Allegro Quick Reference

Public domain. Version 2006-03-30 (changes).
connellybarnes@yahoo.com
Translated to Japanese by Ohsawa Hirotaka.

1. General:

#include <allegro.h>
Press Ctrl+Alt+End to end any Allegro program.
Call allegro_init() at the very start of all programs.
Place END_OF_MAIN(); after the main() method.

2. Graphics:

```
putpixel   (bmp, x, y, color)
getpixel   (bmp, x, y, color)
line       (bmp, x1, y1, x2, y2, color)
triangle   (bmp, x1, y1, x2, y2, x3, y3, color)
polygon    (bmp, sides, vertex[sides * 2], color)
rect       (bmp, x1, y1, x2, y2, color)
rectfill   (bmp, x1, y1, x2, y2, color)
circle     (bmp, x, y, radius, color)
circlefill (bmp, x, y, radius, color)
ellipse    (bmp, x, y, xradius, yradius, color)
ellipsefill(bmp, x, y, xradius, yradius, color)
arc        (bmp, x, y, angle1, angle2, radius, color)
floodfill  (bmp, x, y, color)
```

All graphics commands take integer arguments. See the sections on Anglesand Colorsfor how to convert an angle or color value to Allegro's integer format.

3. Bitmap Commands:

```
BITMAP *bmp;
bmp = create_bitmap(width, height)
bmp = load_bitmap("filename.bmp|pcx|tga|lbm", 0)
destroy_bitmap(bmp)
clear(bmp)
clear_to_color(bmp, color)
draw_sprite(bmp, sprite, x, y)
draw_sprite_v_flip(bmp, sprite, x, y)
draw_sprite_h_flip(bmp, sprite, x, y)
draw_sprite_vh_flip(bmp, sprite, x, y)
rotate_sprite(bmp, sprite, x, y, angle)
stretch_sprite(bmp, sprite, x, y, spritew, spriteh)
blit(source_bmp, dest_bmp, xsrc, ysrc, xdest, ydest, width, height)
stretch_blit(src, dest, xsrc, ysrc, srcw, srch, xdest, ydest, destw, desth)
```

Pronounce the first variable as "bitmap pointer to bmp". A bitmap pointer is simply a reference to some memory which has been created dynamically. Bitmap dimensions can be determined through 'bmp->w' and 'bmp->h'.

Make sure the video mode is set up before calling any of the bitmap functions, or your bitmap will be in the wrong color depth. The operating system will automatically reclaim an application's memory on exit, so you do not necessarily need to call destroy_bitmap(...) on every bitmap that has been created or loaded. However, it is very important that you call destroy_bitmap (...) on temporary bitmaps created within procedures, or else your application will run out of memory and die a painful death.

Bitmaps created with create_bitmap(...) will initially contain random pixels. If you don't want the user to be looking at randomly colored pixels, then the bitmap should be cleared before drawing commands are used with it.

Draw_sprite draws a sprite bitmap onto a larger bitmap, typically the screen or a buffer. To make part of a sprite transparent, the special "mask color" can be used. In truecolor modes, the mask color is RGB(255, 0, 255) - or maximum red and blue, zero green. In 8-bit mode, the mask color is color 0. Both draw_sprite and rotate_sprite draw the sprite with (x, y) as the upper left coordinate. To have a sprite revolve around its center, rather than the upper-left, use the following formula:

```
rotate_sprite(bmp, sprite, x-sprite->w/2, y-sprite->h/2, angle)
```

Blit is an image-copying function that stands for "BLock Image Transfer". The parameters are fairly straightforward: it copies a rectangle of pixels from somewhere on a source bitmap to a different location on a destination bitmap. Stretch_blit allows for the source and destination rectangles to be of different sizes, which produces a stretching effect.

You may also wish to make an intermediate loading function which will quit with an error if a bitmap is not found:

```
BITMAP *load_image(char *filename) {
  BITMAP *result = load_bitmap(filename, 0);
  if (!result) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    printf("Could not load resource: %s\n", filename);
    exit(0);
  }
```

```
    return result;
  }
```

The C++ stream 'cout' could be used in place of printf(...), but printf is good practice for the Allegro text output commands, which take the same formatting strings as printf. See Section 'Text Output'for more information.

* See the section on 'Colors'for how to pass a color value to the clear_to_color(...) function.

4. Video Modes:

```
 set_color_depth(depth)
 set_gfx_mode(GFX_AUTODETECT|GFX_AUTODETECT_WINDOWED, screen_width, screen_height, 0,
0);
```

Color depth can be either 8, 15, 16, 24 or 32, and should always be selected before entering a video mode with set_gfx_mode. The 8-bit color depth uses the standard VGA palette, the same as QBASIC, but isn't as good for complicated graphics.

Valid screen dimensions include: 320x240, 640x480, 800x600 and 1024x768.

Not all monitors support every color depth, so truecolor (non 8-bit) applications should be ready to try different color depths if one color depth fails. This can be done by checking the return value of set_gfx_mode: if the returned value is nonzero, then the color depth is unavailable.

To find an available color depth:

```
int scr_width = 640, scr_height = 480;

int depth_select(int depth) {
  set_color_depth(depth);
  return set_gfx_mode(GFX_AUTODETECT, scr_width, scr_height, 0, 0) == 0;
}

int main() {
  allegro_init();
  depth_select(32) || depth_select(24) || depth_select(16) || depth_select(15);
}
```

5. Double Buffering:

Drawing directly to the screen invariably results in flicker. Due to the undesirable nature of flicker, programmers historically developed complicated methods of erasing old pixels while simultaneously drawing new pixels to the screen. Today, those methods are both inefficient (in terms of code complexity) and counter-intuitive. Double-buffering solves all these problems, and makes the act of erasing old images a thing of the past.

Double Buffering Method:

Create a bitmap which is not visible, but is the same size as the screen. This is called the "back buffer".
Do
  Clear the back buffer.

Draw all graphics to the back buffer.
Copy the back buffer to the screen (or the "front buffer").
Repeat until the program is ended.

Using the double buffer method, all graphics are always redrawn to the newly cleared buffer, during every single frame. So there is a fundamental change from the QBASIC-style of program that would draw the background to the screen, forget about the background, and move objects around, restoring the background colors as the objects move. In the double-buffered rendering method, the background is ALWAYS copied entirely to the buffer, all objects are then drawn to the buffer, and the final resulting image is copied to the screen.

When used for rendering anything other than 3d graphics, the double buffer method often ends up being faster than drawing directly to the screen, because memory bitmaps can be accessed faster than video bitmaps.

Double-buffering with a black background:

```
init_all();      /* choose a video mode and install keyboard */
BITMAP *buffer = create_bitmap(scr_width, scr_height);
while (!key[KEY_ESC]) {
  clear(buffer);
  /* ...draw all objects... */
  blit(buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h);
}
```

Double-buffering with a full-screen bitmapped background:

```
init_all();      /* choose a video mode and install keyboard */
BITMAP *buffer = create_bitmap(scr_width, scr_height);
BITMAP *background = load_bitmap("background.bmp", 0);
while (!key[KEY_ESC]) {
  blit(background, buffer, 0, 0, 0, 0, background->w, background->h);
  /* ...draw all objects... */
  blit(buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h);
}
```

For information on how Allegro's keyboard input routines work, see the section 'Keyboard Input'.

6. Colors:

```
makecol(red, green, blue)
getr(color)
getg(color)
getb(color)
hsv_to_rgb(hue, sat, lum, &r, &g, &b)
rgb_to_hsv(r, g, b, &hue, &sat, &lum)
bitmap_mask_color(bmp)
```

Traditionally, displays have used 256-color palettes. Many programmers become familiar with the VGA palette, where 0=black, 1=blue, 15=white, etc. Those color values can still be used under Allegro, when in an 8-bit video mode. However, only 256 colors

quickly becomes limiting when designing graphics for a game. So many programmers turn to truecolor display modes for the higher graphics potential offered by a greater range of colors. This is the strength of truecolor video modes: up to 16,777,216 colors can be displayed on the screen at once. The weakness is that drawing to the screen can be slower.

In truecolor video modes, bitmap_mask_color(bmp) will always return the shade of magenta that is used for transparency: makecol (255, 0, 255). In 8-bit modes, bitmap_mask_color(bmp) will be color zero, the transparent color.

To make color values in a truecolor video mode, red, green and blue components are combined by the function makecol(...). Makecol() can be used in 8-bit modes as well, but the color chosen is not always in the palette.

Red, green and blue values range from 0 to 255. Makecol takes three color components and combines them into the integer format used by the graphics commands. Getr, getg and getb can be used to decompose a color value back into its red, green and blue color components. Functions like getpixel(...) return color values that are meaningless without splitting them up into color components, so these commands are very useful.

Some color values:

```
black       = makecol( 0,   0,   0   ) =  0
grey        = makecol( 128, 128, 128 )
white       = makecol( 255, 255, 255 )
red         = makecol( 255, 0,   0   )
green       = makecol( 0,   255, 0   )
blue        = makecol( 0,   0,   255 )
yellow      = makecol( 255, 255, 0   )
orange      = makecol( 255, 128, 0   )
cyan        = makecol( 0,   255, 255 )
purple      = makecol( 255, 0,   255 )
light blue  = makecol( 128, 128, 255 )
pink        = makecol( 255, 128, 128 )
light green = makecol( 128, 255, 128 )
dark purple = makecol( 128, 0,   128 )
```

Jump on into Photoshop or Paint Shop Pro to find your favorite color's RGB values. In general, any light color can be made by taking the basic color and increasing any zero values to 128 or more. Dark colors can be made by taking the basic color and halving all color values.

Example - an image 'lighten' filter:

```
for (int y = 0; y < image->h; y++) {
  for (int x = 0; x < image->w; x++) {
    int color = getpixel(image, x, y);
    int r = getr(color) + 100;
    int g = getg(color) + 100;
    int b = getb(color) + 100;
    if (r > 255) { r = 255; }
    if (g > 255) { g = 255; }
    if (b > 255) { b = 255; }
    putpixel(image, x, y, makecol(r, g, b));
  }
}
```

7. Timers:

```
volatile double elapsed_time = 0;
void __inc_elapsed_time() { elapsed_time += .001; }
END_OF_FUNCTION(__inc_elapsed_time);

int main() {
  allegro_init();
  install_timer();

  LOCK_VARIABLE(elapsed_time);
  LOCK_FUNCTION(__inc_elapsed_time);
  install_int_ex(__inc_elapsed_time, BPS_TO_TIMER(1000));
  return 0;
}

END_OF_MAIN();
```

Somewhat cumbersome, but the above code gives a timer accurate to a couple milliseconds. Just check elapsed_time for the floating-point time in seconds. BPS_TO_TIMER(1000) means to make a timer function that runs 1000 times per second, and .001 is the reciprocal of 1000. Note that timers of higher resolutions are not worth making, because Windows will cause the time to be rounded off to the nearest 5 milliseconds, regardless of how fast you try to make a timer.

A more advanced way of tracking time can often be useful for games: declare the global variables 'last_time' and 'dt' as floating-point values. Each pass through the main game-logic loop, compute 'dt' as the difference of elapsed_time and last_time, and then set last_time to elapsed_time. Then, objects can be moved at a constant speed regardless of processor speed by adding velocity times dt to the current object position:

```
x += 100 * dt;
```

Where 100 is the velocity in pixels/second.

If 'dt' is consistantly used for moving objects, then taking a program like exbounce.cppto a very fast computer or a very slow computer will only alter the framerate; the objects will bounce around the screen at the same speed.

8. Text Output:

```
textprintf_ex(bmp, font, x, y, color, -1, string, ...)
textprintf_centre_ex(bmp, font, x, y, color, -1, string, ...)
textprintf_right_ex(bmp, font, x, y, color, -1, string, ...)
```

The standard output (cout or printf) cannot be used reliably in graphics modes, so Allegro provides alternate text-printing functions. 'font' is a global variable that contains an 8x8 font, and can be used for simple text output. The routines above draw a font with 'color' as the foreground color and -1 (transparent) as the background color. Textprintf uses a printf-style format string to print numbers, strings, and so on. Printf-style strings use the following format to insert variables into the output:

```
%f      - print a float value
%i      - print an integer value
%.2f    - print a float value with 2 digits after the decimal
%4.2f   - print a float value with 4 digits before the decimal and two after
%04i    - print an integer with 4 digits before the decimal
%s      - print a string (from a character pointer)
```

Examples:

```
int ival = 10;
double fval = 15.1;
char *str = "Hello World";
textprintf(bmp, font, 0, 0, makecol(255, 0, 0), "Red Text");
int white = makecol(255, 255, 255);
textprintf_centre(bmp, font, scr_width/2, scr_height/2, white, "Centered");
textprintf(bmp, font, 0, 10, white, "An integer: %i", 10);
textprintf(bmp, font, 0, 20, white, "Integer padded with zeros: %03i", ival);
textprintf(bmp, font, 0, 30, white, "A float: %f", fval);
textprintf(bmp, font, 0, 40, white, "2 digits after decimal: %.2f", 23.2134);
textprintf(bmp, font, 0, 50, white, "A string: %s", str);
```

9. Keyboard Input:

```
install_keyboard()
int keypressed()
int readkey()
key[]
```

Call install_keyboard() any time after allegro_init() to set up the keyboard input routines.

The first type of keyboard input is easy: keypressed() will return whether a key was pressed, and readkey() will return that key. Similar to conio's kbhit() and getch(). One thing of note is that readkey() will return the key with extra keyboard status information, which you probably don't want. To remove the extra information, 'mod' the returned value with 256:

```
char ch = readkey() % 256;
```

If no keys are currently pressed, readkey() will wait for a keypress.

The second type of keyboard input is more powerful: an array of keys indicating whether a given key is held down. This key status information is vital for smooth motion in games.

```
if (key[KEY_LEFT]) { /* do something if left arrow is held */ }
if (!key[KEY_ENTER]) { /* do something if enter is not held */ }
```

Key codes:

```
KEY_A     -    KEY_Z
KEY_0     -    KEY_9
KEY_F1    -    KEY_F12

KEY_ESC         KEY_MINUS      KEY_ENTER      KEY_EQUALS     KEY_TAB
KEY_BACKSPACE   KEY_OPENBRACE  KEY_CLOSEBRACE KEY_LCONTROL   KEY_RCONTROL
KEY_COLON       KEY_TILDE      KEY_QUOTE      KEY_LSHIFT     KEY_RSHIFT
KEY_BACKSLASH   KEY_COMMA      KEY_STOP       KEY_SLASH      KEY_ASTERISK
KEY_ALT         KEY_SPACE      KEY_CAPSLOCK   KEY_NUMLOCK    KEY_SCRLOCK
KEY_HOME        KEY_END        KEY_PGUP       KEY_PGDN       KEY_MINUS_PAD
KEY_MENU        KEY_PLUS_PAD   KEY_INSERT     KEY_DEL        KEY_PRTSCR
KEY_UP          KEY_DOWN       KEY_RIGHT      KEY_LEFT       KEY_PAUSE
```

10. Mouse Input:

```
install_mouse()
mouse_x
mouse_y
mouse_b
set_mouse_speed(xspeed, yspeed)
```

Call install_mouse() before using the mouse commands. Mouse_x and mouse_y are integer variables that are always equal to the mouse position. Mouse_b contains all the mouse buttons, but individual buttons can be checked:

```
if (mouse_b) { /* any mouse button is pressed */ }
if (mouse_b & 1) { /* left mouse button is held */ }
if (mouse_b & 2) { /* right mouse button is held */ }
if (mouse_b & 4) { /* center mouse button is held */ }
```

In Allegro, by default, no mouse cursor is drawn to the screen. To make the mouse cursor visible, simply load a mouse cursor bitmap and draw it to the offscreen buffer directly before copying to the screen:

```
BITMAP *cursor = load_bitmap("cursor.bmp", 0);
while (!key[KEY_ESC]) {
  clear(buffer);
  draw_sprite(buffer, cursor, mouse_x, mouse_y);
  blit(buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h);
}
```

The mouse speed defaults to '2' in both x and y directions. Higher values correspond to lower actual mouse speeds: set_mouse_speed(1, 1) is the fastest, and set_mouse_speed(10, 10) is a tenth as fast.

Often, a programmer will want to check whether a mouse button has just been clicked, i.e. gone from the up state to the down state. Checking a mouse click can be done with the following code:

```
int mouse_clicked, last_button, current_button=0;
while (not exited) {
  last_button = current_button;
  current_button = mouse_b;
  mouse_clicked = (current_button && !last_button);
  if (mouse_clicked) {
    /* happens once per click as mouse is pressed down */
  }
  /* more code */
}
```

Checking for a mouse 'declick' event can be done in the same way.

11. Sound Output:

```
install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, 0)
SAMPLE *sample = load_sample("filename.wav|voc")
play_sample(sample, volume, pan, frequency, loop)
stop_sample(sample)
destroy_sample(sample)
MIDI *midi = load_midi("filename.mid")
play_midi(midi, loop)
destroy_midi(midi)
set_volume(sample_volume, midi_volume)
```

In DOS, Allegro's sound output tends to either:

1. Not work.
2. Sound horrible.

However, don't be discouraged. The sound quality is fine under the Windows version of Allegro, which can be compiled without too much trouble under Visual C/C++ or Mingw. Or you can use the DOS version and ignore the white noise in the sound output.

To use the sound functions, first call install_sound(...). If you are not planning on using either the digital sample output or the MIDI output, you can place zeros in place of DIGI_AUTODETECT or MIDI_AUTODETECT.

Once the sound driver has been initialized, call load_sample(...) and play_sample(...) anywhere in your program as you see fit. Multiple sounds can be played at once, and will be mixed. For the play_sample function, sound volume ranges from 0...255, panning ranges from 0...255 (0 is left speaker, 128 is center, 255 is right speaker), frequency represents the sound playback speed: 1000 will play the sample at normal rate, 500 will play it at half rate, etc. If the loop parameter is nonzero, the sound will loop until stopped by stop_sample(...).

Set_volume(...) will alter the global sound volume for digital and MIDI playback. Volume ranges 0...255.

12. Collision Detection:

```
Pseudo-code for collision-detection routine:

For every object in the game
  For every other object
    Check if first object collides with second
    If so, process the collision:
       first.collide_event(second)
       second.collide_event(first)
    End If
  Next
Next
```

The simplest form of collision detection can be done by treating every object in the game as a rectangle and checking if any rectangles collide.

Two rectangles of coordinates (xmin1, ymin1)-(xmax1, ymax1) and (xmin2, ymin2)-(xmax2, ymax2) are overlapping if the following is true:

```
if (xmin1 < xmax2 && ymin1 < ymax2 && xmin2 < xmax1 && ymin2 < ymax1) {
  /* objects collided */
}
```

This condition works in three dimensions as well, and the rule for remembering it is: "The mins are less than the maxes". If any minimum value for one rectangle is less than the other rectangle's maximum value for the same dimension, then the two rectangles collided. This equation can be derived by considering the coordinates of the overlapping region of the two rectangles.

Normally, the coordinate minimums and maximums are computed inside a check_collide(a, b) type routine, and are based on the sprite's width and height. That function would presumably return either true or false, depending on whether the two sprites collide.

The rectangle-bounded collision detection often works very well; try it before giving up and using guess-math to determine if, say two Mortal Kombat style fighters are in contact. However, if two sprites contain great amounts of empty space, then collisions can occur when they shouldn't; a player's missiles explode without actually touching a boss, a player skims close to an enemy and finds his ship has been annihilated because he moved through the enemy's bounding box. If these sorts of things become too obvious, then another bounding shape is called for. In 2D, checking whether individual pixels actually overlap is quite fast. This is called "pixel-perfect" collision and is demonstrated in experfect.cpp.

13. Translucency:

```
set_trans_blender(0, 0, 0, opacity)
draw_trans_sprite(bmp, sprite, x, y)
drawing_mode(DRAW_MODE_TRANS, 0, 0, 0)
  /* ...any graphics commands... */
drawing_mode(DRAW_MODE_SOLID, 0, 0, 0)

/* Other useful blenders are:
   set_add_blender(0, 0, 0, opacity) - lightening effect, useful for particles.
   set_multiply_blender(0, 0, 0, 255) - multiply, often used for lightmaps. */
```

Set_trans_blender should always be called before drawing translucent graphics. Opacity is an integer value from 0...255 representing the solidity of the drawn object. Normal graphics commands (those in the 'Graphics'section) are only translucent if drawing_mode is called with the DRAW_MODE_TRANS parameter:

```
set_trans_blender(0, 0, 0, 128);
drawing_mode(DRAW_MODE_TRANS, 0, 0, 0);
circlefill(buffer, 160, 100, 80, makecol(0, 255, 0));
circlefill(buffer, 220, 200, 80, makecol(0, 255, 255));
circlefill(buffer, 100, 200, 80, makecol(0, 0, 255));
drawing_mode(DRAW_MODE_SOLID, 0, 0, 0);
```

14. Angles and Math:

Allegro uses a bizarre angle format for angles in all graphics commands. To convert to an Allegro angle:

```
#include <math.h>
#define DEGREES(x) int((x)/360.0*0xFFFFFF)
#define RADIANS(x) int((x)/2/M_PI*0xFFFFFF)
```

Then simply use the conversion macros when an angle is called for, as in the following example:

```
arc(bmp, x, y, DEGREES(45), DEGREES(180), radius, color);
  - or -
arc(bmp, x, y, RADIANS(M_PI/4), RADIANS(M_PI), radius, color);
```

To find the angle from one point (x1, y1) to another (x2, y2), use the following command:

```
#include <math.h>
double angle = atan2(y2-y1, x2-x1);
```

Note that the angle returned will be in radians, and will be measured clockwise from the right horizontal, opposite that in math classes.

To compute the (x, y) position of a radial point rotated angle radians around the origin:

```
#include <math.h>
double x = cos(angle) * radius;
double y = sin(angle) * radius;
```

The (x, y) values generated above can easily generate circular motion for an object by using a timer for the angle and a constant radius.

15. Examples:

Example programs are provided with Allegro. Additional examples are also available with this quick reference (download examples):

```
exbitmap.cpp:  Truecolor bitmaps and double buffering example.
exbounce.cpp:  Double buffering with moving sprites example.
exmag.cpp:     Magnification effect, uses 8-bit (palettized) color.
exmouse.cpp:   Detecting mouse input and drawing mouse sprite.
experfect.cpp: Pixel-perfect collision.
expixel.cpp:   Double-buffering and getpixel()/putpixel().
explasma.cpp:  Panning background example with double buffering.
exrotate.cpp:  Rotating sprite with double buffering.
exshade.cpp:   2D polygon rendering via polygon3d_f(...).
extext.cpp:    Time routines and text output.
extrans.cpp:   Translucency via draw_trans_sprite().
```

16. Technical Information:

Allegro is available: http://www.talula.demon.co.uk/allegro
DJGPP (DOS GCC) is available: http://www.delorie.com/djgpp
Mingw (Windows GCC) is available: http://www.mingw.org/

Typically, you will want the stable branch source code (4.x.x). On Windows, it is easiest to compile and install Allegro with Mingw: install Mingw, unzip the dx70_mgw.zip file to your Mingw directory, then

```
% cd allegro
% fix mingw32
% set MINGDIR=(Full path to Mingw installation)
% make
% make install
```

Now copy allegro/lib/mingw32/alleg4x.dll to your Windows/System32 directory.

Compile and link with:

```
g++ source_file[s].cpp -o out.exe -O3 -Wall -lalleg -s
```

To stop the DOS box from appearing on Windows, use the -mwindows flag.

For more precise directions, see allegro/docs/build/your_compiler.txt. For the general Allegro documentation, see allegro/docs/txt/allegro.txt. Also note that most of Allegro's graphics commands are rendered purely in software. Thus, the rendering speed and frames/second of any application are directly proportional to the CPU speed, and nothing else. With Allegro programs, an onboard video card will perform the same as a $500 NVidia card.
