Joshua Learn - jrl2196
Parallel Functional Programming - Stephen Edwards
Final Project Report

Polygon Renderer

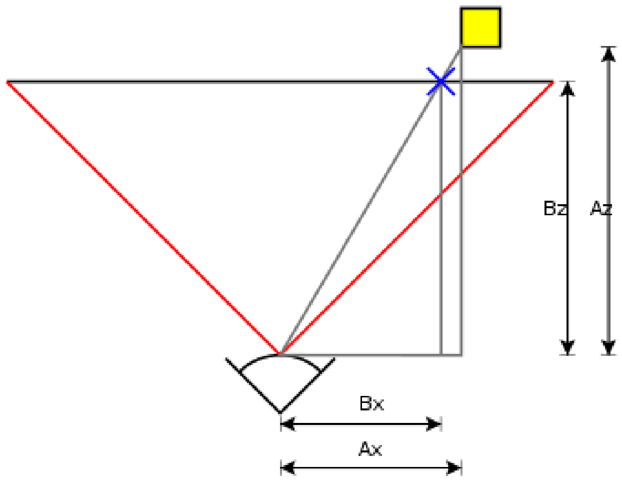
For this project I wrote a solution to the problem of polygon rendering and surface illumination. Many 3-dimensional figures are modelled digitally using a collection of enclosing polygons. One common file format for storing this information is cally PLY file format, which is the format of data I worked with for this project. PLY files store a collection of vertex coordinates, surfaces using those vertices, and any relevant color information about each vertex. The process of polygon rendering takes these inputs and projects them onto a 2-dimensional grid and then rasterizes an image based on that information. There are several steps that go into this: translation of coordinates to a new origin, projection of 3-dimensional coordinates onto 2-dimensional grid, hidden surface elimination, and lighting/illumination.

Translation of Coordinates:

It can be beneficial to move the origin of the coordinate system to be equal to the perspective of the viewer. This simplifies many vector calculations as we are often interested in how outgoing vectors from the viewer perspective coincide with the modeled objects. To translate a coordinate system one must merely subtract the value of the new origin with respect to the old origin from all existing coordinates.

Projection on 2-dimensional Surface:

Once the coordinates are translated, pixel locations of vertices are calculated by projected all vertices onto a 2-dimensional plane in front of the viewer perspective. This calculation involves the focal distance and vector from the viewer perspective to the vertex coordinate.



https://en.wikipedia.org/wiki/3D_projection#Diagram

Hidden Surface Elimination:

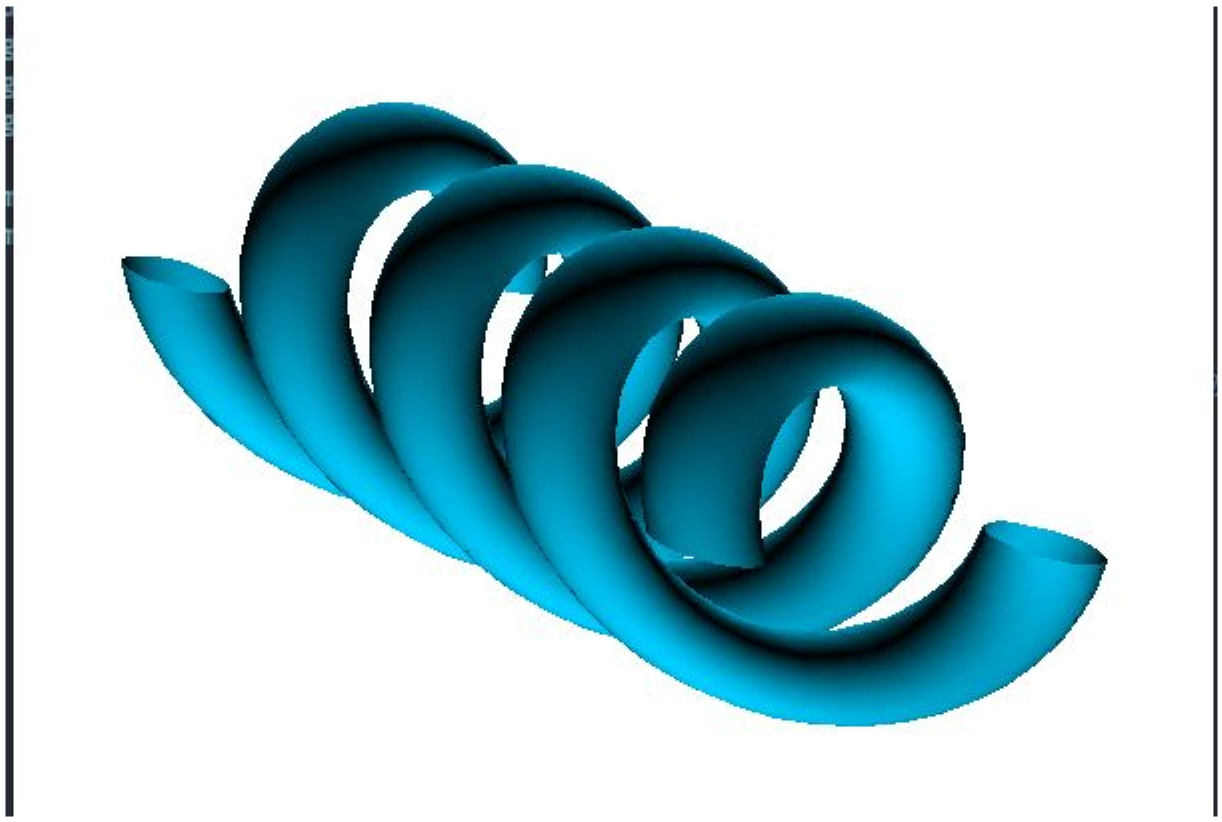
When viewing an opaque surface, it is necessary to remove certain parts of the polygon from being rendered because they are blocked by another part of the object. For this project I used the z-buffer method of hidden surface elimination. This keeps track of the depth that corresponds to all pixels after they are projected on the 2D surface. When the pixels are being drawn, only the lowest depth pixel is chosen to be displayed, as it corresponds to the least amount of distance from the viewing perspective.

Lighting/Illumination:

In order to show a realistic lighting effect, the renderer must use a shading algorithm along side an illumination model in order to choose the color intensity at any pixel. This project uses the Phong shading model in conjunction with the simpler Lambert illumination model for a directional light source vector. There are several parts that go into this calculation. First you must interpolation the normal vector at each vertex to be the average vector of all object faces that use that vertex. This method is used to get the normal vectors at all vertices. These vectors along with the vertex color information are then interpolated across the surfaces of the polygon. This results in an individual color and normal vector value for each pixel being drawn. The interpolation is done by simply incrementing the vector by some delta while drawing the scan lines across the face of a polygon. The Lambert illumination model gives us the light intensity as being proportional to the cosine between the normal vector and the vector to the light source. Using this cosine we can appropriately weight the brightness of the RGB color values at all pixels.

Drawing:

Once all the calculations are finished, you simply go through each polygon face and draw scan lines horizontally from one side of the polygon to the other. At each point you calculate the light intensity and draw the pixel appropriately. I emply Bresenham's line algorithm for drawing lines between two pixel coordinates. Below is an example of the final product of the drawing of a helix with a directional light source located below the object:



helix.ply sample file render

Parallelization:

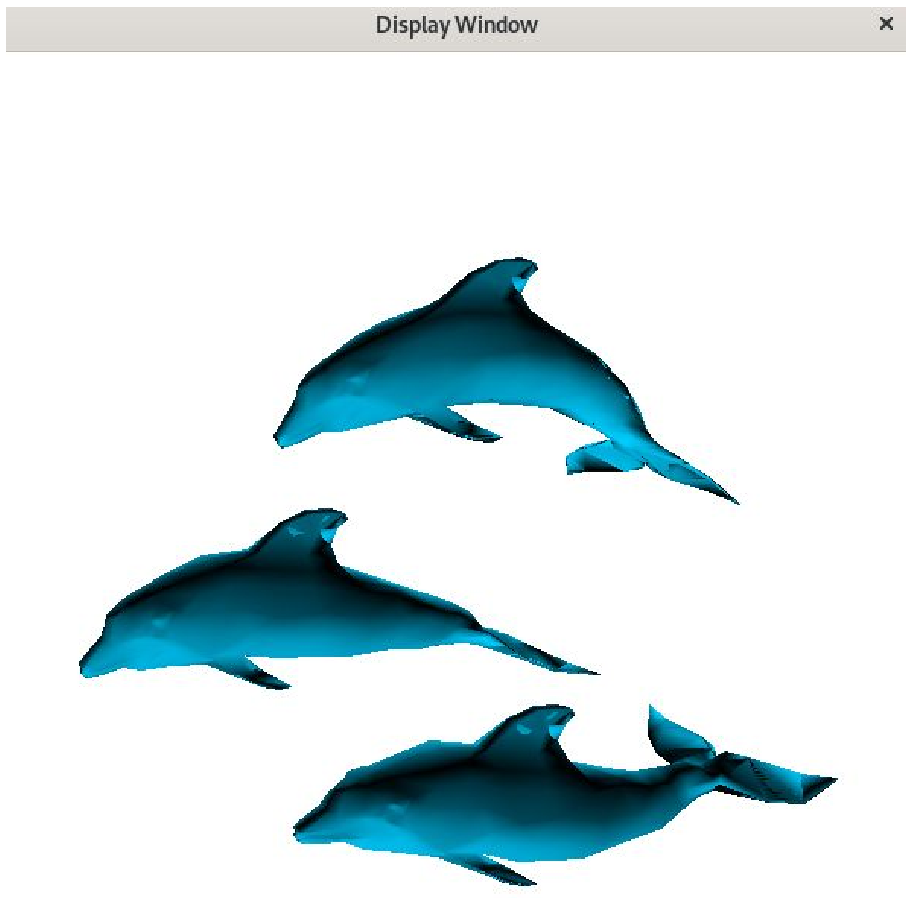
There are a few notable places that this algorithm should be easily parallelizable:

a. Translating coordinates
b. Computing illumination
c. interpolating value weights across edges

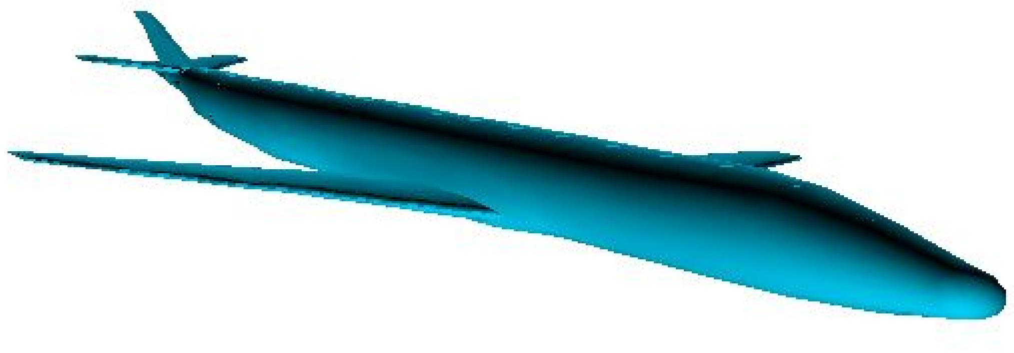
I found that I wasn't able to increase the performance notably when parallelizing the code. The performance seemed to suffer some from the overhead of parallelizing. I tried chunking the data operations to reduce the granularity but to no avail. I suspect this is due to the high amount of data dependency in this problem. This stems from the interpolation portion of the algorithm. Since values are interpolated across faces there is often a limited amount that can be done in parallel due to data dependency on previous calculations. I suspect there is a clever way to factor this in and pipeline the calculations for a performance gain, but I haven't been able to do it successfully.

Examples:

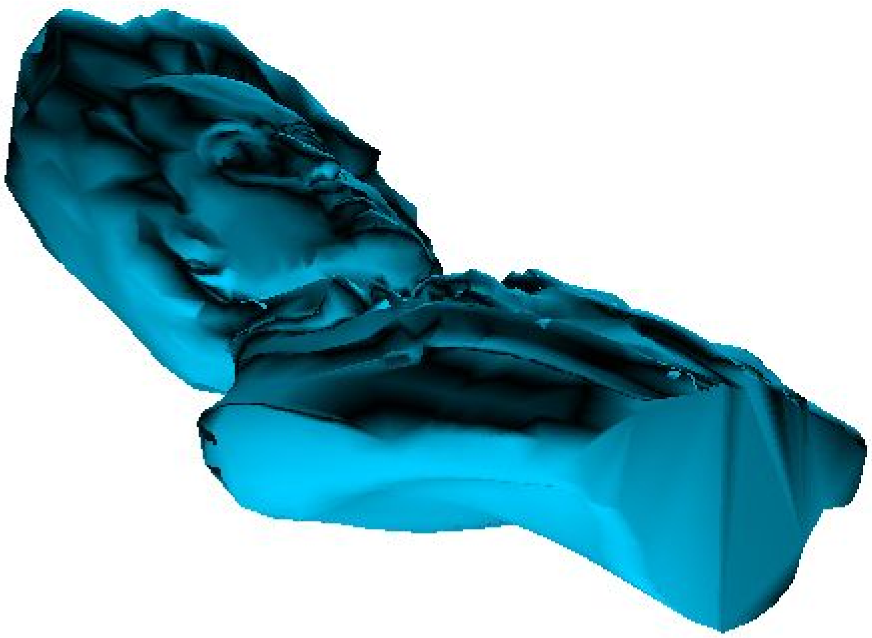
(Most PLY files don't have color information, so I annotated the PLY files with blue color data to make them more interesting. These are all included in the "ply_files" directory of my code submission)



dolphins.ply



airplane.ply



beethoven.ply

Code:

```
-- Main.hs
module Main where

import           Perf
import           System.Environment
import           Display
import           Projection                      ( genPixels )
import           Scene                           ( fromPLY
                                                 , projectScene
                                                 )


data FileArg = Name String | Invalid
getFilename :: [String] -> FileArg
getFilename [fname] = Name fname
getFilename _       = Invalid

main :: IO ()
main = do

  args  <- System.Environment.getArgs
  scene <- case (getFilename args) of
    Name s  -> fromPLY s ((-1000), (-1000), 500) ((1), (1), (1))
800
    Invalid -> error "Usage: polyrend-exe <filename>"
  projection <- return $ projectScene scene

  pixels     <- return $ genPixels projection 500
  let y1 = pixels
  Perf.time y1

  Display.render 500 pixels


----------------------------------------------------------------
-- Scene.hs
{-
3D SCENE

This module does the initial work of loading in the 3D data
and parsing it.
-}

module Scene
  ( fromPLY
  , projectScene
  )
where

import           PLY.Types
import           Lib
import           PLY
import           Data.Array
import           Data.Vector
import           Data.Map.Strict                ( Map
                                                , empty
                                                , insertWith
```

```
                                                       ,
findWithDefault
                                                       )
import qualified Data.Cross                           as Vec
import           Data.Maybe
import           Data.ByteString.Char8
import           Projection                            ( Projection(..)
                                                       , Vertex2(..)
                                                       )
import           Control.Parallel.Strategies

data Coor3D = Coor3D (Float, Float, Float) deriving (Show)
type Face = [Vertex3]
type NormalMap = Map Vertex3 (Vec.Three Float)

data Vertex3 = Vertex3 { coor3 :: Vec.Three Float
                       , rgb3 :: Rgb
                       } deriving (Show, Eq,  Ord)
data Scene = Scene { vertexList3 :: [Vertex3]
                   , faceList3 :: [Face]
                   , light3 :: Vec.Three Float
                   , focalDist :: Float
                   } deriving (Show)

fromPLY :: [Char] -> Vec.Three Float -> Vec.Three Float -> Float
-> IO Scene
fromPLY filename o lightVec focalD = do
  vertexData <- parseVertex3Ply
    (loadElements (Data.ByteString.Char8.pack "vertex")
filename)
  faceData <- parseFacePly
    (loadElements (Data.ByteString.Char8.pack "face") filename)
  return $ buildScene vertexData faceData o lightVec focalD

buildScene
  :: [Vertex3]
  -> [[Int]]
  -> Vec.Three Float
  -> Vec.Three Float
  -> Float
  -> Scene
buildScene vertexData faceData o lightVec focalD = Scene {
vertexList3 = vList
                                                          ,
faceList3   = fList
                                                          ,
light3 = lightVec
                                                          ,
focalDist   = focalD
                                                          }
 where
  vList = chunkPar (translateOrigin o) vertexData
  fList = constructFaceList vList faceData


translateOrigin :: Vec.Three Float -> Vertex3 -> Vertex3
translateOrigin ((ox, oy, oz)) Vertex3 { coor3 = (x, y, z), rgb3
= rgb3' } =
  Vertex3 { coor3 = (x - ox, y - oy, z - oz), rgb3 = rgb3' }


constructVertex3Vector :: Vector (Vector Scalar) -> [Vertex3]
constructVertex3Vector vs =
  catMaybes $ chunkPar constructVertex3 (Data.Vector.toList vs)

unpackFaceIndices :: Vector (Vector Scalar) -> [[Int]]
unpackFaceIndices faces = chunkPar
  (\x -> ((chunkPar unpackInt) . Data.Vector.toList) x)
  (Data.Vector.toList faces)
 where
  unpackInt (Sint x) = x
  unpackInt _        = error "Invalid face data"

parseVertex3Ply :: IO (Either String (Vector (Vector Scalar)))
-> IO [Vertex3]
parseVertex3Ply input = do
  elements <- input
  case elements of
```

```
    Left  _   -> error "Invalid input"
    Right dat -> return $ constructVertex3Vector dat

parseFacePly :: IO (Either String (Vector (Vector Scalar))) ->
IO [[Int]]
parseFacePly input = do
  elements <- input
  case elements of
    Left  _   -> error "Invalid input"
    Right dat -> return $ unpackFaceIndices dat


constructFaceList :: [Vertex3] -> [[Int]] -> [Face]
constructFaceList vList faceData = chunkPar
  (chunkPar ((Data.Array.!) vertexArr))
  faceData
 where
  vertexArr = array (0, Prelude.length vList - 1) (Prelude.zip
[0 ..] vList)

constructVertex3 :: Vector Scalar -> Maybe Vertex3
constructVertex3 v = case v' of
  -- Vertex3 with full color data
  [Sfloat x', Sfloat y', Sfloat z', Suchar r', Suchar g', Suchar
b'] ->
    Just $ Vertex3 { coor3 = (x', y', z'), rgb3 = (r', g', b') }
  -- Vertex3 with no color data, default RGB3 value to (0,0,0)
  [Sfloat x', Sfloat y', Sfloat z'] ->
    Just $ Vertex3 { coor3 = (x', y', z'), rgb3 = (0, 0, 0) }
  -- Invalid vertex data
  _ -> Nothing
  where v' = Data.Vector.toList v



computeFaceNormal :: Face -> Vec.Three Float
computeFaceNormal (a : b : c : _) = computeNormal a b c
 where
  computeNormal v0 v1 v2 =
    Vec.cross3 (computeDifference v0 v1) (computeDifference v0
v2)
  computeDifference Vertex3 { coor3 = (x0, y0, z0) } Vertex3 {
coor3 = (x1, y1, z1) }
    = (x1 - x0, y1 - y0, z1 - z0)
computeFaceNormal _ = error "Not enough vertices in face spec"


vertexNormalMap :: [Face] -> NormalMap
vertexNormalMap fs = Prelude.foldl (insertWith')
Data.Map.Strict.empty fs
 where
  insertWith' hm face = Prelude.foldl insertNormal hm face
where
    insertNormal hm' v =
      Data.Map.Strict.insertWith (\new old -> addVec new old) v
faceNormal hm'
    faceNormal = computeFaceNormal face


projectVertex3 :: Float -> NormalMap -> Vertex3 -> Vertex2
projectVertex3 dist m v@Vertex3 { coor3 = c@(x, y, z), rgb3 =
rgb } = Vertex2
  { coor2  = (x * fd, z * fd)
  , depth  = vecLen c
  , normal = normalize $ Data.Map.Strict.findWithDefault (0, 0,
0) v m
  , rgb2   = rgb
  }
  where fd = dist / y


{- Mapping from 3D to 2D while retaining necessary information
-}
projectScene :: Scene -> Projection
projectScene Scene { faceList3 = fs, focalDist = dist, light3 =
lit } =
  Projection { vertexList2 = Prelude.concat fs', faceList2 =
fs', light2 = lit }
```

```
 where
  vNorms = vertexNormalMap fs
  fs'    = chunkPar (Prelude.map (projectVertex3 dist vNorms))
fs

----------------------------------------------------------------
-- Projection.hs
{-
2D PROJECTION INFO

This tracks the information of the 2D projection of our scene in
order
to properly calculate the pixel display.
-}

module Projection
  ( Projection(..)
  , Vertex2(..)
  , genPixels
  )
where

import qualified Data.Cross                    as Vec
import           Data.Word                      ( Word8 )
import           Control.Parallel.Strategies
import           Lib
import           PixelDraw
import           HSE


type Face = [Vertex2]

data Vertex2 = Vertex2 { coor2 :: Vec.Two Float
                       , depth :: Float
                       , normal :: Vec.Three Float
                       , rgb2 :: Rgb
                       } deriving (Show, Eq,  Ord)
data Projection = Projection { vertexList2 :: [Vertex2]
                   , faceList2 :: [Face]
```

```
                    , light2 :: Vec.Three Float
                    }

minMaxCoors :: [Vertex2] -> (Float, Float, Float, Float)
minMaxCoors [] = error "Not enough vertices"
minMaxCoors (Vertex2 { coor2 = (x, y) } : tl) = minMaxCoor' (x,
y, x, y) tl
 where
  minMaxCoor' (minx, miny, maxx, maxy) (Vertex2 { coor2 = (x',
y') } : tl') =
    minMaxCoor' (min minx x', min miny y', max maxx x', max maxy
y') tl'
  minMaxCoor' res [] = res


faceCoors2Pixels :: Int -> [Face] -> [[Pixel]]
faceCoors2Pixels numPixels fs = parMap rpar (parMap rpar f) fs
 where
  (minX, minY, maxX, maxY) = minMaxCoors $ Prelude.concat fs
  pixelWidth = ((maxX + 1) - (minX - 1)) / (fromIntegral
numPixels)
  f (Vertex2 { coor2 = (x, y), depth = d, normal = n, rgb2 = rgb
}) = Pixel
    { pX     = (round ((x - minX + 1) / pixelWidth))
    , pY     = (round ((y - minY + 1) / pixelWidth))
    , pDepth = d
    , pRgb   = rgb
    , pNorm  = n
    }


shadePixel :: Projection -> Pixel -> Pixel
shadePixel (Projection { light2 = light }) Pixel { pX = x, pY =
y, pDepth = d, pRgb = (r, g, b), pNorm = norm }
  = Pixel { pX     = x
          , pY     = y
          , pDepth = d
          , pRgb   = snap (multRgb r, multRgb g, multRgb b)
          , pNorm  = norm
```

```
         }
 where
  intensity = abs $ (negDotProd light norm) / ((vecLen light) *
(vecLen norm))
  multRgb rgbVal = fromIntegral (round val)
    where val = (fromIntegral (toInteger rgbVal)) * intensity



{-
Main function that generates all pixels to be drawn given a 2D
projection spec
-}
genPixels :: Projection -> Int -> [Pixel]
genPixels proj@(Projection { faceList2 = fs }) screenW = parMap
  rpar
  (shadePixel proj)
  uniquePixels
 where
  uniquePixels =
    HSE.removeHiddenPixels filledInPixels $ Prelude.concat
outlines
  facePixels     = faceCoors2Pixels screenW fs
  outlines       = parMap rpar genSegments facePixels
  filledInPixels = Prelude.concat $ parMap rpar getFacePixels
outlines


----------------------------------------------------------------
-- HSE.hs
{-
HIDDEN SURFACE ELIMINATION

This module takes care of checking the z-buffer for the
shallowest depth pixel to
be included in the render.
-}

module HSE
```

```
  ( removeHiddenPixels
  )
where

import            Lib
import            Data.Map.Strict                ( Map
                                                 , lookup
                                                 , insert
                                                 , empty
                                                 , elems
                                                 , fromList
                                                 , union
                                                 )

removeHiddenPixels :: [Pixel] -> [Pixel] -> [Pixel]
removeHiddenPixels ps borderPixels =
  elems $ Prelude.foldl compareDepth Data.Map.Strict.empty ps


compareDepth :: Map (Int, Int) Pixel -> Pixel -> Map (Int, Int)
Pixel
compareDepth hm p@(Pixel { pX = x, pY = y, pDepth = d }) =
  Data.Map.Strict.insert (x, y) minPixel hm
 where
  minPixel = case (Data.Map.Strict.lookup (x, y) hm) of
    Nothing                          -> p
    Just pOld@Pixel { pDepth = d' } -> if d' > d then p else
pOld


----------------------------------------------------------------
-- PixelDraw.hs
{-
PIXEL DRAWING PROCEDURES

These functions take care of properly tracing lines of pixels
and doing scan-lines of surfaces.
-}
```

```
module PixelDraw
  ( getFacePixels
  , genSegments
  , linePixels
  )
where

import           Data.Map.Strict                ( lookup
                                                , insert
                                                , empty
                                                , elems
                                                )
import           Control.Parallel.Strategies
import           Lib

getFacePixels :: [Pixel] -> [Pixel]
getFacePixels border = Prelude.concat $ (chunkPar) linePixels
scanLines
 where
  scanLines =
    Data.Map.Strict.elems $ Prelude.foldl f
Data.Map.Strict.empty border
  f hm p = entryUpdate p hm
  entryUpdate newP@(Pixel { pX = x, pY = y }) hm =
Data.Map.Strict.insert
    y
    (newMin, newMax)
    hm
   where
    (newMin, newMax) = case (Data.Map.Strict.lookup y hm) of
      Nothing -> (newP, newP)
      Just (oldMinP@(Pixel { pX = oldMin }), oldMaxP@(Pixel { pX
= oldMax }))
        -> if x < oldMin
          then (newP, oldMaxP)
          else if x > oldMax then (oldMinP, newP) else (oldMinP,
oldMaxP)

genSegments :: [Pixel] -> [Pixel]
genSegments facePixels = Prelude.concat $ chunkPar linePixels
(segs facePixels)
  where segs a = Prelude.zip a $ Prelude.tail $ cycle a


linePixelsLow :: (Pixel, Pixel) -> [Pixel]
linePixelsLow (p0@Pixel { pX = x0, pY = y0, pDepth = d0, pRgb =
rgb0, pNorm = norm0 }, p1@Pixel { pX = x1, pY = y1, pDepth = d1,
pRgb = rgb1, pNorm = norm1 })
  = if dx == 0
    then (straightVertLine (p0, p1))
    else linePixelsLow' d0 rgb0 norm0 dStart yStart [x0 .. x1]
 where
  dz        = abs $ (d1 - d0) / dist
  dx        = x1 - x0
  dRgb      = rgbDelta rgb0 rgb1 dist
  dVec      = vecDelta norm0 norm1 dist
  dist      = (fromIntegral (x1 - x0))

  (yi, dy) = if y1 < y0 then ((-1), (-1) * (y1 - y0)) else (1,
y1 - y0)
  dStart    = (2 * dy) - dx
  yStart    = y0

  linePixelsLow' _ _ _ _ _ [] = []
  linePixelsLow' z rgb norm d y (x : tl) =
    (Pixel { pX = x, pY = y, pDepth = z, pRgb = rgb, pNorm =
norm })
      : linePixelsLow' (z + dz) (incRgb rgb dRgb) (incVec norm
dVec) d' y' tl
   where
    (y', d') =
      if d > 0 then (y + yi, d - (2 * dx) + (2 * dy)) else (y, d
+ (2 * dy))

linePixelsHigh :: (Pixel, Pixel) -> [Pixel]
linePixelsHigh (p0@Pixel { pX = x0, pY = y0, pDepth = d0, pRgb =
rgb0, pNorm = norm0 }, p1@Pixel { pX = x1, pY = y1, pDepth = d1,
pRgb = rgb1, pNorm = norm1 })
```

```
  = if dx == 0
    then (straightVertLine (p0, p1))
    else linePixelsHigh' d0 rgb0 norm0 dStart xStart [y0 .. y1]
 where
  dz       = abs $ (d1 - d0) / dist
  dy       = y1 - y0
  dRgb     = rgbDelta rgb0 rgb1 dist
  dVec     = vecDelta norm0 norm1 dist
  dist     = (fromIntegral (y1 - y0))

  (xi, dx) = if x1 < x0 then ((-1), (-1) * (x1 - x0)) else (1,
x1 - x0)
  dStart   = (2 * dx) - dy
  xStart   = x0

  linePixelsHigh' _ _ _ _ _ [] = []
  linePixelsHigh' z rgb norm d x (y : tl) =
    (Pixel { pX = x, pY = y, pDepth = z, pRgb = rgb, pNorm =
norm })
      : linePixelsHigh' (z + dz) (incRgb rgb dRgb) (incVec norm
dVec) d' x' tl
   where
    (x', d') =
      if d > 0 then (x + xi, d - (2 * dy) + (2 * dx)) else (x, d
+ (2 * dx))

straightVertLine :: (Pixel, Pixel) -> [Pixel]
straightVertLine (Pixel { pX = x, pY = y0, pDepth = d0, pRgb =
rgb0, pNorm = norm0 }, Pixel { pY = y1, pDepth = d1, pRgb =
rgb1, pNorm = norm1 })
  = straightVertLine' d0 rgb0 norm0 [y0 .. y1] where
  dz   = abs $ (d1 - d0) / dist
  dRgb = rgbDelta rgb0 rgb1 dist
  dVec = vecDelta norm0 norm1 dist
  dist = (fromIntegral (y1 - y0))
  -- interpolate color data and depth data along line
  straightVertLine' _ _ _ [] = []
  straightVertLine' z rgb norm (hd : tl) =
    (Pixel { pX = x, pY = hd, pDepth = z, pRgb = rgb, pNorm =
norm })
      : straightVertLine' (z + dz) (incRgb rgb dRgb) (incVec
norm dVec) tl

linePixels :: (Pixel, Pixel) -> [Pixel]
linePixels (p0@Pixel { pX = x0, pY = y0 }, p1@Pixel { pX = x1,
pY = y1 }) =
  xyLine
 where
  xyLine = if (abs (y1 - y0)) < (abs (x1 - x0))
    then if x0 > x1 then linePixelsLow (p1, p0) else
linePixelsLow (p0, p1)
    else if y0 > y1 then linePixelsHigh (p1, p0) else
linePixelsHigh (p0, p1)


----------------------------------------------------------------
-- Lib.hs
{-
CONVENIENCE FUNCTIONS

This module holds several convenience functions for various
necessary
calculations.
-}


module Lib
  ( Pixel(..)
  , Rgb(..)
  , vecLen
  , incVec
  , incRgb
  , vecDelta
  , snap
  , negDotProd
  , rgbDelta
  , normalize
  , addVec
  , chunkPar
  )
where
import           Data.Word                    ( Word8 )
import qualified Data.Cross                   as Vec
import           Control.Parallel.Strategies



type Rgb = (Word8, Word8, Word8)
data Pixel = Pixel { pX :: Int
                   , pY :: Int
                   , pDepth :: Float
                   , pRgb :: Rgb
                   , pNorm :: (Vec.Three Float)
                   } deriving (Show)



incRgb :: Rgb -> (Float, Float, Float) -> Rgb
incRgb ((w0, w1, w2)) (dr, dg, db) = (addRgb w0 dr, addRgb w1
dg, addRgb w2 db)
 where
  addRgb rgbVal delta = fromIntegral (round fVal)
    where fVal = (fromIntegral (toInteger rgbVal)) + delta

vecDelta :: Vec.Three Float -> Vec.Three Float -> Float ->
Vec.Three Float
vecDelta (x0, y0, z0) (x1, y1, z1) dist =
  ((x1 - x0) / dist, (y1 - y0) / dist, (z1 - z0) / dist)

incVec :: Vec.Three Float -> Vec.Three Float -> Vec.Three Float
incVec (x, y, z) (dx, dy, dz) = normalize (x + dx, y + dy, z +
dz)


normalize :: Vec.Three Float -> Vec.Three Float
normalize (x, y, z) = (x / vLen, y / vLen, z / vLen)
```

```
  where vLen = vecLen (x, y, z)

vecLen :: Vec.Three Float -> Float
vecLen (x, y, z) = sqrt ((x ** 2) + (y ** 2) + (z ** 2))

snap :: Rgb -> Rgb
snap ((r, g, b)) = (snapVal r, snapVal g, snapVal b)
 where
  snapVal rgbVal = fromIntegral (min 255 (max 0 val))
    where val = (fromIntegral (toInteger rgbVal))

negDotProd :: Vec.Three Float -> Vec.Three Float -> Float
negDotProd (x0, y0, z0) (x1, y1, z1) =
  (-1) * ((x0 * x1) + (y0 * y1) + (z0 * z1))

rgbDelta :: Rgb -> Rgb -> Float -> (Float, Float, Float)
rgbDelta ((r0, g0, b0)) ((r1, g1, b1)) dist =
  (wordDiffDiv r0 r1 dist, wordDiffDiv g0 g1 dist, wordDiffDiv
b0 b1 dist)
 where
  wordDiffDiv w0 w1 d =
    (fromIntegral ((toInteger w1) - (toInteger w0)) :: Float) /
dist

addVec :: Vec.Three Float -> Vec.Three Float -> Vec.Three Float
addVec (x0, y0, z0) (x1, y1, z1) = (x0 + x1, y0 + y1, z0 + z1)


chunkedParMap :: Int -> Strategy b -> (a -> b) -> [a] -> [b]
chunkedParMap n strat f = withStrategy (parListChunk n strat) .
map f

chunkPar = chunkedParMap 1000 rpar


-----------------------------------------------------------------
-- Perf.hs

{-
PERFORMANCE MEASUREMENT

Small module to get the runtime of the algorithm so that the
graphics display
doesn't change the runtime calculation
-}

module Perf
  ( time
  )
where

import           System.CPUTime
import           Text.Printf
import           Control.Exception
import           Control.Parallel.Strategies
import           Control.Monad
import           Control.DeepSeq
import           System.Environment
import           System.Directory

lim :: Int
lim = 10 ^ 6

myrnf a = a `seq` ()

time :: t -> IO ()
time y = do
  start <- getCPUTime
  replicateM_ lim $ do
    x <- evaluate $ y
    myrnf x `seq` return ()
  end <- getCPUTime
  let diff = (fromIntegral (end - start)) / (10 ^ 12)
  Prelude.putStrLn "Rendering Computation Runtime:"
  printf "Computation time: %0.9f sec\n" (diff :: Double)
  printf "Individual time: %0.9f sec\n"  (diff / fromIntegral
lim :: Double)
  return ()
```

```
{-
DISPLAY CODE

Used to render the actual pixels to the screen using Gloss
-}


module Display
  ( render
  )
where

import           Graphics.Gloss
import           Data.Word                       ( Word8 )
import           Lib                             ( Pixel(Pixel)
                                                 , Pixel(pX)
                                                 , Pixel(pY)
                                                 , Pixel(pRgb)
                                                 , Rgb
                                                 )
import           Data.Map                        ( Map
                                                 , fromList
                                                 ,
findWithDefault
                                                 )
import           Data.ByteString                 ( ByteString
                                                 , pack
                                                 )

render :: Int -> [Pixel] -> IO ()
render screenW pixels = display
  (InWindow "Display Window" (600, 600) (60, 60))
  white
  picture
 where
  picture = bitmapOfByteString screenW
                               screenW
                               (BitmapFormat BottomToTop PxRGBA)
```

```
                                bitmapData
                                True
  bitmapData = pixels2BitMap screenW screenW pixels

pixels2BitMap :: Int -> Int -> [Pixel] -> ByteString
pixels2BitMap screenW screenH pixels = pack
  (genBitMap 0 (screenW * screenH) filledPixels)
  where filledPixels = fromList $ Prelude.map (pixel2ByteIndex
screenW) pixels


genBitMap :: Int -> Int -> Map Int Rgb -> [Word8]
genBitMap n len hm = if n < len
  then r : g : b : 255 : (genBitMap (n + 1) len hm)
  else []
  where (r, g, b) = Data.Map.findWithDefault ((255, 255, 255)) n
hm

pixel2ByteIndex :: Int -> Pixel -> (Int, Rgb)
pixel2ByteIndex w (Pixel { pX = x, pY = y, pRgb = rgb' }) = ((y
* w) + x, rgb')
```