The Gameduino 2 Tutorial, Reference and Cookbook

James Bowman

`jamesb@excamera.com`

Excamera Labs,
Pescadero CA,
USA

First edition: December 2013

Contents

Part I

Tutorial

Chapter 1

Plug in. Power up. Play something

1. Download the Gameduino 2 library `Gameduino2.zip` from `http://gameduino.com/code` and install it in the Arduino IDE. Instructions for doing this are at `http://arduino.cc/en/Guide/Libraries`.

2. Attach Gameduino 2 to the Arduino, making sure that the pins are aligned properly, and that none are bent.

3. Power up the Arduino. Nothing will appear on the Gameduino 2 screen

until a sketch is loaded on the Arduino.

4. Restart the Arduino IDE, and load one of the sketches, for example
   `File ▷ Sketchbook ▷ libraries ▷ Gameduino2 ▷ 1.Basics ▷ helloworld`
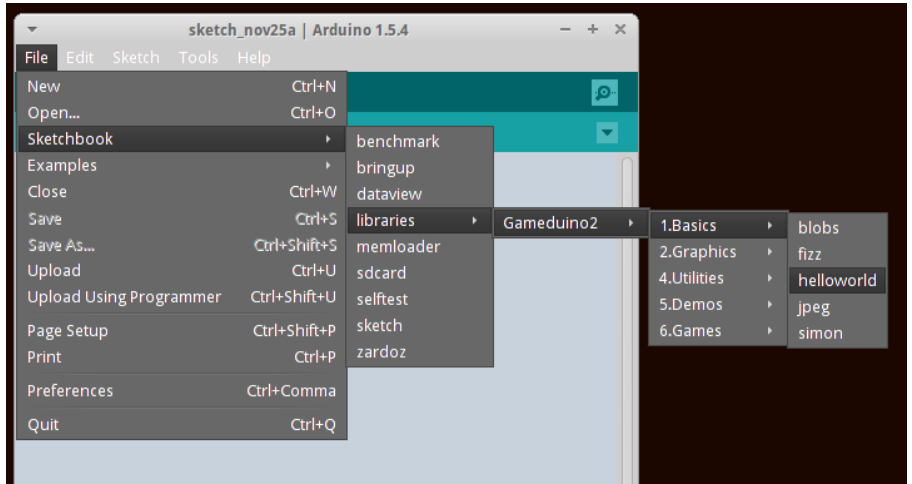
5. Compile and download the sketch to the Arduino. Gameduino 2 should
   start up and run the sketch.

6. Many of the examples use data files from the microSD card. To run them,
   format a microSD card and copy the files from `Gameduino2sd.zip` (also
   at `http://gameduino.com/code`) onto it.

7. Enjoy!

Chapter 2

Quick start

This chapter introduces the basics of Gameduino 2 - from "hello world" to creating a simple touch-based game.

2.1   Hello world


```
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

void setup()
{
  GD.begin();
}

void loop()
{
  GD.ClearColorRGB(0x103000);
  GD.Clear();
  GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
  GD.swap();
}
```

The code in `loop()` clears the screen to a relaxing deep green color, then writes the text centered on the screen. This code runs continuously, redrawing the screen 60 times every second. Because the same image is being drawn every time, the screen does not move.

`ClearColorRGB` is a drawing command: it sets the color used for screen clears. Here the color is `0x103000`, a dark green. Gameduino 2 uses standard hex triplets for colors, like HTML.

`Clear` is another drawing command. It clears the screen to the dark green color. Drawing must always begin with a clear screen.

`cmd_text` is a higher-level GPU command. It draws a text string in a particular font on the screen. In this case the text is being drawn in the center of the screen, at (x, y) coordinates (240, 136). `OPT_CENTER` draws the text so that its center is at (240, 136). The font number 31 is the largest available built-in font.

Finally, `GD.swap()` tells the graphics system that the drawing is finished and it can be made visible on the screen. There are always two copies of the screen: one that is visible, and one that is work-in-progress. Calling `GD.swap()` exchanges the screens so that the work-in-progress screen becomes visible. This swapping system (sometimes called *double buffering*) means that the display updates smoothly, only updating when the program calls `GD.swap()`.

Gameduino 2's screen is 480 pixels wide and 272 high. $(0, 0)$ is the top-left corner of the screen, $(479, 271)$ is the bottom right. The center pixel on the screen is at $(240, 136)$.

$x = y = 0$　　　　　　　　　　　　　　　　　　　　　$x = 480$

Hello world

$y = 272$

$x = 240, y = 136)$
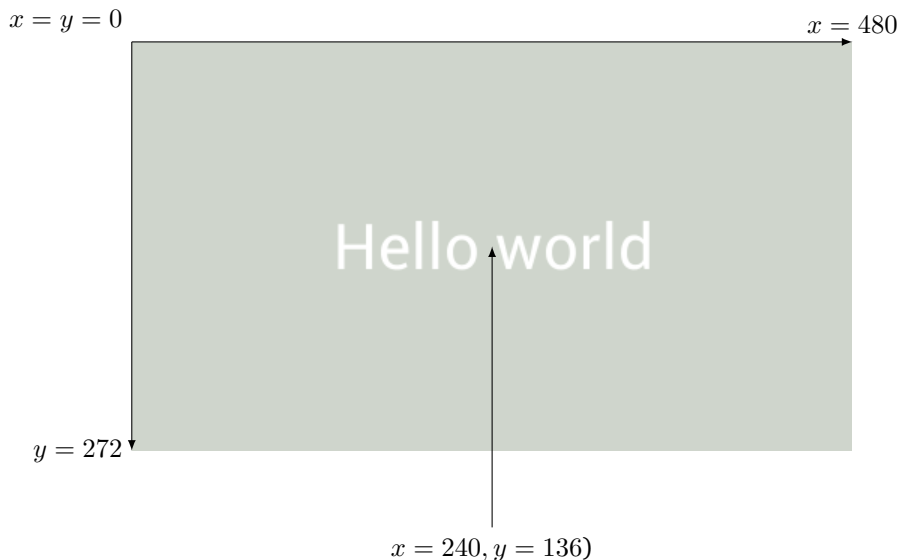
2.2   Circles are large points

What about graphics? Graphics drawing uses two commands: `Begin` and `Vertex2ii`. `Begin` tells the graphics system what to draw, and `Vertex2ii` draws it. So this code draws two single-pixel points, at coordinates (220,100) and (260,170):

```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.Begin(POINTS);
GD.Vertex2ii(220, 100);
GD.Vertex2ii(260, 170);
GD.swap();
```


The `Begin(POINTS)` tells the GPU to start drawing points. Then each call to `Vertex2ii` draws another point at the specified (x, y) screen coordinates. Note that if you don't supply a `Begin`, then the hardware doesn't know what to draw so it ignores the following `Vertex2ii` calls.

The GD library also has a `PointSize` call that lets you set the size of points. Its argument is the radius of the points, in units of $1/16$th of a pixel. The hardware uses these *subpixel* units to give you fine control over the dimensions of drawn objects, much finer than the width of a pixel. For situations where you want to specify a distance in pixels, it is usually clearest to multiply the distance by 16 in the code.

Adding a call to `PointSize` before the drawn points gives huge points with a radius of 30 pixels (and hence a *diameter* of 60 pixels).


```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.PointSize(16 * 30);  // means 30 pixels
GD.Begin(POINTS);
GD.Vertex2ii(220, 100);
GD.Vertex2ii(260, 170);
GD.swap();
```

In fact, the hardware always draws `POINTS` using mathematical circles, but when the point size is small the circles happen to look like single pixels.

2.3   Color and transparency

Adding calls to `ColorRGB` before each vertex changes their colors to orange and teal:


```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.PointSize(16 * 30);
GD.Begin(POINTS);
GD.ColorRGB(0xff8000);   // orange
GD.Vertex2ii(220, 100);
GD.ColorRGB(0x0080ff);   // teal
GD.Vertex2ii(260, 170);
GD.swap();
```

   `ClearColorRGB`, `ColorRGB` and `PointSize` are examples of *graphics state*, like variables in the graphics hardware. When changed by program, they will affect later drawing operations. To keep life simple, all graphics state is set to a default value at the start of the frame. As you might have already guessed, the default `ColorRGB` is white, which is why "Hello world" is white. The default `PointSize` is 8, meaning a radius of half a pixel.
   You can also set the transparency of drawing, just like color. Transparency is controlled by an alpha channel, which has a value from 0 to 255. 0 means

completely transparent. 255 – the default – means completely opaque. Setting the transparency to 128 with `ColorA` before the point drawing makes the points 50% transparent.


```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.PointSize(16 * 30);
GD.Begin(POINTS);
GD.ColorA(128);         // 50% transparent
GD.ColorRGB(0xff8000);  // orange
GD.Vertex2ii(220, 100);
GD.ColorRGB(0x0080ff);  // teal
GD.Vertex2ii(260, 170);
GD.swap();
```

The color commands `ColorRGB` and `ClearColorRGB` also accept (R,G,B) triplet values. So ColorRGB(0x0080ff) can be written ColorRGB(0, 128, 255) instead:

```
GD.ColorRGB(0, 128, 255); // teal
GD.Vertex2ii(260, 170);
```

2.4   Demo: fizz


```
void loop()
{
  GD.Clear();
  GD.Begin(POINTS);
  for (int i = 0; i < 100; i++) {
    GD.PointSize(GD.random(16 * 50));
    GD.ColorRGB(GD.random(256),
                GD.random(256),
                GD.random(256));
    GD.ColorA(GD.random(256));
    GD.Vertex2ii(GD.random(480), GD.random(272));
  }
  GD.swap();
}
```

Every frame, "fizz" draws 100 points, with random size, color and transparency. Because each frame is 1/60th of a second, the effect is a seething frenzy of luridly colored disks.
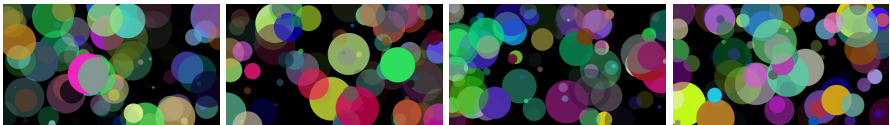
2.5 Playing notes

Gameduino 2 has several ways of making sound, but the easiest to use is its built-in sample library. Calling `GD.play()` with an instrument and a MIDI note number starts a note playing:

```
GD.play(PIANO, 60);
delay(1000);
GD.play(ORGAN, 64);
```

The code continues immediately after `GD.play()`, so here the `delay()` waits for the first note to finish before playing the second note. The list of available sampled instruments is: `HARP`, `XYLOPHONE`, `TUBA`, `GLOCKENSPIEL`, `ORGAN`, `TRUMPET`, `PIANO`, `CHIMES`, `MUSICBOX` and `BELL`. While the samples are not particularly hi-fi [1], they are quite handy for quick tests. Each instrument can play notes in the MIDI range 21-108.

The continuous sounds `SQUAREWAVE`, `SINEWAVE`, `SAWTOOTH` and `TRIANGLE` are also available. These are good for matching the sound effects of retro games. Unlike the synthesized instrument sounds, these sounds continue indefinitely. In addition to the musical sounds, there is a small selection of percussive samples. These samples do not use the MIDI note parameter, so it can be omitted. For example `GD.play(NOTCH)`. The full list is

```
CLICK
SWITCH
COWBELL
NOTCH
HIHAT
KICKDRUM
POP
CLACK
CHACK
```

The sound hardware can only play one sound at a time. So starting a sound playing instantly interrupts any previously playing sound.

To stop sound, call `GD.play(SILENCE)`.

---

[1]`PIANO` is probably the worst, but `TUBA` comes close second for sounding unlike any musical instrument, ever.

2.6   Touch tags

Every pixel on the screen has a color. It also has an invisible tag value, useful for detecting touches. Setting the tag value for the colored circles to 100 and 101:

```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.PointSize(16 * 30);
GD.Begin(POINTS);
GD.ColorRGB(0xff8000);
GD.Tag(100);
GD.Vertex2ii(220, 100);
GD.ColorRGB(0x0080ff);
GD.Tag(101);
GD.Vertex2ii(260, 170);
GD.swap();
```

gives the same image as before:

Now, when the system detects a touch on either circle, it reports back its touch code, in this case either 100 or 101. Here is a loop that senses touches (get_inputs updates all the sensor inputs in GD.inputs) and prints out the touch value on the serial connection:

```
for (;;) {
  GD.get_inputs();
  Serial.println(GD.inputs.tag);
}
```

As you press each disk, the code 100 or 101 appears on the serial output. Touch tags are a useful shortcut. Without them, the code would have to sense the touch (x, y) coordinates, and then for each disk check whether the (x, y) was inside its perimeter. Using touch tags, the code uses `Tag` to assign a code to each drawn object. Then the hardware performs the pixel testing and reports the code of the touched object. To do this, the hardware keeps track of the `Tag` for each pixel it draws. This is called the "tag buffer" - here is its contents for this example:
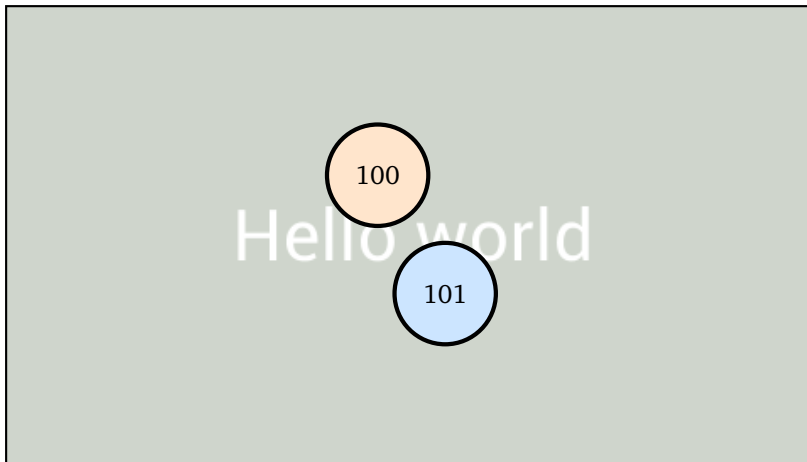
Whenever the screen is touched the hardware looks up the pixel in the tag buffer and reports it in `GD.inputs.tag`.

2.7   Game: Simon
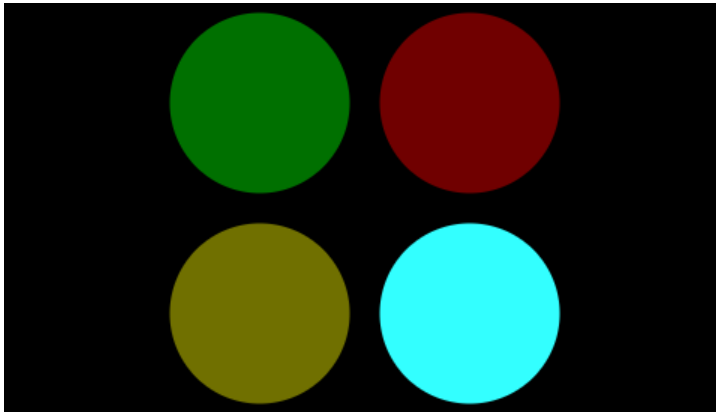
As you probably remember, "Simon" is a memory game from 1978. It plays out a random sequence on its four lights, and if you repeat it correctly, it extends the sequence by one more note. This repeats until you make a mistake, and the "game" is over.

The code first defines some primary colors:

```
#define DARK_GREEN      0x007000
#define LIGHT_GREEN     0x33ff33
#define DARK_RED        0x700000
#define LIGHT_RED       0xff3333
#define DARK_YELLOW     0x707000
#define LIGHT_YELLOW    0xffff33
#define DARK_BLUE       0x007070
#define LIGHT_BLUE      0x33ffff
```

and then uses them to draw the game screen. The game screen is four large points, each working as a button. One is highlighted depending on the pressed argument. The calls to Tag before each button mean that a touch anywhere on that button will report the value in GD.inputs.tag.

```
void drawscreen(int pressed)
{
  GD.get_inputs();
  GD.Clear();

  GD.PointSize(16 * 60);  // 60-pixel radius points
  GD.Begin(POINTS);
  GD.Tag(1);
  if (pressed == 1)
    GD.ColorRGB(LIGHT_GREEN);
  else
    GD.ColorRGB(DARK_GREEN);
  GD.Vertex2ii(240 - 70, 136 - 70);

  GD.Tag(2);
  if (pressed == 2)
    GD.ColorRGB(LIGHT_RED);
  else
    GD.ColorRGB(DARK_RED);
  GD.Vertex2ii(240 + 70, 136 - 70);

  GD.Tag(3);
  if (pressed == 3)
    GD.ColorRGB(LIGHT_YELLOW);
  else
    GD.ColorRGB(DARK_YELLOW);
  GD.Vertex2ii(240 - 70, 136 + 70);

  GD.Tag(4);
  if (pressed == 4)
    GD.ColorRGB(LIGHT_BLUE);
  else
    GD.ColorRGB(DARK_BLUE);
  GD.Vertex2ii(240 + 70, 136 + 70);

  GD.swap();
}
```

The function `play()` plays a different note according to the pressed button, while lighting up its circle for half a second, or 30 frames.

```
void play(int pressed)
{
  //                    G    R    Y    B
  //                    E3   A4   C#4  E4
  byte note[] = { 0, 52, 69, 61, 64 };
  GD.play(BELL, note[pressed]);
  for (int i = 0; i < 30; i++)
    drawscreen(pressed);
  for (int i = 0; i < 15; i++)
    drawscreen(0);
}
```

get_note() is the player input routine. It draws the game screen, and each time checks GD.inputs.tag to see whether one of the buttons is pressed. As soon as a button is pressed, it calls play() to play its note, and returns the button value.

```
static int get_note()
{
  byte pressed = 0;
  while (pressed == 0) {
    GD.random();
    drawscreen(0);
    if ((1 <= GD.inputs.tag) && (GD.inputs.tag <= 4))
      pressed = GD.inputs.tag;
  }
  play(pressed);
  return pressed;
}
```

The loop() function handles a complete game cycle. It adds a random button to the current sequence, plays it, asks the player to repeat it, and if the

player repeats the sequence perfectly, repeats. If the player makes a mistake, it plays a taunting "lose" sound effect and returns, so the next `loop()` call starts a new game.

```c
void loop()
{
  int sequence[100];
  int length = 0;

  while (1) {
    delay(500);

    sequence[length++] = random_note();

    for (int i = 0; i < length; i++)
      play(sequence[i]);

    for (int i = 0; i < length; i++) {
      int pressed = get_note();
      if (pressed != sequence[i]) {
        for (int i = 69; i > 49; i--) {
          GD.play(BELL, i);
          delay(50);
        }
        return;
      }
    }
  }
}
```

Chapter 3

Bitmaps

Gameduino 2's GPU has hardware support for drawing images, which it calls "bitmaps." Gameduino 2 bitmaps can be:

- any size, from 1×1 pixel to 512×512

- layered on top of other bitmaps, dozens deep

- recolored and drawn with transparency

- repeated indefinitely in both x and y (known as *tiling*)

- rotated, zoomed and shrunk

Bitmaps' graphic data – the bytes that represent the pixels of the bitmap – are in the GPU's main memory. This main memory is 256 KBytes in size.

In some ways, bitmaps are the descendants of the *sprites* of old video hardware. In some of the code you will see the term *sprite* – it usually just means a bitmap that is being moved around.

3.1   Loading a JPEG
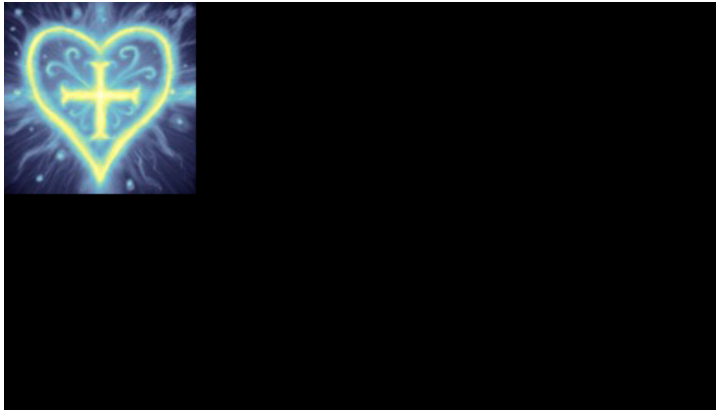

```
void setup()
{
  GD.begin();
  GD.cmd_loadimage(0, 0);
  GD.load("healsky3.jpg");
}

void loop()
{
  GD.Clear();
  GD.Begin(BITMAPS);
  GD.Vertex2ii(0, 0);
  GD.swap();
}
```

cmd_loadimage tells the GPU to expect a JPEG to follow, and load it into memory at address 0. load reads the file healsky3.jpg[1] from the microSD card and feeds it directly to the GPU, which loads it into graphics memory as a bitmap. The Begin call uses BITMAPS, so each call to Vertex2ii draws the loaded bitmap with its top-left corner at the given (x, y).

---

[1]Artwork by J. W. Bjerk (eleazzaar) – www.jwbjerk.com/art

3.2 Bitmap size

Each call to `Vertex2ii` draws a bitmap, so this code

```
GD.Clear();
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0);
GD.Vertex2ii(100, 100);
GD.Vertex2ii(200, 0);
GD.Vertex2ii(300, 100);
GD.swap();
```

produces
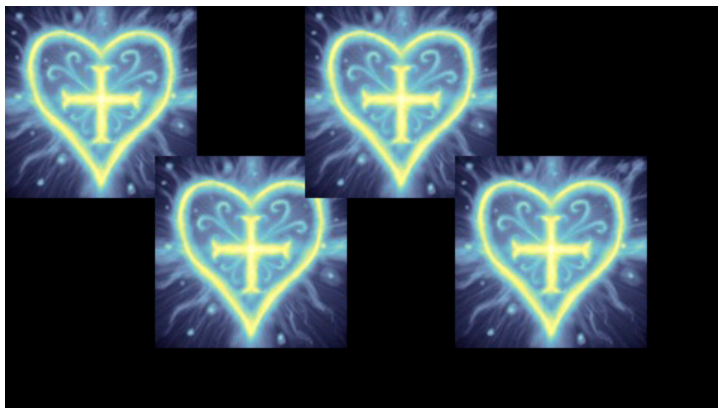
The original JPEG file was sized 128×128, so each call to `Vertex2ii` paints the bitmap into a 128×128 pixel area. But you can adjust the area painted by the bitmap using `BitmapSize`. For example

```
GD.Clear();
GD.Begin(BITMAPS);
GD.BitmapSize(NEAREST, BORDER, BORDER, 64, 64);
GD.Vertex2ii(0, 0);
GD.swap();
```

changes the bitmap size so that only a 64×64 area is drawn:
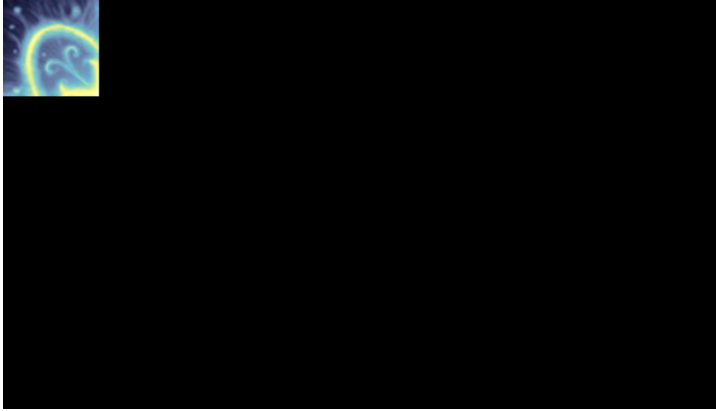
Increasing the bitmap size to 480×272 like this

```
GD.Clear();
GD.Begin(BITMAPS);
GD.BitmapSize(NEAREST, BORDER, BORDER, 480, 272);
GD.Vertex2ii(0, 0);
GD.swap();
```

does not make the bitmap bigger than the original 128×128
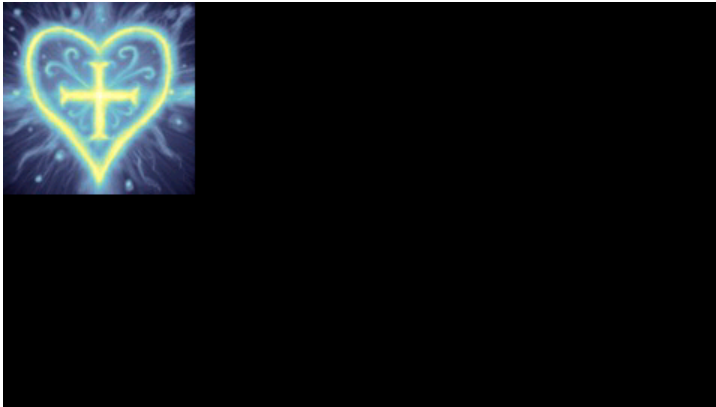
Why not? Well, the GPU actually is painting 480×272 pixels. But because the source image is only 128×128, it fills in the area outside the original image with transparent black pixels. Changing the wrapping arguments from BORDER to REPEAT tells the GPU to repeat the source image infinitely in both directions.

```
GD.Clear();
GD.Begin(BITMAPS);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 480, 272);
GD.Vertex2ii(0, 0);
GD.swap();
```

3.3   Bitmap handles
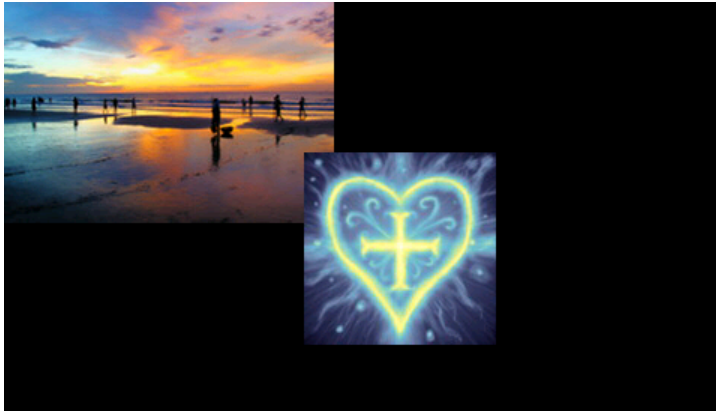

```
void setup()
{
  GD.begin();

  GD.BitmapHandle(0);
  GD.cmd_loadimage(0, 0);
  GD.load("sunrise.jpg");

  GD.BitmapHandle(1);
  GD.cmd_loadimage(-1, 0);
  GD.load("healsky3.jpg");
}

void loop()
{
  GD.Clear();
  GD.Begin(BITMAPS);
  GD.Vertex2ii(0,   0,   0);      // handle 0: sunrise
  GD.Vertex2ii(200, 100, 1);      // handle 1: healsky3
  GD.swap();
}
```

This code loads two JPEGs into graphics memory – a sunrise image at address 0, and the `healsky3` image. The second `load_image()` gives -1 as the address argument, which is a special value that tells the JPEG loader to place the JPEG immediately after the last one in graphics memory. This is handy for loading a series of JPEGs without having to compute their load addresses.

By setting the bitmap handle before each load, the graphics hardware can keep track of the two images' properties – size, format, memory address – and drawing code can select an image using one of these two handles.

`Vertex2ii` has a third parameter, used when drawing `BITMAPS`. The parameter specifies the bitmap handle to use for drawing. If it is omitted then handle 0 is used. Drawing 100 bitmaps with a random handle 0 or 1 gives a random mosaic, animating furiously like Demo: fizz.

```
GD.Clear();
GD.Begin(BITMAPS);
for (int i = 0; i < 100; i++)
  GD.Vertex2ii(GD.random(480), GD.random(272), GD.random(2));
GD.swap();
```
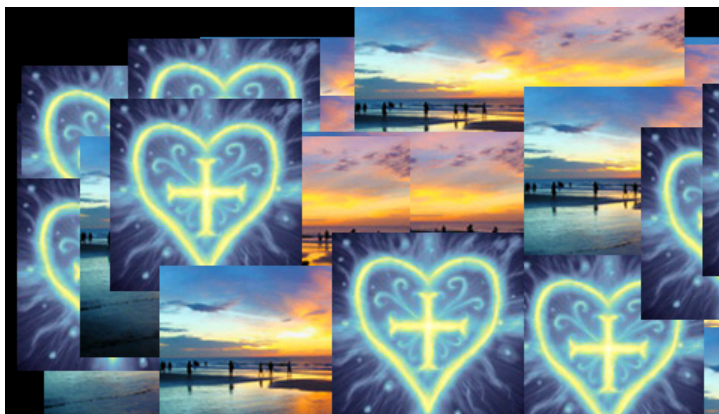

The hardware has 16 bitmap handles available for user graphics, numbered 0-15. You can re-assign bitmap handles during drawing, so a game can use more than 16 source graphics. In practice, most games use fewer than 16 graphical elements, so they can assign the handles once in `setup()`.

3.4   Bitmap pixel formats

In an ideal world, all bitmaps would be stored with unlimited color accuracy. However, because memory – both the GPU's and the Arduino's – is finite, the graphics hardware gives you various ways of balancing color precision against memory use.

L8   Eight bits per pixel, highest quality monochrome format.

L4   Four bits per pixel. Suitable for monochrome icons or fonts.

L1   One bit per pixel. Used for a minimal retro look. Also sometimes a useful format for layering and stencil effects.

RGB565

16 bits per pixel: five bits for red and blue, six bits for green. Most suitable for photos and other artwork without any transparency channel.

ARGB1555

16 bits per pixel: five bits for red, green and blue, and a single bit for alpha. The single-bit alpha channel allows simple on/off transparency.

ARGB4

16 bits per pixel: four bits each for red, green, blue and alpha. A good choice for artwork with smooth transparent edges, e.g. color icons and sprites.

RGB332

Two bits for red and blue, three for green. Sometimes used for images and icons.

ARGB2

Two bits each for red, green, blue and alpha. Not usually suitable for images, but works well for retro gaming sprites and low-color icons.

3.5 Bitmap color

Like other drawn elements, BITMAPS are drawn using the current color. The original image pixels are multiplied by the color, very much like viewing the bitmap through colored glass.

```
GD.Clear();
GD.Begin(BITMAPS);

GD.ColorRGB(0x00ff00);                  // pure green
GD.Vertex2ii(240 - 130, 136 - 130, 1);

GD.ColorRGB(0xff8080);                  // pinkish
GD.Vertex2ii(240     , 136 - 130, 1);

GD.ColorRGB(0xffff80);                  // yellowish
GD.Vertex2ii(240 - 130, 136     , 1);

GD.ColorRGB(0xffffff);                  // white
GD.Vertex2ii(240     , 136     , 1);
GD.swap();
```

If the current RGB color is white (0xffffff) then the original bitmap colors are used. If black (0x000000), then all pixels are drawn black.

3.6 Converting graphics

Converting graphics for the Gameduino 2 – taking images and formatting their pixel data for use in bitmaps – can be complex. To help, the Gameduino 2 tools include an asset converter that reads image files and produces data that can be used in your sketch.

The output of the asset converter is a single header file, named `x_assets.h`, where x is the name of your sketch. This file defines a macro `LOAD_ASSETS()` that loads the image data into the Gameduino 2's graphics RAM. To use it, include the header file and then call `LOAD_ASSETS()` immediately after calling `GD.begin()`:

```
#include "walk_assets.h"

void setup()
{
  GD.begin();
  LOAD_ASSETS();
```

After the `LOAD_ASSETS()` finishes, all the bitmaps are loaded into graphics memory, and all the bitmap handles are set up to use the bitmaps. These handles are also defined in the header file, so the code can use them as:

```
GD.Vertex2ii(x, y, WALK_HANDLE);
```

There are two advantages to using the asset converter, rather than loading images from JPEG files. The first is that JPEGs are always loaded to an RGB565 bitmap. There is no transparency in JPEGs, so this is the only format supported. Second, the asset converter compresses[2] the graphics *losslessly*, whereas JPEG uses lossy compression. For photographic images, the difference is small. For carefully drawn game art, icons and fonts, the difference between lossless and lossy compression can be quite noticeable.

---

[2] The compression scheme is zlib INFLATE (`http://www.zlib.net/`), as used in gzip and Zip. The GPU supports zlib INFLATE in hardware.

3.7   Bitmap cells
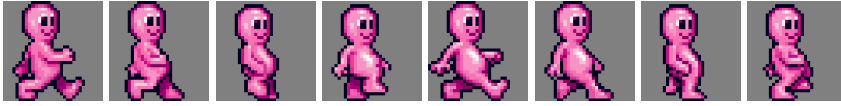
These eight 32×32 `ARGB1555` bitmap images are an animated walk cycle. Because all the images are the same size and format, they can share the same bitmap handle. The drawing commands select which image (or *cell*) to draw. For bitmaps drawn with `Vertex2ii` the fourth parameter selects the cell. So for example this code draws the eight animation frames across the screen:

```
GD.Begin(BITMAPS);
GD.Vertex2ii(  0, 10, WALK_HANDLE, 0);
GD.Vertex2ii( 50, 10, WALK_HANDLE, 1);
GD.Vertex2ii(100, 10, WALK_HANDLE, 2);
GD.Vertex2ii(150, 10, WALK_HANDLE, 3);
GD.Vertex2ii(200, 10, WALK_HANDLE, 4);
GD.Vertex2ii(250, 10, WALK_HANDLE, 5);
GD.Vertex2ii(300, 10, WALK_HANDLE, 6);
GD.Vertex2ii(350, 10, WALK_HANDLE, 7);
```


Each bitmap handle can contain up to 128 cells, and the cells are arranged consecutively in graphics memory. Cells are useful for animation. By loading the animation sequence into the cells of a single bitmap handle, you can animate an object by changing its cell number.

Using the same eight-frame animated sequence, the *walk* demo animates 256 sprites in a walk cycle crossing the screen. Each sprite has a counter that controls its x position and its animation frame. The color of the sprites changes from black (`0x000000`) at the top of the screen to white (`0xffffff`) at the bottom.
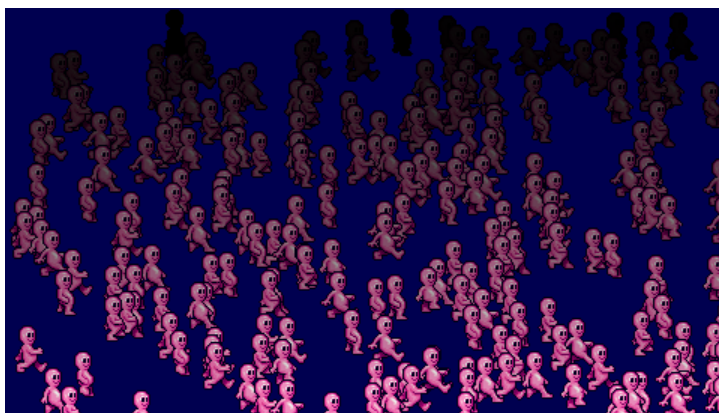
```
static int a[256];

#include "walk_assets.h"

void setup()
{
  GD.begin();
  LOAD_ASSETS();
  for (int i = 0; i < 256; i++)
    a[i] = GD.random(512);
}

void loop()
{
  GD.ClearColorRGB(0x000050);
  GD.Clear();
  GD.Begin(BITMAPS);
  for (int i = 0; i < 256; i++) {
    GD.ColorRGB(i, i, i);
    GD.Vertex2ii(a[i], i, WALK_HANDLE, (a[i] >> 2) & 7);
    a[i] = (a[i] + 1) & 511;
  }
  GD.swap();
}
```

3.8   Rotation, zoom and shrink
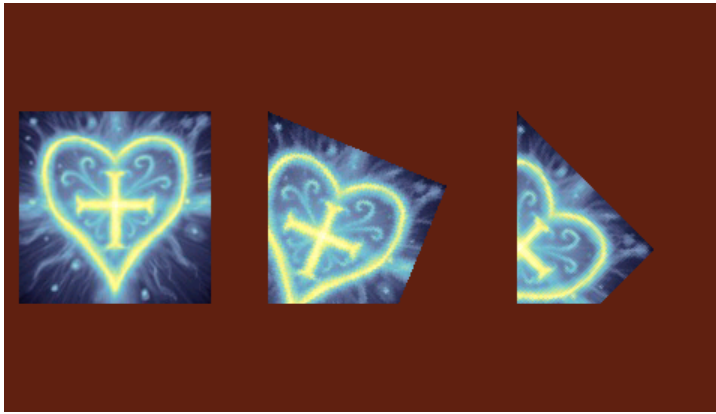

```
GD.ClearColorRGB(0x602010);
GD.Clear();
GD.Begin(BITMAPS);

GD.Vertex2ii(10, 72);

GD.cmd_rotate(DEGREES(22.5));
GD.cmd_setmatrix();
GD.Vertex2ii(176, 72);

GD.cmd_rotate(DEGREES(22.5));
GD.cmd_setmatrix();
GD.Vertex2ii(342, 72);

GD.swap();
```

Bitmap rotation and zooming is controlled by the *bitmap transform matrix*. This matrix is part of the graphics state. It controls the mapping of the bitmap's image pixels onto the screen. Fortunately, much of the math of the transform matrix is handled by the hardware, so we can use higher-level operations like rotate and scale.

Here the bitmap is being drawn three times, with 0°, 22.5°, and finally 45° clockwise rotation, because `cmd_rotate` operations are cumulative. Parts of the bitmap disappear as it rotates around its top-left corner. This version shows what is happening more clearly because transparency has been turned off using a BlendFunc of (`SRC_ALPHA`, `ZERO`):
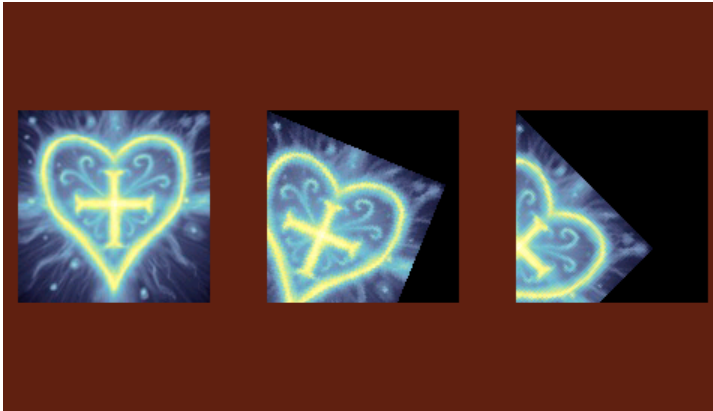
Each `Vertex2ii` draws 128×128 pixels, because that is the size of the bitmap. However, the source image pixels are determined by the current bitmap transform matrix, and as it changes the source image rotates.

This rotation is centered on the top-left pixel of the bitmap, pixel $(0, 0)$. A more useful effect is rotation about the image center, which in this case is at bitmap pixel $(64, 64)$. To do this, the steps are:

1. *translate* the image so that $(64, 64)$ is moved to $(0, 0)$

2. rotate the image around $(0, 0)$ using `cmd_rotate`

3. translate the image back, returning pixel $(0, 0)$ to $(64, 64)$

This function `rotate_64_64` performs these three steps:

```
// Apply a rotation around pixel (64, 64)
static void rotate_64_64(uint16_t a)
{
    GD.cmd_translate(F16(64),F16(64));
    GD.cmd_rotate(a);
    GD.cmd_translate(F16(-64), F16(-64));
}
```

It uses the `cmd_translate` command to move the bitmap by 64 pixels in both x and y. `cmd_translate` uses 16.16 fixed-point values for its arguments, for subpixel precision, so here the `F16()` macro does the conversion from integer pixel values.
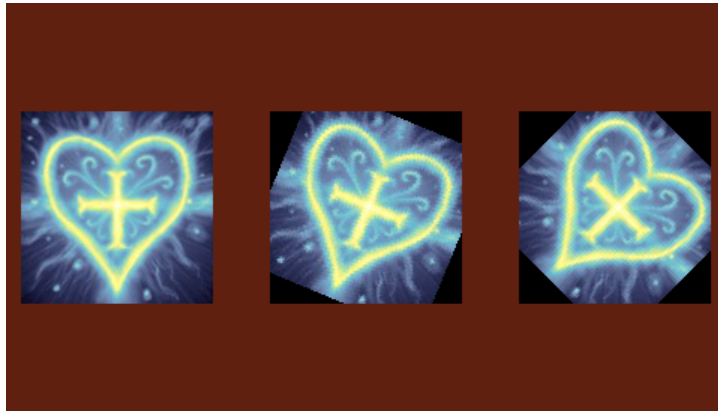

```
GD.ClearColorRGB(0x602010);
GD.Clear();
GD.BlendFunc(SRC_ALPHA, ZERO);
GD.Begin(BITMAPS);

GD.Vertex2ii(10, 72);

rotate_64_64(DEGREES(22.5));
GD.cmd_setmatrix();
GD.Vertex2ii(176, 72);

rotate_64_64(DEGREES(22.5));
GD.cmd_setmatrix();
GD.Vertex2ii(342, 72);

GD.swap();
```

Scaling the image – either increasing the scale to zoom it, or decreasing the scale to shrink it – uses the `cmd_scale` command. Here a similar function `scale_64_64()` applies a scale around pixel $(64, 64)$. The first bitmap has no scale. The second is scaled by a factor of 2.0, doubling it in size. The third bitmap is scaled by 0.4, shrinking it.
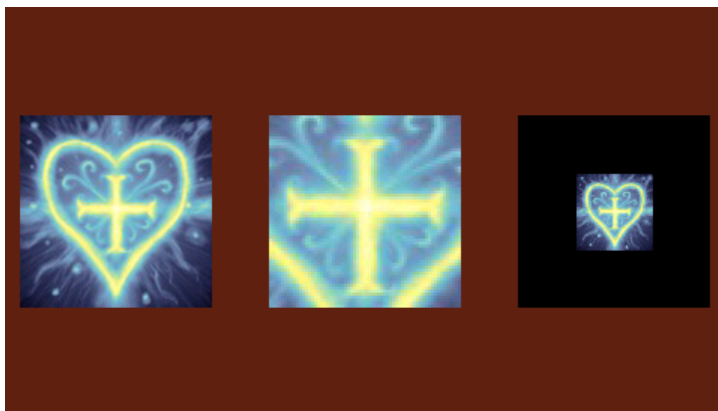

```
GD.ClearColorRGB(0x602010);
GD.Clear();
GD.Begin(BITMAPS);

GD.Vertex2ii(10, 72);

scale_64_64(F16(2.0), F16(2.0));
GD.cmd_setmatrix();
GD.Vertex2ii(176, 72);

GD.cmd_loadidentity();
scale_64_64(F16(0.4), F16(0.4));
GD.cmd_setmatrix();
GD.Vertex2ii(342, 72);

GD.swap();
```

Chapter 4

More on graphics

4.1 Lines
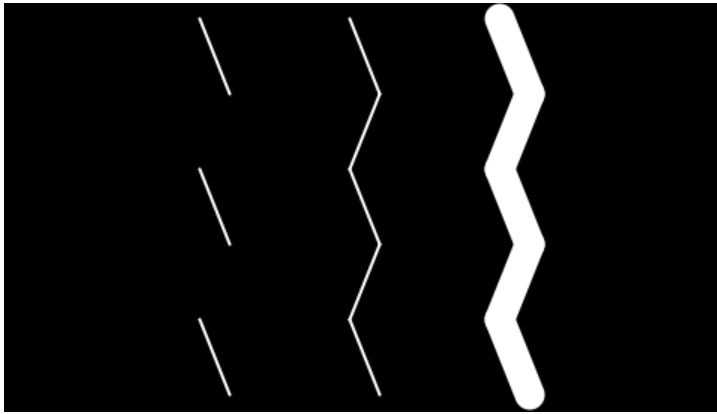
To draw lines, use Begin(LINES) or LINE_STRIP. LINES connects every pair of vertices, whereas LINE_STRIP joins all the vertices together.

```
static void zigzag(int x)
{
  GD.Vertex2ii(x - 10,    10); GD.Vertex2ii(x + 10,    60);
  GD.Vertex2ii(x - 10,  110); GD.Vertex2ii(x + 10,  160);
  GD.Vertex2ii(x - 10,  210); GD.Vertex2ii(x + 10,  260);
}

void loop()
{
  GD.Clear();
  GD.Begin(LINES);
  zigzag(140);
  GD.Begin(LINE_STRIP);
  zigzag(240);
  GD.LineWidth(16 * 10);
  GD.Begin(LINE_STRIP);
  zigzag(340);
  GD.swap();
}
```

The NightStrike game uses LINES to draw spark showers that "pop" out of each explosion. Each spark is tracked as an object with a position (x,y), a velocity (xv,yv), and an age. At the start of the explosion each spark is positioned at the explosion center, but has a random velocity in (xv,yv). Every frame the sparks travel along their path, and are drawn by this code:

```
GD.LineWidth(GD.rsin(size, age[i] << 11));
GD.Vertex2f(x[i], y[i]);
GD.Vertex2f(x[i] + xv[i], y[i] + yv[i]);
```

This draws a line from the spark's current (x, y) position to its *next* position. The size of the spark (LineWidth) uses a sinusoidal formula, so that it starts off thin, gets wide, then becomes thin again at the end of its cycle.
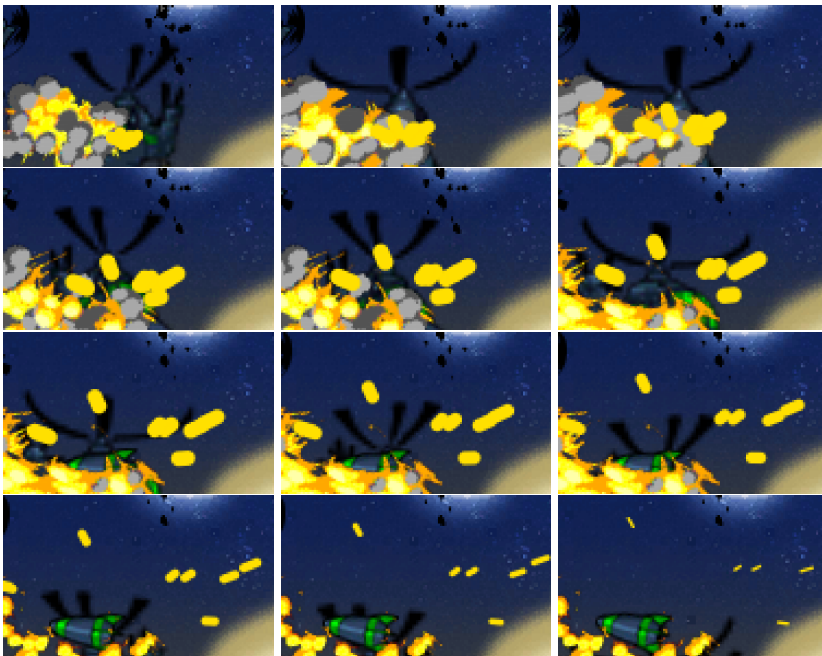
This 12-frame detail shows the sparks' progress. The game runs at 60 fps, so the spark sequence takes about 1/5th of a second.

4.2   Rectangles

To draw rectangles, use `Begin(RECTS)` and supply opposite corners of the rectangle. The order of the two corners does not matter. The rectangles are drawn with round corners, using the current line width as the corner radius. The round corners are drawn *outside* the rectangle, so increasing the corner radius draws more pixels. This example draws a 420×20 rectangle three times with increasing corner radius.

```
GD.Clear();
GD.Begin(RECTS);

GD.Vertex2ii(30, 30);
GD.Vertex2ii(450, 50);

GD.LineWidth(10 * 16);      // corner radius 10.0 pixels
GD.Vertex2ii(30, 120);
GD.Vertex2ii(450, 140);

GD.LineWidth(20 * 16);      // corner radius 20.0 pixels
GD.Vertex2ii(30, 220);
GD.Vertex2ii(450, 230);

GD.swap();
```

4.3 Gradients

Gradients - smooth color blends across the screen - are a useful graphical element. You *could* use a large background image of the gradient, but it's more efficient to use `cmd_gradient`.

`cmd_gradient` draws a smooth gradient on the whole screen, given a starting point and color, and an ending point and color. In this example the starting point is top-left $(0, 0)$ and the starting color is deep blue (`0x0060c0`). The ending point is bottom-left $(0, 271)$, deep orange (`0xc06000`).

```
GD.cmd_gradient(0, 0,   0x0060c0,
                0, 271, 0xc06000);
GD.cmd_text(240, 136, 31, OPT_CENTER, "READY PLAYER ONE");
GD.swap();
```

These two points produce a vertical gradient, but `cmd_gradient` can draw gradients at any angle. You can get a horizontal or diagonal gradient by appropriate placement of the two control points.

`cmd_gradient` used like this writes every pixel on the screen, so it can be used in place of `Clear`. To draw a gradient on part of the screen, you can use `ScissorXY` and `ScissorSize` to limit drawing to a rectangular area. For more complex shapes, `cmd_gradient` can be used with a stencil test.

4.4   Blending
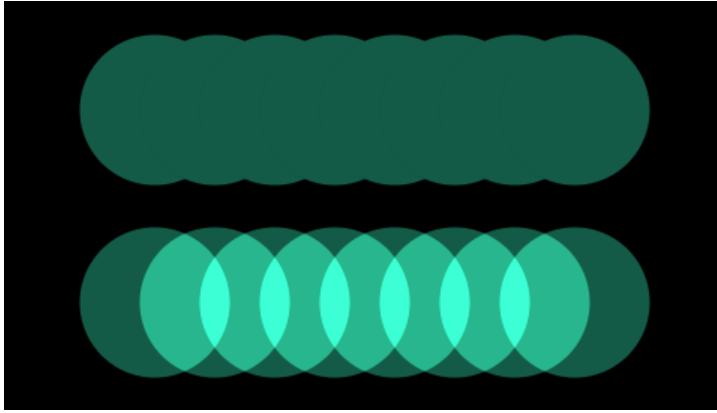

```
GD.Begin(POINTS);  // draw 50-pixel wide green circles
GD.ColorRGB(20, 91, 71);
GD.PointSize(50 * 16);

for (int x = 100; x <= 380; x += 40)
  GD.Vertex2ii(x, 72);

GD.BlendFunc(SRC_ALPHA, ONE);   // additive blending
for (int x = 100; x <= 380; x += 40)
  GD.Vertex2ii(x, 200);
```

Usually each drawn pixel *replaces* the pixel that was on the screen. This is useful for layering, but sometimes you need a different mixing operation. BlendFunc controls this operation. The first argument to BlendFunc is the *source blend factor*, which controls the incoming color pixels. The second is the *destination blend factor*, which similarly controls the pixels that are already in the color buffer. After the blend factors have modified the colors, the two results are summed to produce the final pixel color.

incoming
pixel color

*source
blend factor*

existing
pixel color

*destination
blend factor*

$\times$

$\times$

$+$

final pixel
color
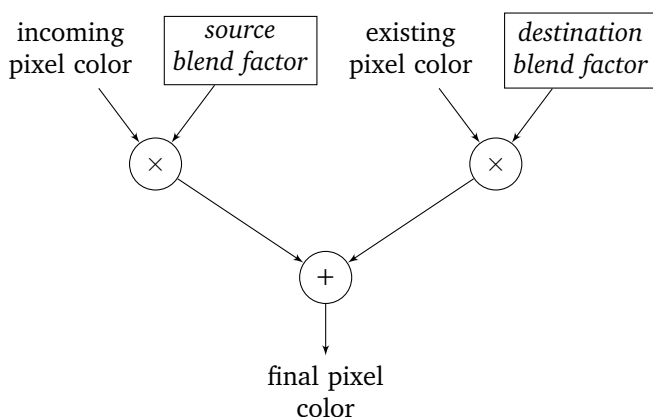
The default mode is `BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)`, which means that the incoming pixel color is mixed with the existing screen color in proportion to the alpha channel value.

Another frequently used mode is `BlendFunc(SRC_ALPHA, ONE)`. The `ONE` factor for the destination means that the incoming pixel color is *added* to the existing pixel color. This is useful for glows and overlays. In the code above you can see this mode means that every disk drawn is added to the existing screen contents, so their brightness accumulates.

`BlendFunc(SRC_ALPHA, ZERO)` is the *replace* operation; it disables transparency. The `ZERO` factor for the destination means that existing screen pixels are discarded at the blending stage.

4.5 Fonts
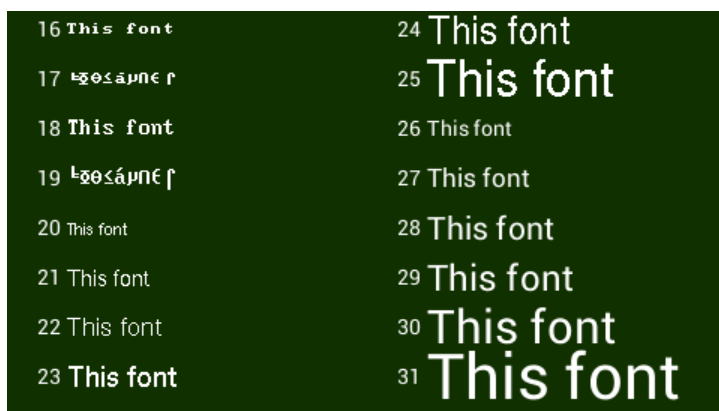
The Gameduino 2's GPU has sixteen built-in fonts, numbered 16-31. Numbers 26-31 are the high-quality anti-aliased fonts. You can also load TrueType (`.ttf`) fonts using the asset converter (see *Converting graphics* on p.39). After loading it, you can use the new font with any of the drawing commands.

```
byte font = NIGHTFONT_HANDLE;
GD.cmd_text(240, 40, font, OPT_CENTER, "abcdefghijklm");
GD.cmd_text(240, 100, font, OPT_CENTER, "nopqrstuvwxyz");
GD.cmd_text(240, 160, font, OPT_CENTER, "ABCDEFGHIJKLM");
GD.cmd_text(240, 220, font, OPT_CENTER, "NOPQRSTUVWXYZ");
```
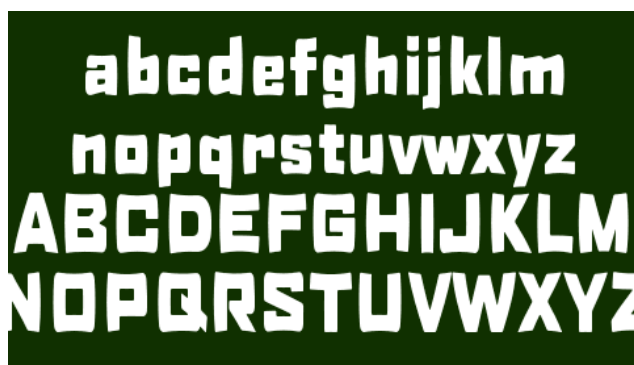
4.6 Subpixel coordinates

So far all drawing has used `Vertex2ii` to supply the vertices. `Vertex2ii` can only handle (x, y) that are integers in the range 0-511. What if you want to supply a vertex with a negative coordinate, or a coordinate that isn't on an exact pixel? In these cases, you can use `Vertex2f`, which uses a finer scale for x and y, and has a much larger range. `Vertex2f` coordinates are in 16ths of a pixel. So this call using `Vertex2ii`:

```
GD.Vertex2ii(1, 100);
```

becomes this call using `Vertex2f`:

```
GD.Vertex2f(16, 1600);
```

but it is often clearer to multiply by 16 explicitly in the code, like this:

```
GD.Vertex2f(16 * 1, 16 * 100);
```

`Vertex2f` draws `POINTS`, `LINES` and `RECTS` with much finer precision. Each of these 16 points is drawn with a y coordinate that increases by 1/16th of a pixel. As you can see, the graphics system adjusts the shading of the pixels very slightly as the point moves downwards by 1/16th of a pixel. This extra precision means that "pixel" effects are much less noticable.
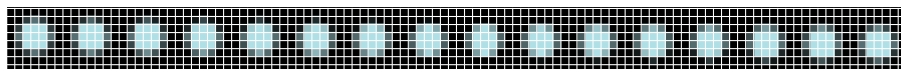
4.7   Angles in Furmans

The GD library refers to angles in several places:

- the `cmd_rotate` command, which rotates bitmaps, takes an angle parameter

- the `cmd_dial` command draws a dial widget at a particular angle

- the widget tracking system, controlled by `cmd_track`, reports the angle of rotary widgets

- the `rsin`, `rcos`, `polar` and `atan2` math functions all use angles

In each case, the angle is specified in Furmans, not in degrees or radians. A Furman is an angle measure; there are 65536 Furmans in a circle. So

$$1 \text{ Furman } = \frac{1}{65536} \text{ circle } = \frac{360}{65536} \text{ degrees } = \frac{2\pi}{65536} \text{radians}$$

Orientations are represented by clockwise angles, with an angle of 0 meaning straight down:
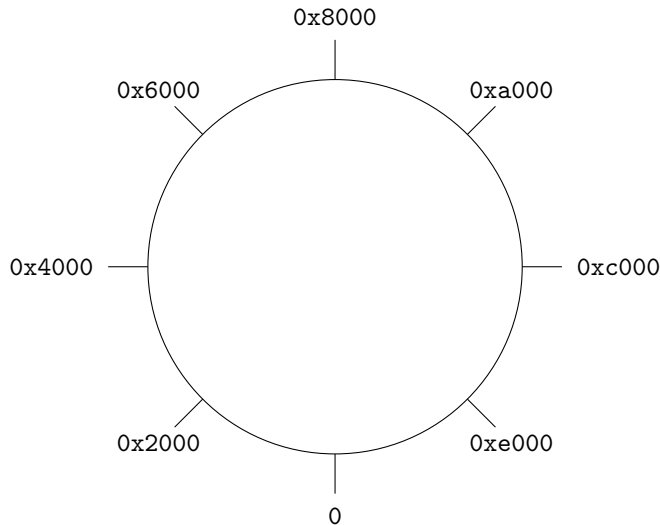
As a convenience, the GD library defines a macro DEGREES(), which converts angles in degrees to Furmans:

```
#define DEGREES(n) ((65536UL * (n)) / 360)
```

The advantage of representing angles as Furmans is that they make angular arithmetic significantly cheaper. For example, when incrementing angles for a rotating object, normal 16-bit arithmetic means that the Furman measurement wraps from 65535 back to zero.

In the NightStrike game the player's touch controls the angle of the gun turret. At startup, the game requests the GPU track angles for any touch on tag number 0x01:

```
GD.cmd_track(240, 271, 1, 1, 0x01);
```

After the `cmd_track` command, `GD.inputs.track_val` holds the angle in Furmans from screen position (240, 271) to the touch location. The game copies this value into the turret's angle variable.

```
if ((GD.inputs.track_tag & 0xff) == 1)
  turret.angle = GD.inputs.track_val;
```

When the turret is drawn, the angle variable is used with `cmd_rotate` to rotate the turret's bitmap, which is in the "0 Furmans" orientation:

4.8   The context stack


```
static void blocktext(int x, int y, byte font, const char *s)
{
  GD.SaveContext();
  GD.ColorRGB(0x000000);
  GD.cmd_text(x-1, y-1, font, 0, s);
  GD.cmd_text(x+1, y-1, font, 0, s);
  GD.cmd_text(x-1, y+1, font, 0, s);
  GD.cmd_text(x+1, y+1, font, 0, s);
  GD.RestoreContext();

  GD.cmd_text(x, y, font, 0, s);
}
```

The function blocktext, used in *NightStrike*'s title screen, draws the string
s with a black outline around the text. To do this, it changes the color to black
using ColorRGB, and draws the text four times slightly offset. Now it needs to
draw the text itself, in the original color. But the original color is lost, because
it was changed to black.

The SaveContext preserves all the graphics state – including the color
– in a private storage area. When RestoreContext executes, it copies this
saved state back. Hence the color is restored to the value it had at the original
SaveContext.

Using SaveContext or RestoreContext, functions can change graphics
state, and then restore it afterwards. As far as any caller is concerned, the
function leaves the state untouched. The GPU has enough internal storage to
preserve three extra copies of the graphics state in this way.

Chapter 5

Touch

5.1 Reading the touch inputs

The function `get_inputs` reads all the Gameduino 2's inputs, including the touch sensors. After making this call, you can read the touch coordinates from `GD.inputs.x` and `GD.inputs.y`. The best time in the game cycle to sense inputs is just before starting to draw the screen, so a frequent pattern is:

```
GD.get_inputs();
GD.Clear();
```

If there is no touch, then both `GD.inputs.x` and `GD.inputs.y` are set to -32768. The touch (x, y) are *screen pixel coordinates*, so they range $(0 \leq x \leq 479)$ and $(0 \leq y \leq 271)$. To convert these pixel coordinates into subpixel coordinates, multiply by 16. For example:

```
blobs[blob_i].x = GD.inputs.x << 4;
blobs[blob_i].y = GD.inputs.y << 4;
```

5.2   Demo: blobs
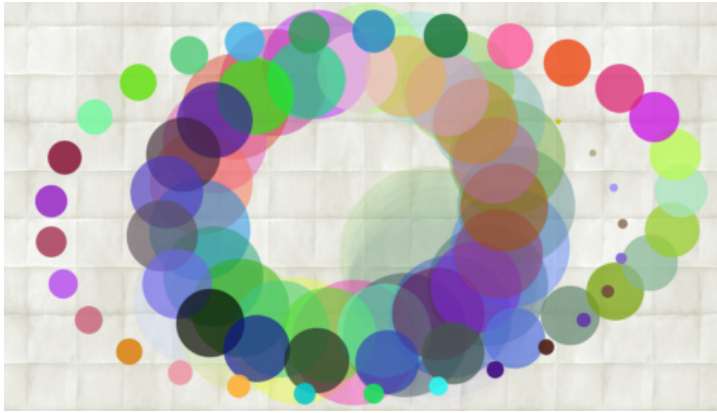
The "blobs" demo lets you sketch a trail of groovy expanding colored circles with a finger or stylus. The code keeps track of 128 points, animating each with radius and alpha to create a pleasing "fade out" effect.

At startup, begin() sets all the blobs' positions to off screen.

```
#define NBLOBS      128
#define OFFSCREEN   -16384

struct xy {
  int x, y;
} blobs[NBLOBS];

void setup()
{
  GD.begin();

  for (int i = 0; i < NBLOBS; i++) {
    blobs[i].x = OFFSCREEN;
    blobs[i].y = OFFSCREEN;
  }
}
```

If the screen is being touched, the subpixel coordinates of the touch are added into the `blobs` ring. Each blob's transparency and radius depends on its age. The "random" color is actually computed using modulo arithmetic.

```c
void loop()
{
  static byte blob_i;
  GD.get_inputs();
  if (GD.inputs.x != -32768) {
    blobs[blob_i].x = GD.inputs.x << 4;
    blobs[blob_i].y = GD.inputs.y << 4;
  } else {
    blobs[blob_i].x = OFFSCREEN;
    blobs[blob_i].y = OFFSCREEN;
  }
  blob_i = (blob_i + 1) & (NBLOBS - 1);

  GD.ClearColorRGB(0xe0e0e0);
  GD.Clear();

  GD.Begin(POINTS);
  for (int i = 0; i < NBLOBS; i++) {
    // Blobs fade away and swell as they age
    GD.ColorA(i << 1);
    GD.PointSize((1024 + 16) - (i << 3));

    // Random color for each blob, keyed from (blob_i + i)
    uint8_t j = (blob_i + i) & (NBLOBS - 1);
    byte r = j * 17;
    byte g = j * 23;
    byte b = j * 147;
    GD.ColorRGB(r, g, b);

    // Draw it!
    GD.Vertex2f(blobs[j].x, blobs[j].y);
  }
  GD.swap();
}
```

5.3   Tags

Tagging is a GPU feature that makes touch detection easier. In many situations the exact coordinates of the touch are not important – you are only interested in *what* has been touched. As you draw each object on the screen, you can tag its pixels with a byte value. For example, to create a 'OK' button, you can tag its pixels with code 44. Then each time the user touches the button, the number 44 appears in the tag register. Tag values are a single byte, ranging from 0 to 255. However, the hardware uses a tag value of 255 to indicate "no touch", so the usable range of values is 0 to 254.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	

This example sets the `Tag` value before drawing each number. Because the pixels belonging to each number are then "tagged", `GD.inputs.tag` is set to the number being currently touched, and the Arduino can access it and print it on the serial output.

```
GD.Clear();
for (int i = 0; i <= 254; i++) {
  GD.Tag(i);
  GD.cmd_number((i % 16) * 30, (i / 16) * 17, 26, 0, i);
}
GD.swap();
GD.get_inputs();
```

5.4 Sketching

Sketching here means taking touch input and plotting pixels. The Gameduino 2 GPU has built-in sketching so you can 'paint' pixels into the bitmap without any input from the Arduino. The cmd_sketch command starts sketching. When you want sketching to stop, call cmd_stop. Note the call to GD.flush(). This forces any buffered commands to the GPU immediately.

```
void setup()
{
  GD.begin();
  GD.cmd_memset(0, 0, 480UL * 272UL);   // clear the bitmap
  GD.Clear();                           // draw the bitmap
  GD.BitmapLayout(L8, 480, 272);
  GD.BitmapSize(NEAREST, BORDER, BORDER, 480, 272);
  GD.Begin(BITMAPS);
  GD.Vertex2ii(0, 0);
  GD.swap();
  GD.cmd_sketch(0, 0, 480, 272, 0, L8); // start sketching
  GD.finish();                          // flush all commands
}

void loop() { }
```

5.5   Widgets and tracking controls

Gameduino 2 has a rich set of widgets: dials, buttons, gauges, sliders and scroll bars. To help manage the user interactions with the adjustable widgets, a feature called *tracking controls* extends the simple tag system. Using tags, the program can know when an object is touched. With a tracking control, the program can also know the relative position of the touch.

The `widgets` demo draws all widget types, and for the three adjustable widgets – the dial, slider and toggle – uses tracking controls to manage their touches. Each adjustable widget is drawn with a unique tag value, here 200, 201 and 202.
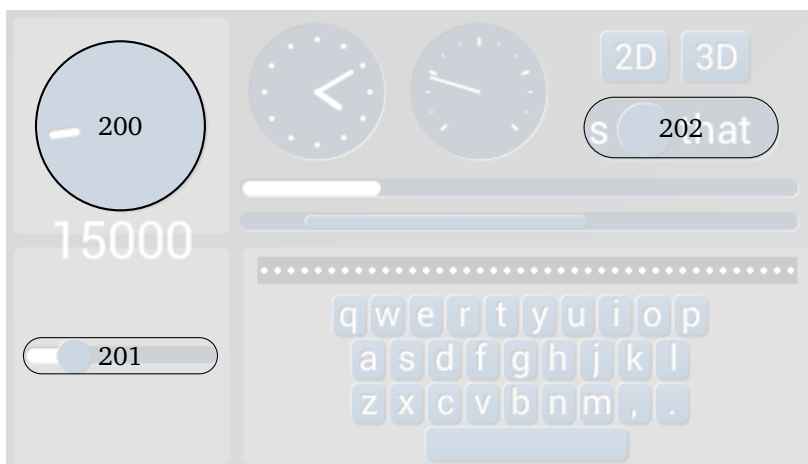
The program then tells the GPU to track touches on the three regions. First it draws the dial with tag 200. Then it calls `cmd_track` so that the GPU will track touches for that particular tag, and compute the position relative to pixel position (68, 68), the center of the dial:

```
GD.Tag(TAG_DIAL);
GD.cmd_dial(68, 68, 50, options, value);
GD.cmd_track(68, 68, 1, 1, TAG_DIAL);
```

Similarly for the linear widgets, the slider and toggle:

```
GD.Tag(TAG_SLIDER);
GD.cmd_slider(16, 199, 104, 10, options, value, 65535);
GD.cmd_track(16, 199, 104, 10, TAG_SLIDER);

GD.Tag(TAG_TOGGLE);
GD.cmd_toggle(360, 62, 80, 29, options, value,
              "that" "\xff" "this");
GD.cmd_track(360, 62, 80, 20, TAG_TOGGLE);
```

With these set up, the program can detect a relative touch on any widget by reading `GD.inputs.track_tag` and `GD.inputs.track_val`. The value in `GD.inputs.track_val` is an unsigned 16-bit number in the range 0-65535. For the rotary track – the dial – this is the touch angle in Furmans. For the linear tracks, it is the touch position within the rectangle. 0 means far left and 65535 means far right.

```
switch (GD.inputs.track_tag & 0xff) {
case TAG_DIAL:
case TAG_SLIDER:
case TAG_TOGGLE:
  value = GD.inputs.track_val;
}
```

Chapter 6

Sound

Gameduino 2 has two sound systems. The first – the *synthesizer* – can generate a set of fixed sounds and musical notes. The synthesizer is useful for quickly adding audio to a project, but because the sound set is fixed, is not very flexible. The second is the *sample playback*. It plays sampled audio from main memory in a variety of formats. This system is much more flexible but you will need to prepare and load samples into RAM.

6.1   Clicks and pops

The synthesizer provides several short "percussive" sounds, mostly for use in user interfaces. To trigger the sound, call `GD.play()` with the sound ID. The full list of available sounds is:

```
CLICK
SWITCH
COWBELL
NOTCH
HIHAT
KICKDRUM
POP
CLACK
CHACK
```

6.2   Instrument playback

The synthesizer can also produce musical instrument notes. Again, calling
`GD.play()` triggers the sound, and the first parameter is an instrument:

```
HARP
XYLOPHONE
TUBA
GLOCKENSPIEL
ORGAN
TRUMPET
PIANO
CHIMES
MUSICBOX
BELL
```

In this case `GD.play()` also accepts an optional second argument that specifies
a MIDI note number in the range 21 to 108. If this argument is omitted then
it plays in C4, MIDI note 60.

The synthesizer can also play a few other miscellaneous sounds. Instru-
ments `SQUAREWAVE`, `SINEWAVE`, `SAWTOOTH` and `TRIANGLE` generate continuous
simple waves, useful for retro effects. Instruments `BEEPING`, `ALARM`, `WARBLE`
and `CAROUSEL` generate various alarm tones. And the ASCII codes '0' - '9', '*'
and '#' generate the corresponding touch-tone DTMF tones.

To detect when a note has finished playing, read the hardware's `PLAY` reg-
ister. When it is zero, the note is finished:

```
GD.play(PIANO, 60);
while (GD.rd(REG_PLAY) != 0)
  ;
```

MIDI note	ANSI note	freq. (Hz)	MIDI note	ANSI note	freq. (Hz)
21	A0	27.5	65	F4	349.2
22	A#0	29.1	66	F#4	370.0
23	B0	30.9	67	G4	392.0
24	C1	32.7	68	G#4	415.3
25	C#1	34.6	69	A4	440.0
26	D1	36.7	70	A#4	466.2
27	D#1	38.9	71	B4	493.9
28	E1	41.2	72	C5	523.3
29	F1	43.7	73	C#5	554.4
30	F#1	46.2	74	D5	587.3
31	G1	49.0	75	D#5	622.3
32	G#1	51.9	76	E5	659.3
33	A1	55.0	77	F5	698.5
34	A#1	58.3	78	F#5	740.0
35	B1	61.7	79	G5	784.0
36	C2	65.4	80	G#5	830.6
37	C#2	69.3	81	A5	880.0
38	D2	73.4	82	A#5	932.3
39	D#2	77.8	83	B5	987.8
40	E2	82.4	84	C6	1046.5
41	F2	87.3	85	C#6	1108.7
42	F#2	92.5	86	D6	1174.7
43	G2	98.0	87	D#6	1244.5
44	G#2	103.8	88	E6	1318.5
45	A2	110.0	89	F6	1396.9
46	A#2	116.5	90	F#6	1480.0
47	B2	123.5	91	G6	1568.0
48	C3	130.8	92	G#6	1661.2
49	C#3	138.6	93	A6	1760.0
50	D3	146.8	94	A#6	1864.7
51	D#3	155.6	95	B6	1975.5
52	E3	164.8	96	C7	2093.0
53	F3	174.6	97	C#7	2217.5
54	F#3	185.0	98	D7	2349.3
55	G3	196.0	99	D#7	2489.0
56	G#3	207.7	100	E7	2637.0
57	A3	220.0	101	F7	2793.8
58	A#3	233.1	102	F#7	2960.0
59	B3	246.9	103	G7	3136.0
60	C4	261.6	104	G#7	3322.4
61	C#4	277.2	105	A7	3520.0
62	D4	293.7	106	A#7	3729.3
63	D#4	311.1	107	B7	3951.1
64	E4	329.6	108	C8	4186.0

6.3   Samples

Gameduino 2's audio system can also play back samples. The sample playback system is independent – you can play a sample while playing a fixed sound, and the hardware will mix the two sounds together, according to their assigned volumes.

To play back a sample from memory, call GD.sample with the sample's base address, length, frequency and format as arguments. So if you have a ULAW sample at address 0, of length 22050 bytes, with frequency 44100 Hz, you play it by calling:

```
GD.sample(0, 22050, 44100, ULAW_SAMPLES);
```

Three sample formats are supported:

format	bit per sample	encoding
LINEAR_SAMPLES	8	signed 8-bit linear, -128 to 127
ULAW_SAMPLES	8	standard 8-bit encoded μ-law
ADPCM_SAMPLES	4	standard IMA ADPCM

Gameduino 2's asset converter tool (see *Converting graphics* on p.39) converts mono .wav files to any of these formats. The hardware supports playback rates up to 48 kHz, and at this rate the sound quality of ADPCM samples is similar to AM radio. Of course this rate uses 24 kBytes/s of memory. To save memory, use ADPCM with a low sample rate. The noisy demo plays back voice samples of the digits 0-9, encoded in ADPCM at 8 kHz. The samples are quite intelligible, and together take about 26 kBytes of graphics memory.

Note that there is a hardware-imposed restriction on the arguments to GD.sample(). The base address and length must be multiples of 8. However, the asset converter tool aligns and pads samples automatically, so that you don't need to do anything special.

```cpp
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

#include "noisy_assets.h"

void setup()
{
  GD.begin();
  LOAD_ASSETS();
}

static void saydigit(byte n)
{
  uint32_t base, len;
  switch (n) {
    case 0: base = DIGIT_0; len = DIGIT_0_LENGTH; break;
    case 1: base = DIGIT_1; len = DIGIT_1_LENGTH; break;
    case 2: base = DIGIT_2; len = DIGIT_2_LENGTH; break;
    case 3: base = DIGIT_3; len = DIGIT_3_LENGTH; break;
    case 4: base = DIGIT_4; len = DIGIT_4_LENGTH; break;
    case 5: base = DIGIT_5; len = DIGIT_5_LENGTH; break;
    case 6: base = DIGIT_6; len = DIGIT_6_LENGTH; break;
    case 7: base = DIGIT_7; len = DIGIT_7_LENGTH; break;
    case 8: base = DIGIT_8; len = DIGIT_8_LENGTH; break;
    case 9: base = DIGIT_9; len = DIGIT_9_LENGTH; break;
  }
  GD.sample(base, len, 8000, ADPCM_SAMPLES);
}

void loop()
{
  saydigit(GD.random(10)); delay(1000);
}
```
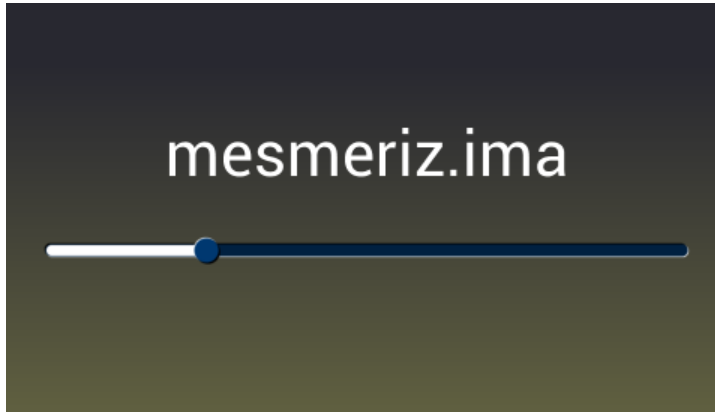
6.4   Continuous playback

For long samples, keeping the entire sample in graphics memory is usually not practical. Instead the hardware's sample playback system can continuously play back a loop of samples, while the Arduino keeps the loop fed with fresh samples from SDcard. The `Streamer` class in the GD2 library conveniently handles these details. To use it, call its `begin()` method with the name of a music file on the microSD card. `Streamer` defaults to a sample rate of 44.1 kHz, and IMA ADPCM sample encoding. It uses a 4K buffer for the streaming samples, located at the very top of graphics memory.

Calling `Streamer`'s `feed()` method reads samples from the file into the buffer. Your application should call `feed()` often enough to prevent the buffer running out of samples. Here the code calls `stream.feed()` once per frame.

`Streamer` can also tell how far the file has progressed. `progress()` returns a pair of 16-bit numbers, `val` and `range`, which represent progress as a fraction (`val/range`). This example passes `val` and `range` directly to a `cmd_slider` widget.

If you have the `sox` audio utility installed, you can convert `.wav` samples to ADPCM IMA format on the command-line like this:

```
$ sox mesmeriz.wav -c 1 mesmeriz.ima
$ play -r 44100 mesmeriz.ima
```

The `play` plays back the converted file. Because `.ima` is a headerless format, `play` needs to know the sample rate, in this case 44100 Hz.

```
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

#define MUSICFILE    "mesmeriz.ima"

static Streamer stream;

void setup()
{
  GD.begin();
  stream.begin(MUSICFILE);
}

void loop()
{
  GD.cmd_gradient(0, 40, 0x282830,
                  0, 272, 0x606040);
  GD.cmd_text(240, 100, 31, OPT_CENTER, MUSICFILE);
  uint16_t val, range;
  stream.progress(val, range);
  GD.cmd_slider(30, 160, 420, 8, 0, val, range);
  GD.swap();
  GD.finish();
  stream.feed();
}
```

Chapter 7

Accelerometer

Gameduino 2 has a 3-axis accelerometer sensor connected to the Arduino's analog inputs A0, A1 and A2. To read the accelerometer inputs, call `get_accel` like this:

```
int x, y, z;
GD.get_accel(x, y, z);
```

get_accel() corrects the values so that the force of gravity, 1G, gives a deflection in x, y or z of 256. Tilting Gameduino 2 all the way to the left, for example, gives x = -256, y = 0, z = 0. Because the accelerometer is an analog input, these values are only approximate, and in the tilt demo you can see that there is a little "noise" in the accelerometer reports.

```
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

void setup()
{
  GD.begin();
}

void loop()
{
  GD.get_inputs();
  int x, y, z;
  GD.get_accel(x, y, z);

  GD.Clear();
  GD.LineWidth(16 * 3);
  int xp = 240 + x;
  int yp = 136 + y;
  GD.Begin(LINES);
  GD.Vertex2f(16 * 240, 16 * 136);
  GD.Vertex2f(16 * xp, 16 * yp);

  GD.PointSize(16 * 40);
  GD.Begin(POINTS);
  GD.Vertex2f(16 * xp, 16 * yp);

  GD.swap();
}
```

Chapter 8

MicroSD card

Gameduino 2 includes a standard microSD card slot. It is connected to the same SPI bus as Gameduino 2's GPU, and is enabled by Arduino pin 9. There are several software libraries available for driving microSD cards. The GD library includes a lightweight library that handles FAT microSD cards, and integrates with the GD graphics system.

MicroSD files are usually fed directly to the GPU's command stream. The `GD.load()` command does exactly this: it reads a file from the microSD card and copies its contents to the Gameduino 2 's GPU. For example, to load a JPEG file, executing:

```
GD.cmd_loadimage(0);
GD.load("kitten.jpg");
```

The `cmd_loadimage` tells the GPU to expect a JPEG, and then `load` streams all the JPEG data into the GPU. As another example, loading 16 Kbytes of raw data from "`mem.raw`" from the microSD card can be done with:

```
GD.cmd_memwrite(0, 16384);
GD.load("mem.raw");
```

Again, `cmd_memwrite` tells the GPU to expect the following data, and then
`load` streams the data in.

The files with suffix `.gd2` created by the asset converter (*Converting graph-
ics* on p.39) are pure command streams, just by executing:

```
GD.load("frogger.gd2");
```

all the graphics required by the `frogger` game are loaded into memory, and
all bitmap handles are set up. When the asset converter generates the header
file `frogger_assets.h`, it includes a line:

```
#define LOAD_ASSETS()   GD.load("frogger.gd2")
```

So the `frogger` sketch can just use `LOAD_ASSETS()` in its `setup()` to set up all
its graphics resources.

Part II

Reference

Chapter 9

Drawing commands

9.1   `AlphaFunc`

Sets the alpha-test function

```
void AlphaFunc(byte func,
               byte ref);
```

    func   comparison function, one of NEVER, LESS, LEQUAL, GREATER, GEQUAL, EQUAL, NOTEQUAL, or ALWAYS

    ref   reference value for comparison function

    The alpha test function tests each pixel's alpha value, and only draws the pixel if the test passes. For example

```
GD.AlphaText(GEQUAL, 160);
```

means that only pixels with $(A \geq 160)$ are drawn. The default state is ALWAYS, which means that pixels are always drawn.
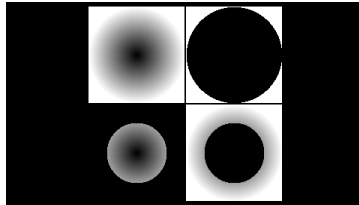

```
GD.Begin(BITMAPS);
GD.Vertex2ii(110, 6);                   // Top left: ALWAYS
GD.AlphaFunc(EQUAL, 255);               // Top right: (A == 255)
GD.Vertex2ii(240, 6);
GD.AlphaFunc(LESS, 160);                // Bottom left: (A < 160)
GD.Vertex2ii(110, 136);
GD.AlphaFunc(GEQUAL, 160);              // Bottom right: (A >= 160)
GD.Vertex2ii(240, 136);
```

9.2 Begin

Selects the graphics primitive for drawing

```
void Begin(byte prim);
```

    prim   one of BITMAPS, POINTS, LINES, LINE_STRIP, EDGE_STRIP_R,
           EDGE_STRIP_L, EDGE_STRIP_A, EDGE_STRIP_B or RECTS

The Begin command sets the current graphics draw primitve. It does not draw anything - that is done by a later Vertex2f or Vertex2ii command. The drawing primitive can be:

BITMAPS   each vertex draws a bitmap
POINTS   each vertex draws an anti-aliased point
LINES   each pair of vertices draws an anti-aliased line
RECTS   each pair of vertices draws an anti-aliased rectangle
LINE_STRIP   the vertices define a connected line segment
EDGE_STRIP_A like LINE_STRIP, but draws pixels above the line
EDGE_STRIP_B like LINE_STRIP, but draws pixels below the line
EDGE_STRIP_L like LINE_STRIP, but draws pixels left of the line
EDGE_STRIP_R like LINE_STRIP, but draws pixels right of the line
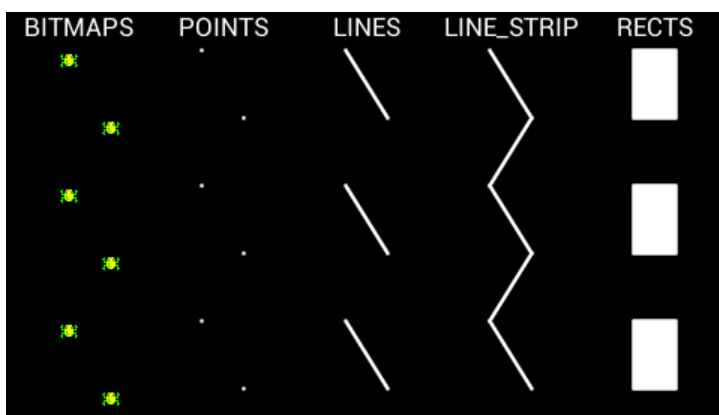
9.3 `BitmapHandle`

Sets the bitmap handle

```
void BitmapHandle(byte handle);
```

    `handle`   integer handle number, 0-15

The `BitmapHandle` command sets the current bitmap handle, used by `Vertex2f`, `BitmapSource`, `BitmapLayout` and `BitmapSize`.

The bitmap handle is part of the graphics context; its default value is 0.

9.4 `BitmapLayout`

Sets the bitmap layout

```
void BitmapLayout(byte format,
                  uint16_t linestride,
                  uint16_t height);
```

format   pixel format of the bitmap, one of: `ARGB1555`, `L1`, `L4`, `L8`, `RGB332`, `ARGB2`, `ARGB4`, `RGB565`, `PALETTED`, `TEXT8X8`, `TEXTVGA`, `BARGRAPH`.

linestride   the size in bytes of one line of the bitmap in memory

height   height of the bitmap in pixels

The `BitmapLayout` command sets the current bitmap's layout in memory. The `format` controls how memory data is converted into pixels. Each pixel in memory is 1,4,8 or 16 bits. The color is extracted from these bits as follows, where "v" is the pixel data.

format	bits per pixel	alpha	red	green	blue
L1	1	v_0	1.0	1.0	1.0
L4	4	$v_{3..0}$	1.0	1.0	1.0
L8	8	$v_{7..0}$	1.0	1.0	1.0
RGB332	8	1.0	$v_{7..5}$	$v_{4..2}$	$v_{1..0}$
RGB565	16	1.0	$v_{15..1}$	$v_{10..5}$	$v_{4..0}$
ARGB2	8	$v_{7..6}$	$v_{5..4}$	$v_{3..2}$	$v_{1..0}$
ARGB4	16	$v_{15..12}$	$v_{11..8}$	$v_{7..4}$	$v_{3..0}$

1 and 4 bit pixels are packed in bytes from left to right, so leftmost pixels in the bitmap occupy the most significant bits in a byte. 16 bit pixels are little-endian in graphics memory, and must be aligned on even memory boundaries.

`PALETTED` format uses 8 bits per pixel, each pixel is an index into the 256 entry 32-bit color table loaded at `RAM_PALETTE`.

9.5 `BitmapSize`

Sets the bitmap size and appearance

```
void BitmapSize(byte filter,
                byte wrapx,
                byte wrapy,
                uint16_t width,
                uint16_t height);
```

    filter   bitmap pixel filtering, NEAREST or BILINEAR

    wrapx x wrapping mode, BORDER or REPEAT

    wrapy y wrapping mode, BORDER or REPEAT

    width   on-screen drawn width, in pixels

    height   on-screen drawn height, in pixels

   The BitmapSize command controls how the current bitmap appears on
screen.

9.6 BitmapSource

Sets the bitmap source address

```
void BitmapSource(uint32_t addr);
```

    addr   base address for bitmap 0x00000 - 0x3ffff

The BitmapSource command sets the base address for the bitmap. For 16-bit bitmaps, this address must be even.

9.7 `BlendFunc`

Sets the color blend function

```
void BlendFunc(byte src,
               byte dst);
```

   src   source blend factor, one of ZERO, ONE, SRC_ALPHA, DST_ALPHA,
         ONE_MINUS_SRC_ALPHA, ONE_MINUS_DST_ALPHA.

   dst   destination blend factor, one of ZERO, ONE, SRC_ALPHA,
         DST_ALPHA, ONE_MINUS_SRC_ALPHA, ONE_MINUS_DST_ALPHA.

   The `BlendFunc` command sets the blend function used to combine pixels
with the contents of the frame buffer. Each incoming pixel's color is multiplied
by the source blend factor, and each frame buffer pixel is multiplied by the
destination blend factor. These two results are added to give the final pixel
color.
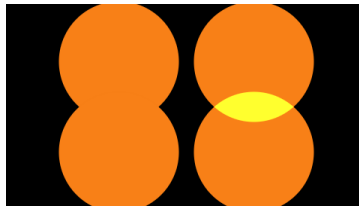

```
GD.Begin(POINTS);
GD.ColorRGB(0xf88017);
GD.PointSize(80 * 16);
GD.BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA);
GD.Vertex2ii(150, 76); GD.Vertex2ii(150, 196);
GD.BlendFunc(SRC_ALPHA, ONE);
GD.Vertex2ii(330, 76); GD.Vertex2ii(330, 196);
```

9.8 Cell

Sets the bitmap cell

```
void Cell(byte cell);
```

cell   cell number 0-127

The Cell command sets the current bitmap cell used by the Vertex2f command.


```
GD.Begin(BITMAPS);
GD.Vertex2ii(  0, 10, WALK_HANDLE, 0);
GD.Vertex2ii( 50, 10, WALK_HANDLE, 1);
GD.Vertex2ii(100, 10, WALK_HANDLE, 2);
GD.Vertex2ii(150, 10, WALK_HANDLE, 3);
GD.Vertex2ii(200, 10, WALK_HANDLE, 4);
GD.Vertex2ii(250, 10, WALK_HANDLE, 5);
GD.Vertex2ii(300, 10, WALK_HANDLE, 6);
GD.Vertex2ii(350, 10, WALK_HANDLE, 7);
```

9.9 ClearColorA

Sets the alpha component of the clear color

```
void ClearColorA(byte alpha);
```

alpha   Clear color alpha component, 0-255

The ClearColorA command sets the clear color A channel value. A subsequent Clear writes this value to the frame buffer alpha channel.

9.10 Clear

Clears the screen

```
void Clear(byte c = 1,
           byte s = 1,
           byte t = 1);
```

c   if set, clear the color buffer

s   if set, clear the stencil buffer

t   if set, clear the tag buffer

The Clear command clears the requested frame buffers.


```
GD.ClearColorRGB(0x0000ff); // Clear color to blue
GD.ClearStencil(0x80);      // Clear stencil to 0x80
GD.ClearTag(100);           // Clear tag to 100
GD.Clear(1, 1, 1);          // Go!
```

9.11 ClearColorRGB

Sets the R,G,B components of the clear color

```
void ClearColorRGB(byte red,
                   byte green,
                   byte blue);
void ClearColorRGB(uint32_t rgb);
```

    red   red component 0-255

    green   green component 0-255

    blue   blue component 0-255

    rgb   24-bit color in RGB order, 0x000000-0xffffff

The `ClearColorRGB` command sets the clear color R,G and B values. A subsequent `Clear` writes this value to the frame buffer R,G and B channels.


```
GD.ClearColorRGB(0x008080);          // teal
GD.Clear();
GD.ScissorSize(100, 200);
GD.ScissorXY(10, 20);
GD.ClearColorRGB(0xf8, 0x80, 0x17); // orange
GD.Clear();
```

9.12 `ClearStencil`

Sets the stencil clear value

```
void ClearStencil(byte s);
```

    s   stencil buffer clear value 0-255

The `ClearStencil` command sets the stencil buffer clear value. A subsequent `Clear` writes this value to the stencil buffer.

9.13 `ClearTag`

Sets the tag clear value

```
void ClearTag(byte s);
```

    s   tag value 0-255

The `ClearTag` command sets the tag buffer clear value. A subsequent `Clear` writes this value to the tag buffer.

9.14 ColorA

Sets the A component of the current color

```
void ColorA(byte alpha);
```

    alpha   alpha value 0-255

The ColorA command sets the alpha component of the current drawing color.
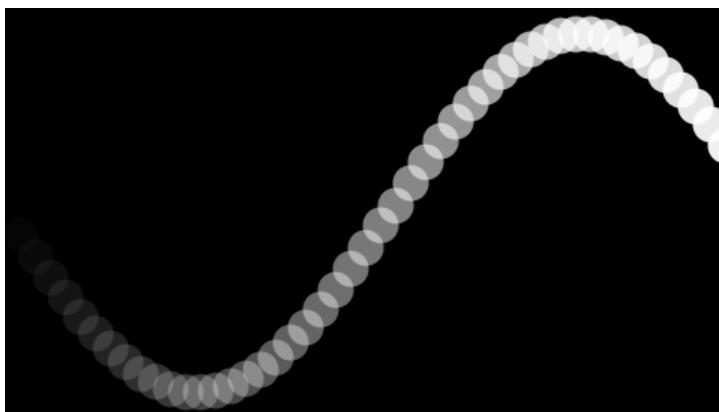

```
GD.Begin(POINTS);
GD.PointSize(12 * 16);
for (int i = 0; i < 255; i += 5) {
  GD.ColorA(i);
  GD.Vertex2ii(2 * i, 136 + GD.rsin(120, i << 8));
}
```

9.15   ColorMask

Sets the mask controlling color channel writes

```
void ColorMask(byte r,
               byte g,
               byte b,
               byte a);
```

r   if set, enable writes to the red component

g   if set, enable writes to the green component

b   if set, enable writes to the blue component

a   if set, enable writes to the alpha component

The ColorMask command sets the color mask, which enables color writes
to the frame buffer R,G,B and A components.
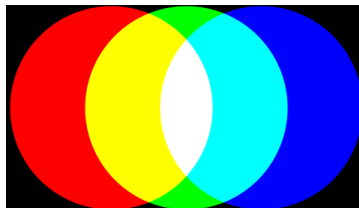

```
GD.Begin(POINTS);
GD.ColorMask(1, 0, 0, 0); // red only
GD.Vertex2ii(240 - 100, 136);
GD.ColorMask(0, 1, 0, 0); // green only
GD.Vertex2ii(240, 136);
GD.ColorMask(0, 0, 1, 0); // blue only
GD.Vertex2ii(240 + 100, 136);
```

9.16 ColorRGB

Sets the R,G,B components of the current color

```
void ColorRGB(byte red,
              byte green,
              byte blue);

void ColorRGB(uint32_t rgb);
```

    red   red component 0-255

    green   green component 0-255

    blue   blue component 0-255

    rgb   24-bit color in RGB order, 0x000000-0xffffff

The ColorRGB command sets the current color. This color is used by all drawing operations.


```
GD.Begin(RECTS);
GD.ColorRGB(255, 128, 30);    // orange
GD.Vertex2ii(10, 10); GD.Vertex2ii(470, 130);
GD.ColorRGB(0x4cc417);        // apple green
GD.Vertex2ii(10, 140); GD.Vertex2ii(470, 260);
```

9.17 `LineWidth`

Set the line width

```
void LineWidth(uint16_t width);
```

width   line width in $1/16$th of a pixel

The `LineWidth` command sets the line width used when drawing `LINES` and `LINE_STRIP`. The width is specified in $1/16$th of a pixel, so `LineWidth(16)` sets the width to 1 pixel. Note that the width is the distance from the center of the line to its outside edge, rather like the radius of a circle. So the total width of the line is double the value specified.

The maximum line width is 4095, or $255\frac{15}{16}$ pixels.
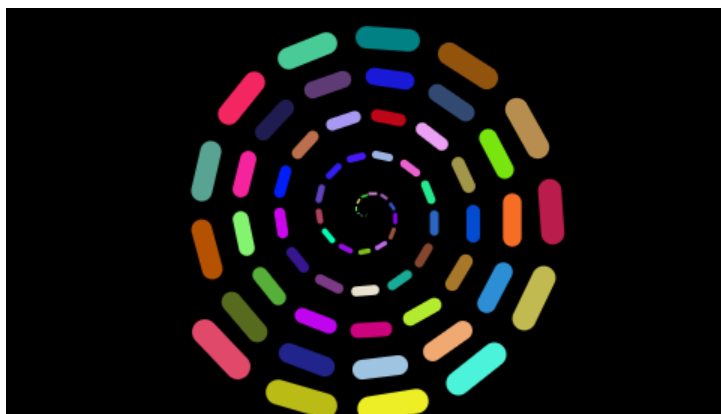

```
GD.Begin(LINES);
for (int i = 0; i < 136; i++) {
  GD.ColorRGB(GD.random(255), GD.random(255), GD.random(255));
  GD.LineWidth(i);
  GD.polar(x, y, i, i * 2500);
  GD.Vertex2ii(240 + x, 136 + y);
}
```

9.18 `PointSize`

Set the point size

```
void PointSize(uint16_t size);
```

   size   point size in $1/16$th of a pixel

   The `PointSize` command sets the point size used when drawing `POINTS`. The size is specified in $1/16$th of a pixel, so `LineWidth(16)` sets the width to 1 pixel. Note that the size is the distance from the center of the point to its outside edge, that is, the radius. So the total width of the point is double the value specified.
   The maximum point size is 4095, or $255\frac{15}{16}$ pixels.
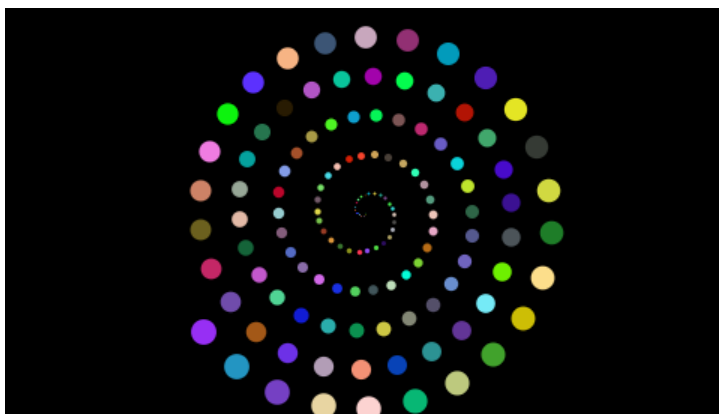

```
GD.Begin(POINTS);
for (int i = 0; i < 136; i++) {
  GD.ColorRGB(GD.random(255), GD.random(255), GD.random(255));
  GD.PointSize(i);
  GD.polar(x, y, i, i * 2500);
  GD.Vertex2ii(240 + x, 136 + y);
}
```

9.19 RestoreContext

Restore the drawing context to a previously saved state

```
void RestoreContext(void);
```

The collected graphics state is called the *graphics context*. The SaveContext command saves a copy of this state, and the RestoreContext command restores this saved copy.

The hardware can preserve up to four graphics contexts in this way. The graphics context consists of:

state	drawing commands
alpha-test function	AlphaFunc
bitmap handle	BitmapHandle
blend function	BlendFunc
bitmap cell	Cell
color clear value	ClearColorA, ClearColorRGB
stencil clear value	ClearStencil
tag clear value	ClearTag
color write mask	ColorMask
color	ColorA, ColorRGB
line width	LineWidth
point size	PointSize
scissor rectangle	ScissorSize, ScissorXY
stencil test function	StencilFunc
stencil write mask	StencilMask
stencil operation	StencilOp
tag write mask	TagMask
tag value	Tag

9.20 SaveContext

Save the graphics context

```
void SaveContext(void);
```

The collected graphics state is called the *graphics context*. The SaveContext command saves a copy of this state, and the RestoreContext command restores this saved copy.


```
GD.cmd_text(240,  64, 31, OPT_CENTER, "WHITE");
GD.SaveContext();
GD.ColorRGB(0xff0000);
GD.cmd_text(240, 128, 31, OPT_CENTER, "RED");
GD.RestoreContext();
GD.cmd_text(240, 196, 31, OPT_CENTER, "WHITE AGAIN");
```

9.21 `ScissorSize`

Set the size of the scissor rectangle

```
void ScissorSize(uint16_t width,
                 uint16_t height);
```

width   scissor rectangle width, in pixels, 0-512

height   scissor rectangle height, in pixels, 0-512

The `ScissorSize` command sets the dimensions of the scissor rectangle. The scissor rectangle limits drawing to a rectangular region on the screen.


```
GD.ScissorSize(400, 100);
GD.ScissorXY(35, 36);
GD.ClearColorRGB(0x008080); GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Scissor Example");
GD.ScissorXY(45, 140);
GD.ClearColorRGB(0xf88017); GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Scissor Example");
```

9.22 `ScissorXY`

Set the top-left corner of the scissor rectangle

```
void ScissorXY(uint16_t x,
               uint16_t y);
```

x x coordinate of top-left corner of the scissor rectangle, 0-511

y y coordinate of top-left corner of the scissor rectangle, 0-511

The `ScissorXY` command sets the top-left corner of the scissor rectangle. The scissor rectangle limits drawing to a rectangular region on the screen.

9.23 `StencilFunc`

Set the stencil test function

```
void StencilFunc(byte func,
                 byte ref,
                 byte mask);
```

   func   set the stencil comparison operation, one of NEVER, LESS,
          LEQUAL, GREATER, GEQUAL, EQUAL, NOTEQUAL, or ALWAYS

   ref   set the stencil reference value used for the comparison, 0-255

   mask   an 8-bit mask that is anded with both ref and the pixel's stencil
          value before comparison, 0-255

The `StencilFunc` command controls the stencil testing operation. During drawing, the stencil test is applied to each pixel, and if the test fails, the pixel is not drawn. Setting `func` to ALWAYS means that pixels are always drawn.

9.24 `StencilMask`

Sets the mask controlling stencil writes

```
void StencilMask(byte mask);
```

mask   Each set bit enables the corresponding bit write to the stencil
        buffer

The `StencilMask` command controls writes to the stencil buffer. Because
the stencil buffer is 8 bits deep, each bit in `mask` enables writes to the same bit
of the stencil buffer. So a `mask` of `0x00` disables stencil writes, and a `mask` of
`0xff` enables stencil writes.

9.25 `StencilOp`

Set the stencil update operation

```
void StencilOp(byte sfail,
               byte spass);
```

   sfail   the operation to be applied to pixels that fail the stencil test.
           One of ZERO, KEEP, REPLACE, INCR, DECR or INVERT.

   spass   the operation to be applied to pixels that pass the stencil test.
           One of ZERO, KEEP, REPLACE, INCR, DECR or INVERT.

   The `StencilOp` command controls how drawn pixels modify the stencil
buffer. If the the pixel failed the stencil test, then the operation specified by
`sfail` is performed. Otherwise the operation `spass` is performed.
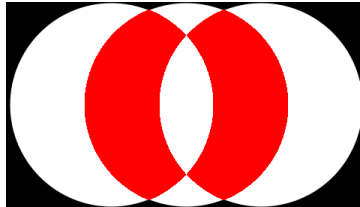

```
GD.StencilOp(INCR, INCR); // incrementing stencil
GD.PointSize(135 * 16);
GD.Begin(POINTS);            // Draw three white circles
GD.Vertex2ii(240 - 100, 136);
GD.Vertex2ii(240, 136);
GD.Vertex2ii(240 + 100, 136);
GD.ColorRGB(0xff0000);       // Draw pixels with stencil==2 red
GD.StencilFunc(EQUAL, 2, 255);
GD.Begin(RECTS);             // Visit every pixel on the screen
GD.Vertex2ii(0,0); GD.Vertex2ii(480,272);
```

9.26 TagMask

Sets the mask controlling tag writes

```
void TagMask(byte mask);
```

mask   if set, drawn pixels write to the tag buffer

The `TagMask` command controls writes to the tag buffer. If `mask` is 1, then as each pixel is drawn the byte value set by `Tag` is also written to the tag buffer. If the `mask` is 0, then drawing does not affect the tag buffer.

9.27   Tag

Set the tag value for drawing

```
void Tag(byte s);
```

   s   tag value, 0-255

The Tag command sets the byte value that is drawn into the tag buffer.

9.28 Vertex2f

Draw at a subpixel position

```
void Vertex2f(int16_t x,
              int16_t y);
```

    x x coordinate of vertex in $1/16$ths of a pixel. -16384 to 16383.

    y y coordinate of vertex in $1/16$ths of a pixel. -16384 to 16383.

The `Vertex2f` command specifies a screen position for drawing. What gets drawn depends on the current drawing object specified in `Begin`. `Vertex2f` specifies *subpixel* coordinates, so it has a precision of $1/16$th of a pixel. It also allows a coordinate range much larger than the physical screen - this is useful for drawing objects that are larger than the screen itself.

When drawing `BITMAP` `Vertex2f` uses the bitmap handle and cell currently set by `BitmapHandle` and `Cell`.

9.29 `Vertex2ii`

Draw at a integer pixel position

```
void Vertex2ii(uint16_t x,
               uint16_t y,
               byte handle = 0,
               byte cell = 0);
```

    x x coordinate of vertex in pixels, 0-511

    y y coordinate of vertex in pixels, 0-511

  handle   bitmap handle, 0-31

   cell   bitmap cell, 0-127

The `Vertex2ii` command specifies a screen position for drawing. What gets drawn depends on the current drawing object specified in `Begin`. `Vertex2ii`
When drawing `BITMAP` `Vertex2ii` uses the bitmap handle and cell specified in `handle` and `cell`. The graphics state of commands `BitmapHandle` and `Cell` are neither used not altered by this command.

Chapter 10

Higher-level commands

This section describes some of the high-level commands that Gameduino 2's GPU supports. Each of these commands runs on the GPU itself, freeing up the main MCU to run game or application code.

Some of these commands are for creating widgets, useful for UI elements. Others provide efficient ways to initialize and manage GPU memory, and load graphic and other resources.

This section only includes the hardware commands most useful for games and interactive applications. For a complete list, consult the FT800 programming guide.

10.1 cmd_append

```
void cmd_append(uint32_t ptr,
                uint32_t num);
```

The append command executes num bytes of drawing commands from graphics memory at ptr. This can be useful for using graphics memory as a cache for frequently used drawing sequences, much like OpenGL's display lists.

10.2   cmd_bgcolor

```
void cmd_bgcolor(uint32_t c);
```

The bgcolor command sets the background color used when drawing widgets. All widgets share a common style; they use the current background color (cmd_bgcolor) for non-interactive elements, and the current foreground color (cmd_fgcolor) for interactive elements. Using a darker color for cmd_bgcolor and a lighter one for cmd_fgcolor is probably a good idea.

10.3   cmd_button

```
void cmd_button(int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                byte font,
                uint16_t options,
                const char *label);
```

The `button` command draws a button widget at screen
(x, y) with pixel size `w` × `h`. `label` gives the text label.
By default this widget has a 3D look. Setting `options`
to `OPT_FLAT` gives it a 2D look.

10.4 cmd_calibrate

```
void cmd_calibrate(void);
```

The `calibrate` command runs the GPU's interactive touchscreen calibration
procedure.

10.5 cmd_clock

```
void cmd_clock(int16_t x,
               int16_t y,
               int16_t r,
               uint16_t options,
               uint16_t h,        // hours 0-23
               uint16_t m,        // minutes 0-59
               uint16_t s,        // seconds 0-59
               uint16_t ms);      // milliseconds 0-999
```

The clock command draws an analog clock at screen (x, y) with pixel radius r. The displayed time is h, m, s and ms. By default this widget has a 3D look. Setting options to OPT_FLAT gives it a 2D look.

10.6 cmd_coldstart

```
void cmd_coldstart(void);
```

The coldstart command resets all widget state to its default value.

10.7 cmd_dial

```
void cmd_dial(int16_t x,
              int16_t y,
              int16_t r,
              uint16_t options,
              uint16_t val);
```

The dial command draws a dial at screen (x, y) with pixel radius r. val gives the dial's angular position in Furmans. By default this widget has a 3D look. Setting options to OPT_FLAT gives it a 2D look.

10.8 cmd_fgcolor

```
void cmd_fgcolor(uint32_t c);
```

The `fgcolor` command Sets the foreground color used for drawing widgets. All widgets share a common style; they use the current background color (cmd_bgcolor) for non-interactive elements, and the current foreground color (cmd_fgcolor) for interactive elements. Using a darker color for cmd_bgcolor and a lighter one for cmd_fgcolor is probably a good idea.

10.9 cmd_gauge

```
void cmd_gauge(int16_t x,
               int16_t y,
               int16_t r,
               uint16_t options,
               uint16_t major,
               uint16_t minor,
               uint16_t val,
               uint16_t range);
```

The gauge command draws an analog gauge at screen (x, y) with pixel radius r. major and minor are the number of major and minor tick marks on the gauge's face. The fraction (val / range) gives the gauge's value. By default this widget has a 3D look. Setting options to OPT_FLAT gives it a 2D look.
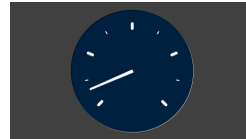
10.10 cmd_getprops

```
void cmd_getprops(uint32_t &ptr,
                  uint32_t &w,
                  uint32_t &h);
```

The getprops command queries the GPU for the properties of the last image loaded by cmd_loadimage. ptr is the image base address, and (w, h) gives its size in pixels.

10.11 cmd_gradient

```
void cmd_gradient(int16_t x0,
                  int16_t y0,
                  uint32_t rgb0,
                  int16_t x1,
                  int16_t y1,
                  uint32_t rgb1);
```

The gradient command draws a smooth color gradient, blended from color rgb0 at screen pixel (x0, y0) to rgb1 at (x1, y1). For an example of cmd_gradient(), see Gradients.

10.12 cmd_inflate

```
void cmd_inflate(uint32_t ptr);
```

The `inflate` command decompresses data into main graphics memory at `ptr`. The compressed data should be supplied after this command. The format of the compressed data is zlib DEFLATE.

10.13 `cmd_keys`

```
void cmd_keys(int16_t x,
              int16_t y,
              int16_t w,
              int16_t h,
              byte font,
              uint16_t options,
              const char *keys);
```

The `keys` command draws a rows of keys, each labeled with the characters of `chars`, at screen (x, y) with pixel size w × h. By default this widget has a 3D look. Setting `options` to `OPT_FLAT` gives it a 2D look. Specifying a character code in `options` highlights that key.

10.14 `cmd_loadidentity`

```
void cmd_loadidentity(void);
```

The loadidentity command sets the bitmap transform to the identity matrix.

10.15 cmd_loadimage

```
void cmd_loadimage(uint32_t ptr,
                   int32_t options);
```

The loadimage command uncompresses a JPEG image
into graphics memory at address ptr. The image pa-
rameters are loaded into the current bitmap handle.
The default format for the image is RGB565. If options
is OPT_MONO then the format is L8.   For an example of
cmd_loadimage(), see Bitmap handles.

10.16 cmd_memcpy

```
void cmd_memcpy(uint32_t dest,
                uint32_t src,
                uint32_t num);
```

The memcpy command copies num bytes from src to dest in graphics memory.

10.17 cmd_memset

```
void cmd_memset(uint32_t ptr,
                byte value,
                uint32_t num);
```

The `memset` command sets `num` bytes starting at `ptr` to `value` in graphics memory.

10.18 cmd_memwrite

```
void cmd_memwrite(uint32_t ptr,
                  uint32_t num);
```

The `memwrite` command copies the following `num` bytes into graphics memory, starting at addresss `ptr`.

10.19 cmd_regwrite
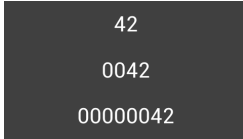
```
void cmd_regwrite(uint32_t ptr,
                  uint32_t val);
```

The `regwrite` command sets the GPU register `ptr` to `val`.

10.20 cmd_number

```
void cmd_number(int16_t x,
                int16_t y,
                byte font,
                uint16_t options,
                uint32_t val);
```

The `number` command renders a decimal number `val` in font `font` at screen (`x`, `y`). If options is n, then leading zeroes are added so that n digits are always drawn.   If options is `OPT_CENTERX` the text is centered horizontally, if `OPT_CENTERY` then vertically, and if `OPT_CENTER` then both horizontally and vertically. Adding `OPT_SIGNED` causes `val` to be treated as signed, and displayed with a leading minus sign if negative.

10.21   `cmd_progress`

```
void cmd_progress(int16_t x,
                  int16_t y,
                  int16_t w,
                  int16_t h,
                  uint16_t options,
                  uint16_t val,
                  uint16_t range);
```

The `progress` command draws a progress bar at screen (`x`, `y`) with pixel size `w` × `h`. The fraction (`val` / `range`) gives the bar's value.   This widget draws itself horizontally if `w` > `h`, otherwise it draws vertically.

10.22   `cmd_rotate`

```
void cmd_rotate(int32_t a);
```

The `rotate` command applies a rotation of a Furmans to the current bitmap transform matrix.

10.23 cmd_scale

```
void cmd_scale(int32_t sx,
               int32_t sy);
```

The `scale` command scales the current bitmap transform matrix by (sx, sy). These arguments are expressed as signed 16.16 fixed point values, so 65536 means 1.0. As a convenience, the macro `F16()` converts from floating-point to signed 16.16 representation.

10.24 cmd_scrollbar

```
void cmd_scrollbar(int16_t x,
                   int16_t y,
                   int16_t w,
                   int16_t h,
                   uint16_t options,
                   uint16_t val,
                   uint16_t size,
                   uint16_t range);
```

The `scrollbar` command draws a scroll bar at screen
(x, y) with pixel size w × h. The fraction (val / range)
gives the bar's value, and (size / range) gives its size.
This widget draws itself horizontally if w > h, otherwise
it draws vertically.

10.25 cmd_setfont

```
void cmd_setfont(byte font,
                 uint32_t ptr);
```

The `setfont` command defines a RAM `font` numbered 0-15 using a font block
at `ptr`. Before calling `setfont`, the font graphics must be loaded into memory
and the bitmap handle must be set up. The font block is 148 bytes in size, as
follows:

address	size	value
ptr + 0	128	width of each font character, in pixels
ptr + 128	4	font bitmap format, for example L1, L4 or L8
ptr + 132	4	font line stride, in bytes
ptr + 136	4	font width, in pixels
ptr + 140	4	font height, in pixels
ptr + 144	4	pointer to font graphic data in memory

10.26 cmd_setmatrix

```
void cmd_setmatrix(void);
```

The `setmatrix` command applies the current bitmap transform matrix to the next drawing operation.

10.27 cmd_sketch

```
void cmd_sketch(int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint32_t ptr,
                uint16_t format);
```

The `sketch` command starts a continuous sketching process that paints touched pixels into a bitmap in graphics memory. The bitmap's base addresss is given in `ptr`, its size in (`w`, `h`) and its position on screen is (`x`, `y`). The `format` of the bitmap can be either L1 or L8. The sketching continues until `cmd_stop` is executed. For an example of `cmd_sketch`, see *Sketching* on p.63.

10.28 cmd_slider

```
void cmd_slider(int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint16_t options,
                uint16_t val,
                uint16_t range);
```

The slider command draws a control slider at screen
(x, y) with pixel size w × h. The fraction (val / range)
gives the slider's value.   This widget draws itself hor-
izontally if w > h, otherwise it draws vertically.   By
default this widget has a 3D look. Setting options to
OPT FLAT gives it a 2D look.

10.29   cmd_spinner

```
void cmd_spinner(int16_t x,
                 int16_t y,
                 byte style,
                 byte scale);
```

The spinner command starts drawing an animated
"waiting" spinner centered at screen pixel (x, y). style
gives the spinner style; 0 is circular, 1 is linear, 2 is a
clock, and 3 is rotating disks. scale gives the size of
the graphic; 0 is small and 2 is huge.
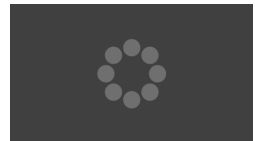
10.30   cmd_stop

```
void cmd_stop(void);
```

The `stop` command stops the current animating spinner, or the currently executing sketching operation. See `cmd_spinner` and `cmd_sketch`.

10.31 `cmd_text`

```
void cmd_text(int16_t x,
              int16_t y,
              byte font,
              uint16_t options,
              const char s);
```

The `text` command draws a text string number `s` in font `font` at screen (x, y). If options is `OPT_CENTERX` the text is centered horizontally, if `OPT_CENTERY` then vertically, and if `OPT_CENTER` then both horizontally and vertically. Adding `OPT_SIGNED` causes `val` to be treated as signed, and displayed with a leading minus sign if negative.

Hello world

For an example of `cmd_text()`, see Hello world.

10.32 `cmd_toggle`

```
void cmd_toggle(int16_t x,
                int16_t y,
                int16_t w,
                byte font,
                uint16_t options,
                uint16_t state,
                const char *s);
```

The `toggle` command draws a toggle control at screen (x, y) with width `w` pixels. The position of the toggle is given by `state`; 0 means the toggle is in the left position, 65535 in the right. The label contains a pair of strings, separated by ASCII character code `0xff`.
By default this widget has a 3D look. Setting `options` to `OPT_FLAT` gives it a 2D look.
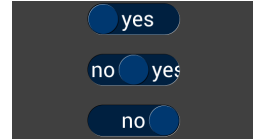
10.33 `cmd_track`

```
void cmd_track(int16_t x,
               int16_t y,
               uint16_t w,
               uint16_t h,
               byte tag);
```

The `track` command instructs the GPU to track presses that touch a pixel with `tag` and report them in `GD.inputs.track_val`. The screen rectangle at (x, y) with size (w, h) defines the track area. If the track area is 1×1 pixels in size, then the tracking is angular, and `GD.inputs.track_val` reports an angle in Furmans relative to the tracking center (x, y). Angular tracking is useful for the `cmd_dial` and `cmd_clock` widgets, and for games with a rotating control element, e.g. NightStrike.

Otherwise, the tracking is linear along the long axis of the (w, h) rectangle, and the value reported in `GD.inputs.track_val` is the distance along the rectangle, normalized to the range 0-65535. Linear tracking is useful for the `cmd_scrollbar`, `cmd_toggle` and `cmd_slider` widgets.

For an example of `cmd_track`, see Widgets and tracking controls.

10.34 `cmd_translate`

```
void cmd_translate(int32_t tx,
                   int32_t ty);
```

The `translate` command applies a translation of (`tx`, `ty`) to the bitmap transform matrix. These arguments are expressed as signed 16.16 fixed point values, so 65536 means 1.0. As a convenience, the macro `F16()` converts from floating-point to signed 16.16 representation.

Chapter 11

Management commands

11.1 `begin`

```
void begin();
```

Initialise the Gameduino 2 object. This method must be called one at program startup, before any other `GD` methods.

11.2 `finish`

```
void finish();
```

Send all pending commands to the GPU, then wait for them to complete. See `finish`.

11.3 `flush`

```
void flush();
```

Sends all pending commands to the GPU. This command only ensures that all preceding commands are sent to the GPU. It does not wait for the commands to execute. See `flush`.

11.4 `get_accel`

```
void get_accel(int &x, int &y, int &y);
```

Samples the values from the 3-axis accelerometer. The return values are scaled so that 1G gives a count of 256. The axes are oriented so that positive x is right, positive y is down, and positive z is *into* the screen.

11.5 `get_inputs`

```
void get_inputs();
```

Calls `finish`, then collects all touch and sensor input in `GD.inputs`. These inputs are:

x	touch x pixel coordinate, or -32768 if no touch
y	touch y pixel coordinate, or -32768 if no touch
rz	touch pressure, or 32767 if no touch
tag	touch tag 0-255
tag_x	touch tag x pixel coordinate
tag_y	touch tag y pixel coordinate
track_tag	touched tag for tracking controls
track_value	value for tracking controls
ptag	Software tag result

11.6 load

```
byte load(const char *filename,
          void (*progress)(long, long) = NULL);
```

Reads the contents of `filename` from the microSD card and feeds it to the GPU. If the file is not found then returns nonzero. As the file is loading, calls `progress` with the current file position, and the total file size. The function for `progress` can use these numbers to display a progress indicator.

11.7 play

```
void play(uint8_t instrument, uint8_t note = 0);
```

Play a sound. `instrument` is one of the defined instruments (p. 68). `note` is a MIDI note number (p. 69).

11.8 self_calibrate

```
void self_calibrate(void);
```

Run the built-in touch-screen calibration sequence.

11.9 sample

```
void sample(uint32_t start,
            uint32_t len,
            uint16_t freq,
            uint16_t format,
            int loop = 0);
```

Starts playback of an audio sample from main GPU memory. `start` is the starting address of the sample, `len` is its length in bytes. `freq` is the frequency of the sample in Hertz. `format` is the audio format, one of LINEAR_SAMPLES, ULAW_SAMPLES, or ADPCM_SAMPLES. If `loop` is 1, then the sample repeats indefinitely. To stop sample playback immediately, call `sample` with a zero `len`. Note that `start` and `len` must be multiples of 8.

11.10 swap

```
void swap(void);
```

Swaps the working and displayed images. This command must be called at the end of frame drawing.

Chapter 12

Math utilities

The GD library's main business is as a thin interface to the Gameduino 2's hardware. But it also includes a handful of small, useful functions for games writing. These functions don't deal with hardware – but they do some common things that games often need.

These math functions are all carefully implemented in integer only 8-bit code, so you can enjoy intense high-speed arcade action on a CPU that really ought to be running a toaster.

12.1 atan2

```
uint16_t atan2(int16_t y, int16_t x);
```

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{x}{-y}\right) & x < 0 \\ \arctan\left(\frac{x}{y}\right) + \pi & x \geq 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

atan2 returns the angle in Furmans from $(0,0)$ to the point (x, y). The range of the result is 0-65535. This function is the inverse of polar.

12.2 `polar`

```
void polar(int &x, int &y, int16_t r, uint16_t th);
```
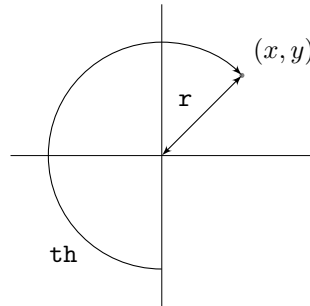

Returns the (x, y) coordinates of a point that is distance `r` from the origin, at angle `th`. `th` is in Furmans. This function is the inverse of `atan2`.

12.3 `random`

```
uint16_t random();
uint16_t random(uint16_t n);
```

Returns a random number. If no argument is given then the number is in the range 0-65535. With an argument `n`, returns a random number x such that $(0 \leq x < n)$. `random()` returns **low-quality** random numbers **quickly**. This is useful for graphics and games, where performance is often more important than true randomness.

An old trick to improve the quality of the random numbers is to call `random()` while waiting for user input. This means that when the game starts, the random number generator is in a much less predictable state.

12.4 `rcos`

```
int16_t rcos(int16_t r, uint16_t th);
```

Returns an integer approximation to the cosine of `th` (which is in Furmans) multiplied by `r`:

$$\left\lfloor \mathrm{r} \cos \frac{2\pi\mathrm{th}}{65536} \right\rfloor$$

See also `rsin`, `polar`.

12.5 `rsin`

```
int16_t rsin(int16_t r, uint16_t th);
```

Returns an integer approximation to the sine of `th` (which is in Furmans) multiplied by `r`:

$$\left\lfloor \mathrm{r} \sin \frac{2\pi\mathrm{th}}{65536} \right\rfloor$$

See also `rcos`, `polar`.

Part III

Cookbook

Chapter 13

Graphics Elements

This chapter covers some ways of using Gameduino 2's graphics features to create effects for games.

13.1   Tiled backgrounds
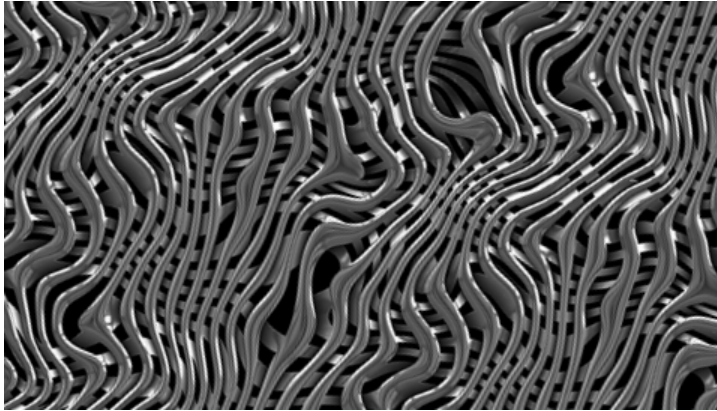

```
# include <EEPROM.h>
# include <SPI.h>
# include <GD2.h>

# include "tiled_assets.h"

void setup()
{
  GD.begin();
  LOAD_ASSETS();
}

void loop()
{
  GD.Clear();
  GD.Begin(BITMAPS);
  GD.BitmapSize(BILINEAR, REPEAT, REPEAT, 480, 272);
  GD.cmd_rotate(3333);
  GD.cmd_setmatrix();
  GD.Vertex2ii(0, 0);
  GD.swap();
}
```

The source image is this 256×256 seamless texture, created by Patrick Hoesly[1].
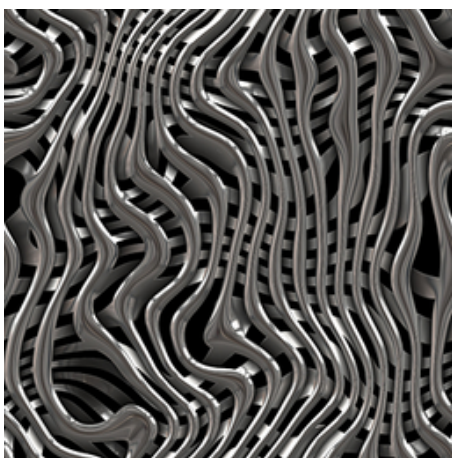
tiling the image across the entire 480×272 screen using `BitmapSize` gives:
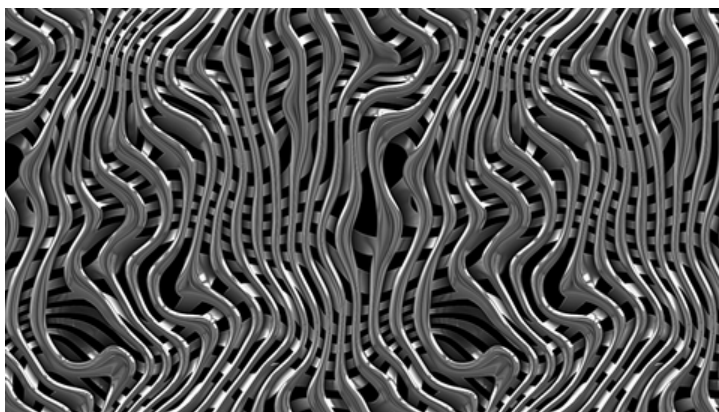
Which is nice but the repetition is clearly visible. Rotating the bitmap using `cmd_rotate` by an odd angle – 3333 Furmans is about $18°$ – makes the repetition much less obvious.

---

[1] One of the hundreds that he has made available on Flickr, all under a Creative Commons licence.

13.2 Drop shadows

To give text a drop shadow, first draw the text in the shadow color, offset by a short distance in both x and y. Then draw the text a second time in the foreground color.

```
GD.cmd_gradient(0, 0,    0x0060c0,
                0, 271, 0xc06000);
GD.ColorRGB(0x000000);
GD.cmd_text(237, 139, 31, OPT_CENTER, "READY PLAYER ONE");
GD.ColorRGB(0xffffff);
GD.cmd_text(240, 136, 31, OPT_CENTER, "READY PLAYER ONE");
GD.swap();
```

This example uses a 3-pixel offset, which works well with a large font. A smaller font looks better with a 2- or 1-pixel offset.

A similar technique works for bitmaps: draw the bitmap in black for the shadow, then draw the bitmap offset by a few pixels. Here the logo's shadow is lightened by drawing it with 50% transparency.

13.3 Fade in and out

A *fade* is when the whole screen gradually turns to one color. Fades to black and white are most common, but of course any color is possible. One way of creating a fade is to draw an alpha-blended rectangle covering the whole screen. The value of the alpha controls how much of the original screen is replaced by the solid rectangle. By animating alpha from 0 to 255 a fade happens.

Here is *NightStrike*'s code that fades the title screen to black.

```
GD.TagMask(0);
GD.ColorA(fade);
GD.ColorRGB(0x000000);
GD.Begin(RECTS);
GD.Vertex2ii(0, 0);
GD.Vertex2ii(480, 272);
```

Note the `TagMask` command, which disables writes to the tag buffer. Without this command, the full-screen rectangle would overwrite all the tag values on the screen.

13.4   Motion blur

Motion blur is a graphics technique that smooths out the appearance of moving objects, to give a more realistic motion. The white knight moving in the chess board above is rendered with motion blur. In freeze-frame the piece looks smeared, but as part of an animation your eye-brain interprets the blurs as a fast-moving object.

One way of rendering motion blur on Gameduino 2 is to draw the object multiple times with transparency. In the "chess" demo the number of passes is controlled by OVERSAMPLE, which is set to 8. The renderer makes multiple passes, each time computing a slightly different position for the piece. The eight stacked transparent pieces give a nicely blurred motion along the movement path.

```
GD.ColorRGB(0xffffff);
GD.ColorA((255 / OVERSAMPLE) + 50);
for (int j = 0; j < OVERSAMPLE; j++) {
  byte linear = 255 * (i * OVERSAMPLE + j) /
                      (OVERSAMPLE * MOVETIME - 1);
  byte scurve = sinus(linear);
  int x = x0 + (long)(x1 - x0) * scurve / 255;
  int y = y0 + (long)(y1 - y0) * scurve / 255;

  GD.Vertex2f(x, y);
}
```

13.5 Colors for additive blending
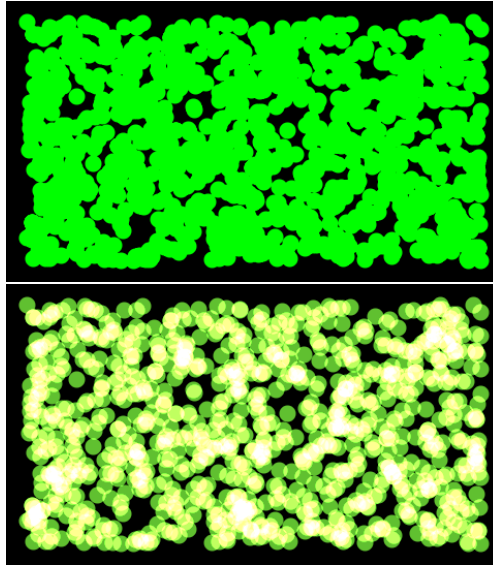
Both of these images are drawn by the code below. The code draws 1000 8-pixel radius circles with a `BlendFunc` set so that each pixel's colors are added to the background's. The difference is the drawing color: the top image uses a color of pure green (`0x00ff00`) but the bottom image uses a more subtle greenish shade (`0x60c030`).

When the color is pure green, any pixels that get drawn more than once simply stay green. But using the color `0x60c030` – which contains a lot of green, some red and a little blue – means that pixels get brighter each time they are drawn. A pixel that gets drawn 6 times reaches `0xffffff`, pure white, and cannot get any brighter.

```
GD.Clear();
GD.BlendFunc(SRC_ALPHA, ONE);
GD.PointSize(8 * 16);
GD.Begin(POINTS);
for (int i = 0; i < 1000; i++)
  GD.Vertex2ii(20 + GD.random(440), 20 + GD.random(232));
GD.swap();
```

13.6   Efficient rectangles
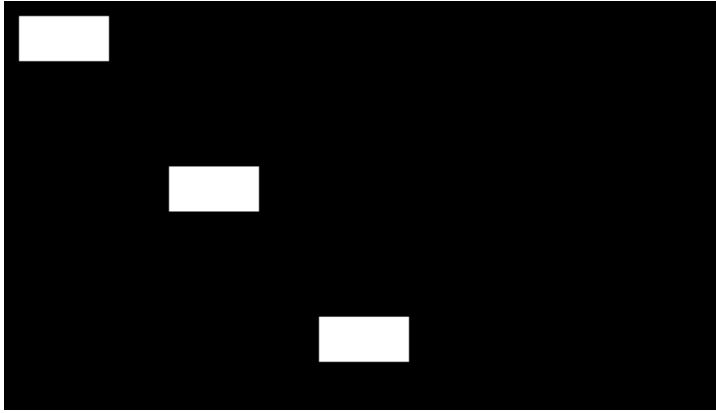
The code sets up an L8 bitmap that is sized 1×1 pixel. The blend function
(ONE, ZERO) is a straight replace operation: pixels are copied from the bitmap
into the color buffer. For an L8 bitmap, the RGB pixel values are the current
color.

The code sets the bitmap drawn size to 60×30 pixels. This means that
the 60×30 bitmap will be filled with the current color, so the result is a solid
rectangle for each vertex drawn.

Using this method for drawing rectangles differs from using RECTS in a
couple of ways. Firstly, these rectangles are not anti-aliased: in this example
they are exact 60×30 blocks of pixels with no edge smoothing. The second dif-
ference is that here every Vertex2ii draws one rectangle, but when drawing
with RECTS two vertices define each rectangle.

```
GD.BitmapLayout(L8, 1, 1);
GD.BlendFunc(ONE, ZERO);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 60, 30);

GD.Begin(BITMAPS);
GD.Vertex2ii(10, 10); // each vertex draws a 60X30 rectangle
GD.Vertex2ii(110, 110);
GD.Vertex2ii(210, 210);

GD.swap();
```

13.7   1D bitmaps
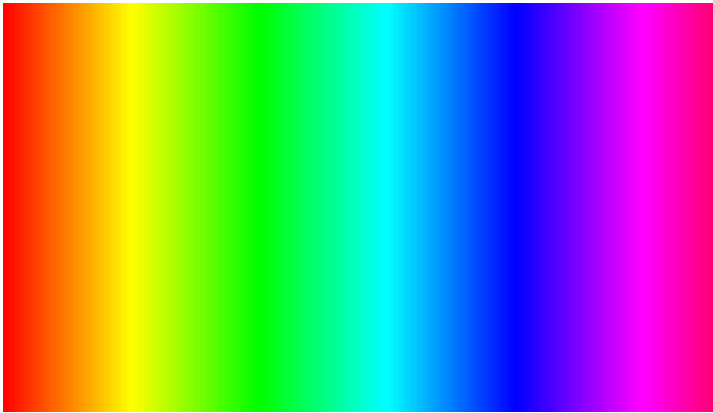

```
GD.BitmapHandle(SPECTRUM_HANDLE);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 512, 512);
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0, SPECTRUM_HANDLE);
```

The code uses this 512×1 "spectrum" image, loaded into graphics memory as an RGB565 bitmap

The BitmapSize command with REPEAT tiles the bitmap, and since the bitmap height is one pixel, it repeats on every pixel line. As with any bitmap, the 1D bitmap can be rotated and scaled. Here is the same bitmap scaled by 0.3 and rotated by 80°

13.8   Drawing polygons
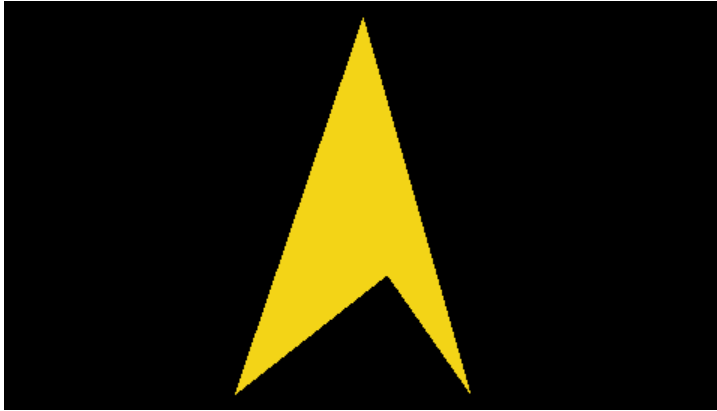
The GD library has a helper object for drawing polygons. To draw a polygon, first call its `begin()` method, then supply the vertices of the polygon in order to the `v()` method, using subpixel coordinates. Calling `draw()` draws the polygon, using the current color.

This example uses only four vertices, but a polygon can have up to 16 vertices, and can be any shape. The vertices do not all need to be on the screen.

The `Poly` object uses a sequence of stencil buffer operations to compute the outline of the polygon. It then draws the interior using the current color and alpha, much like `POINTS` or `LINES` do.

```
GD.Clear();
GD.ColorRGB(0xf3d417);
Poly po;
po.begin();
po.v(16 * 154, 16 * 262);
po.v(16 * 256, 16 * 182);
po.v(16 * 312, 16 * 262);
po.v(16 * 240, 16 *  10);
po.draw();
GD.swap();
```

Unlike POINTS or LINES, the polygon's edges are not smoothed – you can see jagged pixels along the boundary. One remedy is to outline the polygon with a line. Here a broad black outline is drawn using Poly's handy outline() method, which draws the perimeter vertices with LINE STRIP

```
GD.ColorRGB(0x000000);
GD.LineWidth(3 * 16);
po.outline();
```


Using a similar technique, the *cobra* example first draws all the panels of the spacecraft using Poly, then draws the glowing blue edges.
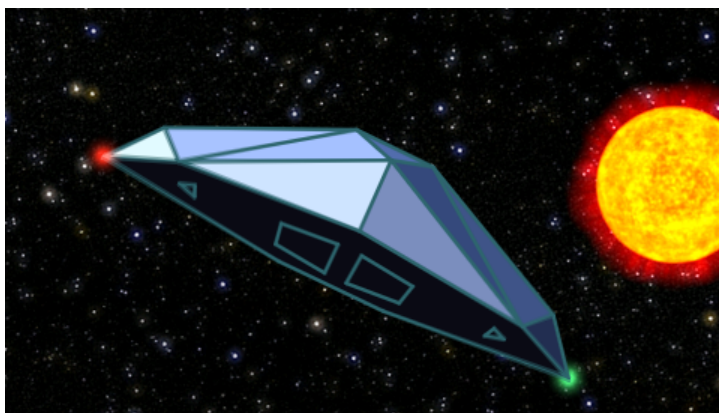
13.9   Lines everywhere

Because of their round end-caps, wide lines can be used as convenient graphic elements. The background for the text in the *sprites* demo is a 28 pixel wide line, drawn in black with about 50% alpha transparency.


```
GD.ColorRGB(0x000000);
GD.ColorA(140);
GD.LineWidth(28 * 16);
GD.Begin(LINES);
GD.Vertex2ii(240 - 110, 136, 0, 0);
GD.Vertex2ii(240 + 110, 136, 0, 0);
```

NightStrike draws the power meter at the top of the screen using three horizontal wide lines. First a 10 pixel wide transparent black line makes a background bar for the power meter. Two lines draw the power display. The first is 10 pixels wide in pale orange, the second is 4 pixels wide in bright orange. Together they give a pleasing glow effect.
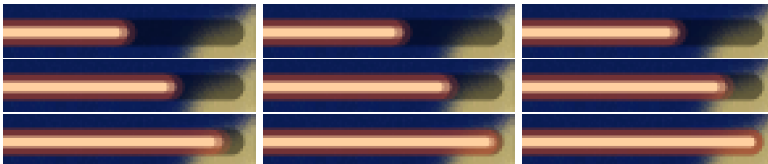
13.10 Vignette
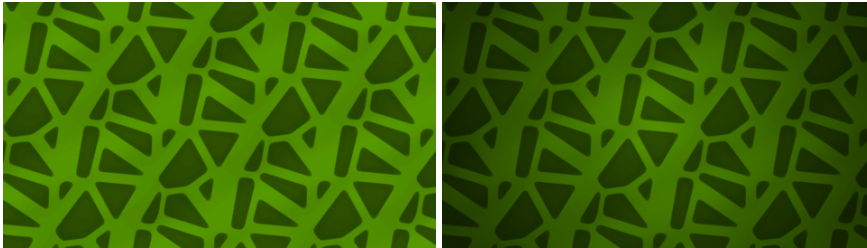
In photography, *vignetting* is a darkening towards the edges of an image. The image on the right has a vignette applied as an extra layer. The vignette image itself is a 480×272 L8 radial gradient, becoming more opaque towards the edge:
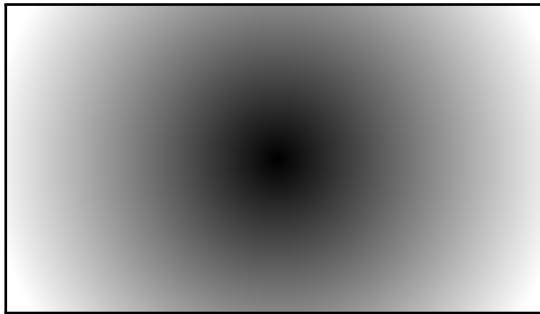
Drawing this bitmap in white will produce a milky transparency towards the edges of the screen. But drawing this bitmap in black darkens the screen towards the edges. The amount of darkening can be controlled either by changing the original bitmap, or by making the bitmap less opaque with `ColorA`.

```
GD.ColorA(0x90);
GD.ColorRGB(0x000000);
GD.Vertex2ii(0, 0, VIGNETTE_HANDLE, 0);
```

Because the vignette bitmap does not contain any fine detail, it can be reduced in resolution without affecting image quality. A 240×136 vignette bitmap is almost indistinguishable from the 480×272 version, and uses one fourth of the memory.

13.11   Mirroring sprites
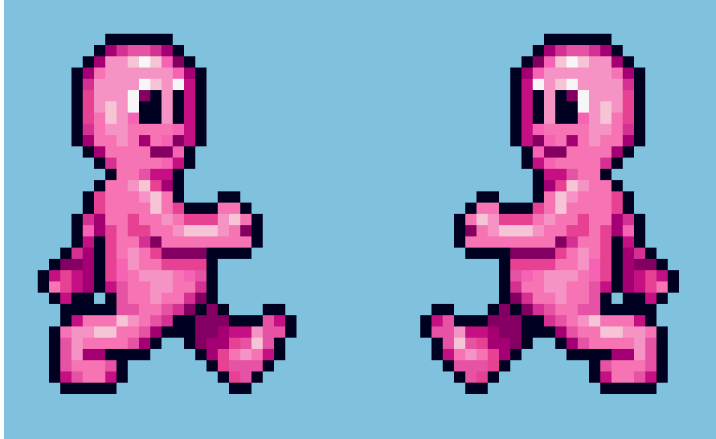
Mirroring a sprite – flipping it so that it faces in the opposite direction – is very useful in 2D games. The Gameduino 2's GPU can perform complex bitmap transforms, so a simple flip is no problem at all. You can think of a left-right flip as a scale by -1 in the x axis, which is done by `cmd_scale`. The two `cmd_translate` calls adjust the bitmap's position so that the flip happens across the halfway point of the sprite. In this case, the sprite is 32 pixels wide, so the halfway point is at $x = 16$:


```
GD.Begin(BITMAPS);
GD.Vertex2ii(  0, 10, WALK_HANDLE, 0);
GD.cmd_translate(F16(16), F16(0));
GD.cmd_scale(F16(-1), F16(1));
GD.cmd_translate(F16(-16), F16(0));
GD.cmd_setmatrix();
GD.Vertex2ii( 30, 10, WALK_HANDLE, 0);
GD.swap();
```

13.12 Silhouettes and edges
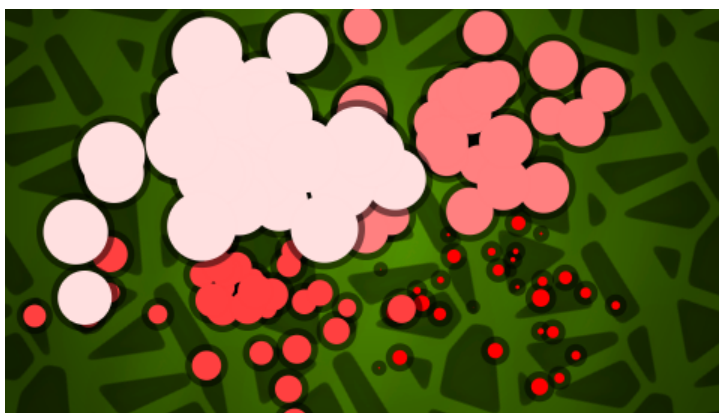
The Gameduino 2's hardware primitives only draw solid circles and lines. To draw an edge around a circle – like the dark slightly transparent edges above – draw the circle twice. The first circle is drawn in the edge color and the second, slightly smaller, circle is drawn the with the center color. For clusters of objects, the drawing order affects the final image. Here the outlines are all drawn first, which gives a common outline to the whole cluster of circles.
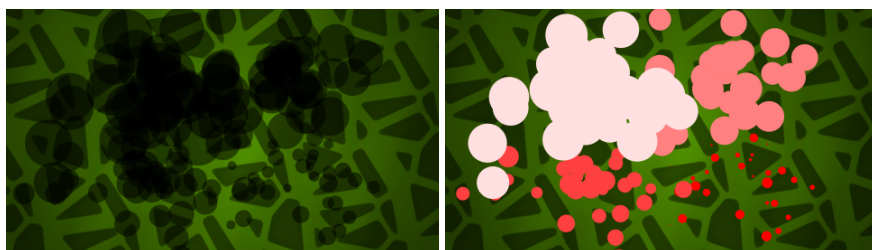
This button graphic is drawn with two wide lines. The first line is white, and four pixels wider than the blue inner line. Edges like this can be drawn with POINTS, LINES and RECTS.

13.13   Chunky pixels
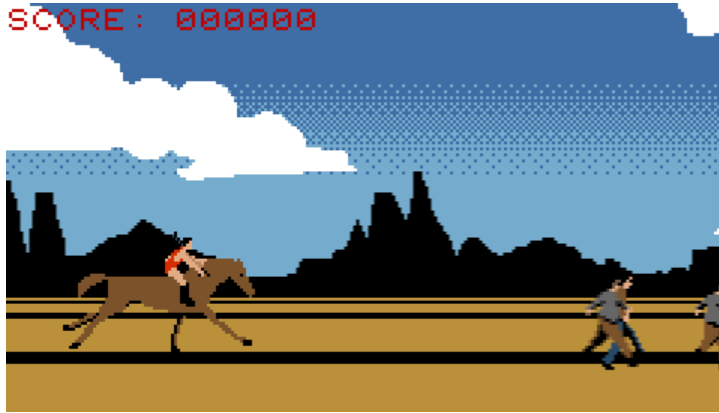
For the *Zardoz* project by artist and designer Nick Criscuolo, we wanted a more chunky, 8-bit look from the Gameduino 2's graphics. So all graphics are drawn at double-size. At the start of every frame, the game does

```
GD.cmd_scale(F16(2), F16(2));
GD.cmd_setmatrix();
```

to double up the size of every bitmap it draws. The result is that every pixel in the bitmap becomes 2×2 pixels on the screen. This makes the effective screen resolution a suitably retro 240×136.

13.14   Vector graphics
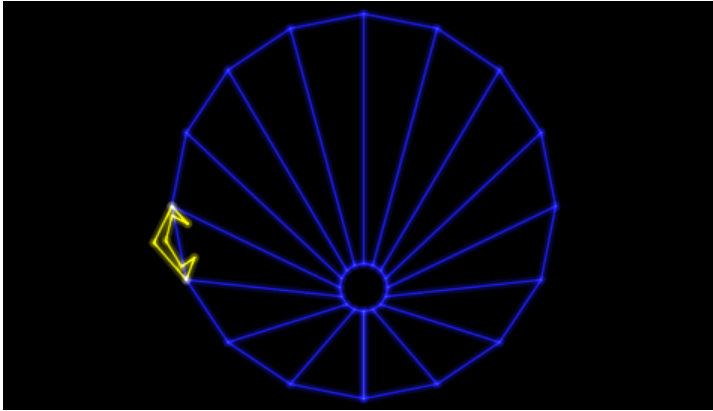
Vector graphics games from the 70s, 80s and 90s used special-purpose hardware to draw bright, colored lines onto a CRT display.   As a graphics technology, the main benefit was a lack of pixels: everything was drawn with smooth, bright lines and points. Gameduino 2 has excellent smooth line and point support, so can easily emulate the vector style of graphics.

However a straight line-drawing game looks a little too 'clean'. The original displays produced a nice glowing look to the lines, which the code below mimics using two passes. The function `drawgame()` draws the whole of the screen, using `LINES`. The first pass draws the background glow for all the lines, so it calls `drawgame()` with wide, dim lines. The second pass draws exactly the same screen, but with bright, thin lines. This two-pass drawing gives an appropriate subtle glow to all the lines.

```
GD.Clear();                      // Clear to black

GD.ColorA(0x30);                 // Draw background glows
GD.LineWidth(48);
drawgame();

GD.ColorA(0xff);                 // Draw foreground vectors
GD.LineWidth(10);
GD.BlendFunc(SRC_ALPHA, ONE); // additive blending
drawgame();
```

13.15   Handmade graphics
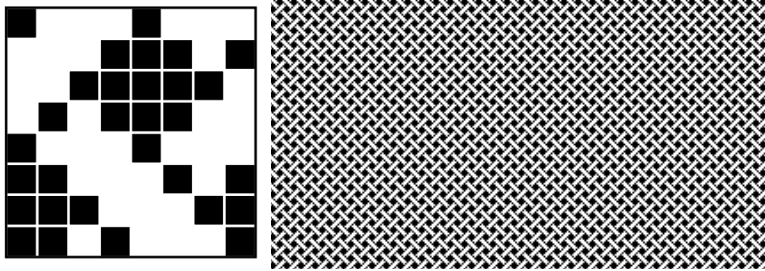
Early humans didn't have the sophisticated tools we enjoy today. They had to draw graphics on paper, then turn them into data by hand. You can still make graphics the same way today.   cmd_memwrite writes the 8-byte pattern into memory starting at address zero. The 1-bit graphic data is coded in picture[] – The "0b" prefix means binary. Then it sets up the bitmap parameters, again by hand, for an $8{\times}8$ L1 bitmap, repeating in x and y.

```
GD.cmd_memwrite(0, 8);
static const PROGMEM prog_uchar picture[] = {
  0b01110111,
  0b11100010,
  0b11000001,
  0b10100011,
  0b01110111,
  0b00111010,
  0b00011100,
  0b00101110,
};
GD.copy(picture, 8);

GD.BitmapSource(0);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 480, 272);
GD.BitmapLayout(L1, 1, 8);

GD.Clear();
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0);
```

Chapter 14

Compositing

This chapter covers some more sophisticated graphics techniques that push the GPU hardware a little further. They all use compositing – controlling the color pipeline using the alpha channel.

14.1   Alpha compositing

This stylish clock graphic is drawn using a white circle, a smaller black circle, then a `cmd_clock` widget to draw the hands.

```
GD.Begin(POINTS);
GD.PointSize(16 * 120); // White outer circle
GD.Vertex2ii(136, 136);
GD.ColorRGB(0x000000);
GD.PointSize(16 * 110); // Black inner circle
GD.Vertex2ii(136, 136);

GD.ColorRGB(0xffffff);
GD.cmd_clock(136, 136, 130,
             OPT_NOTICKS | OPT_NOBACK, 8, 41, 39, 0);
```

But adding a JPEG background image gives an unfortunate result

The huge black disk spoils the look. It would be much better to make the central area transparent. Here is one way of doing that, using the *alpha buffer*.

Every pixel on the screen has an alpha value 0-255 that is usually updated at the same time as the color buffer. Normally this alpha buffer is not important, because it is not visible and its contents do not affect the final image. But ColorMask can enable and disable writes to the color and alpha buffers. This technique "paints" the graphic into the alpha buffer, then draws every pixel on the screen using a blend mode of BlendFunc(DST_ALPHA, ONE), which draws the pixels where the alpha buffer is non-zero.

After drawing the tree background image, the first step is to use ColorMask to disable color buffer updates, and set the BlendFunc to a value that writes incoming alpha directly into the alpha buffer, by specifying a *source blend factor* of ONE:

```
GD.ColorMask(0,0,0,1);
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
```

Now draw the outer circle just as before

```
GD.Begin(POINTS);
GD.PointSize(16 * 120); // outer circle
GD.Vertex2ii(136, 136);
```

Nothing appears on the screen yet, because the drawing is only affecting the alpha buffer, which is – so far – invisible. Drawing the inner circle requires a blend mode that clears any drawn pixels to zero, so the source blend factor is ZERO:

```
GD.BlendFunc(ZERO, ONE_MINUS_SRC_ALPHA);
GD.PointSize(16 * 110); // inner circle
GD.Vertex2ii(136, 136);
```

Finally the clock widget itself is drawn, again using a source blend factor of ONE:

```
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
GD.cmd_clock(136, 136, 130,
             OPT_NOTICKS | OPT_NOBACK, 8, 41, 39, 0);
```

After these operations, nothing has been drawn on the visible screen yet – because the ColorMask disabled writes to R,G and B. But the alpha buffer now contains this image:

The final step is to make the alpha buffer visible. The code does this by drawing a gray rectangle over the entire screen. Of course, this would simply result in a gray screen, but the source blend factor is set to DST_ALPHA, so the transparency values come from the alpha buffer.

```
GD.ColorMask(1,1,1,0);
GD.BlendFunc(DST_ALPHA, ONE);
GD.ColorRGB(0x808080);
GD.Begin(RECTS);          // Visit every pixel on the screen
GD.Vertex2ii(0,0);
GD.Vertex2ii(480,272);
```

The complete code is

```
GD.Clear();                 // now alpha is all zeroes
GD.ColorMask(1,1,1,0);      // draw tree, but leave alpha zero
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0);

GD.ColorMask(0,0,0,1);
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
GD.Begin(POINTS);
GD.PointSize(16 * 120); // outer circle
GD.Vertex2ii(136, 136);
GD.BlendFunc(ZERO, ONE_MINUS_SRC_ALPHA);
GD.PointSize(16 * 110); // inner circle
GD.Vertex2ii(136, 136);
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
GD.cmd_clock(136, 136, 130,
             OPT_NOTICKS | OPT_NOBACK, 8, 41, 39, 0);

GD.ColorMask(1,1,1,0);
GD.BlendFunc(DST_ALPHA, ONE);
GD.ColorRGB(0x808080);
GD.Begin(RECTS);            // Visit every pixel on the screen
GD.Vertex2ii(0,0);
GD.Vertex2ii(480,272);
```

14.2   Slot gags

Slot gags are a very old animation technique for masking two elements together[1]. In this example of a slot gag, drawing the Gameduino logo bitmap sets the alpha buffer to 255 in the logo's pixels, and 0 elsewhere. Then by drawing a wide diagonal line with `BlendFunc` set to (DST_ALPHA, ONE) the line is masked with the alpha buffer, so only draws pixels where the alpha buffer is 255. The result is an animated glint effect.

```
GD.Vertex2ii(240 - GAMEDUINO_WIDTH / 2,
             136 - GAMEDUINO_HEIGHT / 2,
             GAMEDUINO_HANDLE);

static int x = 0;
GD.LineWidth(20 * 16);
GD.BlendFunc(DST_ALPHA, ONE);
GD.Begin(LINES);
GD.Vertex2ii(x, 0);
GD.Vertex2ii(x + 100, 272);
x = (x + 20) % 480;
```

---

[1] See "Digital Compositing for Film and Video" by Steve Wright.

14.3   Patterned text

Patterned text uses a similar technique to slot gags. First the text is drawn into the alpha buffer *only*. Then a full-screen repeating 8×8 bitmap pattern is drawn using the alpha buffer to control the amount of opacity.


```
GD.BitmapHandle(STRIPE_HANDLE);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 480, 272);

GD.ClearColorRGB(0x103000);
GD.Clear();

GD.ColorMask(0, 0, 0, 1);    // write A only
GD.BlendFunc(ONE, ONE);
GD.cmd_text(240, 136, 31, OPT_CENTER,
            "STRIPES ARE IN, BABY!");

GD.ColorMask(1, 1, 1, 0);    // write R,G,B only
GD.BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA);
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0, STRIPE_HANDLE);
```

14.4   Alpha operators
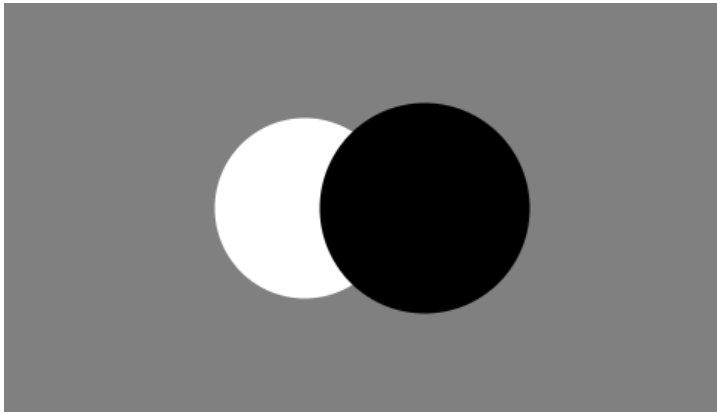
You can make alpha-compositing code easier to read by using a more expressive name for the blend modes. Here PAINT_ALPHA() means "draw something into the alpha buffer". CLEAR_ALPHA() means "erase something from alpha buffer".

```
#define PAINT_ALPHA()  GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA)
#define CLEAR_ALPHA()  GD.BlendFunc(ZERO, ONE_MINUS_SRC_ALPHA)

  GD.ClearColorA(0x80);
  GD.Clear();

  PAINT_ALPHA();
  draw_left_circle();

  CLEAR_ALPHA();
  draw_right_circle();
```

14.5   Round-cornered images

On the left is a 128×128 bitmap drawn using a single `Vertex2ii` call. On the right is the same bitmap drawn using beautiful anti-aliased round corners. The technique is to draw a round-cornered rectangle into the alpha buffer, then use this rectangle to control the bitmap transparency, using `BlendFunc`.

The corner radius `r` controls how round the corners are, in pixels. A value of 1 is no rounding, higher values give a more rounded look.

```
GD.Begin(BITMAPS);

GD.Vertex2ii( 52, 50);                          // left bitmap

GD.ColorMask(0, 0, 0, 1);                        // only draw A
GD.Clear();
int r = 20;                                      // corner radius
GD.LineWidth(16 * r);
GD.Begin(RECTS);
GD.Vertex2ii(300 + r,       50 + r);             // top-left
GD.Vertex2ii(300 + 127 - r, 50 + 127 - r);       // bottom-right

GD.ColorMask(1, 1, 1, 0);                        // draw bitmap
GD.BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA);
GD.Begin(BITMAPS);
GD.Vertex2ii( 300, 50);
```

14.6   Transparent buttons

The code first draws a full-screen background image, then the buttons are each drawn into the alpha buffer by the function `button()`. Finally, the last six lines make the alpha buffer visible.

```
GD.cmd_loadimage(0, 0);
GD.load("tree.jpg");
GD.Clear();
GD.ColorMask(1, 1, 1, 0);
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0);

GD.ColorMask(0, 0, 0, 1);
int x0 = 160, x1 = 240, x2 = 320;
int y0 =  56, y1 = 136, y2 = 216;
button(x0, y0, 1); button(x1, y0, 2); button(x2, y0, 3);
button(x0, y1, 4); button(x1, y1, 5); button(x2, y1, 6);
button(x0, y2, 7); button(x1, y2, 8); button(x2, y2, 9);

GD.ColorMask(1, 1, 1, 1);
GD.ColorRGB(0xffffff);
GD.BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA);
GD.Begin(RECTS);
GD.Vertex2ii(0, 0); GD.Vertex2ii(480, 272);
```

The `button()` function first paints a white rectangle, then a slightly smaller one with partial transparency. Lastly it draws the label using `cmd_number` with `OPT_CENTER` to center the text.

```
static void button(int x, int y, byte label)
{
  int sz = 18;                        // button size in pixels

  GD.Tag(label);
  PAINT_ALPHA();
  GD.Begin(RECTS);
  GD.LineWidth(16 * 20);
  GD.Vertex2ii(x - sz, y - sz);
  GD.Vertex2ii(x + sz, y + sz);

  CLEAR_ALPHA();
  GD.ColorA(200);
  GD.ColorA(200);
  GD.LineWidth(16 * 15);
  GD.Vertex2ii(x - sz, y - sz);
  GD.Vertex2ii(x + sz, y + sz);

  GD.ColorA(0xff);
  PAINT_ALPHA();
  GD.cmd_number(x, y, 31, OPT_CENTER, label);
}
```

14.7   Reflections

After drawing the top bitmap, the code draws a 1D vertical 128×1 gradient bitmap into the alpha buffer, repeated horizontally so it covers the screen:
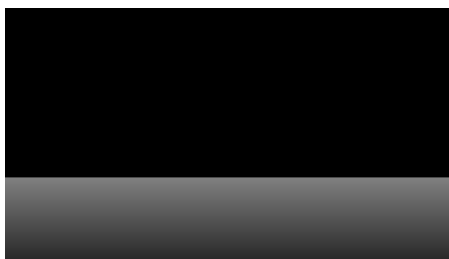
The code then sets a bitmap transform that flips the logo in the Y axis, using a method similar to the one in *Mirroring sprites* on p.154. It then draws the logo's alpha channel into the alpha buffer, masked with the existing alpha contents. Masking uses macro MASK_ALPHA(), which multiplies the logo's alpha channel with the existing alpha buffer values:

The final step is to draw the mirrored logo using the alpha buffer to control transparency.

```
#define MASK_ALPHA()    GD.BlendFunc(ZERO, SRC_ALPHA)

void loop()
{
  int x = 240 - GAMEDUINO_WIDTH / 2;

  GD.BitmapHandle(GRADIENT_HANDLE);
  GD.BitmapSize(NEAREST, REPEAT, BORDER, 480, 272);

  GD.Clear();
  GD.ColorMask(1, 1, 1, 0);               // don't touch A yet
  GD.cmd_gradient(0, 40, 0x505060,
                  0, 272, 0xc0c080);

  GD.Begin(BITMAPS);                      // top bitmap
  GD.Vertex2ii(x, 80, GAMEDUINO_HANDLE);

  GD.ColorMask(0, 0, 0, 1);
  GD.BlendFunc(ONE, ZERO);
  GD.Vertex2ii(0, 180, GRADIENT_HANDLE);


                                          // invert the image
  GD.cmd_translate(0, F16(GAMEDUINO_HEIGHT / 2));
  GD.cmd_scale(F16(1), F16(-1));
  GD.cmd_translate(0, -F16(GAMEDUINO_HEIGHT / 2));
  GD.cmd_setmatrix();

  MASK_ALPHA();                           // mask with gradient
  GD.Vertex2ii(x, 190, GAMEDUINO_HANDLE);

  GD.ColorMask(1, 1, 1, 0);               // draw the reflection
  GD.BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA);
  GD.Vertex2ii(x, 190, GAMEDUINO_HANDLE);
  GD.swap();
}
```

Chapter 15

Saving memory

15.1   Two-color images


```
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

#include "mono_assets.h"

void setup()
{
  GD.begin();
  LOAD_ASSETS();
}

void loop()
{
  GD.ClearColorRGB(0x375e03);
  GD.Clear();
  GD.Begin(BITMAPS);
  GD.ColorRGB(0x68b203);
  GD.BitmapSize(NEAREST, REPEAT, REPEAT, 480, 272);
  GD.Vertex2ii(0, 0);
  GD.swap();
}
```

The source image is this seamless texture, created by Patrick Hoesly

It would be fine to encode this image in `RGB565` format. But because it only uses two colors – and shades in-between – with some manipulation it can be loaded as an `L4` bitmap, saving 75% of the graphics memory. (See `BitmapLayout` for a list of available pixel formats and their memory usage.)

The first step is to load the original image into a graphics program and measure the two colors, in this case dark green is `0x375e03` and light green is `0x68b203`. Then convert the image to monochrome and stretch the contrast so that it ranges from pure black to pure white
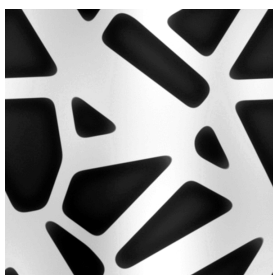
The code first clears the screen to the dark green color, then draws the `L4` bitmap in light green color. The result is looks very like the original, but uses much less graphics memory. This 128 × 128 bitmap `L4` bitmap uses 8 Kbytes.

As a bonus, you can recolor the bitmap at runtime by adjusting the foreground and background colors, *strictly* guided by the rules of good taste, of course.

15.2   The missing `L2` format

Gameduino 2's GPU supports three monochrome formats: `L1`, `L4` and `L8`. The is no `L2` format, but by drawing in two passes the format can be simulated. The trick is to split the original 2-bit image into two 1-bit `L1` bitmaps, one for the high bit and one for the low bit:
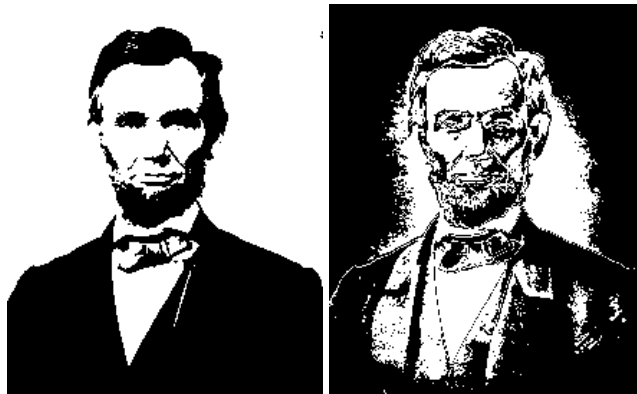
The `ABE` graphic is loaded into bitmap handle `ABE_HANDLE` in cells 0 and 1. Cell 0 contains bit 0, and cell 1 contains bit 1. The code draws the image in three passes. The first two passes draw the two bits of the bitmap into the alpha buffer. The third pass makes the alpha buffer visible.

The drawing code first disables color buffer writes using `ColorMask`. Then it draws the high bit with alpha 0xaa, and the low bit with 0x55. This gives the following values in the alpha buffer for the four 2-bit pixel codes:

high bit	low bit	alpha
0	0	0x00
0	1	0x55
1	0	0xaa
1	1	0xff

```
GD.ClearColorRGB(0x00324d);
GD.Clear();

GD.ColorMask(0, 0, 0, 1);
GD.Begin(BITMAPS);

GD.BlendFunc(ONE, ONE);
GD.ColorA(0xaa);  // draw bit 1 into A
GD.Vertex2ii(240 - ABE_WIDTH / 2, 0, ABE_HANDLE, 1);
GD.ColorA(0x55);  // draw bit 0 into A
GD.Vertex2ii(240 - ABE_WIDTH / 2, 0, ABE_HANDLE, 0);

// Now draw the same pixels, controlled by DST_ALPHA
GD.ColorMask(1, 1, 1, 1);
GD.ColorRGB(0xfce4a8);
GD.BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA);
GD.Vertex2ii(240 - ABE_WIDTH / 2, 0, ABE_HANDLE, 1);

GD.swap();
```

The final `Vertex2ii` draws the same pixels as the two earlier `Vertex2ii` commands. But because the `BlendFunc` is (DST_ALPHA, ONE_MINUS_DST_ALPHA) the bitmap pixels are ignored – instead only the contents of the alpha buffer pixels affect the pixel color.

15.3   Separated mattes
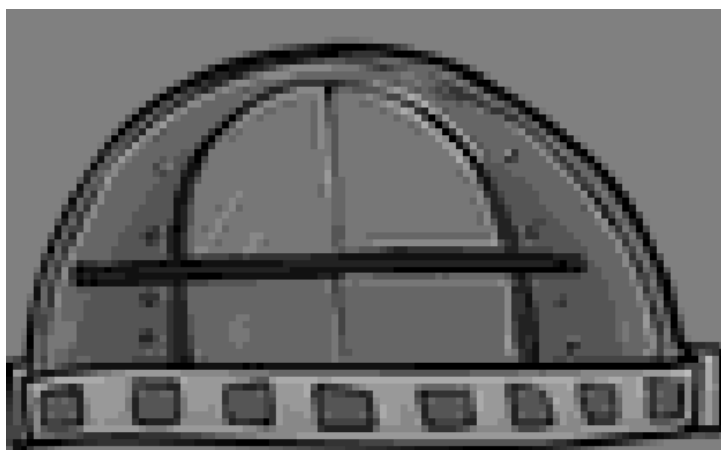
The base graphic in *NightStrike* is monochrome with an alpha channel. There is no direct support in the hardware for a monochrome-with-alpha format, but it can be emulated by splitting the bitmap into a *matte* – the original alpha channel – and a foreground bitmap.
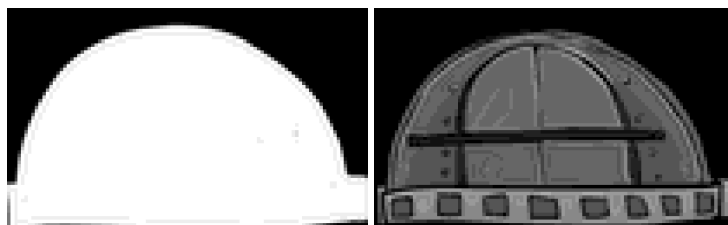
Both images are loaded as `L4` bitmaps. The matte bitmap is drawn first, with `ColorRGB` set to black. This clears every pixel covered by the matte. Then the foreground bitmap is drawn as usual.
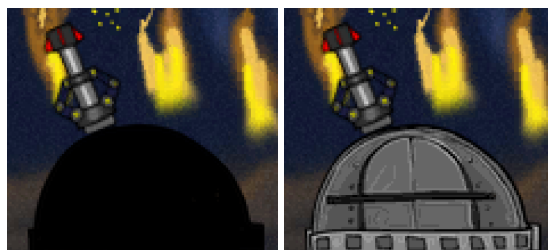
15.4 Half-resolution bitmaps

For *NightStrike,* the helicopter animation is two frames, 100 pixels wide in format `RGBA4`.

There is another four frame animation, used when the stricken helicopter is crashing to the ground. Because this animation is seen only briefly, when the sprite is moving fast, a loss of detail is not too noticable. This version is half the resolution, less than 50 pixels wide, but is scaled by `cmd_scale` so that it appears 100 pixels wide on the screen. Halving the resolution of a bitmap reduces its memory usage by 75%.
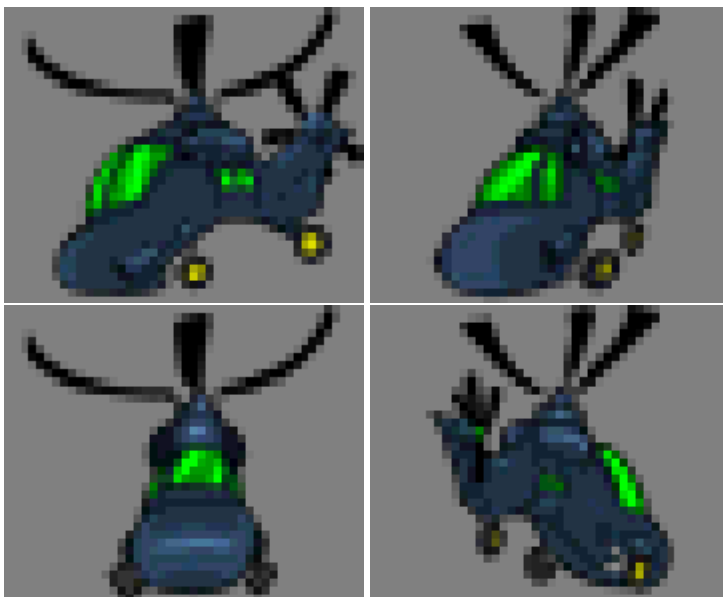
15.5   8-bit formats

For most images, the 16-bit per pixels formats – `RGB565`, `ARGB1555` and `ARGB4` – are appropriate. But sometimes the 8-bit per pixel formats can be used instead with little noticable loss in quality.

The playing cards above are encoded in `RGB332` format, so only use one byte per pixel. The explosion animation below uses `ARGB2` format.   The playing cards use bold colors, so the reduced color range of `RGB332` is not very noticable. The explosion animates very fast – the whole sequence below takes 1/4th of a second – and this rapid motion conceals the reduced color range.
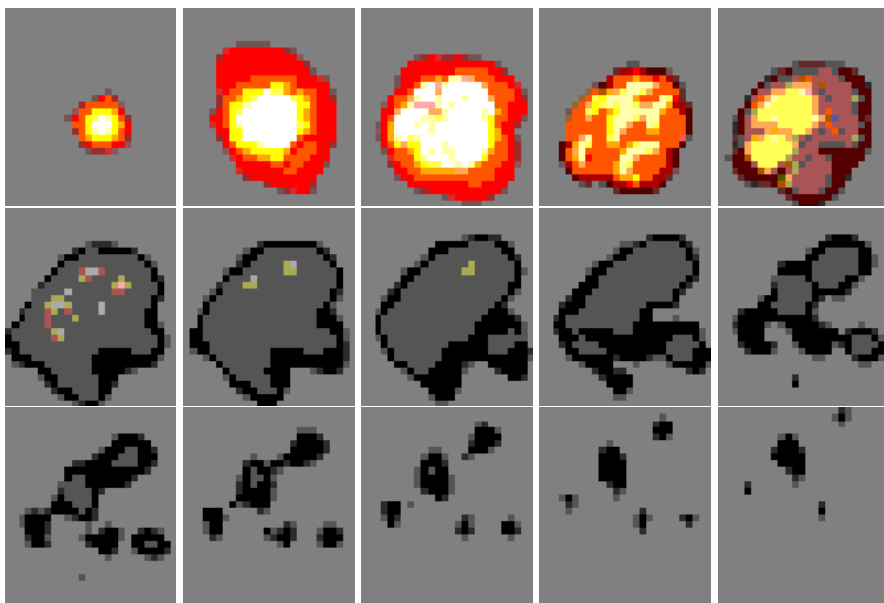
15.6 DXT1
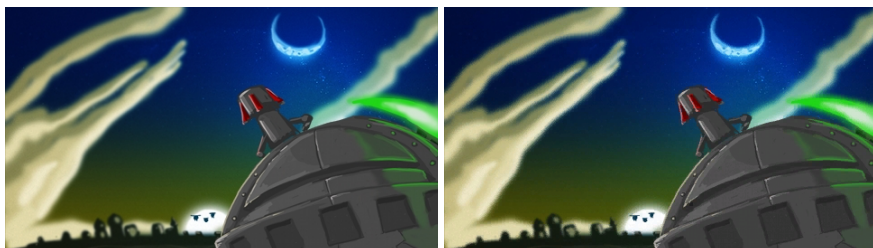
DXT is a texture compression system – it squeezes a bigger image into less graphics memory. The image on the left is the artist's original rendering of the *NightStrike* welcome screen. On the right is a screenshot from the DXT1-compressed version on the running Gameduino 2 . As you can see, the differences are quite small. But because it uses DXT1, the image on the right uses *one fourth* of the graphics memory of the image on the left.

Roughly, DXT1 works by splitting the image into 4×4 pixel tiles. Each tile is given two colors, call them C_0 and C_1. Then each pixel in the tile is specified only as a mixture of C_0 and C_1. DXT1 uses RGB565 format for the colors C_0 and C_1, and for each pixel it uses a two bit code:

code	final color
00	C_0
01	$0.666 \times C_0 + 0.333 \times C_1$
10	$0.333 \times C_0 + 0.666 \times C_1$
11	C_1

so the storage for each 4×4 tile, in bits, is:

C_0	16
C_1	16
pixels	$16 \times 2 = 32$

Each tile uses 64 bits, and there are sixteen pixels in each tile, so the *per pixel* storage is only 4 bits. If the bitmap were encoded in plain RGB565 it would need 16 bits per pixel, so the graphics memory saving is 75%!

Gameduino 2 's graphics hardware does not directly support DXT1, but by using multiple drawing passes and multiple bitmaps, it can simulate the DXT1 format.

Taking the *NightStrike* welcome screen as an example, the source image is 480×272 pixels. Hence the width in 4×4 tiles is 120×68. So the C_0 and C_1 images are both encoded in RGB565 at a resolution of 120×68:

$C_0 =$

$C_1 =$
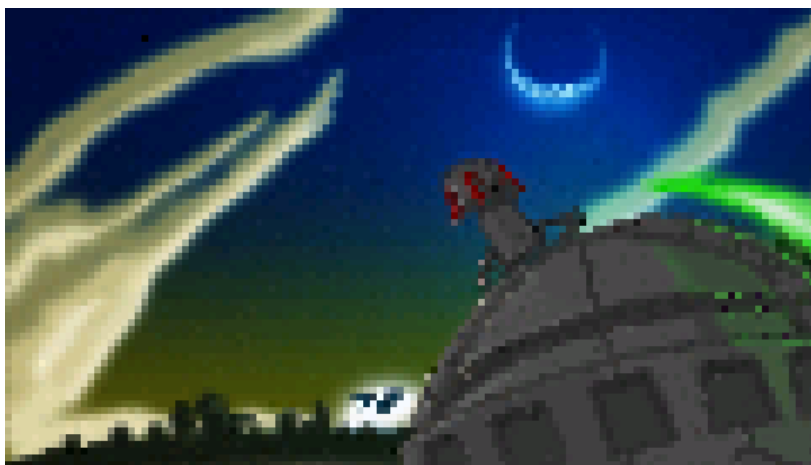
Because the hardware does not directly support 2-bit bitmaps, the two bit layer is split into two 1-bit images, using the same technique as *The missing* L2 *format* on p.176.

$b_0 =$ $b_1 =$
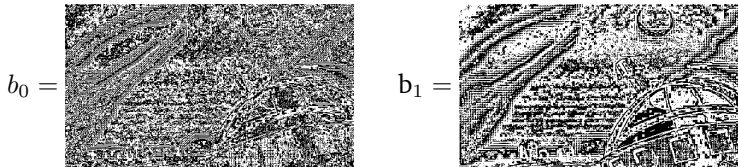
NightStrike's DXT1 background drawing function first builds the 2-bit image in the alpha buffer. It then writes the C_0 and C_1 bitmaps (zoomed $4\times$ using `cmd_scale`) into the color buffer, with the blend values from the alpha buffer.

```
void draw_dxt1(byte color_handle, byte bit_handle)
{
  GD.Begin(BITMAPS);

  GD.BlendFunc(ONE, ZERO);
  GD.ColorA(0x55);
  GD.Vertex2ii(0, 0, bit_handle, 0);

  GD.BlendFunc(ONE, ONE);
  GD.ColorA(0xaa);
  GD.Vertex2ii(0, 0, bit_handle, 1);

  GD.ColorMask(1,1,1,0);
  GD.cmd_scale(F16(4), F16(4));
  GD.cmd_setmatrix();

  GD.BlendFunc(DST_ALPHA, ZERO);
  GD.Vertex2ii(0, 0, color_handle, 1);

  GD.BlendFunc(ONE_MINUS_DST_ALPHA, ONE);
  GD.Vertex2ii(0, 0, color_handle, 0);

  GD.RestoreContext();
}
```

Chapter 16

Games and demos

16.1   Kenney

The *kenney* demo uses public domain artwork[1] by artist Kenney.nl [2] . It draws six scrolling layers, and runs smoothly at 60Hz. Most layers are drawn using bitmaps, all encoded with format ARGB4. The first layer is a vertical gradient, with carefully chosen sky-blue colors.


```
GD.cmd_gradient(0,    0, 0xa0a4f7,
                0, 272, 0xd0f4f7);
```

---

[1]http://opengameart.org/content/platformer-art-deluxe
[2]http://www.kenney.nl/

The next layer uses a single sprite loaded in CLOUD_HANDLE drawn in the random positions in state.clouds[]. The clouds all move downwards at slightly different rates, and slower-moving clouds are more transparent. As clouds reach the bottom of the screen, they wrap around to the top. The clouds' positions are subpixel coordinates, in 1/16th of a pixel. This extra precision means that as the clouds move, they don't "snap" between integer pixel coordinates.
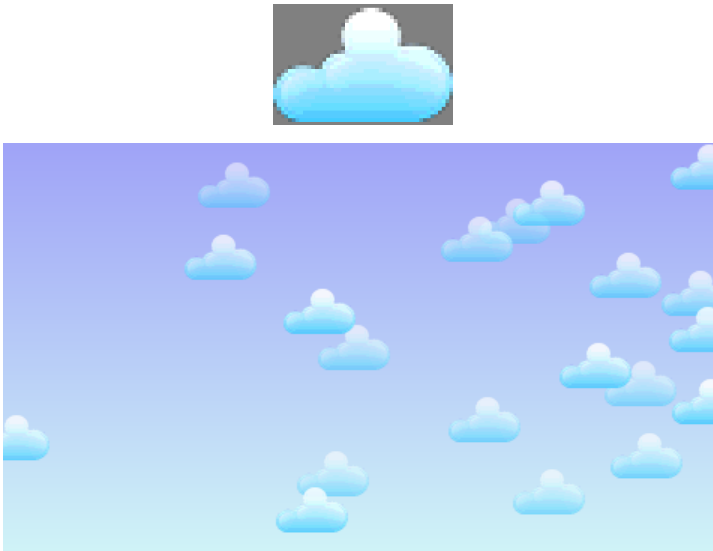

```
GD.Begin(BITMAPS);
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
GD.BitmapHandle(CLOUD_HANDLE);
GD.Cell(0);
for (int i = 0; i < 20; i++) {
  byte lum = 128 + 5 * i;
  GD.ColorA(lum);
  GD.ColorRGB(lum, lum, lum);
  GD.Vertex2f(state.clouds[i].x, state.clouds[i].y);
  state.clouds[i].y += (4 + (i >> 3));
  if (state.clouds[i].y > (16 * 272))
    state.clouds[i].y -= 16 * (272 + CLOUD_HEIGHT);
}
```

The next layer is the tile map.   The tile map
was created using the "tiled" editor (http://www.
mapeditor.org/), and imported using the asset con-
verter into character array layer1_map. The 20 tiles
are each 32×32 bitmaps, encoded as ARGB4 in handle
TILES_HANDLE. Tile code zero is the blank cell, and the
code checks for this and only draws tiles with non-zero
codes.
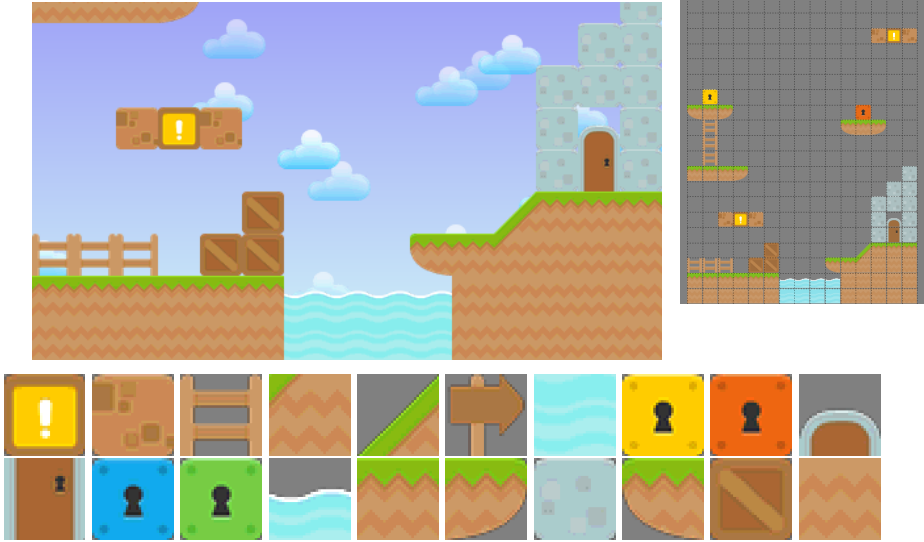
```
GD.Begin(BITMAPS);
GD.BitmapHandle(TILES_HANDLE);
const PROGMEM prog_uchar *src = layer1_map + (y >> 5) * 15;
byte yo = y & 31;
for (byte j = 0; j < 10; j++)
  for (byte i = 0; i < 15; i++) {
    byte t = pgm_read_byte_near(src++);
    if (t != 0) {
      GD.Cell(t - 1);
      GD.Vertex2f(16 * 32 * i, 16 * ((32 * j) - yo));
    }
  }
```

The foreground characters are `ARGB4` bitmaps, rotated at random and following gentle sinusoidal paths. Their coordinates are in `state.p`.

The trails of colored "bubbles" following each character are transparent `POINT`s, handled in the code by three `state.trail` objects.
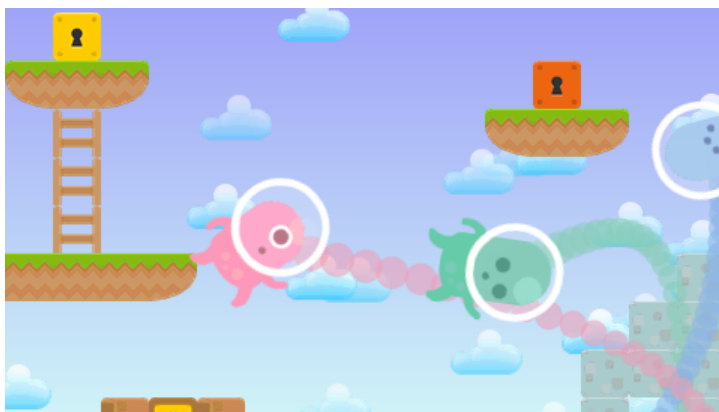

```
GD.BitmapHandle(PLAYER1_HANDLE);
GD.Begin(BITMAPS);
for (int i = 0; i < 3; i++) {
  rotate_player(a + i * 0x7000);
  GD.Cell(i);
  GD.Vertex2f(state.p[i].x - (16 * PLAYER1_SIZE / 2),
              state.p[i].y - (16 * PLAYER1_SIZE / 2));
}
```

The "sunburst" graphic is drawn between the sky and cloud layers. It uses a series of stencil operations to create the ray objects. The function `burst()` draws a complete sunburst, and `sunrise()` draws both the orange and yellow `burst()`s, and animates their bouncing expansion.

The final layer is the shower of hearts falling down the screen. These are handled like the "cloud" layer. Array `state.hearts[]` tracks their position, and they move downwards, wrapping around from bottom to top.
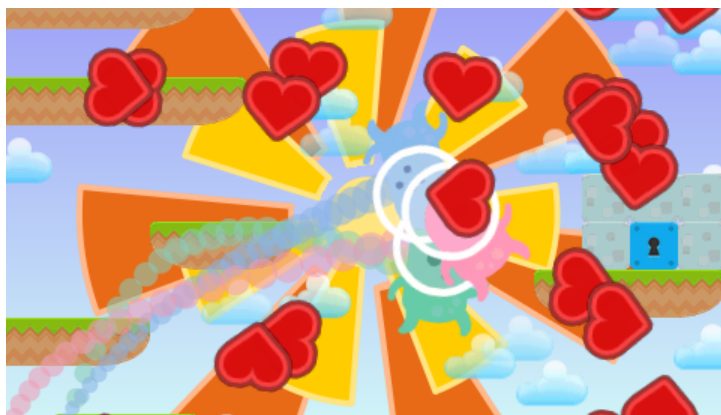
16.2   NightStrike

The *NightStrike* game uses public domain artwork [3] by artist MindChamber. It uses the touch screen to track dozens of independent objects, and plays smoothly at 60Hz. The game uses some special techniques to fit its graphics into the available 256 KBytes of memory.

The game has five levels, and each level is a separate asset file, containing all the background and foreground graphics for the level, as well as sound effects. The backgrounds for each level are 480×272 images, encoded using DXT1-like compression (see *DXT1* on p.181) so they only use 65 Kbytes of video memory.

---

[3] http://opengameart.org/content/nightstrike-png-assets
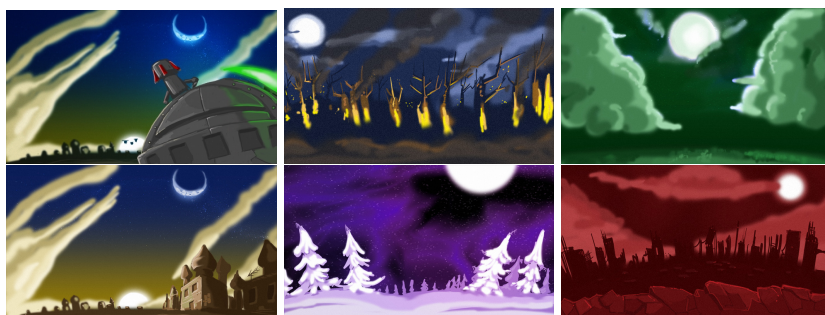
The sprites in the game have smooth transparency, so most are encoded in `ARGB4` format.
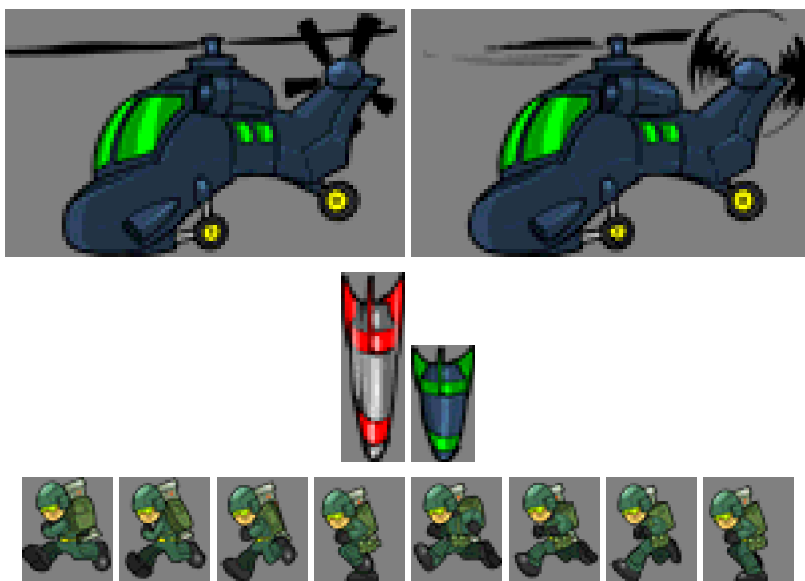
The explosions in the game animate quickly and have bright colors, so can use `ARGB2` format to save memory (see *8-bit formats* on p.180).
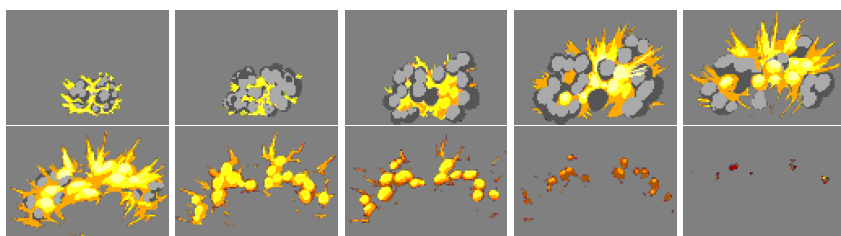
The player's "defensor" is monochrome, so it is drawn using two `L4` bitmaps (see *Separated mattes* on p.178).
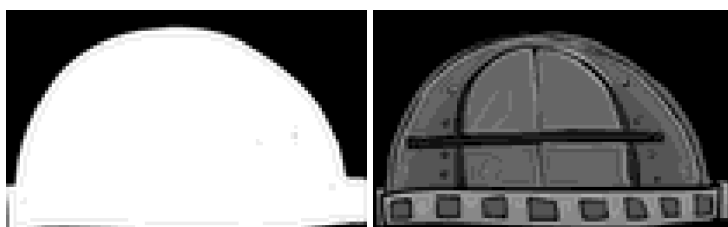
As well as using animated bitmaps for explosions, the game draws explod-ing sparks using wide lines (see *Lines* on p.48). Because they use LINES, the sparks require no graphics memory
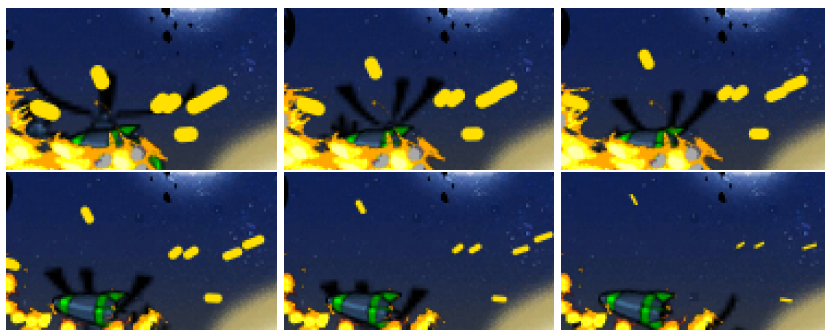
In game text uses a small 12×19 pixel font, converted from an original True-Type font. This font's bitmaps are in L4 format, to preserve its smooth edges.
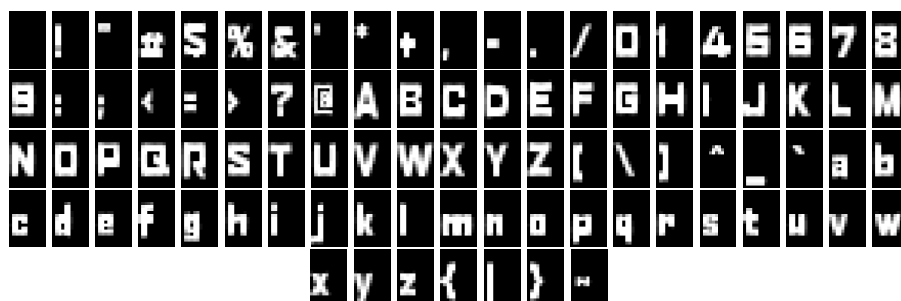
The game code uses a class for each kind of visble object in the game. These class are: `Fires`, `Explosions`, `Sparks`, `SoldierObject`, `MissileObject`, `Rewards`, `BaseObject`, and `HeliObject`. Each object has a `draw()` method to draw the object on the screen, and an `update()` method to animate, move and compute any collisions. Object collision testing uses a simple overlapping box scheme.

NightStrike uses 20 Kbytes of the Arduino's 32 Kbytes of flash, and the game loop runs in about 7ms on a standard 16MHz Arduino.

16.3   Invaders
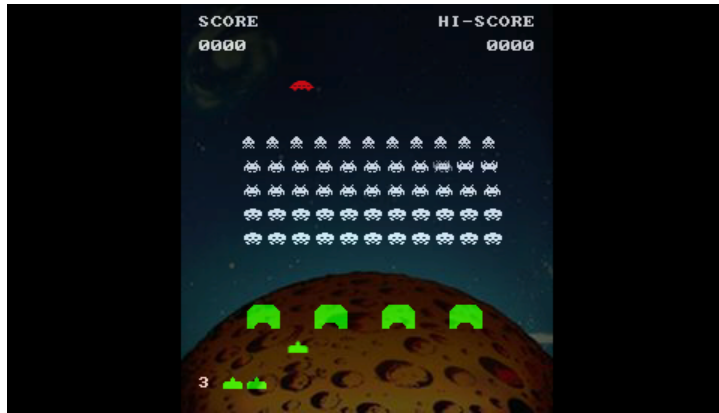
Invaders is a port of the original Gameduino game [4]. But to make it more authentic, it includes the original arcade cabinet artwork. The original '70s arcade cabinet only had a monochrome CRT screen, but to jazz it up the designers used a cunning mirror arrangement to project a suitable space picture behind the graphics. If you look closely you can see that the pixels are slightly transparent because of this. Their other trick was to put a colored overlay on the CRT screen with a green band at the bottom and a red one at the top. In the original game you can see the missiles and bombs change color as they pass through the colored zones.

Gameduino 2 *invaders* loads the background image as a 248×272 JPEG. Because it only uses 6K it fits easily in the Arduino's flash memory. Encoding the full-brightness image then drawing a darker version in the game helps to conceal some of the JPEG compression artefacts in the sky gradient.

The game's pixel action is drawn into the alpha buffer. To make it visible, the 28×32 overlay bitmap is drawn zoomed 8X so that its size is 248×256, with `BlendFunc (DST_ALPHA, ONE_MINUS_DST_ALPHA)`. This paints the game's pixels, using the color from the overlay.

---

[4] http://artlum.com/gameduino/gameduino.html
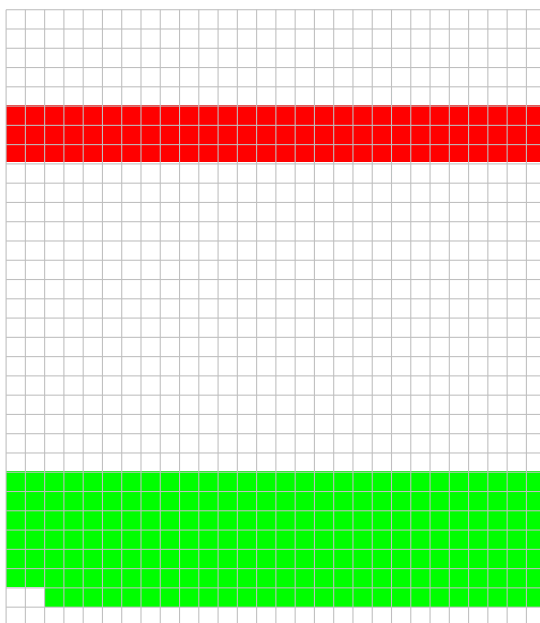
Appendix A

The asset converter

The latest Gameduino 2 asset converter is available at
`http://gameduino.com/code`
Detailed documentation is available online.

Index

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

[SparkFun Electronics](#):

  [SPX-15091](#)