Gameduino 2
入门指南，编程参考与设计实例

作者：James Bowman (美)
jamesb@excamera.com
Excamera Labs

译者：戴晓天[1]，徐欣宇[2]

(Version 1.0)

[1]云飞机器人实验室，http://www.yfworld.com，翻译章节 1-2, 13-16，全文校稿
[2]英国谢菲尔德大学，邮箱：xinyu.xu@live.cn，翻译章节 3 - 12

首次出版时间: 2013年10月

Bowman, James.
The Gameduino 2 Tutorial, Reference and Cookbook / James Bowman. – 1st Excamera Labs ed.
200 p.
Includes illustrations, bibliographical references and index.
ISBN 978-1492888628
1. Microcontrollers – Programming I. Title

目录

Part I

入门教程

Chapter 1

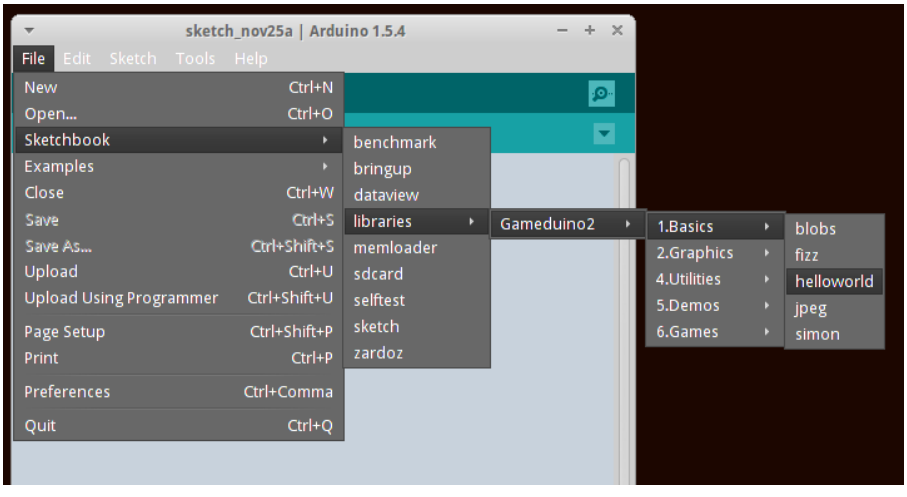
连接USB，开始你的创作！

1. 从 http://gameduino.com/code 下载 Gameduino 2 的程序库 Gameduino2.zip 并且将其安装至Arduino IDE中。 如果不知道如何安装Arduino第三方程序库，可以在 http://arduino.cc/en/Guide/Libraries 找到详细步骤。

2. 将 Gameduino 2 连接至Arduino的插槽中，连接时确认所有的引脚都正确对齐。

3. 将Arduino插入电脑USB端口为其通电。因为当前还没有程序载入到Arduino中，此时 Gameduino 2 的屏幕上不会有任何内容显示。

4. 启动Arduino IDE并且载入一个Gameduino 2的示例程序，如
   `File` ▷ `Sketchbook` ▷ `libraries` ▷ `Gameduino2` ▷ `1.Basics` ▷ `helloworld`



5. 点击下载按钮将程序编译并载入到Arduino中，等待一段时间后，Gameduino 2 将会启动并开始绘制载入的程序。

6. 很多示例程序需要从microSD中读取数据。为了能正常运行这些程序，需要准备一张格式化后的microSD卡，并将 `Gameduino2sd.zip` 压缩包中的文件拷入到SD卡中 (该文件可从 `http://gameduino.com/code` 上下载)。

7. 试着运行其他的例程，劲情享受!

Chapter 2

快速入门

本章介绍了 Gameduino 2 的基本编程方法 - 从 “hello world” 到创建一个简单的触屏游戏。

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();
}
```

以上示例代码中，函数 loop() 首先将屏幕背景设置为深绿色，之后将一段文字显示在屏幕的正中央。这个代码将以每秒钟 60 次的速度连续执行，但由于每次写入的内容相同，屏幕上不会看出任何变化。

ClearColorRGB 是一个绘图指令：它设置了屏幕清屏时所使用的背景色。这里所用的颜色是深绿色 0x103000。 Gameduino 2 和HTML一样，使用三个

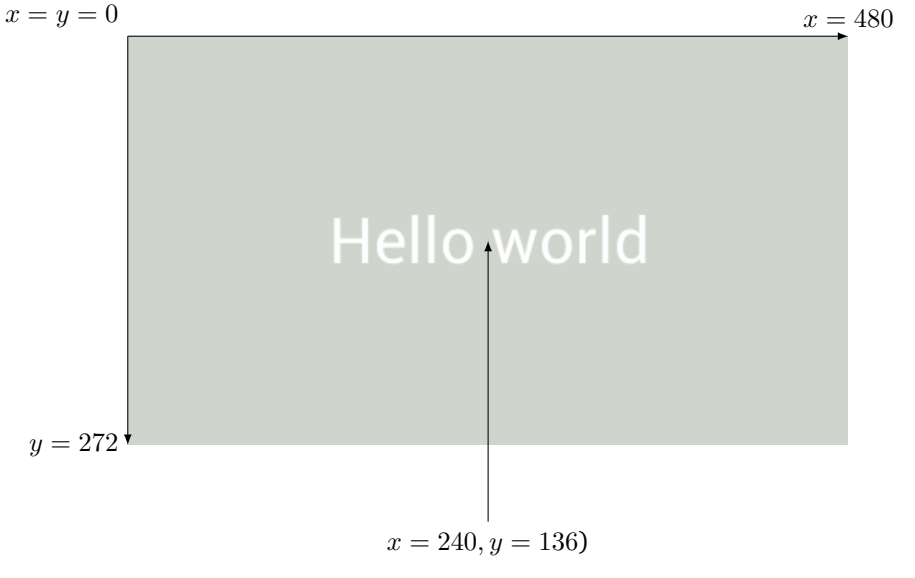
16 进制数来表示颜色。

Clear 是另外一个绘图指令，它将屏幕清除为ClearColorRGB()中所设置的颜色。所有的绘图都是从清屏指令开始的。

cmd_text 是一个高级GPU绘图指令。该指令在屏幕指定位置用特定字体绘制字符串。在这个例子中，文字被绘制在屏幕的正中心，(x, y) 坐标 $(240, 136)$ 处。OPT_CENTER 参数的作用是指明绘制文本时提供的坐标为中心坐标，所以文本绘制时的中心将被设置为 $(240, 136)$。参数中的 31 是文本的字号，31 是内置字体中能设置的最大字号。

最后 GD.swap() 函数告知图形系统图形绘制已经完成，并将绘制完成后的图形内容显示到显示屏上。在Gameduino中有两份图像的拷贝，一份是在屏幕上实际显示出来的图形，一份是在缓存中用户正在编辑的图形。调用 GD.swap() 函数将把缓存中的内容同步到屏幕上。这种交换机制 (又称双缓冲) 可以使显示的内容更新时更加平滑，只有程序调用 GD.swap() 时才会进行一次真正的绘制。

Gameduino 2 的屏幕分辨率为宽 480 像素，高 272 像素。屏幕的左上角坐标是 $(0, 0)$，屏幕的右下角坐标是 $(479, 271)$，屏幕中心坐标点在 $(240, 136)$。



2.2 点与圆

那么如何使用 Gameduino 2 绘制图形呢？图形绘制有两个指令： Begin 和 Vertex2ii。其中 Begin 告知图形系统将要绘制的图形类型，而 Vertex2ii 将该图形在指定位置绘制出来。以下代码实现了在 $(220, 100)$ 和 $(260, 170)$ 两处分别绘制两个单位像素大小的坐标点：

```
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();
```

Begin(POINTS) 函数告知GPU准备开始进行点绘制。之后的 Vertex2ii 函数调用在指定的屏幕坐标 (x, y) 处绘制一个新点。注意如果不先调用准备绘制函数 Begin，那么硬件无从得知将要绘制的图形类型，从而导致其忽略接下来所有的 Vertex2ii 的调用。

GD图形库还包含 PointSize 调用，该函数用于设置绘制点的大小。此函数仅有一个参数：绘制点的直径，单位为 1/16 像素。硬件系统使用子像素机制 (*subpixel*) 让用户对绘制物体的尺寸有更好的控制。子像素相比以像素作为单位，可以显著提高显示的精度和效果。当你希望使用像素表示长度或直径的时候，最好在代码中显式将对应的数值乘以 16 。

这里，在调用点绘制函数之前增加了一个 PointSize 调用，绘制了一个半径为 30 像素（即直径 60 像素）的圆点。


```
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();
```

实际上，硬件系统总是以和绘制圆相同的方式来绘制点(POINTS)。但当绘制圆的半径很小时，此时绘制的圆看上去等效于一个像素点。所以在Gameduino中，点与圆如同手足不可分割。

2.3 色彩与透明度

在绘制vertex图形之前，还可以通过 ColorRGB 函数改变绘制的颜色：


```
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 和 PointSize 都属于设置图形状态的功能函数，这些图形状态好比是图形硬件系统中的变量一样。当这些状态被程序更改之后，将会影响到之后的绘图操作。为了让生活美好一点，所有的图形状态在每一帧的开始处被恢复成了默认值。你可能想到了，对于 ColorRGB 来说，默认的颜色就是是白色的，这也是为什么之前的 “Hello world” 会是白色的。而默认的 PointSize 是 8，也就是半个像素的直径大小。

你也可以如同改变颜色一样自如的改变绘图的透明度。透明度是通过 alpha 通道控制的，有效设置范围为 $0 \sim 255$。 其中 0 表示完全透明，而默

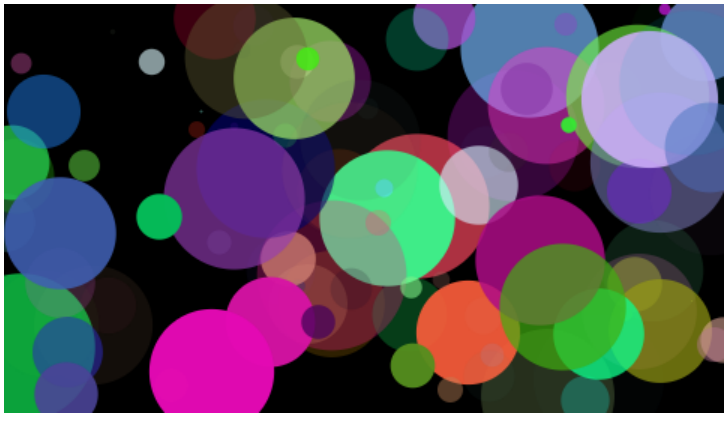
认值 255 表示完全不透明。以下是利用 ColorA 函数将透明度设置为 128，也就是 50% 的透明度，得到的效果：


```
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();
```

颜色修改函数 ColorRGB 和 ClearColorRGB 同样接受 (R, G, B) 分立值作为参数。所以 ColorRGB(0x0080ff) 也可以写为 ColorRGB(0, 128, 255)：

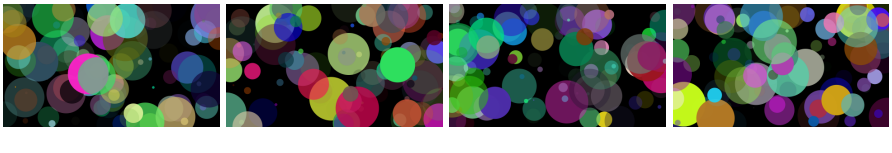
```
GD.ColorRGB(0, 128, 255); // teal
GD.Vertex2ii(260, 170);
```

2.4 设计示例: 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();
}
```

"fizz" 程序在每一帧中绘制 100 个圆，每一个圆都具有随机的大小、颜色和透明度。虽然每一帧的执行时间仅有 1/60 秒，但因为人眼存在视觉暂留，一眼看上去就好像无数五彩斑斓的圆盘一样。



2.5 播放音符

Gameduino 2 有两种方式播放声音，一种是使用内置的音频采样文件，一种是播放用户自己的采样文件。对于前者，只要调用 GD.play() 函数，并指明使用的乐器及MIDI音符编号就可以播放对应音符：

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

GD.play() 函数是非阻塞执行的：即使之前的音乐没有播放完，下一个音乐也会立刻开始播放。为了防止这种情况的出现，这里调用了 delay() 延时1秒来等待之前的音符播放完毕。Gameduino 2 可用的内建乐器列表如下：HARP, XYLOPHONE, TUBA, GLOCKENSPIEL, ORGAN, TRUMPET, PIANO, CHIMES, MUSICBOX 和 BELL。虽然这些内置采样的音质并不高[1]，但这里只是将它们作为简易手段来测试音频功能。确定了播放乐器之后，还需要确定MIDI编号。在 Gameduino 2 中，每个乐器可以播放的MIDI编号范围是 $21 \sim 108$。

还有一些可以连续播放的声音：SQUAREWAVE，SINEWAVE，SAWTOOTH 和 TRIANGLE。这些声音特别适合用于创造独特的声音效果，产生复古的音效感。不同于之前的乐器音效，这些声音如果不主动中断，将会无限期地播放下去。除了这些乐器音效以外，这里还有一小部分打击乐的采样音效。这些音效不使用MIDI编号参数，所以在程序编写时直接忽略，如 GD.play(NOTCH)。这部分可用的音效列表如下：

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

注意负责音频播放的硬件同时只能播放一个音效，播放任何其他的声音都将中断上一个声音的播放。

如果需要主动停止声音播放，可以使用函数 GD.play(SILENCE)。

[1] 其中PIANO 可能是最糟糕的了，其次是TUBA：听起来都不像一个常规的乐器。

2.6 触摸标签

屏幕上的每个可见像素除了有对应的颜色之外，还有一个不可见的触摸标签值。这个标签值用来判断触摸发生的位置。以下代码演示了如何将两个彩色圆盘的标签值分别设置成为 100 和 101：

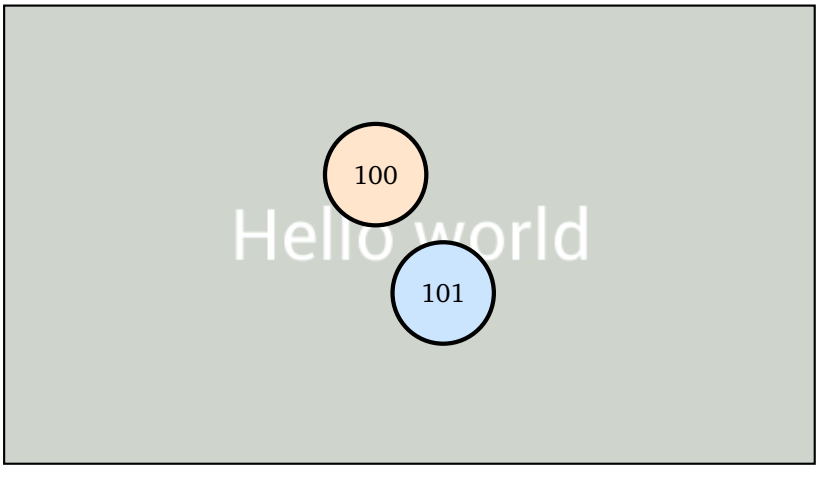
```
GD.ClearColorRGB(0x103000);
GD.Clear();
GD.cmd_text(240, 136, 31, OPT_CENTER, "Hello world");
GD.PointSize(16 * 30);
GD.Begin(POINTS);
GD.ColorRGB(0xff8000);
GD.Tag(100);
GD.Vertex2ii(220, 100);
GD.ColorRGB(0x0080ff);
GD.Tag(101);
GD.Vertex2ii(260, 170);
GD.swap();
```

这个代码会生成和之前一样的图形：

但是这个程序不同之处在于，一旦系统检测到在这两个圆上发生了触摸动作，就会将对应的标签值（这里是 100 或 101）记录保存下来。之后再使用一个循环结构不断读取当前的触摸标签值（get_inputs 函数会将最新的触摸标签值更新至 GD.inputs），并将其输出到屏幕上：

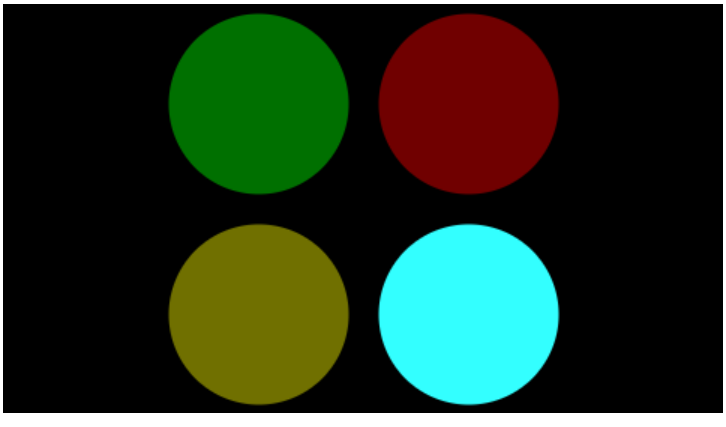
```
for (;;) {
  GD.get_inputs();
  Serial.println(GD.inputs.tag);
}
```

这样一来，每当你按下其中一个圆盘的时候，从串口输出中就会返回 100 或 101 的代码。触摸标签机制提供了一种非常实用的交互方法，如果没有这个功能，那么代码必须检测每一次触摸的 (x, y) 坐标，并依次判断该坐标是否落在对应的按钮半径范围内。而通过使用触摸标签，代码为每个对象分配一个 Tag 作为标签值，随后硬件自动进行像素判断并记录触摸对象的标签值。为了实现这个功能，设备还需要持续追踪每个像素的 Tag 值。这被称为“标签缓存”——下面是标签缓存的一个例子：



每当屏幕被点击，设备都将查找标签缓存中对应像素标签值，并将其记录在 GD.inputs.tag 中。

2.7 游戏设计: Simon



这是一个诞生于 1978 年的记忆游戏“Simon”。这个游戏的规则是：系统按照随机顺序点亮屏幕中的四个指示灯之一，而玩家必须按照正确的顺序重复出来。如果你全部答对，系统就会在原序列中再额外增加一个步骤，如此重复直到你出现失误为止。

以下代码首先定义了游戏中使用到的主要颜色，之后将利用这些颜色绘制游戏界面：

```
#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
```

游戏的主界面是四个大圆盘，每一个对应一个按钮。按钮的颜色根据 pressed 参数决定是否要高亮显示。在绘制按钮之前调用了 Tag 函数设置标签缓冲，这样在该按钮的任意位置点击都会在 GD.inputs.tag 中返回对应数值。

```
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();
}
```

函数 play() 根据被点击的按钮播放不同的音效，同时点亮相应的圆盘半秒（即 30 帧）。

```
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() 是一个用户输入程序。它的作用是绘制游戏界面，同时每次绘制都将查看 GD.input.tag 以确定是否有按钮被点击。一旦有按钮被点击，它将调用 play() 来播放音效与处理高亮，最后返回按钮的对应数值。

```
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;
}
```

loop() 函数实现了完整的游戏循环过程。程序每次都将在原按钮序列后添加一个新的随机按钮，演示这一新序列，并要求玩家按顺序重复按钮序列。如果玩家重复点击的顺序完全正确，游戏会一直继续下去；如果玩

家的点击出现任何错误，游戏将发出表示"失败"的音效并退出，随后下一次 loop() 函数将重新启动游戏。

```
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

位图

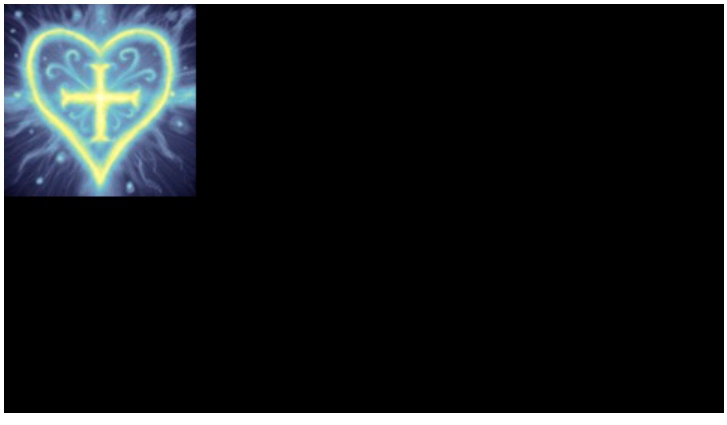
Gameduino 2 的GPU模块对绘制位图有硬件级别的支持，Gameduino 2 中支持的位图种类有：

- 任意尺寸，从 1×1 像素到 512×512
- 可在其他位图图像上进行重复叠加
- 重新着色并在绘制时增加透明度
- 在x和y轴方向进行无限重复绘制（又称 *tiling*）
- 旋转，拉伸和收缩

位图图像的数据——代表位图各像素点信息的字节数据——存储在GPU的主存储器中。这一主存储器的大小是 256 KB。

从某种程度上说，位图是过去游戏机硬件所使用的图元（*sprites*）的传承。在一些代码中仍可看到 *sprite* 这个词——它通常只是表示在到处移动的位图。

3.1 载入一张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();
}
```

以上代码中， cmd_loadimage 的作用是通知GPU将会有一张JPEG载入到内存地址 0 处。 load 从microSD卡中读取文件 healsky3.jpg[1] 并直接将其提供给GPU，GPU将它作为一张位图存储到显存中。 Begin 调用使用 BITMAPS 参数，这样随后每次调用 Vertex2ii 都将以给定的坐标 (x, y) 作为左上顶角来绘制载入的图像。

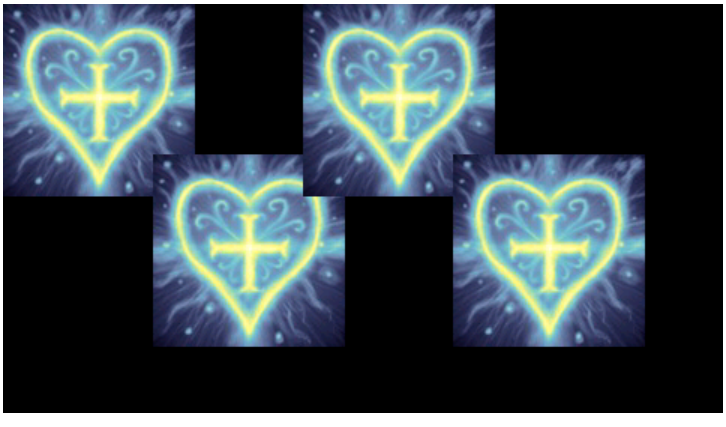
[1]Artwork by J. W. Bjerk (eleazzaar) – www.jwbjerk.com/art

3.2 位图尺寸

之前我们知道了每次调用 Vertex2ii 都会绘制一幅位图，例如这段代码

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

将会生成下图：



原始JPEG文件的尺寸为 128×128，因此每次调用 Vertex2ii 都将在大小为 128×128 的像素区域中绘制这一位图图像。不过你可以通过 BitmapSize 来调节绘制位图的区域尺寸。例如：

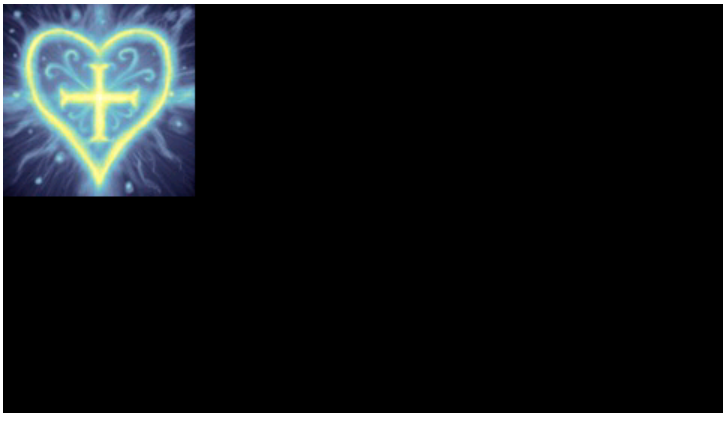
```
GD.Clear();
GD.Begin(BITMAPS);
GD.BitmapSize(NEAREST, BORDER, BORDER, 64, 64);
GD.Vertex2ii(0, 0);
GD.swap();
```

改变位图绘制尺寸后，只有 64×64 的区域被绘制出来：

如果我们增大位图绘制区域到比原图更大的尺寸，如 480×272：

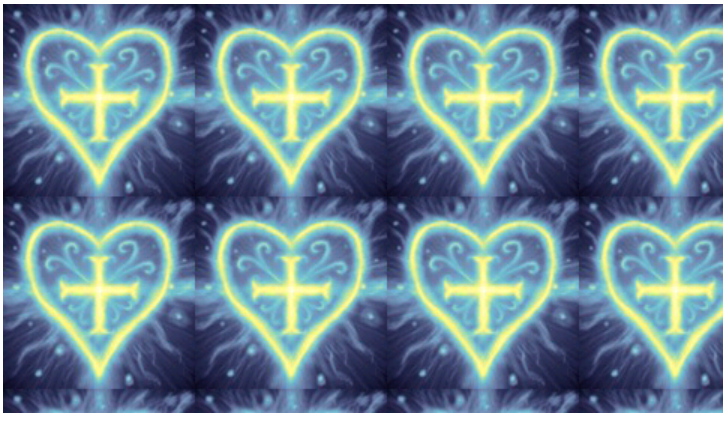
```
GD.Clear();
GD.Begin(BITMAPS);
GD.BitmapSize(NEAREST, BORDER, BORDER, 480, 272);
GD.Vertex2ii(0, 0);
GD.swap();
```

却并没有得到比原图 128×128 更大的尺寸：

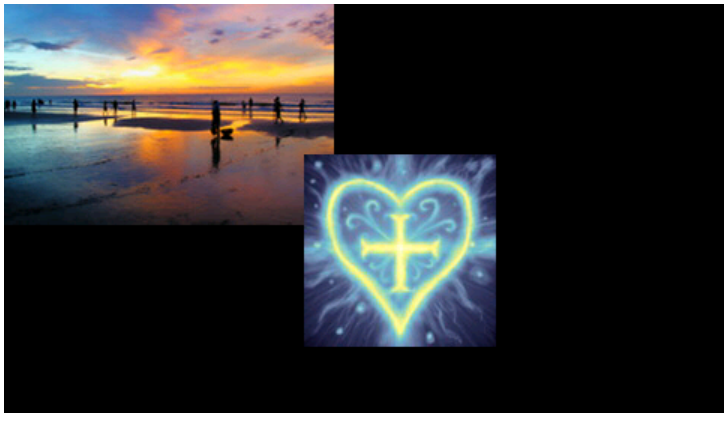


为什么会这样呢？ 其实，GPU的确绘制了一幅 480×272 尺寸的图像， 但是因为原图的尺寸只有 128×128 ， 多余的空间就会被用透明的黑色像素填满。 但如果将参数 BORDER 改为 REPEAT ， GPU就会在绘制区域内的x和y轴方向上无限地重复绘制原图像。

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



3.3 位图句柄



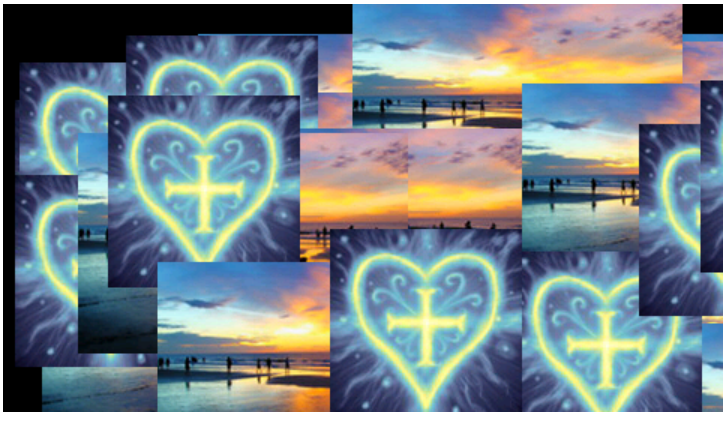
```
void setup()
{
  GD.begin();

  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();
}
```

以上这段代码的功能是在显存中存放了两张JPEG图片——其中一张日出图片存放在了在地址 0 处，还有一张名为 healsky3 的图片。值得注意的是，其中第二个 load_image() 的调用中将 −1 作为地址参数，这一特殊数值的含义是告知载入程序将这张JPEG图片紧跟在当前显存的最后一张图片之后。通过这种方式可以很方便的存储一连串图片，而无需计算每张图片的具体存储位置。

通过在将图片载入前设定相应的位图句柄，设备能够时刻追踪到这两张图片的各项参数——大小、格式、存储地址—— 而在绘图代码中也可以通过这两个句柄对这两张图片进行选择。

在绘制位图（ BITMAPS ）时， Vertex2ii 还有第 3 个参数。这个参数用于指定绘制位图的相应句柄，其缺省值为 0 。以下代码在位图句柄 0 和 1 中随机选择，并连续绘制了 100 张位图。最终的结果呈现出一幅随机混杂的图像，视觉上和 设计示例: fizz 相似。

```
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();
```



硬件系统提供 16 个可用的位图句柄供用户使用，其有效范围是 0 ∼ 15。用户可以在绘制过程中重新分配位图句柄，因而可供应用程序使用的位图不局限于 16 张。事实上，大部分应用程序不会用到 16 张以上的图片，因此各图片通常在 setup() 中就分配好对应的句柄。

3.4 位图的像素格式

在理想情况下，所有位图的色彩都应该实现百分之百的还原。不过受限于GPU和Arduino的内存，用户只能在色彩还原度和存储空间上寻找一个平衡点。对此，Gameduino 2 提供了多种可供使用的图片格式：

L8

每像素 8 位数据，最高质量的单色图片格式。

L4

每像素 4 位数据，适用于单色图标或字体。

L1

每像素 1 位数据，用于实现复古的图形风格；同样也适用于分层和蒙版效果。

RGB565

每像素 16 位数据：其中各有 5 位用于存储红色和蓝色信息，6 位用于绿色信息。适用于大多数没有透明度设置的照片或图片。

ARGB1555

每像素 16 位数据：其中各有 5 位用于存储红色、绿色和蓝色信息，另有 1 位记录透明度信息。这个仅有 1 位的透明度通道只可实现简单的开启或关闭设置。

ARGB4

每像素 16 位数据：各有 4 位用于存储红色、绿色、蓝色和透明度信息。适用于存储需要有光滑透明边缘的图片，例如：彩色图标和图元。

RGB332

各有 2 位用于存储红色和蓝色信息，3 位用于绿色信息。有时也用于图像显示和图标。

ARGB2

各有 2 位用于存储红色、绿色、蓝色和透明度信息，不太适用于图片的显示，但可用于复古风格的图元或者色彩度较低的图标中。

3.5 位图着色

和其它绘图指令一样， BITMAPS 也存在当前颜色的设定。该颜色会与原图像的颜色混合，呈现出一种透过彩色玻璃看它的感觉。

```
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();
```

如果当前 RGB 颜色设定为白色（0xffffff)，则原图像色彩保持不变。而如果将颜色设定为黑色（0x000000），则原图像的所有像素都将被填涂为黑色。

3.6 转化图形

将一个图片资源转换为 Gameduino 2 可用的位图格式的步骤并不简单。对此， Gameduino 2 的内置工具中包含一个资源转换器（asset converter），该工具可以读取图片源文件并将其转换为 Gameduino 2 可用的位图格式。

资源转换器的输出为一个名为 x_assets.h 的头文件，其中 x 是用户转换图片的文件名。在这个头文件中，定义了一个宏函数 LOAD_ASSETS()，它的作用是将转换后的图片载入到 Gameduino 2 的显存中。该文件的使用方法是：首先在程序开头包含这个头文件，然后在调用 GD.begin() 后，调用函数 LOAD_ASSETS()：

```
#include "walk_assets.h"

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

在 LOAD_ASSETS() 完成后，所有的位图都将加载到显存中，并且位图句柄也将设定完毕以便用其实现位图操作。这些位图句柄同样在头文件中说明，之后的代码可以直接使用这些句柄，如下所示：

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

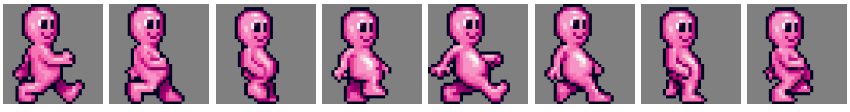
使用资源转换器和之前直接从JPEG文件中载入图片相比有两个优点：

第一，JPEG图片只能被加载为 RGB565 格式的位图，这是因为在JPEG图片中没有透明度信息，所以这是唯一支持的格式；

第二，资源转换器对图片进行无损（ *losslessly* ）压缩[2]，而JPEG使用的是有损压缩。对于一般的摄影图像差别虽然不大，但是对于精细绘制的游戏插图、图标和字体而言，有损和无损压缩的差别是相当明显的。

[2] 压缩方案为 zlib INFLATE （ http://www.zlib.net/ ），即gzip和Zip所使用的压缩方案。GPU为zlib INFLATE提供硬件支持。

3.7 位图单元

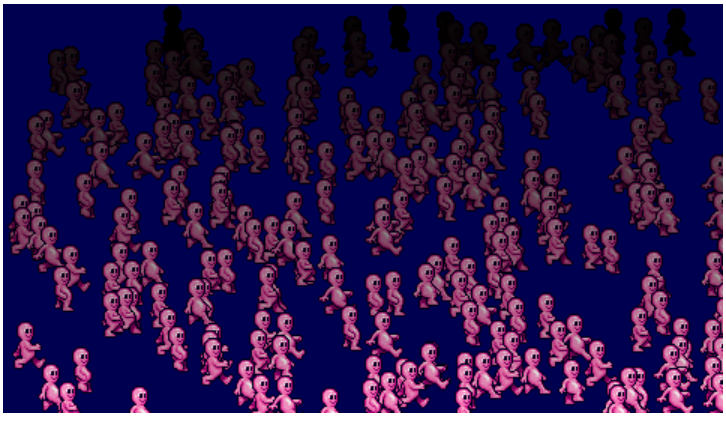


这 8 张 32×32 ARGB1555 格式的位图图片循环构成一组走路的动画。由于所有图片都采用乐同样的尺寸和格式，他们可以共用同一个位图句柄。绘制命令会对具体绘制哪一个位图单元（*cell*）进行选择，这是在调用 Vertex2ii 绘制位图时的第 4 个参数。下面这段代码演示了如何在屏幕上顺序显示这 8 张动画图片：

```
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);
```

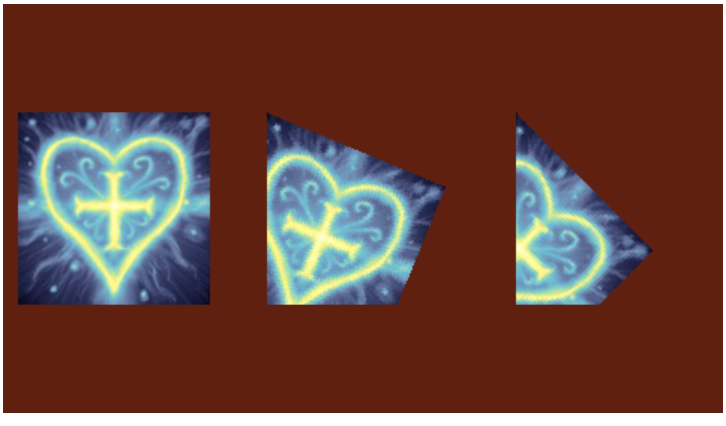
每个位图句柄最多能够控制 128 张位图单元，而这些位图单元在显存中是连续存放的。位图单元对于显示动画是十分实用的，用户只需将动画每一帧的图片按顺序存储为一个位图句柄控制的位图单元，就可以通过改变位图单元编号来控制相应的动画显示。

使用这 8 帧动画序列，例程 *walk* 在屏幕上创建了 256 个走路的循环动画。每组动画都有对应的计数器来控制它的 x 坐标以及对应动画的帧编号。另外，动画图片所用的颜色从屏幕顶端的黑色（0x000000）逐渐变化成屏幕底端的白色，从而产生了不同远近距离的效果。



```
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();
}
```

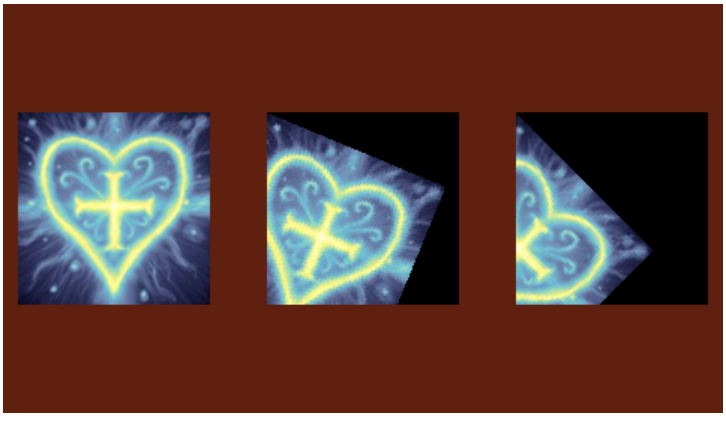
3.8 旋转，拉伸和收缩



```
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();
```

位图的旋转和拉伸是由位图变换矩阵（*bitmap transform matrix*）所控制的。这一矩阵属于图像属性的一部分，它控制着位图像素在屏幕上的映射方式。值得欣慰的是，变换矩阵的数学计算大部分由设备硬件处理，因此用户可以轻松的实现像旋转和缩放这样的高级操作。

由于 cmd_rotate 会累加操作，所以上例中将一幅位图分别以顺时针旋转 0°、22.5° 和 45° 绘制了 3 次。可以注意到这里图像只显示了一部分，这是因为旋转指令是以图片的左上角为中心进行旋转的。为了更好地观察这一现象，以下使用 (SRC_ALPHA, ZERO) 参数调用 BlendFunc 函数将透明度设置取消：



Vertex2ii 每次绘制 128×128 像素的图像，也就是所绘制图片的尺寸。由于位图绘制是由变换矩阵决定的，当变换矩阵应用旋转变换后，图片也将随之旋转。

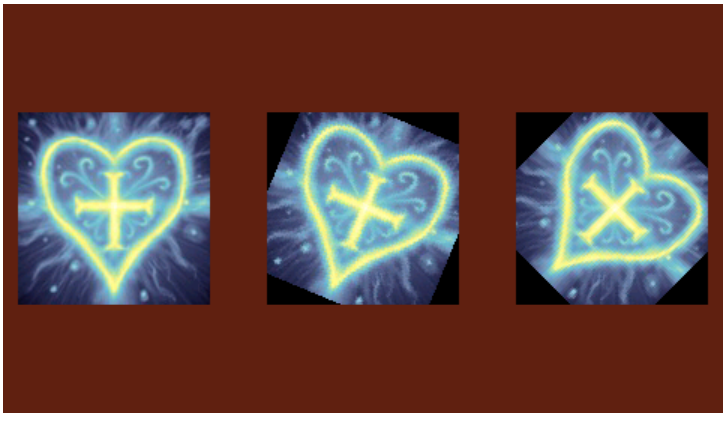
旋转发生的中心是位图图像的左上角，即例中图像 $(0,0)$ 的坐标位置。但一般对于旋转操作来说，通常将图片中心作为旋转中心，即例中图像 $(64,64)$ 的坐标位置。这个效果要分以下步骤实现：

1. 对图像进行平移变换（*translate*），使原坐标 $(64,64)$ 移至 $(0,0)$ 处；
2. 调用 cmd_rotate，以当前 $(0,0)$ 为中心旋转图像；
3. 对图像进行平移变换回原位置，即将现坐标 $(0,0)$ 移回 $(64,64)$ 处。

下面这个函数 rotate_64_64 演示了上述三个步骤：

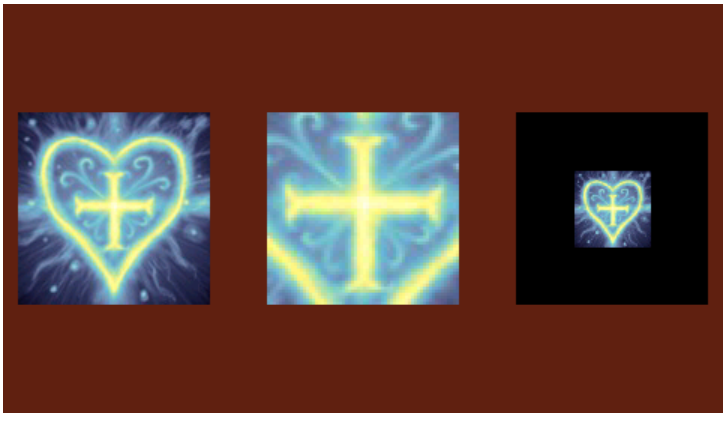
```
// Apply a rotation around pixel (64, 64)
static void rotate_64_64(uint16_t a)
{
    GD.cmd_translate(F16(64),F16(64));
    GD.cmd_rotate(a);
    GD.cmd_translate(F16(-64), F16(-64));
}
```

上例调用 cmd_translate 命令使图像在 x 和 y 方向上各平移了 64 像素。cmd_translate 使用 16.16 定点浮点数作为其参数，F16() 宏函数将整数像素值转换为子像素坐标。



```
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();
```

cmd_scale 指令可以用来缩放图像，通过增减比例实现拉伸或收缩图像。下面例子中 scale_64_64() 函数以图像 $(64, 64)$ 像素位置作为中心对图像进行缩放。其中第一张图片是没有缩放的原图片；第二张图片以 2.0 作为缩放因子，将图片尺寸放大到两倍；第三张图片以 0.4 作为缩放因子，对图片进行收缩。

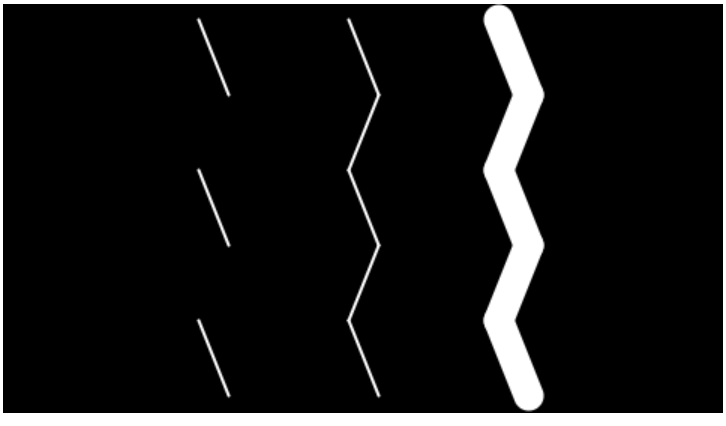


```
GD.ClearColorRGB(0x602010);
GD.Clear();
GD.Begin(BITMAPS);

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

更多的图形元素

4.1 直线



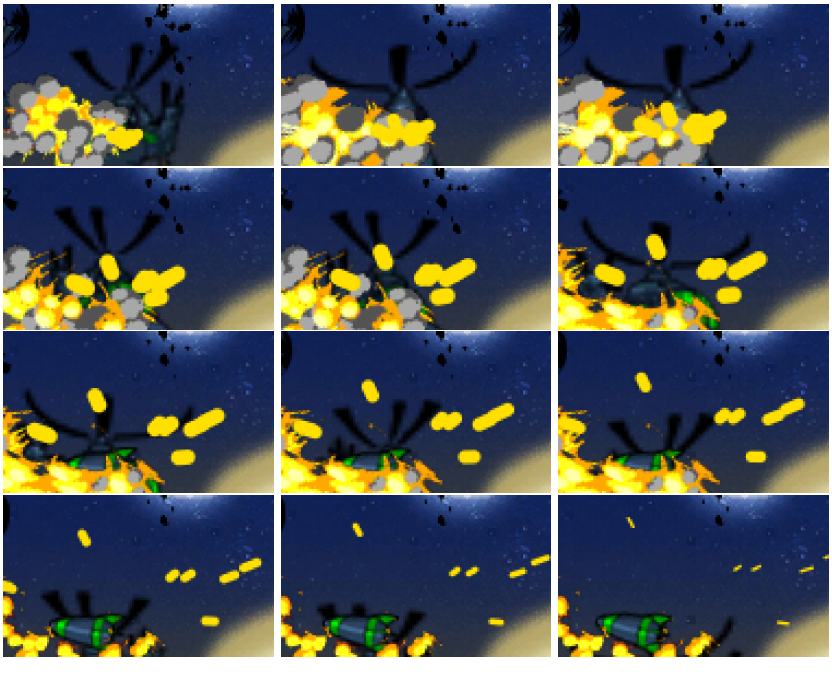
绘制直线的功能可以通过 Begin(LINES) 或者 Begin(LINE_STRIP)函数实现。 LINES 只成对连接端点绘制线段，而 LINE_STRIP 将所有端点按顺序首尾相接绘制线条。

```
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();
}
```

游戏NightStrike使用 LINES 绘制爆炸时“火花四溅”的效果。每个火花都是一个包含当前位置 (x,y)、速度 (xv,yv) 和生命周期的对象。当爆炸发生时，每一个火花以爆炸中心作为起始位置，并被赋予一个随机速度 (xv,yv)。而后每一个火花都会沿着自己的速度方向发散，绘制代码如下所示：

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

这段代码绘制了一条以火花当前位置坐标点 (x, y) 和下一位置坐标点作为两端的线段。火花的大小（LineWidth）使用正弦公式计算得出，其开始时会由细变粗，之后再由粗变细直到彻底消失。



以上连续 12 帧的图片展现了一个火花四射的完整过程，因为游戏运行帧数为 60 fps，所以每个火花的生命周期大约是 0.2 秒。

4.2 矩形

绘制矩形则可以使用 Begin(RECTS) 调用，该函数将矩形的对角坐标作为参数，两端点不分先后顺序。使用该函数绘制的矩形包含了圆角，圆角半径是由当前设定的直线宽度决定的。圆角部分的面积是在矩形范围之外，所以增加圆角半径将会扩大绘制出的矩形面积。此例中以不断增加的圆角半径绘制了三次 420×20 的矩形：

```
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 渐变

渐变是一个十分实用的图像功能，可以在屏幕上实现平滑的色彩过渡。虽然通过加载一张渐变图片可以实现同样的功能，但是 cmd_gradient 调用更加方便高效。

通过提供起始和结束的颜色和位置， cmd_gradient 就会在全屏范围绘制出平滑的渐变。在下面的例子中，起始点是左上角 $(0,0)$，起始色是深蓝色（0x0060c0）；而结束点是左下角 $(0,271)$，结束色是深橙色（0xc06000）。

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

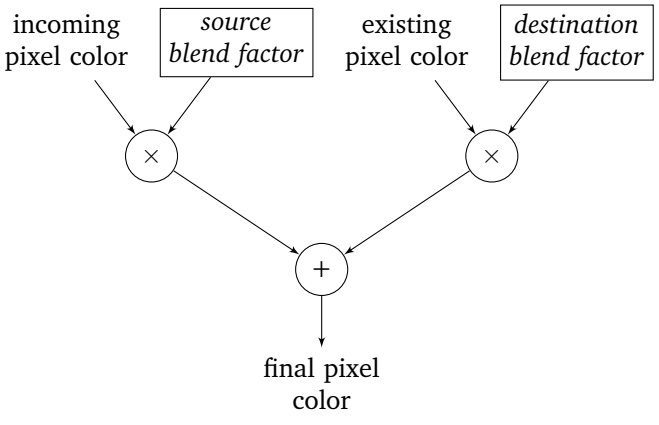
上例通过两个坐标点生成了一个垂直渐变，不仅如此， cmd_gradient 也能够绘制以任意角度变化的渐变色。通过设置两个坐标控制点的坐标，用户也可以绘制出水平或是对角线变化的渐变色。

像这样对 cmd_gradient 进行调用是对屏幕上所有像素点进行绘制，所以它可以用来替换 Clear。当只要求对屏幕的一部分绘制渐变色时，可以使用 ScissorXY 和 ScissorSize 来规定绘制的矩形区域。如果要设置更加复杂的渐变区域，可以与蒙板配合使用。

4.4 图像混合

```
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);
```

通常在绘制图像时，新的像素都是直接覆盖原像素的。这种方式对实现分层绘制十分实用，但有的时候我们还需要进行图像混合的操作。 BlendFunc 即为控制此类操作的函数， BlendFunc 的第一个参数是源混合因子，负责控制输入的图像像素。第二个参数是目标混合因子，负责控制当前已经在颜色缓冲中存在的图像。两个图像在经过图像因子修正颜色之后，最终合并产生最后的图像。



默认的模式为 BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)，这种模式下，输入图像的色彩根据当前alpha通道的大小和当前屏幕内容等比例混合。

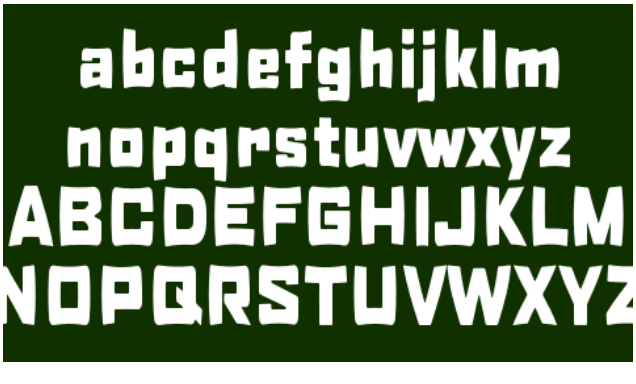
另一个常用的模式为 BlendFunc(SRC_ALPHA, ONE)。ONE 作为目标混合因子表示将引入图像的色彩直接叠加到屏幕已有的内容上，这一模式十分适用于光晕和层叠效果。如上面的代码所示，这一模式下绘制的每个圆盘都与屏幕现有的图像进行混合，因此重合部分的亮度也是累加的。

另外还有 BlendFunc(SRC_ALPHA, ZERO) 这一替换模式；它将禁用透明度设定。目标混合因子 ZERO 表示当前屏幕画面在图像混合过程中将会被直接舍弃。

4.5 字体

Gameduino 2 的GPU内置 16 种字体，字号范围为 16 ～ 31 。其中字号 26 ～ 31 是含抗锯齿的高质量字体。另外，用户可以使用资源转换器（第 39 页：转化图形）加载TrueType格式（ `.ttf` ）的字体。加载后，用户可以在任意绘制指令中使用这个新字体。

```
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 子坐标系

自始至此，所有关于绘制点的指令都是通过调用 Vertex2ii 实现的，但是 Vertex2ii 只能绘制 $0 \sim 511$ 范围的整数坐标点 (x, y)。那么如何绘制一个坐标为负值的图形或是非整数坐标点呢？这些情况下可以调用 Vertex2f，它对于 x 和 y 的坐标参数有着更精细的单位和更大的取值范围。Vertex2f 坐标的单位是 $1/16$ 像素。因此，如下所示调用 Vertex2ii 的代码：

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

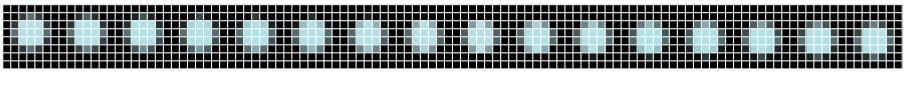
当调用 Vertex2f 实现时将变为：

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

但通常为了使代码更加简单明了，这一代码会以如下所示乘以 16 的方式进行编写：

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

通过 Vertex2f 绘制点（POINTS）、线（LINES）和矩形（RECTS）可以更加精细。下图中的 16 个圆点是由 y 坐标值依次增加 $1/16$ 像素所绘制的。如图所示，当圆点以每次向下运动 $1/16$ 像素时，系统会对像素的阴影部分进行十分微小的调节。这种出色的精细度将大大削减“像素颗粒”效应。



4.7 Furmans角度

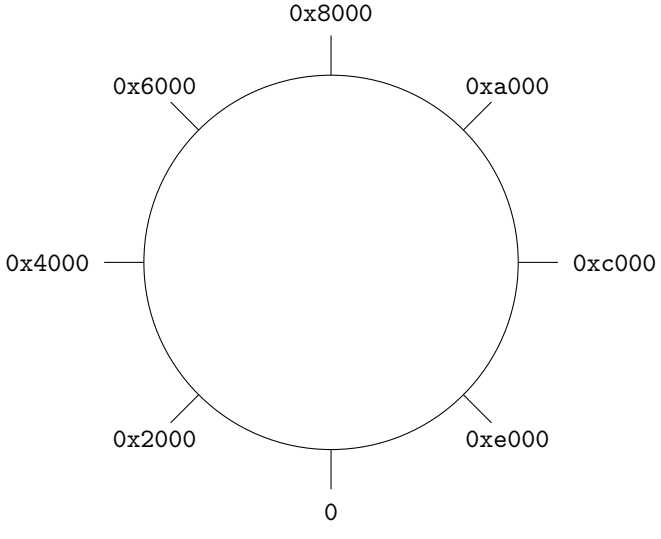
GD程序库在以下指令中会涉及到角度的应用：

- cmd_rotate 指令旋转图像时需要用到角度参数；
- cmd_dial 指令以某一特定的角度绘制表盘控件；
- 调用 cmd_track 跟踪控件时输出的转动控件的旋转角度；
- 数学函数rsin 、 rcos 、 polar 和 atan2 都需要使用角度参数。

上列所有情况，角度的单位都特定为Furmans角度，而不是角度和弧度。Furman是一种角度度量单位，定义一个圆周是 65536 个Furmans度。其转化关系如下列公式所示：

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

角度的方向是顺时针方向，0 刻度值在正下方：



为了方便，GD程序库定义了一个宏函数 DEGREES() ，它可以将角度转换成相应的Furmans角度：

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

使用Furmans角度作为单位的优点在于它可以使包含角度的数学运算更加易于计算。例如，当增大某个旋转物体的角度时，只需要用普通的 16 位运算就可表示从 65535 到 0 度的Furman角度运算。

在游戏NightStrike中，玩家通过触按屏幕控制炮塔的角度。游戏启动时，游戏程序命令GPU跟踪任意触摸标签为 0x01 的触击：

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

cmd_track 指令执行后，GD.inputs.track_val将记录下从屏幕坐标 $(240, 271)$ 到触摸点坐标这一向量的Furmans角度，之后程序再将这一数据赋值给炮塔的角度变量。

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

当绘制炮塔时，cmd_rotate 指令将使用这一角度变量旋转炮塔的图像，下图为炮塔在 0 Furmans角度时的原图像：

4.8 环境堆栈


```
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);
}
```

NightStrike 的标题界面使用函数 blocktext 来绘制带有黑色轮廓的字符串 s 。实现这一效果时，要使用 ColorRGB 将颜色变为黑色，然后分别以左上、左下、右上、右下四个方向微小的位置偏差将文本绘制四遍。而后需要再次以文本原有的颜色绘制文本，但先前的操作已经将文本的颜色设置为黑色，导致文本原有的颜色丢失。

为了解决这个情况，在绘制前调用 SaveContext 将所有的图形系统的状态 —— 包括颜色信息 —— 存储在一个私立的存储区。之后再执行 RestoreContext 便可恢复先前存储的所有属性，这样文本的颜色也会恢复为先前调用 SaveContext 时的颜色。

通过配合使用 SaveContext 和 RestoreContext，图形系统的状态可以被保存，修改，之后再进行可以恢复。只要未被调用，函数所存储的属性信息不会被修改。Gameduino的GPU有足够的内存空间支持存储额外三份这样的图形属性备份。

Chapter 5

触摸屏

5.1 读取触摸屏的输入

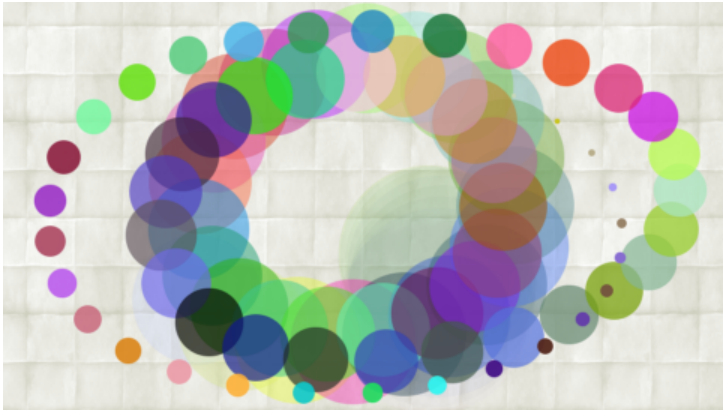
函数 get_inputs 可以读取Gameduino 2的所有输入数据，其中包括触摸传感器的输入。调用此函数后, 你可以通过 GD.inputs.x 和 GD.inputs.y 得到按击位置的相应坐标。程序中检测触摸输入的最佳时间是在开始绘制游戏界面之前，如下所示：

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

没有触摸信号时， GD.inputs.x 和 GD.inputs.y 的值为 -32768 。触摸所得的坐标值 (x, y) 是在屏幕上的像素坐标，其范围分别是 $(0 \leq x \leq 479)$ 和 $(0 \leq y \leq 271)$ ， 将此类像素坐标转换为子坐标系坐标时只需将相应数值乘以 16 。如下所示：

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

5.2 设计实例: blobs



在演示程序“blobs”中，你可以用手指或是手写笔绘制出一连串五颜六色不断扩散的圆点。这段代码支持连续捕捉 128 个点，每个点都通过不断改变半径和透明度呈现出一种有趣的淡出效果。

程序启动时，`begin()` 函数首先将所有blobs的坐标设定在屏幕范围外。

```
#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;
  }
}
```

当屏幕被点击时，点击位置的子坐标系坐标将被添加至 blobs 序列中。各个blob的透明度和半径取决与其当前的生命周期。所谓“随机”的颜色实际上是通过取模运算得到的。

```
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 触摸标签

作为GPU的一项特色功能,触摸标签使触摸检测变得更加轻松。在多数情况下，触摸的具体坐标并不重要 —— 你只需关注被触摸的对象是什么。每当你在屏幕上绘制一个对象时，你可以对该对象所包含的所有像素点赋予一个字节的编码。例如，创建一个'OK'按钮，并对其所包含的像素点赋予代码值 44 。此后，每当用户点击这个按钮，标签寄存器中的值就会变为 44 。触摸标签值均为单字节数值，范围从 $0 \sim 255$ 。不过，设备将 255 这一标签值视为"无触摸产生"，因此标签值的实际可用范围为 $0 \sim 254$ 。

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	

下面这个例子在绘制每个数字之前设定了对应标签值，这样每个数字所含的像素点都会被标记出来。在触摸对应数字后，GD.inputs.tag 会被设定为当前点击的数字，Arduino可以读取出来并通过串口输出到屏幕上。

```
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 涂鸦

涂鸦在这里表示采用触摸输入在屏幕上进行绘图。Gameduino 2 的GPU已内置涂鸦功能，因此你无需通过Arduino输入任何指令即可绘制像素图像。cmd_sketch 指令用于启动绘图功能，而 cmd_stop 指令用于停止绘图。注意，GD.finish() 调用会将使缓冲区的指令立即强制送入GPU中。

```
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 控件与跟踪控制

Gameduino 2 有丰富的控件可供使用：仪表盘、按钮、计量器、滑动条和滚动条。为了更好地管理与这些可调控件的交互，可以使用跟踪控制这个功能拓展前面提到的标签系统。利用标签功能，每当相应对象被点击时程序都能够立刻得知；而通过跟踪控制，程序不仅可以得知发生了触摸事件，还可以得知触摸发生的相对位置。

`widgets` 例程绘制出了所有的控件类型，并且对于其中三个可调控件——表盘，滑动条和拨动开关——使用了跟踪控制功能对它们的触控进行管理。在这列，这三个可调控件在绘制时都被赋予独立的标签值：200、201 和 202。

之后程序命令GPU跟踪这三个可调控件区域内的触控。程序首先对表盘赋予 200 的标签值，然后调用 cmd_track 令GPU跟踪所有在这个标签值下的触按，并计算其对于表盘中心（屏幕坐标位置为 $(68, 68)$ ）的相对位置：

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

对于线性控件如滑动条和拨动开关也有类似的控制方法：

```
  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);
```

这样之后，程序能够通过读取 GD.inputs.track_tag 和 GD.inputs.track_val 检测触摸对于任意控件的相对位置。 GD.inputs.track_val 中的值为无符号 16 位整型数值，范围为 $0 \sim 65535$。对于转动跟踪——仪表盘——该值代表对应的Furmans触按角度；而对于线性跟踪，该值代表在矩形中的触摸位置：0 代表最左端，65535 代表最右端。

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

Chapter 6

声音

Gameduino 2 有两种声音系统。第一种声音系统——音频合成器——能够生成一系列固定的声音和乐器音符。该系统对于在程序中添加音频十分实用和便捷，但由于声音素材是固定的，因此所添加的音频灵活可变性相对较差。第二种声音系统是采样回放，该系统能够播放存放在内存中的多种格式的音频素材。这种系统相对来说灵活性大大增加，但你需要提前准备音频素材，并在RAM中载入所需音频。

6.1 敲击音效

音频合成器提供若干短促的"打击"音效，主要是为用户交互界面提供音效素材。通过调用 GD.play() 函数并提供音效素材的ID，可以触发对应的音效。所有可用的音效列表如下所示：

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

6.2 乐器音符

音频合成器也能够生成乐器音符。同样是通过调用 GD.play() 来触发这些音符，只是在这个情况下，第一个参数设置为了乐器的名称：

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

这种情况下，GD.play() 还可以接受一个额外的参数，用于设置MIDI音符号，其范围为 21 ∼ 108。当该参数空缺时，程序默认播放C4的音效，即MIDI音符 60 。

音频合成器还可以播放一些其他的音效。乐音 SQUAREWAVE，SINEWAVE，SAWTOOTH 和 TRIANGLE 能够合成简单的连续波形，这可以用于产生老式游戏的音效感。乐音 BEEPING，ALARM，WARBLE 和 CAROUSEL 能够合成各式各样的警报音效。另外，ASCII代码 '0'-'9'，'*' 和 '#' 能够生成对应的DTMF音效（电话机的播号音效）。

通过读取设备的 PLAY 寄存器可以检测声音是否播放结束。当此寄存器内数值为 0 时，代表声音已经播放结束：

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

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

6.3 采样回放

Gameduino 2 的声音系统还可以进行采样回放。采样回放系统是独立的——你可以在播放固定音符的同时播放音频素材，而设备将会通过分配好的音量将它们混合在一起进行播放。

调用 GD.sample 可以播放内存中的采样音频。该函数设置的参数有：音频的基准地址，音频文件大小，频率以及格式。举例来说，若要播放一段位于地址 0，音频文件大小为 22050 字节，频率为 44100 Hz的ULAW格式的音频，程序如下：

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

采样回放系统支持3种音频格式：

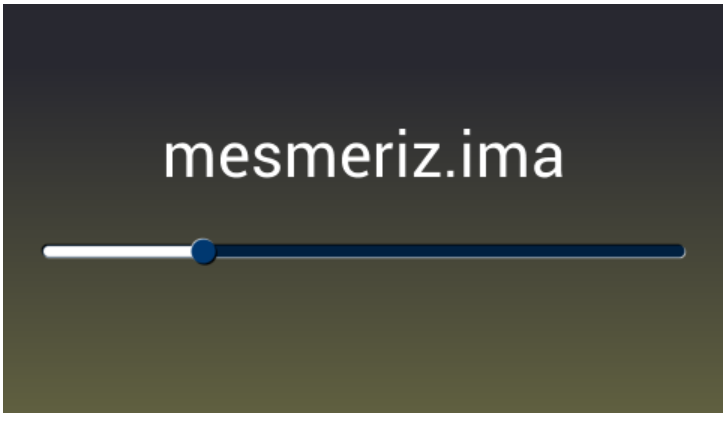
格式	音频位数	音频编码
LINEAR_SAMPLES	8	有符号 8 位线性，−128 - 127
ULAW_SAMPLES	8	标准 8 位μ-law编码
ADPCM_SAMPLES	4	标准IMA ADPCM编码

Gameduino 2 的资源转换器（第 39 页：转化图形）可以将单声道的 .wav 文件转换成上述任意格式。硬件设备最高可以支持 48 kHz的采样率，在此频率下，ADPCM格式音频素材的声音质量和AM收音机等同。当然，在此频率下的音频所占内存的大小也很高，大约是 24 kBytes/s。为了节省内存，最好采用较低采样率的ADPCM音频。noisy 例程播放了数字 0 至 9 的音频采样，采样率为 8 kHz，编码格式为ADPCM。这些采样的声音简单明了，总共只占了 26 kBytes的显存。

值得注意的是，GD.sample() 对于参数有一个强制限制，那就是基准地址的长度必须是 8 的倍数。不过资源转换器会自动对采样音频进行校准，所以用户对此不需再做什么特殊操作。

```
#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 连续播放



对于较长的音频，将整个音频采样存入内存通常不现实。取而代之的是，可以使用采样回放系统通过循环来播放一段音频。在这个情况下，Arduino不断读取SD卡，并并更新需要播放的音频。GD2程序库中的 Streamer 类可以十分轻松的管理这些细节。首先，调用它的 begin() 方法，并以microSD卡上音频文件的名字作为参数。Streamer 默认情况下采用采样率为 44.1 kHz的IMA ADPCM的采样编码。音频流使用 4 K的缓存，这些缓存被安排于显存中的顶端。

调用 Streamer 的 feed() 方法从文件中读取音频采样，并更新至缓存中。应用程序应时常调用 feed() 以防止缓存中的音频采样耗尽。在下例中，程序每一帧都会调用一次 stream.feed() 。

Streamer 还可以获知音频播放进程已经进行到什么位置。progress() 将会返回一对16位的数字， val 和 range ，而当前已经播放的进程比例可以通过 (val/range) 计算而来。在下面的例程中，程序直接将 val 和 range 作为滑动条（ cmd_slider ）的控件参数。

如果有已安装的 sox 音频编辑工具, 可以将 .wav 格式的音频转换为 ADPCM IMA格式， 相应命令行如下所示：

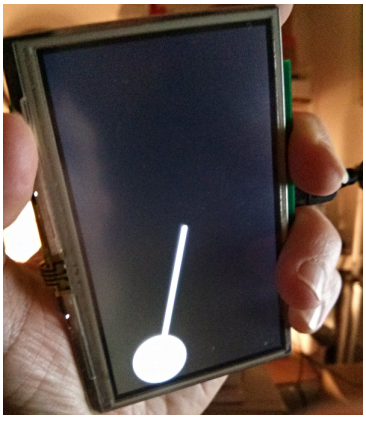
```
$ sox mesmeriz.wav -c 1 mesmeriz.ima
$ play -r 44100 mesmeriz.ima
```

最后，play 将播放转换后的音频文件。由于 .ima 格式没有文件头区域，而对于 play 来说需要得知采样率才可以进行播放，所以在此手动设定为 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

加速度传感器



Gameduino 2 还包含一个三轴加速度传感器，该传感器的输出直接连接到Arduino的A0, A1和A2模拟输入引脚上。用户可以通过 get_accel 函数来获取当前的加速度大小:

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

get_accel() 函数对输出结果进行了矫正，将单位重力在 x, y 或 z 轴上的分量大小调整为 256 。如果将 Gameduino 2 完全倾向左侧，就会得到以下数值： x = −256, y = 0, z =0。因为该加速度传感器是模拟量输入，所以这些值会存在一些误差。在示例程序 tilt 中，你将看到在加速度传感器的输出中就会存在一些微小的“噪声”。

```
#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卡

Gameduino 2 包含一个标准microSD卡接口。 它与 Gameduino 2 的GPU芯片共用一组SPI接口，并通过Arduino的数字引脚9进行使能。 现在有很多程序库可以用于驱动microSD卡，而GD程序库包含了一个相对轻量级的程序库。 该程序库可以用于驱动FAT格式的microSD卡，并且与GD图形系统无缝集成。

MicroSD的文件经常被直接输入到GPU的命令流中。 `GD.load()` 命令就是如此的，它的工作流程是：该函数从SD中读取一个文件，并把它的内容传送到 Gameduino 2 的GPU中。 举一个例子，用户可以通过以下指令来载入一张JPEG格式的图片：

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

函数 `cmd_loadimage` 告知GPU准备接收一个JPEG图片，之后 `load` 函数将图片数据全部读取到GPU中。

下面是另外一个例子，演示了如何从SD卡的“`mem.raw`”文件中读取 16KB的原始数据：

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

同样的，此处 cmd_memwrite 告知GPU准备接收数据，之后由 load 函数将数据装载至GPU中。

由对象转化器(第 39 页：转化图形)生成的 .gd2 格式文件为纯命令流，也可以通过 load 函数载入：

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

通过该命令，所有 frogger 游戏需要的图形元素就被载入到了内存中，并同时设置好了相应的位图句柄。在对象转换器生成的头文件 frogger_assets.h 中，包含了以下的宏定义：

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

所以在 frogger 的程序中，只需简单地在 setup() 函数中调用 LOAD_ASSETS() 就可以载入所有的图形资源了。

Part II

编程参考

Chapter 9

绘图指令

9.1 AlphaFunc

设置透明度测试功能

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

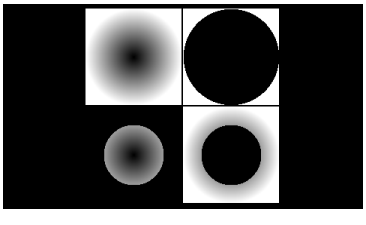
func 检测采用的比较方式，可用参数有 NEVER、 LESS、 LEQUAL、 GREATER、 GEQUAL、 EQUAL、 NOTEQUAL 或 ALWAYS

ref 比较时参考的数值

透明度检测功能用于检测所有像素点的透明度值，并只绘制出满足条件的像素点。 例如：

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

表示只选择绘制透明度值满足 $(A \geq 160)$ 的像素点。 默认状态下的比较方式是 ALWAYS，即绘制所有像素点。



```
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

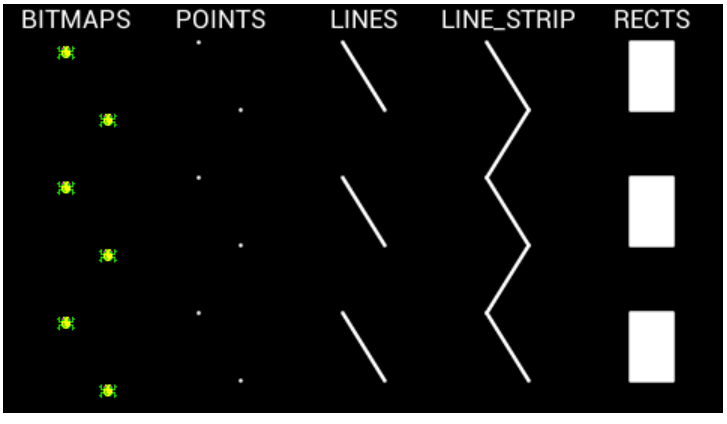
选取用于绘制的绘图元素

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

prim BITMAPS、 POINTS、 LINES、 LINE_STRIP、 EDGE_STRIP_R、 EDGE_STRIP_L、 EDGE_STRIP_A、 EDGE_STRIP_B 或 RECTS 之一

Begin 指令用于设置当前绘制的图形种类。这一指令并不绘制任何图像——绘制命令由之后会介绍的 Vertex2f 或 Vertex2ii 指令实现。其中可供绘制的图元包括：

BITMAPS	每次调用vertex指令时将绘制一幅位图图片
POINTS	每次调用vertex指令时将绘制一个平滑边缘的点
LINES	每次调用一对vertex指令时将绘制一条平滑线段
RECTS	每次调用一对vertex指令时将绘制一个平滑边缘的矩形
LINE_STRIP	调用一组vertex指令时将按顺序绘制一条首尾相接的折线
EDGE_STRIP_A	同 LINE_STRIP 类似， 不过是对指定线上方的像素点进行绘制
EDGE_STRIP_B	同 LINE_STRIP 类似， 不过是对指定线下方的像素点进行绘制
EDGE_STRIP_L	同 LINE_STRIP 类似， 不过是对指定线左侧的像素点进行绘制
EDGE_STRIP_R	同 LINE_STRIP 类似， 不过是对指定线右侧的像素点进行绘制



9.3 BitmapHandle

设置位图句柄

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

handle 整型句柄编号，范围为：$0 \sim 15$

BitmapHandle 指令用于设置位图句柄，位图句柄可用于 Vertex2f ，BitmapSource ， BitmapLayout 和 BitmapSize 。 位图句柄是图形环境的一部分，其默认值为 0 。

9.4 BitmapLayout

设定位图的内存分布

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

format 位图像素的格式，ARGB1555、L1、L4、L8、RGB332、ARGB2、ARGB4、RGB565、PALETTED、TEXT8X8、TEXTVGA、BARGRAPH 中选择其一

linestride 位图图像中每一行数据在存储器中的大小，单位为字节

height 位图的高度，单位为像素

BitmapLayout 指令用于设定位图在存储器中的分布。format 用于控制存储器数据如何转化为像素。每一个图形像素的位数可以为 1、4、8 或是 16 位。色彩信息可以从每个像素的数据中抽取得到，对应关系如下所示，其中"v"表示每像素点的数据。

格式	每像素图像的位数	alpha	红	绿	蓝
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 位和 4 位的像素点按照从左至右的顺序打包成为字节，因此字节中最左边的像素占据该字节的最高位。16 位图像在显存中以小端存储格式存储，而且存储时必须按照偶数内存边界对齐。

PALETTED 格式每像素占 8 位大小，每一像素数据都对应 RAM_PALETTE中存储的 256 个 32 位颜色之一。

9.5 BitmapSize

设置位图的尺寸及外观

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

filter 位图的过滤方式，NEAREST 或 BILINEAR

wrapx x 轴方向滚动模式， BORDER 或 REPEAT

wrapy x 轴方向滚动模式， BORDER 或 REPEAT

width 屏幕上绘制的宽度（像素）

height 屏幕上绘制的高度（像素）

BitmapSize 指令用于设置当前位图在屏幕上的显示方式。

9.6 BitmapSource

设置位图的源地址

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

addr 位图的基地址，取值范围为 0x00000 ~ 0x3ffff

BitmapSource 指令用于设置源位图的基地址。对于 16 位数据格式的位图文件，该地址必须为偶数。

9.7 BlendFunc

设定颜色混合功能

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

src 源混合因子，可选参数为 ZERO、ONE、SRC_ALPHA、DST_ALPHA、ONE_MINUS_SRC_ALPHA 和 ONE_MINUS_DST_ALPHA

dst 目标混合因子，可选参数为 ZERO、ONE、SRC_ALPHA、DST_ALPHA、ONE_MINUS_SRC_ALPHA 及 ONE_MINUS_DST_ALPHA

BlendFunc 指令用于将指定图像与当前帧缓冲器的内容进行图像混合。输入图像的颜色与源混合因子相乘，而当前帧缓冲器中的图像与目标混合因子相乘，最后再将这两部分的结果相加得到最终图像。

```
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);
```

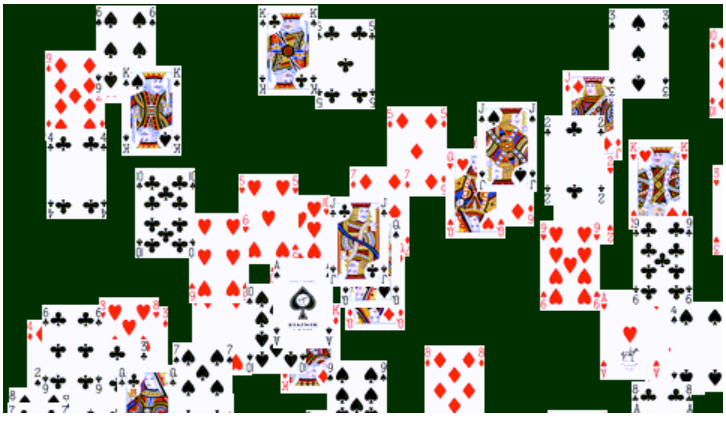
9.8 Cell

设置位图单元

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

cell　位图单元号，取值范围为 0 ~ 127

Cell 指令用于设定当前 Vertex2f 指令使用的位图单元。



```
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

设定清屏背景色的alpha通道分量

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

alpha 清屏背景色的alpha通道分量，取值范围为 $0 \sim 255$

ClearColorA 指令用于设定清屏背景色的Alpha通道值。随后调用的Clear 函数会将此数值写入显示缓冲器的alpha通道中。

9.10 Clear

清屏

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

c 设定该位时，清空颜色缓冲区

s 设定该位时，清空模板缓冲区

t 设定该位时，清空标签缓冲区

Clear 指令用于清空指定的帧缓冲器。

```
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

设定清屏背景色的红、绿、蓝通道各分量

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

red	红色通道值，取值范围为 $0 \sim 255$
green	绿色通道值，取值范围为 $0 \sim 255$
blue	蓝色通道值，取值范围为 $0 \sim 255$
rgb	24位色，顺序依次为红、绿、蓝， 取值范围为 0x000000 ~ 0xffffff

ClearColorRGB 指令用于设定清屏背景色的红、绿、蓝色彩通道分量值。随后调用 Clear 函数将会把这些设定值写入帧缓冲器的色彩通道中。

```
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

设定模板的清除值

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

s 需要清除模板缓冲区中的模板值，取值范围为 $0 \sim 255$

ClearStencil 指令用于设定需要清除的模板值。随后调用 Clear 函数将会把这一值写入当前模板缓冲器中。

9.13 ClearTag

设定需要清除的标签值

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

s 标签值，取值范围为 $0 \sim 255$

ClearTag 指令用于设定需要清除的标签值。随后调用 Clear 将这一值写入当前标签缓冲器中。

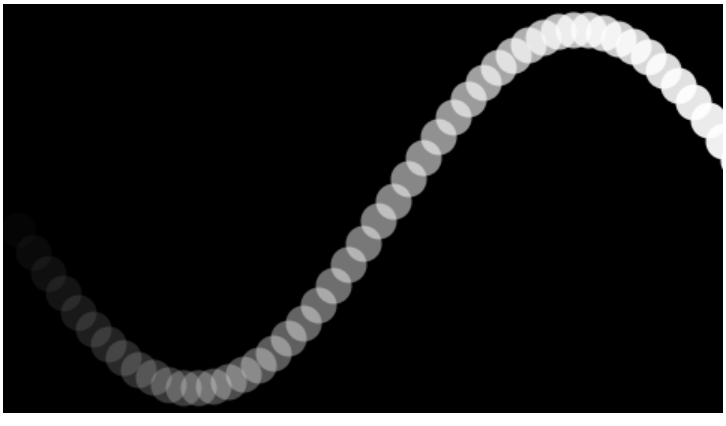
9.14 ColorA

设定当前色彩设置的Alpha通道值

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

alpha　alpha通道值，取值范围为 $0 \sim 255$

ColorA 指令用于设定当前绘制色彩设置的Alpha通道值。



```
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

设定控制色彩通道的颜色蒙板

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

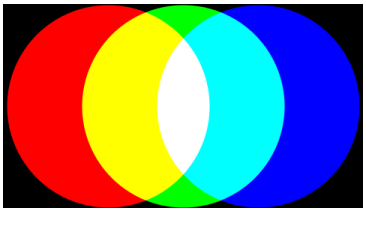
r 设定该位时，使能对红色色彩通道的写入

g 设定该位时，使能对绿色色彩通道的写入

b 设定该位时，使能对蓝色色彩通道的写入

a 设定该位时，使能对alpha通道的写入

ColorMask 指令用于设定色彩通道的颜色蒙板，使能 / 禁止对帧缓冲器中红、绿、蓝及alpha通道的写入。



```
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

设定当前色彩的红、绿、蓝通道值

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

void ColorRGB(uint32_t rgb);
```

red	红色通道值，取值范围为 $0 \sim 255$
green	绿色通道值，取值范围为 $0 \sim 255$
blue	蓝色通道值，取值范围为 $0 \sim 255$
rgb	24 位色，顺序依次为红、绿、蓝， 取值范围为 0x000000 ~ 0xffffff

ColorRGB 指令用于设定当前色彩设置。这一颜色设置适用于所有绘图操作。

```
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

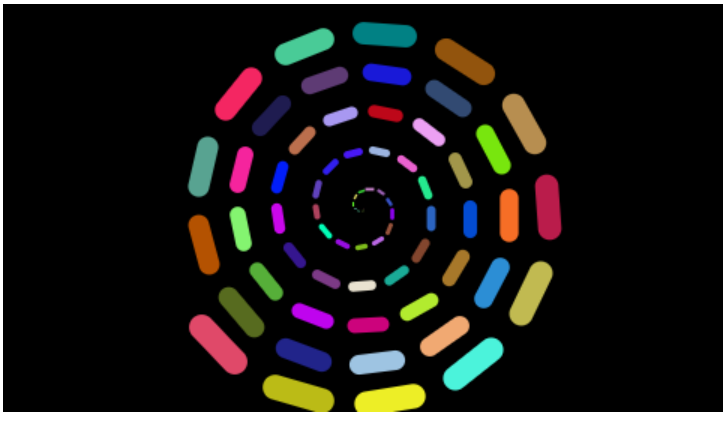
设定绘制线条的宽度

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

width 线宽，单位为 1/16 像素

LineWidth 指令用于设定绘制 LINES 和 LINE_STRIP 时的线条宽度。宽度单位为 1/16 像素，即 LineWidth(16) 表示 1 像素的线宽。值得注意的是，此线宽代表线条从其宽的中心到外边缘的距离，概念类似圆的半径。因此线条的实际宽度是指定值的两倍。

最大支持的线宽值为 4095，即 $255\frac{15}{16}$ 像素。



```
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

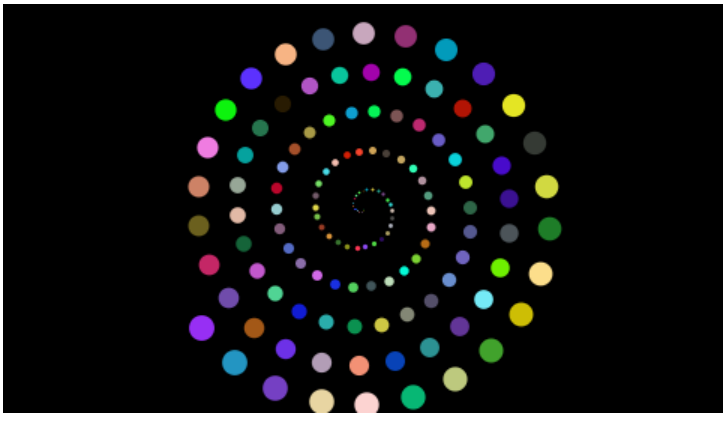
设定点的大小

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

size 点的大小，单位为 1/16 像素

PointSize 指令用于设定绘制 POINTS 时点的大小。其大小单位为 1/16 像素，即 PointSize(16) 表示 1 像素的点大小。值得注意的是，此大小代表从点的中心到其外边缘的距离，即半径。因此点的直径是指定值的两倍。

最大值为 4095 即 $255\frac{15}{16}$ 像素。



```
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

恢复绘制环境为先前保存的状态

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

图形显示属性的集合统称为图形环境（*graphics context*）。由 SaveContext 将这一属性集合存储一份备份，而 RestoreContext 指令用于将图形环境恢复为这一备份的状态。

设备支持存储最多4份备份。其图形环境内容包括：

属性	相应绘图指令
透明度测试功能	AlphaFunc
位图句柄	BitmapHandle
图像混合功能	BlendFunc
位图单元	Cell
清屏背景色彩值	ClearColorA, ClearColorRGB
清除模板值	ClearStencil
清除标签值	ClearTag
色彩图层蒙版	ColorMask
绘图颜色	ColorA, ColorRGB
线宽	LineWidth
点大小	PointSize
裁切矩形窗口	ScissorSize, ScissorXY
模板检测功能	StencilFunc
模板图层蒙板	StencilMask
模板操作	StencilOp
标签蒙板	TagMask
标签值	Tag

9.20 SaveContext

保存图形环境

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

图形显示属性的集合统称为图形环境（*graphics context*）。SaveContext 指令用于将这一属性集合存储一份备份，而 RestoreContext 将图形环境恢复为这一备份的状态。


```
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

设定裁切矩形窗口的尺寸

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

width 裁切矩形窗口的宽度，单位为像素，取值范围为 $0 \sim 512$

height 裁切矩形窗口的高度，单位为像素，取值范围为 $0 \sim 512$

ScissorSize 指令用于设定裁切矩形窗口的尺寸规格。裁切矩形窗口用于将绘图范围限制在一个指定的矩形区域中。


```
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

设定裁切矩形窗口的左上角的坐标位置

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

x 裁切矩形窗口的左上角的 x 轴坐标，取值范围为 $0 \sim 511$

y 裁切矩形窗口的左上角的 y 轴坐标，取值范围为 $0 \sim 511$

ScissorXY 指令用于设定裁切矩形窗口的左上角的坐标位置。裁切矩形窗口用于将绘图范围限制在一个指定的矩形区域中。

9.23 StencilFunc

设定模板测试功能

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

func 设定模板测试操作的比较方式，可选值有：NEVER，LESS，LEQUAL，GREATER，GEQUAL，EQUAL，NOTEQUAL 或 ALWAYS

ref 设定用于比较的模板参考值，取值范围为 $0 \sim 255$

mask 8 位蒙板，该蒙版会在检测前分别与 ref 和图像模板值进行与操作，取值范围为 $0 \sim 255$

StencilFunc 指令用于控制模板测试操作。绘制期间，模板测试会应用于每一个像素，如果测试结果为不成功，该像素将不被绘制。设定 func 为 ALWAYS 表示绘制所有像素点（即忽略蒙版）。

9.24 StencilMask

设定用于控制写入图层模板的掩码

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

mask　每一位对应控制激活模板缓冲区的对应位的写入操作

StencilMask 指令用于控制向模板缓冲区写入数据。由于模板缓冲区有 8 位大小，mask 中的每一位控制模板缓冲区的对应位的写入使能。因此 mask 为 0x00 时将禁止模板写入，而 mask 为 0xff 时将激活模板写入。

9.25 StencilOp

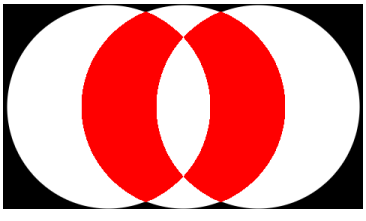
设定模板更新操作

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

sfail 当某像素点的模板测试失败时对其进行的操作。ZERO， KEEP，REPLACE， INCR， DECR 或 INVERT 中选择其一

spass 当某像素点的模板测试通过时对其进行的操作。ZERO， KEEP，REPLACE， INCR， DECR 或 INVERT 中选择其一

StencilOp 指令用于控制对像素点进行绘制操作时如何修改模板缓冲区。当某像素点的模板测试失败时，将执行 sfail 指定的操作。通过测试时，将执行 spass 指定的操作。



```
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

设定标签图层写入掩码

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

mask 设定该位时，绘制的图像将向标签缓存中写入数据

TagMask 指令用于控制标签缓存中数据写入的使能。当 mask 为 1 时，每当有像素点有绘图操作，Tag 将向标签缓存写入对应字节作为其标签值。当 mask 为 0 时，绘制操作对标签缓存不会产生影响。

9.27 Tag

设定绘制对象的标签值

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

s 标签值，取值范围为 $0 \sim 255$

Tag 指令用于指定一个单字节数值作为绘制对象的标签值。

9.28 Vertex2f

在子像素坐标系下绘制图像

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

x 绘制图像位置的 x 坐标值，精确度为 $1/16$ 像素，取值范围为 $-16384 \sim 16383$

y 绘制图像位置的 y 坐标值，精确度为 $1/16$ 像素，取值范围为 $-16384 \sim 16383$

Vertex2f 指令只用于指定绘制图像的屏幕位置，而绘制的内容由当前 Begin 指定的对象决定。Vertex2f 指定子坐标系（*subpixel*）坐标，所以其精确度是 $1/16$ 像素。除此以外，其支持的坐标范围要大于实际屏幕的范围——这对于绘制尺寸大于屏幕的图像对象十分有帮助。

绘制 BITMAP 时，Vertex2f 将调用由当前 BitmapHandle 和 Cell 指定的位图句柄和位图单元。

9.29 Vertex2ii

在整型坐标值像素点处绘制图像

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

x	绘制图像位置的 x 坐标值，取值范围为 $0 \sim 511$
y	绘制图像位置的 y 坐标值，取值范围为 $0 \sim 511$
handle	位图句柄，取值范围为 $0 \sim 31$
cell	位图单元，取值范围为 $0 \sim 127$

Vertex2ii 指令用于指定绘制图像的屏幕位置，而绘制的图像类型由当前 Begin 指定的对象决定。

绘制 BITMAP 时，Vertex2ii 将调用由 handle 和 cell 指定的位图句柄和位图单元。而对于指令 BitmapHandle 和 Cell 中的图形属性信息，该指令既不会使用到也不会对其进行修改。

Chapter 10

高级指令

这一部分主要介绍 Gameduino 2 的GPU支持的一些高级指令。 这些指令都在GPU中运行，以便让主控MCU更专注于游戏运行及执行代码。

这类指令的一部分用于创建控件，助力UI部分的设计。 其它指令则提供高效的方法初始化和管理GPU内存、加载图形及其它资源。

这一部分将精选对于游戏以及交互式程序设计最有帮助的一部分设备指令。 如需高级指令的完整清单，请参见FT800编程手册。

10.1 cmd_append

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

append 指令用于调用显存地址 ptr 开始，长度为 num 字节的绘图指令并执行。这一方式相当于把显存作为高速缓存来使用，对于频繁调用同一段绘图指令的操作十分适用，很像OpenGL中的显示列表。

10.2 cmd_bgcolor

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

bgcolor 指令用于设定绘制控件时使用的背景色。所有控件均包含一些相同的样式；对于非交互式部件使用背景色（ cmd_bgcolor ）绘制，对于交互式部件使用前景色（ cmd_fgcolor ）绘制。通常情况下最好使用较深的颜色作为背景色（ cmd_bgcolor ），较浅的颜色作为前景色（ cmd_fgcolor ）。

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);
```

button 指令用于在屏幕坐标 (x, y) 处绘制大小为 $w \times h$ 的按钮。 label 用于设置文本的标签。 默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。

10.4 cmd_calibrate

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

calibrate 指令用于运行GPU内置的交互式触摸屏校准程序。

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
```

clock 指令用于在屏幕像素坐标 (x, y) 处绘制一个半径为 r 的模拟时钟。 显示的时间由h 、 m 、 s 和 ms 指定。 默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。

10.6 cmd_coldstart

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

coldstart 指令用于将所有控件的状态重置为默认值。

10.7 cmd_dial

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

dial 指令用于在屏幕像素坐标 (x, y) 处绘制一个半径为 r 的标度盘。 val 用于设定标度盘上显示的角度位置，单位为Furmans角度。 默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。

10.8 cmd_fgcolor

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

fgcolor 指令用于设定用于绘制控件时的前景色。 所有控件均包含一些相同的样式；对于非交互式部件使用背景色（ cmd_bgcolor ）绘制， 对于交互式部件使用前景色（ cmd_fgcolor ）绘制。 通常情况下最好使用较深的颜色作为背景色（ cmd_bgcolor ），较浅的颜色作为前景色（ cmd_fgcolor ）。

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);
```

gauge 指令用于绘制一个模拟仪表盘， 其位置位于屏幕像素坐标 (x, y) 处，半径为 r ，单位为像素。 major 和 minor 用于设定仪表盘上最大和最小刻度值。 分数值（ val / range ） 为仪表盘指针显示值。 默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。

10.10 cmd_getprops

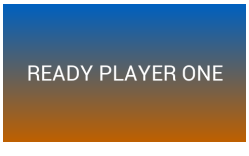
```
void cmd_getprops(uint32_t &ptr,
                  uint32_t &w,
                  uint32_t &h);
```

getprops 指令用于询问GPU上一张由 cmd_loadimage 指令加载的图片属性。ptr 为这一图片的基地址， 而 (w, h) 为这一图片的尺寸大小，单位为像素。

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);
```

gradient 指令用于绘制一幅平滑变化的渐变色背景，由屏幕像素坐标 $(x0, y0)$ 处的颜色 rgb0 逐渐变化，直至屏幕像素坐标 $(x1, y1)$ 处渐变为颜色 rgb1。 关于 cmd_gradient() 的具体例程请参见 渐变 的内容。



10.12 cmd_inflate

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

inflate 指令用于对压缩数据进行解压并存储至主存储器地址 ptr 中。调用该指令后应提供相应的压缩数据，压缩格式使用 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);
```

keys 指令用于绘制一排按键，每个按键依次由字符串（chars）的每个字母进行标注，其位置位于屏幕像素坐标 (x, y) 处，大小为 $w \times h$。默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。在 options 设置中制定一个字母代码将在按键中对其进行高亮显示。

10.14 cmd_loadidentity

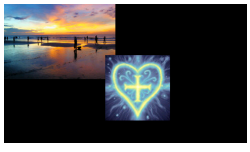
```
void cmd_loadidentity(void);
```

loadidentity 指令用于将位图数据转换为单位矩阵数据。

10.15 cmd_loadimage

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

loadimage 指令用于将一幅JPEG图片解压到从显存地址 ptr 开始的存储空间，并将图片的相关参数存储至当前位图句柄。图片的默认格式为 RGB565，而当 options 设置为 OPT_MONO 时，格式则是 L8。关于 cmd_loadimage() 的具体例程请参见 位图句柄 的内容。



10.16 cmd_memcpy

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

memcpy 指令用于将显存首地址为 src，长度为 num 字节的数据复制到 dest 开始的存储空间。

10.17 cmd_memset

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

memset 指令用于将从显存地址 ptr 开始长度为 num 字节的数据值统一设置为 value 。

10.18 cmd_memwrite

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

memwrite 指令用于将紧随其后长度为 num 字节的数据， 拷贝到从显存地址 ptr 开始的存储空间。

10.19 cmd_regwrite

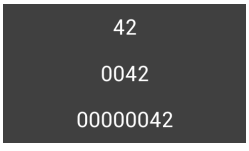
```
void cmd_regwrite(uint32_t ptr,
                  uint32_t val);
```

regwrite 指令用于将GPU寄存器 ptr 的值设定为 val 。

10.20 cmd_number

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

number 指令用于显示一串数字 val，其字体由 font 设定，其位置位于屏幕像素坐标 (x, y) 处。当 options 设置为 n 时，将始终保持显示 n 位数字，如果位数不够，数字前加 0 补齐。当 options 设置为 OPT_CENTERX 时，文本将水平居中显示，为 OPT_CENTERY 时，文本将垂直居中显示，为 OPT_CENTER 时，文本既水平居中，又垂直居中显示。添加 OPT_SIGNED 设置时，val 的值将被识别为有符号型数据，即其值为负数时，在数值前将显示一个负号。



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);
```

progress 指令用于绘制进度条，其位置位于屏幕像素坐标 (x, y) 处，尺寸为 $w \times h$，单位为像素。分数值（val / range）表示当前进度值。当 $w > h$ 时，这一控件将自动水平绘制，否则将竖直绘制。

10.22 cmd_rotate

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

rotate 指令用于对当前位图变换矩阵应用旋转 a Furmans角度的操作。

10.23 cmd_scale

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

scale 指令用于以 (sx, sy) 比例对当前位图变换矩阵进行缩放操作。 上述参数的格式为 16.16 定点浮点数，即 65536 表示 1.0 。 方便起见，调用宏函数 F16() 可将浮点型数据转换为有符号型 16.16 的表达形式。

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);
```

scrollbar 指令用于在屏幕像素坐标 (x, y) 处绘制大小为 $w \times h$ 像素的滚动条。分数值（val / range）用于设定其滚动滑块所处的位置，而分数值（size / range）用于设定滚动滑块的尺寸。当 $w > h$ 时，这一控件将自动水平绘制，否则将竖直绘制。

10.25 cmd_setfont

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

setfont 指令用于定义存储在内存中的标号 font 介于 $0 \sim 15$ 的字体。该字体的属性由存储在内存 ptr 处的字体块定义。在调用 setfont 前，字体库必须已经存储在内存中，且位图句柄必须已经设置好。字体块的大小为 148 字节，具体描述如下所示：

地址	大小	数值
ptr + 0	128	每个字体字符的宽度，单位为像素
ptr + 128	4	字体位图格式，例如 L1 、 L4 或 L8
ptr + 132	4	字体行间距，单位为字节
ptr + 136	4	字体宽度，单位为像素
ptr + 140	4	字体高度，单位为像素
ptr + 144	4	指向内存中字体图形所在位置的指针

10.26 cmd_setmatrix

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

setmatrix 指令用于将当前位图变换矩阵应用到下次绘图操作中。

10.27 cmd_sketch

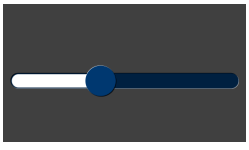
```
void cmd_sketch(int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint32_t ptr,
                uint16_t format);
```

sketch 指令用于开始一次连续的绘图操作。该指令会将之后所有触摸到的像素点记录、绘制到图形内存中。位图的基址由 ptr 给定，其位置位于屏幕像素坐标 (x, y) 处，尺寸为 $w \times h$。位图格式（format）既可以为L1 也可以为 L8。该绘图进程将一直持续，直到用户主动调用 cmd_stop 指令为止。关于 cmd_sketch 的具体例程请参见 第 63 页：涂鸦的内容。

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);
```

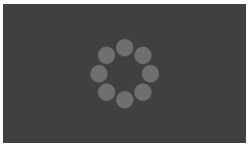
slider 指令用于绘制控制滑块，其位置位于屏幕坐标 (x, y) 处，尺寸为 $w \times h$。分数值（val / range）用于设定滑块的位置。当 $w > h$ 时，这一控件将自动水平绘制，否则将竖直绘制。默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。



10.29 cmd_spinner

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

spinner 指令用于绘制一个“等待”动画，其位置以屏幕像素坐标 (x, y) 为中心。style 用于设定该动画的风格；0 表示圆形动画，1 表示线形动画，2 表示时钟动画，3 表示转盘动画。scale 用于设定动画图片的尺寸：$0 \sim 2$ 表示由小到大，



10.30 cmd_stop

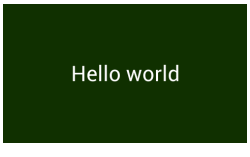
```
void cmd_stop(void);
```

stop 指令用于停止当前运行的小动画或正在执行的绘图操作。具体参见 cmd_spinner 和 cmd_sketch 的内容。

10.31 cmd_text

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

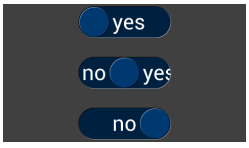
text 指令用于绘制一文本字符串 s，其字体由 font 设定，其位置位于屏幕像素坐标 (x, y) 处。当 options 设置为 OPT_CENTERX 时，文本将水平居中显示，为 OPT_CENTERY 时，文本将垂直居中显示，为 OPT_CENTER 时，文本既水平居中，又垂直居中显示。添加 OPT_SIGNED 设置时，val 的值将被识别为有符号型数据，即其值为负数时，在数值前将显示一个负号。关于 cmd_text() 的具体例程请参见 Hello world 的内容。



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);
```

toggle 指令用于在屏幕像素坐标 (x, y) 处绘制一个宽度为 w 像素的拨动开关。其拨动按钮的位置由 state 设定；0 表示其位置在最左侧，65535 表示在最右侧。其标注文本为一对字符串，由ASCII字符码 0xff 隔开。默认状态下，这一控件使用3D效果的外观显示。将 options 设置为 OPT_FLAT 可令其显示2D效果。



10.33 cmd_track

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

track 指令用于要求GPU追踪对于标签值为 tag 的像素点的触摸，并将相应值送入GD.inputs.track_val。位于屏幕像素坐标 (x, y) 处，尺寸为 $w \times h$ 的矩形空间用于规定追踪的区域。当此区域仅为 1×1 的像素点时，输出值为触摸点相对于起始点 (x, y) 的Furmans角度，由 GD.inputs.track_val 存储该角度。角度追踪功能对于 cmd_dial 、 cmd_clock 这类控件以及含有旋转控制单元的游戏程序（例如 NightStrike ）十分实用。

除此以外，追踪控制均是沿尺寸为 $w \times h$ 的矩形区域长轴方向进行线性追踪的。此时 GD.inputs.track_val 的值表示沿矩形区域长轴上移动的距离，其取值范围为 $0 \sim 65535$ 。线形追踪十分适用于 cmd_scrollbar 、 cmd_toggle 以及 cmd_slider 这类控件。

关于 cmd_track 的具体例程请参见 控件与跟踪控制 的内容。

10.34 cmd_translate

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

`translate` 指令用于对位图变换矩阵进行位移为 (tx, ty) 的平移变换。 上述参数的格式为 16.16 定点浮点数，即 65536 表示 1.0 。 方便起见，调用宏函数 `F16()` 可将浮点型数据转换为有符号型 16.16 的表达形式。

Chapter 11

管理指令

11.1 begin

```
void begin();
```

初始化 Gameduino 2 对象。 程序启动后，需在调用其他任何 GD 指令之前执行这一指令。

11.2 finish

```
void finish();
```

将所有待执行的指令发送到GPU，并等待这些指令执行完毕。 参见 finish 。

11.3 flush

```
void flush();
```

将所有待命指令发送到GPU。这条指令只是确保将所有待命指令发送到GPU，而不会等待这些指令执行完毕。参见 flush 。

11.4 get_accel

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

对三轴加速度传感器的数值进行采样。返回值是经过比例校正的，1G的返回值为 256 。加速度轴的默认方向定义如下：x 轴的正方向为右，y 轴的正方向为下，z 轴的正向为指向屏幕内部。

11.5 get_inputs

```
void get_inputs();
```

调用 get_inputs 时，Gameduino会检测并收集所有触摸及传感器输入数据，并存入 GD.inputs 中。这些输入数据包括：

x	触按位置的 x 轴坐标，没有触摸时，值为 -32768
y	触按位置的 y 轴坐标，没有触摸时，值为 -32768
rz	触按压力值，没有触摸时，值为 32767
tag	触摸标签，数值范围为 $0 \sim 255$
tag_x	触摸标签位置的 x 轴坐标
tag_y	触摸标签位置的 y 轴坐标
track_tag	用于跟踪控制的触摸标签
track_value	用于跟踪控制的数值
ptag	软件标签结果

11.6 load

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

从microSD卡中读取名为 filename 的文件内容，并将其加载入GPU中。如果没有找到相应文件将返回一个非零值。而当相应文件正在加载时，可以调用 progress 并以其当前存储位置及文件总体大小为参数。progress 的功能是通过其参数来计算显示加载过程的进程。

11.7 play

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

播放一个乐器音频。其中 instrument 是从设备内置的众多乐器中挑选其一（ p. 68 ）， note 参数选择MIDI音符号（ p. 69 ）。

11.8 self_calibrate

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

运行内置的触摸屏校定程序。

11.9 sample

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

启动回放功能，播放存储在GPU内存中的一段音频采样文件。 start 为音频文件的开始存储位置； len 为音频文件大小，单位为字节； freq 为音频文件采样率，单位为Hz； format 为音频文件的格式，可选值为 LINEAR_SAMPLES ， ULAW_SAMPLES 或 ADPCM_SAMPLES 其中之一。 若将 loop 值设置为1，音频文件进行循环播放。 如需立即停止采样回放，可调用 sample ，并以0作为 len 参数的值。 注意， start 和 len 参数的值应为8的倍数。

11.10 swap

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

完成当前正在操作的图像，并将其替换到前台进行显示。 这一指令必须在图像绘制代码结束后进行调用。

Chapter 12

数学函数

GD程序库的主要功能是作为 Gameduino 2 硬件的一个封装接口，但是除此之外它还包含了一些游戏程序编写中需要的实用函数。这些函数并不对硬件设备进行操作控制——但它们可以实现游戏程序经常需要的基本运算。

这些数学函数谨慎地采用了整形参数、8位代码进行编写，使得用户可以轻松在CPU上实现高速运算。

12.1 atan2

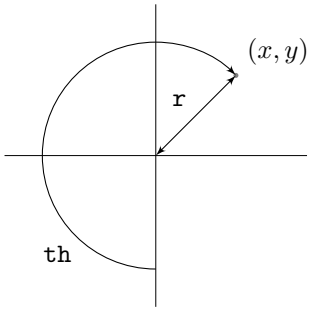
```
uint16_t atan2(int16_t y, int16_t x);
```

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

atan2 函数返回值为从坐标位置 $(0,0)$ 指向坐标位置 (x,y) 的Furmans角度。返回值范围为 $0 \sim 65535$ 。此函数为 polar 的逆运算。

12.2 polar

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



该函数返回以屏幕原点为起点，角度为 th ，距离为 r 的坐标 (x, y)。 th 为Furmans角度。 此函数为 atan2 的逆运算。

12.3 random

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

返回一个随机数。如果没有指定参数，则返回值范围为 $0 \sim 65535$； 若设定了参数 n ，函数将返回一个随机数值 x ，其范围为 $(0 \leq x < \mathrm{n})$ 。 random() 的返回值为一个伪随机数。 这对图像显示及游戏程序设计十分有益，因为性能往往比真正的随机性更重要。

对于提升随机性的传统技巧是在等待用户输入的同时不断调用 random() 。 这意味着当游戏开始时，产生的随机数值更不易于预测。

12.4 rcos

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

该函数将 r 与 th （单位Furmans） 的余弦值进行相乘，并返回计算结果的整数近似值：

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

参见 rsin， polar 。

12.5 rsin

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

该函数将 r 与 th （单位Furmans） 的正弦值进行相乘，并返回计算结果的整数近似值：

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

参见 rcos， polar 。

Part III

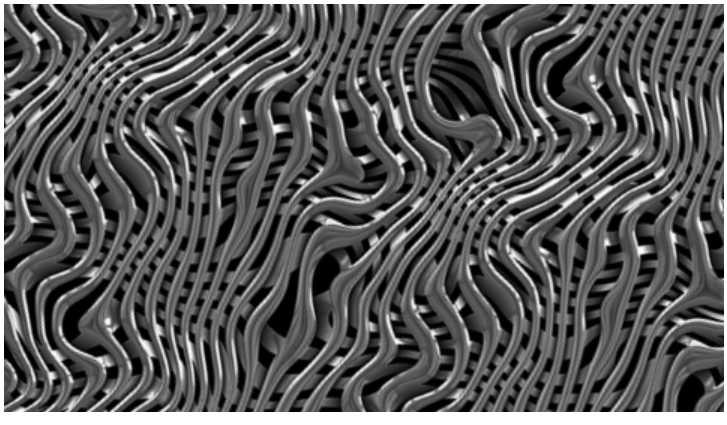
设计实例

Chapter 13

图形元素

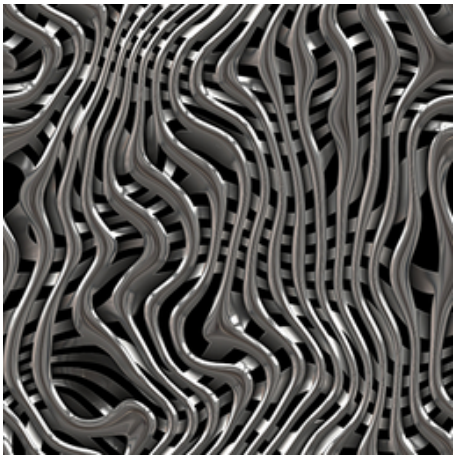
本章主要讲述了如何使用Gameduino 2的图形功能来创造更复杂的游戏特效。

13.1 平铺背景

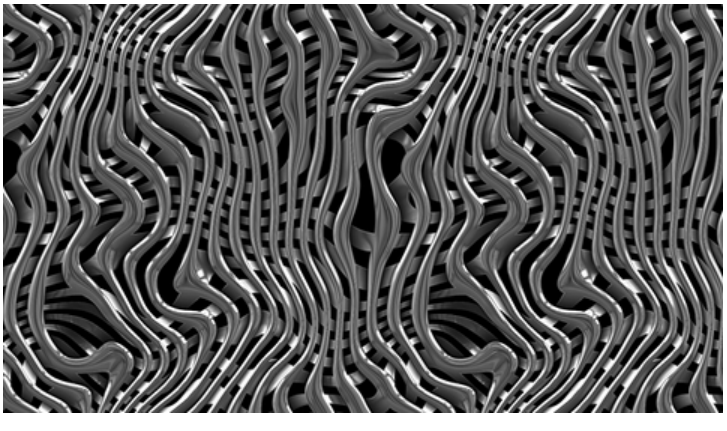


```
#include <EEPROM.h>
#include <SPI.h>
#include <GD2.h>

#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();
}
```

这里所用的原始素材是一个 256 × 256 的无缝纹理, 该素材由 Patrick Hoesly[1]创作。



使用命令 `BitmapSize` 将该图案平铺到整个 480 × 272 大小的屏幕上，可以得到以下结果：



整个结果还算尽人意，不过还是能看出很明显的重复部分。将这个位图使用函数 `cmd_rotate` 旋转一个奇数角度 – 比如3333 Furmans角度，大约为18° – 就可以让重复的部分不显得那么突兀。

[1] 在Flickr网站上还有他成百个作品， 全部基于CC协议。

13.2 绘制阴影

为了实现阴影的效果，需要依次绘制两次图形。首先使用阴影的颜色在目标位置附近（x 和 y 方向上各进行一段偏移）绘制一遍图形，之后使用前景色再次绘制一遍图形。

```
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();
```

这个例子使用了 3 像素的偏移值来绘制阴影，可见这种方法对较大字体比较有效。而对于小一点的字体，1 到 2 像素的偏移看上去会更好一些。

同样的方法也可以用在绘制位图上：首先用黑色将原位图绘制出来，之后用正常的颜色在附近再绘制一次。左图利用这种方法绘制了Gameduino的Logo，采用了50%的透明来起到阴影的效果。

13.3 淡入与淡出

淡入 是一种逐渐改变屏幕颜色的技术。淡入到黑色或者白色是最常见的两种情况，当然淡入到其他颜色也是可能的。 其中一种制造出淡入效果的方法是绘制一个屏幕大小的矩形，通过不断改变矩形的alpha值来实现过渡的效果。 于是Alpha的大小就决定了原图像有多大程度被矩形覆盖了，这样只要逐渐将alpha大小从 0 上升到 255，就可以实现场景过渡的效果。

以下是 *NightStrike* 的一段代码，该代码演示了如何实现将标题画面逐渐过渡到黑色的方法。

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

注意一开始的 TagMask 命令旨在取消写入触摸标签的缓冲。 如果没有这条命令，绘制整个屏幕的矩形将会覆盖所有已存的其他触摸标签。

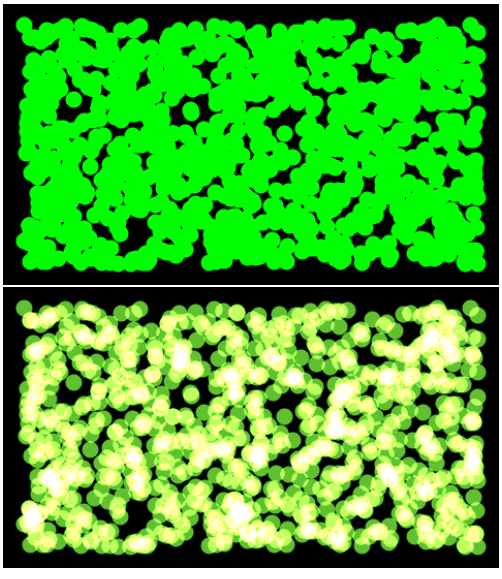
13.4 动态模糊

动态模糊技术被用于平滑物体的运动，从而使物体的运动过程更为自然。图中在棋盘上运动的白色棋子用动态模糊进行了渲染。如果只从单帧上看，物体会显得很不清晰。但如果把帧与帧之间连续起来播放，视觉系统就会把其辨认为一个快速移动的物体。

在 Gameduino 2 中渲染动态模糊效果的方法之一，是在同一时间用多个不同的透明度绘制同一个物体。在这个 “chess” 的演示程序中，用于渲染的数量是由 OVERSAMPLE变量控制的，其值被设置为 8。渲染器按照采样数量同时绘制多次图形，每一个图形之间存在细微的距离差。如此一来，8 个彼此叠加的透明部分就在运动路径上产生了模糊的视觉效果。

```
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 叠加混合

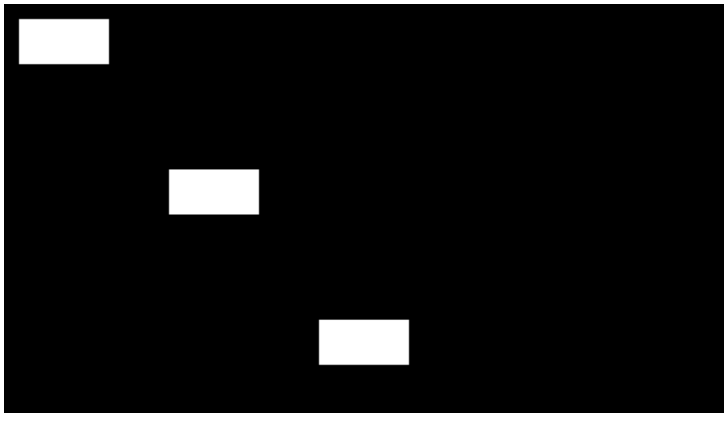


以上两个图片都是由下方的代码完成的。这段代码绘制了 1000 个直径为 8 像素的圆形，在绘制时使用了 BlendFunc 函数。该函数设置了将每个像素的颜色与背景的颜色进行叠加。这两幅图的区别在于，顶上的图片使用的颜色为纯绿色 (0x00ff00) 而底部使用的是一个稍浅的绿色(0x60c030)。

对于纯绿色来说，任何多次绘制的图形依然是绿色的。而对于浅绿色来说 0x60c030 – 包含了绿色，红色和少量蓝色 –，每一次叠加绘制都会让颜色更为明亮一些。不过在这种情况下，同一个像素被绘制 6 次之后就会变成纯白色 0xffffff，此后就无法变得更亮了。

```
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 高效率的矩形绘制



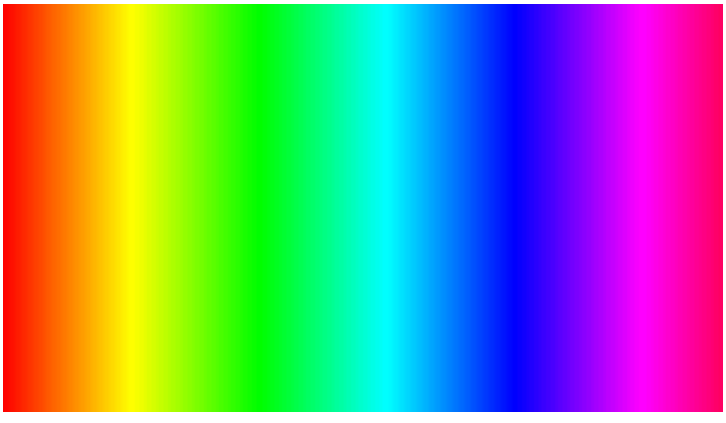
这段代码首先设置了一个大小为 1×1 像素的 L8 位图。混合函数 (ONE, ZERO) 仅仅是一个直接替换操作：像素从位图中拷出，被复制到颜色缓冲中。对一个 L8 位图而言，RGB像素的值就是其当前的颜色。

之后的代码将位图的绘制大小设置为了60×30 像素。这个意味着 60×30 大小的位图都会被当前颜色覆盖。这样的话，每次使用 vertex 绘制的时候都会是一个实心矩形了。

使用这种方式绘制的矩形在多个方面有别于使用 RECTS 绘制。首先，这些矩形没有反走样的功能：它们的大小完完全全是 60×30，而没有任何的边缘平滑处理。第二个不同之处在于，此处每一个 Vertex2ii 都将绘制一个矩形；而在 RECTS 中，每一个矩形需要调用两次vertex。

```
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 一维位图

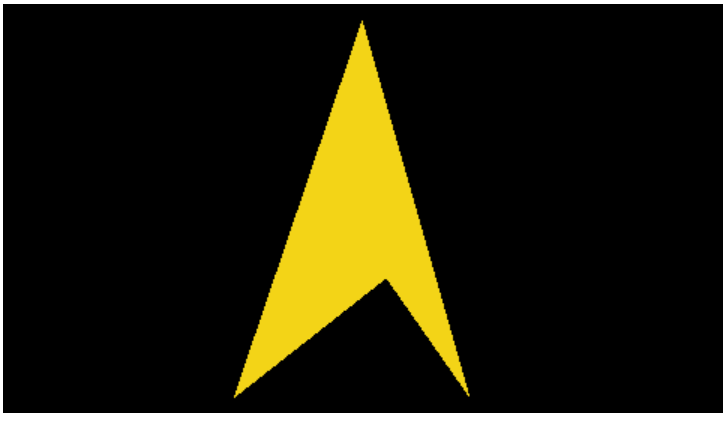


```
GD.BitmapHandle(SPECTRUM_HANDLE);
GD.BitmapSize(NEAREST, REPEAT, REPEAT, 512, 512);
GD.Begin(BITMAPS);
GD.Vertex2ii(0, 0, SPECTRUM_HANDLE);
```

这段代码使用了以下的 512×1 像素的“光谱” 图像。这个图像作为一个RGB565 位图被载入到图形缓存中。

BitmapSize 指令配合 REPEAT 参数可以将位图进行平铺。 由于位图的高度是一个像素，它在每一行都进行了重复。 我们知道对于任何一个一维位图来说，都可以对其进行伸缩和旋转。以下就是同一个位图经过 0.3 倍的伸缩与 80° 旋转之后得到的结果。

13.8 多边形绘制



GD图形库有一个专门用于绘制多边形的辅助对象 *Poly*。如果需要绘制一个多边形，首先需要调用 begin() 方法，之后再将多边形的顶点按照子像素坐标依次传递给 v() 方法。最后使用 draw() 用当前颜色完成多边形的绘制。

例程中只使用了 4 个顶点，但实际上多边形最多可以支持 16 个顶点。多边形可以是任何形状，并且顶点不一定都需要在屏幕可见范围。

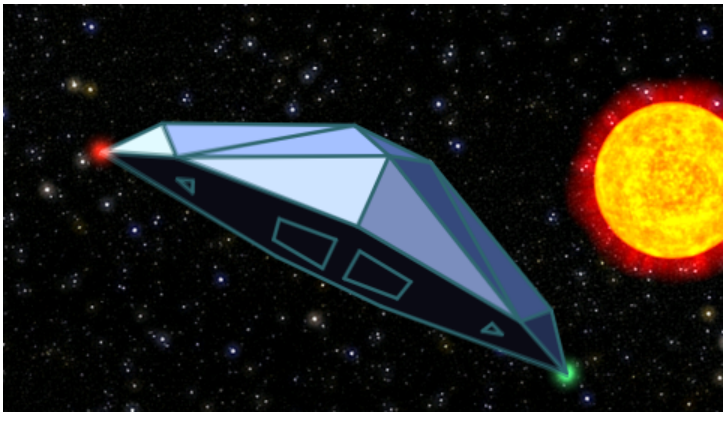
Poly 对象使用了一系列的模板缓冲操作来计算多边形的轮廓。最后它使用当前颜色和Alpha值设置填充多边形的内部，整个过程和 POINTS 或者 LINES 十分类似。

```
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();
```

但是和 POINTS 或者 LINES 方法不同，多边形并没有对边缘进行平滑处理。这导致了多边形的边缘上会出现锯齿。解决方法之一就是在多边形的边缘上再次勾勒直线边缘。此处利用Poly 对象的方法 outline()，一个宽黑色边缘通过 LINE_STRIP 方法绘制了出来。

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

例程 *cobra* 使用了类似的方法，首先用 Poly 绘制了多边形，之后又勾勒了微微发光的蓝色外边界。

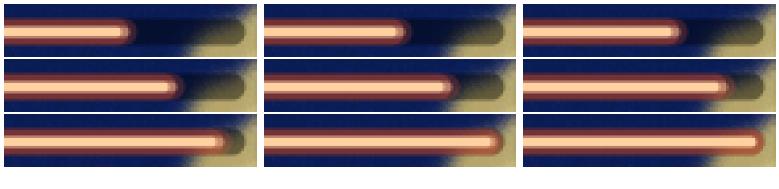


13.9 无所不在的直线

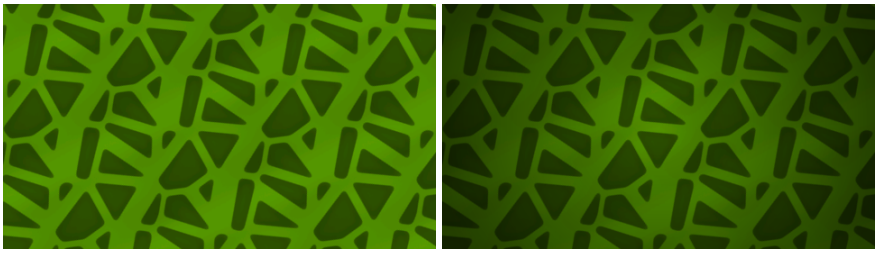
因为直线顶端有宽圆角的存在，宽直线可以作为非常实用的图形元素存在。在例程 *sprites* 中，一个宽 28 像素、Alpha 50%的黑色宽直线就被用于绘制了文本框的背景。


```
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);
```

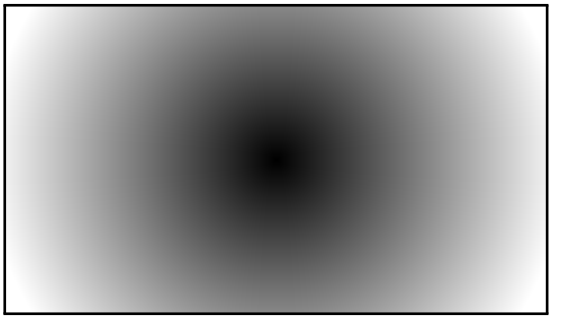
NightStrike 程序中在画面顶端使用了三个水平放置的宽直线来绘制能量槽。首先一个 10 像素的浅黑色长条作为能量条的背景，而当前能量的指示由两条直线配合完成：一条为 10 像素的淡橘色，另一条为 4 像素的亮橙色。这样配合下来就得到了令人满意的发光特效。



13.10 晕映(vignetting)



晕映是在摄影中一种将图片边角暗化的技术。上方右图中就将晕映作为了一个额外的图层。晕映图片本身是一个 480×272 像素的 L8 放射渐变图形，越至边缘透明度越低：

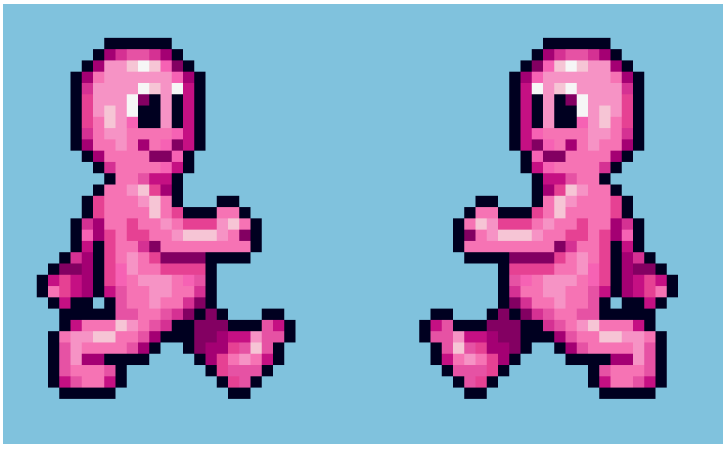


如果用白色绘制这个位图，边缘会得到乳白色的透明效果。而用黑色绘制该位图，则会让边缘更加暗化。暗化的程度即可以通过修改原位图控制，也可通过 ColorA 方法改变位图的不透明度。

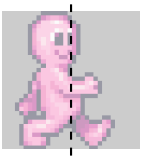
```
GD.ColorA(0x90);
GD.ColorRGB(0x000000);
GD.Vertex2ii(0, 0, VIGNETTE_HANDLE, 0);
```

因为晕映图像自身并没有太多细节，所以降低分辨率也不会对图像质量造成很多的影响。比如一个 240×136 像素的晕映位图和一个 480×272 像素的位图相比并没有太多可观的差别，但是前者只占用了后者四分之一的内存。

13.11 镜像的图元

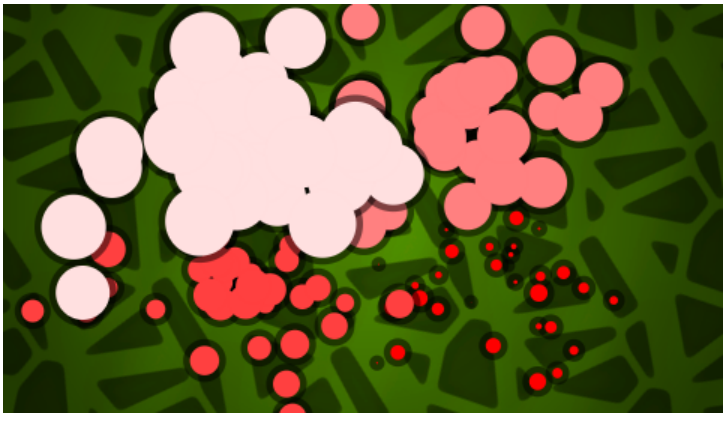


镜像 一个图元，把它调整它的镜像方向上，是在2D游戏中十分有用的操作。 Gameduino 2的GPU单元可以处理复杂的位图变换，自然简单的翻转更不会存在问题。 你可以设想一个左右的翻转实际上就是x轴被乘以了-1倍，该操作用 `cmd_scale` 就可以完成。 代码中两次 `cmd_translate` 调用仅仅是修改了位图的位置，以保证翻转的中心轴在图元的中心处。 在这个例子中，图元是 32 像素宽的，所以翻转是在 $x = 16$ 处完成的：



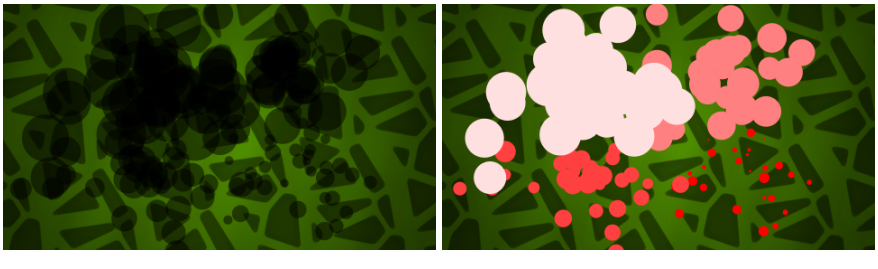
```
GD.Begin(BITMAPS);
GD.Vertex2ii(  0, 10, WALK_HANDLE, 0);
GD.cmd_translate(F16(16), F16(0));
GD.cmd_scale(F16(-1), F16(1));
GD.cmd_translate(F16(-16), F16(0));
GD.cmd_setmatrix();
GD.Vertex2ii( 30, 10, WALK_HANDLE, 0);
GD.swap();
```

13.12 轮廓与边缘



Gameduino 2的硬件底层只支持实心圆和直线的绘制。那么如何实现上图的效果：在圆的周围勾勒出线条呢？答案非常简单 – 绘制圆形两次。

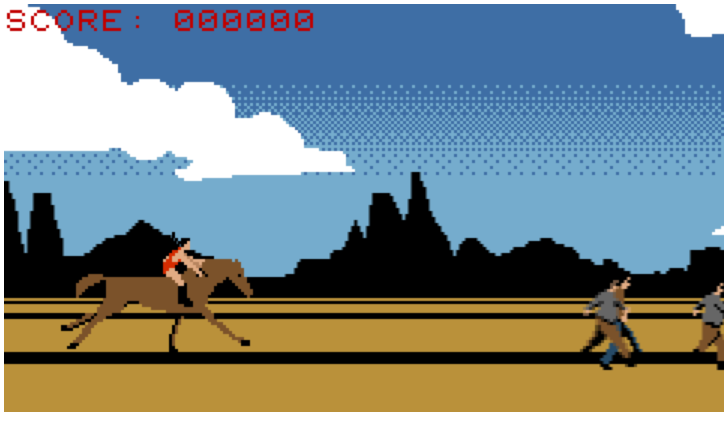
一个稍大的圆首先用边界的颜色绘制出来，之后稍小一点的圆再使用圆内部的颜色绘制。对于一群绘制的物体来说，绘制顺序将影响最终的图像。此处外轮廓的圆先绘制，这样可以给整个集群一个共有的边界。



下图中按钮的边缘效果也是依靠两个直线绘制的 – 一个白色直线和一个比它窄 4 个像素的蓝色直线。

这种边缘绘制方法也可以用于 POINTS， LINES 和 RECTS 中。

13.13 粗糙的像素感

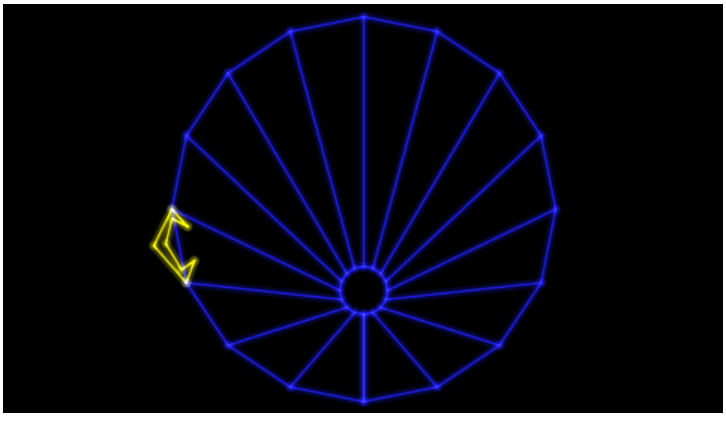


对于艺术家和设计师 Nick Criscuolo 来说，他更希望用Gameduino 2的图形系统实现更加粗糙、8位像素的复古感。在 *Zardoz* 项目中，所有的图形都是以两倍伸展的方式绘制的。在每一帧开始之前，都会调用以下代码来把其绘制的每个位图扩展两倍：

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

这样一来的结果就是原图中的每个像素都变成了屏幕上 2×2 个像素。这导致了游戏的等效分辨率变为了 240×136 (Gameduino 2 的原始分辨率为 480×136)。

13.14 矢量图形

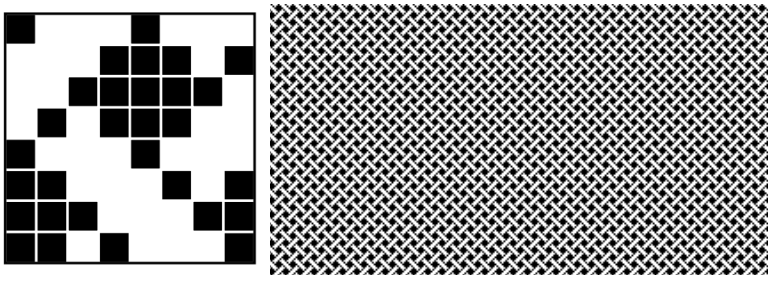


矢量图形游戏在70、80和90年代期间，因为当时特殊的硬件条件，可以在CRT显示器上绘制出明亮、彩色的矢量线条。 作为一个图形技术，这种绘制方式的优势在于不需要像素的概念：所有的一切都是由平滑明亮的直线和点构建而成的。 Gameduino 2 有非常完善的平滑线条和点的绘制支持，所以可以轻易模拟出这种矢量绘图方法。

然而如果仅仅用直线来绘制游戏，恐怕看上去太“单薄”了：还需要模仿原来图形上微微发亮的效果。这个特效将分成两步实现。首先编写 drawgame() 函数，其作用是绘制屏幕上所有的直线，之后将使用两种不同的线条属性来调用该函数进行绘制。 第一步的绘制中，配置直线为稍宽的尺寸，使用背景亮光的颜色来绘制所有直线，之后调用 drawgame()。 第二步绘制的直线位置完全相同，但是在样式上使用了更为明亮，且线宽较细的直线。 这两步绘制之后，所有的直线都实现了微光的效果。

```
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 手绘图形



早期人们并没有像我们一样复杂的设计工具，所有的图形都是首先绘于纸稿之上，最后再徒手转换成数据的。如今，你也可以试着用同样的方法设计图形。 cmd_memwrite 函数将 8 个字节构成的图案写入在起始地址为 0 的内存地址之中。二进制编码的图案被存放在 picture[] 中 – "0b" 前缀表示数据为二进制表示。之后，这个程序将这个 8×8 L1 格式的位图在整个屏幕的 x 和 y 上平铺。

```
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

图形混合

本章介绍了更为复杂的图形技术，挖掘GPU硬件的潜力。他们无一例外的使用了合成技术 – 使用Alpha通道控制颜色通道。

14.1 Alpha合成

这个有型的时钟是由一个白色圆圈，一个小的黑色圆圈和一个 cmd_clock 控件绘制的时针组成的。

```
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);
```

但是如果你想为其添加一个JPEG格式的背景的的话，结果就却是悲剧的：

可见，黑色的圆盘毁掉了画面整体的设计感。如果能将中心圆盘变为透明色，整个画面感就能大幅提升。这里介绍一种解决之道：使用*alpha*缓冲。

屏幕上每个像素都有一个0-255的alpha值，这个alpha值通常和颜色缓冲同时更新。一般来说alpha缓冲的存在并非必要，因为它并不可见，而且不会影响最终图像。而 ColorMask 命令可以允许或禁止对颜色缓冲和alpha缓冲的写入。这个技术首先将图形"绘制"至alpha缓冲中，之后使用 BlendFunc(DST_ALPHA, ONE) 实现对屏幕上所有像素的混合模式绘制。在绘制时，只有alpha缓冲非零时才会被绘制出来。

绘制完绿树的背景之后，首先用 ColorMask 函数禁止对颜色缓冲写入，之后向 BlendFunc 写入一个可以把之后的alpha值直接写入alpha缓冲的参数。设置方法是将源混合因子设置为 ONE:

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

现在和之前一样绘制出外部圆形：

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

这样操作之后，并没有图形真正显示在屏幕上。因为上述操作实际上只会影响alpha缓冲，目前为止还是不可见的。绘制内部圆时需要将混合模式设置为覆盖之前已经绘制的所有像素，即将源混合因子设置为 ZERO：

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

最后时钟控件的本体使用 ONE 的混合因子进行了绘制：

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

经过以上的操作之后，目前还没有任何内容能在屏幕上可见 – 因为 ColorMask 命令静止了对R，G，B通道的绘制。但是在alpha缓冲上，已经有了以下的图像：

最后一步就是让整个alpha缓冲变为可见，代码通过绘制一个整屏的灰色矩形实现。当然，如果仅仅简单这么做自然不行，不同之处在于此处将源混合因子设置为了 DST_ALPHA，这样一来，透明值的大小就来自于alpha缓冲了。

```
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);
```

完整的代码如下：

```
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 是一个非常古老用于将两个元素合成在一起的动画技术[1]。在这个slot gag的例子中，我们将Gameduino的位图logo的alpha缓冲设为了 255，而将其他地方设置为0。之后设置一个宽的对角线，并且将 BlendFunc 的参数设置为 (DST_ALPHA, ONE)。如此一来，alpha缓冲就会作为蒙板来绘制直线，这样只有在alpha缓冲为255 的地方才会绘制像素。最终的效果就是得到一个如图的闪光动画效果。

```
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;
```

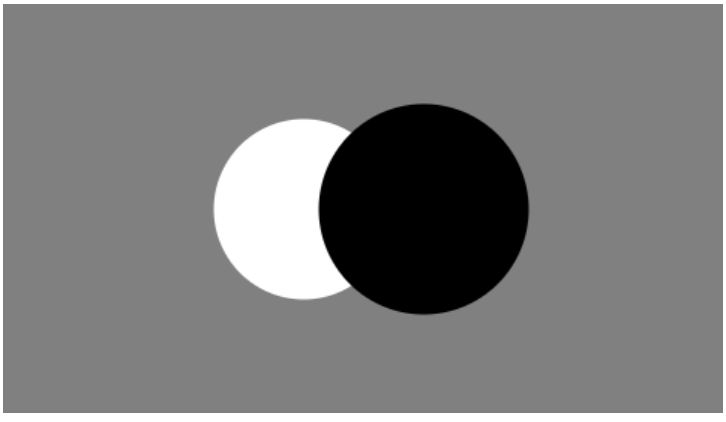
[1] 见 Steve Wright 的 “Digital Compositing for Film and Video” 一书。

14.3 有图案的文本

有图案的文本使用了和之前slot gags相似的技术。首先，文本仅在alpha缓冲中绘制。之后一个 8×8 的位图图案在整个屏幕上绘制出来，绘制的不透明度由aplha缓冲决定。

```
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操作



你可以设置一个宏操作让冗长的alpha合成函数的可读性更好。此处，PAINT_ALPHA() 表示“在alpha缓冲中绘制”；而 CLEAR_ALPHA() 则表示“从alpha缓冲中删除”。

```
#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 圆角图片

上方左图是一个使用 Vertex2ii 绘制的 128×128 位图。右边的图在左图的基础上增加了漂亮的圆角外边。绘制这种图形的技巧在于，首先在alpha缓冲中绘制一个圆角矩形，之后利用这个矩形来控制位图的透明度，依然使用 BlendFunc 函数。

圆角半径 r 控制了圆角的圆滑度，单位为像素。r 设置为1是为方形，数值越大，圆角度也会越大。

```
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 透明的按钮

这段代码首先绘制了全屏大小的背景图片，然后所有的按钮首先通过函数button() 绘制在了alpha缓冲中。而代码的最后六行让alpha通道变为可见。

```
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);
```

button() 函数首先绘制了一个白色矩形，然后是略小透明的另一个矩形。最后使用 cmd_number 命令显示数字值，OPT_CENTER 参数将文本设置为居中显示。

```
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 反射

首先绘制好logo的位图，之后程序在alpha缓冲中绘制了一个 128×1 的渐变位图，该渐变在水平方向上重复，直到全部覆盖屏幕：

之后的代码利用位图变换将logo在Y轴方向上进行了翻转，所用的技术和第 154 页：镜像的图元中使用的类似。之后的代码将logo的alpha通道连同现有的alpha内容全部写入到alpha缓冲中。

蒙版的操作使用了宏 `MASK_ALPHA()`，该宏的作用是将logo的alpha通道和现在的alpha缓冲值进行乘法操作：

最后一步是将镜像后的logo通过alpha缓冲绘制出来。

```
#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

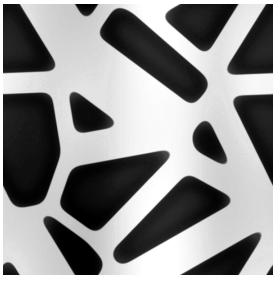
内存节约

15.1 双色位图

```
#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();
}
```

此处所用的源图像来自于 Patrick Hoesly 设计的无缝纹理：

一般情况下，这样的图片会用 RGB565 编码。然而此处因为这张图片只使用了两种颜色，经过适当处理就可以使用 L4 位图格式来存储，如此一来可以节约 75% 的图形内存（具体图片格式与对应的内存占用情况请参见命令 BitmapLayout）。首先将原始图片载入到图片处理软件中，用色彩取样工具分别得到两个颜色的具体RGB数值。此处经过测量，深绿色为 0x375e03，而浅绿色为 0x68b203。之后将位图转换为单色位图，并调整对比度，使其在纯黑与纯白之间自然过渡。

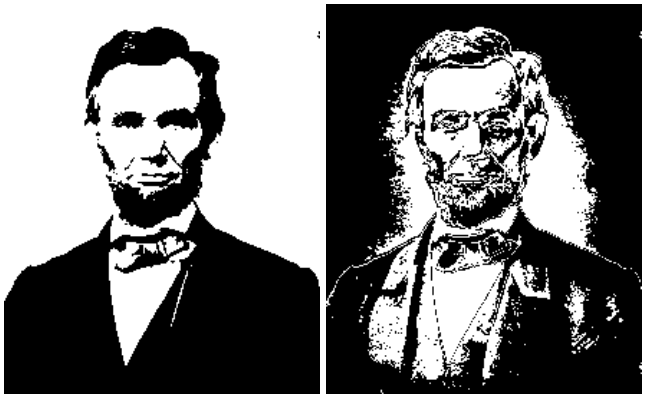


该程序的代码首先将屏幕清除成深绿色，之后将 L4 的位图用浅绿的颜色绘制出来。所得到的结果和最初的图片没有太大区别，但却节约了大量的内存。这个 128×128 的 L4 位图只是用了KB的内存。

除此之外，你还可以在程序运行阶段随意调整前景色和背景色，完全按照你自己的审美标准来就行了。

15.2 潜藏的 L2 格式

Gameduino 2的GPU支持三种单色格式： L1， L4和 L8。 这里虽然没有 L2 格式，但是通过两次绘制却可以仿真出来。 此处的技巧就是将原图片转换成两个1位 L1 位图，其中一个存储低位，另一个存储高位：



以上名为 ABE 图片被载入到了位图单元0和1中，位图的句柄为 ABE_HANDLE。位图单元0中存储位0，位图单元1中存储位1。 代码总共绘制了三次图片。前两次分别将两个位图载入到alpha缓冲中，而第三次操作将alpha缓冲变为可见。

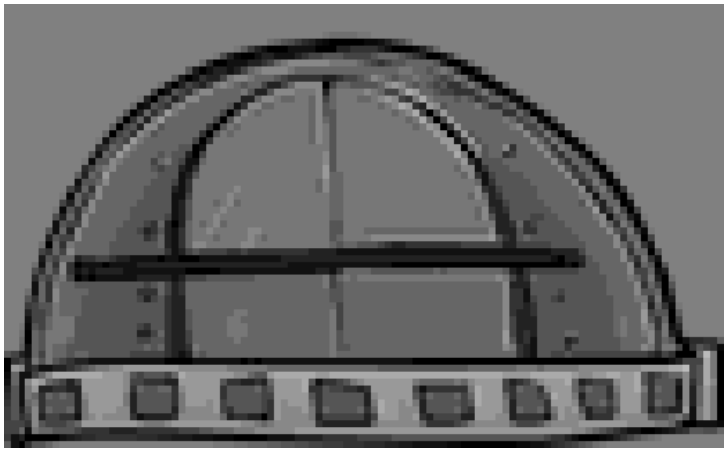
绘制代码首先使用 ColorMask 禁用了颜色缓冲的写入。之后将高位用alpha值 0xaa 表示，而低位则用 0x55 表示。这就得到了alpha值和2位像素之间的转换关系：

高位	低位	alpha值
0	0	0x00
0	1	0x55
1	0	0xaa
1	1	0xff

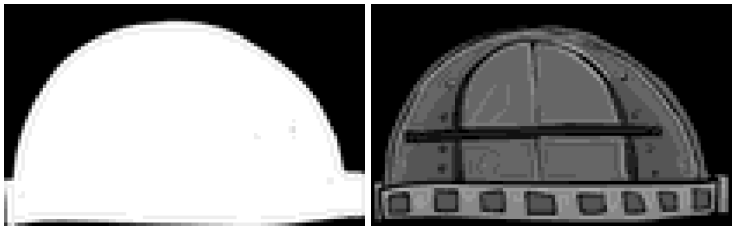
```
GD.ClearColorRGB(0x00324d);
GD.Clear();

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();
```

最后的 Vertex2ii 和之前两个绘制指令绘制的图形完全相同，但是由于 BlendFunc 的参数设置成为了(DST_ALPHA, ONE_MINUS_DST_ALPHA)，图片的原始像素全部被忽略了，取而代之的仅仅是alpha缓冲中的像素会影响最终的颜色内容。

15.3 分离的形板(matte)



NightStrike 游戏中所使用的基座是一个包含alpha通道的单色位图。硬件系统中并没有办法直接支持单色含alpha通道的位图格式，但是可以通过将原始位图分离成一个 形板*(matte)* 来解决。其中一个是原来的alpha通道，另一个是前景图片。

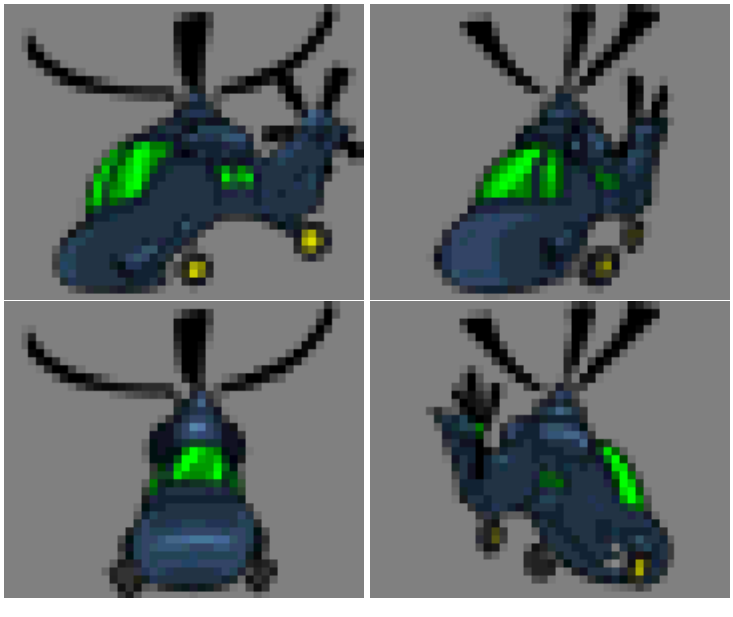


以上两个位图都以 L4 格式被载入系统/ 形板位图首先被绘制出来，绘制时将 ColorRGB 设置为黑色。这个操作将所有被形板覆盖的区域清空，之后前景图片按照通常的方法被绘制。

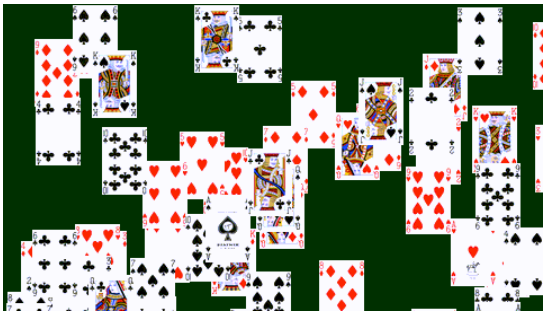
15.4 减半分辨率

在 *NightStrike* 游戏中，直升机的动画使用了两帧100像素宽的 RGBA4 图片：

而直升机从空中坠落到地面的过程则使用了另外一个四帧的动画。因为下落的过程很快，这个动画只是转瞬即逝，一些细节上的损失并不会引起用户的注意。所以在这个动画中使用的分辨率只有之前的一半：小于50像素。在屏幕上显示之前，只要用 `cmd_scale` 将该位图缩放到同样的尺寸就可以了。使用这种方法减半分辨率，可以节约75%的内存使用量。

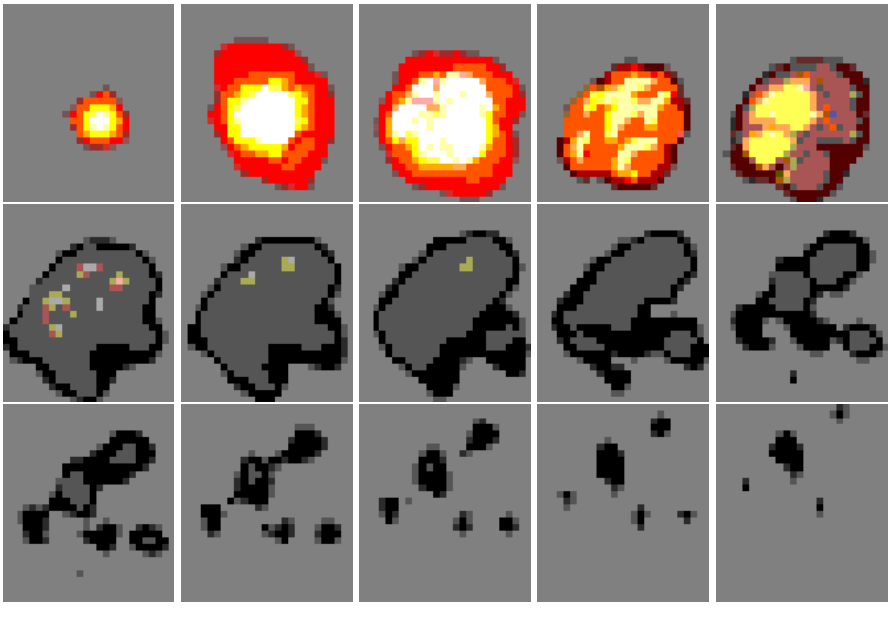


15.5 8位图片格式



对于大部分图片来说，16位的位图格式如 RGB565， ARGB1555 和 ARGB4 才是合适的。但有时8位位图可以替代它们，在起到同样的效果的情况下不会产生明显的失真。

上图中的纸牌就是用 RGB332 格式编码的，每个像素仅占用1个字节（8位）。而下图中的爆炸动画则使用了 ARGB2 格式。对于纸牌来说，其用了大胆鲜明的颜色搭配，所以将颜色下降到 RGB332 并不会产生太大变化；而对于爆炸动画来说，因为整个过程只有0.25秒的时间，快速的动作掩饰掉了颜色范围上的损失。



15.6 DXT1

DXT 是一个贴图压缩系统，它将较大的图片压缩成为更小的位图从而节约图形内存。左方的截图是由原艺术家设计、渲染的 *NightStrike* 欢迎界面，而在右边是在Gameduino2上实际运行着的，并经过DXT1压缩后的版本。两幅图片用肉眼难分差别，但是用DXT1压缩后的图片只使用了原始大小四分之一的内存。

总体上来说，DXT1的工作原理是将原始图片分割成 4×4 像素的区域。每一个区域被赋予两种颜色，名曰 C_0 和 C_1。之后区域内的每个像素都被表示成 C_0 和 C_1 的混合值。DXT1 使用 RGB565 格式表示颜色 C_0 和 C_1，而对每个像素使用两位的编码：

编码	颜色
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

如此一来，每个 4×4 单元所占用的位大小就是：

C_0	16
C_1	16
像素点	$16 \times 2 = 32$

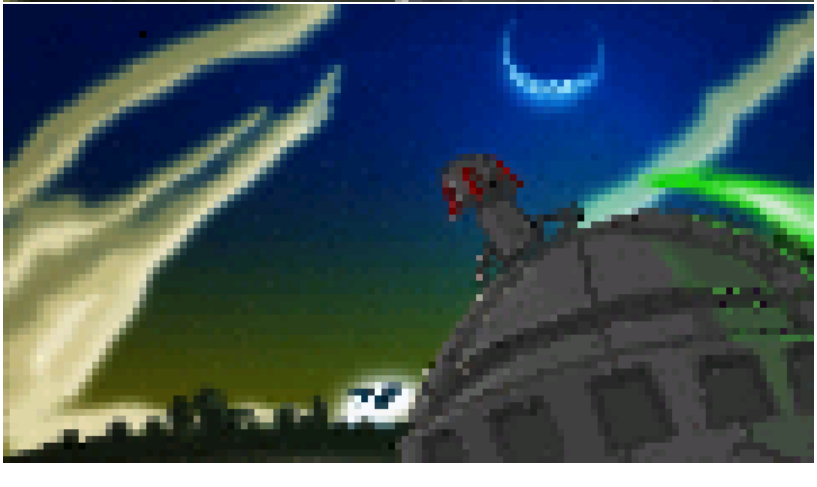
计算得知每一个区域单元占用的大小为64位，因为一个单元中有16个像素点，平均下来每一个像素点所占用的内存仅为4位。设想如果这个位图是用 RGB565 编码的，那么每个像素将占用16位，用DXT1压缩可以节约75%的图形内存！

可惜的是Gameduino 2的硬件并不直接支持DXT1，但是通过多次绘制和相乘的方法可以将其模拟出来。

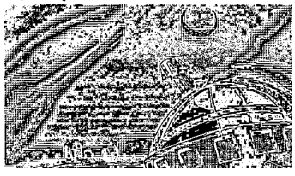
以 *NightStrike* 的欢迎界面为例，源图片的大小为 480×272 像素。因为DXT1以 4×4 大小为一个单元，于是整个图像为 120×68 单元。将所有单元的 C_0 和 C_1 分别保存在两个大小为 120×68 像素、编码为RGB565的图片中：

$C_0 =$

$C_1 =$



因为硬件缺乏对两位图片的支持，两位的编码位图被拆分为两张一位编码的图片，具体作法见 第 176 页：潜藏的 L2 格式 。

$b_0 =$  $b_1 =$ 

NightStrike DXT1 的背景绘制函数首先在alpha缓冲中构建了二位的位图，之后再将 C_0 和 C_1位图使用 `cmd_scale` 命令放大4位并写入颜色缓冲中，最终将其与alpha缓冲的内容进行了混合。

```
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

游戏及演示程序

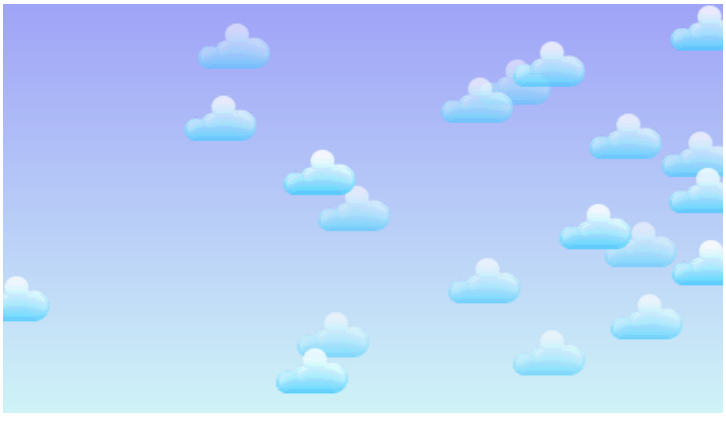
16.1 Kenney

kenney 演示程序使用了艺术家 Kenney.nl [1] 的开放艺术作品[2]。整个程序绘制了六个图层，可以以60Hz的速度流畅运行。其中大部分的图形都是以位图的方式绘制的，所有的位图都是以 ARGB4 编码的。第一层是一个垂直方向的渐变，精心挑选了天空蓝作为颜色。

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

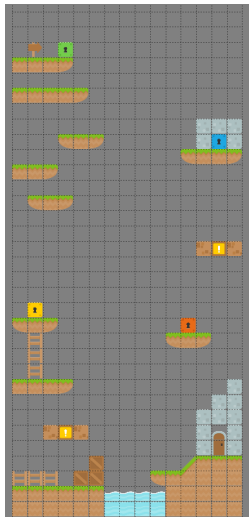
[1] http://www.kenney.nl/
[2] http://opengameart.org/content/platformer-art-deluxe

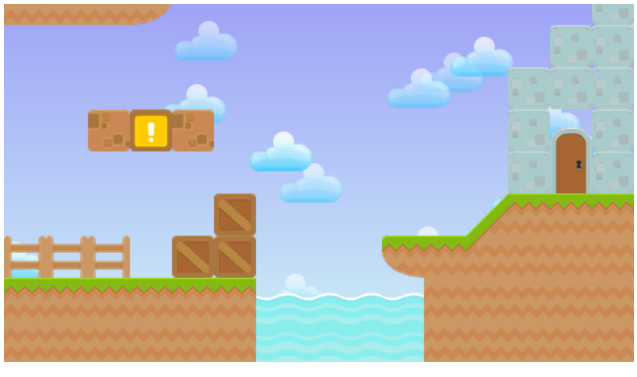
下一个图层使用了一个云朵图元，这个图元被保存在 CLOUD_HANDLE 中。这些云朵的初始位置是随机的，位置信息保存在 state.clouds[] 中。所有的云朵以不同的速度向下方运动，速度越慢的云朵越透明。当云朵到达屏幕顶部之后，它们重新从屏幕上方卷动出现。云朵的位置是以1/16像素为单位的子像素坐标表示的，使用精确的位置可以保证云朵不会在像素之间“跳动”。

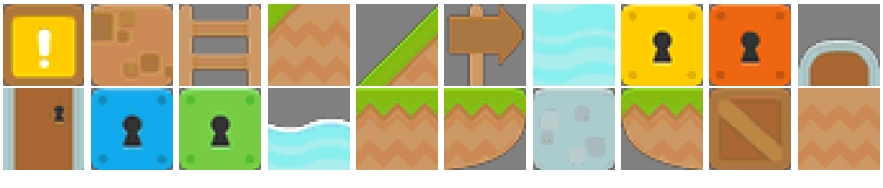


```
GD.Begin(BITMAPS);
GD.BlendFunc(ONE, ONE_MINUS_SRC_ALPHA);
GD.BitmapHandle(CLOUD_HANDLE);
GD.Cell(0);
for (int i = 0; i < 20; i++) {
  byte lum = 128 + 5 * i;
  GD.ColorA(lum);
  GD.ColorRGB(lum, lum, lum);
  GD.Vertex2f(state.clouds[i].x, state.clouds[i].y);
  state.clouds[i].y += (4 + (i >> 3));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

下一个图层是网格地图，网格地图是由网格编辑器生成的 (http://www.mapeditor.org/)。这个地图之后通过元素转换器被写入字符数组 layer1_map。20个平铺网格单元每一个都是 32×32 ARGB4 编码的位图，句柄保存在 TILES_HANDLE。平铺单元位置0处是一个空白单元，程序会检查这一点并且只绘制非0单元。



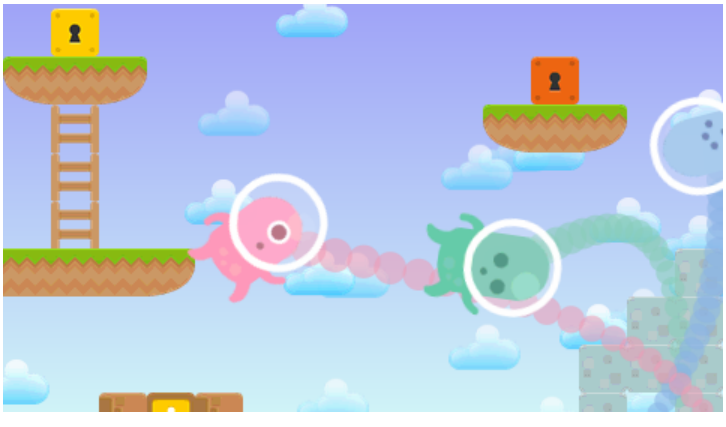




```
GD.Begin(BITMAPS);
GD.BitmapHandle(TILES_HANDLE);
const PROGMEM prog_uchar *src = layer1_map + (y >> 5) * 15;
byte yo = y & 31;
for (byte j = 0; j < 10; j++)
  for (byte i = 0; i < 15; i++) {
    byte t = pgm_read_byte_near(src++);
    if (t != 0) {
      GD.Cell(t - 1);
      GD.Vertex2f(16 * 32 * i, 16 * ((32 * j) - yo));
    }
  }
```

在前景中的角色是 ARGB4 的位图，以正弦路径运动并随机旋转。它们的坐标值被保存在 state.p 中。

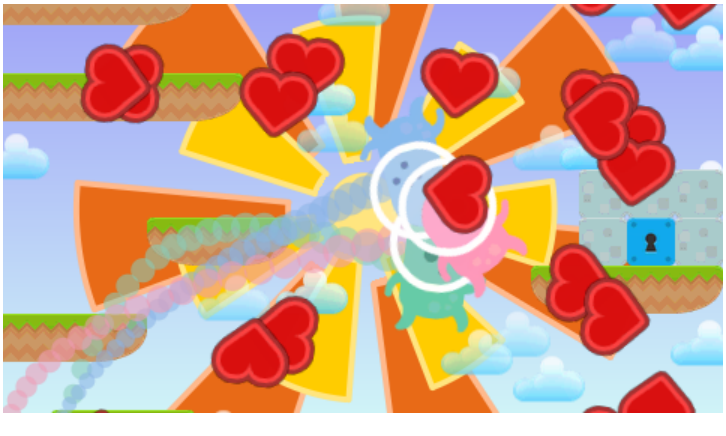
跟随角色的彩色“泡泡”是透明的 POINT，在代码中由三个 state.trail 对象管理。



```
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));
}
```

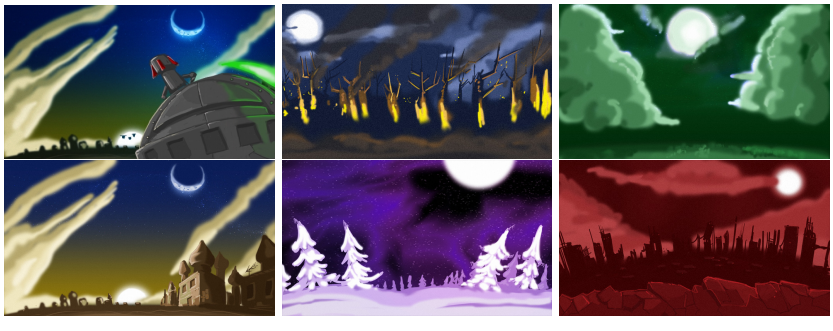
“阳光普照”的图形是在云朵图层和天空的图层之间绘制的。通过使用一系列蒙版操作实现了射线状物体的绘制。函数 burst() 绘制了整个阳光普照，而函数 sunrise() 绘制了黄色和橘色的 burst()，并且将它们跳跃的扩展做成动画效果。

最后一个图层是从天而降的爱心雨，这部分的处理方法和处理云所在的图层基本相似。数组 state.hearts[] 记录爱心的位置，当它们落入到屏幕底部时，会从顶部卷动出现。



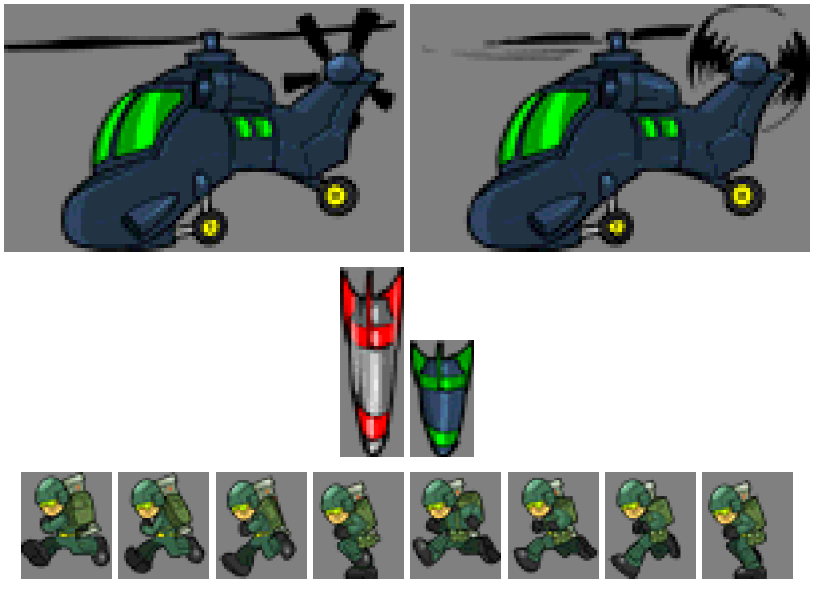
16.2 NightStrike

NightStrike 的游戏素材来自艺术家 MindChamber 的开放艺术作品[3]。这个游戏使用触屏操作追踪大量的移动物体，游戏可以流畅运行在60Hz。游戏运用了很多技巧使其适应图形系统仅有的256 KB内存。整个游戏共有五个关卡，每一个关卡都有一个独立的元素文件。这个文件保存了对应关卡所有需要的前景、背景图片以及音效内容。每关卡的背景都是 480×272 的图片，使用类似DXT1的压缩编码 (第 181 页：*DXT1*)，使其仅占用65 KB的图形缓存。

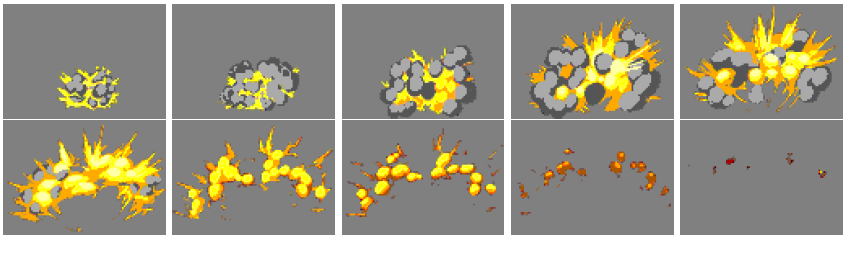


[3] http://opengameart.org/content/nightstrike-png-assets

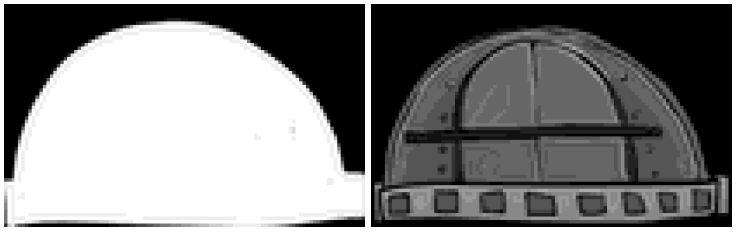
游戏中的图元有很平滑的透明处理，所以大部分都是以 ARGB4 格式编码的。



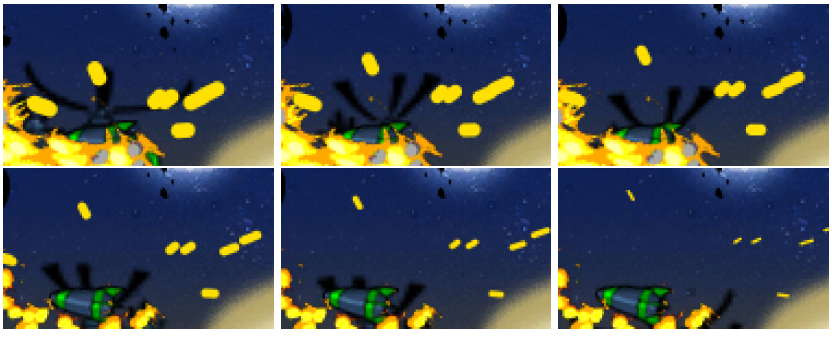
爆炸的动画快速且有明亮的颜色，所以可以使用 ARGB2 格式节约内存 (见第 180 页：*8*位图片格式)。



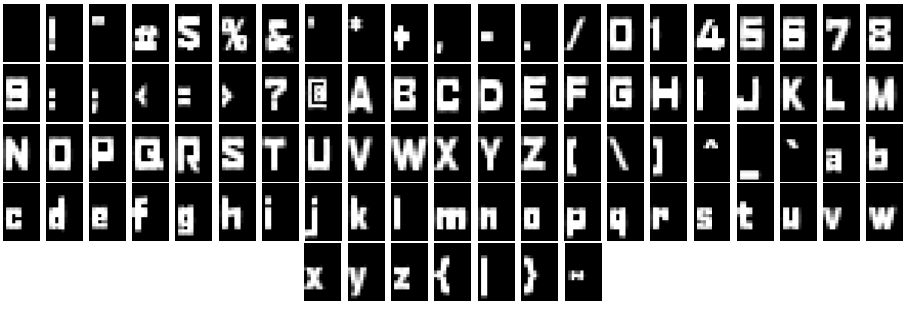
玩家的“炮台”是黑白的，所以用了两个 L4 位图存储 (见第 178 页：分离的形板*(matte)*)。



除了使用运动的位图来实现爆炸效果，游戏还使用了宽线条来模拟爆炸的火花 (见第 48 页：直线)。因为它们仅仅使用 LINES 功能，所以这些火花不需要使用任何图形缓存。



游戏中的所有文字使用了12×19像素的字体，该字体库由原始的TrueType字体转换而来。字体的位图保存为 L4 格式从而保存平滑的边缘过渡。



游戏代码中对每一个可见物体设立了一个类对象进行管理，这些类包括：Fires, Explosions, Sparks, SoldierObject, MissileObject, Rewards, BaseObject, 以及 HeliObject。每一个对象有一个 draw() 方法将物体在屏幕上绘制出来，另有一个 update() 方法用于实现动画、移动以及处理碰撞。物体的碰撞处理仅仅使用了矩形测试法。*NightStrike* 使用了Arduino 32KB Flash中的20 KB，整个游戏在标准16MHz 的Arduino上运行的周期约为7 ms。

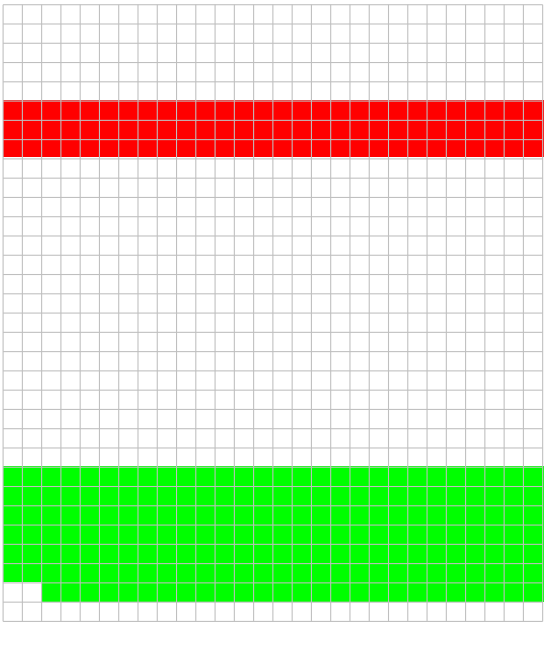
16.3 Invaders

Invaders 是从 Gameduino 1 中移植的版本[4]。为了让其更为接近原版，它使用了街机版本中的艺术手段。传统的70年代游戏机只有一个单色的CRT显示器，为了让它炫酷起来，设计师绞尽脑汁用了一个反射镜将一张太空图片投影到了图形的后方。就是这样因为这个原因，当你离近些看时，就会发现像素变得有些透明。他们用的另一个小技巧就是在CRT屏幕的底部覆盖了一个绿色的矩形薄膜，并且在顶部覆盖了红色的薄膜。在原始版本的游戏中，你可以发现导弹在通过这些区域时改变了颜色。

Gameduino 2 *invaders* 读取了一个 248×272 的JPEG图片作为背景，该图片仅占6 KB内存，所以可以轻易装入Arduino的Flash存储中。首先对全亮度的图像进行编码，之后在游戏中再绘制一个稍暗的版本，这样可以适当掩藏JPEG压缩带来的不自然。

游戏的像素动作首先都是绘制在alpha缓冲中的，之后用覆盖位图让其可见。覆盖位图是 28×32 的位图，被放大8倍到 248×256 后用 `BlendFunc(DST_ALPHA, ONE_MINUS_DST_ALPHA)`绘制。如此一来就绘制了游戏的像素，而颜色全部来自于覆盖图层。

[4] `http://artlum.com/gameduino/gameduino.html`



Appendix A

对象转换器

最新的 Gameduino 2 对象转换器可以在 http://gameduino.com/code 进行下载。相关的文档与帮助详见官方网站。