

```

9 initialize ids proc
10 # (erro, proc) = openWindow Void (janela ids) proc
11 erro <> NoError = proc
12 otherwise = proc
13
14 janela ids
15 = Window "Plota Bitmap"
16 {
17     ButtonControl "Plota Bitmap" [ControlWidth (PixelWidth 100),ControlPos
18     :+;
19     ButtonControl "Apaga Bitmap" [ControlWidth (PixelWidth 100),ControlPos
20     :+;
21     PopUpControl [( "coracao",nada), ("olho",nada), ("lua",nada), ("gota",nada)
22     :+;
23     PopUpControl [( "Coordenada X",nada)]+[(toString x ,nada)\x <- [0,10.
24     :+;
25     PopUpControl [( "Coordenada Y",nada)]+[(toString x ,nada)\x <- [0,10.
26     )
27     [
28         WindowViewSize {w=640,h=480}.
29         WindowPos (LeftTop,OffsetVector {vx=(maxFixedWindowSize.w-640 )/2,
30         :+;
31         WindowClose (noIS closeProcess),
32         WindowId wd1
33     ]

```

Observe que na função **initialize** não se colocou a leitura do bitmap de fundo e que nos atributos da janela também se eliminou a plotagem de um bitmap, configurando a janela em um tamanho fixo igual a 640 x 480 *pixel*, mantendo a mesma centralizada na área de trabalho.

• Colocar uma figura na janela criada

```

42   plota proc
43   # (Just dial, proc) = accPIO(getWindow wd1) proc
44   # (se,Just figura) = getControlText fig dial
45   # (se,Just coordX) = getControlText cx dial
46   # (se,Just coordY) = getControlText cy dial
47   # (maybeBitmap,proc) = openBitmap (figura++".bmp") proc
48   | isNothing maybeBitmap = proc
49   # bitmap = fromJust maybeBitmap
50   # proc = appPIO(appWindowPicture wd1 (drawAt {x=(toInt coordX),y=(toInt coordY)} bitmap)) proc
51   =proc

```

Esta função faz o seguinte:

- Lê os campos dos menus popup para saber qual figura deverá ser colocada (campo identificador **fig**);
- Qual a coordenada x (campo identificador **cx**);
- Qual a coordenada y (campo identificador **cy**) em que o mesmo será *plotado*.

Os nomes das figuras dos textos dos popups são os mesmos nomes dos bitmaps. Como os mesmos foram colocados, neste aplicativo, no mesmo diretório do executável, para carregar o bitmap correspondente basta acrescentar a extensão **.bmp** no mesmo na chamada da função **openBitmap**.

```

49   # (maybeBitmap,proc) = openBitmap (figura++".bmp") proc

```

Para plotar o bitmap na janela, utiliza-se a função **appWindowPicture**.

```

53   # proc = appPIO(appWindowPicture wd1 (drawAt {x=(toInt coordX),y=(toInt coordY)} bitmap)) proc

```

Esta função aplica uma *picture*, o bitmap, em uma janela, no caso **wd1**, desenhada na posição (**drauAt**) dada pela coordenada x e y.

• Apagar a janela criada

Para apagar a janela, basta alterar o *Look* da mesma. A função que altera o *Look* da janela é ***setWindowLook***. Os parâmetros devem ser configurados conforme código mostrado na figura a seguir.

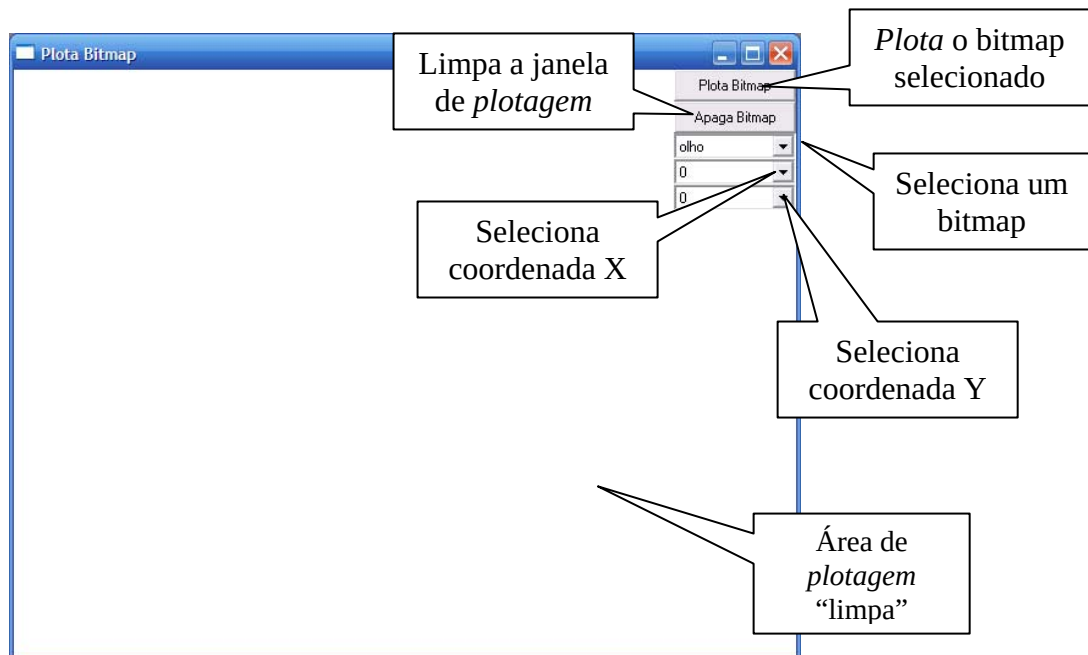
```

53  apaga proc|
54  # proc = appPIO (setWindowLook wd1 True (False, (look2))) proc
55  = proc
56
57  look2 Able {oldFrame} picture
58  # picture = unfill oldFrame picture
59  = picture
60
61

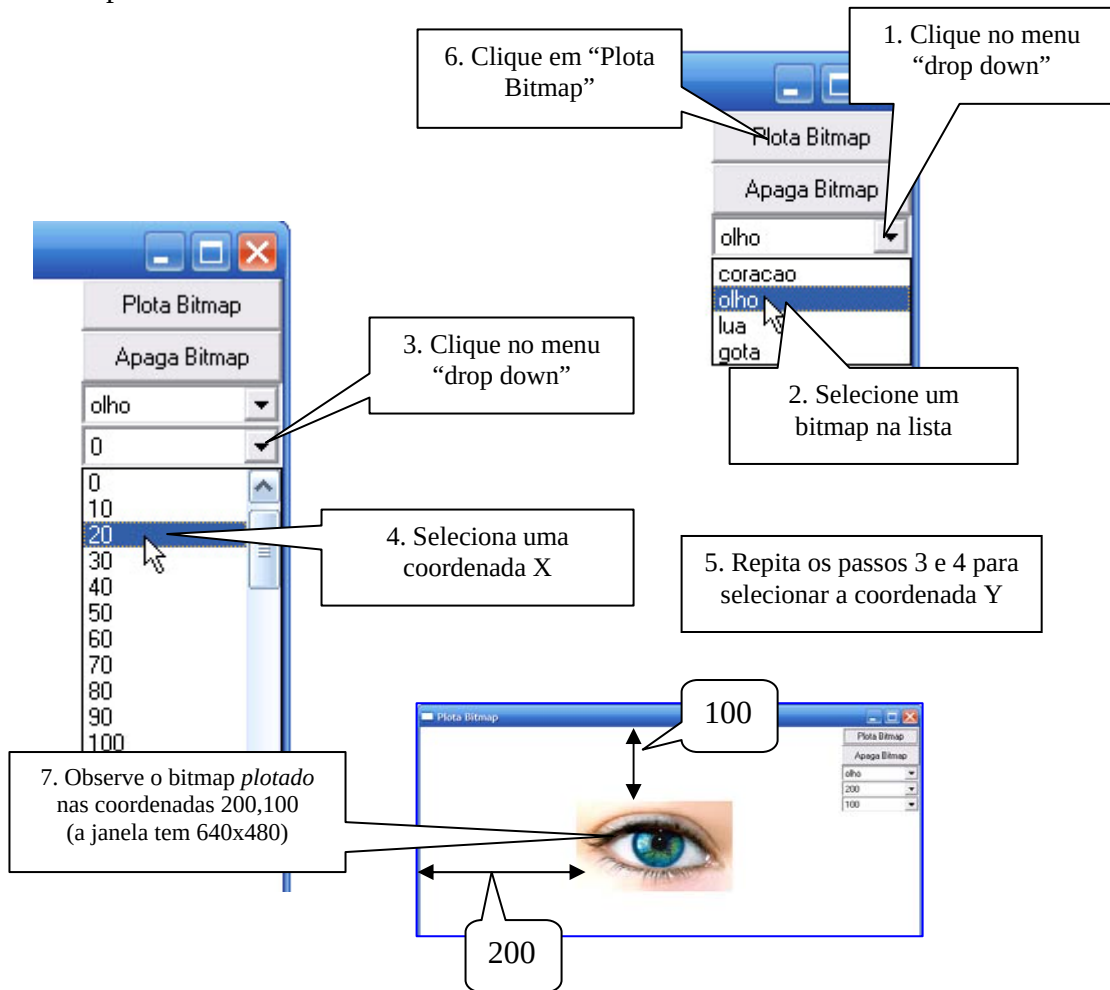
```

A função que apaga a janela, no *Look* da mesma, é ***unfill***. O formato da mesma é mostrado na mesma listagem anterior.

A figura abaixo mostra a interface inicial criada com os botões e menus para *plotagem* dos bitmaps.



A figura a seguir mostra a sequência de utilização da interface na *plotagem* de um bitmap na mesma.



A figura a seguir, mostra o programa completo.

```

1 module bitmapXY
2 import StdEnv, StdIO, StdArq
3
4 Start :: *World -> *World
5 Start world
6 # (ids,world) = openIds 7 world
7 = startIO SDI "" (initialize ids) [ProcessClose closeProcess] world
8
9 initialize ids proc
10 # (erro, proc) = openWindow Void (janela ids) proc
11 | erro <> NoError = proc
12 | otherwise = proc
13
14 janela ids
15 = Window "Plota Bitmap"
16   ( ButtonControl "Plota Bitmap" [ControlWidth (PixelWidth 100),ControlPos (Right,zero),ControlFunction (noLS plotar)]
17   ++ ButtonControl "Apaga Bitmap" [ControlWidth (PixelWidth 100),ControlPos (Right,zero),ControlFunction (noLS apaga)]
18   ++ PopUpControl [{"coracao",nada}, {"olho",nada}, {"lua",nada}, {"gota",nada}] 2 [ControlWidth (PixelWidth 100),ControlPos (Right,zero),ControlId fig]
19   ++ PopUpControl [{"Coord. X",nada}] ++ [(toString x, nada) \x <- [0,10..500]] 2 [ControlWidth (PixelWidth 100),ControlPos (Right,zero),ControlId cx]
20   ++ PopUpControl [{"Coord. Y",nada}] ++ [(toString y, nada) \y <- [0,10..300]] 2 [ControlWidth (PixelWidth 100),ControlPos (Right,zero),ControlId cy] )
21
22 [ WindowViewSize (w=640,h=480),
23   WindowPos (LeftTop,OffsetVector (vx=(maxFixedWindowSize.w-640)/2,vy=(maxFixedWindowSize.h-480)/2)),
24   WindowClose (noLS closeProcess),
25   WindowId wdl ]
26
27 where
28   wdl = ids!!0
29   fig = ids!!1
30   cx = ids!!2
31   cy = ids!!3

```

```

33 nada proc=proc
34
35 plota proc
36 # (Just dial, proc) = accPIO(getWindow wdl) proc
37 # (se,Just figura) = getControlText fig dial
38 # (se,Just coordX) = getControlText cx dial
39 # (se,Just coordY) = getControlText cy dial
40 # (maybeBitmap,proc) = openBitmap (figura++".bmp") proc
41 | isNothing maybeBitmap = proc
42 # bitmap = fromJust maybeBitmap
43 # proc = appPIO(appWindowPicture wdl (drawAt (x=(toInt coordX),y=(toInt coordY)) bitmap)) proc
44 =proc
45
46 apaga proc
47 # proc = appPIO (setWindowLook wdl True (False, (look2))) proc
48 = proc
49
50 look2 Able (oldFrame) picture
51 # picture = unfill oldFrame picture
52 = picture
53

```

Interface para leitura e reconhecimento de Eventos de Mouse

Na maioria dos aplicativos visuais, é de fundamental importância saber ler os eventos do mouse, tais como: *MouseMove*, *MouseUp*, *MouseDown* e *MouseDown*. É fundamental, também, conhecer as coordenadas de posicionamento do ponteiro do mouse.

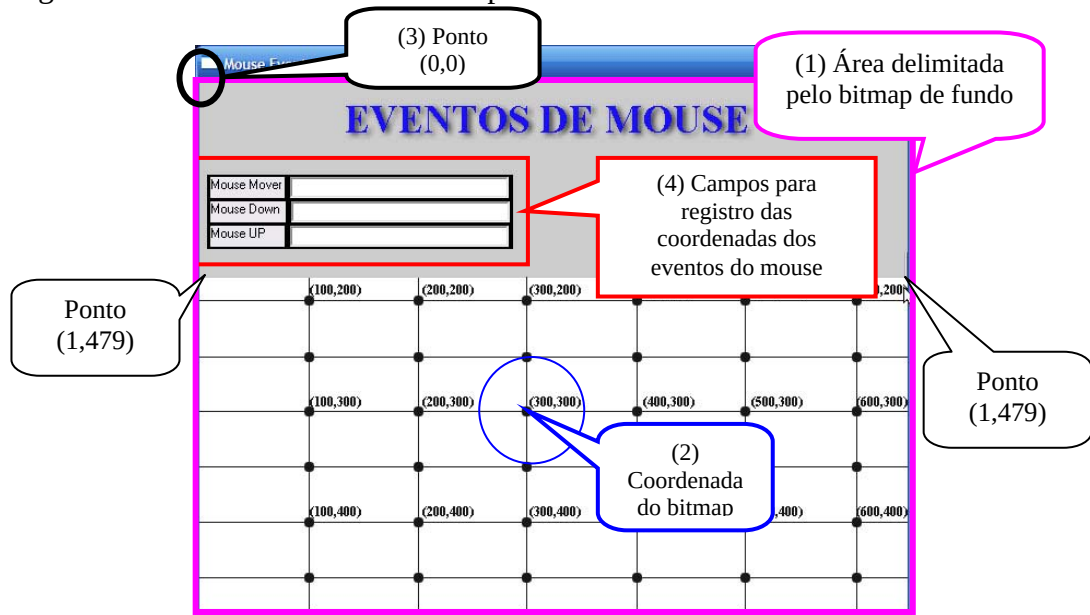
Para se criar aplicativos musicais, tal como um teclado virtual ou um editor de partituras, por exemplo, o conhecimento dos eventos do mouse e o posicionamento do cursor é a base da maioria das funções a serem implementadas. Conhecer a posição do cursor do mouse é fundamental para, por exemplo, eliminar notas de um pentagrama musical, assim como o ato de se clicar em uma tecla de um teclado virtual para tocar uma nota musical ou plotá-la em um pentagrama (estas ações serão vistas no estudo de caso de implementação de um editor de partituras no capítulo 6).

Novidade na interface SDI neste tipo de aplicativo:

Inclusão de uma função que monitora os eventos do mouse em tempo real, nos atributos da janela criada, e passa as mensagens dos eventos como argumento de uma função, permitindo-se, assim, separar e qualificar os tipos de eventos, tais como:

- Movimento (coordenadas);
- *mouse down* (quando a tecla esquerda do mouse é pressionada);
- *mouse up* (quando a tecla é liberada);

A figura abaixo mostra a interface do aplicativo “Eventos de Mouse”.



Esta interface possui:

- Um bitmap de fundo (1) contendo um mapa de pontos (2) com as respectivas coordenadas em relação ao canto superior esquerdo (3) do bitmap (ponto (0,0));
- Três campos de texto (4) para registro das coordenadas dos eventos do mouse.

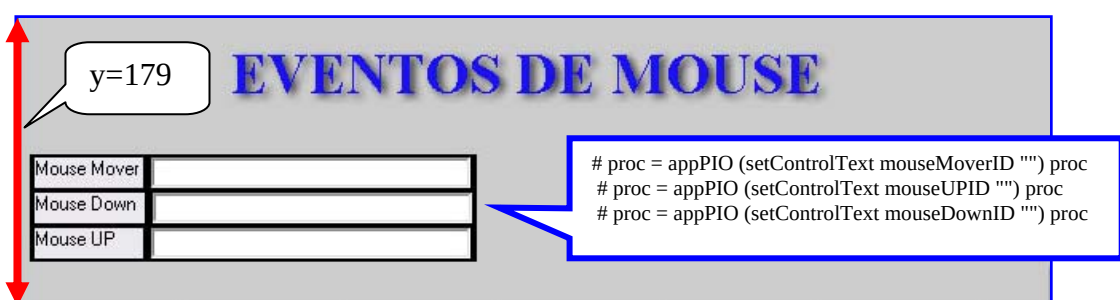
O aplicativo foi desenvolvido para mostrar as coordenadas do mouse apenas na região do bitmap onde está o mapa das coordenadas, conforme pode ser visto no trecho de código apresentado a seguir.

```

76 | posY < 179 |
77 | # proc = appPIO (setControlText mouseMoverID "") proc
78 | # proc = appPIO (setControlText mouseUPID "") proc
79 | # proc = appPIO (setControlText mouseDownID "") proc
80 | = proc

```

Assim, caso o ponteiro do mouse caia fora da região prevista do mapa de coordenadas, ou seja, com a coordenada vertical y, menor que 179, o programa apenas apaga os textos dos campos de registro de eventos de mouse.



Caso caia fora da região da janela, o monitoramento do mouse não é efetuado.

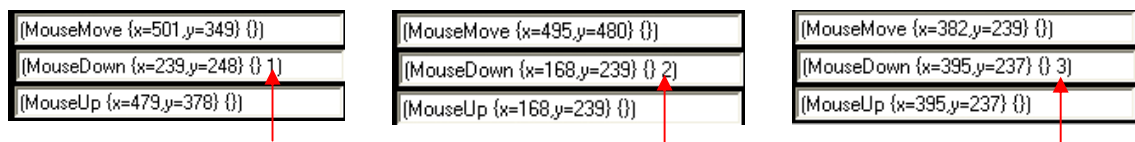
A função que monitora o cursor do mouse na janela é **WindowMouse**, colocada nos atributos da janela, conforme pode ser visto na figura abaixo.

```

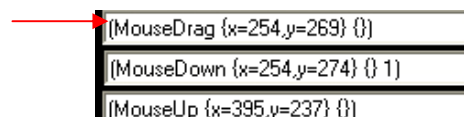
51                                     [ControlId mouseUPID]
52                                     )
53                                     [ WindowMouse (const True) Able (noLS1 (rastMouse)).
54                                       WindowViewSize bitmapsize.
55                                       WindowLook True (\_-->drawAt zero bitmap).
56                                       WindowPos (LeftTop,OffsetVector {vx=(maxFixedWindowSize.w-(bi
57                                       WindowHMargin 10 10,
58                                       WindowVMargin 23 23,
59                                       WindowItemSpace 5 3,
60                                       WindowClose (noLS closeProcess).
61                                       WindowId wd1
62                                     ]
63 where
64   mouseMoverID = ids!!0
65   mouseDownID  = ids!!1
66   mouseUPID    = ids!!2
67   wd1 = ids!!3
68
69   nada proc=proc
70
71   rastMouse x proc
72   # pos = separaCoordenada (toString (drop 11 [y \ \ y <-: (toString x)]))
73   # listaCoordenadas = StringTokens testaDigito pos
74   # posX =toInt( listaCoordenadas!!0)
75   # posY =toInt( listaCoordenadas!!1)
76   | posY < 179 || posX > 646 || posX < 1 || posY >479
77   # proc = appPIO (setControlText mouseMoverID "") proc
78   # proc = appPIO (setControlText mouseUPID "") proc
79   # proc = appPIO (setControlText mouseDownID "") proc
80   = proc

```

Esta função, a **WindowMouse**, dispara outra função, que será encarregada de separar e identificar cada tipo de evento. No exemplo esta função é a **rastMouse**. Ela possui um argumento o qual é, em tempo real, atualizado a todo momento pela **WindowMouse**, indicando a posição do cursor do mouse e o respectivo evento (MouseMove, MouseDrag, MouseDown ou MouseUp). A figura, a seguir, mostra as mensagens enviadas pela função **WindowMouse** como parâmetro de **rastMouse**. Observe que quando o evento é um **MouseDown**, tem-se uma informação numérica a mais do que nos outros eventos (no caso: 1, 2 ou 3). Esta informação diz quantos cliques de mouse ocorreram em **MouseDown**, o que é útil para implementar programas que lêem duplos e triplos cliques.



Quando se mantém o botão do mouse pressionado, no lugar de **MouseDown** uma mensagem de **MouseDrag** é enviada, conforme mostra a figura a seguir:



Para pegar as coordenadas dos eventos de mouse, deve-se eliminar o texto que identifica o evento. Para tanto, deve-se seguir as etapas:

1- Identificar os eventos.

- “(MouseDown“ possui 10 caracteres. Para identificar, basta comparar se a mensagem possui estes 10 caracteres no início dela.

```
82 | ((toString (take 10 [y \ y <-: (toString x)])) == "(MouseDown")
```

10 primeiros caracteres da mensagem de **WindowMouse**

- “(MouseUp“ possui 8 caracteres. Para identificar, basta comparar se a mensagem possui estes 8 caracteres no início dela.

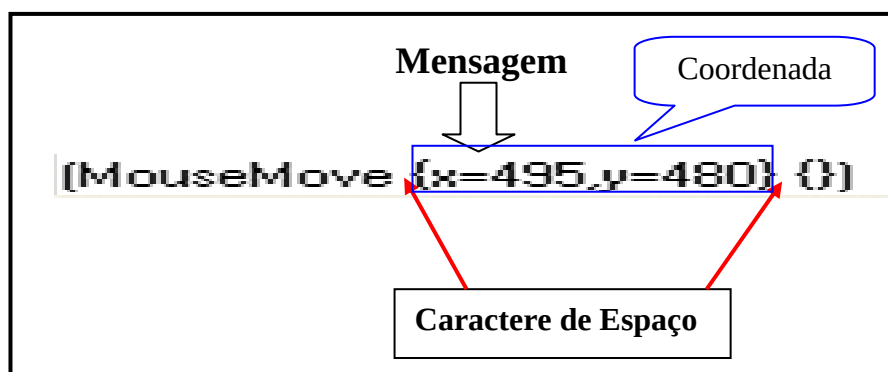
```
89 | ((toString (take 8 [y \ y <-: (toString x)])) == "(MouseUp")
```

- O evento MouseMove é tratado de forma diferente, ou seja, elimina-se os 11 primeiros caracteres da string enviada por **WindowMouse** e mostra as coordenadas sempre.

```
72 # pos = separaCoordenada (toString (drop 11 [y \ y <-: (toString x)]))
```

- O evento MouseDrag não foi tratado, mas é similar ao MouseDown e MouseUp.

A função separaCoordenada é responsável por pegar do restante da string de evento enviada por **WindowMouse**, apenas as coordenadas. Para tanto, utilizou-se novamente a função **StringTokens** da biblioteca StdArq que fará esta tarefa, ou seja, como existe um caractere de espaço entre as coordenadas e o restante da string, conforme mostrado a seguir basta, apenas separar a string pelo caractere de espaço (item 3.4.1 do Capítulo 3), e pegar o primeiro elemento (a coordenada).



Neste exemplo, a coordenada é `{x=495,y=480}`. O trecho de programa, em destaque, que faz esta ação é mostrado na figura a seguir.

```

70
71 rastMouse x proc
72 # pos = separaCoordenada (toString (drop 11 [y \ y <-: (toString x)]))
73 # listaCoordenadas = StringTokens testaDigito pos
74 # posX =toInt( listaCoordenadas!!0)
75 # posY =toInt( listaCoordenadas!!1)
76 | posY < 179 || posX > 646 || posX < 1 || posY >479
77 # proc = appPIO (setControlText mouseMoverID "") proc
78 # proc = appPIO (setControlText mouseUPID "") proc
79 # proc = appPIO (setControlText mouseDownID "") proc
80 = proc
81 # proc = appPIO (setControlText mouseMoverID pos) proc
82 | ((toString (take 10 [y \ y <-: (toString x)])) == "(MouseDown)")
83 # pos = separaCoordenada (toString (drop 11 [y \ y <-: (toString x)]))
84 # listaCoordenadas = StringTokens testaDigito pos
85 # posX =toInt( listaCoordenadas!!0)
86 # posY =toInt( listaCoordenadas!!1)
87 # proc = appPIO (setControlText mouseDownID pos) proc
88 |otherwise=proc
89 | ((toString (take 8 [y \ y <-: (toString x)])) == "(MouseUp)")
90 # pos = separaCoordenada (toString (drop 9 [y \ y <-: (toString x)]))
91 # proc = appPIO (setControlText mouseUPID pos) proc
92 = proc
93 = proc
94
95 separaCoordenada str = hd (StringTokens isCaracEspaco str)
96
97 testaDigito x = not(isDigit x)
98
99

```

2- Pegar a coordenada numérica do evento

Observe que quando se identificou o evento e pegou-se a respectiva coordenada, a mesma ainda vem no tipo string e com caracteres de letras e símbolos. Para manipular a coordenada como um tipo inteiro deve-se eliminar estes caracteres não dígitos e separar as duas coordenadas, como, por exemplo, em uma lista com dois elementos: [coordX,coordY].

O trecho de programa que faz isto é dado a seguir:

```

70
71 rastMouse x proc
72 # pos = separaCoordenada (toString (drop 11 [y \ y <-: (toString x)]))
73 # listaCoordenadas = StringTokens testaDigito pos
74 # posX =toInt( listaCoordenadas!!0)
75 # posY =toInt( listaCoordenadas!!1)
76 | posY < 179 || posX > 646 || posX < 1 || posY >479
77 # proc = appPIO (setControlText mouseMoverID "") proc
78 # proc = appPIO (setControlText mouseUPID "") proc
79 # proc = appPIO (setControlText mouseDownID "") proc
80 = proc
81 # proc = appPIO (setControlText mouseMoverID pos) proc
82 | ((toString (take 10 [y \ y <-: (toString x)])) == "(MouseDown)")
83 # pos = separaCoordenada (toString (drop 11 [y \ y <-: (toString x)]))
84 # listaCoordenadas = StringTokens testaDigito pos
85 # posX =toInt( listaCoordenadas!!0)
86 # posY =toInt( listaCoordenadas!!1)
87 # proc = appPIO (setControlText mouseDownID pos) proc
88 |otherwise=proc
89 | ((toString (take 8 [y \ y <-: (toString x)])) == "(MouseUp)")
90 # pos = separaCoordenada (toString (drop 9 [y \ y <-: (toString x)]))
91 # proc = appPIO (setControlText mouseUPID pos) proc
92 = proc
93 = proc
94
95 separaCoordenada str = hd (StringTokens isCaracEspaco str)
96
97 testaDigito x = not(isDigit x)
98
99

```

A função `StringTokens` aplica a função **testaDigito** na string, eliminando todos os caracteres não dígitos (**not(isDigit)**), ficando apenas as coordenadas separadas em uma lista.

A figura a seguir, mostra o programa completo de rastreamento de mouse e eventos.

```

1 module mousemover
2 import StdEnv, StdIO, StdArq
3
4 Start :: *World -> *World
5 Start world
6 # (ids, world) = openIds 5 world
7
8 = startIO SDI 0 (initialize ids) [ProcessClose closeProcess] world
9
10 initialize ids proc
11 # (maybeBitmap, proc) = openBitmap "mousemover.bmp" proc
12 | isNothing maybeBitmap = proc
13 | otherwise
14   # bitmap = fromJust maybeBitmap
15   # bitmapsize = getBitmapSize bitmap
16   # (erro, proc) = openWindow Void (janela ids bitmap bitmapsize) proc
17   | erro <> NoError = proc
18   | otherwise = proc
19
20 janela ids bitmap bitmapsize
21 = Window "Mouse Events"
22   ( TextControl ("Mouse Mover") [ControlPos (Center, zero), ControlHide]
23   ++ TextControl ("Escolha a Nova PPQ") [ControlPos (Center, zero), ControlHide]
24   ++ TextControl ("Escolha a Nova PPQ") [ControlPos (Center, zero), ControlHide]
25   ++ TextControl ("Mouse Mover") [ControlPos (Left, zero)]
26   ++ EditControl "" (PixelWidth 200) 1 [ControlId mouseMoverID]
27   ++ TextControl ("Mouse Down ") [ControlPos (Left, zero)]
28   ++ EditControl "" (PixelWidth 200) 1 [ControlId mouseDownID]
29   ++ TextControl ("Mouse UP ") [ControlPos (Left, zero)]
30   ++ EditControl "" (PixelWidth 200) 1 [ControlId mouseUPID] )
31   [ WindowMouse (const True) Able (noLS1 (rastMouse)),
32     WindowViewSize bitmapsize,
33     WindowLook True (\_ _->drawAt zero bitmap),
34     WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w) )/2
35                                     ,vy=(maxFixedWindowSize.h- (bitmapsize.h))/2}),
36     WindowHMargin 10 10,
37     WindowVMargin 23 23,
38     WindowItemSpace 5 3,
39     WindowClose (noLS closeProcess),
40     WindowId wd1 ]
41 where
42   mouseMoverID = ids!!0
43   mouseDownID = ids!!1
44   mouseUPID = ids!!2
45   wd1 = ids!!3
46
47   rastMouse x proc
48   # pos = separaCoordenada (toString (drop 11 [y \\ y <-: (toString x)]))
49   # listaCoordenadas = StringTokens testaDigito pos
50   # posX = toInt( listaCoordenadas!!0)
51   # posY = toInt( listaCoordenadas!!1)
52   | posY < 179
53   # proc = appPIO (setControlText mouseMoverID "") proc
54   # proc = appPIO (setControlText mouseUPID "") proc
55   # proc = appPIO (setControlText mouseDownID "") proc
56   = proc
57   # proc = appPIO (setControlText mouseMoverID pos) proc
58   | ((toString (take 10 [y \\ y <-: (toString x)])) == "(MouseDown")
59   # pos = separaCoordenada (toString (drop 11 [y \\ y <-: (toString x)]))
60   # listaCoordenadas = StringTokens testaDigito pos
61   # posX = toInt( listaCoordenadas!!0)
62   # posY = toInt( listaCoordenadas!!1)
63   # proc = appPIO (setControlText mouseDownID pos) proc
64   | otherwise=proc
65   | ((toString (take 8 [y \\ y <-: (toString x)])) == "(MouseUp")
66   # pos = separaCoordenada (toString (drop 9 [y \\ y <-: (toString x)]))
67   # proc = appPIO (setControlText mouseUPID pos) proc
68   = proc
69   | otherwise = proc
70
71 separaCoordenada str = hd (StringTokens isCaracEspaco str)
72
73 testaDigito x = not(isDigit x)
74

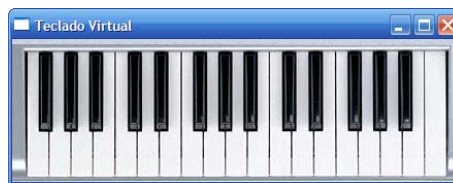
```

Teclado Virtual

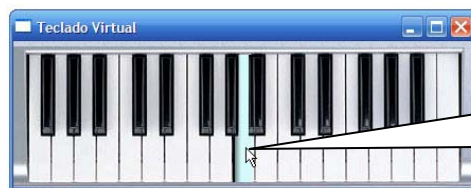
O Teclado Virtual é um exemplo de programa em Clean, que utiliza um conceito muito importante: a utilização de arquivos bmps (*bitmap files*) que devem ser mostrados repetidamente. Esses arquivos criam (ao movimento ou clique do mouse) a ilusão de se pressionar a tecla, sobrepondo-a por um arquivo bmp de outra cor.

Neste aplicativo, é utilizada a rotina de rastreamento da posição do mouse com relação ao arcabouço da janela e a *plotagem* de bitmaps. É criada uma animação que simula uma tecla ser pressionada ao clique ou movimento do mouse sobre as teclas. Neste exemplo ao se movimentar o mouse sobre as teclas, elas mudam de cor (azul), e ao clique do mouse, além de mudar para outra cor (vermelha), um arquivo sonoro (MIDI), equivalente ao som da nota musical pressionada, é disparado, simulando a ação de se tocar ao piano.

A figura abaixo, mostra a interface principal do programa.



- Movimento do ponteiro do mouse sobre as teclas.



Ao movimentar (sem clicar) o ponteiro do mouse sobre as teclas, as mesmas mudam a cor para azul claro.

- Ao clique do mouse, um som de piano será emitido.

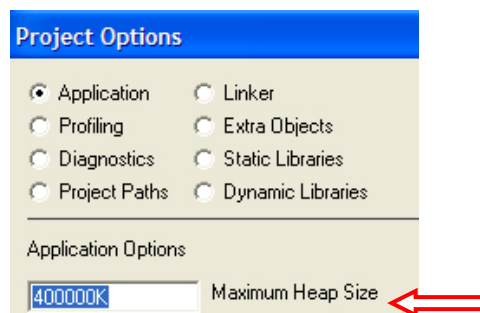


Para emitir sons, basta clicar sobre alguma tecla, e a cor desta, muda para vermelho. Ao mesmo tempo um arquivo sonoro (MIDI) correspondente à nota, é disparado.

Existe um problema a ser solucionado ao se trabalhar com bitmaps que serão reutilizados várias vezes no programa.

O PROBLEMA: Se no programa for utilizada a técnica de abrir o bmp a cada vez que ele for mostrado, já que é pequeno, o programa poderá, após alguns segundos de uso, emitir uma mensagem de erro do sistema operacional. O que ocorre é que o Sistema Operacional Windows tem um *bug* que ocorre ao se tentar abrir o mesmo arquivo repetidamente, mais de vinte vezes.

UMA SOLUÇÃO: Para evitar este problema, utiliza-se a técnica de abrir todos os bitmaps que serão utilizados mais de 20 vezes, ao iniciar o programa. Assim, o problema de abri-lo várias vezes será eliminado. A única desvantagem é que a máquina deverá possuir uma memória RAM que permita se armazenar todos os bitmaps abertos. Feito isto, é só configurar, em **Project Options**, o **Heap** do Clean para que o programa possa rodar.



A figura abaixo, mostra o trecho do programa que cria um tipo de registro com campos que permitirão criar um registro que armazene os bmps que serão abertos.

```

6 :: Registro = {
7     notas      :: [{#Char}]
8     ,nota      :: Int
9     ,notaAnt   :: {#Char}
10    ,teclado    :: (Bitmap)
11    ,tcDoBlue   :: (Bitmap)
12    ,tcDoRed    :: (Bitmap)
13    ,tcMiBlue   :: (Bitmap)
14    ,tcMiRed    :: (Bitmap)
15    ,tcSolBlue  :: (Bitmap)
16    ,tcSolRed   :: (Bitmap)
17    ,tcPretaBlue :: (Bitmap)
18    ,tcPretaRed  :: (Bitmap)
19    ,tcDoUltimoBlue :: (Bitmap)
20    ,tcDoUltimoRed  :: (Bitmap)
21 }

```

Criação das variáveis dos campos do registro que receberão os bmps que serão abertos.

A figura a seguir, mostra o trecho do programa que carrega os bmps.

```

25 Start :: *World -> *World
26 Start world
27
28 # (maybeBitmap,world) = openBitmap "teclado.bmp" world
29 # bitmapTeclado       = (fromJust maybeBitmap)
30 # bitmapsize          = getBitmapSize (snd3 bitmapTeclado)
31 # (maybeBitmap,world) = openBitmap "tcDoBlue.bmp" world
32 # bitmapTcDoBlue      = (fromJust maybeBitmap)
33 # (maybeBitmap,world) = openBitmap "tcDoRed.bmp" world
34 # bitmapTcDoRed       = (fromJust maybeBitmap)
35 # (maybeBitmap,world) = openBitmap "tcMiBlue.bmp" world
36 # bitmapTcMiBlue      = (fromJust maybeBitmap)
37 # (maybeBitmap,world) = openBitmap "tcMiRed.bmp" world
38 # bitmapTcMiRed       = (fromJust maybeBitmap)
39 # (maybeBitmap,world) = openBitmap "tcSolBlue.bmp" world
40 # bitmapTcSolBlue     = (fromJust maybeBitmap)
41 # (maybeBitmap,world) = openBitmap "tcSolRed.bmp" world
42 # bitmapTcSolRed      = (fromJust maybeBitmap)
43 # (maybeBitmap,world) = openBitmap "tcPretaBlue.bmp" world
44 # bitmapTcPretaBlue   = (fromJust maybeBitmap)
45 # (maybeBitmap,world) = openBitmap "tcPretaRed.bmp" world
46 # bitmapTcPretaRed    = (fromJust maybeBitmap)
47 # (maybeBitmap,world) = openBitmap "tcDoUltimoBlue.bmp" world
48 # bitmapTcDoUltimoBlue = (fromJust maybeBitmap)
49 # (maybeBitmap,world) = openBitmap "tcDoUltimoRed.bmp" world
50 # bitmapTcDoUltimoRed  = (fromJust maybeBitmap)
51

```

A figura a seguir, mostra o trecho do programa que cria um registro chamado *parametIni*, inicializando as variáveis dos campos do registro com os bmps lidos, bem como outras que serão utilizadas no aplicativo.

```

55
56 # parametIni = {
57     notas      = [""]
58     ,nota      = 0
59     ,notaAnt    = ""
60     ,teclado    = bitmapTeclado
61     ,tcDoBlue   = bitmapTcDoBlue
62     ,tcDoRed    = bitmapTcDoRed
63     ,tcMiBlue   = bitmapTcMiBlue
64     ,tcMiRed    = bitmapTcMiRed
65     ,tcSolBlue  = bitmapTcSolBlue
66     ,tcSolRed   = bitmapTcSolRed
67     ,tcPretaBlue = bitmapTcPretaBlue
68     ,tcPretaRed = bitmapTcPretaRed
69     ,tcDoUltimoBlue = bitmapTcDoUltimoBlue
70     ,tcDoUltimoRed = bitmapTcDoUltimoRed
71 }
72
73 # (ids,world)= openIds 7 world
74
75 = startIO SDI parametIni (initialize ids ) [ProcessClose closeProcess] world
76

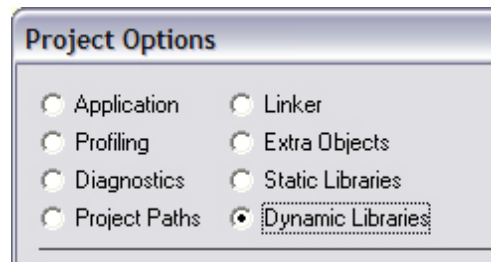
```

Diferença básica das interfaces anteriores deste anexo:

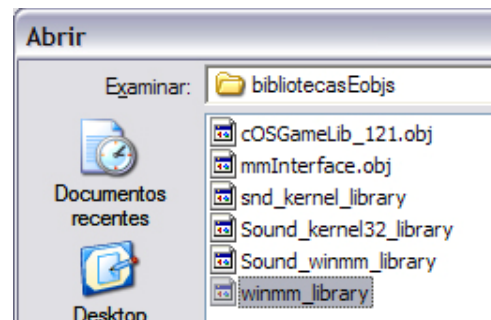
Este programa, para rodar os arquivos MIDI, utiliza bibliotecas e Objs do C (encontradas no CD em anexo na pasta: “bibliotecasEobjs”). Assim, deve-se integrar estas funções no projeto do programa do teclado virtual, caso contrário, várias funções não serão reconhecidas e mensagens de erro serão emitidas pelo sistema.

A inclusão de libs, dlls, Objs em Clean estão definidas com detalhes no trabalho de Rufino [6]. Para incluir estes arquivos no projeto, basta seguir os seguintes passos:

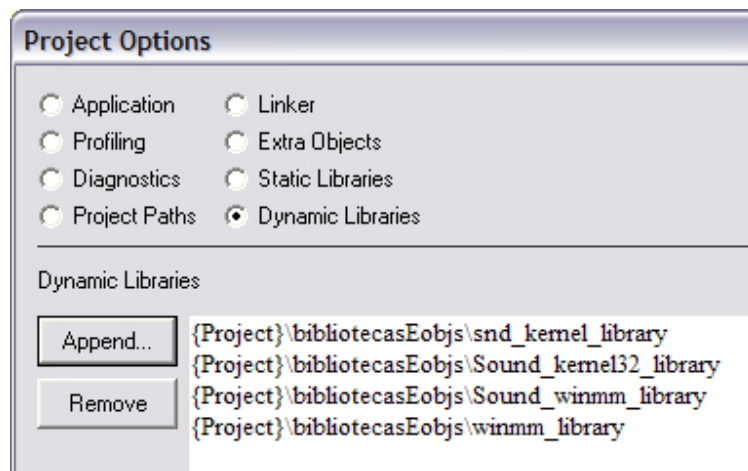
1. Ativar o menu Project e entrar na opção Options. Nesta janela selecione *Dynamic Libraries* para dar entrada às dlls de som e MIDI.



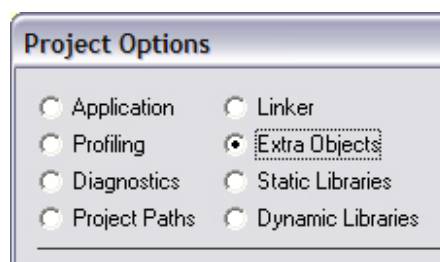
2. Ao clicar no botão **Append** desta janela, abre-se o gerenciador de arquivos para que se escolha, **uma a uma**, as dlls que serão inseridas no projeto.



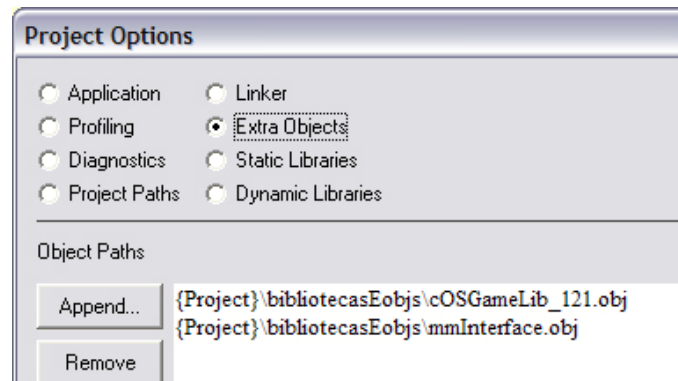
3. Neste aplicativo, quatro dlls devem ser incluídas. Após incluir todas, basta clicar em no botão **OK** desta janela.



4. Neste aplicativo também se utilizam duas Objs do C. Para incluí-las, o processo é idêntico ao de dlls, apenas diferenciando que se deve marcar em **Project Options** a opção **Extra Objects**.



5. As duas Objs incluídas são:



6. Seguidos estes passos, basta compilar o arquivo.

As ações do mouse na interface são:

1- Quando uma tecla é apertada no teclado virtual, o evento de **MouseDown** é detectado, um bitmap é *plotado* e o som correspondente é ativado através da função **playMid** da biblioteca **playMIDI**. Esta função **playMid** interrompe uma música que estiver sendo tocada e dispara outra, segundo um endereço fornecido como argumento. O trecho de programa que realiza esta tarefa é mostrado a seguir:

```
|((toString (take 10 [y \ y <-: (toString x)])) == "(MouseDown")
# proc = appPIO(appWindowPicture wdl (drawAt {x=xX,y=yY} bmp1)) proc
# (_, proc) = playMid ("piano\\"+++nota+++".mid") proc
= {ls = {proc.ls& notaAnt=nota}, io = proc.io}
```

As notas musicais estão armazenadas em um diretório na raiz do aplicativo de nome **piano**.

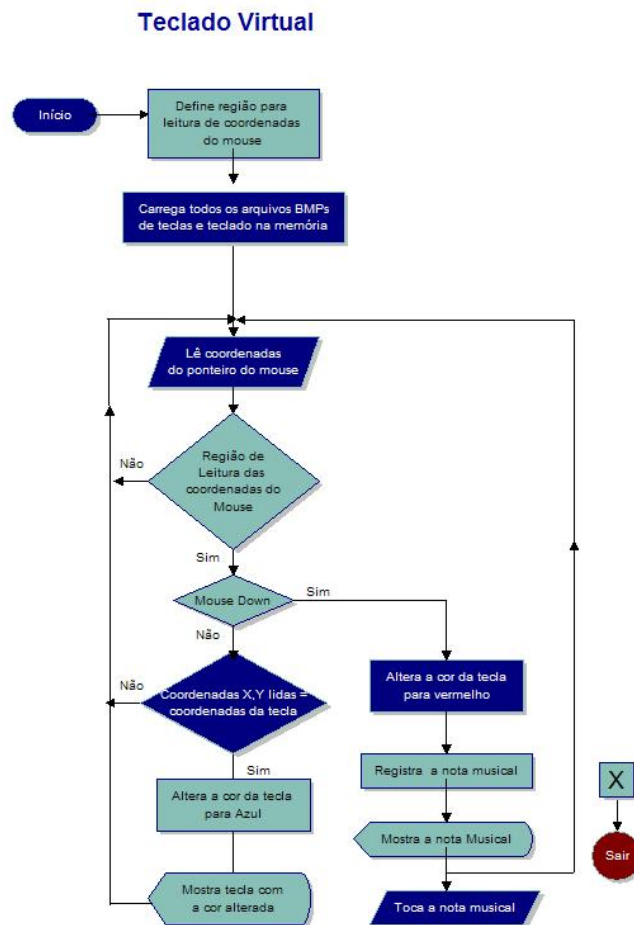
2- Quando a tecla é liberada (**MouseUp**), apenas se *plota* o bitmap novamente com as teclas sem estarem pressionadas.

```
|((toString (take 8 [y \ y <-: (toString x)])) == "(MouseUp")
# proc = appPIO(appWindowPicture wdl (drawAt {x=xX,y=yY} bmp)) proc
= {ls = {proc.ls& notaAnt=nota}, io = proc.io}
```

3- Enquanto o mouse apenas está se movendo, quando o ponteiro está em cima da área das teclas, o mesmo identifica qual é a área, define o bitmap correto e *plota* o mesmo (tecla azul) na posição correta do teclado virtual. O trecho parcial de código, a seguir, mostra a função que faz a seleção do bitmap.

```
testaPosicaoSemClique posX posY proc=: {ls={notaAnt,teclado,tcDoBlue, tcDoRed, tcMiBlue, tcMiRed, tcSolBlue,
// PARTE BAIXA DAS TECLAS BRANCAS
| (posX>15 && posX<35) && (posY>py1 &&posY<py2) = (notaAnt,tcDoBlue,16,py3,"fa3",0,proc)// fa3
| (posX>36 && posX<55) && (posY>py1 &&posY<py2) = (notaAnt,tcSolBlue,36,py3,"sol3",0,proc) //sol3
```

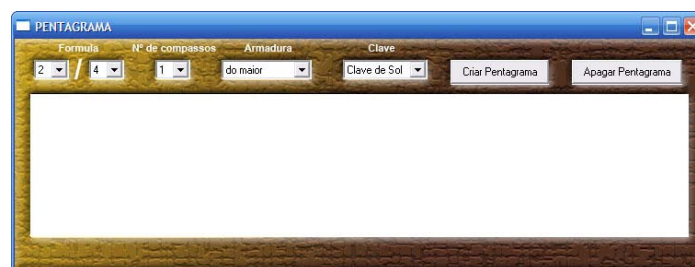
A figura abaixo, mostra o fluxograma da lógica da implementação da interface.



No anexo em CD está colocado o programa completo que implementa este fluxograma.

Criar Pentagrama

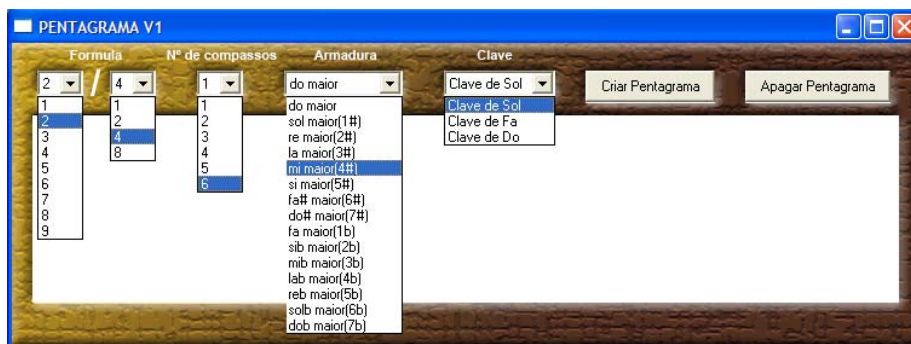
Este aplicativo mostra como criar pentagramas utilizando fontes de texto, onde cada letra corresponde em parte, ou no todo, a um símbolo musical⁷⁴. O aplicativo mostra e implementa como deixar a cargo do usuário a escolha da quantidade de compassos, a fórmula de compasso, a armadura de clave e o tipo de clave. A figura a seguir, mostra a interface inicial do aplicativo, o qual, novamente, utiliza popups e botões, elementos já vistos com detalhes nos itens anteriores.



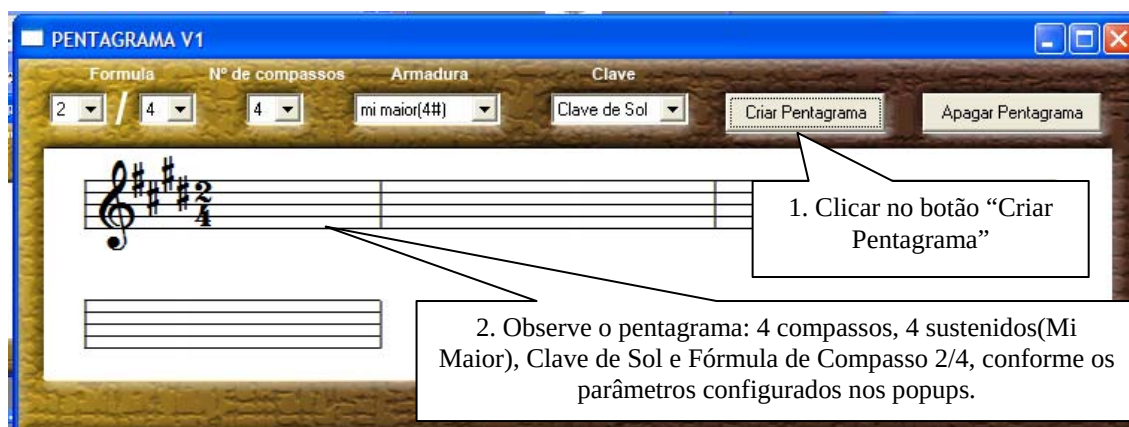
⁷⁴ Para ser ter uma visão correta do aplicativo, deve-se instalar a fonte PE.TTF. (Petrucchi).

Não é objetivo deste aplicativo colocar notas em pentagrama, isto será mostrado no item seguinte, visa-se apenas apresentar o conceito de como utilizar as fontes de textos com símbolos musicais e como plotar as mesmas na janela criada para criar os elementos básicos de uma partitura, tais como pentagramas divididos por compassos, inserção de armadura de clave, tonalidade e fórmula de compasso. No capítulo de estudos de casos, Capítulo 6, será mostrado um aplicativo completo com edição de notas em pentagramas multi-instrumentais e multicanais.

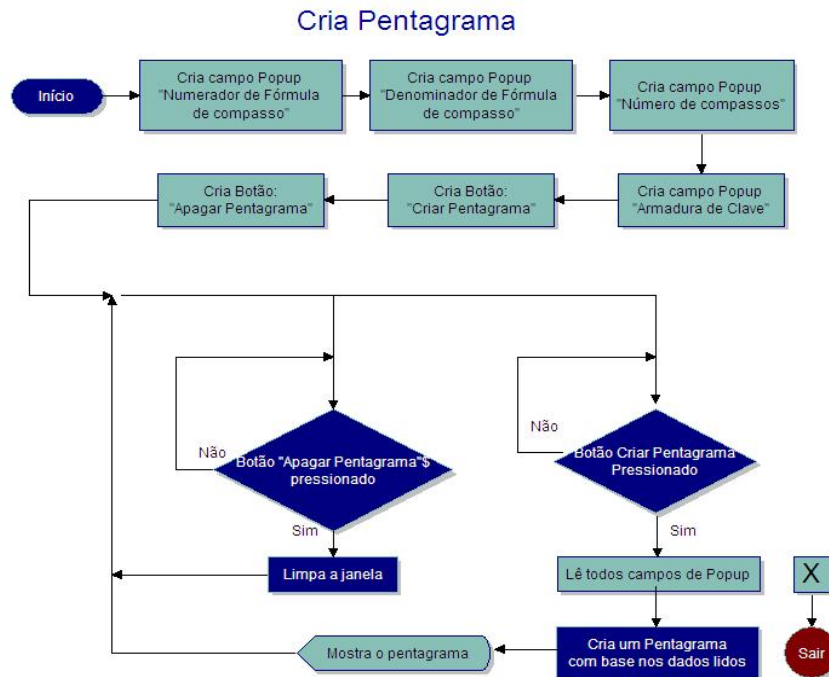
A definição dos parâmetros da fórmula de compasso, número de compassos, da armadura de clave e da clave é feita através de menus popups já definidos anteriormente. Assim, na figura a seguir, omite-se o código de programa de criação dos popups e apresenta-se apenas a interface com os conteúdos dos mesmos.



Para criar o pentagrama, basta clicar no botão **Criar Pentagrama**. A figura a seguir mostra a interface com o pentagrama *plotado* com as configurações feitas nos popups.



O fluxograma a seguir mostra a lógica de criação desta interface.



Devido ao programa ser relativamente longo, já que o código cria todos os elementos da interface, o mesmo foi colocado no CD em anexo, ficando, a seguir, a apresentação apenas de trechos da listagem do programa que definem os conceitos e ferramentas computacionais utilizadas na implementação da interface.

Assim, o programa de criação dos pentagramas conforme configurações feitas na interface inicial, *plota* os elementos ao se clicar no botão **Criar Pentagrama**. Ao se fazer isto, dispara-se a função **criaPentagrama**, cujo código do botão é mostrado a seguir na figura abaixo.

```
63 ButtonControl "Criar Pentagrama" [ControlFunction (noLS criaPentagrama)]
```

Listagem 5.41 – Controle de criação do botão de criar pentagrama

A função **criaPentagrama** inicia lendo os popups de configuração, conforme trecho do programa mostrado na figura a seguir:

```

92 criaPentagrama proc =: {ls={fundo}}
93 # (Just dial, proc) = accPIO(getWindow wd1) proc
94 # (se,Just fCompN) = getControlText t1N dial
95 # (se,Just fCompD) = getControlText t1D dial
96 # (se,Just nCompassos) = getControlText t2 dial
97 # (se,Just armadura) = getControlText t3 dial
98 # (se,Just clave) = getControlText t4 dial

```

Após a leitura dos *popups* de configuração, a função prossegue plotando novamente o bitmap de fundo e as configurações feitas na interface, através da função **lookCria**. Este ato faz com que se apague um pentagrama existente e se coloque outro com as configurações novas, parecendo, ao usuário, que somente o elemento modificado do pentagrama foi alterado. O trecho de programa que faz esta plotagem é mostrado a seguir figura abaixo.

```

99  # proc = appPIO(appWindowPicture wdl (drawAt {x=0,y=0} fundo)) proc
100 # proc = appPIO (setWindowLook wdl True (False,
101   (lookCria "Petrucchi" 22 "pentagrama" (toInt nCompassos) fCompN fCompD clave armadura))) proc

```

A função **lookCria** é responsável por plotar o pentagrama com o primeiro compasso, com a armadura de clave e fórmula de compasso, conforme figura a seguir.

```

203 lookCria lt tm nt nc fCompN fCompD clave armadura Able {newFrame} picture
204
205 |(nt == "pentagrama")
206 # deltaY = 20
207 # picture = setPenColour Black picture
208 # {(ok, fnt), picture} = openFont {fName = lt, fStyles = ["Bold", "", ""], fSize=24} picture
209 # picture = setPenFont fnt picture
210 //primeiro compasso
211 # picture = drawAt {x= 44,y=94+deltaY} "\\\" picture
212 # picture = drawAt {x= 44,y=94+deltaY} "=" picture
213 # picture = drawAt {x= 77,y=94+deltaY} "=" picture
214 # picture = drawAt {x= 110,y=94+deltaY} "=" picture
215 # picture = drawAt {x= 143,y=94+deltaY} "=" picture
216 # picture = drawAt {x= 176,y=94+deltaY} "=" picture
217 # picture = drawAt {x= 209,y=94+deltaY} "=" picture
218 # picture = drawAt {x= 242,y=94+deltaY} "\\\" picture
219 //plota clave
220 # picture = setPenColour Black picture //cor da letra
221 # {(ok, fnt), picture} = openFont {fName = lt, fStyles = ["Bold", "", ""], fSize=(fonteSizeClave clave)} picture
222 # picture = setPenFont fnt picture
223 # picture = drawAt {x= (xClave clave), y= (yClave clave)+deltaY} (pegaCodigoClave clave) picture |
224 //plota formula de compasso
225 # picture = setPenColour Black picture
226 # {(ok, fnt), picture} = openFont {fName = lt, fStyles = ["Bold", "", ""], fSize=22} picture
227 # picture = setPenFont fnt picture
228 # deltaX = armaduraDeltaX armadura
229 # picture = drawAt {x=88+deltaX, y=69+deltaY} fCompN picture
230 # picture = drawAt {x=88+deltaX, y=85+deltaY} fCompD picture
231
232 =picture
233

```

Na linha 207, escolhe-se a cor do elemento de texto que vai ser colocado na interface

```

207 # picture = setPenColour Black picture

```

Na linha 208, escolhe-se o tipo de fonte, no caso, uma fonte musical⁷⁵, bem como o tamanho e estilo da mesma (tamanho 24 e estilo Bold).

```

208 # {(ok, fnt), picture} = openFont {fName = lt, fStyles = ["Bold", "", ""], fSize=24} picture

```

Para escrever a fonte na janela, deve-se pegar a caneta com os atributos da fonte escolhida.

```

209 # picture = setPenFont fnt picture

```

⁷⁵ Fonte **Petrucchi** – escolhida na linha 101, parâmetro da função **lookCria**.

De posse da caneta, pode-se desenhar os símbolos da fonte em uma determinada região na interface. Inicia-se por colocar uma barra vertical para iniciar o pentagrama. O símbolo de barra (início, fim e separação de compasso) é a string “\”. A função **drawAt** se encarrega de colocar o símbolo nas coordenadas {x,y} especificadas.

```
211      # picture = drawAt {x= 44,y=94+deltaY} "\\" picture
```

As linhas dos pentagramas são sucessões do símbolo “=”. Para montar um compasso, adotou-se um conjunto de 6 desta string “=”, finalizados por outro símbolo de fim de compasso “\”.

```
212      # picture = drawAt {x= 44,y=94+deltaY} "=" picture
213      # picture = drawAt {x= 77,y=94+deltaY} "=" picture
214      # picture = drawAt {x= 110,y=94+deltaY} "=" picture
215      # picture = drawAt {x= 143,y=94+deltaY} "=" picture
216      # picture = drawAt {x= 176,y=94+deltaY} "=" picture
217      # picture = drawAt {x= 209,y=94+deltaY} "=" picture
218      # picture = drawAt {x= 242,y=94+deltaY} "\\" picture
```

A clave é inserida da mesma forma que os elementos de compasso, apenas se escolhe fontes de tamanho diferente para dar uma configuração aderente a um pentagrama musical.

```
219      //plota clave
220      # picture = setPenColour Black picture //cor da letra
221      # ((ok, fnt), picture) = openFont {fName = lt,fStyles = ["Bold","", ""],fSize=(fonteSizeClave clave)} picture
222      # picture = setPenFont fnt picture
223      # picture = drawAt {x= (xClave clave),y= (yClave clave)+deltaY} (pegaCodigoClave clave) picture |
```

Poder-se-ia escolher, nesta etapa, uma cor diferente. Para cada clave escolhida, tem-se um tamanho de fonte diferente e uma posição {x,y} também diferente. Por isto, no código apresentado, chama-se a função **fonteSizeClave**, a qual define o tamanho da letra, e a função **pegaCodigoClave**, a qual define as coordenadas da clave.

```
416 pegaCodigoClave cl = hd [snd(x,y) \ x<- claveLista & y<- claveCar | x == cl]
417
418 xClave "Clave de Sol" = 54
419 xClave "Clave de Fa" = 48
420 xClave "Clave de Do" = 54
421
422 yClave "Clave de Sol" = 85
423 yClave "Clave de Fa" = 70
424 yClave "Clave de Do" = 78
425
426 fonteSizeClave "Clave de Sol" = 26
427 fonteSizeClave "Clave de Fa" = 32
428 fonteSizeClave "Clave de Do" = 24
```

A fórmula de compasso segue o mesmo procedimento da clave. Apenas não se tratou a coordenada {x,y} da fonte conforme a clave escolhida.

```

224 // plota formula de compasso
225 # picture = setPenColour Black picture
226 # ((ok, fnt), picture) = openFont {fName = lt,fStyles = ["Bold","", ""],fSize=22} picture
227 # picture = setPenFont fnt picture

```

A armadura de clave é definida pela tonalidade escolhida. Para cada uma se colocam ou não símbolos “#” (sustenido), cada um com uma coordenada {x,y} específica musicalmente. O Trecho de código que define as coordenadas é mostrado a seguir.

```

228 # deltaX = armaduraDeltaX armadura
229 # picture = drawAt {x=88+deltaX, y=69+deltaY} fCompN picture
230 # picture = drawAt {x=88+deltaX, y=85+deltaY} fCompD picture
231
232 =picture
233
234
235 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
236
237 armaduraDeltaX st
238 |st == "do maior" = 0
239 |st == "sol maior(1#)" = 0
240 |st == "re maior(2#)" = 10
241 |st == "la maior(3#)" = 20
242 |st == "mi maior(4#)" = 30
243 |st == "si maior(5#)" = 40
244 |st == "fa# maior(6#)" = 50
245 |st == "do# maior(7#)" = 60
246 |st == "fa maior(1b)" = 0
247 |st == "sib maior(2b)" = 10
248 |st == "mib maior(3b)" = 20
249 |st == "lab maior(4b)" = 30
250 |st == "reb maior(5b)" = 40
251 |st == "solb maior(6b)" = 50
252 |st == "dob maior(7b)" = 60
253 |otherwise = 0

```

O item que falta para mostrar desta interface é o que apaga o campo de edição do pentagrama. Apagar o pentagrama é disparado pelo botão **Apagar Pentagrama**. Nesta função, apenas se *plota* novamente o bitmap de fundo, e, com isto, a figura com o pentagrama *plotado* anteriormente é sobreposta, como se tivesse apagado a região do pentagrama. Na realidade, poder-se-ia fazer um **look** para apagar a janela, mas, neste caso, ela ficaria totalmente em branco, ou seja, apagaria a figura de fundo, o que não é o desejado. O trecho de código que faz a *replotagem* do fundo (apagar) é mostrado a seguir.

```

132 apagaPentagrama proc =: {ls={fundo}}
133 # proc = appPIO(appWindowPicture wdl (drawAt {x=0,y=0} fundo)) proc
134 = proc
135

```