Alumno:	Erik Rangel Limón	**Programación Declarativa**
No. Cuenta:	318159287	*2023-2*
Correo	erikrangel.014@ciencias.unam.mx	Proyecto Final

Cierre Convexo

Uno de los primeros problemas que se abordan en *Geometría Computacional* es el cálculo del cierre convexo para una nube de puntos.

El cierre convexo para una nube de puntos S se define como el polígono convexo más pequeño que contiene a todos los puntos de S. Para calcularlo se proponen múltiples algoritmos que utilizan algunas primitivas geométricas, como la dirección a la que se encuentra un punto con respecto a otros dos, dado un punto verificar que éste se encuentre dentro de un triángulo, entre otros. La utilización de estas primitivas, así como el uso de ciertas propiedades geométricas permiten mejorar la complejidad en tiempo para los algoritmos de esta tarea, llegando a una cota inferior de $O(n \log n)$.

Así, la idea del cierre convexo se extiende para la solución de otros problemas, como el cierre convexo para un conjunto de puntos en más dimensiones, la envolvente convexa de un polígono monótono, y por ejemplo, el que trabajé para éste proyecto, el cierre convexo para un conjunto de discos.

Cierre Convexo para un conjunto de Círculos

El problema ahora es el siguiente: Se tiene un conjunto de discos S en posición general que posiblemente se intersectan y tienen tamaño arbitrario, para el cual se desea calcular la región convexa de S.

Ahora la solución es distinta, pues esta región convexa, sus bordes pueden ser o bien segmentos de recta o segmentos de círculo, y un mismo círculo puede aparecer múltiples veces a lo largo del borde del cierre convexo.

En el artículo *A convex hull algorithm for discs, and applications* se propone una solución que utiliza una estrategia *divide y vencerás*, pues toma el conjunto S, lo particiona en dos conjuntos P y Q de aproximadamente el mismo tamaño, calcula recursivamente el cierre convexo de P y de Q y une ambos cierres en el cierre convexo $P \cup Q$ con un procedimiento *merge*.

El procedimiento *merge* tampoco es difícil de comprender; consiste en elegir una recta dirigida L, en cada paso tomar elementos p y q de P y de Q respectivamente tales que su recta tangente al disco y paralela a L más a la izquierda con respecto a L, P (o Q, según sea el caso) está contenido en su respectiva mitad derecha, y decidir cuál de ambas rectas su mitad de plano contiene a la otra, en cuyo casó será el círculo que añadiremos al cierre convexo del resultado, continuamos el procedimiento girando la línea L en sentido antihorario con respecto al ángulo más pequeño entre los siquientes elementos de p o de q hasta haber visitado cada uno de los círculos.

Implementación

Mi proyecto se divide en dos partes, la primera parte resuelve el cierre convexo de un conjunto de círculos en *Haskell*, ésta incluye la definición de varias primitivas geométricas y la implementación del algoritmo como tal. La segunda es la implementación de una interfaz sencilla para dibujar círculos en *Elm*, lo que incluye guardar los círculos en un archivo de texto y leer un archivo con la solución.

Haskell

En la raiz del repositorio está una carpeta con el nombre `src` y dentro de ésta se encuentran cuatro archivos:

- `CircleReader.hs`
- `ConvexDiscs.hs`
- `Main.hs`
- `Primitives.hs`

Para compilar el programa recomiendo usar las siguientes órdenes:

```
> cd src
> mkdir build lib
> compile Main.hs -o lib/Main -odir build/ -hidir build/
```

Esto dado que `Main` es un programa que se ejecuta con parámetros.

Primitives.hs

En éste módulo como lo indica su nombre, se incluye la especificación de tipos de datos geométricos y funciones que serán las primitivas geométricas que necesitaremos para el algoritmo.

Los tipos de dato que utilicé fueron los siguientes:

- Para representar un punto definido por sus coordenadas.

```haskell
type Point = (Double, Double)
```

- Para representar una dirección colineal, horaria o antihoraria

```haskell
data Orientation = Col
                 | Ccw
                 | Cw deriving (Show, Eq)
```

- Para representar una línea

```haskell
type Line = (Point, Point)
```

- Para representar una línea dirigida que es un punto y su vector director.

```haskell
data VectDir = VectDir { p :: Point
                       , dx :: Double
                       , dy :: Double
                       } deriving Show
```

- Para representar un disco, por un identificador que debe ser único, su centro y su radio

```haskell
data Disc = Disc { did :: String
                 , center :: Point
                 , radius :: Double} deriving (Show, Eq)
```

- Para representar una lista "circular"

```haskell
data InfList a = Inf [a]
```

Las funciones definidas en `Primitives.hs` se apoyan de éstos tipos de dato para los cálculos necesarios y su funcionalidad está explicada en el archivo.

De éstas funciones quisiera hacer énfasis en algunas, dos de ellas son precisamente las que se especifican en el artículo y las demás de cómo funcionan las listas "circulares"

- `dom`

```haskell
dom :: VectDir -> VectDir -> Bool
dom = ((Ccw /=) .) . (. p) . lineOrientation . dirToLine
```

Está escrito en forma libre de puntos, sin embargo lo que hace es tomar dos lineas dirigidas (paralelas) y regresa `true` si la primera se encuentra más a la izquierda que la segunda, lo que es equivalente a revisar si la mitad de plano derecha de la primer linea contiene a la mitad de plano de la segunda.

- `tangentFromDiscToDisc`

```haskell
tangentFromDiscToDisc :: Disc -> Disc -> Maybe Line
tangentFromDiscToDisc discA discB = do
      (l1,l2) <- outterTangents discA discB
      if radius discA < radius discB then do
        let fun (x,y) = (y,x)
        let (fl1, fl2) = (fun l1, fun l2)
        case lineOrientation fl2 (snd fl1) of
          Cw -> return fl2
          _ -> return fl1
        else case lineOrientation l2 (snd l1) of
               Cw -> return l2
               _ -> return l1
```

Ésta función lo que hace es intentar obtener las lineas tangentes exteriores de dos círculos A y B (si es que existen) y regresa aquella que está a la izquierda con respecto a los centros de A a B. Esta función es equivalente a $L(a, b)$ que es la que se utiliza en el artículo.

- `inf` y `mkInf`

```haskell
inf :: [a] -> [a]
inf [] = []
inf xs = xs ++ inf xs

mkInf :: [a] -> InfList a
mkInf = Inf . inf
```

inf hace lo mismo que *cycle* en haskell, sin embargo, éste permite el uso de listas vacías, en cuyo caso regresa la lista vacía, ésto para poder manejar errores más fácilmente. Y *mkInf* aplica la función *inf* pero la introduce en un contexto para especificar que la lista es infinita o "circular".

- `next` y `current`

```haskell
next :: InfList a -> InfList a
next (Inf []) = Inf []
next (Inf (_:xs)) = Inf xs

current :: InfList a -> Maybe a
current (Inf []) = Nothing
current (Inf (x:_)) = Just x
```

next lo que hace es "girar" la lista circular haciendo que ésta apunte al siguiente elemento. Y *current* lo que hace es intentar obtener el elemento que se encuentra en la cabeza de la lista.

ConvexDisc.hs

En éste módulo se implementa la funcionalidad principal para calcular el cierre convexo de un conjunto de círculos.

Destacaré las funciones esenciales para el cierre convexo:

- `convexHull`

```haskell
convexHull :: [Disc] -> Maybe [Disc]
convexHull [] = Just []
convexHull [x] = Just [x]
convexHull xs = do
  let n = length xs `div` 2
  let p = take n xs
  let q = drop n xs
  hullP <- convexHull p
  hullQ <- convexHull q
  hullP `merge` hullQ
```

Este divide la lista en dos y calcula recursivamente su cierre convexo

- `merge`

```
merge :: [Disc] -> [Disc] -> Maybe [Disc]
merge hullP hullQ = do
  let cutP = if length hullP > 1 then cut hullP else hullP
  let cutQ = if length hullQ > 1 then cut hullQ else hullQ
  let hP = if (did (head cutP) == did (last cutP)) && (length cutP > 1) then
             init cutP
           else
             cutP
  let hQ = if (did (head cutQ) == did (last cutQ)) && (length cutQ > 1) then
             init cutQ
           else
             cutQ
  let p = mkInf hP
  let q = mkInf hQ
  cp <- current p
  cq <- current q
  let minY = if (snd (center cp) - radius cp) < (snd (center cq) - radius cq) then
               cp
             else
               cq
  let l = VectDir { p = (fst $ center minY, snd (center minY) - radius minY)
                  , dx = -1
                  , dy = 0}
  let lp = paraFromCircle l cp
  let lq = paraFromCircle l cq
  merge' [] hP hQ p q l lp lq
```

Esta función inicializa lo necesario del procedimiento *merge*, como la linea inicial y los círculos p y q que se necesitan así como sus rectas tangentes lp y lq respectivamente. Hace un procedimiento de recorte para pasos anteriores pues el algoritmo puede regresar más discos de los necesarios, pero no afectan la complejidad pues éstos toman tiempo lineal y además *merge* no se ejecuta recursivamente, por lo que éste recorte sólo se ejecuta dos veces en cada llamada a *merge*.

- *merge'*

```
merge' :: [Disc]
       -> [Disc]
       -> [Disc]
       -> InfList Disc
       -> InfList Disc
       -> VectDir
       -> VectDir
       -> VectDir
       -> Maybe [Disc]
merge' hS [] [] p q l lp lq = do
  cp <- current p
  cq <- current q
  (hS', _, _, _) <- if dom lp lq then
```

```
                    advance (add hS cp) l p q
                  else
                    advance (add hS cq) l q p
  return $ reverse hS'
merge' hS hP hQ p q l lp lq = do
  cp <- current p
  cq <- current q
  (hS', l', x, y) <- if dom lp lq then
                       advance (add hS cp) l p q
                     else
                       advance (add hS cq) l q p
  let (p', q') = if dom lp lq then
                   (x, y)
                 else
                   (y, x)
  cp' <- current p'
  cq' <- current q'
  let (lp', lq') = (paraFromCircle l' cp', paraFromCircle l' cq')
  let (hP', hQ') = case (hP, hQ) of
                     ([], ys) -> ([], remove ys)
                     (xs, []) -> (remove xs, [])
                     (xs, ys) -> if dom lp lq then
                                   (remove xs, ys)
                                 else
                                   (xs, remove ys)
  merge' hS' hP' hQ' p' q' l' lp' lq'
```

Ésta función es la parte recursiva del proceso *merge*, en ésta se revisa la posición de cada círculo, qué circulos se deben agregar, hacia donde deben girar las listas circulares y como marcar los elementos visitados.

- advance

```
advance :: [Disc]
        -> VectDir
        -> InfList Disc
        -> InfList Disc
        -> Maybe ([Disc], VectDir, InfList Disc, InfList Disc)
advance hS l x y = do -- (hS', lineToDir l', x', y')
  cx <- current x
  cy <- current y
  cnx <- current $ next x
  cny <- current $ next y
  let line1 = tangentFromDiscToDisc cx cy
  let line2 = tangentFromDiscToDisc cx cnx
  let line3 = tangentFromDiscToDisc cy cny
  let line4 = tangentFromDiscToDisc cy cx
  let hS' = if isFirstMin l [line1, line2, line3] then
```

```
                if isFirstMin l [line4, line2, line3] then
                  cx : (add hS cy)
                else
                  add hS cy
              else
                hS
    (l', x',y') <-
      if isFirstMin l [line2, line3] then do
        l2 <- line2
        return (lineToDir l2, next x, y)
      else
        case line3 of
          Nothing -> return (l, x, y)
          Just l3 -> return (lineToDir l3, x, next y)
    return (hS', l', x', y')
```

Este procedimiento puede agregar más círculos al cierre convexo dependiendo si encuentra un puente entre ambos cierres, también indica hacia dónde se debe girar la linea principal.

CircleReader.hs

En éste módulo se especifican funciones para pasar de una cadena de texto a una lista de círculos, y para pasar de un conjunto de lineas a una cadena de texto.

También tiene funciones para leer y escribir archivos a partir de las funciones explicadas anteriormente de `CircleReader.hs` y de `ConvexCircles.hs`

Main.hs

En éste módulo se encuentra el programa principal; éste importa la paquetería `System.Environment` para obtener los argumentos con los que fue llamado el programa; el primer argumento debe ser un archivo en el que se encuentren definidos un conjunto de círculos descritos en cada línea del archivo con un identificador, el centro y su radio, cada uno de estos parámetros separados por una coma; el segundo archivo debe ser la ruta que se desea guardar el archivo del resultado.

Después de compilarse, debe ejecutarse como sigue:

```
> ./Main input.txt output.txt
```

Elm

En la raíz del repositorio se encuentra una carpeta con el nombre `canvas`.

Para compilar el proyecto recomiendo usar las siguientes órdenes:

```
> cd canvas
> elm make src/CirclesDraw.elm
```

Se generará un archivo `index.html` que se puede ejecutar en el navegador.

Dentro de la carpeta `canvas/src` se encuentran dos archivos:

- `CirclesDraw.elm`
- `Playground.elm`

Cabe decir que el archivo `Playground.elm` no lo hice yo, y la fuente está en github.
Éste no lo usé en su totalidad, pero lo modifiqué para poder importar algunas funciones para utilizar figuras y convertirlas a `svg`.

CiclesDraw.elm

Elm es un lenguaje de programación funcional diseñado con el propósito principal de facilitar el desarrollo *front-end* para aplicaciones web.
El compilador de *Elm* está escrito en *Haskell*; y se compila directamente a código de *JavaScript*.
Por lo general el código de *Elm* se estructura de la siguiente manera:

- Se define un **modelo**, que puede ser un tipo de dato cualquiera que sea de interés para el uso que se le va a dar en la aplicación.

  En este caso el modelo que utilicé fue un *record* que guardaba información de la pantalla, el ratón y del estado de los círculos que el usuario ha dibujado.

```
type alias Model =
    { screen : Screen
    , mouse : Mouse
    , circleState : CircleState}
```

- Se definen los **mensajes**, que van a ser acciones que se van a aceptar en la aplicación.

  En este caso definí los mensajes para aceptar cambios en la pantalla, en la posición y pulsación del ratón, y algunos otros para procesar las acciones de descargar, cargar y decodificar archivos.

```
type Msg = NewViewport Dom.Viewport
         | Resize Int Int
         | MouseMove Float Float
         | MouseButton Bool
         | Save
         | Load
         | Received File
         | Decoded String
```

- Se define una función **update** que recibe un mensaje y el modelo, en donde a partir del mensaje recibido puedes modificar el modelo y generar un comando que se deseé ejecutar.

  En este caso lo definí para procesar los círculos que se estaban dibujando según la posición del ratón, modificar el estado de la pantalla, y para cargar / descargar archivos.

```
update msg model =
    case msg of
        NewViewport {viewport} ->
            ( act { model | screen = toScreen viewport.width viewport.height}
            , Cmd.none )
```

```
        Resize width height ->
            ( act { model | screen = toScreen (toFloat width) (toFloat height) }
            , Cmd.none )
        Save ->
            ( act model
            , Download.string "text.txt" "text/txt"
                ( String.join "" (List.map stringCircle model.circleState.circles) ) )
        Load ->
            ( act model
            , Select.file ["text/txt"] Received )
        Received file ->
            ( act model
            , Task.perform Decoded ( File.toString file ) )
        Decoded string ->
            ( act { model | circleState = { circles = model.circleState.circles
                                          , hull = List.map readLine (String.lines string)
                                          , current = model.circleState.current } }
            , Cmd.none )
        MouseMove x y ->
            ( act { model | mouse = { x = model.screen.left + x
                                    , y = model.screen.top - y
                                    , down = model.mouse.down } }
            , Cmd.none )
        MouseButton val ->
            ( act { model | mouse = { x = model.mouse.x
                                    , y = model.mouse.y
                                    , down = val } }
            , Cmd.none)

act model =
    let
        left = model.screen.left
        top = model.screen.top
        circles = model.circleState.circles
        current = model.circleState.current
        mouseDown = model.mouse.down
        x = model.mouse.x
        y = model.mouse.y
    in
        if x <= left + 100 && y >= top - 100 then
            model
        else
            case (circles, current) of
                (xs, Nothing) -> if mouseDown then
                                    { model | circleState =
                                                { circles = xs
                                                , hull = []
                                                , current =
                                                    Just { did = assignId xs
                                                         , center = (x,y)
                                                         , radius = 0} } }
                                 else
                                     model
                (xs, Just disc) -> if mouseDown then
                                       { model | circleState =
                                                   { circles = xs
                                                   , hull = []
                                                   , current =
                                                       Just { disc | radius =
                                                                     distance disc.center (x,y)}}}
                                   else
                                       if disc.radius > 0 then
                                           { model | circleState = { circles = disc :: xs
                                                                   , hull = []
```

```
                                        , current = Nothing } }
    else
        { model | circleState = { circles = xs
                                , hull = []
                                , current = Nothing} }
```

- Se define una función **view** la cual se utiliza para imprimir en pantalla el estado del modelo.

  En este caso lo definí para dibujar en todo momento los círculos que ha dibujado el usuario, así como las líneas del resultado del cierre convexo.

```
view model = { title = "Cierre Convexo"
             , body = render model ::
                      [Ht.div [H.style "position" "fixed"]
                           [ Ht.div [] [ Ht.button [He.onClick Save] [ text "Guardar" ] ]
                           , Ht.div [] [ Ht.button [He.onClick Load] [ text "Cargar" ] ]
                           , Ht.div [] [ Ht.text (String.fromFloat model.screen.width ) ]
                           , Ht.div [] [ Ht.text (String.fromFloat model.screen.height ) ]
                           , Ht.div [] [ Ht.text (String.fromFloat model.mouse.x) ]
                           , Ht.div [] [ Ht.text (String.fromFloat model.mouse.y) ]
                           , Ht.div [] [ Ht.text (if model.mouse.down then "Clicked" else "Unclicked")]]]}
```

La idea para usar este proyecto es abrir en un navegador el archivo `index.html`, dibujar los círculos y pulsar el botón de “Guardar” para descargar el archivo con la información de los círculos. Recomiendo guardarlo en la misma carpeta en donde se encuentre el programa “Main”. Posteriormente ejecutar el programa “Main” con el nombre del archivo de los círculos y el nombre del archivo donde se guardará el resultado. Finalmente pulsar el botón de “Cargar” y seleccionar el archivo en donde se guardó el resultado.