

# 3. Streaming Hedgehogs

Rita Borgo

# Vector Visualization



- Data set is given by a vector component and its magnitude
- often results from study of fluid flow or by looking at derivatives (rate of change) of some quantity
- Many visualization techniques proposed:
  - Hedgehogs / glyphs
  - Particle tracing
  - stream -, streak -, time - & path -lines
  - stream -ribbon, stream -surfaces, stream -polygons, stream -tube
  - hyper-stream lines
  - Line Integral Convolution

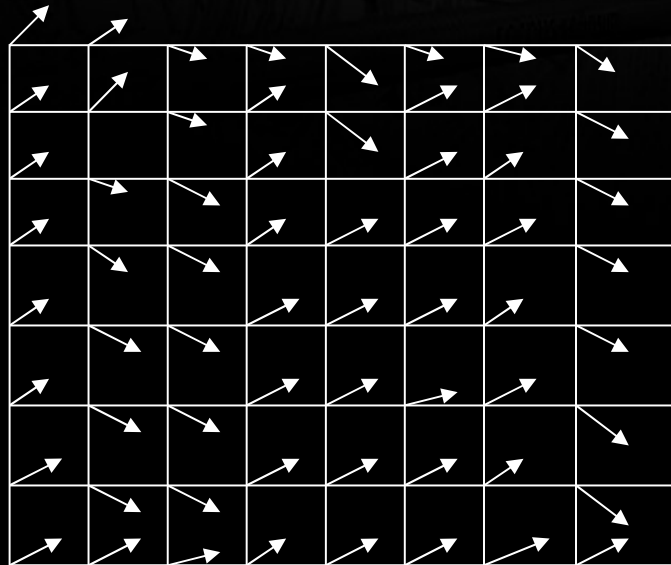
# Vector Field Visualization Techniques



- Local technique : Advection based methods
  - Display the trajectory starting from a particular location
    - Streamxxx
    - Contours
- Global technique:
  - Display the flow direction everywhere in the field
    - Hedgehogs
    - Line Integral Convolution
    - Texture
    - Splats etc.

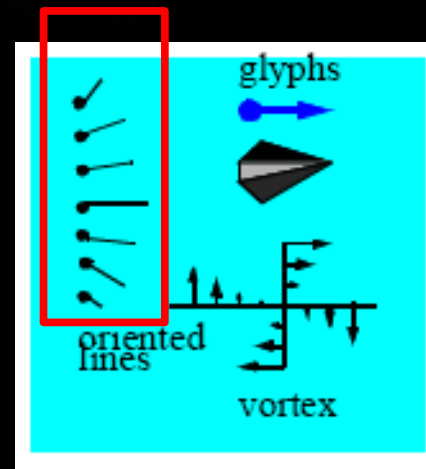
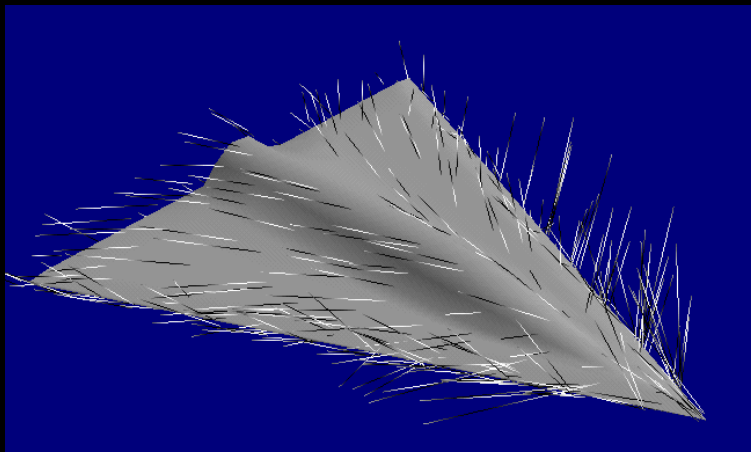
# Global techniques

- Display the entire flow field in a single picture
- Minimum user intervention
- Example: Hedgehogs (global arrow plots)



# Mappings - Hedgehogs, Glyphs

- Put “icons” at certain places in the flow
  - e.g. arrows - represent direction & magnitude
- other primitives are possible



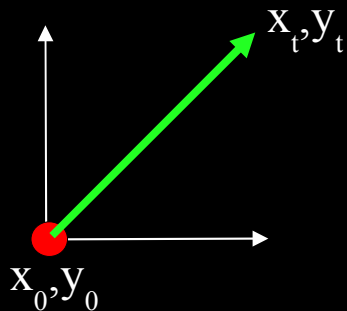
# Problem Definition

Given (typically):

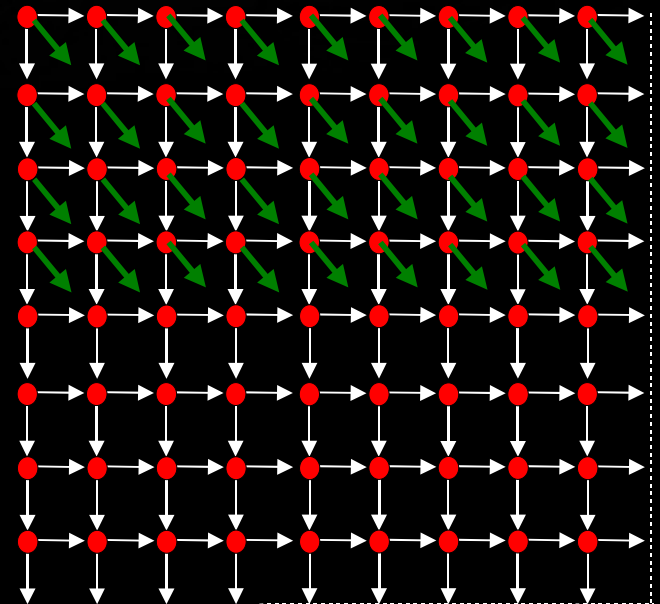
- physical **position** (coordinate system  $x, y$ )
- **velocity** (vector  $v(x, y)$ ),
- plus some other scalar eventually (temperature, pressure)...

We want to visualize a function  $F$

$F :: \text{Coord Float} \rightarrow (\text{Coord Float}, \text{Coord Float})$



Hog Function



# First Step – Defining the Grid Space

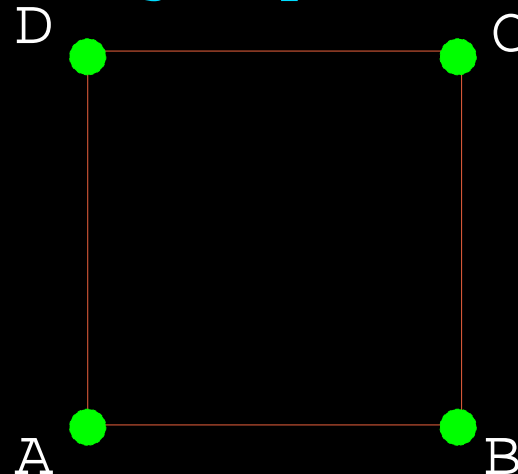
- As our starting point we explore a possible representation of a dataset as a rectilinear regular grid;
- Open the Hedgehogs\_template.hs file,
- First we define a Square Cell as a data type of four values

```
data Square a = Square a a a a deriving (Eq,Ord,Show)
```

- We name its vertices and set up the 3D coordinate system

```
type Coord = Vertex3 (GLfloat)
```

```
data Vertex = A | B | C | D deriving (Eq,Show,Enum)
```



# First Step – Building the Grid Space

- Given: the size of a dataset in \*cells\* (==rowsize, colsize) and the stream of point samples construct a stream of cells.

```
squareGrid :: (Int,Int) -> [a] -> [Square a]
squareGrid (rowsize,colsize)=
  discontinuities (0,0) . make_cells
```

where

```
make_cells stream =
  zipWith4 Square stream
    (drop 1 stream)
    (drop (rowsize+1) stream)
    (drop rowsize stream)
```

```
discontinuities _ [] = []
```

```
discontinuities (i,j) (x:xs)
```

```
| j==colsize-1
```

```
= []
```

```
| i==rowsize-1
```

```
= discontinuities (0,j+1) xs
```

```
| otherwise
```

```
= x: discontinuities (i+1,j) xs
```

# Second Step – Defining Utility Methods

- A cell is a collection of values that lie at the vertices, we need to define a function which given a vertex returns its **value**

```
select :: Vertex -> Square a -> a  
select n (Square a b c d)
```

```
= case n of
```

A	->	a
B	->	b
C	->	c
D	->	a

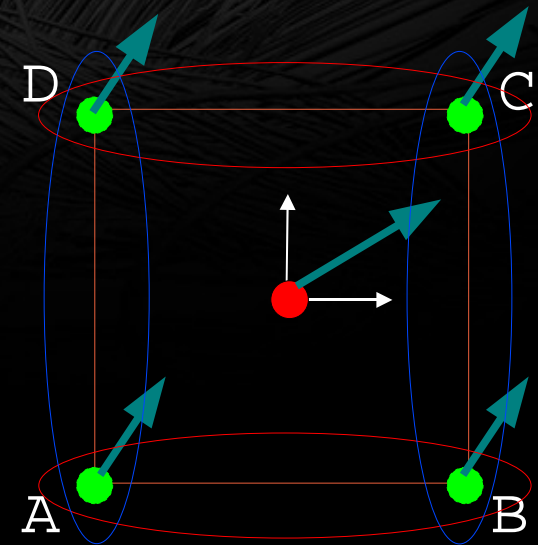
- Example of how to compute the cell centre

```
centre = midc (select A geom) (select C geom)
```

```
midc :: Coord -> Coord -> Coord
```

```
midc (Vertex3 x1 y1 z1) (Vertex3 x2 y2 z2) =
```

```
Vertex3 (x1 + (x2-x1)/2) (y1 + (y2-y1)/2) (z1 + (z2-z1)/2)
```



# Second Step – Defining the Utility Methods



- Finally just routine functions;
  - read\_data just reads in data from a file

```
read_data :: String -> IO [Float]
read_data filename =
  do { h <- openFile (filename ++ ".dat") ReadMode
      ; b <- Fast.hGetContents h
      ; return $ map sampleToFloat . bytesToSamples $ b
    }
```

x0-4-599y0-4-247z124t150.Vx.dat

# Second Step – Defining the Utility Methods



- Function Main, similar to what previously described

```
main :: IO ()
main =
  do { (_, [filename, s, opt]) <- GLUT.getArgsAndInitialize
      ; datasetX <- read_data (filename ++ ".Vx")
      ; datasetY <- read_data (filename ++ ".Vy")
      ; let rngx = maximum datasetX - minimum datasetX
          ; let rngy = maximum datasetY - minimum datasetY

          ; g <- openGraphics "NGHedgehogs" (800,600)
          ; let mkgrid = squareGrid (150,62)
              coords = mkgrid $ [Vertex3 cx cy 124.0 | cy <- [0.0 .. 61.0]
                                   , cx <- [0.0 .. 149.0]]
          ; let hog_strm = ngHedge (rngx,rngy) (read s)
              (mkgrid $ zip datasetX datasetY) coords
          ; if opt == "h" -- "g" for glyphs
            then addScene g $ [line_segments hog_strm]
            else addScene g $ [glyphs hog_strm]
          ; GLUT.mainLoop
      }
```

# Third Step – Hedgehogs

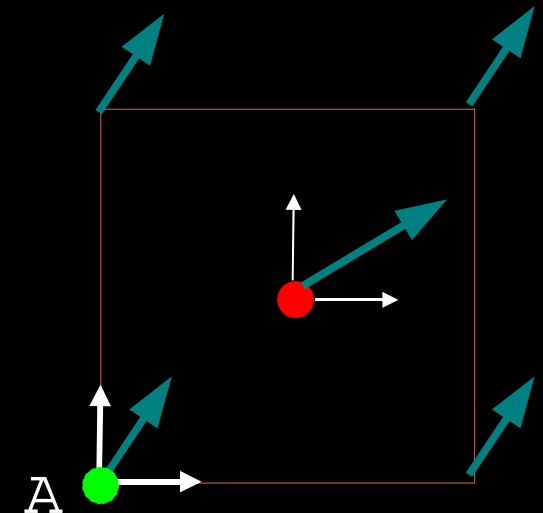
- What do we need now?
  - a function which computes one hedgehog (module and direction) per sample point

```
Hog :: (Float,Float) ->  
      Float ->  
      Square (Float,Float) ->  
      Square Coord ->  
      (Coord, Coord)
```

} Function Signature

```
Hog range scale vectors geom = ...
```

Do not worry about the hedgehog geometry drawing



# Third Step – Hedgehogs Geometry

- The `line_segments` function creates the geometry for the oriented lines

```
line_segments :: [(Coord, Coord)] -> HsScene
line_segments lines =
  Geometry static Lines [HsGeom_cv red verts] (xcx, ycy)
  where
    white = Color4 1.0 1.0 1.0 1.0 :: Color4 GLfloat
    red   = Color4 1.0 0.0 0.0 1.0 :: Color4 GLfloat
    verts = concat $ map mkAbsolute lines
    mkAbsolute (Vertex3 cx cy cz, Vertex3 dx dy dz)
      = [Vertex3 cx cy cz, Vertex3 (cx+dx) (cy+dy) dz]
```



# Third Step – Stream Hedgehogs Finally



- What else do we need?
  - A function that builds a STREAM of hedgehogs

```
Hedge :: (Float,Float) ->  
      Float ->  
      [Square (Float,Float)] ->  
      [Square Coord] ->  
      [(Coord, Coord)]
```

```
Hedge range scale vectors geom = ...
```

# Your Turn Now



- Simple Exercises:

- Provided the function signature for *Hedge* and *Hog* describe a possible implementation of these functions
- As described during the previous session the *zip* family of functions is used to turn a tuple-of-streams into a stream-of-tuples. Provided this information try to develop a streaming implementation of the hedgehogs
- The *Hog* function can be implemented in several different ways:
  - Straightforward hedgehog for each sample (*Hog*)
  - Vector derivative at centre (*HogCentre*)
  - ...

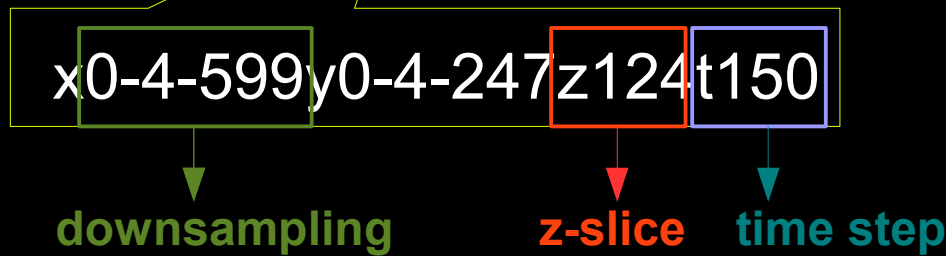
Can you provide implementation of *Hog* and *HogCentre* where the difference between the bodies of the two functions is minimal?

# Fourth Step – 'How to's for your program

- Open the `Hedgehogs_template.hs` file, you will find the skeleton for the *Hedge* and *Hog* functions. You will have to modify their bodies...

- To compile your program simply run :  
`ghc --make Hedgehogs`

- To run your program simply type:  
`./Hedgehogs filename 1 h`



# Exercises Solution

# Possible Solution – The Hog Function

- A function which computes one hedgehog (module and direction) per sample point
  - Base case: hedgehog computed at vertex V0 (== A) of each Cell

```
Hog :: (Float,Float) -> Float -> Square (Float,Float) ->  
      Square Coord -> (Coord, Coord)
```

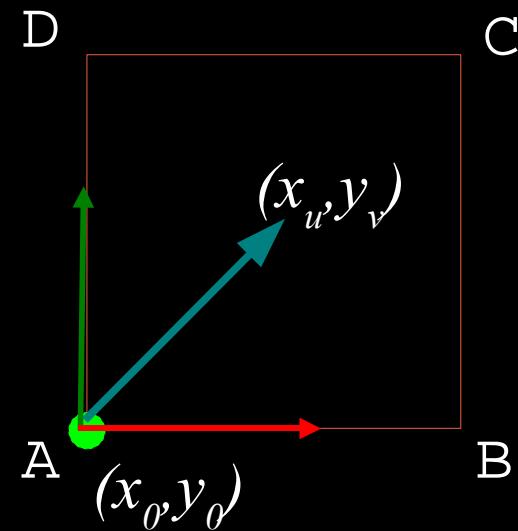
```
Hog range scale vectors geom =
```

```
(v0, Vertex3 (scale* u/rngx) (scale* v/rngy) z)
```

where

```
(u,v) = select A vectors
```

```
v0@(Vertex3 _ _ z) = select A geom
```



# Possible Solution: The Hedge Function



- A function that builds a STREAM of hedgehogs

```
Hedge :: (Float,Float) -> Float -> [Square (Float,Float)] ->  
      [Square Coord] -> [(Coord, Coord)]
```

```
Hedge range scale vectors geom =
```

```
  zipWith (Hog range scale) vectors geom
```

Here goes the STREAM

# Possible Solution – The Hog Function with Vector at the Centre of the Cell



- Advanced case: hedgehog computed at centre of each Cell
  - Vector field components at centre are computed averaging the vector field values at vertices of cell

```
HogCentre :: (Float,Float) -> Float ->  
           Square (Float,Float) -> Square Coord ->  
           (Coord, Coord)
```

```
HogCentre range scale vectors geom =
```

```
  (centre, (Vertex3 (scale* avg fst / rngx)  
            (scale* avg snd / rngy) z))
```

```
where
```

```
  avg xOrY = (/4) . sum . map xOrY .  
            map (flip select vectors) $ [A,B,C,D]  
  centre   = midc (select A geom) (select C geom)  
            (Vertex3 _ _ z) = select A geom
```

# Hog vs HogCentre



```
Hog, HogCentre :: (Float,Float) -> Float ->  
                Square (Float,Float) -> Square Coord ->  
                (Coord, Coord)
```

```
Hog range scale vectors geom =  
  (v0, Vertex3 (scale* u/rngx) (scale* v/rngy) z)  
  where  
    (u,v) = select A vectors  
    v0@(Vertex3 _ _ z) = select A geom
```

```
HogCentre range scale vectors geom =  
  (centre, Vertex3 (scale* avg fst / rngx)  
                 (scale* avg snd / rngy) z)  
  where  
    avg xOrY = (/4) . sum . map xOrY .  
              map (flip select vectors) $ [A,B,C,D]  
    centre = midc (select A geom) (select C geom)  
    (Vertex3 _ _ z) = select A geom
```