# Allegro & freeBASIC

First Edition

By Avinash "aetherFox" Vora
avinashvora@gmail.com
http://fballeg.apeshell.net



## Thanks

This document could not have been written without the fantastic help provided by na_th_an and DrV. Their help has been invaluable in my progress with Allegro. I would like to thank them for sleepless nights of programming help, bug fixes, proof reading and everything else.

Of course, I am also grateful to the creators of Allegro and freeBASIC. Without them, this would be nothing. I also want to thank VonGodric, who's FBIDE was used extensively for outputting the formatted, highlighted syntax that you see in this tutorial.

## Who is this tutorial for?

This tutorial is for anyone fluent in freeBASIC. I am assuming no prior knowledge or experience with any library, Allegro or not. This tutorial is also for those freeBASIC programmers who are newcomers to game programming, or experienced programmers who want to learn about proper programming style.

While freeBASIC and Allegro both are cross-platform, for the sake of simplicity, I am assuming that the platform used it Windows. However, the actual source should work on any platform that both freeBASIC and Allegro share compatibility with.

This tutorial is also for people who want a no-nonsense, professional guide to using Allegro with freeBASIC. I have planned to create a reference manual alongside this series of tutorials, one that could hopefully become a standalone reference manual for Allegro with freeBASIC.

You will not learn about 3D gaming techniques, or the methods of accessing complex mainstream technologies such as pixel shaders. You will find techniques and information on the processes involved with writing 2D games using Allegro and freeBASIC.

## Requirements

You will need the following to compile and run the examples presented in this document:

- **freeBASIC**
  *The examples were tested using freeBASIC v0.13b. This version of freeBASIC includes the necessary headers.*
- **Allegro**
  *You will only need the DLL (alleg40.dll) which will need to be placed in either the same path as the compiled executable, or in your system32 folder under the Windows installation directory to make the DLL accessible to all executables that call Allegro routines.*
- **FBIDE** [optional]
  *This is entirely optional, many people prefer to use their own plaintext editors, but I find that FBIDE does a fairly decent job at being an IDE for fbc.exe.*

## What is Allegro?

Allegro (http://alleg.sf.net) is a library providing low-level routines to programmers that generally deal with games. The type of routines Allegro offers includes ones for graphics, input, precision timing, etc. It is open source software, which means that it is free, and, it is always growing. Comprehensive features are frequently added, but still, the library is vast and extensive, and you will find almost every single routine you need.

## Why use Allegro with freeBASIC?

When freeBASIC first released, I heard little about anything but SDL and OpenGL. While I am not disputing the power, flexibility and usefulness of either of these fantastic libraries, I was immediately discouraged from trying to learn them. I use BASIC because of the ease of use that it presents with complex tasks, and OpenGL and SDL did not keep that fun factor for me with their steep learning curves.

I always knew about Allegro from when I learnt basic C++, but never used it. I recently came across the Allegro headers DrV ported over from the original distribution, and decided to learn how to use the library. It is (in my opinion) easier to use Allegro for graphics programming than it is to use the GFXLib commands. It is intuitive and ingeniously simple. After 15 minutes with the library, I was creating a double-buffering example.

I found next to nothing in terms of resources for Allegro within the BASIC community, and so consulted the Allegro documentation, articles and tutorials and ported the code into BASIC when required. It was simple to pick up Allegro even with this obscure method, and my goal for this tutorial is to make the process even easier for you.

## Getting started

Almost every single programming tutorial that is worth its cheese starts off with a "Hello World!" example, and who am I to stray from tradition:

```
#include "allegro.bi"

'Initialise Allegro
allegro_init

'Create message box
allegro_message "Hello World!"

allegro_exit

end 0
```

SOURCE LISTING 1

`allegro_init` needs to be called before you use any of Allegro's routines or functions; it initializes some important things in the library, and besides, it is essential.

`allegro_exit` should be called at program termination. It is good programming practice, as it de-initialises everything properly.

`allegro_message` is a routine that creates a message box with text passed to it as a parameter. While it is not practical for main program flow, it is useful for error reporting, as we will shortly see.

This example is limited, and the next one will introduce the initializing of graphics modes, and outputting text to the screen.

## Graphics modes

Setting a graphics mode in Allegro requires the use of the `set_gfx_mode()` function. Its syntax is as follows:

```
set_gfx_mode (byval card as integer, byval w as integer, byval h as
integer, byval v_w as integer, byval v_h as integer)
```

SYNTAX LISTING 1

The parameters of the function do the following:

- `card` – would usually be one of `GFX_AUTODETECT`, `GFX_AUTODETECT_WINDOWED` or `GFX_AUTODETECT_FULLSCREEN`. `GFX_AUTODETECT` detects the default graphics mode for the computer, which usually comes out to be full screen. The other two force a full screen or windowed view. For a full list of the possible values for this parameter, consult the manual at http://alleg.sf.net.
- `w`, `h` – the `w`idth and `h`eight of the screen (the resolution) in pixels.

- `v_w`, `v_h` – the width and height of the virtual screen. For the time being, leave these at 0. Virtual screens are a tricky topic that will be discussed at a later date.

### Error checking

Generally, this function is used in conjunction with a check to make sure the video card supports the selected screen mode. While in this tutorial we will not be doing a fail-safe by reverting to another screen mode, it is simple by using an `if…then…else` check and changing to a supported screen mode.

```
if set_gfx_mode (GFX_AUTODETECT_WINDOWED, 320, 240, 0, 0) < 0 then
   allegro_message Unable to set the graphics mode!" + chr(13) +
allegro_error + chr(13)
   end 1
end if
```

SOURCE LISTING 2

The above code will attempt to set the graphics mode to a screen of resolution 320 by 240 pixels, and if unsuccessful, it will gracefully exit with an error message.

Checks like this are important not only in games, but in all types of applications. A user seeing a cryptic error message that is not going to help them at all will not see wonders in the user-friendliness of the program. In this vein of thought, you should account for all possible errors and try to create a fix for them that will allow a user to continue using the program.

### Outputting information to the screen

To see an implementation of using graphics modes, consider source listing 3:

```
#include "allegro.bi"

'Declarations
dim white as integer

'Initialise Allegro
allegro_init

'Initialise Allegro's keyboard routines
install_keyboard

'Set 8-bit colour depth
set_color_depth 8

'Set graphics mode to 320x240 with error checking
if set_gfx_mode (GFX_AUTODETECT_WINDOWED, 320, 240, 0, 0) < 0 then
   'set_gfx_mode(GFX_TEXT, 0, 0, 0, 0)
   allegro_message Unable to set the graphics mode!" + chr(13) +
allegro_error + chr(13)
   end 1
end if
```

```
'Precalculate the colour white
white = makecol (255, 255, 255)

'Write "Hello World!" onto the screen.
textout screen, font, Hello World!", 10, 10, white

'Wait for a key press
readkey

remove_keyboard

allegro_exit

end 0
```

SOURCE LISTING 3

`install_keyboard` is a routine that initialises Allegro's keyboard routines. This must be called before using any of Allegro's keyboard features. In this particular example, it is called for the use of the `readkey` routine.

`remove_keyboard` is a de-initialisation routine that for the keyboard handler. Similar to `allegro_exit`, it is not needed, but is good programming practice.

`makecol` is a function that converts colours from a hardware independent format (red, green, and blue values ranging 0-255) to the pixel format required by the current video mode, calling the preceding 8, 15, 16, 24, or 32 bit `makecol` functions as appropriate. By pre-calculating it, you can gain significant speed, rather than calculating it every time you need to use it. While you may not be able to pre-calculate every possible colour, ones that you know you will use frequently should be pre-calculated for speed.

Its syntax is as follows:

```
makecol (byval r as integer, byval g as integer, byval b as integer)
```

SYNTAX LISTING 2

`set_color_depth` is a routine that should be called before every `set_gfx_mode` function is called. It sets the colour depth to the parameter passed to it (8, 16, 24 or 32 bit).

`textout` is a routine that writes a defined string, of a predefined font, onto a predefined `BITMAP` (the concept of `BITMAP`s will be explained later), at a defined position on screen, in a defined colour. The syntax is as follows:

```
textout (byval bmp as BITMAP ptr, byval f as FONT Ptr, byval s as
string, byval x as integer, byval y as integer, byval c as integer)
```

SYNTAX LISTING 3

In source listing 3, the pointer to `BITMAP` is `screen`. `screen` is defined in the Allegro headers (actually it is defined when `set_gfx_mode` is called), and

is the `BITMAP` for the visible screen. Similarly, the pointer to `FONT` is `font`, the default font, also predefined in Allegro headers.

There are other ways to write text to the screen, for example by having different alignments (e.g. `textout_center`) or changing the text options (e.g. `text_length`). These functions are documented in the manual at http://alleg.sf.net or http://www.allegro.cc.

`readkey` is a function that returns the next keyboard character pressed in ASCII format. If the buffer is empty, it waits until a key is pressed. While Allegro has better functions for detecting key presses, in this particular situation, `readkey` is ideal in its simplicity. As it is a keyboard routine, it requires the calling of `install_keyboard`.

## Graphics primitives

While most games might not ever use graphics primitives, it is extremely useful to learn them. They can be used for many things such as testing. However, certain styles of games use graphics primitives exclusively for creating the graphics – the wire frame "2D vector" style games.

Allegro follows a nice, simple intuitive system that is common throughout most of the routines you will come across – the BPC principle.

### Baked Potato & Cheese

In general, when passing parameters to many graphics routines, the order might not come to you immediately. The BPC principle is your solution:

- **B** – `BITMAP` you want to draw graphics to.
- **P** – Position you want to draw the graphic – this is two parameters, x and y coordinates.
- **C** – Colour you want to draw with.

The simple anagram Baked Potato & Cheese should help you remember what order to pass your parameters in.

### A pretty picture

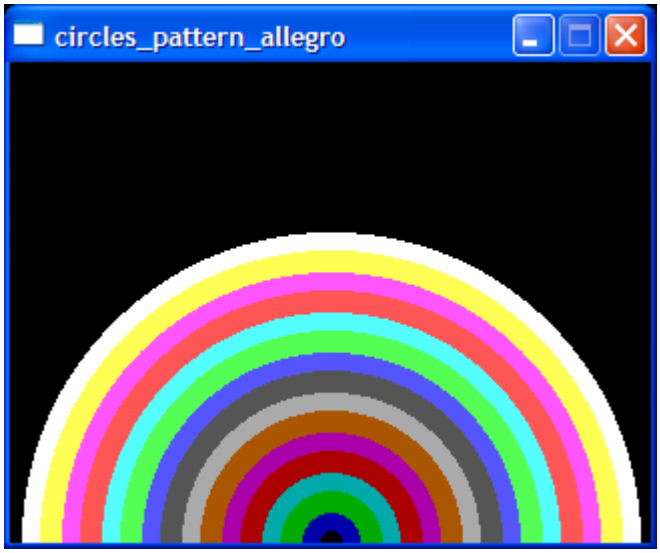
A simple example of using the filled circle function is as follows:

```
#include "allegro.bi"

'Declarations
dim i as integer

'Initialise Allegro routines
allegro_init
install_keyboard

'Set 8-bit colour depth
```

```
set_color_depth 8

'Set graphics mode to 320x240 with error checking
if set_gfx_mode(GFX_AUTODETECT_WINDOWED, 320, 240, 0, 0) < 0 then
        allegro_message Unable to set any graphic mode" + chr(13) +
allegro_error + chr(13)
        end 1
end if

'Draw a pattern with circles
for i = 16 * 10 to 0 step -1
   circlefill screen, SCREEN_W / 2, SCREEN_H - 1, i, i / 10
next i

'Wait for a key press
readkey

end 0
```

SOURCE LISTING 4

The output is as follows:




The main program flow does not need explaining, but the syntax for the `circlefill` routine is as follows:

```
circlefill (byval bmp as BITMAP ptr, byval x as integer, byval y as
integer, byval radius as integer, byval c as integer)
```

SYNTAX LISTING 3

`SCREEN_W` and `SCREEN_H` are extremely useful values that are defined when `set_gfx_mode` is called. They return the width and height of the screen in pixels. Using these as opposed to hard-coding pixel values is preferable, because then your engine becomes generic to whatever resolution is specified by `set_gfx_mode`.

A comprehensive list of graphics primitives is available in the Allegro manual (http://www.allegro.cc/manual/view_category.php?_id=13).

## Animating

Animating is the process of changing the appearance of graphics on the visual buffer over a period. This could be a running character, a flickering torch or even just the scrolling of the game screen. While for now we will be covering simple animation, the basic principles are the same no matter what is being done.

### A bouncing ball

The first example of proper animation is just a simple bouncing ball demonstration. As this is the first long source listing, I have added comments to mark off areas of the source. This is good programming practice, and can allow you to easily reference to sections of the

```
'General:
'--------

option explicit

#include "allegro.bi"

'Declarations:
'-------------

'Types

'Ball type
type BALL_TYPE
   x as single
   y as single
   speedx as single
   speedy as single
   radius as single
end type

'Create a ball object
dim ball as BALL_TYPE

'Check whether game is running
dim end_game as single

'Colours for precalculation
dim white as integer

'Initialisation:
'---------------

'Initialise Allegro routines
allegro_init
install_keyboard

'Set colour depth to 8-bit
set_color_depth 8

'Set graphics mode to 320x240 with error checking
```

```
if set_gfx_mode(GFX_AUTODETECT_WINDOWED, 320, 240, 0, 0) < 0 then
        allegro_message "Unable to set any graphic mode" + chr(13) +
allegro_error + chr(13)
        end 1
end if

'Initialise the variables

'Game is not ended yet
end_game = 0

'Initialise the ball
ball.x = SCREEN_W / 2
ball.y = SCREEN_H / 2
ball.speedx = 5
ball.speedy = 5
ball.radius = 2

'Precalculate colours for speed
white = makecol (255, 255, 255)

'Clear everything off the screen
clear_bitmap screen

'Clear the keyboard buffer
clear_keybuf

'Main Program Flow:
'------------------

'Main game loop while end_game is not set
do
   'Move the ball
   ball.x = ball.x + ball.speedx
   ball.y = ball.y + ball.speedy

   'Check for collisions against the sides
   if ball.x - ball.radius <= 0 or ball.x + ball.radius >= SCREEN_W
then ball.speedx = - ball.speedx
   'Check for collisions against the top
   if ball.y - ball.radius <= 0 or ball.y + ball.radius >= SCREEN_H
then ball.speedy = - ball.speedy

   'Draw the ball
   circlefill screen, ball.x, ball.y, ball.radius, white

   'Clear the previous loop's graphics
   clear_bitmap screen

   'Check for the key Escape, then end_game = 1
   if key(KEY_ESC) then end_game = 1
loop until end_game = 1

'De-initialise allegro
remove_keyboard
allegro_exit

end 0
```

SOURCE LISTING 5

The only thing new here are the keyboard functions, which will be covered later, when discussing input.

One important point to note is the use of flags for checking game progress. While it might seem useless to use the `end_game` flag in this situation, as the main loop becomes more and more complex, checking the end of the loop is much easier by setting the flag than by using complex `exit`s. All variables are also initialised from beforehand, grouped properly. This, along with commenting is programming style and practice, and should be enforced. While you may not follow my style, you should develop your own and keep to it.

Running this produces something that was not intended. Depending on your computer, you might see a large number of semi-transparent balls flickering all over the screen. In reality, there is one ball bouncing on the edges of the screen, but flicker occurs because video memory is slow, and during each loop, the entire screen is constantly cleared, redrawn to and updated. This process is slow and inefficient.

This is a simple demonstration of a bouncing ball, so you can only imagine how much flicker a fully-fledged game might have. There are two techniques to fixing this, and both are applied simultaneously to give the smoothest graphics.

### Wait for the vertical retrace

`vsync` is a function that simply waits for the vertical retrace. The vertical retrace is the time when the monitor is preparing to draw the next frame on the screen. If we erase all objects and draw them before the screen is updated, we should get a smoother view.

This sounds complicated, but really all it entails is calling the `vsync` function. It is a simple method of implementation to yield good results. Usually, it is between your drawing commands and your screen erasing commands. In our example, the lines before the `loop` end change to:

```
    'Draw the ball
    circlefill screen, ball.x, ball.y, ball.radius, white

    'Wait for the vertical retrace
    vsync

    'Clear the previous loop's graphics
    clear_bitmap screen

    'Check for the key Escape, then end_game = 1
    if key(KEY_ESC) then end_game = 1
loop until end_game = 1
```

SOURCE LISTING 6

This solution should present a much better view of what was intended by this little demonstration program. The ball should now bounce around the screen with no flicker at a reasonable speed.

As great a `vsync` is, it is usually not enough when the drawing to the screen becomes more intensive. Even on this demonstration, slower computers might notice slight flicker. The solution to almost all flicker problems comes in the form of a technique that is extremely common in 2D games, and is the standard method (and has been for the last 15 years at least) for eliminating flicker on almost all machines: **double buffering**.

## Double buffering

The technique of double buffering involves drawing all the graphics to a temporary, off-`screen` `BITMAP`, until a whole frame has been updated, and then copying the final image to the real `screen` memory in one big chunk.

This is effective because we aren't erasing the screen, just replacing what is already there with the contents of the buffer. Video memory if slow, and overwriting areas of the screen a lot when drawing a frame is slow. System memory, where a temporary buffer is stored, is much faster, and so doing all the writes to system memory, and copying the entire buffer over in one go to screen memory is more efficient.

This is also better in the long run, because reading from the screen memory is usually even slower, and so by having all the graphics on the buffer, you can read from there and have considerable speed advantages.

Implementing a double buffering system means a few changes and additions to our source:

```
'ball_bounce.bas

'General:
'--------

option explicit

#include "allegro.bi"

'Declarations:
'-------------

'Types

'Ball type
type BALL_TYPE
   x as single
   y as single
   speedx as single
   speedy as single
   radius as single
end type
```

```
'Create a ball object
dim ball as BALL_TYPE

'Create an Allegro BITMAP for double buffering
dim dblbuffer as BITMAP ptr

'Check whether game is running
dim end_game as single

'Colours for precalculation
dim white as integer

'Initialisation:
'---------------

'Initialise Allegro routines
allegro_init
install_keyboard

'Set colour depth to 8-bit
set_color_depth 8

'Set graphics mode to 320x240 with error checking
if set_gfx_mode(GFX_AUTODETECT_WINDOWED, 320, 240, 0, 0) < 0 then
        allegro_message "Unable to set any graphic mode" + chr(13) +
allegro_error + chr(13)
        end 1
end if

'Set window title to "Bouncing ball"
set_window_title "Bouncing ball"

'Create the double buffer surface
dblbuffer = create_bitmap (SCREEN_W, SCREEN_H)

'Check to make sure there was enough memory to create the surface.
if dblbuffer = NULL then
   allegro_exit
   allegro_message "Sorry, not enough memory."
   end 1
end if

'Initialise the variables

'Game is not ended yet
end_game = 0

'Initialise the ball
ball.x = SCREEN_W / 2
ball.y = SCREEN_H / 2
ball.speedx = 5
ball.speedy = 5
ball.radius = 2

'Precalculate colours for speed
white = makecol (255, 255, 255)

'Clear everything off the screen and buffer
clear_bitmap screen
```

```
clear_bitmap dblbuffer

'Clear the keyboard buffer
clear_keybuf

'Main Program Flow:
'------------------

'Main game loop while end_game is not set
do
   'Move the ball
   ball.x = ball.x + ball.speedx
   ball.y = ball.y + ball.speedy

   'Check for collisions against the sides
   if ball.x - ball.radius <= 0 or ball.x + ball.radius >= SCREEN_W
then ball.speedx = - ball.speedx
   'Check for collisions against the top
   if ball.y - ball.radius <= 0 or ball.y + ball.radius >= SCREEN_H
then ball.speedy = - ball.speedy

   'Draw the ball to the buffer
   circlefill dblbuffer, ball.x, ball.y, ball.radius, white

   'Wait for the vertical retrace
   vsync

   'Blit the contents of the buffer to the visible screen
   blit dblbuffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H

   'Clear the previous loop's buffer contents
   clear_bitmap dblbuffer

   'Check for the key Escape, then end_game = 1
   if key(KEY_ESC) then end_game = 1
loop until end_game = 1

'Destroy the double buffer and remove it from memory.
destroy_bitmap dblbuffer

'De-initialise allegro
remove_keyboard
allegro_exit

end 0
```

SOURCE LISTING 7 – BALL_BOUNCE.BAS

This source listing is available in the `/examples` directory that came in the original archive.

The first thing that needs to be done is to dimension the dblbuffer pointer to the `BITMAP` structure. The pointer is an address to the memory location that contains the actual information in the buffer. Therefore, just declaring the pointer does not create the buffer. To create the buffer, you need to invoke the create_bitmap function as follows:

```
'Create the double buffer surface
dblbuffer = create_bitmap (SCREEN_W, SCREEN_H)
```

SOURCE LISTING 8

Once the bitmap is created, it can be drawn to. Before this, it is good practice to clear the bitmap (all `BITMAP`s should be cleared at the start of the program). This is done using the `clear_bitmap` function.

Now that everything has to be drawn to `dblbuffer`, the `circlefill` command needs to be changed to make it draw to `dblbuffer` instead of screen.

After calling `vsync`, we need to put everything on `dblbuffer` onto `screen`. This is done by a routine called `blit`. Blitting is the process of copying from one buffer to another. The syntax of `blit` is:

```
blit (byval source as BITMAP ptr, byval dest as BITMAP ptr, byval
source_x as integer, byval source_y as integer, byval dest_x as
integer, byval dest_y as integer, byval width as integer, byval
height as integer)
```

SYNTAX LISTING 4

This is one of the most useful features of Allegro, and from the parameters, you can see it is very versatile. For example, you can copy part of a buffer to a specified location on the screen. While for now you can leave the last 4 parameters at 0, in the future you may use them.

Previously we cleared the screen, but as discussed this is slow, and instead, we should clear the temporary buffer, `dblbuffer`.

Once the program is complete (`end_game = 1`), then the buffer is no longer needed. The `destroy_bitmap` function removes the buffer from memory, and while it is not required, it should be included for good programming practice and proper de-initialisation.

## Keyboard handling

All games follow three stages as they progress through their main program loop: **input**, **process** and **output**. Input is a very important aspect to game programming, because it can make or break a game. There is a lot to consider, such as the control scheme, the influence the control has on variables etc.

Allegro has the capability to handle mouse and joystick input, however, for now we are going to focus on keyboard handling.