Paratrooper in Haskell

Srineet Sridharan

September 4, 2002

```
> import GraphicsUtils
> import Random
> import List
```

This is the source code for the game *paratrooper* written in Haskell, along with supporting notes and comments.

1 The game

Paratrooper used to be a popular game on DOS. There was another very similar game called Ack Attack. The game consists of a gun stationed on the ground, which the player controls. There are helicopters flying above, whom the player can shoot. Paratroopers jump out of the helicopters. The objective is to not let the paratroopers land. If four paratroopers land, and get to the gun, the gun is destroyed, and the game over.

Paratrooper had another thing that Ack Attack did not, and nor does this program have. It had a fighter plane that came sometimes and fired at the gun. The player then had to shoot the bullet fired at the gun, before it destroys the gun.

2 Motivation

For quite some time, I wanted to write a small but complete program in Haskell. This is the result. Unfortunately, for this program, I can claim much neither in elegance nor in efficieny. This was written in whatever time I got off from work, and am happy that it reached completion. I added this text, and supporting comments only after the entire program was written. The text does not describe the code in a lot of detail, but hopefully makes it more readable and understandable. Hopefully, as I do more Haskell I'll write better code.

Writing the program was real fun. Everything seemed to work the first time around, and I really was hit with very few bugs as such, and no major ones at that. Quite a bit of the credit for that, perhaps, goes to Haskell.

3 Stages in the incremental development

Though what you see is the entire completed source, the program was developed incrementally. (It is really unfortunate, that whenever we see any source, its actually the completed code, and we miss the various stages in development, the mistakes made, and so forth. Of course, it does not matter much for a hurriedly written program like this one).

These are the stages in which this program was built up.

- The flying helicopters.
- The gun that can be controlled by key strokes.
- The bullets shot by the gun.
- Bullets hitting the helicopters (collision)
- Paratroopers jumping out of the helicopters.
- Being able to shoot the paratroopers.
- Score.
- The start screen, help screen, and ending the game.

4 Haskell Graph Library

For all the graphics in the game I used the Haskell Graphics Library by Alaistar Reid (http://www.haskell.org/graphics/). Its a real nice and easy library for building simple images - easy to setup and use.

Only those features in the library which work on both Microsoft Windows, and X11 are used.

I also tried using FRAN - Functional Reactive Animation, by Conal Elliott (http://research.microsoft.com/∼conal/fran/) but could not set it up on my Windows machine. This one seemed really more elegant and suited for moving pictures (or animation).

5 Building the source

If the Haskell Graphics Library is installed in the directory HGL_DIR, the command to build the source is:

```
ghc -i$(HGL_DIR)\lib\win32 -package concurrent
-package win32 --make para.lhs -o para.exe
```

I named the output as para.exe on Windows, the name an be changed to suit other tastes and platforms.

Also, it'll have to be

```
-i$(HGL\_DIR)\lib\X11
```

instead of win32, for Unix platforms.

6 Source code - overall structure

While starting a program, before thinking of the various details to be incorporated, it helps to have an idea of the overall structure. This is especially useful when the familiarity with the language/library is less.

Also, the language as well as the library used very much affects the thought, the design and the structure of a program. The Haskell Graphics Library provides a way to declaratively build images. However, the event handling, and the actual actions like drawing the pictures, getting the keystroke events have more of a imperative nature. This program, I must say, has quite a bit of imperative tinge. Though the pictures are built declaratively, out of Graphic components, the passage of time is taken into account by the getWindowTick IO action, which is essentially imperative. The keyboard events are obtained by the getWindowEvent IO action. This leads to the overall structure described next. I must of course say over here, that there definitely must be various ways to code things more declaratively, but, atleast to one without any great deal of expertise, the imperative side seems inevitable.

On the other hand, a program written using more declarative tools like FRAN would probably have a very different look.

Lets come back to the structure of this program. The entire information at any instant is captured in the data type GameState. The whole screen at any given moment can be rendered using the information in the GameState value. Also the next state of the program can be derived from the current state, and the keyboard event. The main structure of the program, therefore would look like this.

```
loop(currentState):
    draw currentState
    Get currentEvent
    loop(step currentState currentEvent)
```

Given next is the GameState data type. As can be seen, the state of the game consists of

- The flying helicopters (HeliRow)
- The gun

- The bullets
- The paratroopers (those still in the air, and those that have landed)
- paraJumpProb, a number to indicate the rate at which paratroopers will be jumping out (this rate increases with score).
- Current score
- A boolean value that tells whether the game is over yet.

```
> data GameState = GS {hrows :: [HeliRow],
>                      gun :: Gun,
>                      bullets :: [Bullet],
>                      paraAir, paraGround :: [Para],
>                      paraJumpProb :: Int,
>                      score    :: Int,
>                      gameOver :: Bool}
```

7 Useful constants

Given below are some constants frequently used in the program. winxMax and winyMax are the window extents, and the rest are the constants for colours,

Note that many numbers in this program are chosen either arbitrarily or, in some cases. after having played for a while, and deciding what gives the best *feel*.

```
> winxMax, winyMax :: Int
> winxMax = 500
> winyMax = 350

> red   = RGB 255 0 0
> blue  = RGB 0 0 255
> yellow    = RGB 255 255 0
> myTxtColor    = RGB 240 170 170
> brightTxt = RGB 255 200 200
```

8 Helicopters

```
> data Heli = HRight {hpos::Point, stamp::Int}
>           | HLeft  {hpos::Point, stamp::Int}
>           | HBangR {hpos::Point, ticks::Int}
>           | HBangL {hpos::Point, ticks::Int}

> hBangTicks = 5 :: Int
```

The data type Heli represents a helicopter. A helicopter can be flying to the right, flying to the left, flying to the right and hit by a bullet, or flying to the left and hit by a bullet. These are represented by HRight, HLeft, HBangR and HBangL respectively.

Each of those constructors contain hpos that holds the position of the helicopter.

HRight and HLeft contain an Int value called stamp. This is to tackle the problem of when a paratrooper can jump out of a helicopter. Since this has to be random, it presents a rather tricky problem. To solve this and another similar problem, a random number is generated at each tick. The solution used is that, whenever a helicopter enters the screen, it will be associated with a random number between zero and paraJumpProb (part of the game state). This number is held by the stamp field of the helicopter. At any subsequent tick, if the random number which is generated at that tick, modulo paraJumProb, has the same value as stamp for any helicopter, a paratrooper will jump out of that helicopter. It'll be seen later that the value of paraJumpProb decreases as the score increases, leading to a higher rate of paratroopers as the game progresses (making play tougher).

HBangR and HBangL contain a field called ticks, that holds how long this helicopter should be shown with BOOM! written on it, before it vanishes. When a helicopter is hit, ticks has the value hBang ticks which decreases with every tick. When a hit helicopter has ticks equal to zero, it vanishes off the screen.

```
> data HeliRow = HeliRow {helis::[Heli],
>                         newHeliRands :: Range,
>                         y :: Int}

> maxHeliPerRow :: Int
> maxHeliPerRow = 3
```

HeliRow represents a row of helicopters. newHeliRands is of type Range which is a pair of integers – the lower and the higher limits of the range. When the random number for a tick is within the range of any particular row, a new helicopter will appear in that row, unless the row already contains maxHeliPerRow helicopters. The integer y holds the y-coordinate of the row.

Given next are some constants. maxRand is the maximum value of the random number (quite arbirarilty chosen). row1y, row2y, and row3y are y-coordinates of the three rows of helicopters. rowGap is the gap between these rows. heliSpeed, heliLen, and heliHt are the speed, height and length of a helicopter respectively.

```
> maxRand, row1y, row2y, row3y, rowGap :: Int
> heliSpeed, heliLen, heliHt :: Int
> maxRand = 10000
> row1y = rowGap
> row2y = row1y + heliHt + rowGap
> row3y = row2y + heliHt + rowGap
> rowGap    = 10
> heliHt    = 30
> heliLen   = 50
> heliSpeed = 5
```

stepHeli is a function that steps the helicopters at every tick.

```
stepHeli  :: Heli -> [Heli]
```

HRight and HLeft are taken to the left/right by heliSpeed units. If the position is outside the screen, value is an empty list, otherwise its a singleton list containing the stepped Heli. For HBangR and HBangL, in addition to stepping it to the left or right, the value of ticks is decremented. If ticks is zero, an empty list is the value, leading to the helicopter vanishing from screen.

```
> stepHeli  :: Heli -> [Heli]
> stepHeli h@(HRight {hpos=(x, y)})  = let
>   newx = x + heliSpeed
>   in if newx + heliLen > winxMax
>           then []
>           else [h {hpos=(newx, y)}]
>
> stepHeli h@(HLeft {hpos=(x, y)})  = let
>   newx = x - heliSpeed
>   in if newx < 0
>       then []
>       else [h {hpos=(newx, y)}]
>
> stepHeli (HBangR p 0) = []
> stepHeli (HBangR p n) =
>   case stepHeli (HRight {hpos=p, stamp = 0}) of
>       [] -> []
>       [HRight {hpos=p}] -> [HBangR {hpos=p, ticks=(n-1)}]
>
> stepHeli (HBangL {hpos=p, ticks=0}) = []
> stepHeli (HBangL {hpos=p, ticks=n}) =
>   case stepHeli (HLeft {hpos=p, stamp = 0}) of
>       [] -> []
>       [HLeft {hpos=p}] -> [HBangL {hpos=p, ticks=(n-1)}]
```

stepHeliRow steps an entire row of helicopters.

```
stepHeliRow   :: RandNum -> Int -> HeliRow -> HeliRow
```

The first parameter, a randomly generated integer, is used to decide whether an new helicopter should enter this row. A helicopter will enter a row, only if the random number is in the row's range and if the row is not already full with maxHeliPerRow helicopters. The direction of the new helicopter, is determined by the helicopters already in the row, if any. If there are no helicopters in the row, the direction depends upon whether the random number is even or odd. A new helicopters also gets a stamp (to decide when a paratrooper will jump out of it).

stepHeliRow uses stepHeli to step the helicopters in that row.

```
> stepHeliRow   :: RandNum -> Int -> HeliRow -> HeliRow
> stepHeliRow rand jprob hr = let
>   hs' = (concat.map stepHeli) (helis hr)
>   rg  = newHeliRands hr
>   stamp   = rand `mod` jprob
>   hs''    =
>       if inRange rg rand then
>           case hs'    of
>           []  -> if(even rand)
>                  then
>                   [HRight {hpos=(0, y hr), stamp  = stamp} ]
>                  else
>                   [HLeft{hpos=(winxMax-heliLen, y hr),
>                          stamp = stamp}]
>           hs  -> if length hs' < maxHeliPerRow then
>                     case (head hs') of
>                     HRight {hpos=(x',_)}    ->
>                          if x' > heliLen
>                             then (HRight {hpos=(0, y hr),
>                                           stamp = stamp}):hs'
>                             else hs'
>                     HBangR {hpos=(x',_), ticks = n}  ->
>                          if x' > heliLen
>                             then (HRight {hpos=(0, y hr),
>                                           stamp = stamp}):hs'
>                             else hs'
>                       HLeft {hpos=(x',_)}     ->
>                            if x' < winxMax - 2 * heliLen
>                                then
>                                 (HLeft
>                                   {hpos=(winxMax-heliLen, y hr),
>                                    stamp = stamp}):hs'
>                                else hs'
>                       HBangL {hpos=(x',_), ticks=n}  ->
>                            if x' < winxMax - 2 * heliLen
>                                then
>                                 (HLeft
>                                  {hpos=(winxMax-heliLen, y hr),
>                                   stamp = stamp}):hs'
>                                else hs'
>                        else hs'
>           else hs'
>   in
>   hr {helis = hs''}
```

The constants given next, are only some coordinates for the helicopter picture and can be ignored.

```
> hOvalTop   = 8

> hTailWidth    = 5
> hDiskHt   = 3

> hO1, hO2  :: Point
> hO1 = (0, hOvalTop)
> hO2 = (heliLen, heliHt)

> hD1Left, hD2Left, hL1Left, hL2Left    :: Point
> hT1Left, hT2Left, hT3Left, hT4Left    :: Point
> hD1Left   = (0, 0)
> hD2Left   = (heliLen - hTailWidth - 5, 0)
> hL1Left   = (fst hD2Left `div` 2, 0)
> hL2Left   = (fst hL1Left, hOvalTop)
> hT1Left   = (heliLen, (heliHt - hOvalTop) `div` 2 + hOvalTop)
> hT2Left   = (heliLen, 0)
> hT3Left   = (heliLen - hTailWidth, 0)
> hT4Left   = (heliLen - hTailWidth,
>                (heliHt - hOvalTop) `div` 2 + hOvalTop)
> hD1Right, hD2Right, hL1Right, hL2Right    :: Point
> hT1Right, hT2Right, hT3Right, hT4Right    :: Point
> hD1Right   = (heliLen, 0)
> hD2Right   = (hTailWidth + 5, 0)
> hL1Right   =
>   ((fst hD1Right  - (fst hD2Right)) `div` 2
>          + (fst hD2Right),
>                         0)
> hL2Right   = (fst hL1Right, hOvalTop)
> hT1Right   = (0, (heliHt - hOvalTop) `div` 2 + hOvalTop)
> hT2Right   = (0, 0)
> hT3Right   = (hTailWidth, 0)
> hT4Right   = (hTailWidth,
>               (heliHt - hOvalTop) `div` 2 + hOvalTop)
```

hBangPic and heliPic produce the Graphic – the representation of the image, for helicopters. Here can be seen the use of the functions from the Haskell Graphics Library.

```
> hBangPic  :: Point -> Graphic
> hBangPic (x, y)    = text (x + 5, y + 5) "BOOM!"

> heliPic   :: Heli -> Graphic
> heliPic (HLeft {hpos=p})   =  let
>   oval    = ellipse (translate p hO1) (translate p hO2)
>   disk    = polygon [ translate p hD1Left,
>                       translate p hD2Left,
>                       translate p
>                           (fst hD2Left, snd hD2Left + hDiskHt),
>                       translate p
>                           (fst hD1Left, snd hD1Left + hDiskHt)
>                     ]
>   tail    = polygon [ translate p hT1Left,
>                       translate p hT2Left,
>                       translate p hT3Left,
>                       translate p hT4Left
>                     ]
>   stick    = line (translate p hL1Left) (translate p hL2Left)
>   hPic blueb redb = withBrush redb oval `overGraphic`
>                       withBrush blueb (disk `overGraphic`
>                                          stick `overGraphic`
>                                          tail)
>
>   in
>   mkBrush red $
>       \redb ->mkBrush blue (\blueb -> hPic blueb redb)

> heliPic (HRight {hpos=p})   =  let
>   oval    = ellipse (translate p hO1) (translate p hO2)
>   disk    = polygon [ translate p hD1Right,
>                       translate p hD2Right,
>                       translate p
>                        (fst hD2Right, snd hD2Right + hDiskHt),
>                       translate p
>                        (fst hD1Right, snd hD1Right + hDiskHt)
>                     ]
>   tail    = polygon [ translate p hT1Right,
>                       translate p hT2Right,
>                       translate p hT3Right,
>                       translate p hT4Right
>                     ]
>   stick    = line (translate p hL1Right) (translate p hL2Right)
>   hPic blueb redb = withBrush redb oval `overGraphic`
>                      withBrush blueb
>                       (disk `overGraphic`
>                        stick `overGraphic`
>                        tail)
>   in
>   mkBrush red  $
>       \redb -> mkBrush blue (\blueb -> hPic blueb redb)

> heliPic (HBangR {hpos=p})  =
>   hBangPic p `overGraphic` heliPic (HRight {hpos=p, stamp = 0})
> heliPic (HBangL {hpos=p})  =
>   hBangPic p `overGraphic` heliPic (HLeft {hpos=p, stamp = 0})

> hrowPic   :: HeliRow -> Graphic
> hrowPic hr   = foldr overGraphic emptyGraphic
>                           (map heliPic (helis hr))
```

9 The Gun

The only thing significant about the gun at any point in time is the orientation of its barrel, and the direction in which the barrel is moving. This is represented by the gunAngle field (in radians), and the gdir field.

```
> data Gun  = Gun {gunAngle :: Float,
>                  gdir     :: GunDir}

> data GunDir   = GLeft
>               | GRight
>               | GNone
```

gunAngleStep is by angle by which it turns to the left (or right) when the left (right) key is pressed. gunBaseWidth and gunBaseWidth, are used for drawing the gun. gunLen is the length of the gun barrel.

stepGun is the function that keeps the gun moving to the left, or the right, or just keeps it stationary depending upon the direction in its gdir field.

```
> gunAngleStep  :: Float
> gunAngleStep  = 4 * pi / 180      -- 5 degrees

> stepGun g = let
>   ga  = gunAngle g
>   in
>   case gdir g of
>      GNone   -> g
>      GLeft   -> if ga < pi
>                   then g {gunAngle = ga + gunAngleStep}
>                   else g
>      GRight  -> if ga > 0
>                   then g {gunAngle = ga - gunAngleStep}
>                   else g


> gunBaseHt  = 30   :: Int
> gunLen    = 30    :: Int
> gunBaseWidth  = 40    :: Int

> gunBarrelBase :: Point
> gunBarrelBase = (winxMax `div` 2, winyMax - gunBaseHt + 5)
```

gunPic is the function that gives the picture for the gun. gunBasePic draws the rectangular base of the gun.

```
> gunBasePic    :: Graphic
> gunBasePic    = let
>   halfBaseLen = gunBaseWidth `div` 2
>   cx  = winxMax `div` 2
>   cy  = winyMax
>   p1  = (cx - halfBaseLen, cy)
>   p2  = (cx + halfBaseLen, cy)
>   p3  = (cx + halfBaseLen, cy - gunBaseHt)
>   p4  = (cx - halfBaseLen, cy - gunBaseHt)
>   in
>   mkBrush red $
>      \redb -> withBrush redb (polygon [p1, p2, p3, p4])

> gunPic    :: Gun -> Graphic
> gunPic (Gun {gunAngle = ga})  = let
>   fLen    = (fromIntegral gunLen) :: Float
>   fp1x    = ((fromIntegral.fst) gunBarrelBase) :: Float
>   fp1y    = ((fromIntegral.snd) gunBarrelBase) :: Float
>   fp2x    = round (fp1x + fLen * (cos ga))
>   fp2y    = round (fp1y - fLen * (sin ga))
>   barrel  = thickLine 6 gunBarrelBase (fp2x, fp2y)
>   in barrel ‘overGraphic‘ gunBasePic
```

10 Bullets

A bullet is represented by its current position (bx and by); and its horizontal and vertical velocity (vx and vy), which is determined by the position of the gun when the bullet was shot, and does not change thereafter.

bulletSpeed is the speed of the bullet in pixels.

```
> data Bullet   = Bullet {bx, by, vx, vy :: Float}
> bulletSpeed   = 20 :: Float
```

stepBullet is the function to step a single bullet, and to determine whether the bullet has gone past the window boundary. stepBullets uses stepBullet to step a list of bullets.

```
> stepBullet    :: Bullet -> [Bullet]
> stepBullet bu = let
>   bu’ = bu {bx = bx bu + bulletSpeed * (vx bu),
>             by = by bu - bulletSpeed * (vy bu)}
>   in
>   if inRect (0, 0)
>             (winxMax, winyMax)
>             (round (bx bu’), round (by bu’))
>       then [bu’] else []
> stepBullets   :: [Bullet] -> [Bullet]
> stepBullets bus   = (concat.map stepBullet) bus
```

bulletPic gives the image for the bullet, and bulletsPic uses bulletPic to draw a list of bullets.

```
> bulletPic :: Bullet -> Graphic
> bulletPic bu  =  dot (round (bx bu), round (by bu))

> bulletsPic    :: [Bullet] -> Graphic
> bulletsPic bus    =
>   mkPen Solid 1 yellow $
>       \yp-> withPen yp $
>       foldr overGraphic emptyGraphic (map bulletPic (bus))
```

11 The Paratroopers

A paratrooper is represented by the data type Para.

```
> data Para = ParaFull {pos :: Point}
>           | ParaHalf {pos :: Point}
```

ParaFull is a paratrooper with his parachute, and paraHalf is a paratrooper whose parachute has been shot.

```
> paraHt, chuteHt, paraWidth, paraSpeed :: Int
> paraHt    = 25
> chuteHt   = 10
> paraWidth = 12
> paraSpeed = 2
```

paraHt and chuteHt are the heights of the paratrooper and the parachute. paraWidth is the width and paraSpeed is the speed (in pixels per tick).

stepPara is the function the steps the state of a paratrooper. The type

```
stepPara  :: ([Para], [Para]) -> ([Para], [Para])
```

is really

```
stepPara  :: (paraAir, paraGround) -> (newParaAir, newParaGround)
```

For paraAir (the list of paratroopers in the air), each paratrooper will descend by paraSpeed pixels. On reaching the ground, the paratrooper becomes part of paraGround, and is a ParaHalf (with no chute). The paratroopers already on the ground, will move towards the gun with speed paraSpeed.

Note the function separateParaGround. When paratroopers on the ground move towards the gun, eventually they might end up at the same position, and will appear as one due to the superimposed images. This might also happen if a paratrooper lands at a position on a ground where there already is another paratrooper. This is taken care of, by the function separateParaGround. It separates those paratroopers that are pretty much at the same position so that they appear as two (or more as the case may be). iterateUntilConverge is used in separateParaGround. After adjusting a ground paratrooper's position, he may still be in the same position as some other paratrooper with whom his position was not clashing before. iterateUntilConverge makes sure that all paratroopers have distinct positions. The adjust function, separates the paratroopers at the same position to the left of the gun, by shifting the position of some of them more to the left. While those paratroopers to the right of the gun, are separated by moving the clashing ones a bit more to the right.

```
> stepPara  :: ([Para], [Para]) -> ([Para], [Para])
> stepPara (pAir, pGround)  = let
>   fooAir pGround para = let
>       (x, y)  = pos para
>       in
>       if y < winyMax - paraHt
>           then (pGround, [para {pos = (x, y + paraSpeed)}])
>           else case para of
>               ParaFull {} ->
>                       (ParaHalf
>                        {pos=(x, winyMax-paraHt)}:pGround, [])
>               ParaHalf {} -> (pGround, [])
>
>   fooGnd  para    = let
>       gunBaseLeftx    = (winxMax - gunBaseWidth) ‘div‘ 2
>       gunBaseRightx   = (winxMax + gunBaseWidth) ‘div‘ 2
>       (x, y)  = pos para
>       in
>       if x < gunBaseLeftx - paraWidth
>           then para {pos=(x+paraSpeed, y)}
>           else if x > gunBaseRightx
>              then para {pos = (x - paraSpeed, y)}
>                     else para
>
>
>   pGround'    = map fooGnd pGround
>   (pGround'', pAir'') = mapAccumR fooAir pGround' pAir
>   in
>   (concat pAir'', separateParaGround pGround'')

> separateParaGround    :: [Para] -> [Para]
> separateParaGround [] = []
> separateParaGround pGround    = let
>   centrex    = winxMax `div` 2
>
>   cmpParaPos  :: Para -> Para -> Ordering
>   cmpParaPos  para1 para2 = compare ((fst.pos) para1)
>                                     ((fst.pos) para2)
>   isEq para1 para2    = abs
>                         ((fst.pos) para1 - ((fst.pos) para2))
>                         < paraSpeed
>   isEqs paras1 paras2 = and (zipWith isEq paras1 paras2)
>
>   fooConverge :: [Para] -> [Para]
>   fooConverge pGnd    = let
>       pGnd'   = {-sortBy cmpParaPos-} pGnd
>
>       seppGnd'= separate isEq pGnd'
>
>       adjust  :: [Para] -> [Para]
>       adjust paras    = let
>
>           zfoo para n = let
>               (x, y)  = pos para
>               x1' = x - n * paraSpeed
>               x2' = x + n * paraSpeed
>               in
>               if x <= centrex && x1' > 0
>                   then para {pos = (x1', y)}
>                   else if x >= centrex &&
>                           x2' < winxMax - paraWidth
>                           then para {pos = (x2', y)}
>                           else para
>
>               in zipWith zfoo paras [0..]
>       in (concat.map adjust) seppGnd'
>
>   in (iterateTillConverge isEqs fooConverge pGround)
```

paraNewJump is a function that gives a (possibly empty) list of new paratroopers jumped out of helicopters. The policy used to determine when a paratrooper should jump out was discussed earlier. Basically, the random number for the current tick, modulo jprob - the number that controls the likelihood of a paratrooper jumping out, should be equal to the stamp which was randomly associated with the helicopter (when it first appeared), for a paratrooper to jump out. As can be seen, the smaller the value of jprob, the more likely that a paratrooper would jump out. The value of the stamp of a helicopter is in the interval 0 to jprob. The number

```
rand ‘mod‘ jprob
```

will lie in the same interval. Its more likely that the numbers are equal, if the interval is smaller. The value paraJumpProb (part of the GameState) is reduced as the score increases, in order to have more paratroopers jumping out and making the game harder to play.

In paraNewJump the new paratrooper who has jumped out has the position just below the helicopter and at the centre of it.

```
> paraNewJump   :: [HeliRow] -> RandNum -> Int -> [Para]
> paraNewJump hrows rand jprob  = let
>   fooRow  :: HeliRow -> [Para]
>   fooRow hr   = (concat.map fooHeli) (helis hr)
>
>   nowStamp    = rand ‘mod‘ jprob
>
>   fooHeli :: Heli -> [Para]
>   fooHeli (HBangR {}) = []
>   fooHeli (HBangL {}) = []
>   fooHeli heli | stamp heli == nowStamp    =
>       [ParaFull {pos = translate (hpos heli)
>                                 (heliLen ‘div‘ 2, heliHt)}]
>                | otherwise = []
>   in
>   (concat.map fooRow) hrows
```

scoreJumpProb is made up of tuples whose first component is the score at which paraJumpProb changes to the value in the second component. As you can see, the value of paraJumpProb decreases with the score, the minimum being 30 at score 1300, when the game becomes toughest to play. Change this to change the rate at which paratroopers fall off.

```
>{-
> scoreJumpProb :: [(Int, Int)]
> scoreJumpProb = [(1300, 30),
>                  (900, 50),
>                  (400, 60),
>                  (200, 75),
>                  (100, 100)]
>-}

> scoreJumpProb :: [(Int, Int)]
> scoreJumpProb = [
>                  (2000, 50),
>                  (1600, 80),
>                  (1300, 90),
>                  (1000, 100),
>                  (800,  125),
>                  (500, 150),
>                  (100, 250)]
```

chutePic, trooperPic, and paraPic draw the parachute, the trooper, and the whole paratrooper (possible with the chute) respectively.

```
> chutePic  :: Point -> Graphic
> chutePic p   = let
>   halfChuteHt = chuteHt `div` 2
>   p1  = translate p (0, 0)
>   p2  = translate p (paraWidth, halfChuteHt)
>   p3  = translate p (0, halfChuteHt `div` 2)
>   p4  = translate p (paraWidth, halfChuteHt `div` 2)
>   p5  = translate p (paraWidth `div` 2, chuteHt)
>   in mkBrush blue $ \blueb -> withBrush blueb $
>       mkBrush red $ \redb -> withBrush redb (ellipse p1 p2)
>       `overGraphic` (line p3 p5 `overGraphic` (line p4 p5))

> trooperPic    :: Point -> Graphic
> trooperPic p  = let
>   ht  = paraHt - chuteHt
>   h1  = ht `div` 3
>   p1  = translate p (0, 0)
>   p2  = translate p (paraWidth, h1)
>   p3  = translate p (0, h1)
>   p4  = translate p (paraWidth, ht)
>   in
>   mkBrush blue $ \blueb -> withBrush blueb $
>       (ellipse p1 p2 `overGraphic` ellipse p3 p4)

> paraPic   :: Para -> Graphic
> paraPic (ParaFull {pos = p@(x, y)})   =
>     chutePic p `overGraphic` trooperPic (x, y + chuteHt)
> paraPic (ParaHalf {pos = p@(x, y)})   =
>   trooperPic (x, y + chuteHt)
```

12 Shooting the helicopters down

This section contains code to detect when a bullet has hit a helicopter.
The top level function is

```
 hRowColl  :: ([Bullet], Int) -> [HeliRow] ->
                                  (([Bullet], Int), [HeliRow])
```

This function uses mapAccum and hRowColl' on its list of helicopter rows.

```
 hRowColl'  :: ([Bullet], Int) -> HeliRow ->
                                (([Bullet], Int), HeliRow)
```

Just like hRowColl acts on a list of helicopter rows, and hRowColl' on a single helicopter row, heliColl acts on a list of helicopters (in a row), and heliColl' on a single helicopter.

```
heliColl  :: ([Bullet], Int) ->[Heli] ->
                              (([Bullet], Int), [Heli])

hRowColl'  :: ([Bullet], Int) -> HeliRow ->
                                (([Bullet], Int), HeliRow)
```

The thing to note here is the use of mapAccum where the accumulator is the list of bullets, and the score. If a bullet has hit a helicopter, it should be removed from the list of bullets. When a helicopter is hit, the score increases. This is how the list of bullets and the score, changes per helicopter and per helicopter row, and hence is used as an accumulator.

Collision detection is done mainly by using the function hitObject.

```
hitObject :: Point -> Point -> Point -> Int -> Bool
```

The first two points in the are the end points of the boundary box of the object thats to be hit. The third point is the position of the bullet. The Int is the speed of the bullet. Since the bullet moves with finite speed and in discrete time ticks, the bullet steps through discrete points (and not in a conitnuous path), we cannot say that the collision has happened only if the bullet is in the interior of the boundary box. If the boundary box is of smaller dimensions than the bullet speed, we might entirely miss detecting the collision. So we use speed as a parameter, and say that the object has collided, even if the bullet is at a distance of (speed / 2) from the objects boundary box.

This way we will definitely detect collisions (and in a very few cases, a collision would be detected even though the bullet would not have collided if time had been continuous, but its an error in the direction of better score, hence making the player happy).

When a HLeft (or HRight) is hit by a bullet, it becomes a HBangL (or HBangR). The value of ticks, for the HBang, which decides how long the hit helicopter will stay on screen, is set to hBangTicks.

```
> heliColl' :: ([Bullet], Int) -> Heli ->
>                           (([Bullet], Int),  Heli)
> heliColl' (bus, score) h   = let
>
>   doColl  :: Point -> [Bullet] -> (Bool, [Bullet])
>   doColl p bus    = let
>       bulletColl bu = hitObject p
>                              (fst p + heliLen, snd p + heliHt)
>                              (round (bx bu), round (by bu))
>                              (round bulletSpeed)
>       colls   = map bulletColl bus
>       bus'    = [b | (b, False) <- zip bus colls]
>       in (or colls, bus')
>
>   (hit, bus') = doColl (hpos h) bus
>   score'  = if hit then score + heliHitScore else score
>
>   in case h of
>       HRight {hpos=p}    -> if hit then ((bus', score'),
>                                           HBangR p hBangTicks)
>                                     else ((bus', score'), h)
>       HLeft {hpos=p}     -> if hit then ((bus', score'),
>                                           HBangL p hBangTicks)
>                                     else ((bus', score'), h)
>       HBangR {hpos=p}    -> ((bus', score), h)
>       HBangL {hpos=p}    -> ((bus', score), h)

> heliColl  :: ([Bullet], Int) ->[Heli] ->
>                               (([Bullet], Int), [Heli])
> heliColl (bus, score) hs   =
>   mapAccumR heliColl' (bus, score) hs

> hRowColl'  :: ([Bullet], Int) -> HeliRow ->
>                                 (([Bullet], Int), HeliRow)
> hRowColl' bus hr  = let
>   ((bus', score'), helis')  = heliColl bus (helis hr)
>   in ((bus', score'), hr {helis = helis'})
> hRowColl  :: ([Bullet], Int) -> [HeliRow] ->
>                                 (([Bullet], Int), [HeliRow])
> hRowColl (bus, score) hrs  =
>   mapAccumR hRowColl' (bus, score) hrs
```

13 Shooting the paratroopers

paraColl and paraColl' are used to detect when a bullet hits a paratrooper. They are quite similar to helicopter collision functions above (notice the user of mapAccum in paraColl).

There's a slight difference that, a paraFull, when the chute is hit by a bullet becomes a paraHalf; and a paraHalf when hit by the bullet vanishes. So the function needs to look for two kinds of hit – whether the chute is hit, or the paratrooper is hit.

Similar to heliColl above, the bullets and the score are the accumulated values.

```
> paraColl  :: ([Bullet], Int) -> [Para] ->
>                       (([Bullet], Int), [Para])
> paraColl (bus, score) paras    = let
>   ((bus', score'), mparas)  =
>       mapAccumR paraColl' (bus, score) paras
>   paras'  = [p | (Just p) <- mparas]
>   in ((bus', score'), paras')

> paraColl' :: ([Bullet], Int) -> Para ->
>               (([Bullet], Int), Maybe Para)
> paraColl' (bus, score) para    = let
>   (x, y)  = pos para
>
>   isTrooperHit bu = hitObject (x, y + chuteHt)
>                               (x + paraWidth, y + paraHt)
>                               (round (bx bu), round (by bu))
>                               (round bulletSpeed)
>
>   isChuteHit bu   = hitObject (x, y)
>                               (x + paraWidth, y + paraHt)
>                               (round (bx bu), round (by bu))
>                               (round bulletSpeed)
>
>   colls1  = map isTrooperHit bus
>   colls2  = map isChuteHit bus
>   in
>   case para of
>       ParaFull {} -> let
>           bus'    =
>               [b | (b, False) <- zip bus
>                                   (zipWith (||) colls1 colls2)]
>           in
>           if or colls1 then
>                       ((bus', score + trooperHitScore),
>                          Nothing)
>                       else if or colls2
>                              then
>                               ((bus', score + chuteHitScore),
>                                Just (ParaHalf {pos=pos para}))
>                              else ((bus', score), Just para)
>       ParaHalf {} -> let
>           bus'    = [b | (b, False) <- zip bus colls1]
>           in
>           if or colls1 then ((bus', score + trooperHitScore),
>                              Nothing)
>                       else ((bus', score), Just para)
```

14 Scoring

The constants and functions related to scoring are quite easy to read.

Every pGndClearScore points, the landed paratroopers will vanish.

scoreEffects changes the paraJumpProb in the GameState depending on the score, and also is resposible to have the landed paratroopers vanish, if its time for them to (every pGndClearScore points). scoreJumpProb (already given above) is the list which has the values of paraJumpProbs for different scores.

```
> chuteHitScore, trooperHitScore, heliHitScore,
>   bulletShootScore, pGndClearScore    :: Int
> chuteHitScore     = 8
> trooperHitScore   = 15
> heliHitScore      = 20
> bulletShootScore  = 4
> pGndClearScore    = 200

> scoreEffects  :: Int -> GameState -> GameState
> scoreEffects oldScore gs   = let
>   pGnd'   = if oldScore > 0 &&
>                score gs > 0 &&
>               oldScore < score gs
>                          - (score gs `mod` pGndClearScore)
>                  then [] else paraGround gs
>   jProb'  = case filter
>                  (\(s, p) -> s <= score gs)
>                  scoreJumpProb of
>               []      -> paraJumpProb gs
>               ((_, p): xs) -> p
>   in gs { paraGround      = pGnd',
>           paraJumpProb    = jProb'}
```

scorePic is for display of the current score.

```
> scorePic  :: Int -> Graphic
> scorePic score    =
>   withTextColor brightTxt $ text (winxMax - 75, 2) (show score)
```

15 Game over?

isGameOver tells us whether the game has ended. The game ends when atleast 4 paratroopers land and come "near" the gun.

```
> isGameOver :: GameState -> Bool
> isGameOver gs = let
>   pGnds   = paraGround gs
>
>   nearGun para    = let
>       (x, _)  = pos para
>       cx  = winxMax `div` 2
>       hg  = gunBaseWidth `div` 2
>       in
>       isBetween (cx - hg - paraWidth - 4 * paraSpeed)
>                 (cx + hg + paraWidth + 4 * paraSpeed)
>                 x
>
>   in (length. filter nearGun) pGnds >= 4
```

16 External events (key presses)

processEvents takes care of handling key presses. The arrow keys set the current direction of the gun's movement and also move it in that direction. This way, if you keep pressing the left key for instance, the gun will move faster than if you had just pressed the left key once, and had let the gun move on its own accord. This way to accelerate the gun's movement by pressing the arrow key, gives a nice feel to the game.

Note how the horizontal and vertical components of the bullet's speed are determined by the orientation of the gun.

```
> processEvent  :: Maybe Event -> GameState -> GameState
> processEvent me gs = let
>   gun' = gun gs
>   ga  = gunAngle gun'
>   in
>   case me of
>       Just (Key {keysym = k, isDown = True}) ->
>           if isRightKey k then
>               if ga > 0 then
>                   gs {gun = gun'{gunAngle = ga - gunAngleStep,
>                                  gdir = GRight}}
>                   else gs {gun = gun'{gdir = GRight}}
>           else if isLeftKey k then
>               if ga < pi then
>                   gs {gun = gun'{gunAngle = ga + gunAngleStep,
>                                  gdir = GLeft}}
>                   else gs {gun = gun'{gdir = GLeft}}
>           else if isUpKey k then let
>               c   = cos ga
>               s   = sin ga
>               fGunLen = fromIntegral gunLen   :: Float
>               bu  = Bullet
>                      { bx = fromIntegral (fst gunBarrelBase)
>                               + fGunLen * c,
>                        by = fromIntegral (snd gunBarrelBase)
>                               - fGunLen * s,
>                        vx   = c,
>                        vy   = s}
>            in gs {bullets = bu:(bullets gs),
>                   score   = score gs - bulletShootScore,
>                   gun     = gun'{gdir = GNone}}
>           else  if isDownKey k then
>               gs {gun = gun'{gdir = GNone}}
>           else gs
>       _ -> gs
```

17 State of the game

The state of the game is the world, as far as the program is concerned. initState initializes the state. Only point of note, is the ranges for the three helicopter rows, which determine how often will helicopters randomly enter a row.

```
> initState :: GameState
> initState = let
>   hrs = [HeliRow {helis=[],
>                   newHeliRands = (0, maxRand ‘div‘ 78),
>                   y = row1y},
>           HeliRow {helis = [],
>                   newHeliRands = (maxRand ‘div‘ 78,
>                   (2 * maxRand) ‘div‘ 96),
>                   y = row2y},
>           HeliRow {helis = [],
>                   newHeliRands = ((2 * maxRand) ‘div‘ 78,
>                                   (3 * maxRand) ‘div‘ 78),
>                   y = row3y}
>           ]
>   gun = Gun {gunAngle = pi / 2, gdir = GNone}
>   paraAir = []
>   paraGround = []
>   in
>   GS {hrows = hrs, gun = gun,
>       bullets = [],
>       paraAir = paraAir,
>       paraGround = paraGround,
>       paraJumpProb = 300,
>       gameOver = False,
>       score   = 0}
```

step is the real worker function that makes the game move forward every tick. Things are evaluated in the following order.

- paratroopers move forward and new ones jump out. The helicopters move forward, and new ones may enter. The bullets carry on in their path. The gun is moved in the appropriate direction.
- Check is made for whether any bullet has hit any helicopter or paratroopers.
- Check is made to see whether the game is over.
- Keys pressed are taken care of.

- The effects of the updated score materialize upon the GameState.

and the game has moved forward one tick!

```
> step  :: RandNum -> Maybe Event -> GameState -> GameState
> step rand me gs   = let
>   oldScore    = score gs
>   (pAir1, pGnd1)  = stepPara (paraAir gs, paraGround gs)
>   pAir2   = paraNewJump (hrows gs) rand (paraJumpProb gs)
>               ++ pAir1
>   hrows1  = map (stepHeliRow rand (paraJumpProb gs)) (hrows gs)
>   bullets1    = stepBullets (bullets gs)
>   ((bullets2, score1), pAir3) =
>       paraColl (bullets1, score gs) pAir2
>   ((bullets3, score2), pGnd2)  =
>       paraColl(bullets2, score1) pGnd1
>   ((bullets4, score3), hrows2)    =
>       hRowColl (bullets3, score2) hrows1
>   gun'    = stepGun (gun gs)
>   gs1 = gs { hrows    = hrows2,
>              bullets  = bullets4,
>              paraAir  = pAir3,
>              paraGround   = pGnd2,
>              score    = score3,
>              gun      = gun',
>               }
>   gs2 = gs1 {gameOver = isGameOver gs1}
>   gs3 = processEvent me gs2
>   gs4 = scoreEffects oldScore gs3
>   in
>   gs4
```

statePic gives the image for the entire window for a particular tick. The function is straightforward – the image is derived from the state.

```
> statePic  :: GameState -> Graphic
> statePic gs   = let
>   cx  = winxMax `div` 2
>   cy  = winyMax `div` 2
>   gameEndPic | gameOver gs  = withTextColor myTxtColor $
>                                text (cx - 50, cy - 10)
>                                  "G A M E  O V E R!"
>                                      `overGraphic`
>                                hBangPic (cx - 30,
>                                        winyMax - gunBaseHt - 5)
>              | otherwise  = emptyGraphic
>   in
>   gameEndPic `overGraphic`
>   scorePic (score gs) `overGraphic`
>   foldr overGraphic emptyGraphic (map paraPic (paraGround gs))
>       `overGraphic`
>   foldr overGraphic emptyGraphic (map paraPic (paraAir gs))
>       `overGraphic`
>   bulletsPic (bullets gs)
>       `overGraphic`
>   foldr overGraphic emptyGraphic (map hrowPic (hrows gs))
>       `overGraphic`
>   gunPic (gun gs)
```

18 The other screens

startPic and helpPic just show the screens that appear at the start.

```
> startPic  :: Graphic
> startPic  = let
>   cx  = winxMax `div` 2
>   cy  = winyMax `div` 2
>   in
>   withTextColor myTxtColor $
>   text (cx - 70, cy - 70) "P A R A T R O O P E R" `overGraphic`
>   text (cx - 70, cy - 20) "by Srineet Sridharan"  `overGraphic`
>   text (cx - 45, cy + 0) " August 2002"  `overGraphic`
>   text (cx - 130, cy + 70)
>       "Implemented in Haskell: www.haskell.org"

> helpPic :: Graphic
> helpPic = withTextColor myTxtColor $
>   text (winxMax `div` 2 - 30, 15) "H E L P" `overGraphic`
>   text (10, 70) "Keys:"    `overGraphic`
>   text (10, 100) "Left: Left key" `overGraphic`
>   text (10, 120) "Right: Right key" `overGraphic`
>   text (10, 140) "Shoot: Up key" `overGraphic`
>   text (10, 160) "Fix gun position: Down key" `overGraphic`
>   text (10, 180) "Quit: Escape key"  `overGraphic`
>   withTextColor brightTxt (text (10, 210)
>       "Four troopers land, and its all over!") `overGraphic`
>   text (10, 240) ("Every " ++ (show pGndClearScore) ++
>                   " points, the landed troopers vanish")
```

19 main

The main function gets it all running. Most of it is simple – just a sequence of actions. Notics the use of maybeGetWindowEvent to get the keys, the use of setGraphic to draw the picture, and the use of the step function over the state, in a loop.

```
> main = runGraphics (do
>   w <- openWindowEx "Paratrooper"
>          Nothing
>          (winxMax+1,winyMax+1)
>          DoubleBuffered
>          (Just 50)
>   setGraphic w startPic
>   getKey w
>   setGraphic w helpPic
>   getKey w
>   let s   = initState
>   setGraphic w (statePic s)
>   let loop st  = do
>       getWindowTick w
>       setGraphic w (statePic st)
>       if not (gameOver st) then do
>           rand <- getStdRandom (randomR (1, maxRand))
>           e <- maybeGetWindowEvent w
>           let st’ = step rand e st
>           case e of
>               Just (Key {keysym = k, isDown = True}) ->
>                   if isEscapeKey k then return ()
>                                    else loop st’
>               _   -> loop st’
>        else return ()
>   loop s
>   eatEvents w
>   waitForKey w isEscapeKey
>   let cx = winxMax ‘div‘ 2
>   let cy = winyMax ‘div‘ 2
>   setGraphic w $ withTextColor myTxtColor $
>       text (cx - 40, cy - 10) "T H A N K S !"
>   eatEvents w
>   waitForKey w isEscapeKey
>   clearWindow w
>   closeWindow w)
```

20 Miscellaneous Functions

These are some useful functions used in the program.

iterateUntilConverge applies a function again and again, to a value until the value does not change. The function for checking the equality is provided as the first argument.

```
> iterateTillConverge   :: (a -> a -> Bool) -> (a -> a) -> a -> a
> iterateTillConverge isEq foo x    = let
>   x'  = foo x
>   in
>   if x' `isEq` x then x' else iterateTillConverge isEq foo x'
```

separate converts a list into a list of lists, such that each member list in the returned value contains elements that are equal to each other, as determined by the equality checker supplied as the first argument.

```
> separate :: (a -> a -> Bool) -> [a] -> [[a]]
> separate _ []    = [[]]
>
> separate isEq xs@(x:_) = let
>   (ys, zs)    = span (isEq x) xs
>   in ys:(separate isEq zs)
```

thickLine draws a line of given thickness (first argument), between two points.

```
> thickLine :: Int -> Point -> Point -> Graphic
> thickLine thickness (x1, y1) (x2, y2) = let
>   ft  = (fromIntegral thickness) :: Float
>   halfFt  = ft / 2
>   fx1 = (fromIntegral x1) :: Float
>   fy1 = (fromIntegral y1) :: Float
>   fx2 = (fromIntegral x2) :: Float
>   fy2 = (fromIntegral y2) :: Float
>   theta   = atan2 (fy2 - fy1) (fx2 - fx1)
>   theta'  = pi / 2 + theta
>   p1, p2, p3, p4  :: Point
>   p1  = (round (fx1 + halfFt * (cos theta')),
>          round (fy1 + halfFt * (sin theta')))
>   p2  = (round (fx1 - halfFt * (cos theta')),
>          round (fy1 - halfFt * (sin theta')))
>   p3  = (round (fx2 - halfFt * (cos theta')),
>          round (fy2 - halfFt * (sin theta')))
>   p4  = (round (fx2 + halfFt * (cos theta')),
>          round (fy2 + halfFt * (sin theta')))
>   in polygon [p1, p2, p3, p4]

> dot   :: Point -> Graphic
> dot p = line p (fst p + 1, snd p + 1)
```

isBetween checks whether a value (third argument), is between two values (first and second arguments).

```
> isBetween :: Ord a => a -> a -> a -> Bool
> isBetween a b x  | a > b = x >= b && x <= a
>                  | b > a = x >= a && x <= b
>                  | otherwise = x == a
```

intersectRect checks whether two rectangles overlap. The first two arguments are the diagonal points of the first rectangle, and next two points that of the next rectangle.

```
> intersectRect :: Point -> Point -> Point -> Point -> Bool
> intersectRect r1a r1b r2a r2b =
>   inRect r1a r1b r2a ||
>   inRect r1a r1b r2b ||
>   inRect r2a r2b r1a ||
>   inRect r2a r2b r1b
```

inRect checks whether a point is inside a rectangle. The first two arguments are the diagonal points of the rectangle, and the third argument is the point to be checked.

```
> inRect    :: Point -> Point -> Point -> Bool
> inRect (rx1, ry1) (rx2, ry2) (x, y)   = let
>   in
>   isBetween rx1 rx2 x && isBetween ry1 ry2 y
```

hitObject is used to detect collision between a bullet whose position is given by the third argument, and an object whose boundary box is given by the first two arguments. The fourth argument is the speed of the bullet. If the bullet is near enough to the object (at a distance of speed / 2) the bullet is said to have hit the object.

```
> hitObject :: Point -> Point -> Point -> Int -> Bool
> hitObject p1 p2 (x, y) speed   = let
>   hspeed = speed `div` 2
>   r2a = (x - hspeed, y - hspeed)
>   r2b = (x + hspeed, y + hspeed)
>   in intersectRect p1 p2 r2a r2b
```

The rest of the functions are quite straightforward.

```
> type RandNum = Int
> type Range = (RandNum, RandNum)

> inRange   :: Range -> RandNum -> Bool
> inRange (a, b) r  = a <= r && b >= r

> translate :: Point -> Point -> Point
> translate (tx, ty) (x, y) = (tx + x, ty + y)


> eatEvents   :: Window -> IO ()
> eatEvents w = do
>   e <- maybeGetWindowEvent w
>   case e of
>       Nothing -> return ()
>       _   -> eatEvents w


> waitForKey:: Window -> (Key -> Bool) -> IO()
> waitForKey w foo  = do
>   e <- getWindowEvent w
>   case e of
>       Key {keysym = k'} | foo k'  -> return ()
>                 | otherwise   -> waitForKey w foo
>       _   -> waitForKey w foo
```