image window
  cw <- windowNew
  let Size defW defH = defSize
  windowSetDefaultSize cw defW defH
  set cw
    [ containerBorderWidth := 0
    , containerChild := widget gl'
    , windowIcon  := Just icon
    , windowTitle := "gruff cache"
    ]
  cR <- newIORef GruffCache
    { cCenter = (0, 0)
    , cLevel = 0
    , cSize = defSize
    , cQTree = QTree 0 Nothing Nothing Nothing Nothing
    , cCacheDir = cacheDir'
    , cGL = gl'
    , cProg = undefined
    , cFBO = 0
    , cTex = TextureObject 0
    , cTexSize = roundUpSize defSize
    , cRecalc = False
    }
  realizeCallback gl' $= cRealize cR
  reshapeCallback gl' $= cReshape cR
  displayCallback gl' $= cDisplay cR
  keyboardMouseCallback gl' $= cMouse cR
  let cUpdateCoords mre mim mz = do
        case mz of
          Just z | z > 0 -> atomicModifyIORef cR $ \s ->
              ( s { cLevel = max 0 . floor . negate . logBase 2 $ z }, () )
```

```
            _ -> return ()
          case liftM2 (,) mre mim of
            Just (r, i) -> atomicModifyIORef cR $ \s ->
                ( s { cCenter = (toRational' (cLevel s) r, negate $ toRational' (↙
                    ↳ cLevel s) i) }, () )
            Nothing -> return ()
          cUpdate cR True
        cInitializeLlate aExit = do
          _ <- cw `onDestroy` aExit
          _ <- forkIO $ cRescan cR
          return ()
    return (cw, cUpdateCoords, cInitializeLate)

cRescan :: IORef GruffCache -> IO ()
cRescan cR = do
  c' <- readIORef cR
  fs <- getFilesRecursive (cCacheDir c' </> ".")
  let q = depthQ . treeQ .  mapMaybe fromPath $ fs
  atomicModifyIORef cR $ \c -> (c{ cQTree = q }, ())
  postGUISync (cUpdate cR True)

defSize :: Size
defSize = Size 320 200

cUpdate :: IORef GruffCache -> Bool -> IO ()
cUpdate cR rec = do
  c' <- readIORef cR
  atomicModifyIORef cR $ \c -> (c{ cRecalc = cRecalc c || rec }, ())
  postRedisplay (cGL c')

cReshape :: IORef GruffCache -> Size -> IO ()
cReshape iR size' = do
  c <- readIORef iR
  let tsize@(Size tw th) = roundUpSize size'
  textureBinding Texture2D $= Just (cTex c)
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_R32F)
    (fromIntegral tw) (fromIntegral th) 0 gl_LUMINANCE gl_FLOAT nullPtr
  textureBinding Texture2D $= Nothing
  atomicModifyIORef iR $ \s -> (s{ cSize = size', cTexSize = tsize }, ())
  cUpdate iR True

cMouse :: IORef GruffCache -> Key -> KeyState -> [Modifier] -> Position -> IO ()
cMouse sR (MouseButton LeftButton  ) Down _ (Position x y) = do
  atomicModifyIORef sR $ \s -> let level' = (cLevel s + 1) in
    ( s{ cCenter = fromPixel s (round x) (round y), cLevel = level' }
    , ())
  cUpdate sR True
cMouse sR (MouseButton MiddleButton) Down _ (Position x y) = do
  atomicModifyIORef sR $ \s ->
    ( s{ cCenter = fromPixel s (round x) (round y) }
    , ())
  cUpdate sR True
cMouse sR (MouseButton RightButton ) Down _ (Position x y) = do
  atomicModifyIORef sR $ \s -> let level' = (cLevel s - 1) `max` 0 in
    ( s{ cCenter = fromPixel s (round x) (round y), cLevel = level' }
    , ())
  cUpdate sR True
```

```
cMouse _ _ _ _ _ = return ()

roundUpSize :: Size -> Size
roundUpSize (Size w h) = Size (roundUp w) (roundUp h)

roundUp :: Int -> Int
roundUp x = head . filter (>= x) . iterate (2*) $ 1

data QTree a = QTree a (Maybe (QTree a)) (Maybe (QTree a)) (Maybe (QTree a)) (Maybe (QTree a))

payloadQ :: QTree a -> a
payloadQ (QTree w _ _ _ _) = w

childQ :: Child -> QTree a -> Maybe (QTree a)
childQ NorthWest (QTree _ a _ _ _) = a
childQ NorthEast (QTree _ _ b _ _) = b
childQ SouthWest (QTree _ _ _ c _) = c
childQ SouthEast (QTree _ _ _ _ d) = d

emptyQ :: QTree Bool
emptyQ = QTree False Nothing Nothing Nothing Nothing

insertQ :: [Child] -> QTree Bool -> QTree Bool
insertQ [] (QTree _ a b c d) = QTree True a b c d
insertQ (x:xs) (QTree w a b c d) = case x of
  NorthWest -> case a of
    Nothing -> QTree w (Just $ insertQ xs emptyQ) b c d
    Just a' -> QTree w (Just $ insertQ xs a'    ) b c d
  NorthEast -> case b of
    Nothing -> QTree w a (Just $ insertQ xs emptyQ) c d
    Just b' -> QTree w a (Just $ insertQ xs b'    ) c d
  SouthWest -> case c of
    Nothing -> QTree w a b (Just $ insertQ xs emptyQ) d
    Just c' -> QTree w a b (Just $ insertQ xs c'    ) d
  SouthEast -> case d of
    Nothing -> QTree w a b c (Just $ insertQ xs emptyQ)
    Just d' -> QTree w a b c (Just $ insertQ xs d'    )

treeQ :: [[Child]] -> QTree Bool
treeQ = foldr insertQ emptyQ

{-
sizeQ :: QTree Bool -> QTree Integer
sizeQ (QTree w a b c d) = QTree n a' b' c' d'
  where
    s@[a', b', c', d'] = fmap sizeQ `map` [a, b, c, d]
    n = (if w then 1 else 0) + (sum . map payloadQ . catMaybes) s
-}

depthQ :: QTree a -> QTree Integer
depthQ = depthQ' 0
  where
    depthQ' n (QTree _ Nothing Nothing Nothing Nothing) = QTree n Nothing Nothing Nothing Nothing
    depthQ' n (QTree _ a b c d) = QTree m a' b' c' d'
      where
```

```
        s@[a', b', c', d'] = fmap (depthQ' (n + 1)) `map` [a, b, c, d]
        m = maximum . map payloadQ . catMaybes $ s

pruneQ :: Int -> QTree a -> QTree a
pruneQ 0 (QTree w _ _ _ _) = QTree w Nothing Nothing Nothing Nothing
pruneQ n (QTree w a b c d) = QTree w a' b' c' d'
  where
    [a', b', c', d'] = fmap (pruneQ (n - 1)) `map` [a, b, c, d]

cQuads :: GruffCache -> [(Quad, Integer)]
cQuads c = map (fmap payloadQ) . concat . quadsQ view . pruneQ (cLevel c + 12) . cQTree $ c
  where
    Size w h = cSize c
    ((bw,bn),(be,bs)) =
      ( fromPixel c 0 0
      , fromPixel c (fromIntegral w) (fromIntegral h) )
    view = Region
      { regionWest = bw, regionNorth = bn
      , regionEast = be, regionSouth = bs }

quadsQ :: Region -> QTree a -> [[(Quad, QTree a)]]
quadsQ view q = takeWhile (not . null) . iterate (filter keep . children') $ [(root, q)]
  where
    keep = not . outside view . square rootSquare . fst
    children' = catMaybes . liftM2 child' [minBound .. maxBound]
    child' c (quad, qtree) = (,) (child c quad) `fmap` childQ c qtree
```

7 extra/combine.frag

```
uniform sampler2D red;
uniform sampler2D green;
uniform sampler2D blue;

void main(void) {
  float r = texture2D(red,   gl_TexCoord[0].st).x;
  float g = texture2D(green, gl_TexCoord[0].st).x;
  float b = texture2D(blue,  gl_TexCoord[0].st).x;
  gl_FragColor = vec4(r, g, b, 1.0);
}
```

8 extra/compute.cc

```
#include <cmath>
#include <stdint.h>
#include <stdlib.h>

#include <qd/fpu.h>
#include <qd/dd_real.h>
#include <qd/dd_inline.h>
#include <qd/qd_real.h>
#include <qd/qd_inline.h>

#include <new>
using namespace std;
```

```
#ifdef HAVE_MPFR
#include "mp_real.h"
#endif

#define likely(x)   __builtin_expect((x),1)
#define unlikely(x) __builtin_expect((x),0)

static inline float to_float(float x) { return x; }
static inline float to_float(double x) { return x; }
static inline float to_float(dd_real x) { return to_double(x); }
static inline float to_float(qd_real x) { return to_double(x); }

static inline float  sqr(float  x) { return x * x; }
static inline double sqr(double x) { return x * x; }

template <typename T>
struct iter {
  T zx, zy, dx, dy; int32_t n; int16_t i, j;
};

template <typename T>
static int compute(int *stop, float *out_n, float *out_d, float *out_a, T in_cx, T in_cy, int in_level, int in_iters) {
  unsigned int fpu;
  fpu_fix_start(&fpu);
  struct iter<T> *iters[2];
  iters[0] = new (nothrow) iter<T>[256 * 256];
  iters[1] = new (nothrow) iter<T>[256 * 256];
  int active[2];
  active[0] = 0;
  active[1] = 0;
  int from = 0;
  int to = 1;
  int max_iters = -1;
  { /* initialize border */
    struct iter<T> *it = iters[from];
    int n = active[from];
    { /* corners */
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i =   0; it->j =   0; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i = 255; it->j =   0; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i =   0; it->j = 255; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i = 255; it->j = 255; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
    }
    for (int i = 1; i < 255; ++i) { /* edges */
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i =   0; it->j =   i; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i = 255; it->j =   i; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i =   i; it->j =   0; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
      it->zx = it->zy = it->dx = it->dy = it->n = 0; it->i =   i; it->j = 255; out_n[256 * ((int)it->j) + it->i] = -1; it++; n++;
```

```
    }
    active[from] = n;
  }
  const T scale = T(1.0) / (T(32.0) * pow(T(2.0), T(in_level)));
  int progress = 1;
  int progress2 = 1;
  int progressed = 0;
  int min_iters = 0;
  int step_iters = 64;
  int retval = 0;
  while (active[from] && (progressed ? progress2 : 1) && step_iters < in_iters) {
    progress2 = 0;
    progress = 1;
    while (progress) {
      progress = 0;
      int o = 0;
      for (int i = 0; i < active[from]; ++i) {
        if (*stop) { goto cleanup; }
        T zx = iters[from][i].zx;
        T zy = iters[from][i].zy;
        T dx = iters[from][i].dx;
        T dy = iters[from][i].dy;
        T cx = in_cx + scale * iters[from][i].i;
        T cy = in_cy + scale * iters[from][i].j;
        int32_t n = iters[from][i].n;
        { /* iterate */
          const T er2 = 65536;
          T zx2 = sqr(zx);
          T zy2 = sqr(zy);
          T z2 = zx2 + zy2;
          T z2xy = 2 * zx * zy;
          while (likely(n < step_iters && z2 < er2)) {
            T zdzx = zx * dx - zy * dy;
            T zdzy = zx * dy + zy * dx;
            dx = 2 * zdzx + 1;
            dy = 2 * zdzy;
            zx = zx2 - zy2 + cx;
            zy = z2xy + cy;
            zx2 = sqr(zx);
            zy2 = sqr(zy);
            z2xy = 2 * zx * zy;
            z2 = zx2 + zy2;
            ++n;
          }
          if (! (z2 < er2)) {
            int k = ((int) iters[from][i].j) * 256 + iters[from][i].i;
            out_n[k] = to_float(1 + n - log(log(z2) / log(er2))/log(2.0));
            out_d[k] = to_float((log(z2) * sqrt(z2 / (sqr(dx) + sqr(dy)))) / scale);
            out_a[k] = to_float(atan2(zy, zx));
            for (int x = iters[from][i].i - 1; x <= iters[from][i].i + 1; ++x) {
              if (x < 0 || 255 < x) continue;
              for (int y = iters[from][i].j - 1; y <= iters[from][i].j + 1; ++y) {
                if (y < 0 || 255 < y) continue;
                k = y * 256 + x;
```

```
                if (out_n[k] == 0) {
                  iters[to][o].zx = T(0);
                  iters[to][o].zy = T(0);
                  iters[to][o].dx = T(0);
                  iters[to][o].dy = T(0);
                  iters[to][o].n  = 0;
                  iters[to][o].i = x;
                  iters[to][o].j = y;
                  out_n[k] = -1;
                  ++o;
                }
              }
            }
            if (min_iters < n) min_iters = n;
            if (max_iters < n) max_iters = n;
            ++progress;
          } else {
            iters[to][o].zx = zx;
            iters[to][o].zy = zy;
            iters[to][o].dx = dx;
            iters[to][o].dy = dy;
            iters[to][o].n  = n;
            iters[to][o].i = iters[from][i].i;
            iters[to][o].j = iters[from][i].j;
            ++o;
          }
        }
      }
      active[to] = o;
      int temp = from; from = to; to = temp;
      progress2 = progress2 || progress;
    }
    step_iters *= 2;
    progressed = progressed || progress2;
  }
  { /* deinitialize border */
    int k;
    for (int i = 0; i < 256; ++i) { /* edges */
      k = 256 * i   +   0; if (out_n[k] < 0) out_n[k] = 0;
      k = 256 * i   + 255; if (out_n[k] < 0) out_n[k] = 0;
      k = 256 * 0   +   i; if (out_n[k] < 0) out_n[k] = 0;
      k = 256 * 255 +   i; if (out_n[k] < 0) out_n[k] = 0;
    }
  }
  retval = max_iters;
cleanup:
  delete[] iters[0];
  delete[] iters[1];
  fpu_fix_end(&fpu);
  return retval;
}

extern "C" {
  int compute_f32(int *stop, float *out_n, float *out_d, float *out_a, float ↙
      ↳ in_cx, float in_cy, int in_level, int in_iters) {
    return compute(stop, out_n, out_d, out_a, in_cx, in_cy, in_level, in_iters);
  }
```

```
  int compute_f64(int *stop, float *out_n, float *out_d, float *out_a, double ↙
      ↪ in_cx, double in_cy, int in_level, int in_iters) {
    return compute(stop, out_n, out_d, out_a, in_cx, in_cy, in_level, in_iters);
  }
  int compute_f128(int *stop, float *out_n, float *out_d, float *out_a, double *↙
      ↪ in_cx, double *in_cy, int in_level, int in_iters) {
    return compute(stop, out_n, out_d, out_a, dd_real(in_cx), dd_real(in_cy), ↙
        ↪ in_level, in_iters);
  }
  int compute_f256(int *stop, float *out_n, float *out_d, float *out_a, double *↙
      ↪ in_cx, double *in_cy, int in_level, int in_iters) {
    return compute(stop, out_n, out_d, out_a, qd_real(in_cx), qd_real(in_cy), ↙
        ↪ in_level, in_iters);
  }
#ifdef HAVE_MPFR
  int compute_mpfr(int *stop, float *out_n, float *out_d, float *out_a, const ↙
      ↪ char *in_cx, const char *in_cy, int in_level, int in_iters) {
    mpfr_prec_t p(in_level + 20);
    return compute(stop, out_n, out_d, out_a, mp_real(in_cx, p), mp_real(in_cy, ↙
        ↪ p), in_level, in_iters);
  }
#endif
}
```

9 extra/expblend.frag

```
uniform sampler2D outer;
uniform sampler2D inner;

void main(void) {
  vec2 p = gl_TexCoord[0].st;
  vec2 q = 0.5 * pow(0.5, p.y) * vec2(cos(p.x), sin(p.x));
  vec4 o = texture2D(outer, vec2(0.5) + 0.5 * q);
  vec4 i = texture2D(inner, vec2(0.5) +       q);
  gl_FragColor = mix(o, i, p.y);
}
```

10 extra/Exponential2.hs

```
module Main (main) where

import System.FilePath ((</>))
import System.Directory (getAppUserDataDirectory)
import System.IO
import Control.Monad (forM_, when)
import Data.IORef
import Data.Ratio ((%))
import Foreign (malloc, nullPtr, allocaBytes, peek, with, sizeOf, alloca, Ptr)
import Foreign.C (CFloat)
import Numeric (readSigned, readFloat)
import QuadTree
import Shader
import Tile (Tile(..), getTile, freeTile, rootSquare)
import Graphics.Rendering.OpenGL hiding (Position, Size)
import qualified Graphics.Rendering.OpenGL as GL
import Graphics.Rendering.OpenGL.Raw
```

```
    ( glGenFramebuffers, glBindFramebuffer, glFramebufferTexture2D, ↙
        ↳ glGenerateMipmap
    , gl_FRAMEBUFFER, gl_COLOR_ATTACHMENT0, gl_TEXTURE_2D
    , glTexImage2D, gl_R32F, gl_RGBA32F, gl_LUMINANCE, gl_FLOAT, gl_RGBA, ↙
        ↳ gl_UNSIGNED_BYTE
    , gl_FALSE, glClampColor, gl_CLAMP_VERTEX_COLOR, gl_CLAMP_READ_COLOR
    , gl_CLAMP_FRAGMENT_COLOR )
import qualified Graphics.UI.GLUT as GLUT

data E = E{ outer, inner {- , result -} :: TextureObject, qss :: [[Quad]], ↙
    ↳ firstPass :: Bool, combine, blender :: Program, fbo :: GLuint, cache :: ↙
    ↳ String
          , center :: (Rational, Rational), level :: Int, outh :: Handle }

texturef s = do
  [tex] <- genObjectNames 1
  texture Texture2D $= Enabled
  textureBinding Texture2D $= Just tex
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_RGBA) s s 0 gl_RGBA ↙
      ↳ gl_UNSIGNED_BYTE nullPtr
  textureFilter Texture2D $= ((Linear', Just Linear'), Linear')
  textureWrapMode Texture2D S $= (Repeated, ClampToEdge)
  textureWrapMode Texture2D T $= (Repeated, ClampToEdge)
  textureBinding Texture2D $= Nothing
  texture Texture2D $= Disabled
  return tex

main :: IO ()
main = do
  GLUT.initialWindowSize $= GL.Size 600 66
  GLUT.initialDisplayMode $= [GLUT.RGBAMode, GLUT.DoubleBuffered]
  (_, [sre, sim]) <- GLUT.getArgsAndInitialize
  _ <- GLUT.createWindow "expo"
  glClampColor gl_CLAMP_VERTEX_COLOR gl_FALSE
  glClampColor gl_CLAMP_READ_COLOR gl_FALSE
  glClampColor gl_CLAMP_FRAGMENT_COLOR gl_FALSE
  drawBuffer $= BackBuffers
  fbo' <- alloca $ \p -> glGenFramebuffers 1 p >> peek p
  outer'  <- texturef 1024
  inner'  <- texturef 1024
  --result' <- texturef 1024
  combine' <- shader Nothing (Just "colourize.frag")
  blender' <- shader Nothing (Just "expblend.frag")
  appDir <- getAppUserDataDirectory "gruff"
  let cacheDir = appDir </> "cache"
      rootRegion x y = Region{ regionWest = x - 2, regionEast = x + 2, ↙
          ↳ regionNorth = y - 2, regionSouth = y + 2 }
{-
      [(cx0, "")] = readSigned readFloat sre
      [(cy0, "")] = readSigned readFloat sim
-}
      cx0 = ↙
          ↳ (-323646011660011038445643659020796377473921825225709314424092461431870888083812347
          ↳ % ↙
          ↳ 183479889279205720928865671624166955263725199133462489899007107150953830087078784 64
          ↳
      cy0 =   ↙
```

```
            ↳ 16133655142616935754541850251669053667680644979978590956487946345932070567979398899
            ↳  % ↙
            ↳ 57337465399751787790270522382552173519914124729207028093439720984673071902212120201
            ↳
      center0 = (cx0, cy0)
      target = rootRegion cx0 cy0
      level0 = 3
      eqs = equads rootSquare target level0
  outh' <- openBinaryFile "out.raw" WriteMode
  eR <- newIORef E{ outer = outer', inner = inner', {- result = result', -} qss ↙
      ↳ = tail eqs, firstPass = True, combine = combine', blender = blender', ↙
      ↳ fbo = fbo', center = center0, level = level0, cache = cacheDir, outh = ↙
      ↳ outh' }
  GLUT.displayCallback $= eRender eR
  GLUT.mainLoop

equads sq reg l = quads sq reg l : equads sq (shrink reg) (l + 1)
shrink r =
  let x = (regionEast r + regionWest r) / 2
      w = (regionEast r - regionWest r) / 2
      y = (regionSouth r + regionNorth r) / 2
      h = (regionSouth r - regionNorth r) / 2
  in  Region{ regionWest = x - w / 2, regionEast = x + w / 2, regionNorth = y - ↙
      ↳ h / 2, regionSouth = y + h / 2 }

eRender eR = do
  e <- readIORef eR
  let qs = head (qss e)
  viewport $= (GL.Position 0 0, GL.Size 1024 1024)
  loadIdentity
  ortho2D 0 1024 1024 0
  withFBO (fbo e) (inner e) $ do
   forM_ qs $ \q -> do
    Just t@(Tile _ ns ds ts) <- with 0 $ \p -> getTile putStrLn (cache e) p q
    tit <- upload ns
    tde <- upload ds
    ttt <- upload ts
    freeTile t
    currentProgram $= Just (combine e)
    lit <- get $ uniformLocation (combine e) "it"
    lde <- get $ uniformLocation (combine e) "de"
    ltt <- get $ uniformLocation (combine e) "tt"
    lsh <- get $ uniformLocation (combine e) "hshift"
    lsc <- get $ uniformLocation (combine e) "hscale"
    uniform lit $= TexCoord1 (0 :: GLint)
    uniform lde $= TexCoord1 (1 :: GLint)
    uniform ltt $= TexCoord1 (2 :: GLint)
    uniform lsh $= TexCoord1 (0 :: GLfloat)
    uniform lsc $= TexCoord1 (0.5 :: GLfloat)
    activeTexture $= TextureUnit 0
    textureBinding Texture2D $= Just tit
    activeTexture $= TextureUnit 1
    textureBinding Texture2D $= Just tde
    activeTexture $= TextureUnit 2
    textureBinding Texture2D $= Just ttt
    renderPrimitive Quads $ do
      let sq = square rootSquare q
```

```
            t x y = texCoord $ TexCoord2 (x :: GLdouble) (y :: GLdouble)
            v x y = let (x', y') = toPixel e x y
                    in  vertex $ Vertex2
                          (fromRational x' :: GLdouble)
                          (fromRational y' :: GLdouble)
            x0 = squareWest sq
            x1 = squareWest sq  + squareSize sq
            y0 = squareNorth sq
            y1 = squareNorth sq + squareSize sq
          color $ Color3 1 1 (1::GLdouble)
          t 0 1 >> v x0 y1
          t 0 0 >> v x0 y0
          t 1 0 >> v x1 y0
          t 1 1 >> v x1 y1
        textureBinding Texture2D $= Nothing
        activeTexture $= TextureUnit 1
        textureBinding Texture2D $= Nothing
        activeTexture $= TextureUnit 0
        textureBinding Texture2D $= Nothing
        currentProgram $= Nothing
        deleteObjectNames [tit, tde, ttt]
      textureBinding Texture2D $= Just (inner e)
      glGenerateMipmap gl_TEXTURE_2D
      textureBinding Texture2D $= Nothing
      when (not (firstPass e)) $ do
        viewport $= (GL.Position 0 0, GL.Size 600 66)
        loadIdentity
        ortho2D 0 1 0 1
        do
          currentProgram $= Just (blender e)
          lo <- get $ uniformLocation (blender e) "outer"
          li <- get $ uniformLocation (blender e) "inner"
          uniform lo $= TexCoord1 (0 :: GLint)
          uniform li $= TexCoord1 (1 :: GLint)
          activeTexture $= TextureUnit 0
          textureBinding Texture2D $= Just (outer e)
          activeTexture $= TextureUnit 1
          textureBinding Texture2D $= Just (inner e)
          let t x y = texCoord $ TexCoord2 (x :: GLdouble) (y :: GLdouble)
              v x y = vertex $ Vertex2 (x :: GLdouble) (y :: GLdouble)
          renderPrimitive Quads $ do
            color $ Color3 1 1 (1::GLdouble)
            t 0       1 >> v 0 1
            t 0       0 >> v 0 0
            t (2 * pi) 0 >> v 1 0
            t (2 * pi) 1 >> v 1 1
          textureBinding Texture2D $= Nothing
          activeTexture $= TextureUnit 0
          textureBinding Texture2D $= Nothing
          currentProgram $= Nothing
          let bytes = 600 * 66 * 3
          allocaBytes bytes $ \ptr -> do
            readPixels (GL.Position 0 0) (GL.Size 600 66) $ PixelData RGB ↙
                ↳ UnsignedByte ptr
            hPutBuf (outh e) ptr bytes
            hFlush (outh e)
      atomicModifyIORef eR $ \e' -> (e' { outer = inner e', inner = outer e', qss = ↙
```

```
      ↳ tail (qss e'), firstPass = False, level = level e + 1 }, ())
  GLUT.swapBuffers
  GLUT.reportErrors
  GLUT.postRedisplay Nothing

withFBO :: GLuint -> TextureObject -> IO a -> IO a
withFBO fbo (TextureObject tex) act = do
  glBindFramebuffer gl_FRAMEBUFFER fbo
  glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D tex 0
  r <- act
  glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D 0 0
  glBindFramebuffer gl_FRAMEBUFFER 0
  return r

toPixel :: E -> Rational -> Rational -> (Rational, Rational)
toPixel e = f
  where
    f x y = (x', y')
      where
        x' = ((x - cx) / r' + 0.5) * w'
        y' = ((y - cy) / r  + 0.5) * h'
    w' = 1024
    h' = 1024
    a = w' / h'
    (cx, cy) = center e
    r = 8 / 2 ^ level e * h' / (2 * 256)
    r' = a * r

upload :: Ptr CFloat -> IO TextureObject
upload p = do
  [tex] <- genObjectNames 1
  texture Texture2D $= Enabled
  textureBinding Texture2D $= Just tex
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_R32F) 256 256 0 gl_LUMINANCE ↙
      ↳ gl_FLOAT p
  textureFilter Texture2D $= ((Nearest, Nothing), Nearest)
  textureWrapMode Texture2D S $= (Repeated, ClampToEdge)
  textureWrapMode Texture2D T $= (Repeated, ClampToEdge)
  textureBinding Texture2D $= Nothing
  texture Texture2D $= Disabled
  return tex
```

11 extra/Exponential.hs

```
module Main (main) where

import System.FilePath ((</>))
import System.Directory (getAppUserDataDirectory)
import System.IO
import Control.Monad (forM_, when)
import Data.IORef
import Foreign (malloc, nullPtr, allocaBytes, peek, with, sizeOf, alloca, Ptr)
import Foreign.C (CFloat)
import Numeric (readSigned, readFloat)
import QuadTree
import Shader
import Tile (Tile(..), getTile, freeTile, rootSquare)
```

```
import Graphics.Rendering.OpenGL hiding (Position, Size)
import qualified Graphics.Rendering.OpenGL as GL
import Graphics.Rendering.OpenGL.Raw
  ( glGenFramebuffers, glBindFramebuffer, glFramebufferTexture2D, ↙
      ↳ glGenerateMipmap
  , gl_FRAMEBUFFER, gl_COLOR_ATTACHMENT0, gl_TEXTURE_2D
  , glTexImage2D, gl_R32F, gl_RGBA32F, gl_LUMINANCE, gl_FLOAT, gl_RGBA, ↙
      ↳ gl_UNSIGNED_BYTE
  , gl_FALSE, glClampColor, gl_CLAMP_VERTEX_COLOR, gl_CLAMP_READ_COLOR
  , gl_CLAMP_FRAGMENT_COLOR )
import qualified Graphics.UI.GLUT as GLUT

data E = E{ outer, inner, result :: TextureObject, qss :: [[Quad]], firstPass ::↙
    ↳ Bool, combine, blender :: Program, fbo :: GLuint, cache :: String
          , center :: (Rational, Rational), level :: Int, outh :: Handle }

texturef s = do
  [tex] <- genObjectNames 1
  texture Texture2D $= Enabled
  textureBinding Texture2D $= Just tex
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_RGBA) s s 0 gl_RGBA ↙
      ↳ gl_UNSIGNED_BYTE nullPtr
  textureFilter Texture2D $= ((Linear', Just Linear'), Linear')
  textureWrapMode Texture2D S $= (Repeated, ClampToEdge)
  textureWrapMode Texture2D T $= (Repeated, ClampToEdge)
  textureBinding Texture2D $= Nothing
  texture Texture2D $= Disabled
  return tex

main :: IO ()
main = do
  GLUT.initialWindowSize $= GL.Size 400 44
  GLUT.initialDisplayMode $= [GLUT.RGBAMode, GLUT.DoubleBuffered]
  (_, [sre, sim]) <- GLUT.getArgsAndInitialize
  _ <- GLUT.createWindow "expo"
  glClampColor gl_CLAMP_VERTEX_COLOR gl_FALSE
  glClampColor gl_CLAMP_READ_COLOR gl_FALSE
  glClampColor gl_CLAMP_FRAGMENT_COLOR gl_FALSE
  drawBuffer $= BackBuffers
  fbo' <- alloca $ \p -> glGenFramebuffers 1 p >> peek p
  outer'  <- texturef 256
  inner'  <- texturef 256
  result' <- texturef 512
  combine' <- shader Nothing (Just "colourize.frag")
  blender' <- shader Nothing (Just "expblend.frag")
  appDir <- getAppUserDataDirectory "gruff"
  let cacheDir = appDir </> "cache"
      rootRegion x y = Region{ regionWest = x - 2, regionEast = x + 2, ↙
          ↳ regionNorth = y - 2, regionSouth = y + 2 }
      [(cx0, "")] = readSigned readFloat sre
      [(cy0, "")] = readSigned readFloat sim
      center0 = (cx0, cy0)
      target = rootRegion cx0 cy0
      eqs = equads rootSquare target
  outh' <- openBinaryFile "out.raw" WriteMode
  eR <- newIORef E{ outer = outer', inner = inner', result = result', qss = tail↙
      ↳ eqs, firstPass = True, combine = combine', blender = blender', fbo = ↙
```

```
      ↳ fbo', center = center0, level = 0, cache = cacheDir, outh = outh' }
  GLUT.displayCallback $= eRender eR
  GLUT.mainLoop

eRender eR = do
  e <- readIORef eR
  let qs = head (qss e)
  viewport $= (GL.Position 0 0, GL.Size 256 256)
  loadIdentity
  ortho2D 0 256 256 0
  withFBO (fbo e) (inner e) $ do
   forM_ qs $ \q -> do
    Just t@(Tile _ ns ds ts) <- with 0 $ \p -> getTile putStrLn (cache e) p q
    tit <- upload ns
    tde <- upload ds
    ttt <- upload ts
    freeTile t
    currentProgram $= Just (combine e)
    lit <- get $ uniformLocation (combine e) "it"
    lde <- get $ uniformLocation (combine e) "de"
    ltt <- get $ uniformLocation (combine e) "tt"
    lsh <- get $ uniformLocation (combine e) "hshift"
    lsc <- get $ uniformLocation (combine e) "hscale"
    uniform lit $= TexCoord1 (0 :: GLint)
    uniform lde $= TexCoord1 (1 :: GLint)
    uniform ltt $= TexCoord1 (2 :: GLint)
    uniform lsh $= TexCoord1 (0 :: GLfloat)
    uniform lsc $= TexCoord1 (0.5 :: GLfloat)
    activeTexture $= TextureUnit 0
    textureBinding Texture2D $= Just tit
    activeTexture $= TextureUnit 1
    textureBinding Texture2D $= Just tde
    activeTexture $= TextureUnit 2
    textureBinding Texture2D $= Just ttt
    renderPrimitive Quads $ do
      let sq = square rootSquare q
          t x y = texCoord $ TexCoord2 (x :: GLdouble) (y :: GLdouble)
          v x y = let (x', y') = toPixel e x y
                  in  vertex $ Vertex2
                        (fromRational x' :: GLdouble)
                        (fromRational y' :: GLdouble)
          x0 = squareWest sq
          x1 = squareWest sq  + squareSize sq
          y0 = squareNorth sq
          y1 = squareNorth sq + squareSize sq
      color $ Color3 1 1 (1::GLdouble)
      t 0 1 >> v x0 y1
      t 0 0 >> v x0 y0
      t 1 0 >> v x1 y0
      t 1 1 >> v x1 y1
    textureBinding Texture2D $= Nothing
    activeTexture $= TextureUnit 1
    textureBinding Texture2D $= Nothing
    activeTexture $= TextureUnit 0
    textureBinding Texture2D $= Nothing
    currentProgram $= Nothing
    deleteObjectNames [tit, tde, ttt]
```

```
    textureBinding Texture2D $= Just (inner e)
    glGenerateMipmap gl_TEXTURE_2D
    textureBinding Texture2D $= Nothing
    when (not (firstPass e)) $ do
      viewport $= (GL.Position 0 0, GL.Size 400 44)
      loadIdentity
      ortho2D 0 1 0 1
      do
        currentProgram $= Just (blender e)
        lo <- get $ uniformLocation (blender e) "outer"
        li <- get $ uniformLocation (blender e) "inner"
        uniform lo $= TexCoord1 (0 :: GLint)
        uniform li $= TexCoord1 (1 :: GLint)
        activeTexture $= TextureUnit 0
        textureBinding Texture2D $= Just (outer e)
        activeTexture $= TextureUnit 1
        textureBinding Texture2D $= Just (inner e)
        let t x y = texCoord $ TexCoord2 (x :: GLdouble) (y :: GLdouble)
            v x y = vertex $ Vertex2 (x :: GLdouble) (y :: GLdouble)
        renderPrimitive Quads $ do
          color $ Color3 1 1 (1::GLdouble)
          t 0       1 >> v 0 1
          t 0       0 >> v 0 0
          t (2 * pi) 0 >> v 1 0
          t (2 * pi) 1 >> v 1 1
        textureBinding Texture2D $= Nothing
        activeTexture $= TextureUnit 0
        textureBinding Texture2D $= Nothing
        currentProgram $= Nothing
        let bytes = 400 * 44 * 3
        allocaBytes bytes $ \ptr -> do
          readPixels (GL.Position 0 0) (GL.Size 400 44) $ PixelData RGB ↙
              ↳ UnsignedByte ptr
          hPutBuf (outh e) ptr bytes
          hFlush (outh e)
    atomicModifyIORef eR $ \e' -> (e' { outer = inner e', inner = outer e', qss = ↙
        ↳ tail (qss e'), firstPass = False, level = level e + 1 }, ())
    GLUT.swapBuffers
    GLUT.reportErrors
    GLUT.postRedisplay Nothing

withFBO :: GLuint -> TextureObject -> IO a -> IO a
withFBO fbo (TextureObject tex) act = do
    glBindFramebuffer gl_FRAMEBUFFER fbo
    glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D tex 0
    r <- act
    glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D 0 0
    glBindFramebuffer gl_FRAMEBUFFER 0
    return r

toPixel :: E -> Rational -> Rational -> (Rational, Rational)
toPixel e = f
    where
      f x y = (x', y')
        where
          x' = ((x - cx) / r' + 0.5) * w'
          y' = ((y - cy) / r  + 0.5) * h'
```

```
      w' = 256
      h' = 256
      a = w' / h'
      (cx, cy) = center e
      r = 8 / 2 ^ level e * h' / (2 * 256)
      r' = a * r

upload :: Ptr CFloat -> IO TextureObject
upload p = do
  [tex] <- genObjectNames 1
  texture Texture2D $= Enabled
  textureBinding Texture2D $= Just tex
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_R32F) 256 256 0 gl_LUMINANCE ↙
      ↳ gl_FLOAT p
  textureFilter Texture2D $= ((Nearest, Nothing), Nearest)
  textureWrapMode Texture2D S $= (Repeated, ClampToEdge)
  textureWrapMode Texture2D T $= (Repeated, ClampToEdge)
  textureBinding Texture2D $= Nothing
  texture Texture2D $= Disabled
  return tex
```

12 extra/feature-database.hs

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
import Data.List (intercalate)
import Data.Ratio ((%), numerator, denominator)

import qualified Data.Map as M

import Data.Vec (NearZero(nearZero))
import Data.Number.MPFR (MPFR, RoundMode(Near, Up), Precision, getPrec, int2w, ↙
    ↳ fromIntegerA, toString)
import Data.Number.MPFR.Instances.Near ()

import Fractal.RUFF.Types.Complex (realPart, imagPart)
import Fractal.RUFF.Mandelbrot.Address
  ( Angle, period, kneading
  , AngledInternalAddress(..), angledInternalAddress
  , addressPeriod, prettyAngledInternalAddress
  )
import Fractal.RUFF.Mandelbrot.Atom (MuAtom(..), findAtom_)

instance NearZero MPFR where
  nearZero x = let p = getPrec x in not (abs x > int2w Up p 1 (4 - fromIntegral ↙
      ↳ p))

newtype R = R{ unR :: MPFR }
  deriving (Eq, Ord, Floating, Real, RealFrac, NearZero)

instance Num R where
  R a + R b = R (a + b)
  R a * R b = R (a * b)
  R a - R b = R (a - b)
  negate (R a) = R (negate a)
  abs (R a) = R (abs a)
  signum (R a) = R (signum a)
  fromInteger i = R (fromIntegerA Near bits i)
```

```
instance Fractional R where
  R a / R b = R (a / b)
  recip (R a) = R (recip a)
  fromRational r = R (fromIntegerA Near bits (numerator r) / fromIntegerA Near ↙
      ↳ bits (denominator r))

instance Show R where
  show (R m) = toString (ceiling $ (2::Double) + log 2 / log 10 * fromIntegral (↙
      ↳ getPrec m)) m

bits :: Precision
bits = 1000

isIsland :: AngledInternalAddress -> Bool
isIsland (Unangled 1) = True
isIsland (Angled p r (Unangled q)) = q /= p * denominator r
isIsland (Angled _ _ a) = isIsland a

angles :: [(AngledInternalAddress, (Angle, Angle))]
angles = concat $
  [ merge M.empty $
    [ (addr, a)
    | n <- [ 0 .. d ]
    , let a = n % d
    , (period . kneading) a == Just p
    , let Just addr = angledInternalAddress a
    ]
  | p <- [ 1 .. 16 ]
  , let d = 2 ^ p - 1
  ]

merge :: M.Map AngledInternalAddress Angle -> [(AngledInternalAddress, Angle)] ↙
    ↳ -> [(AngledInternalAddress, (Angle, Angle))]
merge m [] | M.null m = []
merge m ((addr, a):rest) = case M.lookup addr m of
  Nothing -> merge (M.insert addr a m) rest
  Just b -> (addr, (b, a)) : merge (M.delete addr m) rest

pretty :: (AngledInternalAddress, (Angle, Angle)) -> String
pretty (addr, (lo, hi)) = intercalate "," $
  [ show (addressPeriod addr)
  , show (isIsland addr)
  , prettyAngledInternalAddress addr
  , show (numerator   lo)
  , show (denominator lo)
  , show (numerator   hi)
  , show (denominator hi)
  ] ++ case (isIsland addr && addressPeriod addr > 1, findAtom_ addr) of
    (False, _) -> ["","","","",""]
    (True, Just mu) ->
      let digits = ceiling $ 8 - logBase 10 (muSize mu)
          showR = toString digits . unR
      in  [ show  . muOrient $ mu
          , show  . muSize   $ mu
          , showR . realPart . muNucleus $ mu
          , showR . imagPart . muNucleus $ mu
```

```
            ]
    _ -> ["","","","",""]

main :: IO ()
main = mapM_ (putStrLn . pretty) angles
```

13 extra/FixedPrecision.hs

```
{-# LANGUAGE BangPatterns, EmptyDataDecls, Rank2Types #-}
module FixedPrecision where

import Data.Bits (bit, shiftL, shiftR)
import Data.Ratio ((%), numerator, denominator)
import TypeLevel.NaturalNumber -- type-level-natural-number
import Text.FShow.Raw -- floatshow

newtype Fixed p = F Integer

getPrecision :: Fixed p -> p
getPrecision _ = undefined

withPrecision :: NaturalNumber p => Int -> p -> (forall q. NaturalNumber q => q ↙
    ↳ -> a) -> a
withPrecision 0 p f = f p
withPrecision n p f = withPrecision (n - 1) (successorTo p) f

instance NaturalNumber p => BinDecode (Fixed p) where
  decode l@(F x) = (x, negate . naturalNumberAsInt . getPrecision $ l)
  showDigits l = 2 + floor (fromIntegral (1 + naturalNumberAsInt (getPrecision l↙
      ↳ )) * logBase 10 2 :: Double)

showF :: NaturalNumber p => Fixed p -> String
showF x = binDecFormat (Generic (Just (-5, 5))) Nothing x

instance Eq (Fixed p) where F x == F y = x == y
instance Ord (Fixed p) where F x `compare` F y = x `compare` y
instance NaturalNumber p => Num (Fixed p) where
  F x + F y = F $  x + y
  F x - F y = F $  x - y
  l@(F x) * F y = F $ (x * y) `shiftR` naturalNumberAsInt (getPrecision l)
  negate (F x) = F (negate x)
  abs (F x) = F (abs x)
  signum (F x) = F (signum x)
  fromInteger x = let r = F (x `shiftL` naturalNumberAsInt (getPrecision r)) in ↙
      ↳ r
instance NaturalNumber p => Fractional (Fixed p) where
  l@(F x) / F y = F ((x `shiftL` naturalNumberAsInt (getPrecision l)) `div` y)
  fromRational x =
    let r = F ((numerator x `shiftL` naturalNumberAsInt (getPrecision r)) `div` ↙
        ↳ denominator x) in r
instance NaturalNumber p => Real (Fixed p) where
  toRational l@(F x) = x % bit (naturalNumberAsInt $ getPrecision l)

incrPrecision :: NaturalNumber p => Fixed p -> Fixed (SuccessorTo p)
incrPrecision (F x) = F (x `shiftL` 1)

decrPrecision :: NaturalNumber p => Fixed (SuccessorTo p) -> Fixed p
```

```
decrPrecision (F x) = F (x `shiftR` 1)
```

14 extra/gruff-cache-stats.sh

```
#!/bin/bash
cachedir="${HOME}/.gruff/cache/"
find "${cachedir}" -type f |
while read fname
do
  score="$(./gruff-tile-score < "${fname}")"
  echo "${score} ${fname}"
done
```

15 extra/gruff-tile-score.c

```
#include <stdio.h>
#include <stdlib.h>

#define TSIZE 256

int main(int argc, char **argv) {
  char header[16];
  const unsigned int count = TSIZE * TSIZE;
  const unsigned int bytes = count * sizeof(float);
  float *dwell = malloc(bytes);
  if (! dwell) { return 1; }
  if (1 != fread(header, 16, 1, stdin)) { free(dwell); return 1; }
  if (1 != fread(dwell, bytes, 1, stdin)) { free(dwell); return 1; }
  double score = 0;
  for (unsigned int i = 0; i < count; ++i) {
    double d = dwell[i];
    if (d > 0) { score += d; }
  }
  printf("%.16e\n", score);
  free(dwell);
  return 0;
}
```

16 extra/Makefile

```
gruff-tile-score: gruff-tile-score.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -o gruff-tile-score gruff-tile-score.c
```

17 extra/mp_real.h

```
#ifndef MP_REAL_H
#define MP_REAL_H 1

#include <mpfr.h>
#define MPFR_RNDN GMP_RNDN

class mp_real {
public:
  mpfr_t m;
```

```
    mp_real() {
      mpfr_init2(m, 53);
      mpfr_set_d(m, 0, MPFR_RNDN);
    }
    mp_real(const double &s) {
      mpfr_init2(m, 53);
      mpfr_set_d(m, s, MPFR_RNDN);
    }
    mp_real(const int32_t &s) {
      mpfr_init2(m, 53);
      mpfr_set_si(m, s, MPFR_RNDN);
    }
    mp_real(const mp_real &s) {
      mpfr_init2(m, mpfr_get_prec(s.m));
      mpfr_set(m, s.m, MPFR_RNDN);
    }
    mp_real(mpfr_prec_t p) {
      mpfr_init2(m, p);
    }
    mp_real(mpfr_t s, mpfr_prec_t p) {
      mpfr_init2(m, p);
      mpfr_set(m, s, MPFR_RNDN);
    }
    mp_real(const char *s, mpfr_prec_t p) {
      mpfr_init2(m, p);
      mpfr_set_str(m, s, 10, MPFR_RNDN);
    };
    mp_real(double s, mpfr_prec_t p) {
      mpfr_init2(m, p);
      mpfr_set_d(m, s, MPFR_RNDN);
    };
    ~mp_real() {
      mpfr_clear(m);
    }
  #define maxprec(x, y) max(mpfr_get_prec(x.m), mpfr_get_prec(y.m))
    inline mp_real& operator=(mp_real const &y) {
      mpfr_set_prec(m, maxprec((*this), y));
      mpfr_set(m, y.m, MPFR_RNDN);
      return *this;
    }
  };

  inline bool operator<(mp_real const &x, mp_real const &y) {
    return mpfr_less_p(x.m, y.m);
  }

  inline mp_real operator+(mp_real const &x, mp_real const &y) {
    mp_real r(maxprec(x, y));
    mpfr_add(r.m, x.m, y.m, MPFR_RNDN);
    return r;
  }

  inline mp_real operator-(mp_real const &x, mp_real const &y) {
    mp_real r(maxprec(x, y));
    mpfr_sub(r.m, x.m, y.m, MPFR_RNDN);
    return r;
  }
```

```
inline mp_real operator*(mp_real const &x, mp_real const &y) {
  mp_real r(maxprec(x, y));
  mpfr_mul(r.m, x.m, y.m, MPFR_RNDN);
  return r;
}

inline mp_real operator/(mp_real const &x, mp_real const &y) {
  mp_real r(maxprec(x, y));
  mpfr_div(r.m, x.m, y.m, MPFR_RNDN);
  return r;
}

inline mp_real sqr(mp_real const &x) {
  mp_real r(mpfr_get_prec(x.m));
  mpfr_sqr(r.m, x.m, MPFR_RNDN);
  return r;
}

inline mp_real sqrt(mp_real const &x) {
  mp_real r(mpfr_get_prec(x.m));
  mpfr_sqrt(r.m, x.m, MPFR_RNDN);
  return r;
}

inline mp_real log(mp_real const &x) {
  mp_real r(mpfr_get_prec(x.m));
  mpfr_log(r.m, x.m, MPFR_RNDN);
  return r;
}

inline mp_real atan2(mp_real const &y, mp_real const &x) {
  mp_real r(maxprec(x, y));
  mpfr_atan2(r.m, y.m, x.m, MPFR_RNDN);
  return r;
}

inline mp_real pow(mp_real const &y, mp_real const &x) {
  mp_real r(maxprec(x, y));
  mpfr_pow(r.m, y.m, x.m, MPFR_RNDN);
  return r;
}

inline float to_float(mp_real const &x) {
  return mpfr_get_d(x.m, MPFR_RNDN);
}

#undef maxprec

#endif
```

18 extra/MuAtom.hs

```
{-# LANGUAGE BangPatterns #-}
module MuAtom(MuAtom(..), MuProgress(..), muFromAddress, MuProgress'(..), ↙
    ↳ muToAddress, MuProgress''(..), muLocate) where
```

```
import Control.Arrow ((***))
import Data.Ratio ((%))
import Data.Vec (NearZero, nearZero)

import Fractal.RUFF.Mandelbrot.Address (AngledInternalAddress, Angle, ↙
    ↳ splitAddress, addressPeriod, externalAngles, angledInternalAddress)
import Fractal.RUFF.Mandelbrot.Nucleus (findNucleus, findBond, findPeriod)
import Fractal.RUFF.Mandelbrot.Ray (externalRay, externalRayOut)
import Fractal.RUFF.Types.Complex (Complex, magnitude, magnitude2, phase, ↙
    ↳ mkPolar)

import Number (R)

data MuAtom = MuAtom
  { muNucleus :: !(Complex R)
  , muSize    :: !Double
  , muOrient  :: !Double
  , muPeriod  :: !Integer
  }
  deriving (Read, Show, Eq)

data MuProgress
  = MuSplitTodo
  | MuSplitDone AngledInternalAddress [Angle]
  | MuAnglesTodo
  | MuAnglesDone !Rational !Rational
  | MuRayTodo
  | MuRay !Integer
  | MuRayDone !(Complex R)
  | MuNucleusTodo
  | MuNucleus !Integer
  | MuNucleusDone !(Complex R)
  | MuBondTodo
  | MuBond !Integer
  | MuBondDone !(Complex R)
  | MuSuccess !MuAtom
  | MuFailed
  deriving (Read, Show, Eq)

muFromAddress :: AngledInternalAddress -> [MuProgress]
muFromAddress addr = MuSplitTodo :
  let (!iaddr, !caddr) = splitAddress addr
      !p = addressPeriod iaddr
  in  MuSplitDone iaddr caddr : MuAnglesTodo : case externalAngles iaddr of
    Nothing -> [MuFailed]
    Just (!lo, !hi) -> MuAnglesDone lo hi : MuRayTodo :
      let sharpness = 8
          er = 65536
          accuracy = 1e-10
          ok w = magnitude2 w < 2 * er ^ (2::Int) -- NaN -> False
          rayl = externalRay accuracy sharpness er lo
          rayh = externalRay accuracy sharpness er hi
          ray' = takeWhile (uncurry (&&) . (ok *** ok) . snd) $ [ 1 .. ] `zip` (↙
              ↳ rayl `zip` rayh)
          rgo [] _ = [MuFailed]
          rgo [_] _ = [MuFailed]
          rgo ((i, (xl, xh)):m@((_, (yl, yh)):_)) f
```

```
              | i < fromIntegral sharpness * (p + 16) = MuRay i : rgo m f
              | dl + dh > dx + dy = MuRay i : rgo m f
              | otherwise = MuRayDone x : f x
              where
                x  = 0.5 * (xl + xh)
                dl = magnitude2 (xl - yl)
                dh = magnitude2 (xh - yh)
                dx = magnitude2 (xl - xh)
                dy = magnitude2 (yl - yh)
        in  rgo ray' $ \rayend -> MuNucleusTodo :
          let nuc = findNucleus p rayend
              nuc' = takeWhile (ok . snd) $ [ 1 .. ] `zip` nuc
              ngo []  _ = [MuFailed]
              ngo [_] _ = [MuFailed]
              ngo ((i, x):m@((_, y):_)) f
                | not (nearZero (x - y)) = MuNucleus i : ngo m f
                | otherwise = MuNucleusDone x : f x
          in  ngo nuc' $ \nucleus -> MuBondTodo :
            let bnd = findBond p nucleus 0.5
                bnd' = takeWhile (ok . snd) $ [ 1.. ] `zip` bnd
                bgo []  _ = [MuFailed]
                bgo [_] _ = [MuFailed]
                bgo ((i, x):m@((_, y):_)) f
                  | not (nearZero (x - y)) = MuBond i : bgo m f
                  | otherwise = MuBondDone x : f x
            in  bgo bnd' $ \bond ->
              let delta  = bond - nucleus
                  size   = realToFrac $ magnitude delta / 0.75
                  orient = realToFrac $ phase delta
                  atom   = MuAtom{ muNucleus = nucleus, muSize = size, muOrient = ↙
                      ↳ orient, muPeriod = p }
              in  if 10 > size && size > 0 then [MuSuccess atom] else [MuFailed]

data MuProgress'
  = MuCuspTodo
  | MuCuspDone !(Complex R)
  | MuDwellTodo
  | MuDwell !Integer
  | MuDwellDone !Integer
  | MuRayOutTodo
  | MuRayOut !Double
  | MuRayOutDone !(Complex R)
  | MuExternalTodo
  | MuExternalDone !Rational
  | MuAddressTodo
  | MuSuccess' AngledInternalAddress
  | MuFailed'
  deriving (Read, Show, Eq)

muToAddress :: MuAtom -> [MuProgress']
muToAddress mu = MuCuspTodo :
  let cusp = muNucleus mu - mkPolar (realToFrac (muSize mu)) (realToFrac ((↙
      ↳ muOrient mu)))
      er = 65536
      er2 = er * er
  in  MuCuspDone cusp : MuDwellTodo :
    let dgo z n f = MuDwell n : if magnitude2 z > er2 then f n else dgo (z * z +↙
```

```
        ↳ cusp) (n + 1) f
    in  dgo 0 0 $ \n -> MuDwellDone n : MuRayOutTodo :
      let rgo ((i,!_):izs@(_:_)) f = MuRayOut (fromIntegral i / (fromIntegral ↙
          ↳ sharpness * fromIntegral n)) : rgo izs f
          rgo [(_,!z)] f | magnitude2 z > er2 = MuRayOutDone z : f z
          rgo _ _ = [MuFailed']
          accuracy = 1e-16
          sharpness = 16
          epsilon0 = realToFrac (muSize mu) * accuracy
      in  rgo ([(1 :: Integer) ..] `zip` externalRayOut (fromIntegral n + 100) ↙
          ↳ epsilon0 accuracy sharpness er cusp) $ \rend -> MuExternalTodo :
        let den = 2 ^ muPeriod mu - 1
            num' = fromIntegral den * warp (phase rend / (2 * pi))
            num = round num'
            warp t
              | t > 0 = t
              | otherwise = t + 1
            angle = num % den
        in  MuExternalDone angle : MuAddressTodo : case angledInternalAddress ↙
            ↳ angle of
              Nothing -> [MuFailed']
              Just addr -> if addressPeriod addr /= muPeriod mu then [MuFailed'] ↙
                  ↳ else [MuSuccess' addr]

data MuProgress''
  = MuScanTodo
  | MuScan
  | MuScanDone !Integer
  | MuNucleusTodo'
  | MuNucleus' !Integer
  | MuNucleusDone' !(Complex R)
  | MuBondTodo'
  | MuBond' !Integer
  | MuBondDone' !(Complex R)
  | MuSuccess'' !MuAtom
  | MuFailed''
  deriving (Read, Show, Eq)

muLocate :: Complex R -> Double -> [MuProgress'']
muLocate c r = MuScanTodo : MuScan : case findPeriod 10000000 (realToFrac r) c ↙
    ↳ of
  Nothing -> [MuFailed'']
  Just p -> MuScanDone p : MuNucleusTodo' :
    let ok w = magnitude2 w < 16 -- NaN -> False
        nuc = findNucleus p c
        nuc' = takeWhile (ok . snd) $ [ 1 .. ] `zip` nuc
        ngo [] _ = [MuFailed'']
        ngo [_] _ = [MuFailed'']
        ngo ((i, x):m@((_, y):_)) f
          | not (nearZero (x - y)) = MuNucleus' i : ngo m f
          | otherwise = MuNucleusDone' x : f x
    in  ngo nuc' $ \nucleus -> MuBondTodo' :
      let bnd = findBond p nucleus 0.5
          bnd' = takeWhile (ok . snd) $ [ 1.. ] `zip` bnd
          bgo [] _ = [MuFailed'']
          bgo [_] _ = [MuFailed'']
          bgo ((i, x):m@((_, y):_)) f
```

```
            | not (nearZero (x - y)) = MuBond' i : bgo m f
            | otherwise = MuBondDone' x : f x
      in  bgo bnd' $ \bond ->
        let delta  = bond - nucleus
            size   = realToFrac $ magnitude delta / 0.75
            orient = realToFrac $ phase delta
            atom   = MuAtom{ muNucleus = nucleus, muSize = size, muOrient = orient, muPeriod = p }
        in  if 10 > size && size > 0 then [MuSuccess'' atom] else [MuFailed'']
```

19 extra/Number.hs

```
{-# LANGUAGE CPP, GeneralizedNewtypeDeriving #-}
module Number (R, toRational') where

import Data.Bits (bit)
import Data.Ratio ((%))
import Data.Vec (NearZero)

import Numeric.QD (QuadDouble)
import Numeric.QD.Vec ()

type R = QuadDouble

toRational' :: RealFrac a => Int -> a -> Rational
toRational' l a = round (a * fromInteger b) % b
  where b = bit (l + 16)
```

20 extra/Poincare.hs

```
import Data.Maybe (mapMaybe)
import Numeric.LinearAlgebra ((><), nullVector, toList)
import Numeric.QD (QuadDouble)
import Numeric.QD.Vec ()
import Fractal.GRUFF
import Fractal.RUFF.Mandelbrot.Address (parseAngledInternalAddress)
import Fractal.RUFF.Mandelbrot.Atom (MuAtom(..), findAtom_)
import Fractal.RUFF.Types.Complex(Complex((:+)), magnitude, magnitude2, cis, phase)

import Debug.Trace

debug p x = trace (p ++ ": " ++ show x) x

type R = QuadDouble

data Poincare a = Poincare !(Complex a) !a
  deriving Show

distance (Poincare x1 y1) (Poincare x2 y2) = acosh (1 + (magnitude2 dx + dy * dy) / (2 * y1 * y2))
  where
    dx = x2 - x1
    dy = y2 - y1

geodesic p1@(Poincare x1 y1) p2@(Poincare x2 y2)
```

```
    | x1 == x2  = \t -> Poincare x1 (y1 * (y2 / y1) ** t)
    | otherwise = \t -> let x:+y = go (exp (t * dt)) in Poincare (x1 + (x :+ 0) * dir) y
  where
    a =  d * y1
    b = -c * y1
    [c, d] = debug "cd" . normalize (map realToFrac . (id :: [Double] -> [Double]) . toList . nullVector . (2><2) . map realToFrac $ ms
    dt = distance p1 p2
    eT = exp dt
    ms = debug "ms" [ eT * mdx, y2 - y1 * eT, y1 - eT * y2, mdx ]
    mdx = magnitude dx
    dir = cis (phase dx)
    dx = x2 - x1
    go t = (b :+ (a * t)) / (d :+ (c * t))

{-
    (a i     + b) / (c i     + d) =    0    :+ y1
    (a i e^T + b) / (c i e^T + d) = |x2-x1| :+ y2
=>
    (a i     + b) = (c i     + d) * (   0    :+ y1)
    (a i e^T + b) = (c i e^T + d) * (|x2-x1| :+ y2)
=>
    a   =  d * y1
    b   = -c * y1
    a e = c e |x2 - x1| + d y2
    b   = d   |x2 - x1| - c e y2
=>
    d y1 e = c e |x2 - x1| + d y2
    -c y1  = d   |x2 - x1| - c e y2
=>
    c e |x2 - x1|    + d (y2 - y1 e) = 0
    c (-(e y2 - y1)) + d |x2 - x1|   = 0
-}

poincare mu = Poincare (muNucleus mu) (16 * realToFrac (muSize mu))

mus = mapMaybe (\s -> findAtom_ =<< parseAngledInternalAddress s)
        [ "1 1/3 3 " ++ (unwords . map show . take m) [(4 :: Int) .. ] | m <- [1 .. 16] ]

pcs :: [Poincare R]
pcs = Poincare 0 4 : map poincare mus ++ [ Poincare 0 4 ]

dgs = zipWith (\p q -> (distance p q, geodesic p q)) pcs (tail pcs)

fps = 25

segment (d, g) = segment' 0
  where
    segment' n | n > fps * d = []
               | otherwise = g (n / (fps * d)) : segment' (n + 1)

scenes = zipWith scene [0..] (concatMap segment dgs)

scene s (Poincare (cx :+ cy) r) =
  let f = filename s
```

```
        i = Image
80              { imageLocation = Location
                    { center = toRational cx :+ toRational cy
                    , radius = realToFrac r
                    }
                , imageViewport = Viewport
85                  { aspect = 1080 / 576
                    , orient = 0
                    }
                , imageWindow   = Window
                    { width = 1080
90                  , height = 576
                    , supersamples = 4
                    }
                , imageColours  = Colours
                    { colourInterior = Colour 1 0.5 0
95                  , colourBoundary = Colour 0 0 0
                    , colourExterior = Colour 1 1 1
                    }
                , imageLabels   = []
                , imageLines    = []
100             }
      in (i, f)

filename :: Integer -> FilePath
filename s = (reverse . take 8 . (++ "00000000") . reverse . show) s ++ ".ppm"
105

main = defaultMain scenes
```

21 extra/rts.c

```
char *ghc_rts_opts = "-N";
```

22 extra/SoftFloat.hs

```
{-# LANGUAGE BangPatterns, DeriveDataTypeable #-}

module SoftFloat where

5   import Control.Monad.Instances ()
    import Data.Bits (bit, shiftL, shiftR)
    import Data.Int (Int32, Int64)
    import Data.Monoid (mappend)
    import Data.Ratio ((%), numerator, denominator)
10  import Numeric (readSigned, readFloat, showSigned, showGFloat)
    import Data.Data (Data)
    import Data.Typeable(Typeable)

    type Mantissa = Int32
15  type Mantissa' = Int64
    type Exponent = Int32

    data SoftFloat = SoftFloat !Mantissa !Exponent
      deriving (Data, Typeable)
20

    up :: Mantissa -> Mantissa'
```

```
up = fromIntegral

down :: Mantissa' -> Mantissa
down = fromIntegral

softFloat :: Mantissa' -> Exponent -> SoftFloat
softFloat 0 !_ = SoftFloat 0 0
softFloat m !e
  | m < up minBound || up maxBound < m = softFloat (m `shiftR` 1) (e + 1)
  | up minBound `shiftR` 1 < m && m < up maxBound `shiftR` 1 = softFloat (m `↙
      ↳ shiftL` 1) (e - 1)
  | otherwise = SoftFloat (down m) e

up' :: Mantissa -> Integer
up' = fromIntegral

down' :: Integer -> Mantissa
down' = fromIntegral

softFloat' :: Integer -> Mantissa -> SoftFloat
softFloat' 0 !_ = SoftFloat 0 0
softFloat' m !e
  | m < up' minBound || up' maxBound < m = softFloat' (m `shiftR` 1) (e + 1)
  | up' minBound `shiftR` 1 < m && m < up' maxBound `shiftR` 1 = softFloat' (m `↙
      ↳ shiftL` 1) (e - 1)
  | otherwise = SoftFloat (down' m) e

instance Show SoftFloat where
  show f = showSigned (showGFloat (Just 12)) 9 f ""

instance Read SoftFloat where
  readsPrec _ = readSigned readFloat

instance Eq SoftFloat where
  SoftFloat m e == SoftFloat m' e' = m == m' && e == e'
  SoftFloat m e /= SoftFloat m' e' = m /= m' || e /= e'

instance Ord SoftFloat where
  SoftFloat m e `compare` SoftFloat m' e'
    | m  == 0 = 0 `compare` m'
    | m' == 0 = m `compare` 0
    | m > 0 && m' > 0 = (e `compare` e') `mappend` (m `compare` m')
    | m > 0 && m' < 0 = GT
    | m < 0 && m' > 0 = LT
    | m < 0 && m' < 0 = (e' `compare` e) `mappend` (m `compare` m')
    | otherwise = error "gruff.SoftFloat.Ord.compare"

instance Num SoftFloat where
  SoftFloat 0 _ + f = f
  f + SoftFloat 0 _ = f
  SoftFloat m e + SoftFloat m' e'
    | e >  e' = softFloat (up m + (up m' `shiftR` fromIntegral (e - e'))) e
    | e == e' = softFloat (up m + up m') e
    | e <  e' = softFloat ((up m `shiftR` fromIntegral (e' - e)) + up m') e'
    | otherwise = error "gruff.SoftFloat.Num.+"
  SoftFloat 0 _ - f = negate f
  f - SoftFloat 0 _ = f
```

```
  SoftFloat m e - SoftFloat m' e'
    | e >  e' = softFloat (up m - (up m' `shiftR` fromIntegral (e - e'))) e
    | e == e' = softFloat (up m - up m') e
    | e <  e' = softFloat ((up m `shiftR` fromIntegral (e' - e)) - up m') e'
    | otherwise = error "gruff.SoftFloat.Num.-"
  negate (SoftFloat m e) = SoftFloat (negate m) e
  f@(SoftFloat 0 _) * _ = f
  _ * f@(SoftFloat 0 _) = f
  SoftFloat m e * SoftFloat m' e' = softFloat (up m * up m') (e + e')
  fromInteger i = softFloat' i 0 -- FIXME large integers
  abs (SoftFloat m e) = SoftFloat (abs m) e
  signum (SoftFloat m _) = softFloat (up $ signum m) 0

instance Fractional SoftFloat where
  f@(SoftFloat 0 _) / _ = f
  _ / (SoftFloat 0 _) = error "/0"
  SoftFloat m e / SoftFloat m' e' = softFloat ((up m `shiftL` 32) `div` up m') (e - e' - 32)
  fromRational r = fromInteger (numerator r) / fromInteger (denominator r)

instance Real SoftFloat where
  toRational (SoftFloat m e)
    | e >= 0 = fromInteger (toInteger m * bit (fromIntegral e))
    | e <  0 = toInteger m % bit (negate (fromIntegral e))
    | otherwise = error "gruff.SoftFloat.Real.toRational"

instance RealFrac SoftFloat where
  properFraction f@(SoftFloat m e)
    | e >= 0 = (fromIntegral m * fromInteger (bit (fromIntegral e)), 0)
    | e <= (-31) = (0, f)
    | otherwise = (fmap fromRational . properFraction . toRational) f

instance Floating SoftFloat where
  pi = realToFrac (pi :: Double)
  sqrt f = go 1
    where
      go !r =
        let r' = (r + f / r) / 2
        in  if r == r' then r else go r'
  exp f = go 0 1 1 1
    where
      go !e !nf !fn !n =
        let e' = e + fn / nf
        in  if e == e' then e else go e' (nf * n) (f * fn) (n + 1)
  log f@(SoftFloat _ e)
    | f > 0 = pi / (2 * agm 1 (2^^(2 - m) / f)) - fromIntegral m * ln2
    | otherwise = error "gruff.SoftFloat.Floating.log"
    where
      m = 37 - e
      agm !a! b =
        let a' = (a + b) / 2
            b' = sqrt (a * b)
        in  if a' == b' || (a == a' && b == b') then a' else agm a' b'
      ln2 = realToFrac (log 2 :: Double)

  sin = viaDouble sin
  cos = viaDouble cos
```

```
  tan = viaDouble tan
  sinh = viaDouble sinh
  cosh = viaDouble cosh
  tanh = viaDouble tanh
  asin = viaDouble asin
  acos = viaDouble acos
  atan = viaDouble atan
  asinh = viaDouble asinh
  acosh = viaDouble acosh
  atanh = viaDouble atanh

viaDouble :: (Double -> Double) -> (SoftFloat -> SoftFloat)
viaDouble f = uncurry encodeFloat . decodeFloat . checkFloat . f . uncurry ↙
    ↳ encodeFloat . decodeFloat

checkFloat :: Double -> Double
checkFloat f
  | isNaN f = error "NaN"
  | isInfinite f = error "Inf"
  | otherwise = f

instance RealFloat SoftFloat where
  floatRadix _ = 2
  floatDigits _ = 31
  floatRange _ = (fromIntegral (minBound :: Exponent), fromIntegral (maxBound :: ↙
      ↳ Exponent))
  decodeFloat (SoftFloat m e) = (fromIntegral m, fromIntegral e)
  encodeFloat m e = softFloat' m (fromIntegral e)
  isNaN _ = False
  isInfinite _ = False
  isDenormalized _ = False
  isNegativeZero _ = False
  isIEEE _ = False
```

23 extra/test.hs

```
import Data.List (nub)
import Data.Maybe (catMaybes, mapMaybe)
import Data.Ratio ((%))
import System.Environment (getArgs)
import System.IO (hFlush, stdout)

import Fractal.RUFF.Mandelbrot.Address (AngledInternalAddress, splitAddress, ↙
    ↳ parseAngledInternalAddress, prettyAngledInternalAddress, ↙
    ↳ angledInternalAddress)
import Fractal.RUFF.Types.Complex (imagPart)

import MuAtom

{-
ray tracing failures (wanted from angle : found from reverse trace)
  1 1/3 3 6 7 : 1 1/3 3 4 5 7
  1 1/3 3 4 7 : 1 1/3 3 4 5 7
  1 2 1/3 6 7 : 1 2 1/3 5 6 7
  1 2 1/3 5 7 : 1 2 1/3 5 6 7
  1 1/3 3 6 8 : 1 1/3 3 5 6 8
  1 1/3 3 5 8 : 1 1/3 3 5 6 8
```

```
    1 1/3 3 4 7 8 : 1 1/3 3 4 5 6 8
    1 1/3 3 4 5 8 : 1 1/3 3 4 5 6 8
    1 2 1/3 6 8 : 1 2 1/3 5 6 7 8
    1 2 1/3 5 7 8 : 1 2 1/3 5 6 7 8
    1 2 1/3 5 6 8 : 1 2 1/3 5 6 7 8
    1 2 3 1/3 8 : 1 2 3 1/3 7 8
ie, 12 unexpected failures from 116 test cases
FIXED: silly assumption in forward trace finding ray end...

-- reverse ray tracing failures:
islands' :: [AngledInternalAddress]
islands' = mapMaybe parseAngledInternalAddress
  [ "1 2 1/3 5 6 8 9"
  , "1 1/6 6 8 10"
  , "1 1/5 5 8 9 10"
  , "1 1/3 3 6 2/3 10"
  , "1 1/3 3 2/3 8 9 10"
  , "1 3/8 8 9 10"
  , "1 2/5 5 8 10"
  , "1 2 1/3 6 7 8 10"
  ]
-- ie, 8 unexpected failures from 510 test cases
-- FIXED: not enough accuracy (1e-8 -> 1e-16) or sharpness (8 -> 16) in ray ↙
    ↳ trace...

-- current regressions FAIL (reverse trace):
islands' :: [AngledInternalAddress]
islands' = mapMaybe parseAngledInternalAddress
  [ "1 1/4 4 6 8"
  , "1 3/8 8 9"
  , "1 2 3 5 1/3 9"
  , "1 1/6 6 7 10"
  , "1 1/3 3 4 6 7 8 10"
  , "1 2/5 5 6 8 10"
  , "1 1/7 7 9 11"
  ]

-}

main :: IO ()
main = do
  args <- getArgs
  let n = case map reads args of
            [[(i, "")]] -> i
            _ -> 0
  mapM_ test (drop n islands)

islands :: [AngledInternalAddress]
islands = (nub . filter island . catMaybes . map angledInternalAddress . nub . ↙
    ↳ filter (0.5 >)) [ num % den | p <- [ 2 .. ], let den = 2 ^ (p :: Int) - 1,↙
    ↳  num <- [ 1 .. den - 1] ]
  where island = null . snd . splitAddress

test :: AngledInternalAddress -> IO ()
test addr = putStr (prettyAngledInternalAddress addr ++ " : ") >> hFlush stdout ↙
    ↳ >> case last (take limit $ muFromAddress addr) of
  MuSuccess mu
```

```
    | (1e-16 <) . abs . imagPart . muNucleus $ mu ->
        case last (take limit $ muToAddress mu) of
          MuSuccess' addr'
            | addr == addr' -> putStrLn "OK"
            | otherwise -> putStrLn (prettyAngledInternalAddress addr' ++ " FAIL↙
              ↳ ")
          e -> putStrLn $ "FAIL (reverse trace) " ++ show e
    | otherwise -> putStrLn "AXIS"
  e -> putStrLn $ "FAIL (forward trace) " ++ show e

limit :: Int
limit = 10000
```

24 extra/TileFeatures.hs

```
{-# LANGUAGE BangPatterns #-}
import Control.Exception (evaluate)
import Control.Monad (forM)
import Control.Monad.Instances ()
import Data.List (nubBy)
import Data.Maybe (mapMaybe,catMaybes)
import Numeric.QD.Vec ()

import Foreign (Ptr, peekElemOff)
import Foreign.C (CFloat)

import Fractal.RUFF.Mandelbrot.Atom
import Fractal.RUFF.Mandelbrot.Address
import Fractal.RUFF.Types.Complex

computeFeatures :: Tile -> IO [(MuAtom Double, AngledInternalAddress)]
computeFeatures (Tile q _ ds _) = do
  let s = square rootSquare q
      cx, cy, sx, sy :: Double
      cx = fromRational (squareWest s)
      cy = fromRational (squareNorth s)
      cs = fromRational (squareSize s)
      sx = cs / fromIntegral width
      sy = cs / fromIntegral height
      inq (x:+y) = cx <= x && x <= cx + cs && cy <= y && y <= cy + cs
  sss <- forM [2,4,8] $ \d -> do
    let dx = width `div` d
        dy = height `div` d
        c i j = (cx + fromIntegral (i * dx) * sx) :+ (cy + fromIntegral (j * dy)↙
          ↳ * sy)
        r = 2 * cs / fromIntegral d
    forM [1 .. d - 1] $ \j -> do
      forM [1 .. d - 1] $ \i -> do
        b <- containsBorder ds (j * dy) (i * dx) dy dx
        return $ if b then locate_ (c i j) r else Nothing
  let fs = mapMaybe (\ !(mu@MuAtom{ muNucleus = mx :+ my }) -> fmap ((,) mu) (↙
      ↳ findAddress' (take 10000 $ findAddress (mu{ muNucleus = realToFrac mx :+↙
      ↳  realToFrac my :: Complex DoubleDouble }))))) . nubBy muEqual . filter (↙
      ↳ inq . muNucleus) . catMaybes . concat . concat $ sss
  !hack <- evaluate (length (show fs)) -- force evaluation in this thread
  return fs
```

```
containsBorder :: Ptr CFloat -> Int -> Int -> Int -> Int -> IO Bool
containsBorder p y x dy dx = do
  any (\d -> 0 < d && d < 4) `fmap` sequence [ peekElemOff p (width * j + i) | j <- [ y - dy .. y + dy - 1], i <- [x - dx .. x + dx - 1] ]

muEqual :: MuAtom Double -> MuAtom Double -> Bool
muEqual m n = magnitude2 (muNucleus m - muNucleus n) < (muSize m * muSize m + muSize n * muSize n) / 4


--     , windowDecorated := False
module Overlay where

import Graphics.Rendering.OpenGL

import Number (R)

overlay :: Size -> TextureObject -> IO ()
overlay (Size w h) t = do

type Image = StorableArray (Int, Int, Int) Word8
type Bitmap = StorableArray (Int, Int) Bool

getPixels :: StorableArray (Int, Int, Int) Word8 -> IO ()
getPixels a = do
  ((y0,x0,c0),(y1,x1,c1)) <- getBounds a
  let w = x1 - x0 + 1
      h = y1 - y0 + 1
  guard (c1 - c0 + 1 == 3)
  withStorableArray a $ \ptr ->
    readPixels (Position (fromIntegral x0) (fromIntegral y0)) (Size (fromIntegral w) (fromIntegral h)) (PixelData RGB UnsignedByte ptr)

getAtoms :: Bitmap -> IO [MuAtom]
getAtoms a = do
  bs@((y0,x0),(y1,x1)) <- getBounds a
```

25 extra/UnComplex.hs

```
{-# LANGUAGE DeriveDataTypeable #-}

-- Complex numbers without the 'RealFloat' constraint.
module UnComplex
  ( Complex((:+)), cis, mkPolar
  , realPart, imagPart, conjugate
  , magnitude2, magnitude, phase, polar
  ) where

import Data.Data (Data)
import Data.Typeable (Typeable)

-- | Complex number type without the 'RealFloat' constraint.
data Complex r = !r :+ !r
  deriving (Read, Show, Eq, Data, Typeable)

instance Functor Complex where fmap f (x:+y) = f x :+ f y
```

```
instance Num r => Num (Complex r) where
  (x :+ y) + (u :+ v) = (x + u) :+ (y + v)
  (x :+ y) - (u :+ v) = (x - u) :+ (y - v)
  (x :+ y) * (u :+ v) = (x * u - y * v) :+ (x * v + y * u)
  negate (x :+ y) = negate x :+ negate y
  abs = error "Fractal.RUFF.Types.Complex.Num.abs"
  signum = error "Fractal.RUFF.Types.Complex.Num.signum"
  fromInteger n = fromInteger n :+ 0

instance Fractional r => Fractional (Complex r) where
  (x :+ y) / (u :+ v) = ((x * u + y * v) / d) :+ ((y * u - x * v) / d) where d =↙
      ↳ u * u + v * v
  fromRational r = fromRational r :+ 0

instance (Ord r, Floating r) => Floating (Complex r) where
  pi = pi :+ 0
  exp (x :+ y) = mkPolar (exp x) y
  log z = let (r, t) = polar z in log r :+ t
  sin (x :+ y) = (sin x * cosh y) :+        (cos x * sinh y)
  cos (x :+ y) = (cos x * cosh y) :+ negate (sin x * sinh y)
  tan z = sin z / cos z
  asin z = negate i * log (i * z + sqrt (1 - z*z)) where i = 0:+1
  acos z = negate i * log (z + sqrt (z*z - 1)) where i = 0:+1
  atan z = 1/2 * i * log ((1 - iz)/(1 + iz)) where i = 0:+1 ; iz = i * z
  sinh z = (exp z - exp (-z)) / 2
  cosh z = (exp z + exp (-z)) / 2
  tanh z = let ez2 = exp (2 * z) in (ez2 - 1) / (ez2 + 1)
  asinh z = log (z + sqrt (z*z + 1))
  acosh z = log (z + sqrt (z*z - 1))
  atanh z = 1/2 * log ((1 + z) / (1 - z))

-- | Extract the real part.
realPart :: Complex r -> r
realPart (r :+ _) = r

-- | Extract the imaginary part.
imagPart :: Complex r -> r
imagPart (_ :+ i) = i

-- | Complex conjugate.
conjugate :: Num r => Complex r -> Complex r
conjugate (r :+ i) = r :+ negate i

-- | Complex magnitude squared.
magnitude2 :: Num r => Complex r -> r
magnitude2 (r :+ i) = r * r + i * i

-- | Complex magnitude.
magnitude :: Floating r => Complex r -> r
magnitude = sqrt . magnitude2

-- | Complex phase.
phase :: (Ord r, Floating r) => Complex r -> r
phase (r :+ i)
  | r > 0 && i > 0 =      atan (    i /     r)
  | r > 0 && i < 0 =    - atan (abs i /     r)
  | r < 0 && i > 0 = pi - atan (    i / abs r)
```

```
    | r < 0 && i < 0 =      atan (abs i / abs r) - pi
    | i > 0          =      pi / 2
    | i < 0          =    - pi / 2
    | r < 0          =      pi
    | otherwise      =      0

-- | Complex number with the given magnitude and phase.
mkPolar :: Floating r => r -> r -> Complex r
mkPolar r t = (r * cos t) :+ (r * sin t)

-- | Complex number with magnitude 1 and the given phase.
cis :: Floating r => r -> Complex r
cis t = cos t :+ sin t

-- | Convert to polar form.
polar :: (Ord r, Floating r) => Complex r -> (r, r)
polar z = (magnitude z, phase z)
```

26 .gitignore

```
dist
-
*.hi
*.o
```

27 gruff.cabal

```
Name:                gruff
Version:             1.0
Synopsis:            fractal explorer GUI using the ruff library
Description:
    Mandelbrot Set fractal explorer using the ruff library.
    .
    Requires GTK, OpenGL, and GLSL fragment shader support; lots of RAM
    and multiple CPU cores recommended.
    .
    gruff-0.4 changes:
    .
      * Fixed-point computations for image tiles (based on Integer).
    .
      * MPFR support removed due to poor integer-simple performance.
    .
      * Colouring adjusted to show finer detail.
    .
    Features in this version include:
    .
      * Interactive fractal browser display with mouse controls:
        .
          * Left click to zoom in (press shift for bigger jumps).
        .
          * Right click to zoom out (press shift for bigger jumps).
        .
          * Middle click to center.
        .
          * Shift middle click to auto-focus.
        .
          * Control left click to label with period.
```

```
            .
              * Control right click to label with angled internal address (slow!).
        .
          * Session persistance (stored in @~\/.gruff\/state.gruff@ - states can
            also be loaded from and saved to other files, including labels and
            rays). Note that the file format is incompatible with previous
            releases of gruff.
        .
          * Tile cache (by default in @~\/.gruff\/cache@ - symlink it somewhere
            with a few GB of space if you plan on exploring a lot).
        .
          * Limited amount of customizable colouring (colours for interior,
            border, and exterior points, as well as labels and rays).
        .
          * Supersampling for more detailed images (useful range is 1 to 16).
        .
          * Rudimentary scripting support (see the gruff-examples package).
        .
        Future versions will focus on improving performance.

License:             GPL-2
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Category:            Graphics

Build-type:          Simple
Cabal-version:       >=1.8

Data-dir:            data
Data-files:          icon.png, merge.frag, minimal.frag
Extra-source-files:  TODO

Library
  Hs-source-dirs:    src
  Exposed-modules:   Fractal.GRUFF
  Build-depends:     base >= 3 && < 5,
                     ruff >= 1.0 && < 1.1
  GHC-options:       -Wall

Executable gruff
  Hs-source-dirs:    src
  Main-is:           gruff.hs
  Other-modules:     Browser
                     Compute
                     GLUTGtk
                     Interact
                     Logger
                     Progress
                     QuadTree
                     Shader
                     Snapshot
                     StatusDialog
                     Tile
                     TileShake
                     Utils
                     View
```

```
    Build-depends:      gruff == 1.0,
                        base >= 4 && < 5,
                        containers >= 0.3 && < 0.5,
                        directory >= 1 && < 2,
                        filepath >= 1 && < 2,
                        floatshow >= 0.2 && < 0.3,
                        FTGL == 1.333,
                        gtk >= 0.11 && < 0.13,
                        gtkglext >= 0.11 && < 0.13,
                        old-locale >= 1 && < 2,
                        OpenGL >= 2.4 && < 3,
                        OpenGLRaw >= 1.1 && < 2,
                        shake >= 0.2.10 && < 0.3,
                        qd >= 1 && < 2,
                        time >= 1 && < 2,
                        ruff >= 1.0 && < 1.1,
                        type-level-natural-number >= 1.1 && < 1.2,
                        bitwise >= 0.1 && < 0.2,
                        bytestring,
                        mtl
    GHC-options:        -Wall -threaded -rtsopts
    GHC-Prof-Options:   -auto-all -caf-all

source-repository head
    type:     git
    location: http://code.mathr.co.uk/gruff.git

source-repository this
    type:     git
    location: http://code.mathr.co.uk/gruff.git
    tag:      v1.0
```

28 LICENSE

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

```
 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                        Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
```

this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you

conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices
stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any
part thereof, to be licensed as a whole at no charge to all third
parties under the terms of this License.

c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License. (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to

this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author

to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

```
Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```

29 Setup.hs

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

30 src/Browser.hs

```
module Browser (Browser(..), browserNew, browserRenders) where

import Prelude hiding (log, lines)
import Control.Concurrent
  ( MVar, newEmptyMVar, newMVar, takeMVar, putMVar, tryTakeMVar )
import Control.Monad (foldM, forM_, unless, when)
import Data.IORef (IORef, newIORef, readIORef, atomicModifyIORef)
import Data.List (sortBy)
import qualified Data.Map as M
import Data.Map (Map)
import Data.Maybe (fromMaybe)
import Data.Ord (comparing)
import qualified Data.Set as S
import Foreign (Ptr, alloca, peek, nullPtr)
import Foreign.C (CUChar)
import Graphics.Rendering.OpenGL hiding
  ( viewport, Angle, Color, Point, Position, Size, S, R )
```

```
import qualified Graphics.Rendering.OpenGL as GL
import Graphics.Rendering.OpenGL.Raw
  ( glTexImage2D, gl_TEXTURE_2D, gl_RGBA, gl_LUMINANCE
  , gl_UNSIGNED_BYTE, gl_FALSE, glClampColor
  , gl_CLAMP_VERTEX_COLOR, gl_CLAMP_READ_COLOR, gl_CLAMP_FRAGMENT_COLOR
  , glGenFramebuffers, glBindFramebuffer, glFramebufferTexture2D
  , gl_FRAMEBUFFER, gl_COLOR_ATTACHMENT0, glGenerateMipmap
  , gl_FRAMEBUFFER_COMPLETE, glCheckFramebufferStatus
  , glGetError
  )
import Graphics.UI.Gtk hiding
  ( get, Window, Viewport, Region, Size, Action, Image, Label, labelText )
import qualified Graphics.UI.Gtk as GTK

import Graphics.Rendering.FTGL as FTGL

import Fractal.GRUFF (Complex((:+)))

import GLUTGtk
  ( GLUTGtk, Size(Size), Key(MouseButton), KeyState(Down), Position(Position)
  , postRedisplay, widget, realizeCallback, reshapeCallback
  , displayCallback, keyboardMouseCallback
  )
import Shader (shader)
import QuadTree (Quad(..))
import Tile (Tile(Tile), freeTile)
import TileShake (computeTiles)
import Logger (Logger, LogLevel(Debug))
import qualified Logger as Log
import Snapshot (writeSnapshot)
import View
  ( Image(..), Window(..), Viewport(..), Label(..), Line(..)
  , BufferSize(..), tileSize, delta
  , Colours(..), Colour(..)
  , pixelLocation, originQuad, visibleQuads, locationPixel
  , defImage, defWindow, windowSize'
  )
import Interact (MouseCallbacks, Mod(..), But(..))

import Paths_gruff (getDataFileName)

type QuadList = [(Complex Int, Quad)]
data GruffImage = GruffImage
  { image :: Image
  -- tile cache
  , tiles :: Map Quad TextureObject
  , queue :: MVar [Tile]
  , jobs :: MVar [Quad]
  , viewQuads :: (QuadList, QuadList)
  , worker :: Maybe (IO ())
  , gl :: GLUTGtk
  , cacheDir :: FilePath
  , log :: LogLevel -> String -> IO ()
  , prog :: Program
  , combineProg :: Program
  , tsheet0 :: TextureObject
  , tsheet1 :: TextureObject
```

```
    , sheetCount0 :: Int
    , sheetCount1 :: Int
    , sheetCountTarget :: Int
    , fbo :: GLuint
    , cacheSizeMin :: Int
    , cacheSizeMax :: Int
    -- callbacks
    , exitCallback :: Maybe (IO ())
    , reshapeCallback :: Maybe (Int -> Int -> IO ())
    , mouseCallback :: MouseCallbacks
    , doneCallback :: Maybe (IO ())
    , abortCallback :: Maybe (IO ())
    , doClear :: Bool
    , font :: FTGL.Font
    }

sheetSize :: GruffImage -> BufferSize
sheetSize g =
  let i = image g
      w = imageWindow i
      d = sqrt (supersamples w) * windowSize' w
      t = roundUp2 (ceiling d)
  in  BufferSize{ texels = t }

roundUp2 :: Int -> Int -- fails for too small and too large inputs
roundUp2 x = head . dropWhile (x >=) . iterate (2 *) $ 1

zoomPhase :: GruffImage -> Double
zoomPhase = delta . imageLocation . image

rotationA :: GruffImage -> Double
rotationA = orient . imageViewport . image

sheetBlend :: GruffImage -> (Int, (Double, Double))
sheetBlend g =
  let z = zoomPhase g
      b = logBase 4 (4 ** z)
      h = floor b
      d = fromIntegral h - b
  in  (h, (d, 1 - (1 - 4**d) / (4**(d+1) - 4**d)))

blendFactor :: GruffImage -> Double
blendFactor = snd . snd . sheetBlend

iDisplay :: IORef GruffImage -> IO ()
iDisplay iR = do
  checkError "display begin"
  s0 <- readIORef iR
  mtls <- tryTakeMVar (queue s0)
  case mtls of
    Just tls -> do
      putMVar (queue s0) []
      forM_ tls $ \tile@(Tile q ds) -> do
        let (v0, v1) = viewQuads s0
            vqs = map snd (v0 ++ v1)
        if q `elem` vqs
          then do
```

```
              tde <- upload ds
              checkError "upload ds"
              freeTile tile
              atomicModifyIORef iR $ \s' ->
                ( s'{ tiles = M.insert q tde (tiles s') }, () )
            else do
              freeTile tile
      Nothing -> return ()
    s <- readIORef iR
    log s Debug $ "displayCallback " ++ show (imageLocation (image s))
    c <- atomicModifyIORef iR $ \s' -> (s'{ doClear = False }, doClear s')
    when c $ do
      clearSheet s False
      checkError "clear sheet F"
      clearSheet s True
      checkError "clear sheet T"
      atomicModifyIORef iR $ \s' -> (s'{ sheetCount0 = 0, sheetCount1 = 0 }, ())
    (todo0, upped0) <- renderSheet s False
    checkError "render sheet F"
    (todo1, upped1) <- renderSheet s True
    checkError "render sheet T"
    atomicModifyIORef iR $ \s' ->
      ( s'{ viewQuads = (todo0, todo1)
          , sheetCount0 = sheetCount0 s' + upped0
          , sheetCount1 = sheetCount1 s' + upped1
          }, () )
    let w = width (imageWindow (image s))
        h = height (imageWindow (image s))
    GL.viewport $=
      (GL.Position 0 0, GL.Size (fromIntegral w) (fromIntegral h))
    checkError "clear V"
    loadIdentity
    checkError "clear I"
    ortho2D 0 (fromIntegral w) 0 (fromIntegral h)
    checkError "clear O"
    clearColor $= Color4 0.5 0.5 0.5 1
    checkError "clear CC"
    clear [ColorBuffer]
    checkError "clear C"
    currentProgram $= Just (combineProg s)
    checkError "shader 1"
    lsheet0 <- get $ uniformLocation (combineProg s) "sheet0"
    checkError "shader 2a"
    lsheet1 <- get $ uniformLocation (combineProg s) "sheet1"
    checkError "shader 2b"
    lblend  <- get $ uniformLocation (combineProg s) "blend"
    checkError "shader 2c"
    uniform lsheet0 $= TexCoord1 (0 :: GLint)
    checkError "shader 3a"
    uniform lsheet1 $= TexCoord1 (1 :: GLint)
    checkError "shader 3b"
    uniform lblend $= TexCoord1 (realToFrac (blendFactor s) :: GLfloat)
    checkError "shader 3c"
    activeTexture $= TextureUnit 0
    checkError "shader 4"
    textureBinding Texture2D $= Just (tsheet0 s)
    checkError "shader 5"
```

```
    activeTexture $= TextureUnit 1
    checkError "shader 6"
    textureBinding Texture2D $= Just (tsheet1 s)
    checkError "shader 7"
    let t x0 y0 = texCoord $ TexCoord2 (0.5 + x' :: GLdouble) (0.5 + y' :: GLdouble)
          where
            p = realToFrac . sqrt . aspect . imageViewport . image $ s
            x = k * x0 * p
            y = k * y0 / p
            a = - rotationA s
            co = realToFrac $ cos a
            si = realToFrac $ sin a
            x' =  co * x + si * y
            y' = -si * x + co * y
        v :: Int -> Int -> IO ()
        v x y = vertex $ Vertex2
          (fromIntegral x :: GLdouble) (fromIntegral y :: GLdouble)
        k = 0.125 * realToFrac (0.5 ** zoomPhase s) :: GLdouble
    renderPrimitive Quads $ do
      t (-1) 1 >> v 0 h
      t (-1) (-1) >> v 0 0
      t 1 (-1) >> v w 0
      t 1 1 >> v w h
    checkError "quad"
    textureBinding Texture2D $= Nothing
    checkError "shader 8"
    activeTexture $= TextureUnit 0
    checkError "shader 9"
    textureBinding Texture2D $= Nothing
    checkError "shader A"
    currentProgram $= Nothing
    checkError "shader B"

    -- save state
    m <- get matrixMode
    matrixMode $= Modelview 0
    colour0 <- get currentColor
    -- pixel mapping
    let i = image s
        locP = locationPixel (imageWindow i) (imageViewport i) (imageLocation i)
        locL (a, b) = [locP a, locP b]
    -- draw lines
    forM_ (imageLines i) $ \l -> do
      let ps = concatMap locL (lineSegments l)
          Colour r g b = lineColour l
      currentColor $= Color4 (realToFrac r) (realToFrac g) (realToFrac b) 1
      renderPrimitive Lines $ forM_ ps $ \(x, y) -> do
        vertex $ Vertex2 (realToFrac x) (fromIntegral h - realToFrac y :: GLdouble)
    -- draw labels
    forM_ (imageLabels i) $ \l -> do
      let (x, y) = locP (labelCoords l)
          Colour r g b = labelColour l
      currentColor $= Color4 (realToFrac r) (realToFrac g) (realToFrac b) 1
      translate (Vector3 (realToFrac x) (fromIntegral h - realToFrac y) (0 :: GLdouble))
```

```
      renderFont (font s) (labelText l) All
      translate (Vector3 (-realToFrac x) (realToFrac y - fromIntegral h) (0 :: ↙
          ↳ GLdouble))
245   -- restore state
      currentColor $= colour0
      matrixMode $= m

      s'' <- readIORef iR
250   when (sheetCount0 s'' + sheetCount1 s'' >= sheetCountTarget s'') $ do
        case doneCallback s of
          Nothing -> return ()
          Just act -> do
            atomicModifyIORef iR $ \s' -> (s'{ doneCallback = Nothing, abortCallback↙
                ↳ = Nothing }, ())
255         act
      prune iR
      checkError "display end"

    clearSheet :: GruffImage -> Bool -> IO ()
260 clearSheet s b = do
      checkError "clearSheet begin"
      let tw' = texels $ sheetSize s
          th' = tw'
          tw = if b then tw' * 2 else tw'
265       th = if b then th' * 2 else th'
          tsheet = (if b then tsheet1 else tsheet0) s
      bindFBO (fbo s) tsheet
      checkError "clearSheet fbo bind"
      GL.viewport $=
270     (GL.Position 0 0, GL.Size (fromIntegral tw) (fromIntegral th))
      checkError "clearSheet V"
      loadIdentity
      checkError "clearSheet I"
      clearColor $= Color4 0 0 1 1
275   checkError "clearSheet CC"
      clear [ColorBuffer]
      checkError "clearSheet C"
      unbindFBO
      checkError "clearSheet fbo unbind"
280   textureBinding Texture2D $= Just tsheet
      checkError "clearSheet tex bind"
      glGenerateMipmap gl_TEXTURE_2D
      checkError "clearSheet tex mipmap"
      textureBinding Texture2D $= Nothing
285   checkError "clearSheet tex unbind"
      checkError "clear end"

    renderSheet :: GruffImage -> Bool -> IO ([(Complex Int, Quad)], Int)
    renderSheet s b = do
290   checkError "renderSheet begin"
      let tw' = texels $ sheetSize s
          th' = tw'
          tw = if b then tw' * 2 else tw'
          th = if b then th' * 2 else th'
295       tsheet = (if b then tsheet1 else tsheet0) s
          vquads = (if b then snd else fst) (viewQuads s)
      if null vquads then return ([], 0) else do
```

```
      bindFBO (fbo s) tsheet
      checkError "renderSheet fbo bind"
      GL.viewport $=
        (GL.Position 0 0, GL.Size (fromIntegral tw) (fromIntegral th))
      checkError "renderSheet V"
      loadIdentity
      checkError "renderSheet I"
      ortho2D 0 (fromIntegral tw) 0 (fromIntegral th)
      checkError "renderSheet O"
      currentProgram $= Just (prog s)
      checkError "renderSheet shader 1"
      lde <- get $ uniformLocation (prog s) "de"
      checkError "renderSheet shader u1"
      lit <- get $ uniformLocation (prog s) "it"
      checkError "renderSheet shader u2"
      ltt <- get $ uniformLocation (prog s) "tt"
      checkError "renderSheet shader u3"
      lint <- get $ uniformLocation (prog s) "interior"
      checkError "renderSheet shader u4"
      lbrd <- get $ uniformLocation (prog s) "border"
      checkError "renderSheet shader u5"
      lext <- get $ uniformLocation (prog s) "exterior"
      checkError "renderSheet shader u6"
      uniform lde $= TexCoord1 (0 :: GLint)
      checkError "renderSheet shader t1"
      uniform lit $= TexCoord1 (1 :: GLint)
      checkError "renderSheet shader t2"
      uniform ltt $= TexCoord1 (2 :: GLint)
      checkError "renderSheet shader t3"
      let (ci, cb, ce) = fromColours . imageColours . image $ s
      uniform lint $= ci
      checkError "renderSheet shader s1"
      uniform lbrd $= cb
      checkError "renderSheet shader s2"
      uniform lext $= ce
      checkError "renderSheet shader s3"
      todo <- foldM (\a t -> do a' <- drawQuad (tiles s) t ; return (a' ++ a)) [] ↙
          ↳ vquads
      checkError "renderSheet drawQuads"
      currentProgram $= Nothing
      checkError "renderSheet shader 2"
      unbindFBO
      checkError "renderSheet fbo unbind"
      let upped = length vquads - length todo
      when (upped > 0) $ do
        textureBinding Texture2D $= Just tsheet
        checkError "renderSheet tex bind"
        glGenerateMipmap gl_TEXTURE_2D
        checkError "renderSheet tex mipmap"
        textureBinding Texture2D $= Nothing
        checkError "renderSheet tex unbind"
      checkError "renderSheet end"
      return (todo, upped)

drawQuad :: Map Quad TextureObject -> (Complex Int, Quad) -> IO [(Complex Int, ↙
    ↳ Quad)]
drawQuad m ijq@(i :+ j, q) = case q `M.lookup` m of
```

```
    Nothing -> return [ijq]
    Just tde -> do
      checkError "drawQuad begin"
      let t x y = texCoord $ TexCoord2 (x :: GLdouble) (y :: GLdouble)
          v x y = vertex $ Vertex2 (x :: GLdouble) (y :: GLdouble)
          x0 = fromIntegral i
          y0 = fromIntegral j
          x1 = x0 + fromIntegral tileSize
          y1 = y0 + fromIntegral tileSize
      activeTexture $= TextureUnit 0
      checkError "drawQuad tex 0"
      textureBinding Texture2D $= Just tde
      checkError "drawQuad tex 0b"
      renderPrimitive Quads $ do
        color $ Color3 1 1 (1::GLdouble)
        t 0 1 >> v x0 y1
        t 0 0 >> v x0 y0
        t 1 0 >> v x1 y0
        t 1 1 >> v x1 y1
      checkError "drawQuad render"
      activeTexture $= TextureUnit 0
      checkError "drawQuad tex 4"
      textureBinding Texture2D $= Nothing
      checkError "drawQuad tex 5b"
      return []

iReshape :: IORef GruffImage -> Maybe Double -> Size -> IO ()
iReshape iR ms size'@(Size w h) = do
  s' <- readIORef iR
  log s' Debug $ "reshapeCallback " ++ show size'
  let ss = case ms of Nothing -> supersamples (imageWindow (image s')) ; Just r ↙
      ↳ -> r
  atomicModifyIORef iR $ \s -> (s{ image = (image s)
    { imageWindow = (imageWindow (image s)){ width = w, height = h, supersamples↙
        ↳ = ss }
    , imageViewport = (imageViewport (image s)){ aspect = fromIntegral w / ↙
        ↳ fromIntegral h }
    } }, ())
  s'' <- readIORef iR
  unless (sheetSize s' == sheetSize s'') (reallocateBuffers iR)
  case Browser.reshapeCallback s' of
    Nothing -> return ()
    Just act -> act w h

iMouse :: IORef GruffImage -> Key -> KeyState -> [Modifier] -> Position -> IO ()
iMouse sR (MouseButton but) Down mods p@(Position x y) | but `elem` buts = do
  let shift = Shift   `elem` mods
      ctrl  = Control `elem` mods
      m | shift && ctrl = SC
        | ctrl          = C
        | shift         = S
        | otherwise     = U
      b = case but of
        LeftButton   -> L
        MiddleButton -> M
        RightButton  -> R
        _ -> error "weasels!!!!111one"
```

```
  s <- readIORef sR
  log s Debug $ "mouse " ++ show (b, m) ++ " " ++ show p
  case (b, m) `M.lookup` mouseCallback s of
    Nothing -> return ()
    Just cb -> do
      let i = image s
          c = pixelLocation (imageWindow i) (imageViewport i) (imageLocation i) ↙
              ↳ x y
      cb c i
  where buts = [LeftButton, MiddleButton, RightButton]
iMouse _ _ _ _ _ = return ()

quadDistance :: Quad -> Quad -> (Int, Double)
quadDistance q0 q1 =
  let Quad{ quadLevel = l0, quadWest = r0, quadNorth = i0 } = q0
      Quad{ quadLevel = l, quadWest = r, quadNorth = i} = q1
      dl = sqr (fromIntegral l - fromIntegral l0)
      d x x0
        | l > l0 = fromIntegral $ sqr (x - x0 * 2 ^ (l - l0))
        | l == l0 = fromIntegral $ sqr (x - x0)
        | l < l0 = fromIntegral $ sqr (x0 - x * 2 ^ (l0 - l))
      d _ _ = error "score"
  in (l, dl + d r r0 + d i i0)

sqr :: Num a => a -> a
sqr x = x * x

prune :: IORef GruffImage -> IO ()
prune sR = do
  s0 <- readIORef sR
  let cacheSize = M.size (tiles s0)
  when (cacheSize > cacheSizeMax s0) $ do
    checkError "prune begin"
    log s0 Debug . concat $
      [ "pruning texture cache "
      , show cacheSize, " > ", show (cacheSizeMax s0)
      , " --> ", show (cacheSizeMin s0)
      ]
    bad <- atomicModifyIORef sR $ \s ->
      let Just q0 = originQuad (imageLocation (image s)) (sheetSize s)
          score = quadDistance q0
          o = comparing (score . fst)
          (good, bad)
            = splitAt (cacheSizeMin s) . sortBy o . M.toList . tiles $ s
      in (s{ tiles = M.fromList good }, bad)
    deleteObjectNames $ map snd bad
    checkError "prune end"

update :: IORef GruffImage -> IO ()
update sR = do
  s' <- readIORef sR
  log s' Debug $ "updateCallback "
  (todo', mstop) <- atomicModifyIORef sR $ \s ->
    let i = image s
        vq@(qs0, qs1) = fromMaybe ([],[]) $
          visibleQuads (imageWindow i) (imageViewport i) (imageLocation i) (↙
              ↳ sheetSize s)
```

```
          qs = qs0 ++ qs1
          todo = S.filter (`M.notMember` tiles s) (S.fromList (map snd qs))
      in  (s{ viewQuads = vq, sheetCountTarget = length qs, worker = Nothing }, (↙
          ↳ todo, worker s))
  -- cancel in-progress jobs that aren't still needed
  fromMaybe (return ()) mstop
  stop <- computeTiles (log s' Debug) (doneTile sR) (cacheDir s') todo'
  atomicModifyIORef sR $ \s -> (s{ worker = Just stop }, ())

doneTile :: IORef GruffImage -> Tile -> IO ()
doneTile sR t = do
  s <- readIORef sR
  let qu = queue s
  ts <- takeMVar qu
  putMVar qu (t:ts)

timer :: IORef GruffImage -> IO ()
timer sR = do
  s <- readIORef sR
  mtls <- tryTakeMVar (queue s)
  case mtls of
    Just tls -> do
      putMVar (queue s) tls
      unless (null tls) $ postRedisplay (gl s)
    _ -> return ()

upload :: Ptr CUChar -> IO TextureObject
upload p = do
  checkError "upload begin"
  [tex] <- genObjectNames 1
  checkError "upload 1"
  texture Texture2D $= Enabled
  checkError "upload 2"
  textureBinding Texture2D $= Just tex
  checkError "upload 3"
  glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_LUMINANCE)
    (fromIntegral tileSize) (fromIntegral tileSize) 0
      gl_LUMINANCE gl_UNSIGNED_BYTE p
  checkError "upload 4"
  textureFilter Texture2D $= ((Nearest, Nothing), Nearest)
  checkError "upload 5"
  textureWrapMode Texture2D GL.S $= (Repeated, ClampToEdge)
  checkError "upload 6"
  textureWrapMode Texture2D GL.T $= (Repeated, ClampToEdge)
  checkError "upload 7"
  textureBinding Texture2D $= Nothing
  checkError "upload 7"
  texture Texture2D $= Disabled
  checkError "upload end"
  return tex

msPerFrame :: Int
msPerFrame = 200

data Browser = Browser
  { browserWindow :: GTK.Window
  , browserGL :: GLUTGtk
```

```
    , browserRender :: Image -> IO () -> IO () -> IO ()
    , browserAbort  :: IO ()
    , browserSaveImage :: FilePath -> IO ()
    , browserSetExitCallback :: IO () -> IO ()
    , browserSetReshapeCallback :: (Int -> Int -> IO ()) -> IO ()
    , browserSetMouseCallback :: MouseCallbacks -> IO ()
    }

browserRenders :: Browser -> [(Image, FilePath)] -> IO ()
browserRenders _ [] = print "done"
browserRenders b ((i, f):ifs) = do
  result <- newEmptyMVar
  postGUISync $ do
          browserRender b i
            (browserSaveImage b f >> putMVar result True)
            (putMVar result False)
          postRedisplay (browserGL b)
  r <- takeMVar result
  if r then browserRenders b ifs else print "aborted"

browserNew :: GLUTGtk -> Pixbuf -> Logger -> FilePath -> IO Browser
browserNew gl' icon lg cacheDir' = do
  -- image window
  iw <- windowNew
  let defW = width  defWindow
      defH = height defWindow
  windowSetDefaultSize iw defW defH
  windowSetGeometryHints iw (Nothing `asTypeOf` Just iw)
    (Just (160, 120)) (Just (4096, 4096)) Nothing Nothing Nothing
  set iw
    [ containerBorderWidth := 0
    , containerChild := widget gl'
    , windowIcon  := Just icon
    , windowTitle := "gruff browser"
    ]
  queue' <- newMVar []
  jobs' <- newMVar []
  iR <- newIORef GruffImage
      -- image parameters
    { image = defImage
      -- callbacks
    , exitCallback = Nothing
    , Browser.reshapeCallback = Nothing
    , mouseCallback = M.empty
    , doneCallback = Nothing
    , abortCallback = Nothing
      -- job queue
    , tiles = M.empty
    , queue = queue'
    , jobs = jobs'
    , viewQuads = ([], [])
    , gl = gl'
    , cacheDir = cacheDir'
    , log = Log.log lg
    , worker = Nothing
    , prog = error "prog"
    , fbo = 0
```

```
575       , combineProg = error "combineProg"
          , tsheet0 = TextureObject 0
          , tsheet1 = TextureObject 0
          , sheetCount0 = 0
          , sheetCount1 = 0
580       , sheetCountTarget = maxBound
          , cacheSizeMin = 1000
          , cacheSizeMax = 1500
          , doClear = False
          , font = error "font"
585       }
        realizeCallback gl' $= iRealize iR
        GLUTGtk.reshapeCallback gl' $= iReshape iR Nothing
        displayCallback gl' $= iDisplay iR
        let browserAbort' = do
590           (mwork, act) <- atomicModifyIORef iR $ \s -> (s
                { doneCallback = Nothing
                , abortCallback = Nothing
                , worker = Nothing
                }, (worker s, abortCallback s))
595           fromMaybe (return ()) mwork
              fromMaybe (return ()) act
            atExit = do
              i <- readIORef iR
              case exitCallback i of
600             Nothing -> return ()
                Just act -> act
            browserRender' img done aborted = do
              s <- readIORef iR
              let wr = width  (imageWindow img)
605               hr = height (imageWindow img)
              unless (width  (imageWindow (image s)) == wr &&
                      height (imageWindow (image s)) == hr) $ do
                windowResize iw wr hr
              atomicModifyIORef iR $ \s' -> (s'{ image = img }, ())
610           s' <- readIORef iR
              unless (sheetSize s == sheetSize s') (reallocateBuffers iR)
              atomicModifyIORef iR $ \i -> (i
                { doneCallback = Just done
                , abortCallback = Just aborted
615             , doClear = True
                }, ())
              update iR
            browserSaveImage' fname = do
              s <- readIORef iR
620           writeSnapshot fname (GL.Position 0 0)
                (GL.Size (fromIntegral (width (imageWindow (image s))))
                         (fromIntegral (height (imageWindow (image s)))))
            browserSetExitCallback' act = atomicModifyIORef iR $ \i ->
              ( i{ exitCallback = Just act }, () )
625         browserSetReshapeCallback' act = atomicModifyIORef iR $ \i ->
              ( i{ Browser.reshapeCallback = Just act }, () )
            browserSetMouseCallback' act = atomicModifyIORef iR $ \i ->
              ( i{ mouseCallback = act }, () )
        keyboardMouseCallback gl' $= iMouse iR
630     _ <- timeoutAdd (timer iR >> return True) msPerFrame
        _ <- iw `onDestroy` atExit
```

```
    return Browser
      { browserWindow = iw
      , browserGL = gl'
      , browserRender = browserRender'
      , browserAbort = browserAbort'
      , browserSaveImage = browserSaveImage'
      , browserSetExitCallback = browserSetExitCallback'
      , browserSetReshapeCallback = browserSetReshapeCallback'
      , browserSetMouseCallback = browserSetMouseCallback'
      }

fromColours :: Colours -> (Color3 GLfloat, Color3 GLfloat, Color3 GLfloat)
fromColours c =
  ( fromColour (colourInterior c)
  , fromColour (colourBoundary c)
  , fromColour (colourExterior c)
  )

fromColour :: Colour -> Color3 GLfloat
fromColour (Colour r g b) = Color3 (realToFrac r) (realToFrac g) (realToFrac b)

iRealize :: IORef GruffImage -> IO ()
iRealize iR = do
  checkError "realize begin"
  s <- readIORef iR
  log s Debug "realizeCallback"
  drawBuffer $= BackBuffers
  checkError "realize db"
  glClampColor gl_CLAMP_VERTEX_COLOR gl_FALSE
  checkError "realize clv"
  glClampColor gl_CLAMP_READ_COLOR gl_FALSE
  checkError "realize clr"
  glClampColor gl_CLAMP_FRAGMENT_COLOR gl_FALSE
  checkError "realize clf"
  f <- getDataFileName "minimal.frag"
  prog' <- shader Nothing (Just f)
  checkError "realize s1"
  f' <- getDataFileName "merge.frag"
  combineProg' <- shader Nothing (Just f')
  checkError "realize s2"
  fbo' <- newFBO
  checkError "realize fbo"
  [tsheet0', tsheet1'] <- genObjectNames 2
  checkError "realize sheets"
  -- FIXME hardcoded font path
  font' <- createTextureFont "/usr/share/fonts/truetype/ttf-dejavu/↙
      ↳ DejaVuSansMono-Bold.ttf"
  -- FIXME destroyFont (font s) at exit
  _ <- setFontFaceSize font' 16 72
  atomicModifyIORef iR $ \i -> (i
    { prog = prog'
    , fbo = fbo'
    , combineProg = combineProg'
    , tsheet0 = tsheet0'
    , tsheet1 = tsheet1'
    , sheetCount0 = 0
    , sheetCount1 = 0
```

```
      , sheetCountTarget = maxBound
      , font = font'
690   }, ())
    reallocateBuffers iR
    checkError "realize end"

reallocateBuffers :: IORef GruffImage -> IO ()
695 reallocateBuffers iR = do
    checkError "reallocate begin"
    s <- readIORef iR
    let tw' = texels $ sheetSize s
        th' = tw'
700     ts = [tsheet0 s, tsheet1 s]
    forM_ (ts `zip` [(tw', th'), (tw' * 2, th' * 2)]) $ \(t, (tw, th)) -> do
      texture Texture2D $= Enabled
      checkError "reallocate 1"
      textureBinding Texture2D $= Just t
705   checkError "reallocate 2"
      glTexImage2D gl_TEXTURE_2D 0 (fromIntegral gl_RGBA)
        (fromIntegral tw) (fromIntegral th) 0 gl_RGBA gl_UNSIGNED_BYTE
        nullPtr
      checkError "reallocate 3"
710   textureFilter Texture2D $= ((Linear', Just Linear'), Linear')
      checkError "reallocate 4"
      textureWrapMode Texture2D GL.S $= (Repeated, ClampToEdge)
      checkError "reallocate 5"
      textureWrapMode Texture2D GL.T $= (Repeated, ClampToEdge)
715   checkError "reallocate 6"
      textureBinding Texture2D $= Nothing
      checkError "reallocate 7"
      texture Texture2D $= Disabled
      checkError "reallocate 8"
720 atomicModifyIORef iR $ \s' ->
      (s'{ sheetCount0 = 0, sheetCount1 = 0, sheetCountTarget = maxBound }, ())
    checkError "reallocate end"

newFBO :: IO GLuint
725 newFBO = alloca $ \p -> glGenFramebuffers 1 p >> peek p

bindFBO :: GLuint -> TextureObject -> IO ()
bindFBO f (TextureObject t) = do
    checkError "bindFBO begin"
730 glBindFramebuffer gl_FRAMEBUFFER f
    checkError "bindFBO 1"
    glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D t 0
    checkError "bindFBO 2"
    err <- glCheckFramebufferStatus gl_FRAMEBUFFER
735 when (err /= gl_FRAMEBUFFER_COMPLETE) $
      putStrLn ("OpenGL framebuffer error: " ++ show err)
    checkError "bindFBO end"

unbindFBO :: IO ()
740 unbindFBO = do
    checkError "unbindFBO begin"
    glFramebufferTexture2D gl_FRAMEBUFFER gl_COLOR_ATTACHMENT0 gl_TEXTURE_2D 0 0
    checkError "unbindFBO 1"
    glBindFramebuffer gl_FRAMEBUFFER 0
```

```
745   checkError "unbindFBO end"

checkError :: String -> IO ()
checkError s = do
  err <- glGetError
750   when (err /= 0) $ putStrLn ("OpenGL error (" ++ s ++ "): " ++ show err)
```

31 src/Compute.hs

```
{-# LANGUAGE FlexibleContexts, MultiParamTypeClasses #-}
module Compute (compute) where

import Control.Monad (forM_, forM, liftM2, when)
 5 --import Control.Parallel.Strategies (parMap, rseq)
import Data.Bits (bit)
import Data.Either (partitionEithers)
import Data.IORef (IORef, newIORef, readIORef, writeIORef)
import Data.Ix (range, inRange)
10 import Data.Maybe (catMaybes)
import Data.Monoid (mappend)
import Data.Set (Set)
import qualified Data.Set as S
import Foreign (Ptr, peekElemOff, pokeElemOff)
15 import Foreign.C (CUChar)

--import Numeric.QD (DoubleDouble(DoubleDouble), QuadDouble(QuadDouble))
import Data.Array.BitArray.IO hiding (map)

20 import Fractal.GRUFF

data Iter z d = Iter
  { iterZ :: !(Complex z)
  , iterD :: !(Complex d)
25   , iterN :: !Int
  , iterI :: !Int
  , iterJ :: !Int
  }

30 instance Eq (Iter z d) where
  a == b = iterI a == iterI b && iterJ a == iterJ b
  a /= b = iterI a /= iterI b || iterJ a /= iterJ b

instance Ord (Iter z d) where
35   a `compare` b = (iterI a `compare` iterI b) `mappend` (iterJ a `compare` iterJ↙
       ↳ b)

initialIter :: (Num z, Num (Complex z), Num d, Num (Complex d)) => Int -> Int ->↙
    ↳ Iter z d
initialIter i j = Iter{ iterZ = 0, iterD = 0, iterN = 0, iterI = i, iterJ = j }

40 tileBounds :: ((Int, Int), (Int, Int))
tileBounds = ((0, 0), (255, 255))

compute :: NaturalNumber p => Ptr CUChar -> Complex (VFixed p) -> Int -> Int -> ↙
    ↳ IO Int
compute io_de in_c@(cx0:+_) in_level in_iters = do
45   maxIters <- newIORef (-1)
```

```
    active <- newArray tileBounds False
    finished <- newArray tileBounds False
    let prec = precision cx0
        loop'
          ::(NaturalNumber p,
             Floating d,
             Fractional z,
             Num (Complex z),
             Num (Complex d),
             Real z,
             Real d,
             ComplexRect (Complex z) z,
             ComplexRect (Complex d) d,
             RealToFrac' (VFixed p) z,
             RealToFrac' z d,
             RealToFrac' d Double)
            => Complex (VFixed p) -> Set (Iter z d) -> IO Int
        loop' c = loop c active finished maxIters False 1024
        init'
          :: (Num z, Num (Complex z), Num d, Num (Complex d))
          => z -> d -> IO (Set (Iter z d))
        init' = initialize active finished
    case prec of
      _ | prec < 48  -> loop' in_c =<< init' double double
--      | prec < 96  -> loop' in_c =<< init' ddouble double
--      | prec < 192 -> loop' in_c =<< init' qdouble double
      | prec < 480 -> loop' in_c =<< init' cx0 double
      | otherwise  -> loop' in_c =<< init' cx0 f24

    where

      double :: Double
      double = 0
{-
      ddouble :: DoubleDouble
      ddouble = 0
      qdouble :: QuadDouble
      qdouble = 0
-}
      initialize
        :: (Num z, Num (Complex z), Num d, Num (Complex d))
        => IOBitArray (Int, Int) -> IOBitArray (Int, Int) -> z -> d -> IO (Set (↙
            ↳ Iter z d))
      initialize active finished _ _ = (S.fromList . map (uncurry initialIter) . ↙
          ↳ catMaybes) `fmap` mapM (initialize1 active finished) (range tileBounds↙
          ↳ )

      initialize1 :: IOBitArray (Int, Int) -> IOBitArray (Int, Int) -> (Int, Int) ↙
          ↳ -> IO (Maybe (Int, Int))
      initialize1 active finished ij = do
        me <- peekElemOff io_de (off ij)
        case me of
          _ | me == 255 -> writeArray finished ij True >> return Nothing
            | me >  0   -> writeArray active   ij True >> return (Just ij)
            | otherwise ->                                return Nothing
{-
              do
```

```
                edge <- fmap or $ forM (range ((i-1,j-1),(i+1,j+1))) $ \iijj -> do
                  case inRange tileBounds iijj of
                    False -> return False
                    True  -> (0 <) `fmap` peekElemOff io_de (off iijj)
                case edge of
                  False -> return Nothing
                  True  -> return (Just ij)
-}

    off :: (Int, Int) -> Int
    off (i, j) = 256 * j + i

    loop
      :: (NaturalNumber p, RealToFrac' (VFixed p) z, RealToFrac' z d, RealToFrac
          ↪ ' d Double, Real z, Real d, Fractional z, Fractional d, Floating d,
          ↪ ComplexRect (Complex z) z, ComplexRect (Complex d) d, Num (Complex z
          ↪ ), Num (Complex d))
      => Complex (VFixed p) -> IOBitArray (Int, Int) -> IOBitArray (Int, Int) ->
          ↪  IORef Int -> Bool -> Int -> Set (Iter z d) -> IO Int
    loop c0 active finished maxIters progressed step_iters activeSet
      | S.null activeSet = done maxIters activeSet
      | step_iters >= in_iters = done maxIters activeSet
      | otherwise = loop' False S.empty activeSet
      where
        loop' escaped unescaped todoss
          | S.null todoss && not escaped && progressed = done maxIters unescaped
          | S.null todoss && not escaped = loop c0 active finished maxIters
            ↪ progressed (2 * step_iters) unescaped
          | S.null todoss &&     escaped = loop c0 active finished maxIters True
            ↪  (2 * step_iters) unescaped
          | otherwise = do
              let (escapees, unescapees) = partitionEithers . map (\todo ->
                  ↪ calculate step_iters (coord c0 todo) todo) . S.toList $
                  ↪ todoss
              nexts <- (S.fromList . concat) `fmap` mapM (finalize active
                  ↪ finished maxIters) escapees
              let olds = S.fromList unescapees `S.union` unescaped
                  news = nexts `S.difference` olds -- left biased
                  escaped' = escaped || not (S.null news)
              loop' escaped' olds news

    escapeRadius2 :: Num z => z
    escapeRadius2 = 65536

    scaleI :: Integer
    scaleI = 32 * bit in_level

    scale' :: Fractional z => z
    scale' = recip (fromInteger scaleI)

    scale :: Num d => d
    scale = fromInteger scaleI

    coord :: (NaturalNumber p, RealToFrac' (VFixed p) z) => Complex (VFixed p)
        ↪ -> Iter z d -> Complex z
    coord c0 iter = fmap realToFrac' $
      c0 + scale' .* (fromIntegral (iterI iter) :+ fromIntegral (iterJ iter))
```

```
    calculate :: (RealToFrac' z d, Real z, Real d, Fractional z, Fractional d, ↙
        ↳ ComplexRect (Complex z) z, Num (Complex z), Num (Complex d)) => Int ->↙
        ↳  Complex z -> Iter z d -> Either (Iter z d) (Iter z d)
    calculate step_iters c iter
      | magnitudeSquared (iterZ iter) >= escapeRadius2 = Left iter
      | step_iters <= 0 = Right iter
      | otherwise = calculate (step_iters - 1) c (step c iter)

    step :: (RealToFrac' z d, Real z, Real d, Fractional z, Fractional d, ↙
        ↳ ComplexRect (Complex z) z, Num (Complex z), Num (Complex d)) => ↙
        ↳ Complex z -> Iter z d -> Iter z d
    step c i@Iter{ iterZ = z, iterD = d, iterN = n } =
      i{ iterZ = sqr z + c, iterD = let zd = fmap realToFrac' z * d in zd + zd +↙
          ↳  1, iterN = n + 1 }

    finalize
      :: (RealToFrac' z d, RealToFrac' d Double, Real z, Real d, Fractional z, ↙
          ↳ Fractional d, Floating d, ComplexRect (Complex d) d, Num (Complex z)↙
          ↳ , Num (Complex d))
      => IOBitArray (Int, Int) -> IOBitArray (Int, Int) -> IORef Int -> Iter z d↙
          ↳  -> IO [Iter z d]
    finalize active finished maxIters Iter{ iterZ = z, iterD = d, iterN = n, ↙
        ↳ iterI = i, iterJ = j } = do
      m <- readIORef maxIters
      when (n > m) $ writeIORef maxIters n
      let k = off (i, j)
          z' = fmap realToFrac' z `asTypeOf` d
          z2 = magnitudeSquared z'
          diste :: Double
          diste = realToFrac' $ log z2 * sqrt (z2 / magnitudeSquared d) * scale
          disteB
            | diste > 4 = 255
            | otherwise = round . clamp 1 254 . (* 255) . (/ 4) $ diste
      pokeElemOff io_de k disteB
      writeArray finished (i, j) True
      writeArray active (i, j) False
      fmap catMaybes $ forM (liftM2 (,) [i-1,i,i+1] [j-1,j,j+1]) $ \iijj@(ii, jj↙
          ↳ ) -> do
        if not (inRange tileBounds iijj) then return Nothing else do
          fin <- readArray finished iijj
          act <- readArray active iijj
          if fin || act then return Nothing else do
            writeArray active iijj True
            return $ Just (initialIter ii jj)

    done :: IORef Int -> Set (Iter z d) -> IO Int
    done maxIters activeSet = do
      forM_ (S.toList activeSet) $ \Iter{ iterI = i, iterJ = j } -> do
        pokeElemOff io_de (off (i, j)) 0
      readIORef maxIters

clamp :: Ord a => a -> a -> a -> a
clamp mi ma x = mi `max` x `min` ma

class RealToFrac' a b where
  realToFrac' :: a -> b
```

```
instance RealToFrac' Double Double where
  realToFrac' = id

instance NaturalNumber p => RealToFrac' (VFixed p) (VFixed p) where
  realToFrac' = id

instance NaturalNumber p => RealToFrac' (VFloat p) (VFloat p) where
  realToFrac' = id

instance NaturalNumber p => RealToFrac' (VFixed p) Double where
  realToFrac' = realToFrac

instance (NaturalNumber p, NaturalNumber q) => RealToFrac' (VFixed p) (VFloat q)↙
    ↳ where
  realToFrac' = realToFrac -- FIXME better way?

instance NaturalNumber p => RealToFrac' (VFloat p) Double where
  realToFrac' = uncurry encodeFloat . decodeFloat
{-
instance RealToFrac' DoubleDouble Double where
  realToFrac' (DoubleDouble x _) = realToFrac x

instance RealToFrac' QuadDouble Double where
  realToFrac' (QuadDouble x _ _ _) = realToFrac x

instance NaturalNumber p => RealToFrac' (VFixed p) DoubleDouble where
  realToFrac' = realToFrac

instance NaturalNumber p => RealToFrac' (VFixed p) QuadDouble where
  realToFrac' = realToFrac
-}
```

32 src/Fractal/GRUFF.hs

```
{- |
Module      :  Fractal.GRUFF
Copyright   :  (c) Claude Heiland-Allen 2011,2012
License     :  GPL-2

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Support library for rendering animations of the Mandelbrot Set fractal
using the gruff executable.

An example program might have a structure similar to:

  > import Fractal.GRUFF
  >
  > animation :: Animation
  > animation = ...
  >
  > main :: IO ()
  > main = defaultMain animation
```

```
with its output fed to the gruff executable.

See also:

  * the @ruff@ package for feature location algorithms; and

  * the @gruff-examples@ package for concrete examples.

-}
module Fractal.GRUFF
  ( module Fractal.Mandelbrot
  , Animation, Image(..), Window(..), Viewport(..), Location(..)
  , Colours(..), Colour(..), Label(..), labelAppend, labelPrepend
  , Line(..), fromScreenCoords, toScreenCoords, defaultMain
  , fromVComplex
  ) where

import System.IO (hFlush, stdout)

import Fractal.Mandelbrot

fromVComplex :: NaturalNumber n => CF n -> Complex Rational
fromVComplex = fmap toRational . toComplex

-- | Animation specification
type Animation = [(Image, FilePath)]

-- | Image specification.
data Image = Image
  { imageWindow    :: !Window
  , imageViewport  :: !Viewport
  , imageLocation  :: !Location
  , imageColours   :: !Colours
  , imageLabels    :: [Label]
  , imageLines     :: [Line]
  }
  deriving (Read, Show, Eq)

-- | Window specification.
data Window = Window
  { width  :: !Int
  , height :: !Int
  , supersamples :: !Double
  }
  deriving (Read, Show, Eq)

-- | Viewport specification
data Viewport = Viewport
  { aspect :: !Double -- width / height
  , orient :: !Double -- in radians
  }
  deriving (Read, Show, Eq)

-- | Location specification (center of view).
data Location = Location
  { center :: !(Complex Rational)
  , radius :: !Rational
```

```
    }
    deriving (Read, Show, Eq)

-- | Colour specification.
data Colours = Colours
    { colourInterior :: !Colour
    , colourBoundary :: !Colour
    , colourExterior :: !Colour
    }
    deriving (Read, Show, Eq)

-- | RGB colour data (each channel between 0 and 1).
data Colour = Colour !Double !Double !Double
    deriving (Read, Show, Eq)

-- | Labels can be added to points in the complex plane.
data Label = Label
    { labelCoords :: !(Complex Rational)
    , labelColour :: !Colour
    , labelText   :: String
    }
    deriving (Read, Show, Eq)

-- | Append a label to an image.
labelAppend :: Complex Rational -> Colour -> String -> Image -> Image
labelAppend c v t i = i{ imageLabels = imageLabels i ++ [Label{ labelCoords = c,↙
    ↳  labelColour = v, labelText = t }] }

-- | Prepend a label to an image.
labelPrepend :: Complex Rational -> Colour -> String -> Image -> Image
labelPrepend c v t i = i{ imageLabels = [Label{ labelCoords = c, labelColour = v↙
    ↳ , labelText = t }] ++ imageLabels i }

-- | Line segments.
data Line = Line
    { lineSegments :: [(Complex Rational, Complex Rational)]
    , lineColour :: !Colour
    }
    deriving (Read, Show, Eq)

-- | Transform a point from screen coordinates.
fromScreenCoords :: (Fractional r, Real r) => Window -> Viewport -> Location -> ↙
    ↳ Complex Double -> Complex r
fromScreenCoords w v l = \(x :+ y) ->
  let x1 = (2 * x - w') / w' * p
      y1 = (h' - 2 * y) / h' / p
      x2 =  co * x1 + si * y1
      y2 = -si * x1 + co * y1
      x3 = r * toRational x2
      y3 = r * toRational y2
      x4 :+ y4 = (x3 :+ y3) + c
  in  fromRational x4 :+ fromRational y4
  where
    p = sqrt (aspect v)
    w' = fromIntegral (width  w)
    h' = fromIntegral (height w)
    a = - orient v
```

```
      co = cos a
      si = sin a
      r = toRational $ radius l
      c = center l

-- | Transform a point to screen coordinates.
toScreenCoords :: Real r => Window -> Viewport -> Location -> Complex r -> ↙
    ↘ Complex Double
toScreenCoords w v l = \(x' :+ y') ->
  let x = (x1 / p * w' + w') /   2
      y = (y1 * p * h' - h') / (-2)
      x1 =  co * x2 + si * y2
      y1 = -si * x2 + co * y2
      x2 = fromRational (x3 / r)
      y2 = fromRational (y3 / r)
      x3 :+ y3 = (toRational x' :+ toRational y') - c
  in  x :+ y
  where
    p = sqrt (aspect v)
    w' = fromIntegral (width  w)
    h' = fromIntegral (height w)
    a = orient v
    co = cos a
    si = sin a
    r = toRational (radius l)
    c = center l

-- | Serialize an animation to stdout.
defaultMain :: Animation -> IO ()
defaultMain a = do
  mapM_ (\f -> print f >> hFlush stdout) a
```

33 src/GLUTGtk.hs

```
module GLUTGtk where

import Control.Monad (join)
import Control.Monad.Trans (liftIO)
import Data.IORef (IORef, newIORef, readIORef)
import Graphics.UI.Gtk hiding (Size)
import Graphics.UI.Gtk.OpenGL

type RealizeCallback = IO ()
type ReshapeCallback = Size -> IO ()
type DisplayCallback = IO ()
type KeyboardMouseCallback = Key -> KeyState -> [Modifier] -> Position -> IO ()

data Size = Size Int Int
  deriving (Eq, Ord, Show)

data Position = Position Double Double
  deriving (Eq, Ord, Show)

data KeyState = Down | Up
  deriving (Eq, Ord, Show)

data Key = MouseButton MouseButton
```

```
  deriving (Eq, Show)

data GLUTGtk = GLUTGtk
  { realizeCallback :: IORef RealizeCallback
  , reshapeCallback :: IORef ReshapeCallback
  , displayCallback :: IORef DisplayCallback
  , keyboardMouseCallback :: IORef KeyboardMouseCallback
  , postRedisplay :: IO ()
  , widget :: EventBox
  }

glut :: Size -> IO GLUTGtk
glut (Size width height) = do
  realizeCallback' <- newIORef $ return ()
  displayCallback' <- newIORef $ return ()
  reshapeCallback' <- newIORef $ \_ -> return ()
  keyboardMouseCallback' <- newIORef $ \_ _ _ _ -> return ()
  config <- glConfigNew [ GLModeRGBA, GLModeDouble ]
  canvas <- glDrawingAreaNew config
  widgetSetSizeRequest canvas width height
  eventb <- eventBoxNew
  set eventb [ containerBorderWidth := 0, containerChild := canvas ]
  _ <- onRealize canvas $ withGLDrawingArea canvas $ \_ -> join (readIORef ↙
    ↳ realizeCallback')
  _ <- canvas `on` configureEvent $ tryEvent $ do
    (w, h) <- eventSize
    liftIO $ do
      cb <- readIORef reshapeCallback'
      cb (Size w h)
  _ <- canvas `on` exposeEvent $ tryEvent $ liftIO $ withGLDrawingArea canvas $ ↙
    ↳ \gl -> do
    join (readIORef displayCallback')
    glDrawableSwapBuffers gl
  let handleButton s = do
        b <- eventButton
        (x, y) <- eventCoordinates
        ms <- eventModifier
        liftIO $ do
          cb <- readIORef keyboardMouseCallback'
          cb (MouseButton b) s ms (Position x y)
  _ <- eventb `on` buttonPressEvent   $ tryEvent $ handleButton Down
  _ <- eventb `on` buttonReleaseEvent $ tryEvent $ handleButton Up
  return $ GLUTGtk
    { realizeCallback = realizeCallback'
    , displayCallback = displayCallback'
    , reshapeCallback = reshapeCallback'
    , keyboardMouseCallback = keyboardMouseCallback'
    , postRedisplay = widgetQueueDraw canvas
    , widget = eventb
    }
```

34 src/gruff.hs

```
module Main (main) where

import Prelude hiding (catch, log)
import Control.Concurrent (forkIO)
```

```
import Control.Monad (forM_)
import Data.IORef (IORef, newIORef, readIORef, atomicModifyIORef, writeIORef)
import Data.Maybe (mapMaybe)
import Graphics.UI.Gtk hiding (get, Region, Size, Window, Viewport)
import qualified Graphics.UI.Gtk as GTK
import Graphics.UI.Gtk.OpenGL (initGL)
import Numeric.QD (QuadDouble)
import System.Directory (getAppUserDataDirectory, createDirectoryIfMissing)
import System.FilePath ((</>))

import Fractal.GRUFF (Complex((:+)), realPart, imagPart)

import Paths_gruff (getDataFileName)
import Browser (Browser(..), browserNew, browserRenders)
import Progress (progressNew)
import Interact (mouseCallbacks)
import View
  ( Image(..), Location(..), Viewport(..), Window(..)
  , Colours(..), Colour(..), defImage, defLocation
  )
import GLUTGtk (glut, Size(Size), postRedisplay)
import Logger (logger, LogLevel(Debug))
import qualified Logger as Log
import Utils (safeRead, catchIO)

exit :: (LogLevel -> String -> IO ()) -> FilePath -> Gruff -> IO ()
exit lg stateFile g = do
  lg Debug "exitCallback"
  aDoSave stateFile g
  mainQuit

data GruffGUI = GruffGUI
  -- buttons
  { bHome
  , bLoad
  , bSave
  , bStop
  , bClear                  :: Button
  -- entries
  , eRealM, eRealE
  , eImagM, eImagE
  , eSizeM, eSizeE
  , eRota
  , eWidth
  , eHeight
  , eSamples                :: Entry
  -- colour pickers
  , cInterior
  , cBorder
  , cExterior               :: ColorButton
  -- windows
  , wMain
  , wImage                  :: GTK.Window
  }

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

```
    -- contexts
    _ <- initGUI
    _ <- initGL
    gl' <- glut minSize
    -- directories
    appDir <- getAppUserDataDirectory "gruff"
    let cacheDir' = appDir </> "cache"
        logDir    = appDir </> "log"
        stateFile = appDir </> "state.gruff"
    createDirectoryIfMissing False appDir
    createDirectoryIfMissing False logDir
    lg <- logger logDir
    icon <- pixbufNewFromFile =<< getDataFileName "icon.png"
    browser <- browserNew gl' icon lg cacheDir'
    let iw = browserWindow browser
    sg <- sizeGroupNew SizeGroupHorizontal

    let spacing = 2

        entryNewWithMnemonic m = entryNewWithMnemonic' m 30

        entryNewWithMnemonic' m wc = do
          e <- entryNew
          entrySetWidthChars e wc
          l <- labelNewWithMnemonic m
          labelSetMnemonicWidget l e
          sizeGroupAddWidget sg l
          h <- hBoxNew False spacing
          boxPackStart h l PackNatural 0
          boxPackStartDefaults h e
          return (e, h)

        entryNewExponent = do
          e <- entryNew
          entrySetWidthChars e 4
          l <- labelNew (Just "e")
          h <- hBoxNew False spacing
          boxPackStart h l PackNatural 0
          boxPackStart h e PackNatural 0
          return (e, h)

        frameNewWithContents box t r ws = do
          f <- frameNew
          frameSetLabel f t
          frameSetLabelAlign f (if r then 1 else 0) 0.5
          v <- box False spacing
          forM_ ws $ boxPackStartDefaults v
          set f [ containerChild := v ]
          return f

    b01@bHome' <- buttonNewWithLabel "Home"
    b02@bLoad' <- buttonNewWithLabel "Load"
    b03@bSave' <- buttonNewWithLabel "Save"
    b04@bStop' <- buttonNewWithLabel "Stop"
    b05@bClear' <- buttonNewWithLabel "Clear"

    (eRealM', fc1m) <- entryNewWithMnemonic "_Real"
```

```
    (eRealE', fc1e) <- entryNewExponent
    (eImagM', fc2m) <- entryNewWithMnemonic "_Imag"
    (eImagE', fc2e) <- entryNewExponent
    (eSizeM', fc3m) <- entryNewWithMnemonic "Si_ze"
    (eSizeE', fc3e) <- entryNewExponent
    (eRota', fvR)    <- entryNewWithMnemonic' "R_otation" 15
    (eWidth', fvW)   <- entryNewWithMnemonic' "_Width"     5
    (eHeight', fvH)  <- entryNewWithMnemonic' "_Height"    5
    (eSamples', fvS) <- entryNewWithMnemonic' "_Samples"   3

    cInterior' <- colorButtonNewWithColor red
    cBorder'   <- colorButtonNewWithColor black
    cExterior' <- colorButtonNewWithColor white

    labelColourR <- newIORef (fromColor blue)
    cLabels'   <- colorButtonNewWithColor blue
    _ <- cLabels' `onColorSet` (colorButtonGetColor cLabels' >>= writeIORef ↙
        ↳ labelColourR . fromColor)

    fb <- frameNewWithContents  vBoxNew "Actions" False [b01, b02, b03, b04, b05]
    fh <- frameNewWithContents  hBoxNew "Colours" True [cInterior', cBorder', ↙
        ↳ cExterior', cLabels']
    fv <- frameNewWithContents  hBoxNew "Viewport" True [fvR, fvW, fvH, fvS]

    let packMantissaExponent m e = do
          h <- hBoxNew False spacing
          boxPackStartDefaults h m
          boxPackStart h e PackNatural 0
          return h
    fc1 <- packMantissaExponent fc1m fc1e
    fc2 <- packMantissaExponent fc2m fc2e
    fc3 <- packMantissaExponent fc3m fc3e
    fc <- frameNewWithContents vBoxNew "Coordinates" True [fc1, fc2, fc3]
    v <- vBoxNew False spacing
    mapM_ (\w -> boxPackStart v w PackNatural 0) [fc, fv, {- fa, -} fh]
    h <- hBoxNew False spacing
    boxPackStart h fb PackNatural 0
    boxPackStartDefaults h v

    ww <- windowNew
    set ww [ windowIcon := Just icon, windowTitle := "gruff control" ]

    containerAdd ww h
    mg <- aDoLoad stateFile
    gR <- newIORef $ case mg of
      Nothing -> initialGruff
      Just g -> g

    let g0 = GruffGUI
               { bHome                 = bHome'
               , bLoad                 = bLoad'
               , bSave                 = bSave'
               , bStop                 = bStop'
               , bClear                = bClear'
               , eRealM                = eRealM'
               , eRealE                = eRealE'
               , eImagM                = eImagM'
```

```
            , eImagE                = eImagE'
            , eSizeM                = eSizeM'
            , eSizeE                = eSizeE'
            , eRota                 = eRota'
            , eWidth                = eWidth'
            , eHeight               = eHeight'
            , eSamples              = eSamples'
            , cInterior             = cInterior'
            , cBorder               = cBorder'
            , cExterior             = cExterior'
            , wMain                 = ww
            , wImage                = iw
            }
      but b a = do
        _ <- b g0 `onClicked` wrapA g0 gR a
        return ()
      butI b a = do
        _ <- b g0 `onClicked` (wrapA g0 gR a >> upI)
        return ()

      butO b a = do
        _ <- b g0 `onClicked` a
        return ()

      entI e a = do
        _ <- e g0 `onEntryActivate` (entryGetText (e g0) >>= wrapE g0 gR a >> ↙
            ↳ upI)
        return ()

      colI cbi cbb cbe a = forM_ [cbi, cbb, cbe] $ \c -> do
        _ <- c g0 `onColorSet` (do
          ci <- colorButtonGetColor (cbi g0)
          cb <- colorButtonGetColor (cbb g0)
          ce <- colorButtonGetColor (cbe g0)
          _ <- a ci cb ce gR
          upI)
        return ()

      entME m e a = do
        let a' = do
                 ms <- entryGetText (m g0)
                 es <- entryGetText (e g0)
                 let s = ms ++ if null es then "" else "e" ++ es
                 wrapE g0 gR a s
                 upI
        _ <- m g0 `onEntryActivate` a'
        _ <- e g0 `onEntryActivate` a'
        return ()

      upI :: IO ()
      upI = do
        g <- readIORef gR
        browserRender browser g (return ()) (return ())
        postRedisplay gl'

      aUpdate :: Maybe View.Image -> IO ()
      aUpdate Nothing = return ()
```

```
230         aUpdate (Just i) = do
              writeIORef gR i
              uReal g0 i
              uImag g0 i
              uSize g0 i
235           uRota g0 i
              upI

            aReshape :: Int -> Int -> IO ()
            aReshape w' h' = do
240           atomicModifyIORef gR $ \g ->
                ( g { imageWindow = (imageWindow g){ width = w', height = h' }
                    , imageViewport = (imageViewport g){ aspect = fromIntegral w' / ↙
                        ↳ fromIntegral h' }
                    }
                , () )
245           g <- readIORef gR
              uEverything g0 g
              upI

        butI bHome aHome
250     butI bLoad aLoad
        but  bSave aSave
        butO bStop (browserAbort browser)
        butI bClear aClear

255     entME eRealM eRealE aReal
        entME eImagM eImagE aImag
        entME eSizeM eSizeE aSize
        entI eRota aRota
        entI eWidth aWidth
260     entI eHeight aHeight
        entI eSamples aSamples
        colI cInterior cBorder cExterior aColours
        let aExit' = (exit (Log.log lg) stateFile =<< readIORef gR)
        browserSetExitCallback browser aExit'
265
        progress <- progressNew icon
        let mcbs = mouseCallbacks (readIORef labelColourR) progress aUpdate
        browserSetMouseCallback browser mcbs

270     browserSetReshapeCallback browser aReshape
        _ <- ww `onDestroy` aExit'
        g <- readIORef gR
        uEverything g0 g
        upI
275     widgetShowAll iw
        widgetShowAll ww
        _ <- forkIO $ script browser
        mainGUI

280 type Gruff = View.Image

    initialGruff :: Gruff
    initialGruff = defImage

285 -- button actions
```

```
type A = GruffGUI -> Gruff -> IO Gruff

wrapA :: GruffGUI -> IORef Gruff -> A -> IO ()
wrapA g0 gR a = do
  g <- readIORef gR
  g' <- a g0 g
  writeIORef gR $! g'

aHome :: A
aHome g0 g = do
  let g' = g{ imageLocation = defLocation, imageViewport = (imageViewport g){ ↙
      ↳ orient = 0 } }
  uEverything g0 g'
  return g'

aClear :: A
aClear g0 g = do
  let g' = g{ imageLabels = [], imageLines = [] }
  uEverything g0 g'
  return g'

aDoLoad :: FilePath -> IO (Maybe Gruff)
aDoLoad ff = (do
    gr <- safeRead `fmap` readFile ff
    case gr of
      g@(Just _) -> return g
      _ -> putStrLn "file format not supported, sorry" >> return Nothing
  ) `catchIO` const (return Nothing)

aLoad :: A
aLoad g0 g = do
  fc <- fileChooserDialogNew (Just "gruff load") (Just $ wMain g0) ↙
      ↳ FileChooserActionOpen [("Load", ResponseAccept), ("Cancel", ↙
      ↳ ResponseCancel)]
  widgetShow fc
  r <- dialogRun fc
  g' <- case r of
    ResponseAccept -> do
      mf <- fileChooserGetFilename fc
      case mf of
        Nothing -> return g
        Just f -> do
          mg <- aDoLoad f
          case mg of
            Nothing -> return g
            Just g' -> uEverything g0 g' >> return g'
    _ -> return g
  widgetDestroy fc
  return g'

aDoSave :: FilePath -> Gruff -> IO ()
aDoSave f g = writeFile f (show g) `catchIO` const (return ())

aSave :: A
aSave g0 g = do
  fc <- fileChooserDialogNew (Just "gruff save") (Just $ wMain g0) ↙
```

```
      ↪ FileChooserActionSave [("Save", ResponseAccept), ("Cancel", ↙
      ↪ ResponseCancel)]
340   widgetShow fc
      r <- dialogRun fc
      case r of
        ResponseAccept -> do
          mf <- fileChooserGetFilename fc
345       case mf of
            Nothing -> return ()
            Just f -> aDoSave f g
        _ -> return ()
      widgetDestroy fc
350   return g

--- entry update

type U = GruffGUI -> Gruff -> IO ()
355
uEverything, uReal, uImag, uSize, uRota, uColours, uWidth, uHeight, uSamples :: ↙
    ↪ U
uEverything g0 g = forM_ [uReal, uImag, uSize, uRota, uColours, uWidth, uHeight,↙
    ↪  uSamples] $ \u -> u g0 g
uReal g0 g = uMantissaExponent (eRealM g0) (eRealE g0) (show . (fromRational :: ↙
    ↪ Rational -> QuadDouble{- FIXME -}) . realPart . center . imageLocation $ g↙
    ↪ )
uImag g0 g = uMantissaExponent (eImagM g0) (eImagE g0) (show . (fromRational :: ↙
    ↪ Rational -> QuadDouble{- FIXME -}) . imagPart . center . imageLocation $ g↙
    ↪ )
360 uSize g0 g = uMantissaExponent (eSizeM g0) (eSizeE g0) (show . (fromRational :: ↙
    ↪ Rational -> Double    {- FIXME -}) . radius . imageLocation $ g)
uRota g0 g = entrySetText (eRota g0) (show . orient . imageViewport $ g)
uColours g0 g = do
  colorButtonSetColor (cInterior g0) (fromColour . colourInterior . imageColours↙
      ↪ $ g)
  colorButtonSetColor (cBorder   g0) (fromColour . colourBoundary . imageColours↙
      ↪ $ g)
365   colorButtonSetColor (cExterior g0) (fromColour . colourExterior . imageColours↙
      ↪ $ g)
uWidth   g0 g = entrySetText (eWidth   g0) (show . width  . imageWindow $ g)
uHeight  g0 g = entrySetText (eHeight  g0) (show . height . imageWindow $ g)
uSamples g0 g = entrySetText (eSamples g0) (show . supersamples . imageWindow $ ↙
    ↪ g)

370 uMantissaExponent :: (EntryClass a, EntryClass b) => a -> b -> String -> IO ()
uMantissaExponent m e s = do
  let (ms, me) = break (== 'e') s
  entrySetText m ms
  entrySetText e (drop 1 me)
375
--- entry actions

type E = Gruff -> String -> Gruff

380 wrapE :: GruffGUI -> IORef Gruff -> E -> String -> IO ()
wrapE _g0 gR e s = do
  g <- readIORef gR
  let g' = e g s
```

```
    writeIORef gR $! g'

aReal, aImag, aSize, aRota, aWidth, aHeight, aSamples :: E
aReal    g s = let _ :+ i = center (imageLocation g) in case safeRead s :: Maybe↙
    ↳ QuadDouble of
  Nothing -> g
  Just r -> g{ imageLocation = (imageLocation g){ center = toRational r :+ i } }
aImag    g s = let r :+ _ = center (imageLocation g) in case safeRead s :: Maybe↙
    ↳ QuadDouble of
  Nothing -> g
  Just i -> g{ imageLocation = (imageLocation g){ center = r :+ toRational i } }
aSize    g s = case safeRead s :: Maybe Double of
  Nothing -> g
  Just r -> g{ imageLocation = (imageLocation g){ radius = toRational r } }
aRota    g s =  case safeRead s :: Maybe Double of
  Nothing -> g
  Just a -> g{ imageViewport = (imageViewport g){ orient = a } }
aWidth   g s = case safeRead s of
  Nothing -> g
  Just r -> (\g' -> g'{ imageViewport = (imageViewport g'){ aspect = (↙
      ↳ fromIntegral . width . imageWindow) g' / (fromIntegral . height . ↙
      ↳ imageWindow) g' } }) (g{ imageWindow  = (imageWindow g){ width  = r } })
aHeight  g s = case safeRead s of
  Nothing -> g
  Just r -> (\g' -> g'{ imageViewport = (imageViewport g'){ aspect = (↙
      ↳ fromIntegral . width . imageWindow) g' / (fromIntegral . height . ↙
      ↳ imageWindow) g' } }) (g{ imageWindow  = (imageWindow g){ height = r } })
aSamples g s = case safeRead s of
  Nothing -> g
  Just r -> g{ imageWindow = (imageWindow g){ supersamples = r } }

aColours :: Color -> Color -> Color -> IORef Gruff -> IO ()
aColours i b e gR = atomicModifyIORef gR $ \g -> (g{ imageColours = Colours{ ↙
    ↳ colourInterior = fromColor i, colourBoundary = fromColor b, colourExterior↙
    ↳  = fromColor e} }, ())

minSize :: Size
minSize = Size 160 100

red, black, white, blue :: Color
red = Color 65535 0 0
black = Color 0 0 0
white = Color 65535 65535 65535
blue = Color 0 0 65535

fromColor :: Color -> Colour
fromColor (Color r g b) = Colour (fromIntegral r / 65535) (fromIntegral g / ↙
    ↳ 65535) (fromIntegral b / 65535)

fromColour :: Colour -> Color
fromColour (Colour r g b) = Color (round $ r * 65535) (round $ g * 65535) (round↙
    ↳  $ b * 65535)

script :: Browser -> IO ()
script b = do
  c <- getContents
  case mapMaybe readMay (lines c) of
```

```
    [] -> return ()
    images@(_:_) -> browserRenders b images

readMay :: Read a => String -> Maybe a
readMay s = case reads s of
  [(a, "")] -> Just a
  _ -> Nothing
```

35 src/Interact.hs

```
module Interact
  ( mouseCallbacks
  , MouseCallbacks
  , MouseCallback
  , Click
  , But(..)
  , Mod(..)
  ) where

import Data.Map (Map, fromList)

import Fractal.GRUFF

import Progress

type MouseCallbacks = Map Click MouseCallback

type Click = (But, Mod)

data But = L | M | R      deriving (Eq, Ord, Show)
data Mod = U | S | C | SC deriving (Eq, Ord, Show)

type MouseCallback = Complex Rational -> Image -> IO ()

type MouseHandler = Progress -> Complex Rational -> Image -> Cont Image

mouseCallbacks :: IO Colour -> Progress -> (Maybe Image -> IO ()) -> ↙
    ↳ MouseCallbacks
mouseCallbacks gc p next = fromList
  [ ( (L, U), w $ mouseZoom 0.9    )
  , ( (R, U), w $ mouseZoom 1.1    )
  , ( (M, U), w $ mouseCenter      )
  , ( (L, S), w $ mouseZoom 0.1    )
  , ( (R, S), w $ mouseZoom 10     )
  , ( (M, S), w $ mouseLocateGo    )
  , ( (L, C), w $ mousePeriod  gc  )
  , ( (R, C), w $ mouseAddress gc  )
  ]
  where w cb c i = cb p c i next

mouseCenter :: MouseHandler
mouseCenter _ c i next = do
  next . Just $ i{ imageLocation = (imageLocation i){ center = c } }

mouseZoom :: Double -> MouseHandler
mouseZoom f _ c i next = do
  let o = center (imageLocation i)
```

```
      r = radius (imageLocation i)
      f' = toRational f :+ 0
      r' = toRational f * r
      o' = f' * o + (1 - f') * c
  next . Just $ i{ imageLocation = (imageLocation i){ center = o', radius = r' }↙
      ↳ }

mouseLocateGo :: MouseHandler
mouseLocateGo _p _c _i next = next Nothing {- do
  let r0 = fromRational (radius (imageLocation i) / 32)
  progressLocate p c r0 $ \mmu -> case mmu of
    Nothing -> next Nothing
    Just mu -> do
      let n = muNucleus mu
          r = muSize    mu * 16
          a = muOrient  mu - (toRational (pi/2 :: Double))
      next . Just $ i{ imageLocation = (imageLocation i){ center = n, radius = r↙
          ↳ }, imageViewport = (imageViewport i){ orient = fromRational a } }
-}

mousePeriod :: IO Colour -> MouseHandler
mousePeriod _gv _p _c _i next = next Nothing {- do
  let r = fromRational (radius (imageLocation i) / 32)
  progressLocate p c r $ \mmu -> case mmu of
    Nothing -> next Nothing
    Just mu -> do
      let n = muNucleus mu
          t = show (muPeriod mu)
      v <- gv
      next . Just $ labelAppend n v t i
-}

mouseAddress :: IO Colour -> MouseHandler
mouseAddress _gv _p _c _i next = next Nothing {- do
  let r = fromRational (radius (imageLocation i) / 32)
  progressLocate p c r $ \mmu -> case mmu of
    Nothing -> next Nothing
    Just mu -> do
      progressAddress p mu $ \mad -> case mad of
        Nothing -> next Nothing
        Just ad -> do
          let n = muNucleus mu
              t = prettyAngledInternalAddress ad
          v <- gv
          next . Just $ labelAppend n v t i
-}
```

36 src/Logger.hs

```
module Logger where

import Prelude hiding (log)
import Control.Concurrent (forkIO, newChan, readChan, writeChan)
import Control.Monad (when)
import Data.Time (formatTime, getCurrentTime, UTCTime)
import System.Locale (defaultTimeLocale)
import System.FilePath ((</>), (<.>))
```

```
import System.IO (openFile, IOMode(WriteMode), hPutStrLn, hClose, hFlush)

data LogLevel = Error | Warn | Info | Debug
  deriving (Read, Show, Eq, Ord, Enum, Bounded)

defLogLevel :: LogLevel
defLogLevel = Debug

data Logger = Logger
  { log :: LogLevel -> String -> IO ()
  , close :: IO ()
  }

logger :: FilePath -> IO Logger
logger logDir = do
  date <- getDate
  let logFile = logDir </> filter (':'/=) date <.> "log"
  h <- openFile logFile WriteMode
  c <- newChan
  let log' lvl msg = when (lvl <= defLogLevel) $ do
        d <- getDate
        writeChan c (Just $ d ++ "\t" ++ show lvl ++ "\t" ++ show msg)
      close' = log' Info "closing log" >> writeChan c Nothing
      writer = do
        m <- readChan c
        case m of
          Nothing -> hClose h >> return ()
          Just s -> hPutStrLn h s >> hFlush h >> writer
  _ <- forkIO writer
  log' Info "opening log"
  return Logger{ log = log', close = close' }
  where
    logFormat :: UTCTime -> String
    logFormat = formatTime defaultTimeLocale "%FT%T"
    getDate :: IO String
    getDate = logFormat `fmap` getCurrentTime
```

37 src/Progress.hs

```
module Progress (Progress(..), progressNew, Cont) where

--import Control.Monad (forM_, when)
import Graphics.UI.Gtk (Pixbuf)
--import Numeric.QD (QuadDouble)
import Fractal.GRUFF

--import StatusDialog

-- | A continuation gets passed the result.
type Cont a = (Maybe a -> IO ()) -> IO ()

-- | Cancellable actions with status updates.
data Progress = Progress
  { progressLocate  :: Complex Rational -> Double -> Cont (Atom N53)
  , progressAtom    :: AngledInternalAddress -> Cont (Atom N53)
  , progressAddress :: Atom N53 -> Cont AngledInternalAddress
  }
```

```
20  -- | Create a new Progress with a status dialog.
    progressNew :: Pixbuf -> IO Progress
    progressNew _icon = do
      -- FIXME this is a horrible hack to avoid race conditions when re-using the ↙
          ↳ same status dialog
    --  sd1 <- statusDialogNew icon
25  --  sd2 <- statusDialogNew icon
    --  sd3 <- statusDialogNew icon
      return Progress
        { progressLocate  = undefined -- progressLocate'  sd1
        , progressAtom    = undefined -- progressAtom'    sd2
30      , progressAddress = undefined -- progressAddress' sd3
        }

    {-
    -- | Much like Fractal.RUFF.Mandelbrot.Atom.locate_
35  progressLocate' :: StatusDialog -> Complex Rational -> Double -> Cont (MuAtom ↙
        ↳ Rational)
    progressLocate' sd (re :+ im) r next = do
      statusDialog sd "gruff status" $ \progress' -> do
        let c = fromRational re :+ fromRational im :: Complex QuadDouble
        forM_ (locate eps2qd c (realToFrac r)) $ \mp -> case mp of
40        LocateScanTodo       -> progress' "Scanning for period..."
          LocateScan           -> progress' "Scanning for period..."
          LocateScanDone p     -> progress'$"Scanning for period... " ++ show p
          LocateNucleusTodo    -> progress' "Computing nucleus..."
          LocateNucleus i      -> when (i `mod` 20 == 0) . progress'$"Computing ↙
              ↳ nucleus... " ++ show i
45        LocateNucleusDone _  -> progress' "Computing nucleus... done"
          LocateBondTodo       -> progress' "Computing bond..."
          LocateBond i         -> when (i `mod` 20 == 0) . progress'$"Computing bond↙
              ↳ ... " ++ show i
          LocateBondDone _     -> progress' "Computing bond... done"
          LocateSuccess mu     -> do
50          progress' "Success!"
            let (re' :+ im') = muNucleus mu
            next $ Just mu{ muNucleus = toRational re' :+ toRational im', muSize = ↙
                ↳ toRational (muSize mu), muOrient = toRational (muOrient mu) }
          LocateFailed         -> do
            progress' "Failed!"
55          next $ Nothing

    -- | Much like Fractal.RUFF.Mandelbrot.Atom.findAddress_
    progressAddress' :: StatusDialog -> MuAtom Rational -> Cont ↙
        ↳ AngledInternalAddress
    progressAddress' sd mu next = do
60    statusDialog sd "gruff status" $ \progress' -> do
        let re :+ im = muNucleus mu
            c = fromRational re :+ fromRational im :: Complex QuadDouble
        forM_ (findAddress mu{ muNucleus = c, muSize = fromRational (muSize mu), ↙
            ↳ muOrient = fromRational (muOrient mu) }) $ \mp' -> case mp' of
          AddressCuspTodo      -> progress' "Computing cusp..."
65        AddressCuspDone _    -> progress' "Computing cusp... done"
          AddressDwellTodo     -> progress' "Computing dwell..."
          AddressDwell i       -> when (i `mod` 100 == 0) . progress'$"Computing ↙
              ↳ dwell... " ++ show i
```

```
        AddressDwellDone _     -> progress' "Computing dwell... done"
        AddressRayOutTodo      -> progress' "Tracing rays..."
        AddressRayOut i        -> progress'$"Tracing rays... " ++ show (round $ i *↙
            ↳ 100 :: Int) ++ "%"
        AddressRayOutDone _    -> progress' "Tracing rays... done"
        AddressExternalTodo    -> progress' "Computing angle..."
        AddressExternalDone _  -> progress' "Computing angle... done"
        AddressAddressTodo     -> progress' "Finding address..."
        AddressSuccess a       -> do
          progress' "Success!"
          next $ Just a
        AddressFailed          -> do
          progress' "Failed!"
          next $ Nothing

-- | Much like Fractal.RUFF.Mandelbrot.Atom.findAtom_
progressAtom' :: StatusDialog -> AngledInternalAddress -> Cont (MuAtom Rational)
progressAtom' sd addr next = do
  statusDialog sd "gruff status" $ \progress' -> do
    forM_ (findAtom eps2qd addr) $ \mp -> case mp of
      AtomSplitTodo        -> progress' "Splitting address..."
      AtomSplitDone _ _    -> progress' "Splitting address... done"
      AtomAnglesTodo       -> progress' "Computing angles..."
      AtomAnglesDone _ _   -> progress' "Computing angles... done"
      AtomRayTodo          -> progress' "Tracing rays..."
      AtomRay n            -> when (n `mod` 20 == 0) . progress'$"Tracing rays... ↙
          ↳ " ++ show n
      AtomRayDone _        -> progress' "Tracing rays... done"
      AtomNucleusTodo      -> progress' "Computing nucleus..."
      AtomNucleus n        -> when (n `mod` 20 == 0) . progress'$"Computing ↙
          ↳ nucleus... " ++ show n
      AtomNucleusDone _    -> progress' "Computing nucleus... done"
      AtomBondTodo         -> progress' "Computing bond..."
      AtomBond n           -> when (n `mod` 20 == 0) . progress'$"Computing bond↙
          ↳ ... " ++ show n
      AtomBondDone _       -> progress' "Computing bond... done"
      AtomSuccess mu       -> do
        progress' "Success!"
        let (re' :+ im') = muNucleus mu :: Complex QuadDouble
        next $ Just mu{ muNucleus = toRational re' :+ toRational im', muSize = ↙
            ↳ toRational (muSize mu), muOrient = toRational (muOrient mu) }
      AtomFailed           -> do
        progress' "Failed!"
        next $ Nothing

eps2qd :: QuadDouble
eps2qd = 1e-120
-}
```

38 src/QuadTree.hs

```
module QuadTree
  ( Child(..), north, south, west, east
  , Quad(..), root, child, children, parent, parents
  , filename, unsafeName, fromFilename
  , Square(..), square, Point(..), contains, Region(..), expand, outside
  , quads
```

```
  ) where

import Control.Monad (replicateM)
import Data.Bits (bit, shiftL, shiftR, testBit, (.|.))
import Data.List (unfoldr, sort)
import Data.Ratio ((%))
import System.FilePath (splitDirectories)
import qualified Data.Map as M

data Child = NorthWest | NorthEast | SouthWest | SouthEast
  deriving (Read, Show, Eq, Ord, Enum, Bounded)

north, south, west, east :: Child -> Bool
east c = fromEnum c `testBit` 0
south c = fromEnum c `testBit` 1
north = not . south
west = not . east

data Quad = Quad{ quadLevel :: !Int, quadWest, quadNorth :: !Integer }
  deriving (Read, Show, Eq, Ord)

root :: Quad
root = Quad{ quadLevel = 0, quadWest = 0, quadNorth = 0 }

child :: Child -> Quad -> Quad
child c Quad{ quadLevel = l, quadWest = x, quadNorth = y } = Quad
  { quadLevel = l + 1
  , quadWest  = x `shiftL` 1 .|. (fromIntegral . fromEnum . east ) c
  , quadNorth = y `shiftL` 1 .|. (fromIntegral . fromEnum . south) c
  }

children :: [Child] -> Quad
children = foldr child root

parent :: Quad -> Maybe (Child, Quad)
parent Quad{ quadLevel = l, quadWest = x, quadNorth = y }
  | l > 0  = Just
      ( toEnum (fromEnum (y `testBit` 0) `shiftL` 1 .|. fromEnum (x `testBit` 0)↙
          ↳ )
      , Quad{ quadLevel = l - 1, quadWest = x `shiftR` 1, quadNorth = y `shiftR`↙
          ↳  1 }
      )
  | otherwise = Nothing

parents :: Quad -> [Child]
parents = unfoldr parent

filename :: Quad -> Maybe ([FilePath], FilePath)
filename q
  | not (0 <= quadNorth q && quadNorth q < bit (quadLevel q) && 0 <= quadWest q ↙
    ↳ && quadWest q < bit (quadLevel q)) = Nothing
  | null cs = Just ([], "_")
  | otherwise = Just (init cs, last cs)
  where
    cs = chunk 2 . map unsafeName . chunk 2 . reverse . parents $ q

unsafeName :: [Child] -> Char
```

```
unsafeName [c]   = ['a'..'d'] !! (fromEnum c)
unsafeName [c,d] = ['e'..'t'] !! (fromEnum c `shiftL` 2 .|. fromEnum d)
unsafeName _ = error "QuadTree.unsafeName"

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (ys, zs) = splitAt n xs in ys : chunk n zs

fromFilename :: FilePath -> Maybe Quad
fromFilename "_" = Just root
fromFilename p = fmap (children . reverse . concat) . mapM f . concat . ↙
    ↳ splitDirectories $ p
  where
    f c = c `M.lookup` chars
    chars = M.fromList [ (unsafeName l, l) | n <- [1, 2], l <- replicateM n [↙
        ↳ minBound .. maxBound] ]

data Square = Square{ squareSize, squareWest, squareNorth :: !Rational }
  deriving (Read, Show, Eq, Ord)

square :: Square -> Quad -> Square
square Square{ squareSize = s0, squareWest = x0, squareNorth = y0 } Quad{ ↙
    ↳ quadLevel = l, quadWest = x, quadNorth = y } =
  Square{ squareSize = s0 / fromInteger r, squareWest = x0 + s0 * (x % r), ↙
      ↳ squareNorth = y0 + s0 * (y % r) } where r = bit l

data Region = Region{ regionNorth, regionSouth, regionWest, regionEast :: !↙
    ↳ Rational }
  deriving (Read, Show, Eq, Ord)

expand :: Rational -> Region -> Region
expand f r =
  let (x,  y ) = ((regionEast r + regionWest r) / 2, (regionNorth r + ↙
      ↳ regionSouth r) / 2)
      (rx, ry) = ((regionEast r - regionWest r) / 2, (regionNorth r - ↙
          ↳ regionSouth r) / 2)
  in  Region{ regionNorth = y + f * ry, regionSouth = y - f * ry, regionEast = x↙
      ↳  + f * rx, regionWest = x - f * rx }

outside :: Region -> Square -> Bool
outside r s
  =  regionSouth r < squareNorth s
  || regionEast  r < squareWest  s
  || regionNorth r > squareNorth s + squareSize s
  || regionWest  r > squareWest  s + squareSize s

data Point = Point{ pointWest, pointNorth :: !Rational }
  deriving (Read, Show, Eq, Ord)

contains :: Point -> Square -> Bool
contains p s
  =  squareNorth s <= pointNorth p && pointNorth p <= squareNorth s + squareSize↙
      ↳  s
  && squareWest  s <= pointWest  p && pointWest  p <= squareWest  s + squareSize↙
      ↳  s

quads :: Square -> Region -> Int -> [Quad]
```

```
quads rootSquare region level =
  [ Quad{ quadLevel = level, quadWest = w, quadNorth = n }
  | n <- [ floor nlo' .. ceiling nhi' - 1]
  , w <- [ floor wlo' .. ceiling whi' - 1]
  ]
  where
    [nlo', nhi'] = sort [nlo, nhi]
    [wlo', whi'] = sort [wlo, whi]
    nlo = (regionSouth region - squareNorth rootSquare) / squareSize rootSquare ↙
        ↳ * l
    nhi = (regionNorth region - squareNorth rootSquare) / squareSize rootSquare ↙
        ↳ * l
    wlo = (regionWest  region - squareWest  rootSquare) / squareSize rootSquare ↙
        ↳ * l
    whi = (regionEast  region - squareWest  rootSquare) / squareSize rootSquare ↙
        ↳ * l
    l = bit level % 1
```

39 src/Shader.hs

```
module Shader (shader) where

import Graphics.Rendering.OpenGL

shader :: Maybe FilePath -> Maybe FilePath -> IO Program
shader mV mF = do
  [p] <- genObjectNames 1
  vs <- case mV of
    Nothing -> return []
    Just v -> do
      [vert] <- genObjectNames 1
      source <- readFile v
      shaderSource vert $= [ source ]
      compileShader vert
--      print =<< get (shaderInfoLog vert)
      return [vert]
  fs <- case mF of
    Nothing -> return []
    Just f -> do
      [frag] <- genObjectNames 1
      source <- readFile f
      shaderSource frag $= [ source ]
      compileShader frag
--      print =<< get (shaderInfoLog frag)
      return [frag]
  attachedShaders p $= (vs, fs)
  linkProgram p
--  print =<< get (programInfoLog p)
  return p
```

40 src/Snapshot.hs

```
module Snapshot (hSnapshot, writeSnapshot, snapshotWith) where

import Control.Monad(forM_)
import System.IO(Handle())
```

```
import Graphics.Rendering.OpenGL(
  readPixels,
  Position,
  Size(Size),
  PixelData(PixelData),
  PixelFormat(RGB),
  DataType(UnsignedByte))
import Foreign.Marshal.Alloc(allocaBytes)
import Foreign.Ptr(plusPtr)
import qualified Data.ByteString.Internal as BSI
import qualified Data.ByteString as BS

-- save a screenshot as binary PPM
snapshotWith :: (BS.ByteString -> IO b) -> Position -> Size -> IO b
snapshotWith f p0 vp@(Size vw vh) = do
  let fi q = fromIntegral q
      p6 = "P6\n" ++ show vw ++ " " ++ show vh ++ " 255\n"
  allocaBytes (fi (vw*vh*3)) $ \ptr -> do
    readPixels p0 vp $ PixelData RGB UnsignedByte ptr
    px <- BSI.create (fi $ vw * vh * 3) $ \d -> forM_ [0..vh-1] $ \y ->
      BSI.memcpy
        (d`plusPtr`fi(y*vw*3))
        (ptr`plusPtr`fi ((vh-1-y)*vw*3))
        (fi(vw*3))
    f $ BS.pack (map (toEnum . fromEnum) p6) `BS.append` px

hSnapshot :: Handle -> Position -> Size -> IO ()
hSnapshot h = snapshotWith (BS.hPutStr h)

writeSnapshot :: FilePath -> Position -> Size -> IO ()
writeSnapshot f = snapshotWith (BS.writeFile f)
```

41 src/StatusDialog.hs

```
module StatusDialog where

import Control.Concurrent (forkIO, killThread)
import Graphics.UI.Gtk

type Status = (String -> IO ()) -> IO ()
data StatusDialog = StatusDialog{ statusDialog :: String -> Status -> IO () }

statusDialogNew :: Pixbuf -> IO StatusDialog
statusDialogNew icon = do
  window <- windowNew
  set window [ windowIcon := Just icon ]
  windowSetModal window True
  windowSetDefaultSize window 320 200
  b <- vBoxNew False 2
  label <- labelNew (Just "")
  cancel <- buttonNewWithLabel "Cancel"
  boxPackStartDefaults b label
  boxPackStartDefaults b cancel
  set window [ containerChild := b ]
  return StatusDialog{ statusDialog = \title task -> postGUIAsync $ do
    windowSetTitle window title
    labelSetText label ""
```

```
    widgetSetSensitivity cancel True
    child <- forkIO $ do
      task $ postGUIAsync . labelSetText label
      postGUISync $ widgetHide window
    _ <- cancel `onClicked` do
      killThread child
      widgetHide window
    widgetShowAll window
  }
```

42 src/Tile.hs

```
{-# LANGUAGE ForeignFunctionInterface #-}
module Tile where

import Prelude hiding (log)

import Control.Exception (bracketOnError)
import Control.Monad (forM_, when)
import Data.Bits (shiftL, shiftR, (.&.))
import Foreign (Ptr, castPtr, mallocArray, free, Word8, allocaBytes, withArray, ↙
    ↳ peekArray, peekElemOff, pokeElemOff)
import Foreign.C (CInt(..), CUChar(..), CSize(..), withCStringLen)
import System.Directory (createDirectoryIfMissing)
import System.FilePath ((</>), (<.>))
import System.IO (withBinaryFile, IOMode(ReadMode,WriteMode), hPutBuf, hGetBuf)
--import Data.BEncode

import Fractal.GRUFF hiding (width, height, (</>))

import Compute (compute)
import QuadTree
import Utils (catchIO)

width, height, count, bytes :: Int
width  = 256
height = 256
count  = width * height
bytes  = count

rootSquare :: Square
rootSquare = Square{ squareSize = 8, squareWest = -4, squareNorth = -4 }

data Tile = Tile
  { tileQuad :: Quad
  , tileData :: Ptr CUChar
  }

mallocTile :: Quad -> IO Tile
mallocTile cs = do
  ds <- mallocArray count
  return $ Tile cs ds

freeTile :: Tile -> IO ()
freeTile (Tile _ ds) = do
  free ds
```

```
45  header :: String
    header = "RuFfTtLe001\n"

    writeTile :: FilePath -> Tile -> IO ()
    writeTile cacheDir (Tile q ds) = do
50    case filename q of
        Nothing -> return ()
        Just (dirs, file) -> do
          let dir = foldr1 (</>) (cacheDir : dirs)
          createDirectoryIfMissing True dir
55        withBinaryFile (dir </> file <.> "t") WriteMode $ \h -> do
            withCStringLen header $ \(p, l) -> hPutBuf h p l
            withArray (wordBE bytes) $ \p -> hPutBuf h p 4
            hPutBuf h ds bytes
    --        hPut h . bPack . metaData $ q
60    where
        wordBE :: Int -> [Word8]
        wordBE n = map (\b -> fromIntegral $ (n `shiftR` (8 * b)) .&. 0xFF) [3, 2, ↙
            ↳ 1, 0]

    {-
65  metaData :: Quad -> BEncode
    metaData q@Quad{ quadLevel = l, quadWest = w, quadNorth = n } =
      BDict $ fromList [ ("tile", BDict $ fromList
        [ ("about", BDict $ fromList
          [ ("version", BInt 1)
70        , ("generator", BString $ pack "gruff-0.1") ]
        , ("images", BDict . fromList . map image . zip [0..] $
          [ "continuous dwell", "normalized distance", "final angle"]) ]
        ]
      where
75      image plane alg = (alg, BDict $ fromList
          [ ("width", BInt (fromIntegral width))
          , ("height", BInt (fromIntegral height))
          , ("real", BInt w)
          , ("imag", BInt (negate n))
80        , ("scale", BInt (fromIntegral l + 2))
          , ("format", BString $ pack "float32le")
          , ("order", BString $ pack "lr,tb")
          , ("data offset", BInt (fromIntegral $ plane * count * sizeOf (0 :: CFloat↙
              ↳ )))
          ])
85  -}

    readTile :: FilePath -> Quad -> IO (Maybe Tile)
    readTile cacheDir q = flip catchIO (\_ -> return Nothing) $ do
      case filename q of
90      Nothing -> return Nothing
        Just (dirs, file) -> do
          let dir = foldr1 (</>) (cacheDir : dirs)
          bracketOnError (mallocTile q) freeTile $ \t@(Tile _ ds) -> do
            withBinaryFile (dir </> file <.> "t") ReadMode $ \h -> do
95            let headerBytes = 12
              allocaBytes headerBytes $ \p -> do
                headerBytes' <- hGetBuf h p headerBytes
                when (headerBytes /= headerBytes') $ fail "readTile header fail"
                header' <- peekArray headerBytes p
```

```
100             when (header' /= (map (fromIntegral . fromEnum) header :: [Word8])) ↙
                    ↳ $ fail "readTile header mismatch"
              dataBytes <- allocaBytes 4 $ \p -> do
                lenBytes' <- hGetBuf h p 4
                when (lenBytes' /= 4) $ fail "readTile header length fail"
                unwordBE `fmap` peekArray 4 p
105           when (dataBytes /= bytes) $ fail "readTile header length mismatch"
              bytes' <- hGetBuf h ds bytes
              when (bytes /= bytes') $ fail ("readTile " ++ show bytes ++ " /= " ++ ↙
                  ↳ show bytes')
              return $ Just t
    where
110   unwordBE :: [Word8] -> Int
      unwordBE = sum . zipWith (\b n -> fromIntegral n `shiftL` (8 * b)) [3, 2, 1,↙
          ↳ 0]

clearTile :: Tile -> IO ()
clearTile (Tile _ ds) = clear ds >> return ()
115 where
      clear p = c_memset (castPtr p) 0 (fromIntegral bytes)

computeTile :: (String -> IO ()) -> Tile -> IO Bool
computeTile log (Tile q@Quad{ quadLevel = l } ds) = do
120   its <- compute'
      log $ show ("getTile", q, "computed", its)
      return $ its /= 0
    where
      compute' = reifyPrecision (fromIntegral l + 8) $ \prec ->
125     let _ = fmap precisionOf c `asTypeOf` (prec :+ prec)
            c = cx :+ cy
            cx = fromRational (squareWest s)
            cy = fromRational (squareNorth s)
        in  compute ds c l m'
130   m' = 1000000
      s = square rootSquare q

data TileResult = Interrupted | Completed !Tile | BlockedOn !Quad

135 getTile :: (String -> IO ()) -> FilePath -> Quad -> IO TileResult
getTile log cacheDir q = do
  mTile <- readTile cacheDir q
  case mTile of
    Just t -> do
140   log $ show ("getTile", q, "read")
      return (Completed t)
    Nothing -> case parent q of
      Just (ch, q') -> do
        mptile <- readTile cacheDir q'
145     case mptile of
          Nothing -> do
            log $ show ("getTile", q, "blocked")
            return (BlockedOn q')
          Just t' -> do
150         log $ show ("getTile", q, "compute")
            t <- mallocTile q
            inheritTile ch t' t
            freeTile t'
```

```
            touchTileBorder t
            ok <- computeTile log t
            if ok
              then writeTile cacheDir t >> return (Completed t)
              else freeTile t >> return Interrupted
        Nothing -> do
          log $ show ("getTile", q, "compute")
          t <- mallocTile q
          clearTile t
          touchTileBorder t
          ok <- computeTile log t
          if ok
            then writeTile cacheDir t >> return (Completed t)
            else freeTile t >> return Interrupted

inheritTile :: Child -> Tile -> Tile -> IO ()
inheritTile c (Tile _ from) (Tile _ to) = do
  let coff = case c of
               NorthWest -> 0
               NorthEast -> 128
               SouthWest -> 256 * 128
               SouthEast -> 256 * 128 + 128
  forM_ [0..255] $ \j -> do
    forM_ [0..255] $ \i -> do
      let toff = 256 * j + i
          foff = 256 * (j `div` 2) + (i `div` 2) + coff
      pokeElemOff to toff =<< peekElemOff from foff

touchTileBorder :: Tile -> IO ()
touchTileBorder (Tile _ t) = do
  let o i j = 256 * j + i
  forM_ [0..255] $ \e -> do
    pokeElemOff t (o   0   e) 254
    pokeElemOff t (o   e   0) 254
    pokeElemOff t (o 255   e) 254
    pokeElemOff t (o   e 255) 254

foreign import ccall unsafe "string.h memset"
  c_memset :: Ptr Word8 -> CInt -> CSize -> IO (Ptr Word8)
```

43 src/TileShake.hs

```
module TileShake (computeTiles) where

import Development.Shake
import Development.Shake.FilePath

import GHC.Conc (numCapabilities)
import Control.Concurrent (forkIO, killThread, newEmptyMVar, putMVar, takeMVar)
import Control.Exception (finally)
import Control.Monad (forM_)
import Data.List (isPrefixOf)
import Data.Set (Set)
import qualified Data.Set as S
import System.Directory (createDirectoryIfMissing)

import Tile
```

```
import QuadTree

toFilename :: FilePath -> Quad -> FilePath
toFilename cacheDir q = case filename q of
  Just (dirs, file) -> foldr1 (</>) (cacheDir : dirs) </> file <.> "t"
  Nothing -> error $ "gruff.TileShake.toFilename: " ++ show (cacheDir, q)

toQuad :: FilePath -> FilePath -> Maybe Quad
toQuad cacheDir f
  | takeExtension f == ".t" && cacheDir' `isPrefixOf` f = fromFilename f'
  | otherwise = Nothing
  where
    cacheDir' = addTrailingPathSeparator cacheDir
    f' = drop (length cacheDir') . dropExtension $ f

computeTiles :: (String -> IO ()) -> (Tile -> IO ()) -> FilePath -> Set Quad -> ↙
    ↳ IO (IO ())
computeTiles output done cacheDir qs = do
  let shakeDir = cacheDir </> "_shake"
  createDirectoryIfMissing True shakeDir
  m <- newEmptyMVar
  t <- forkIO . flip finally (putMVar m ()) $ do
    shake shakeOptions
        { shakeFiles = shakeDir
        , shakeThreads = numCapabilities
        , shakeReport = Nothing -- Just (cacheDir </> "_shake-report.html")
        , shakeVerbosity = Quiet
        } $ do
      want (map (toFilename cacheDir) (S.toList qs))
      "//*.t" *> \out -> do
        let Just q = toQuad cacheDir out
        mme <- liftIO $ readTile cacheDir q
        case mme of
          Just t -> liftIO $ if q `S.member` qs then done t else freeTile t
          Nothing -> case parent q of
            Nothing -> traced ("compute " ++ show q) $ do
              t <- mallocTile q
              clearTile t
              touchTileBorder t
              ok <- computeTile output t
              if ok
                then writeTile cacheDir t >> if q `S.member` qs then done t else↙
                    ↳ freeTile t
                else freeTile t >> fail "intr"
            Just (ch, q') -> (need [toFilename cacheDir q'] >>) . traced ("↙
                ↳ compute " ++ show q) $ do
              mptile <- readTile cacheDir q'
              case mptile of
                Nothing -> fail "eeps"
                Just t' -> do
                  t <- mallocTile q
                  inheritTile ch t' t
                  freeTile t'
                  touchTileBorder t
                  ok <- computeTile output t
                  if ok
                    then writeTile cacheDir t >> if q `S.member` qs then done t ↙
```

```
                        ↳ else freeTile t
                    else freeTile t >> fail "intr"
      forM_ (S.toList qs) $ \q -> do
        mt <- readTile cacheDir q
        case mt of
          Nothing -> return ()
          Just t -> done t
    return (killThread t >> takeMVar m)
```

44 src/Utils.hs

```
module Utils where

import Prelude hiding (catch)
import Control.Exception (catch, IOException)
import Control.Monad (forM)
import System.Directory (getDirectoryContents, doesFileExist, doesDirectoryExist↙
    ↳ )
import System.FilePath ((</>))

safeRead :: Read a => String -> Maybe a
safeRead s = case filter (null . snd) (reads s) of
  [(a, "")] -> Just a
  _ -> Nothing

catchIO :: IO a -> (IOException -> IO a) -> IO a
catchIO = catch

getFilesRecursive :: FilePath -> IO [([FilePath], FilePath)]
getFilesRecursive d = (do
  fs0 <- getDirectoryContents d
  let fs = filter (`notElem` [".", ".."]) fs0
  ffs <- forM fs $ \f -> do
    let df = d </> f
    fe <- doesFileExist df
    fd <- doesDirectoryExist df
    case (fe, fd) of
      (True, False) -> return [([], f)]
      (False, True) -> map (\(ds, f') -> (f:ds, f')) `fmap` getFilesRecursive df
      _ -> return []
  return (concat ffs)) `catchIO` (\_ -> return [])

safeLast :: [a] -> Maybe a
safeLast [] = Nothing
safeLast [x] = Just x
safeLast (_:xs) = safeLast xs
```

45 src/View.hs

```
module View
  ( Location(..), zoom, translate, defLocation
  , Viewport(..), rotate, defViewport
  , Window(..), windowSize', defWindow
  , Colours(..), defColours, Colour(..)
  , Label(..), Line(..)
  , Image(..), defImage
```

```
    , BufferSize(..)
    , pixelLocation, delta, tileSize, locationPixel
    , visibleQuads, originQuad
    ) where

import Control.Monad (guard)
import Data.Bits (bit)
import Data.Ratio ((%))

import Fractal.GRUFF

import QuadTree (Quad(..), Child(..), child, Square(..), square)
import Tile (rootSquare)

pixelLocation :: Window -> Viewport -> Location -> Double -> Double -> Complex ↙
    ↳ Rational
pixelLocation w v l = let f = fromScreenCoords w v l in \x y -> f (x :+ y)

locationPixel :: Window -> Viewport -> Location -> Complex Rational -> (Double, ↙
    ↳ Double)
locationPixel w v l = let f = toScreenCoords w v l in \c -> let (x :+ y) = f c ↙
    ↳ in (x, y)

zoom :: Double -> Location -> Location
zoom f l = l{ radius = radius l * toRational f}

translate :: Complex Rational -> Location -> Location
translate u l = l{ center = center l + u }

rotate :: Double -> Viewport -> Viewport
rotate a v = v{ orient = orient v + a }

windowSize' :: Window -> Double
windowSize' w = (sqrt . fromIntegral . diagonal2) w

diagonal2 :: Window -> Int
diagonal2 w = width w * width w + height w * height w

data BufferSize = BufferSize
  { texels :: !Int -- power of two
  }
  deriving (Read, Show, Eq)

level :: Location -> Int
level = floor . negate . logBase 2 . (fromRational :: Rational -> Double {- ↙
    ↳ FIXME -}) . radius

radius' :: Location -> Double
radius' l = 0.5 ** fromIntegral (level l)

delta :: Location -> Double -- in [0,1)
delta l = logBase 2 $ radius' l / fromRational{- FIXME -} (radius l)

tileSize :: Int
tileSize = 256

tileLevel :: Location -> BufferSize -> Int
```

```
tileLevel l b = level l + (floor . logBase (2 :: Double) . fromIntegral) (texels b `div` tileSize)

tileOrigin :: Complex Rational
tileOrigin = negate $ 4 :+ 4

tileOriginRadius :: Complex Rational
tileOriginRadius = 8

bufferOrigin :: Location -> Quad -> Maybe (Complex Int)
bufferOrigin l Quad{ quadLevel = ql, quadWest = qw, quadNorth = qn } = do
  guard $ ql >= 0
  let qd = bit ql
      qc = (qw % qd) :+ (qn % qd)
      tx :+ ty = fromIntegral tileSize * fromIntegral qd * (qc - (center l - tileOrigin) / tileOriginRadius)
  return (floor tx :+ floor ty)

originQuad :: Location -> BufferSize -> Maybe Quad
originQuad l b =
  let cx :+ cy = center l
      ql = tileLevel l b
      qs = bit ql % 1
      qw = floor $ (cx + 4) / 8 * qs
      qn = floor $ (cy + 4) / 8 * qs
  in  if ql <= 0 then Nothing else Just Quad{ quadLevel = ql, quadWest = qw, quadNorth = qn }

bufferQuads :: Location -> BufferSize -> Maybe [(Complex Int, Quad)]
bufferQuads l b = do
  q0 <- originQuad l b
  i0 :+ j0 <- bufferOrigin l q0
  let m = texels b
      u = fromIntegral $ (m `div` 2) `div` tileSize
      v = fromIntegral $ (m `div` 2) `div` tileSize
  return
    [ (i :+ j, q0{ quadWest = w, quadNorth = n })
    | (i, w) <- takeWhile ((< m) . fst) $ [ i0, i0 + tileSize .. ] `zip` [ quadWest q0 - u .. ]
    , (j, n) <- takeWhile ((< m) . fst) $ [ j0, j0 + tileSize .. ] `zip` [ quadNorth q0 - v .. ]
    ]

childQuads :: (Complex Int, Quad) -> [(Complex Int, Quad)]
childQuads (i :+ j, q) =
  let i0 = 2 * i
      j0 = 2 * j
      i1 = i0 + tileSize
      j1 = j0 + tileSize
  in  [ (i0 :+ j0, NorthWest `child` q)
      , (i0 :+ j1, SouthWest `child` q)
      , (i1 :+ j0, NorthEast `child` q)
      , (i1 :+ j1, SouthEast `child` q)
      ]

visibleQuads :: Window -> Viewport -> Location -> BufferSize -> Maybe ([(Complex Int, Quad)], [(Complex Int, Quad)])
```

```
visibleQuads w v l b = do
  let x0 = 0
      y0 = 0
      x1 = fromIntegral (width  w)
      y1 = fromIntegral (height w)
      toScreen = toScreenCoords w v l
      visible (_, q) =
        let s = square rootSquare q
            d = squareSize s / 2
            r  = magnitude $ p - p0
            c0 =  squareWest s      :+  squareNorth s
            c  = (squareWest s + d) :+ (squareNorth s + d)
            p0         = toScreen c0
            p@(x :+ y) = toScreen c
        in  not $ x < x0 - r || y < y0 - r || x1 + r < x || y1 + r < y
  qs0 <- bufferQuads l b
  let qs0' = filter visible qs0
      qs1 = concatMap childQuads qs0'
      qs1' = filter visible qs1
  return (qs0', qs1')

defImage :: Image
defImage = Image
  { imageWindow = defWindow
  , imageViewport = defViewport
  , imageLocation = defLocation
  , imageColours = defColours
  , imageLabels = []
  , imageLines = []
  }

defColours :: Colours
defColours = Colours
  { colourInterior = Colour 1 0 0
  , colourBoundary = Colour 0 0 0
  , colourExterior = Colour 1 1 1
  }

defLocation :: Location
defLocation = Location{ center = 0, radius = 2 }

defWindow :: Window
defWindow = Window{ width = 512, height = 288, supersamples = 1 }

defViewport :: Viewport
defViewport = Viewport{ aspect = 16/9, orient = 0 }
```

46 TODO

```
annoyments
  catch exceptions in shake thread (eg: disk full)
  figure out where rare libpango segfault is coming from - threading?

enhancements
  navigation
    configurable zoom mode: center on click vs expand about click
    continuous zoom mode while button is pressed
```

```
    undo/history tree navigation
    log annotation and search
    log browser view with filtering
    log level settings (Set Loggable ?)
  persistence
    image save as png
    save/load viewing parameters in a common plain text format
  colouring
    multiple colouring modes - minimal / rainbow / hypertile
    gui for each colouring mode with appropriate persistence
  memory cache
    option for maximum memory cache size
    warn when memory cache is not enough for window settings
    memory cache model - LRU or similar?
  disk cache
    option for maximum disk cache size
    option to disable disk cache for casual exploratory browsing
    disk cache management: keep most expensive vs most used?
    tile usage statistics database
  tile renderer
    compute tile cost metrics (iteration stats, boundary size, bit bepth)
    which quadrants are interior/exterior etc
    re-use pixels from parent/child tiles in renderer
    don't compute known-interior tiles
  features
    scan whole image for atoms, subdivide for more
    draw internal rays in feature overlay
    feature database - try lookup before calculating, cache results
  algorithms
    child finding algorithm
    parent finding algorithm - need to research
    box counting fractal dimension estimates
  misc
    tiled rendering mode for huge images without seams
  scripting
    helper library for using GHCI via fifo to gruff exe

internal
  source code documentation
```