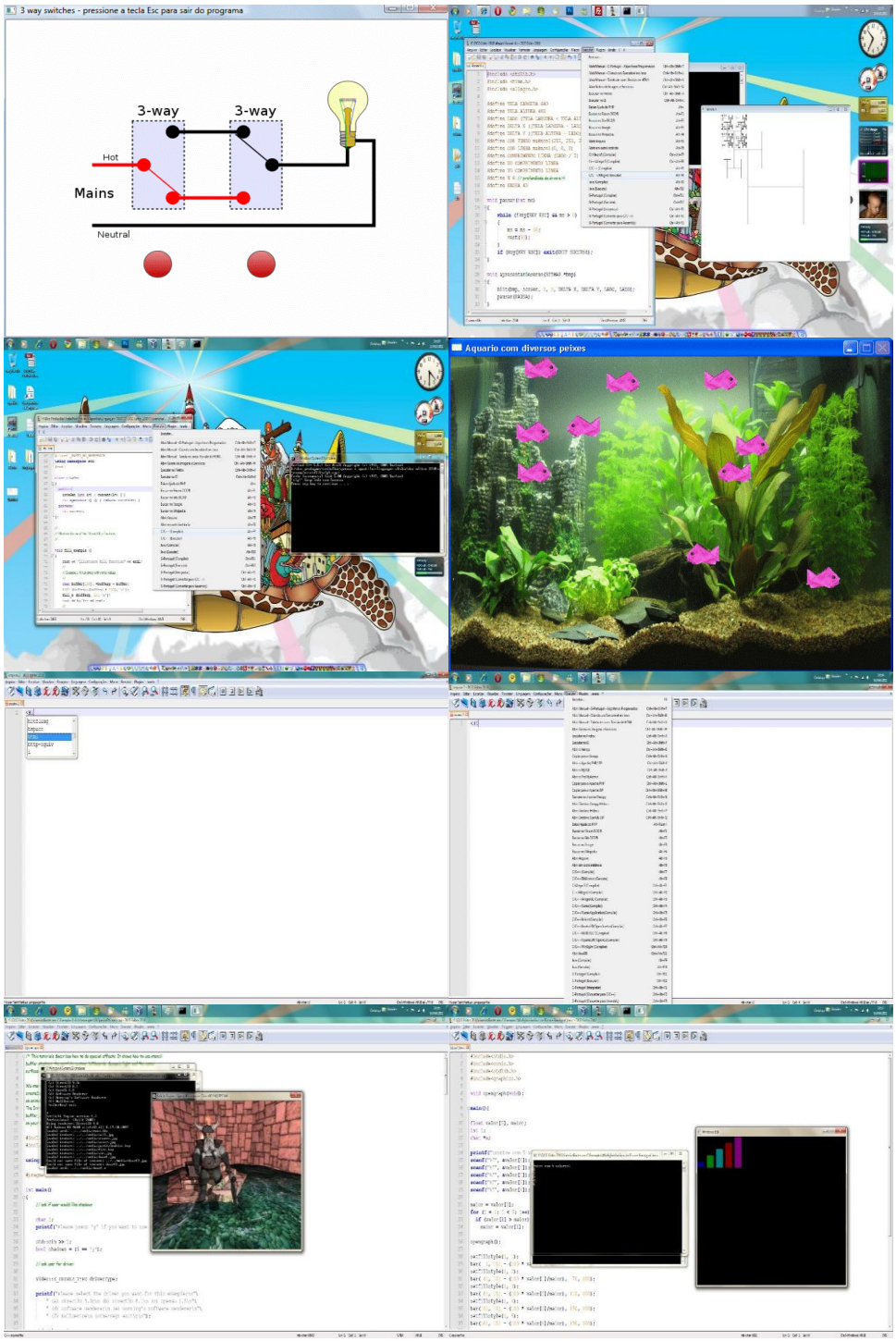
DCO® Editor 2010 - é um compilador e editor de diversas linguagens, entre elas JAVA, C, C++, PHP, ASP, HTML, Algoritmos e muito mais...

Manual Completo para Desenvolvimento Allegro



DCO® Editor 2010 foi criado em cima do Sistema Source Code **Notepad++** com várias modificações.

O Software é livre e inclui manuais, exercícios, resources, templates, texturas, apostilas, as auto instalações de compiladores como JAVA, C e C++ além da galeria de imagens e exercícios para treinamento contidos no Programa de Instalação.

Além de Centenas de Bibliotecas para Linguagens, incluindo Bibliotecas Gráficas de **Criação de Jogos** para C e C++ como **Allegro, OpenGl, DirectX, Irrlicht, Open Invetor, WinBgIm, Nvidia Pro** e muito mais..

Servidores de Banco de Dados como JavaDB e MySQL.

+ Apache 2.2.17
+ MySQL 5.5.8 (Community Server)
+ PHP 5.3.5 (VC6 X86 32bit) + PEAR
+ XAMPP Control Version 2.5 from www.nat32.com
+ XAMPP Security
+ SQLite 2.8.15
+ OpenSSL 0.9.8o
+ phpMyAdmin 3.3.9
+ ADOdb 5.11
+ Mercury Mail Transport System v4.62
+ FileZilla FTP Server 0.9.37
+ Webalizer 2.01-10
+ Zend Optimizer 3.3.0
+ Perl 5.10.1
+ Mod_perl 2.0.4
+ Tomcat 7.0.3

Bibliotecas Instaladas em C e C++

Allegro - 4.4.0.1
Allegro - 4.9.2.8
AllegroGL - OpenGL - 0.2.5
Binary utils (binutils) - 2.15.91
Coin3D - Alto Level 3D - 2.4.4
CONIO - 2.0
Dev-C++ Map file - Geração de Relátórios de Bug - 4.9.9.2
Dev-C++ Help file - Arquivo de Ajuda Dev-C++ - 1.63
GNU Compiler Collection para C - 3.4.2
GNU Compiler Collection para C++ - 3.4.2
GNU Debugger - 5.2.1

Glui GLUT/FreeGLUT e OpenGL Avançado 2.2
GLUT OpenGL Avançado - 7.6
Irrlicht - Engine 3D, suporte a D3d, OpenGl e DirectX. para Linux, Windows, Mac e Android - 1.3
GNU Make - Compilar Gnu para Linux, Windows, Mac e Android - 3.8
MinGW - Biblioteca Gráfica Básica - 3.7
OpenGLUT - Licença Alternativa para OpenGL Avançado 0.6.3
Pdcurses - Desenvolvimento X11, Win32, DOS e OS/2 - 3.2
Windows32 API - Desenvolvimento de APIs para Windows - 3.2
WinBGIm - Plataforma de Gráficos BGI para Windows - 6.0

Entre em contato se tiver dúvidas.

Aqui mesmo no Fórum da Xiglute.com

Ao Efetuar o Pagamento envie nos um e-mail ou ligue e receba de brinde várias apostilas de C, C++, Java, Algoritmos, HTML, CSS e muito mais...

Atenciosamente,

Dário Cândido de Oliveira

{ allegro.cc }
game developing community network

A Biblioteca Gráfica Allegro

1. Introdução

Uma importante área da computação, mas que normalmente é deixada em segundo plano no meio acadêmico, é a de jogos. Os jogos, porém, constituem um ramo com grande potencial de exploração, devido ao uso de diversos elementos da computação em sua produção.

Além dos elementos mais aparentes para um leigo num jogo, como a inteligência artifical (com destaque para os jogos de tabuleiro) e a computação gráfica, outros estão "escondidos", como o de algoritmos (visando a eficiência para respostas mais rápidas pela máquina) e de estruturas de dados (facilitando ao criar uma abstração para os objetos utilizados nos jogos).

Assim, devido à complexidade e à completeza da área de jogos, entendemos que estes são fonte inesgotável de pesquisas e, portanto, devem ter maior foco no meio acadêmico.

O objetivo principal desta apostila, no entanto, é tentar ensinar a usar, pelo menos de maneira básica, a biblioteca Allegro, uma das mais populares ferramentas para confecção de jogos existente. Explicações de algumas técnicas de programação de jogos aparecerão naturalmente como necessidade para contornar determinados problemas e/ou simplificar determinadas tarefas.

Como efeito da programação de jogos utilizada nesta apostila, pretendemos criar uma nova mentalidade a respeito dos mesmos, mostrando a necessidade da pesquisa nessa área. Assim, como segundo objetivo e resultado do primeiro, esta apostila visa introduzir o aluno ao mundo da programação de jogos, que pode, nos próximos anos, tornar-se uma tendência no meio acadêmico, caso o preconceito em relação a ele seja superada.

A apostila está dividida em dois blocos: um didático e outro de referência.

O primeiro bloco, por sua vez, está subdividido em quatro partes: a primeira delas, da qual esta introdução faz parte, apenas ensina como instalar o Allegro e alguns passos básicos a serem tomados antes de começar a programar; a segunda ensina como usar as funções mais importantes do Allegro, sempre com exemplos que ilustrem o uso das mesmas; a terceira ensina algumas técnicas de programação de jogos que foram mostradas durante a segunda parte, ou que sejam interessantes para a confecção de um jogo básico; e a quarta ensina como montar um jogo utilizando as funções mostradas na segunda parte e algumas técnicas utilizadas na terceira.

O segundo bloco contém um guia de referência com as funções do Allegro mais úteis, e deve ser consultado quando as explicações do primeiro bloco não forem profundas o suficiente, ou quando se desejar saber, de forma rápida e eficiente, sobre o funcionamento de determinada função.

---

2. Instalando o Allegro

Devido a restrições de compatibilidade com versões do sistema Windows a instalação com MinGW deve ser a preferida. Para quem usa o sistema Unix simplesmente siga os passos indicados no arquivo da bibliteca allegro.

2.1 Com MINGW

Para instalar o MinGw vá para como conseguir compiladores. Uma vez instalado o compilador é necessário trazer a biblioteca Allegro. Há uma versão compactada em binário para o compilador MinGw no site allegro.cc. Uma vez na página procure pelo link Files e não Downloads como na maioria das páginas. Basta então descompactar o arquivo no diretório onde foi instalado o compilador MinGw. Preste atenção que junto com os arquivos da biblioteca são trazidos três DLLs (all*.dll) que devem ser copiadas para o diretório apropriado do Windows. Estas dlls estarão no diretório principal do MinGw.

2.2 Com DJGPP

Primeiramente, entre no site allegro.cc para fazer o download da versão mais recente e estável. Caso exista algum arquivo de nome `all`**`xyz`**`.zip` (versão **`x.y.z`**) entre os arquivos de instalação do DJGPP/GCC, não será necessário baixá-lo novamente, ao menos que se deseje adquirir uma versão mais recente. A versão mais recente do Allegro, até a última atualização desta página, é a 4.0.2, que pode ser obtida clicando-se diretamente aqui.

Em seguida, crie um subdiretório dentro do diretório do DJGPP/GCC (normalmente `C:\DJGPP\Allegro`) e, então, extraia para este diretório os arquivos do arquivo .ZIP anteriormente mencionado (caso o arquivo .ZIP já possua uma indicação para criação do diretório `Allegro`, extraia o seu conteúdo para o diretório `C:\DJGGP`, por exemplo).

O próximo passo, e talvez o mais importante e (muito) demorado, é a compilação dos arquivos do Allegro, uma vez que o arquivo .ZIP é constituído apenas pelos arquivos-fonte.
Para isto, abra um Prompt do MS-DOS, mude para o diretório do Allegro e, em seguida, digite `make`. Caso ocorra algum erro durante o processo de compilação do Allegro, tente digitar `make all`. Se ainda assim ocorrerem erros, tente executar o make novamente.
Ao final deste processo, digite `make install` para que o Allegro seja instalado de forma correta, fazendo com que o GCC/DJGPP possa utilizá-lo.

Após o make, é recomendável verificar se os arquivos `allegro.h` e `liballeg.a` estão, respectivamente, nos diretórios `C:\DJGPP\Include` e `C:\DJGPP\Lib`. Caso não estejam, copie-os manualmente a partir do diretório do Allegro.

Devido a alguns problemas com o make, resolvemos disponibilizar, também, um link para a biblioteca Allegro pré-compilada. Assim, clique aqui para baixar o arquivo e, depois, apenas extraia o seu conteúdo para o diretório do DJGPP (`C:\DJGPP`). Não é

---

necessário o uso do make.

3. Passos Básicos

Assim como se faz com qualquer outra biblioteca, é necessário incluir o arquivo de cabeçalho do Allegro. Para tal, **APÓS** todas as outras diretivas `#include` das bibliotecas padrão (ex: `stdio.h`, `stdlib.h`), inclua a seguinte linha:

```
#include <allegro.h>
```

O próximo passo é fazer com que o compilador C possa linkar corretamente a biblioteca Allegro (liballeg.a).
Para tanto, os usuários do RHIDE devem ir ao menu Options->Libraries, adicionar a palavra alleg na primeira linha em branco e marcar o quadrado ao lado. Já para os usuários do GCC, é necessário adicionar o parâmetro -lalleg à linha de comando. Por exemplo:

```
gcc -o jogo.exe jogo.c -Wall -lalleg
```

Programando com o Allegro

1. Esquema de um Programa

A maioria dos programas feitos usando o Allegro segue um certo esquema de montagem. Veremos, a seguir, esse esquema.

1.1. Configurações Iniciais

Todo programa que usa o Allegro deve ter, antes da chamada de qualquer outra função do Allegro, uma chamada a função `allegro_init`. Ela inicializa algumas variáveis e reserva memória para algumas operações do Allegro, deixando-o pronto para o uso. Esta função não possui parâmetros, devendo ser chamada da seguinte forma:

```
allegro_init();
```

Apás chamada a função `allegro_init`, são chamadas as funções `install_keyboard`, `install_mouse` e `install_timer`, que inicializam, respectivamente, o teclado, o

mouse e os temporizadores. Elas também não possuem parâmetros e são chamadas assim:

```
install_keyboard();
install_mouse();
install_timer();
```

Normalmente não é necessário checar o valor de retorno dessas funções, uma vez que são raros os casos em que a inicialização de um desses componentes falha. Porém, para projetos maiores, é recomendado que sempre sejam checados os valores de retorno dessas e de outras funções.

Ao final do programa, utilizamos a função `allegro_exit` para retirar os domínios do Allegro sobre o computador. A função, que também não possui parâmetros, é chamada da seguinte forma:

```
allegro_exit();
```

1.2. Configuração do Modo Gráfico

Existem duas funções para a configuração do modo gráfico: `set_color_depth` e `set_gfx_mode`.

A primeira, que deve ser chamada antes da segunda, determina o número de bits de cores a ser usado pelos gráficos. O número de bits pode ser 8 (256 cores), 15 (32768), 16 (65536), 24 (aproximadamente 32 milhões) e 32 (aproximadamente 4 bilhões de cores), sendo este o único parâmetro da função, a ser chamada, por exemplo, da seguinte forma:

```
set_color_depth(16);
```

Caso não seja chamada esta função, o padrão para o número de bits é 8.

Já a segunda função, `set_gfx_mode`, é responsável pela inicialização do modo gráfico. O primeiro parâmetro representa o driver gráfico a ser utilizado pelo Allegro, e que deve ser uma das constantes definidas pelo Allegro. Os segundo e terceiro parâmetros indicam, respectivamente, o tamanho horizontal e vertical da tela, em pixels. Assim, para inicializar o modo gráfico com uma resolução de 640x480, deveríamos chamar a função da seguinte maneira:

```
set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0);
```

Os quarto e quinto parâmetros, ignorados no exemplo anterior ao atribuirmos o valor 0 (zero), indicam a resolução de uma possível tela virtual. Isto é utilizado em alguns jogos em que apenas uma parte (tela física) de um mapa (tela virtual), por exemplo, pode ser visualizada. Podemos, então, chamar a função da seguinte maneira:

```
set_gfx_mode(GFX_VESA1, 640, 480, 3200, 2400);
```

Com isso, a tela física continuará com uma resolução de 640x480; porém, através de funções de *scroll* do Allegro, poderemos visualizar uma parte de uma tela virtual de 3200x2400 (armazenada na memória) naquela tela física.

Ao contrário do que acontece com as funções vistas anteriormente, é interessante verificar o valor de retorno desta função, pois é comum os casos em que não é possível inicializar o modo gráfico, pelo não suporte do driver de vídeo ou da resolução pelo hardware. Cabe ao programador tentar "driblar" esses problemas.

1.3. Configuração do Som

Existe apenas uma função de configuração do som no Allegro, que inicializa tanto os dispositivos digitais quanto os dispositivos MIDI. Esta função, `install_sound`, possui três parâmetros: o primeiro indica o controlador de som digital a ser usado pelo Allegro; o segundo, o controlador de som MIDI; e o terceiro existe apenas por motivos de compatibilidade com versões antigas do Allegro, e deve ser ignorado passando-se o valor **`NULL`**. Assim, para inicializarmos o som para ser usado com o Allegro, chamamos a função da seguinte forma:

```
install_sound(DIGI_SB, MIDI_SB_OUT, NULL);
```

O exemplo acima inicializa dispositivos de som digital e MIDI do tipo Sound Blaster. Se quisermos que o Allegro escolha automaticamente qual o controlador de som mais apropriado, podemos chamar a função da seguinte maneira:

```
install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, NULL);
```

Assim como acontece com a inicialização do modo gráfico, a inicialização do som também está sujeita a falhas, pois o controlador de som escolhido pode não ser compatível com o hardware ou o Allegro pode não ser capaz de escolher automaticamente o controlador correto. Com isso, é interessante checar o valor de retorno desta função, com o intuito de tentar escolher o controlador mais adequado para o hardware existente.

1.4. Esqueleto de um Programa

Com as funções vistas anteriormente, podemos criar uma base para montar programas que usem o Allegro. Chamaremos esta base de "esqueleto" de um programa. Abaixo, vemos o código do esqueleto.

```
#include <stdio.h>
#include <stdlib.h>
#include <allegro.h>

#define MAX_X         800
#define MAX_Y         600
#define V_MAX_X       0
#define V_MAX_Y       0
#define COLOR_BITS    8
#define VIDEO_CARD    GFX_VESA1
```

```
#define DIGI_CARD     DIGI_SB16
#define MIDI_CARD     MIDI_SB_OUT

int inicia(void);
void principal(void);
void finaliza(void);

int main (void)
{

      if (!inicia())
      {

            finaliza();
            exit(-1);

      }

      principal();

      finaliza();
      exit(0);

}

int inicia (void)
{

      allegro_init();

      install_keyboard();
      install_mouse();
      install_timer();

      set_color_depth(COLOR_BITS);
      if (set_gfx_mode(VIDEO_CARD, MAX_X, MAX_Y, V_MAX_X,
      V_MAX_Y) < 0)
      {

            if (set_gfx_mode(GFX_AUTODETECT, MAX_X, MAX_Y,
            V_MAX_X, V_MAX_Y) < 0)
            {

                  printf("Erro ao inicializar o modo grafico");
                  return (FALSE);

            }

      }

      if (install_sound(DIGI_CARD, MIDI_CARD, NULL) < 0)
      {

            if (install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT,
            NULL) < 0)
            {
```

```
                    printf("Erro ao inicializar o som");
                    return (FALSE);

                }

        }

        return (TRUE);

}

void principal (void)
{



}

void finaliza (void)
{

        allegro_exit();

}
```

2. Programando com Exemplos

Veremos, agora, as principais funções da biblioteca Allegro, utilizando alguns exemplos para melhor explicá-las. Em alguns casos, apenas a descrição da função e um breve exemplo não serão suficientes; nestes casos, daremos programas completos para exemplificar o uso da função ou de alguma técnica de programação, partindo sempre da função `principal` do esqueleto de programa.

2.1. Os Gráficos

Tendo em vista que a principal característica do Allegro é o uso da parte gráfica, o Allegro dispõe de diversas funções de desenho. Porém, antes de explicá-las, é necessário entender o modo como o Allegro trata a parte gráfica.

2.1.1. Bitmaps

O Allegro trata todo gráfico que pode ser guardado na memória ou desenhado na tela como um bitmap, que nada mais é do que um modo de representar gráficos em formato binário. Para tal, o Allegro define um tipo `BITMAP`, com o qual o usuário pode manipular facilmente esses bitmaps. Não é necessário, para o programador, conhecer os elementos desta estrutura; basta saber utilizar as funções de bitmaps.

Se quiséssemos, por exemplo, usar um bitmap chamado `quadro`, poderíamos declará-lo da seguinte forma:

```
BITMAP *quadro;
```

Antes de usar um bitmap é necessário criá-lo, pois apenas a declaração do bitmap não é suficiente para o uso do mesmo. Para isso, usamos a função `create_bitmap`, da seguinte forma:

```
quadro = create_bitmap(50, 25);
```

Isto criará um bitmap de largura 50 e altura 25, e que será referenciado pela variável `quadro`.

Após a criação do bitmap, podemos utilizar as várias funções de bitmaps que o Allegro oferece. Se quiséssemos, por exemplo, desenhar um retângulo no bitmap `quadro`, poderíamos usar a função `rect`, como abaixo:

```
rect(quadro, 10, 5, 40, 20, 0);
```

Este exemplo desenhará um retângulo desde a coordenada (10, 5) até a coordenada (40, 20), dentro do bitmap `quadro`, utilizando a cor de código 0.

Porém, tudo o que foi feito até aqui com o bitmap `quadro` ocorreu na memória. Em nenhum momento o bitmap foi mostrado na tela. Se quiséssemos, agora, colocar o bitmap na tela, poderíamos utilizar a função `blit`. Ela copia uma parte de um bitmap para dentro de outro bitmap. Além da função `blit`, utilizamos a variável `screen`, que também é do tipo `BITMAP` e é pré-definida pelo Allegro. Assim, a cópia do bitmap `quadro` para a tela ficaria da seguinte forma:

```
blit(quadro, screen, 0, 0, 200, 100, 50, 25);
```

Isto copiará uma parte do bitmap `quadro` começando na coordenada (0, 0) e de tamanho 50 x 25 (ou seja, todo o bitmap), para o bitmap `screen`, que nada mais é do que a forma do Allegro representar a tela, através de uma variável do tipo `BITMAP`.

Poderíamos, neste caso, ter usado a função `draw_sprite`, que copia um bitmap inteiro para dentro de outro. Assim, a sua chamada ficaria da seguinte forma:

```
draw_sprite(screen, quadro, 200, 100);
```

Isto é equivalente à chamada da função `blit` feita anteriormente, com a diferença de que os pontos do bitmap `quadro` que possuem código 0 não desenhados, funcionando como pontos transparentes.

Podemos, ainda, utilizar as funções de desenho do Allegro diretamente no bitmap `screen`. Assim, se quiséssemos, por exemplo, desenhar um círculo preenchido na tela, poderíamos simplesmente escrever

```
circlefill(screen, 300, 200, 40, 15);
```

o que desenharia, na tela, um círculo de raio 40, com centro na coordenada (300, 200) e totalmente preenchido com a cor de código 15.

Às vezes, porém, torna-se inviável desenhar, durante a execução do programa, todos os gráficos necessários. Para esses casos, podemos utilizar a função `load_bitmap`, que

carrega um bitmap diretamente de um arquivo, fazendo com que seja possível a confecção de grande parte dos gráficos do jogo antes de sua execução. Assim, se quiséssemos, por exemplo, carregar um bitmap de um arquivo com nome `figura.bmp`, guardá-lo na memória e referenciá-lo através da variavél `quadro`, deveríamos utilizar a função da seguinte maneira:

```
quadro = load_bitmap("figura.bmp", NULL);
```

Com isso, o bitmap `quadro` fica disponível para que sejam realizadas todas as operações disponíveis para bitmaps, inclusive o uso das funções `blit` e `draw_sprite`, para mostrá-lo na tela.

2.1.2. Paletas

Quando trabalhamos com bitmaps de 256 cores, temos alguns problemas com a apresentação dos mesmos na tela. Isso se deve ao fato de que, com 8 bits de cores, há uma certa dificuldade em representar um grande número delas. Para contornar tal dificuldade, foram criadas paletas, que nada mais são do que tabelas de cores, com um índice que varia de 0 a 255. Assim, ao invés de cada código de cor representar **sempre** a mesma cor (o que acontece nos modos de 15, 16, 24 e 32 bits), temos cores que variam de acordo com a paleta em uso. Podemos ter, por exemplo, uma paleta com 256 tons de vermelho diferentes, ou outra com 128 tons de amarelo e 128 tons de azul, dependendo dos números guardados dentro de cada código de cor (três números são guardados dentro de cada código de cor, representando as quantidades de vermelho, verde e azul existentes naquela cor).

Podemos, então, definir uma paleta como sendo um vetor de 256 posições, no qual cada posição corresponde a um código de cor, e em cada uma delas temos três números para representar as quantidades de vermelho, verde e azul existentes na cor correspondente.

Assim como para os bitmaps, existe também, no Allegro, um tipo para paletas. Podemos, então, criar uma paleta usando a seguinte declaração:

```
PALETTE paleta;
```

Existem duas maneiras de se modificar uma paleta: a primeira é fazer isso manualmente, utilizando os códigos de cor e as quantidades de vermelho, verde e azul desejadas; a segunda, e, naturalmente, menos trabalhosa, é carregar uma paleta correspondente a determinado bitmap durante a leitura deste.

Na primeira maneira, modificamos os valores da estrutura `PALETTE` diretamente. Por exemplo:

```
paleta[15].r = 0;
paleta[15].g = 63;
paleta[15].b = 0;
```

modificaria o código 15 de `paleta` para o verde mais claro possível, uma vez que as quantidades de vermelho, verde e azul de cada código variam de 0 a 63 (6 bits). Note que as letras `r`, `g` e `b` correspondem às cores vermelha, verde e azul, respectivamente.

Na segunda maneira, usamos a função `load_bitmap` para carregar a paleta junto com o bitmap. Assim,

```
quadro = load_bitmap("figura.bmp", paleta);
```

carregaria o bitmap do arquivo `figura.bmp` para `quadro` e, também, a paleta correspondente a esse bitmap em `paleta`.

Após definirmos as cores desejadas de uma paleta (seja pelo primeiro ou pelo segundo método), podemos torná-la ativa usando a função `set_palette`, da seguinte maneira:

```
set_palette(paleta);
```

Isto faz com que **todas** as cores da tela mudem para as cores correspondentes ao código de cor de cada ponto.

Podemos usar isto para fazer algumas animações sem que precisemos redesenhar os gráficos. O exemplo abaixo mostra como fazer isso.

```
void principal (void)
{
	int i;
	int cor;
	PALETTE pal;

	/* Limpa a tela (para o codigo de cor 0). */
	clear(screen);

	/*

		Torna ativa a black_palette, paleta pre-definida
		pelo
		Allegro que contem apenas a cor preta, para todos os
		codigos.
		Fazemos isso para que os retangulos feitos a seguir
		nao sejam
		vistos ao serem desenhados.

	*/
	set_palette(black_palette);

	/*

		Desenha retangulos preenchidos, cada um com codigos
		de cor variando de 0 a 63, em sequencia.

	*/
	for (i = 0; i < 64; i++)
	{

		rectfill(screen, i * 12.5, 0, (i + 1) * 12.5, 599,
		i);
```

```
        }

        /* Gera tons de azuis */
        for (i = 0; i < 64; i++)
        {

              pal[i].r = 0;
              pal[i].g = 0;
              pal[i].b = i;

        }

        while (!keypressed())
        {

              /* Torna ativa a paleta pal */
              set_palette(pal);

              /*

                    Troca os valores de cada codigo de cor,
                    causando um efeito de animacao.

              */
              cor = pal[63].b;
              for (i = 63; i > 0; i--)
              {

                    pal[i].b = pal[i - 1].b;

              }
              pal[0].b = cor;

        }

  }
```

Existem outras duas funções que são muito utilizadas pelos programadores Allegro: `fade_in` e `fade_out`.

A primeira faz com que, partindo da `black_palette`, as cores sejam alteradas aos poucos para uma determinada paleta. Isto causa um ótimo efeito visual, pois as cores vão aparecendo gradativamente. Da mesma maneira, a segunda faz o efeito inverso, ou seja, as cores vão desaparecendo gradativamente. Por exemplo:

```
tela = load_bitmap("figura.bmp", paleta);

set_palette(black_palette);
draw_sprite(screen, tela, 20, 30);

fade_in(paleta, 4);
while (!keypressed());
fade_out(32);
```

O exemplo acima desenha o bitmap `tela` com cores pretas para que a função `fade_in` faça-o aparecer gradativamente, até alcançar as cores da paleta `paleta`. Quando o usuário pressionar alguma tecla, a função `fade_out` fará o bitmap desaparecer gradativamente. Os valores 4 e 32 utilizados nas funções `fade_in` e `fade_out` controlam a velocidade com que as cores são alteradas, podendo ir de 1 (devagar) até 64 (muito rápido).

2.1.3. Textos

Além das funções de desenho vistas anteriormente, o Allegro dispõe de algumas funções para o desenho de textos na tela.

Antes de utilizarmos as funções que desenham os textos, devemos definir o modo como os caracteres serão desenhados. Para tal, usamos a função `text_mode`. O seu único parâmetro define o código de cor que terá o fundo dos caracteres. Caso o código de cor seja menor do que zero, a fundo será transparente. Assim

```
text_mode(15);
```

faria com que o fundo do texto que fosse desenhado **posteriormente** tivesse o código de cor 15. Porém, se tivéssemos

```
text_mode(-1);
```

o fundo seria transparente.

Para desenhar os textos, podemos utilizar quatro funções diferentes: `textout`, `textout_centre`, `textprinft` e `textprintf_centre`. A primeira, `textout`, desenha um texto qualquer num determinado bitmap, como a seguir:

```
textout(screen, font, "O Jogo Acabou", 50, 100, 20);
```

O exemplo anterior escreve a string "O Jogo Acabou" na coordenada (50, 100) da tela (`screen`), utilizando o código de cor 20. A função `textout_centre` teria o mesmo efeito, a não ser pelo fato de que na coordenada (50, 100) seria posicionado o centro da frase "O Jogo Acabou", o que faria o texto ficar centralizado.

Já a função `textprintf` escreve o texto utilizando um formato (da mesma forma que a função `printf` da biblioteca `stdio`). Assim, poderíamos escrever

```
textprintf(tela, font, 400, 100, 15, "Pontos: %d", pontos);
```

o que desenharia, na coordenada (400, 100) do bitmap `tela`, utilizando o código de cor 15, a string "Pontos: %d", utilizando o mesmo formato da função `printf`. Assim, se a variável `pontos` do tipo `int` tivesse o valor 2000, seria desenhada a frase "Pontos: 2000" no bitmap `tela`. Assim como nas funções `textout` e `textout_centre`, a função `textprintf_centre` faz o mesmo que a função `textprintf`, apenas centralizando o texto na coordenada pedida.

Apesar de não serem o "forte" do Allegro, as funções desempenham o seu papel de forma satisfatória.

O modo como o Allegro desenha cada caractere, porém, pode ser diferente em alguns jogos. Para tratar esta diferença de desenho, o Allegro possui o tipo `FONT`. Assim, para definir uma variável com o tipo `FONT`, escrevemos

```
FONT *fonte;
```

É muito difícil, porém, construir fontes manualmente. Para isso, existem alguns programas que podem ser utilizados, por exemplo, para transformar fontes do Windows em fontes utilizáveis pelo Allegro.

Com a fonte já produzida e carregada na memória, podemos utilizar as funções vistas anteriormente normalmente, apenas alterando o parâmetro `font` (que nada mais é do que uma fonte pré-definida pelo Allegro) para a fonte que queremos. Assim, poderíamos utilizar a fonte `fonte` da seguinte forma:

```
textout(screen, fonte, "Exemplo de texto com outra fonte", 200,
200, 50);
```

2.2. O Teclado

Há duas maneiras de ler a entrada usando o Allegro (uma vez que, após o uso da função `install_keyboard` não é mais possível utilizar as funções de entrada padrões), as duas muito simples, e que devem ser escolhidas de acordo com o que se deseja ler.

A primeira maneira utiliza a função `readkey`, que é semelhante à função `getch` da biblioteca `conio`. O valor retornado pela função possui 16 bits, sendo os menos significativos relativos ao código ASCII e os mais significativos chamados de scancode. Assim, podemos testar diversas teclas (ou combinações delas) das seguintes formas:

```
/* Pelo codigo ASCII */
if ((readkey() & 0xFF) == 'd')

      printf("Voce pressionou a tecla 'd'\n");



/* Pelo scancode */
if ((readkey() >> 8) == KEY_SPACE)

      printf("Voce pressionou a tecla Espaço\n");



/* Pressionando CTRL+[letra] */
if ((readkey() & 0xFF) == 3)

      printf("Voce pressionou CTRL+C\n");
```

```
/* Pressionando ALT+[letra] */
if (readkey() == (KEY_X << 8))

      printf("Voce pressionou ALT+X\n");
```

A segunda maneira utiliza o vetor `key`, definido pelo Allegro. Assim, podemos testar o pressionamento de teclas de modo mais dinâmico. Por exemplo:

```
if (key[KEY_D])

      printf("Voce pressionou a tecla 'D'\n");


if (key[KEY_SPACE])

      printf("Voce pressionou a tecla Espaço\n");


if ((key[KEY_CTRL]) && (key[KEY_C]))

      printf("Voce pressionou CTRL+C\n");


if ((key[KEY_ALT] && (key[KEY_X]))

      printf("Voce pressionou ALT+X\n");
```

Naturalmente, devemos escolher a opção que seja mais simples para o que se quer fazer. Se quisermos, por exemplo, ler o nome do jogador, é preferível utilizar a função `readkey`, que dá a possibilidade de ler vários caracteres diferentes. Porém, se quisermos apenas verificar, por exemplo, se a tecla Enter está pressionada, utilizamos `key[KEY_ENTER]`.

Além desses elementos para o uso do teclado, existem outras duas funções muito úteis para o uso do mesmo: `clear_keybuf` e `keypressed`.

A primeira limpa o *buffer* do teclado, isto é, apaga das possíveis teclas lidas aquelas que foram pressionadas até o momento. Isto é particularmente útil quando, por exemplo, queremos ler teclas apenas a partir de um determinado momento, ignorando as que já foram pressionadas anteriormente e que estão esperando serem lidas. Como a função não possui parâmetros, ela deve ser chamada da seguinte forma:

```
clear_keybuf();
```

A segunda verifica se há alguma tecla pressionada no momento, da mesma maneira que a função `kbhit` da biblioteca `conio`. Assim, podemos fazer um esquema para o *loop* principal de um jogo, de maneira que o programa não páre à espera de uma tecla:

```
while (!key[KEY_ESC])
{

      /* alguns comandos */
      (...)

      if (keypressed())
      {

            tecla = readkey();
            /* reage a tecla */
            (...)

      }

      /* outros comandos */
      (...)

}
```

2.3. O Mouse

Além do teclado, talvez o mouse seja um dos módulos do Allegro mais fáceis de utilizar. O Allegro dispõe de algumas funções e variáveis pré-definidas que auxiliam nessa utilização.

As variáveis `mouse_x` e `mouse_y` guardam, respectivamente, as atuais posições **`x`** e **`y`** do ponteiro do mouse.

A variável `mouse_b` indica qual(is) o(s) botão(ões) do mouse está(ão) sendo pressionado(s). Por exemplo:

```
if (mouse_b & 1)

      printf("Botão esquerdo do mouse pressionado\n");


if (!(mouse_b & 2))

      printf("Botão direito do mouse não pressionado\n");


if (mouse_b & 4)

      printf("Botão do meio do mouse pressionado\n");
```

Além dessas variáveis, a principal função para uso do mouse é a `show_mouse`. Ela faz com que o ponteiro do mouse seja mostrado num determinado bitmap (que pode, inclusive, ser o `screen`). Assim, se escrevermos

```
show_mouse(screen);
```

o ponteiro do mouse será mostrado na tela. Se fizermos

```
show_mouse(NULL);
```

o ponteiro do mouse é escondido.

Há, porém, diversos problemas relacionados à função `show_mouse`. Desenhar o mouse diretamente na tela pode causar manchas devido ao retraço. Se fizéssemos o mouse desaparecer, desenhássemos o que fosse preciso, e reaparecêssemos com o mouse, isto faria o mouse ficar piscando, causando um efeito visual nada interessante. A solução é utilizar a técnica chamada double buffering.

2.4. O Som

O Allegro aceita uma boa variedade de arquivos de som, cobrindo o necessário para a confecção de qualquer jogo. Há, porém, dois tipos distintos de som: os MIDIs e os Samples.

2.4.1. MIDIs

Os MIDIs são responsáveis pela música de fundo dos jogos. Seguindo o padrão do Allegro, existe um tipo `MIDI` para guardar arquivos de música. Podemos, então, declarar um `MIDI` da seguinte forma:

```
MIDI *musica;
```

Como não podemos criar MIDIs durante a execução do jogo, temos que carregá-los a partir de arquivos. Para isso, usamos a função `load_midi`, da seguinte forma:

```
musica = load_midi("musica.midi");
```

Isto carregará o arquivo `musica.midi` para a memória, que será apontada pela variável `musica`, do tipo `MIDI`.

Após carregar o arquivo na memória, podemos tocá-lo utilizando a função `play_midi`. Por exemplo:

```
play_midi(musica, TRUE);
```

tocará o MIDI que está guardado na posição de memória apontada por `musica`. O segundo argumento diz se o MIDI será tocado com *loop*, ou seja, ao chegar ao final do arquivo, volta a

tocá-lo do começo. Neste caso, o argumento é `TRUE` e, portanto, será tocado em *loop*. Caso quiséssemos que o arquivo fosse tocado apenas uma vez, o argumento deveria ser `FALSE`.

Temos duas opção para parar de tocar um MIDI. Uma delas utiliza a função `play_midi` da seguinte forma:

```
play_midi(NULL, FALSE);
```

A outra, utiliza a função `stop_midi`, que não possui argumentos. Assim, se quisermos fazer um MIDI parar de tocar, escrevemos:

```
stop_midi();
```

2.4.2. Samples

Da mesma forma que os MIDIs, os samples possuem seu próprio tipo. Assim, se quisermos definir um arquivo sample, podemos fazê-lo da seguinte forma:

```
SAMPLE *som;
```

Ainda seguindo o que foi visto anteriormente para os MIDIs, precisamos carregar para a memória um arquivo de sample antes de poder tocá-lo. Para isso, utilizamos a função `load_sample`, da seguinte forma:

```
som = load_sample("certo.wav");
```

A chamada de função acima carregará o arquivo `certo.wav` na memória e apontará este endereço de memória com a variável `som`, do tipo `SAMPLE`.

Para tocar o arquivo carregado na memória pela função `load_sample`, utilizamos a função `play_sample`. Ela possui vários argumentos, e pode ser chamada, por exemplo, da seguinte forma:

```
play_sample(som, 255, 128, 1000, FALSE);
```

o que fará com que o sample guardado em `som` seja tocado. O segundo argumento indica o volume com o qual o sample deve ser tocado, e pode variar de 0 a 255. O terceiro argumento indica o balanço das caixas de som, podendo variar, tabmém, de 0 a 255, sendo o 0 para o balanço todo para um lado, e 255 para o outro; neste caso, ao utilizarmos 128, definimos um balanço igualmente distribuído. O quarto argumento indica a frequência com a qual o sample irá ser tocado: 1000 para a frequência normal, 500 para a metade, 2000 para o dobro, e assim por diante. Já o último argumento indica, assim como na função `play_midi`, se o sample será tocado em *loop*.

Seguindo o padrão load-play-stop, temos a função `stop_sample`, que é usada, naturalmente, para fazer um sample parar de tocar. Assim

```
stop_sample();
```

faria o sample que está sendo tocado atualmente parar de tocar.

Técnicas de Programação de Jogos

1. Double Buffering

Quando vamos fazer animações usando o Allegro, surgem alguns problemas relacionados aos vários métodos que podem ser utilizados. O método mais simples que podemos imaginar é aquele em que limpamos a tela, desenhamos os objetos, limpamos a tela novamente, desenhamos os objetos nas novas posições, e assim por diante. Este método, porém, tem um grave problema: a tela pisca a cada limpeza.

Para contornar este tipo de problema, existem várias técnicas de animação. Veremos a mais popular delas, o double buffering.

Podemos concluir, a partir do nome, como funciona esta técnica. Dispomos de um bitmap auxilixar (chamado de *buffer*) que, normalmente, possui o tamanho da tela (ou o tamanho da região onde ocorre a animação). Desenhamos, neste *buffer*, os objetos que devem ser apresentados na tela. Após isso, desenhamos o conteúdo do *buffer* na tela, fazendo com que os objetos apareçam. Limpamos, então, o *buffer*, desenhamos os objetos novamente em suas novas posições, passamos o conteúdo do *buffer* para a tela, e assim por diante.

Esta técnica, além de contornar o problema da tela piscando, é popular pela sua velocidade (pois as outras técnicas que existem são, normalmente, mais lentas) e pela facilidade de implementação. O exemplo abaixo mostra a diferença entre o método primário e o double buffering.

```
void principal (void)
{

      BITMAP *buf;
      int x, y;
      int dx, dy;

      buf = create_bitmap(MAX_X, MAX_Y);

      x = 100;
      y = 100;
      dx = 2;
      dy = 2;

      set_palette(desktop_palette);

      while (!keypressed())
      {

            clear(screen);
            textout(screen, font, "Sem Double Buffering
            (pressione uma tecla para continuar)", 0, 0, -1);
            rectfill(screen, x, y, x + 40, y + 40, 4);
            x += dx;
            y += dy;
```

```
            if ((x < 2) || (x > (MAX_X - 42)))
            {

                  dx = -dx;

            }
            if ((y < 2) || (y > (MAX_Y - 42)))
            {

                  dy = -dy;

            }
            rest(10);

      }
      readkey();

      while (!keypressed())
      {

            clear(buf);
            textout(buf, font, "Com Double Buffering (pressione
            uma tecla para continuar)", 0, 0, -1);
            rectfill(buf, x, y, x + 40, y + 40, 4);
            blit(buf, screen, 0, 0, 0, 0, MAX_X, MAX_Y);
            x += dx;
            y += dy;
            if ((x < 2) || (x > (MAX_X - 42)))
            {

                  dx = -dx;

            }
            if ((y < 2) || (y > (MAX_Y - 42)))
            {

                  dy = -dy;

            }
            rest(10);

      }
      readkey();

}
```

1.1. Mouse

Um dos grandes problemas do Allegro reside no uso do mouse. Se, ao utilizarmos a função `show_mouse`, desenhássemos o mouse diretamente na tela, poderiam aparecer manchas devido ao retraço. Utilizar a técnica do desenha-apaga-desenha também não é uma boa idéia, pois o mouse ficaria piscando. Assim, a melhor solução para este problema é utilizar o double buffering.

Para isso, utilizamos como *buffer* o mesmo bitmap que já é utilizado como tal pelo programa. Assim, DEPOIS de termos desenhado tudo que é necessário, utilizamos a função `show_mouse`, com destino para o bitmap de *buffer*. Quando formos redesenhar o que for necessário, utilizamos a função `show_mouse` com parâmetro `NULL`, para fazer o mouse desaparecer.

O programa abaixo exemplifica estes procedimentos, fazendo o mouse aparecer e se deslocar normalmente enquanto retângulos de cores e posições aleatórias são desenhados na tela.

```
void principal (void)
{

      BITMAP *tela;
      BITMAP *retang;
      int x, y, w, h, c;

      tela = create_bitmap(MAX_X, MAX_Y);
      retang = create_bitmap(MAX_X, MAX_Y);

      clear(retang);

      set_palette(desktop_palette);

      srand(time(0));

      while (!keypressed())
      {

            x = rand() % MAX_X;
            y = rand() % MAX_Y;
            w = rand() % (MAX_X - x);
            h = rand() % (MAX_Y - y);
            c = rand() % (1 << COLOR_BITS);
            rectfill(retang, x, y, x + w, y + h, c);
            show_mouse(NULL);
            clear(tela);
            blit(retang, tela, 0, 0, 0, 0, MAX_X, MAX_Y);
            show_mouse(tela);
            blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);

      }
      readkey();


}
```

2. Scrolling

Uma das principais técnicas utilizadas em jogos (principalmente nos que possuem movimento de cenário) é o scrolling. O scrolling consiste em movimentar o fundo do cenário e, normalmente, deixar o personagem controlado parado, o que causa uma sensação de movimento.

O scrolling pode ser horizontal, vertical ou em ambas as direções. O exemplo abaixo demonstra como utilizar o scrolling, desenhado um boneco (parado) no meio da tela, enquanto o fundo se move de baixo para cima, fazendo com que tenhamos a sensação de que o boneco está caindo.

```
void principal (void)
{

	BITMAP *tela;
	BITMAP *fundo;
	BITMAP *boneco;
	PALETTE pal;
	int fx, fy;
	int x, y;
	int i;

	tela = create_bitmap(MAX_X, MAX_Y);
	fundo = create_bitmap(MAX_X, MAX_Y);

	boneco = load_bitmap("boneco.bmp", pal);

	clear(fundo);
	for (i = 0; i < 100; i++)
	{

		x = rand() % MAX_X;
		y = rand() % MAX_Y;
		putpixel(fundo, x, y, 255);

	}
	fx = fy = 0;

	while (!keypressed())
	{

		clear(tela);
		blit(fundo, tela, 0, 0, fx, fy, MAX_X, MAX_Y);
		blit(fundo, tela, 0, 0, fx, fy - MAX_X, MAX_X,
		MAX_Y);
		draw_sprite(tela, boneco, ((MAX_X) - boneco->w) / 2,
		((MAX_Y) - boneco->h) / 2);
		blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);
		fy -= 3;
		if (fy < 0)
		{
```

```
                fy += MAX_Y;

            }

        }
        readkey();

}
```

2.1. Parallax Scrolling

Um tipo especial de scrolling que é muito utilizado é o parallax scrolling. Ele consite num scrolling em que vários fundos se movimentam em velocidades diferentes, causando uma sensação de profundidade. O exemplo abaixo demonstra um parallax scrolling horizontal.

```
#define FUNDOS      4

(...)

void principal (void)
{

        BITMAP *fundo[FUNDOS];
        BITMAP *tela;
        PALETTE pal;
        int fx[FUNDOS];
        int x, y;
        int i, j;

        srand(time(0));

        for (i = 0; i < FUNDOS; i++)
        {

            fundo[i] = create_bitmap(MAX_X, MAX_Y);
            fx[i] = 0;
            clear(fundo[i]);
            for (j = 0; j < (FUNDOS - i) * 30; j++)
            {

                x = rand() % MAX_X;
                y = rand() % MAX_Y;
                rectfill(fundo[i], x, y, x + i, y + i, 1);

            }

        }

        tela = create_bitmap(MAX_X, MAX_Y);

        pal[0].r = 0;
        pal[0].g = 0;
```

```
        pal[0].b = 0;
        pal[1].r = 63;
        pal[1].g = 63;
        pal[1].b = 63;
        set_palette(pal);

        while (!keypressed())
        {

              clear(tela);
              for (i = 0; i < FUNDOS; i++)
              {

                    draw_sprite(tela, fundo[i], fx[i], 0);
                    draw_sprite(tela, fundo[i], fx[i] - MAX_X, 0);
                    fx[i] += (i + 1) * 2;
                    if (fx[i] > MAX_X)
                    {

                          fx[i] -= MAX_X;

                    }

              }
              blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);

        }
        readkey();

}
```

3. Sprites

Chamamos de sprite o conjunto de dados que definem determinado objeto ou personagem num jogo. Para uma pessoa, por exemplo, podemos ter um sprite que contenha as posições vertical e horizontal dela no mundo, a direção para onde ela está virada e os bitmaps que podem representá-la durante o jogo.

O exemplo abaixo apresenta um carro que percorre a tela em diversas direções, demonstrando claramente o que foi mencionado no parágrafo anterior.

```
#define DX          3
#define DY          3
#define MAX_FRAMES  8

typedef struct _SPRITE
{

      BITMAP **frame;
      int x, y;
      int dx, dy;
      int dir;
```

```
} SPRITE;

enum { DIR_N, DIR_NE, DIR_E, DIR_SE, DIR_S, DIR_SO, DIR_O,
DIR_NO };

(...)

void principal (void)
{

      BITMAP *tela;
      PALETTE pal;
      SPRITE sp;

      sp.frame = (BITMAP **)malloc(MAX_FRAMES * sizeof(BITMAP
      *));
      sp.frame[DIR_N] = load_bitmap("CarroN.bmp", pal);
      sp.frame[DIR_NE] = load_bitmap("CarroNE.bmp", pal);
      sp.frame[DIR_E] = load_bitmap("CarroE.bmp", pal);
      sp.frame[DIR_SE] = load_bitmap("CarroSE.bmp", pal);
      sp.frame[DIR_S] = load_bitmap("CarroS.bmp", pal);
      sp.frame[DIR_SO] = load_bitmap("CarroSO.bmp", pal);
      sp.frame[DIR_O] = load_bitmap("CarroO.bmp", pal);
      sp.frame[DIR_NO] = load_bitmap("CarroNO.bmp", pal);

      sp.x = 370;
      sp.y = 50;
      sp.dx = -2 * DX;
      sp.dy = 0;
      sp.dir = DIR_O;

      tela = create_bitmap(MAX_X, MAX_Y);

      set_palette(pal);
      while (!keypressed())
      {

            sp.x += sp.dx;
            sp.y += sp.dy;
            switch (sp.dir)
            {

                  case DIR_N:

                        if (sp.y < 100)
                        {

                              sp.dx = -DX;
                              sp.dy = -DY;
                              sp.dir = DIR_NO;

                        }
                        break;

                  case DIR_NE:
```

```
            if (sp.x > (MAX_X - 100))
            {

                  sp.dx = 0;
                  sp.dy = -2 * DY;
                  sp.dir = DIR_N;

            }
            break;

      case DIR_E:

            if (sp.x > (MAX_X - 150))
            {

                  sp.dx = DX;
                  sp.dy = -DY;
                  sp.dir = DIR_NE;

            }
            break;

      case DIR_SE:

            if (sp.y > (MAX_Y - 100))
            {

                  sp.dx = 2 * DX;
                  sp.dy = 0;
                  sp.dir = DIR_E;

            }
            break;

      case DIR_S:

            if (sp.y > (MAX_Y - 150))
            {

                  sp.dx = DX;
                  sp.dy = DY;
                  sp.dir = DIR_SE;

            }
            break;

      case DIR_SO:

            if (sp.x < 50)
            {

                  sp.dx = 0;
                  sp.dy = 2 * DY;
                  sp.dir = DIR_S;

            }
            break;
```

```
                        case DIR_O:

                              if (sp.x < 100)
                              {

                                    sp.dx = -DX;
                                    sp.dy = DY;
                                    sp.dir = DIR_SO;

                              }
                              break;

                        case DIR_NO:

                              if (sp.y < 50)
                              {

                                    sp.dx = -2 * DX;
                                    sp.dy = 0;
                                    sp.dir = DIR_O;

                              }
                              break;

                  }
                  clear(tela);
                  draw_sprite(tela, sp.frame[sp.dir], sp.x, sp.y);
                  blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);
                  rest(20);

            }
            readkey();
```

```
      }
```

4. Detecção de Colisão

Frequentemente, é necessário verificar se dois sprites estão sobrepostos; para isso, existem diferentes métodos de detectar a possível colisão.

Como o método que chamamos de força bruta (checar cada ponto de um sprite com cada ponto de outro sprite) é ineficiente, a maior parte dos outros métodos são aproximativos. Veremos o principal deles, que consiste em dividir os sprites em retângulos, de forma que possamos verificar se cada retângulo está ou não sobreposto a outro.

Montando um Jogo Usando o Allegro

1. Visão Geral do Jogo

O objetivo do jogo que iremos montar é fazer desaparecer todos os monstros que andam pela tela; para isso, basta clicá-los com o ponteiro do mouse.

Como os monstros têm velocidades diferentes, a pontuação deve variar de acordo com a velocidade (quanto maior, mais pontos). Ao errar o clique (não clicar em nenhum monstro), o jogador perde pontos. Ao final da partida, depois que todos os monstros desaparecerem, é descontado do total de pontos um valor proporcional ao tempo

---

transcorrido desde o início da partida.

2. Definições e Outras Funções

O cenário do jogo consistirá de duas partes: a parte esquerda da tela é onde os monstros se movimentam, e a parte direita é onde está o placar, o contador de monstros e o tempo transcorrido. Definiremos que, do total de 800x600 da tela, 600x600 sejam da parte esquerda, e o resto, naturalmente, da direita. Para isso, escrevemos

```
#define ESP_PONTOS      200
#define ARENA_X         (MAX_X) - (ESP_PONTOS)
#define ARENA_Y         (MAX_Y)
```

Devemos definir, também, o número de monstros que aparecerão inicialmente, bem como os tamanhos horizontal e vertical de seu bitmap e os máximos deslocamentos horizontal e veritcal possíveis. Para tanto, fazemos

```
#define MAX_MONSTRO     30
#define MONSTRO_X       32
#define MONSTRO_Y       32
#define MAX_DX          5
#define MAX_DY          5
```

Utilizaremos um sprite denominado `MONSTRO`, definido da seguinte maneira:

```
typedef struct _MONSTRO
{

      int x, y;
      int dx, dy;
      int show;

} MONSTRO;
```

Além dos `#define`'s, precisamos escrever a função que será chamada pelo temporizador para incrementar o tempo. Note que, devido ao uso que se faz desta função (temporizador), é necessário o uso da macro que se encontra no final daquela.

```
void aumenta_tempo (void)
{

      tempo++;

}
END_OF_FUNCTION(aumenta_tempo);
```

Precisamos, ainda, incluir algum código na função `inicia` do esqueleto do programa, para que seja inicializado o arquivo de recordes

```
set_config_file("jogo.rec");
```

e as macros do temporizador

```
LOCK_VARIABLE(tempo);
```

```
LOCK_FUNCTION(aumenta_tempo);
```

3. As Inicializações

Devemos, primeiramente, declarar as variáveis que vamos utilizar no jogo. Para tal, dentro da função `principal`, escrevemos

```
MONSTRO monstro[MAX_MONSTRO];
BITMAP *bmp_monstro;
BITMAP *tela;
PALETTE pal;
SAMPLE *certo, *errado;
MIDI *musica;
int mx, my;
int num_monstros;
int pontos, maior;
int acertou;
int i;
```

Há, ainda, uma variável a ser declarada como global

```
volatile int tempo;
```

e que será utilizada no contador de tempo.

Podemos, então, fazer as inicializações necessárias. Primeiro, criamos o bitmap de double buffering, declarado como a variável `tela`,

```
tela = create_bitmap(MAX_X, MAX_Y);
```

Depois, carregamos o bitmap `monstro.bmp` na memória

```
bmp_monstro = load_bitmap("monstro.bmp", pal);
```

e os waves (certo.wav e errado.wav) e midi (caverns.mid) que serão utilizados

```
certo = load_sample("certo.wav");
errado = load_sample("errado.wav");
musica = load_midi("caverns.mid");
```

Após isso, inicializamos com valores aleatórios o vetor de elementos do tipo `MONSTRO`, da seguinte forma:

```
srand(time(0));
for (i = 0; i < MAX_MONSTRO; i++)
{

      monstro[i].x = rand() % (ARENA_X - MONSTRO_X);
      monstro[i].y = rand() % (ARENA_Y - MONSTRO_Y);
      do
      {

            monstro[i].dx = (rand() % (MAX_DX * 2)) - MAX_DX;

      } while (monstro[i].dx == 0);
      do
      {

            monstro[i].dy = (rand() % (MAX_DY * 2)) - MAX_DY;

      } while (monstro[i].dy == 0);
      monstro[i].show = TRUE;

}
```

e inicializamos o número de monstros

```
num_monstros = MAX_MONSTRO;
```

Para terminar, inicializamos as variáveis `mx` e `my`, que guardarão os valores da última posição do mouse

```
mx = my = -1;
```

e inicializamos, também, as variáveis `pontos` e `maior`

```
pontos = 0;
maior = get_config_int(NULL, "HighScore", 0);
```

Devemos, também, adicionar algumas linhas ao final da função `inicia` (antes do último `return`) para que possamos utilizar algumas outras funções.

```
set_config_file("jogo.rec");

LOCK_VARIABLE(tempo);
LOCK_FUNCTION(aumenta_tempo);
```

4. O Loop Principal

Antes de iniciarmos o loop principal do jogo, fazemos algumas chamadas de inicialização que, por certos motivos, foram preferencialmente deixadas para esta parte. Assim, podemos chamar uma função de fade

```
fade_in(pal, 4);
```

a função para iniciar a música

```
play_midi(musica, TRUE);
```

zerar o tempo e iniciar o temporizador da função `aumenta_tempo`

```
tempo = 0;
install_int(aumenta_tempo, 1000);
```

Após isso, vamos, finalmente, ao loop principal. A parte `do-while` consiste de

```
clear_keybuf();
do
{


      /* Loop Principal */

} while ((!key[KEY_ESC]) && (num_monstros > 0));
```

Então, dentro do loop principal, inserimos o restante do código. Primeiro, as rotinas que irão gerenciar o clique com o botão do mouse

```
if ((mouse_b & 1) && ((mouse_x != mx) || (mouse_y != my)))
{

      mx = mouse_x;
      my = mouse_y;
      for (i = 0, acertou = FALSE; i < MAX_MONSTRO; i++)
      {

            if ((mouse_x >= monstro[i].x) && (mouse_x <=
            (monstro[i].x + MONSTRO_X)) &&
                (mouse_y >= monstro[i].y) && (mouse_y <=
            (monstro[i].y + MONSTRO_Y)) &&
                (monstro[i].show))
            {

                  monstro[i].show = FALSE;
                  acertou = TRUE;
                  pontos += (abs(monstro[i].dx) +
                  abs(monstro[i].dy)) * 50;
                  num_monstros--;
```

```
                }

        }
        if (acertou)
        {

                play_sample(certo, 255, 128, 1000, FALSE);

        }
        else
        {

                play_sample(errado, 255, 128, 1000, FALSE);
                pontos -= 100;

        }

}
```

e, depois, as rotinas de double buffering

```
show_mouse(NULL);
clear(tela);
for (i = 0; i < MAX_MONSTRO; i++)
{

        if (monstro[i].show)
        {

                monstro[i].x += monstro[i].dx;
                monstro[i].y += monstro[i].dy;
                if ((monstro[i].x < 0) || (monstro[i].x > (ARENA_X -
                MONSTRO_X)))
                {

                        monstro[i].dx = -monstro[i].dx;

                }
                if ((monstro[i].y < 0) || (monstro[i].y > (ARENA_Y -
                MONSTRO_Y)))
                {

                        monstro[i].dy = -monstro[i].dy;

                }
                draw_sprite(tela, bmp_monstro, monstro[i].x,
                monstro[i].y);

        }

}
imprime_placar(tela, pontos, maior, num_monstros);
show_mouse(tela);
blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);
```

Com isso, terminamos o loop principal do jogo.

5. As Finalizações

Quando saímos do loop principal, não podemos apenas terminar o programa. É necessário fazer uma série de chamadas, tais como a de parar a música

```
stop_midi();
```

e a de remover o temporizador

```
remove_int(aumenta_tempo);
```

Além disso, precisamos calcular os pontos do jogador e, se for o caso, armazenar essa pontuação como a maior até o momento. Qualquer que seja a situação, devemos, também, imprimir esta pontuação final obtida. Para isso,

```
pontos -= num_monstros * 50;
pontos -= (tempo - 30) * 5 / 3;
if (pontos > maior)
{

      maior = pontos;
      set_config_int(NULL, "HighScore", pontos);

}
clear(tela);
imprime_placar(tela, pontos, maior, num_monstros);
blit(tela, screen, 0, 0, 0, 0, MAX_X, MAX_Y);
```

Para que o jogador possa ver o seu placar, devemos, ainda, fazer um loop de espera por uma tecla. Assim,

```
while (key[KEY_ESC]);
clear_keybuf();
textout_centre(screen, font, "Pressione qualquer tecla para
continuar", (ARENA_X) / 2,
               (ARENA_Y) / 2 - 8, 8);
while (!keypressed());
```

Apenas para finalizar de uma boa forma, utilizamos uma função de fade

```
fade_out(4);
```

As Funções do Allegro

1. Funções Básicas

`int allegro_init();`

Inicializa a biblioteca Allegro. Não faz muita coisa a não ser inicializar algumas variáveis globais e reservar memória. Retorna zero em caso de sucesso (na verdade, não

é necessário checar se a função foi bem sucedida, pois, ao ser usada no início do programa, como é recomendada, ela nunca falha). Deve ser a primeira função do Allegro a ser chamada.

```
void allegro_exit();
```

Fecha o Allegro. Isto inclui retornar ao modo texto e remover qualquer rotina de mouse, teclado ou temporizador que tenha sido instalada. Não há necessidade (embora seja recomendável) de chamar explicitamente essa função pois, normalmente, isto é feito automaticamente quando o programa termina.

2. Funções de Vídeo

2.1. Funções Básicas de Vídeo

```
void set_color_depth(int depth);
```

Esta função configura o número de bits de cores (determinado pelo parâmetro **`depth`**) com que os gráficos serão exibidos. Os possíveis valores para **`depth`** são 8, 15, 16, 24 e 32. Esta função deve ser chamada sempre **ANTES** da função `set gfx mode`.

```
int set_gfx_mode(int card, int w, int h, int v_w, int v_h);
```

Instrui o Allegro a mudar para o modo gráfico. O parâmetro **`card`** deve ser um dos seguintes valores:

GFX_TEXT	retorna para o modo texto
GFX_AUTODETECT	faz com que o Allegro escolha o driver gráfico mais apropriado (normalmente tem problemas com placas de vídeo mais recentes)
GFX_VGA	modo VGA (320x200, 320x100, 160x120, 80x80)
GFX_MODEX	uma versão planar do modo VGA (tweaked)
GFX_VESA1	usa o driver VESA 1.x
GFX_VESA2B	usa o driver VBE 2.0 em modo banked

GFX_VESA2L	usa o driver VBE 2.0 com framebuffer linear
GFX_VESA3	usa o driver VBE 3.0
GFX_VBEAF	usa o acelerador de hardware API VBE/AF
GFX_XTENDED	usa o driver 640x480 unchained

Os parâmetros **w** e **h** informam o numero de pixels existentes, respectivamente, em uma linha e em uma coluna do vídeo. Ex: w=800 e h=600.

Os parâmetros **v_w** e **v_h** determinam o tamanho da tela virtual, ou seja, o tamanho de uma pseudo-tela da qual pode-se visualizar apenas um pedaço (determinado pelos parâmetros **w** e **h**). Para manipular telas virtuais, o Allegro oferece uma série de funções de rolamento da tela. Porém, não as abordaremos. Para desabilitar este recurso, atribua 0 (zero) a ambos os parâmetros.

O valor retornado por esta função será menor do que 0 (zero) se, e somente se, um erro ocorrer. É interessante conferir o valor retornado por esta função, de modo a escolher o driver de vídeo adequado, caso o selecionado não funcione. Por exemplo:

```
#define VIDEO_CARD GFX_VESA1
#define TAM_X 800
#define TAM_Y 600
#define ALT_TAM_X 640
#define ALT_TAM_Y 480

(...)

if (set_gfx_mode(VIDEO_CARD, TAM_X, TAM_Y, 0, 0) < 0) {

      if (set_gfx_mode(GFX_AUTODETECT, ALT_TAM_X, ALT_TAM_Y, 0,
      0) < 0) {

            /* Mensagem de erro. */

            (...)


      }


}
```

2.2. Funções de Bitmaps

O Allegro trata todos os gráficos que podem ser exibidos na tela ou carregados na memória como bitmaps. Bitmaps são matrizes de pixels, em que cada valor indica uma cor. Para declarar um bitmap, digite:

```
BITMAP *nome_do_bitmap;
```

```
extern BITMAP *screen;
```

Para facilitar o uso de suas funções de manipulação de bitmaps, o Allegro trata a tela também como um bitmap, que é definido no arquivo `allegro.h` como `screen`.

Assim, nas funções de manipulação de bitmaps abaixo, sempre que desejarmos apresentar diretamente na tela algum gráfico, passaremos como argumento da função a variável `screen`.

```
BITMAP *create_bitmap(int width, int height);
```

Cria um bitmap na memória com largura **`width`** e altura **`height`**. O valor retornado é o do endereço da área de memória onde foi alocado o espaço; assim, caso a função retorne **`NULL`**, significa que não foi possível alocar a memória necessária para armazenar o bitmap com as dimensões requisitadas.

É importante notar que a função apenas aloca o espaço, sem "limpá-lo", e, com isso, é necessário usar a função `clear` ou `clear to color`, mencionadas adiante.

```
void destroy_bitmap(BITMAP *bitmap);
```

Destrói o bitmap apontado por **`bitmap`**, liberando a memória ocupada por este.

```
BITMAP *load_bitmap(char *filename, PALETTE pal);
```

Carrega um bitmap de um arquivo, cujo nome é **`filename`**, e carrega a paleta por ele usada no endereço apontado por **`pal`**. O tipo do arquivo (BMP, LBM, PCX, TGA) é informado através da extensão do mesmo. Deve-se destruir o bitmap após o uso com a função `destroy bitmap`. A função retorna um NULL se houver algum erro no carregamento do arquivo.

```
int save_bitmap(char *filename, BITMAP *bmp, PALETTE pal);
```

Salva o bitmap apontado por **`bmp`**, com a paleta apontada por **`pal`**, no arquivo de nome **`filename`**. O tipo do arquivo (BMP, LBM, PCX, TGA) é informado através da extensão do mesmo. A função retorna um número diferente de zero caso ocorra algum erro.

```
void clear(BITMAP *bitmap);
```

Limpa o bitmap apontado por **`bitmap`** para a cor 0 (zero).

```
void clear_to_color(BITMAP *bitmap, int color);
```

Limpa o bitmap apontado por **`bitmap`** para a cor especificada por **`color`**.

`void putpixel(BITMAP *bmp, int x, int y, int color);`

Desenha um ponto, no bitmap apontado por **`bmp`**, na coordenada (x, y), utilizando a cor especificada por **`color`**.

`int getpixel(BITMAP *bmp, int x, int y);`

Retorna o código de cor da coordenada (x, y) no bitmap apontado por **`bmp`**; retorna -1 caso o ponto esteja fora do bitmap.

`void line(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);`

Desenha uma linha, no bitmap apontado por **`bmp`**, da coordenada (**`x1`**, **`y1`**) até a coordenada (**`x2`**, **`y2`**), utilizando a cor especificada por **`color`**.

`void hline(BITMAP *bmp, int x1, int y, int x2, int color);`

Desenha uma linha horizontal, no bitmap apontado por **`bmp`**, da coordenada (**`x1`**, **`y`**) até a coordenada (**`x2`**, **`y`**), utilizando a cor especificada por **`color`**.

`void vline(BITMAP *bmp, int x, int y1, int y2, int color);`

Desenha uma linha vertical, no bitmap apontado por **`bmp`**, da coordenada (**`x`**, **`y1`**) até a coordenada (**`x`**, **`y2`**), utilizando a cor especificada por **`color`**.

`void rect(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);`

Desenha a borda de um retângulo, no bitmap apontado por **`bmp`**, da coordenada (**`x1`**, **`y1`**) até a coordenada (**`x2`**, **`y2`**), utilizando a cor especificada por **`color`**.

`void rectfill(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);`

Desenha um retângulo, no bitmap apontado por **`bmp`**, da coordenada (**`x1`**, **`y1`**) até a coordenada (**`x2`**, **`y2`**), utilizando a cor especificada por **`color`**.

`void triangle(BITMAP *bmp, int x1, int y1, int x2, int y2, int x3, int y3, int color);`

Desenha um triângulo, no bitmap apontado por **`bmp`**, com vértices (**`x1`**, **`y1`**), (**`x2`**, **`y2`**) e (**`x3`**, **`y3`**).

`void polygon(BITMAP *bmp, int vertices, int *points, int color);`

Desenha um polígono, no bitmap apontado por **`bmp`**, com **`vertices`** vértices especificados pelo vetor **`points`** de pares de coordenadas (x, y), com cor **`color`**. Por exemplo:

```
int *pontos = { 1, 1, 2, 4, 6, 3, 5, 5 };
BITMAP *bmp;

(...)
```

```
bmp = create_bitmap(50, 50);
polygon(bmp, 4, pontos, 100);

(...)
```

irá desenhar um quadrilátero com vértices (1, 1), (2, 4), (6, 3), (5, 5) no bitmap **bmp** com cor 100.

void circle(BITMAP *bmp, int x, int y, int radius, int color);

Desenha uma circunferência, no bitmap apontado por **bmp**, com centro (**x**, **y**) e raio **radius**, utilizando a cor especificada por **color**.

void circlefill(BITMAP *bmp, int x, int y, int radius, int color);

Desenha um círculo, no bitmap apontado por **bmp**, com centro (**x**, **y**) e raio **radius**, utilizando a cor especificada por **color**.

void blit(BITMAP *source, BITMAP *dest, int source_x, int source_y, int dest_x, int dest_y, int width, int height);

Copia uma área retangular, de largura **width** e altura **height**, da coordenada (**source_x**, **source_y**) do bitmap apontado por **source** para a coordenada (**dest_x**, **dest_y**) do bitmap apontado por **dest**.

void draw_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);

Copia inteiramente o bitmap apontado por **sprite** na coordenada (**x**, **y**) do bitmap apontado por **bmp**. Equivalente a blit(**sprite**, **bmp**, 0, 0, x, y, **sprite->w**, **sprite->h**).

A grande diferença entre as funções `blit` e `draw_sprite` é que a primeira desenha o bitmap como ele é, enquanto a segunda trata como transparente as cores 0 (normalmente preto, no modo 8 bits) e rosa claro (nos outros modos).

2.3. Funções de Paleta

Todas as funções de desenho do Allegro usam parâmetros inteiros para representar cores. No modo de 256 cores, esses valores correspondem a índices de um vetor, onde cada elemento é uma estrutura que contém as intensidades de vermelho, verde e azul de cada uma das 256 cores possíveis. Este vetor é denominado paleta. Para declará-la, digite:

```
PALETTE nome_da_paleta;
```

O Allegro define algumas paletas padrões, que podem ser utilizadas pelo usuário para determinadas situações. São elas:

extern PALETTE desktop_palette;

Esta paleta era utilizada pelo Atari ST. É utilizada pelos programas de teste e exemplo

do Allegro e é a paleta padrão utilizada pelo mesmo, caso nenhuma outra paleta seja setada.

```
extern PALETTE black_palette;
```

Nesta paleta, todas as 256 cores possíveis correspondem a cor preta. Esta paleta é utilizada pelas funcões de *fade*, e pode ser utilizada também pelo usuário em determinadas situações, em conjunto com a função set palette.

```
void set_palette(PALETTE p);
```

Seleciona a paleta especificada por **p** como a paleta a ser utilizada.

```
void set_palette_range(PALETTE p, int from, int to, int vsync);
```

Seleciona, da paleta especificada por **p**, os índices de **from** até **to**. Se **vsync** tiver um valor verdadeiro, o Allegro espera pelo retraço vertical antes de selecionar a paleta. Tem o mesmo efeito da função set palette, com a diferença que apenas um trecho da paleta é selecionado para uso.

```
void get_palette(PALETTE p);
```

Guarda a paleta utilizada atualmente em **p**.

```
void get_palette_range(PALETTE p, int from, int to);
```

Guarda parte da paleta utilizada atualmente em **p**, com índices de **from** até **to**.

```
void fade_in(PALETTE p, int speed);
```

Altera gradualmente as cores da paleta atual, desde a black palette até a paleta **p**, com velocidade **speed** (1 - devagar, 64 - muito rápido).

```
void fade_out(int speed);
```

Altera gradualmente as cores da plaheta atual até atingir a black palette, com velocidade **speed** (1 - devagar, 64 - muito rápido).

```
void fade_in_range(PALETTE p, int speed, int from, int to);
```

Altera gradualmente as cores da paleta atual, dos índices de **from** até **to**, desde a black palette até a paleta **p**, com velocidade **speed** (1 - devagar, 64 - muito rápido).

```
void fade_from(PALETTE source, PALETTE dest, int speed);
```

Atlera gradualmente as cores da paleta atual, desde a paleta **source** até a paleta **dest**, com velocidade **speed** (1 - devagar, 64 - muito rápido).

2.4. Funções de Fonte

Assim como para os bitmaps e paletas, o Allegro define um tipo FONT, que contém a descrição de fontes que podem ser desenhadas na tela. Para declarar uma fonte, digite:

```
FONT *nome_da_fonte;
```

```
extern FONT *font;
```

Esta fonte é utilizada como padrão para determinadas tarefas do Allegro, e pode ser alterada para que o Allegro utilize uma nova fonte nessas funções.

```
void text_mode(int mode);
```

Altera o modo de desenho do texto. Se **mode** for 0 (zero) ou positivo, o fundo dos caracteres terá a cor de código **mode**. Caso seja negativo, o fundo será transparente. O padrão é 0 (zero).

```
void textout(BITMAP *bmp, FONT *f, unsigned char *s, int x, int y, int
color);
```

Escreve a string **s**, na coordenada (**x**, **y**) do bitmap apontado por **bmp**, utilizando a fonte apontada por **f** e a cor **color**. Se o valor de **color** for -1, a cor utilizada será a cor original do bitmap da fonte.

```
void textout_centre(BITMAP *bmp, FONT *f, unsigned char *s, int x, int
y, int color);
```

Semelhante a função textout, porém, imprime a string **s** centralizada na coordenada especificada.

```
void textprintf(BITMAP *bmp, FONT *f, int x, int y, int color, char
*fmt, ...);
```

Escreve um texto formatado (no mesmo estilo da função printf) no bitmap apontado por **bmp**, na coordenada (**x**, **y**), utilizando a fonte apontada por **f** e a cor **color**.

```
void textprintf_centre(BITMAP *bmp, FONT *f, int x, int y, int color,
char *fmt, ...);
```

Semelhante a função textprintf, porém, imprime o texto formatado centralizado na coordenada especificada.

3. Funções de Teclado

```
int install_keyboard();
```

Inicializa o teclado para ser usado pelo Allegro. Deve-se chamar esta função antes de qualquer outra função de teclado. Após a chamada desta função, não se pode mais utilizar as funções padrões de acesso ao teclado (scanf, getchar, etc.). Para usar as funções padrões de acesso ao teclado novamente utilize a função remove_keyboard.

void remove_keyboard();

Devolve o controle do teclado a BIOS. Faz o contrário da função install keyboard. Normalmente, não é necessário chamar esta função, pois a função allegro exit já o faz.

void clear_keybuf();

Limpa o *buffer* do teclado. Qualquer tecla anteriormente pressionada que ainda não tenha sido lida pela função readkey será perdida.

int keypressed();

Retorna verdadeiro caso alguma tecla tenha sido pressionada e esteja no *buffer* a espera da leitura. Equivalente a função kbhit da conio.

int readkey();

Retorna o próximo caractere do *buffer* do teclado. Se o *buffer* estiver vazio, a função espera até que uma tecla seja pressionada. O byte baixo do valor retornado contém o código ASCII do caracter, enquanto o byte alto retorna o scancode da tecla. O scancode não é afetado pelas teclas SHIFT, CTRL e ALT. O código ASCII é afetado normalmente pelas teclas SHIFT e CTRL (a tecla SHIFT altera o caso, enquanto a tecla CTRL retorna a posição da tecla pressionada simultaneamente). Ao pressionar a tecla ALT o código ASCII retornado é 0 (zero), retornando apenas o scancode (ainda no byte alto). Por exemplo:

```
/* Pelo codigo ASCII */
if ((readkey() & 0xFF) == 'd')

      printf("Voce pressionou a tecla 'd'\n");



/* Pelo scancode */
if ((readkey() >> 8) == KEY_SPACE)

      printf("Voce pressionou a tecla Espaço\n");



/* Pressionando CTRL+[letra] */
if ((readkey() & 0xFF) == 3)

      printf("Voce pressionou CTRL+C\n");



/* Pressionando ALT+[letra] */
if (readkey() == (KEY_X << 8))

      printf("Voce pressionou ALT+X\n");
```

```
extern volatile char key[KEY_MAX];
```

Vetor de **KEY_MAX** posições, cada uma representando uma das teclas do teclado. Caso o valor de key[KEY_TECLA] seja verdadeiro, a tecla está sendo pressionada. Por exemplo:

```
if (key[KEY_D])
      printf("Voce pressionou a tecla 'D'\n");


if (key[KEY_SPACE])
      printf("Voce pressionou a tecla Espaço\n");


if ((key[KEY_CTRL]) && (key[KEY_C]))
      printf("Voce pressionou CTRL+C\n");


if ((key[KEY_ALT] && (key[KEY_X]))
      printf("Voce pressionou ALT+X\n");
```

Note que key[KEY_TECLA] apenas informa o estado da tecla, não espera pelo pressionamento da mesma.

4. Funções de Mouse

```
int install_mouse();
```

Inicializa o mouse para ser usado pelo Allegro. É necessário chamar esta função antes de qualquer outra função de mouse. Retorna -1 se ocorrer algum erro; caso contrário, retorna o número de botões do mouse.

```
void remove_mouse();
```

Remove o controle do Allegro sobre o mouse. Normalmente, não é necessário chamar esta função, pois a função `allegro_exit` já o faz.

```
extern volatile int mouse_x;
extern volatile int mouse_y;
```

Guardam os valores da posição x e y do cursor do mouse.

```
extern volatile int mouse_b;
```

Guarda o estado atual do pressionamento dos botões do mouse. O bit 0 de **mouse_b** guarda o estado do botão esquerdo, o bit 1 o do botão esquerdo, e o bit 2 o do botão do meio.

```
if (mouse_b & 1)
      printf("Botão esquerdo do mouse pressionado\n");

if (!(mouse_b & 2))
      printf("Botão direito do mouse não pressionado\n");

if (mouse_b & 4)
      printf("Botão do meio do mouse pressionado\n");
```

```
void set_mouse_range(int x1, int y1, int x2, int y2);
```

Define a área onde o ponteiro do mouse pode se locomover na tela. A área é definida pelo retângulo de coordenada superior esquerda (**x1**, **y1**) e coordenada inferior direita (**x2**, **y2**).

```
void position_mouse(int x, int y);
```

Move o ponteiro do mouse para a coordenada (**x**, **y**).

```
void show_mouse(BITMAP *bmp);
```

Mostra o ponteiro do mouse no bitmap apontado por **bmp**. Funciona apenas se o timer estiver instalado. Para esconder o ponteiro do mouse, passe o valor **NULL** como argumento. Não é recomendável desenhar na tela enquanto o ponteiro do mouse estiver visível, pois isto pode gerar manchas provocadas pelo rastro do mouse. Também não é uma boa idéia esconder o mouse, desenhar na tela e exibir o mouse novamente, pois isto faz com que o mouse fique "piscando" na tela. O ideal é que sejam utilizadas técnicas como o double buffering.

5. Funções de Som

```
int install_sound(int digi_card, int midi_card, char *cfg_path);
```

Inicializa o módulo de som para ser usado com o Allegro. Os parâmetros **digi_card** e **midi_card** referem-se, respectivamente, aos controladores de som digital e MIDI.

O valor de **`digi_card`** deve ser um dos abaixo:

DIGI_AUTODETECT	instrui o Allegro a escolher o driver de som
DIGI_NONE	sem som digital
DIGI_SB	auto-detecta placas do tipo Sound Blaster
DIGI_SB10	Sound Blaster 1.0 (8 bit mono)
DIGI_SB15	Sound Blaster 1.5 (8 bit mono)
DIGI_SB20	Sound Blaster 2.0 (8 bit mono)
DIGI_SBPRO	Sound Blaster Pro (8 bit stereo)
DIGI_SB16	Sound Blaster 16 (16 bit stereo)
DIGI_AUDIODRIVE	ESS AudioDrive
DIGI_SOUNDSCAPE	Ensoniq Soundscape

O valor de **`midi_card`** deve ser um dos abaixo:

MIDI_AUTODETECT	instrui o Allegro a escolher o driver de MIDI
MIDI_NONE	sem som MIDI
MIDI_ADLIB	auto-detecta sintetizadores do tipo Adlib ou Sound Blaster FM
MIDI_OPL2	sintetizador OPL2 (mono, usado em Adlib e Sound Blaster)
MIDI_2XOPL2	sintetizador OPL2 dual (stereo, usado em Sound Blaster Pro-I)
MIDI_OPL3	sintetizador OPL3 (stereo, usado em Sound Blaster Pro-II e acima)
MIDI_SB_OUT	interface MIDI Sound Blaster
MIDI_MPU	interface MIDI MPU-401

MIDI_DIGMID	sample-based software wavetable player
MIDI_AWE32	AWE32 (EMU8000 chip)

O parâmetro **cfg_path** existe apenas para compatibilidade com versões anteriores do Allegro. Ignore-o passando o valor **NULL**.

```
void remove_sound();
```

Remove o módulo de som. Normalmente não é necessário chamar esta função, pois a função allegro_exit já o faz.

```
MIDI *load_midi(char *filename);
```

Carrega um arquivo MIDI, retornando um ponteiro para uma estrutura **MIDI**. Caso ocorra um erro, o valor **NULL** será retornado.

```
int play_midi(MIDI *midi, int loop);
```

Toca a MIDI especificada por **midi**, parando de tocar qualquer música que estivesse sendo tocada anteriormente. Se a flag **loop** estiver setada, a música será tocada até que a função seja novamente chamada para tocar outra música, ou a função stop_midi seja chamada. Caso a flag **loop** não esteja setada, a música irá parar de tocar ao alcançar o final do MIDI. Retorna um valor diferente de zero se um erro ocorrer.

```
void stop_midi();
```

Faz parar de tocar qualquer música que esteja sendo tocada. Tem o mesmo efeito da instrução **play_midi(NULL, FALSE)**.

```
SAMPLE *load_sample(char *filename);
```

Carrega um arquivo sample, retornando um ponteiro para uma estrutura **SAMPLE**. Caso ocorra um erro, o valor **NULL** será retornado. Aceita arquivos do tipo WAV (mono e stereo) e VOC (mono) em formato 8 e 16 bits.

```
int play_sample(SAMPLE *spl, int vol, int pam, int freq, int loop);
```

Inicializa o sample especificado por **spl** utilizando determinados argumentos. O argumento **vol** determina o volume (valores de 0 até 255), o argumento **pan** determina o balanço (valores de 0 até 255), o argumento **freq** determina a frequência com que o sample é tocado (o valor 1000 faz com que o sample seja tocado na mesma frequência que foi gravado; o valor 2000 é o dobro da frequência, etc.) e o argumento **loop** informa se o Allegro deve repetir o sample infinitamente ou não.

```
void stop_sample(SAMPLE *spl);
```

Faz parar de tocar o sample especificado por **spl**. É necessário chamar esta função caso o sample tenha sido incializado com a flag **loop** setada.

6. Outras Funções

6.1. Funções de Timer

`int install_timer();`

Inicializa o temporizador do Allegro. É necessário chamar esta função antes de usar qualquer função de timer, bem como antes de mostrar o ponteiro do mouse ou tocar uma MIDI.

`void remove_timer();`

Remove o temporizador do Allegro. Normalmente, não é necessário chamar esta função, pois a função `allegro_exit` já o faz.

`int install_int(void (*proc)(), int speed);`

Inicializa um temporizador para que, a cada intervalo de **`speed`** milisegundos, a função **`proc`** seja chamada. Caso a função install_int seja chamada antes da inicialização do temporizador do Allegro, a função `install_timer` será automaticamente chamada.

`void remove_int(void (*proc)());`

Remove o temporizador que chama a função **`proc`**. A remoção dos temporizadores é feita automaticamente pela função `allegro_exit`.

`LOCK_VARIABLE(`*`variavel`*`)`
`LOCK_FUNCTION(`*`funcao`*`)`
`END_OF_FUNCTION(`*`funcao`*`)`

Essas três macros são necessárias quando utiliza-se temporizadores. As macros **`LOCK_VARIABLE`** e **`LOCK_FUNCTION`** bloqueiam a área de memória ultilizada pela variável ou função, enquanto a macro **`END_OF_FUNCTION`** determina o final de uma função dentro do programa. Por exemplo:

```
int x;

(...)

int aumenta_x()
{

      x++;


}
END_OF_FUNCTION(aumenta_x);

int main()
{
```

```
            (...)

            /*

                  Bloqueia a variavel x e a funcao aumenta_x
                  e instala um temporizador que chamara a
                  funcao aumenta_x a cada 0,2 segundo (200
                  milisegundos).

            */
            LOCK_VARIABLE(x);
            LOCK_FUNCTION(aumenta_x);
            install_int(aumenta_x, 200);

            (...)

      }
```

`void rest(long time);`

Provoca uma pausa de **`time`** milisegundos no programa.

6.2. Funções de DataFile

DataFile é um tipo especial de arquivo, com extensão .dat, e que pode armazenar diversos tipos de arquivos, como BitMaps, MIDIs, fontes, paletas, que podem ser carregados e utilizados durante a execução do programa.

Para criar um arquivo DataFile existe um utilitário chamado *Grabber*, que se encontra no diretório **`tools`**, dentro do diretório principal do Allegro. O programa possui uma interface que facilita a confecção do arquivo DataFile.

Para acessar um arquivo DataFile num programa, utiliza-se o tipo **`DATAFILE`**, da seguinte maneira:

```
DATAFILE *dat;
```

Depois de determinar um ponteiro para o arquivo DataFile, pode-se abri-lo utilizando a função `load datafile`, descrita abaixo:

`DATAFILE *load_datafile(const char *filename);`

Abre o arquivo de nome **`filename`** como um arquivo datafile, preparando-o para a leitura de seus dados. Retorna **`NULL`** caso ocorra algum erro.

6.3. Funções de Arquivos de Configuração

```
void set_config_file(const char *filename);
```

Instrui o Allegro a utilizar o arquivo **filename** para as futuras chamadas a outras funções de arquivos de configuração. Caso as outras funções sejam chamadas antes da função set_config_file, o Allegro utiliza o arquivo **allegro.cfg** do diretório atual como o arquivo de configuração a ser lido.

```
void set_config_string(const char *section, const char *name, const
char *val);
```

Escreve uma string no arquivo de configuração atual na chave **name** da seção **section**. Caso **val** seja **NULL**, a chave é removida do arquivo. Caso **section** seja **NULL**, a chave é escrita na raiz do arquivo de configuração. Os dados são escritos no arquivo ao chamar a função allegro exit.

As linhas iniciadas com um '#' não são lidas ou escritas no arquivo de configuração, sendo tratadas como comentários.

```
void set_config_id(const char *section, const char *name, int val);
```

Faz o mesmo que a função set config string, exceto pelo fato de escrever um ID no arquivo, ao invés de uma string. Para cada valor de **val** existe um ID de 4 letras correspondente. Isto é útil para guardar informações sobre placas de vídeo e som nos arquivos de configuração.

```
void set_config_int(const char *section, const char *name, int val);
void set_config_hex(const char *section, const char *name, int val);
void set_config_float(const char *section, const char *name, float
val);
```

Fazem o mesmo que a função set config string, exceto pelo fato de escreverem, respectivamente, números inteiros, hexadecimais e de ponto flutuante.

```
const char *get_config_string(const char *section, const char *name,
const char *def);
```

Lê uma string do arquivo de configuração atual da chave **name** da seção **section**. Caso **section** seja **NULL**, a chave é lida da raiz do arquivo de configuração. Caso a chave não seja encontrada, o valor de **def** é retornado.

```
int get_config_id(const char *section, const char *name, int def);
```

Lê uma ID (veja em set config id) do arquivo de configuração atual (mais detalhes em get config string).

```
int get_config_int(const char *section, const char *name, int def);
int get_config_hex(const char *section, const char *name, int def);
float get_config_float(const char *section, const char *name, float
def);
```

Lêem, respectivamente, um número inteiro, hexadecimal e de ponto flutuante, do arquivo de configuração atual (mais detalhes em get config string).