Arduino-Retro-Spielkonsole

Ein günstiger Mini-Bildschirm, zwei Drucktaster und ein analoger Joystick verwandeln einen Arduino Uno im Handumdrehen in eine Spielkonsole. Mit der macht nicht nur das Programmieren Spaß.

von Maik Schmidt

Die massenhafte Produktion von MP3-Spielern, Smartphones und Videospiel-Konsolen ist in mancher Hinsicht ein wahrer Segen für Hobby-Elektroniker. Plötzlich gibt es viele Bauteile, die früher den Profis vorbehalten waren, zu Schnäppchenpreisen an fast jeder Ecke. Das gilt insbesondere für Displays aller Art und so einige davon eignen sich hervorragend für den Einsatz am Arduino.

Zum Beispiel gibt es kostengünstige OLED-Displays, die nur wenig Strom verbrauchen und sich über gerade einmal zwei Pins kontrollieren lassen. Was liegt da näher, als noch einen Joystick und eine Handvoll Drucktaster aus der Werkzeugkiste zu kramen und den Traum von der eigenen Videospiel-Konsole endlich mal Wirklichkeit werden zu lassen?

Dieser Artikel zeigt, wie sich ein solches Projekt mit wenig Aufwand umsetzen lässt. Er zeigt auch, dass knapp Tausend Pixel durchaus genug für unterhaltsame Action-Spiele sind und er erklärt Schritt für Schritt, wie die eingesetzte Hardware funktioniert und wie sie durch ein wenig Software optimal genutzt wird.

Ein Exklusiv-Titel inklusive

Wie es sich gehört, liegt der Mini-Spielkonsole ein erstes Spiel namens Shootduino bei. Der Protagonist in Shootduino ist ein Raumschiff, das der Spieler über den gesamten Bildschirm bewegen kann. Ein Sternenfeld im Hintergrund erweckt den Eindruck, das Raumschiff bewege sich permanent von links nach rechts. Am rechten Bildschirmrand tauchen immer wieder Asteroiden auf, die sich in unterschiedlichen Geschwindigkeiten auf den Spieler zu bewegen.

Ziel des Spiels ist es, so viele Asteroiden wie möglich mit einer Laserkanone zu eliminieren. Dazu hat der Spieler drei Versuche. Sind diese aufgebraucht, ist das Spiel zu Ende. Das Spiel endet allerdings auch, wenn der Spieler mindestens fünf Asteroiden nicht zerstört.

Ein Kilopixel

Für das Projekt kommt ein OLED-Display mit einer Bildschirmdiagonale von 0,96 Zoll und einer Auflösung von 128 × 64 Pixeln zum Einsatz. Das entspricht in etwa dem, was in den Nullerjahren des 21. Jahrhunderts in Handys und MP3-Spielern verbaut wurde. Diese Displays sind preiswert, verbrauchen nur wenig Energie und sind einfach zu programmieren.

Für die einfache Ansteuerung sorgt ein SSD1306-Controller (Links zu Datenblättern, Bibliotheken und Bezugsquellen siehe Ende dieses Artikels). Eine seiner nützlichsten Eigenschaften ist der interne Grafikspeicher von 128 × 64 Bits (1 KByte), der exakt den Inhalt einer Bildschirmseite aufnehmen kann. Solange der Controller keine neuen Kommandos empfängt und der Display-Speicher nicht verändert wird, zeigt er den aktuellen Inhalt des internen Speichers an. Arduino-Anwendungen müssen sonst in der Regel den Inhalt des internen Display-Speichers auch im Speicher repräsentieren. Bei diesem Display reicht es, eine lokale Kopie des Bildschirms (1 KByte) zu erzeugen und nur bei Bedarf zu übertragen, worum sich die Adafruit-Bibliothek kümmert. Es ist möglich, eigene Bibliotheken zu entwickeln, die ganz ohne SRAM auskommen.

Der SSD1306 unterstützt unter anderem die seriellen Protokolle I^2C und SPI und belegt daher nur zwei beziehungsweise vier Pins auf dem Arduino. Allerdings reichen nicht alle Displays die Pins für alle Protokolle nach außen. Das vorliegende Projekt setzt auf I^2C, und das hier verwendete Display hat auch tatsächlich nur vier Pins: Stromversorgung (VCC), Masse (GND), SCL (Clock Line) und SDA (Data Line).

Getreu dem Protokoll

Das Protokoll zwischen Arduino und dem Display ist nicht allzu kompliziert. Dennoch

Kurzinfo

Zeitaufwand:
eine Stunde

Kosten:
rund 40 Euro

Programmieren:
gute Grundkenntnisse mit Arduino

Löten:
höchstens ein paar Lötpunkte

Schwierigkeitsgrad

leicht — schwer

Material

- Arduino Uno
- 0,96-Zoll-OLED-Display
- Analog-Joystick
- Breakout-Board für Analog-Joystick
- zwei Drucktaster
- Breadboard (Steckplatine)
- Kabel

ÄHNLICHE PROJEKTE

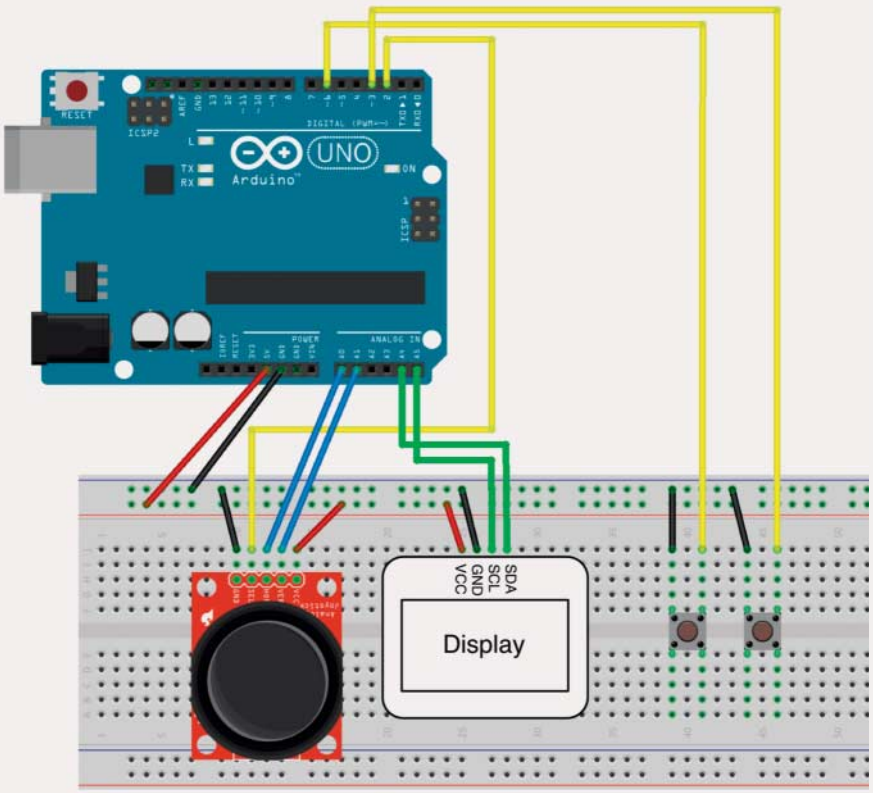
Die Idee, den Arduino mit hochwertigen Displays zu verbinden, ist naheliegend und es überrascht nicht, dass es im Internet einige ähnliche Projekte gibt.

Arduboy wurde erst kürzlich erfolgreich über eine Kickstarter-Kampagne finanziert und die Hardware ähnelt der im Artikel vorgestellten sehr. Allerdings steckt der Arduboy in einem schicken Gehäuse und hat einen Akku, der sich per USB aufladen lässt. Zur Steuerung dienen statt eines Joysticks aber vier Drucktaster; das Spielgefühl unterscheidet sich von unserer Konsole.

Der **TinyDuino** verfügt gleich über zwei Analog-Sticks und ein Farb-Display. Er hat ebenfalls einen Akku und ist extrem klein.

Schließlich gibt es noch den **Arduino Color**, der so etwas wie ein Hybrid aus Arduboy und TinyDuino ist. Er hat einen Analog-Joystick und ein Farb-Display. Portabel ist er auch, aber statt durch einen Akku wird er durch vier Batterien angetrieben.

Ein paar Spiele gibt es bereits für all diese Plattformen. Die Techniken, die in diesem Artikel vermittelt werden, lassen sich auch auf Arduboy & Co. übertragen.



Der Aufbau des Projekts ist einfach, denn so gut wie alle Teile lassen sich über gesteckte Kabel verbinden.

ANDERES DISPLAY?

Falls Sie ein anderes Display verwenden, etwa das Monochrome 1,3" 128 × 64 von Adafruit, finden Sie über den Link am Ende des Artikels Informationen, wie Sie die Schaltung der Spielkonsole verändern müssen.

FirstSteps

```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

const uint8_t I2C_ADDRESS_DISPLAY = 0x3C;
const uint8_t OLED_RESET = 4;

Adafruit_SSD1306 display(OLED_RESET);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, I2C_ADDRESS_DISPLAY);
  display.clearDisplay();
  display.setTextColor(WHITE);
  display.setTextSize(4);
  display.setCursor(16, 16);
  display.print("Make");
  display.drawRect(0, 0, 127, 63, WHITE);
  display.display();
}

void loop() { }
```

wäre es lästig, zeitaufwendig und fehlerträchtig, das ganze Byte-Gefriemel selbst zu implementieren.

Dankenswerterweise haben andere diese Arbeit bereits erledigt und auf Github die Bibliothek Adafruit-SSD1306 veröffentlicht. Diese Bibliothek kapselt alle Interna der SPI- und I²C-Kommunikation und stellt die nativen Funktionen des Displays über eine schlanke C++-Klasse zur Verfügung.

Neben der Darstellung des Display-Speicherinhalts erledigt der SSD1306 auf Hardware-Ebene nur wenige Aufgaben. Dazu gehören die Invertierung der Ausgabe, das Dimmen des Displays oder das Scrollen von Inhalten. Alles andere, wie das Zeichnen von Punkten, Linien, Kreisen, Texten oder Bitmap-Grafiken, muss der Arduino übernehmen.

Auch hier gibt es Erleichterung in Form der Bibliothek Adafruit-GFX. Diese unterstützt neben dem SSD1306 noch weitere Controller und bietet eine Vielzahl an Funktionen zum Zeichnen von Grafiken.

Die Installation von Bibliotheken erfolgt in der aktuellen Arduino-IDE am besten über den Library-Manager. Der verbirgt sich hinter dem Menüpunkt „Sketch/Include Library“ und bietet eine komfortable Suchfunktion. Neue Bibliotheken kann man hier per Mausklick installieren, sofern sie auf der Liste der offiziellen Arduino-Bibliotheken stehen. Sowohl für Adafruit-SSD1306 als auch für Adafruit-GFX ist das der Fall.

Hallo Display!

Um die Funktionstüchtigkeit des Displays zu testen, bietet es sich an, das Beispiel ssd1306_128x64_i2c in der Arduino-IDE über das Menü „File/Examples/Adafruit SSD1306“ zu laden. Dieser Sketch enthält eine umfangreiche Demo, die alle Möglichkeiten der Adafruit-GFX-Bibliothek zeigt.

Bleibt das Display allerdings schwarz, nachdem der Sketch übersetzt und auf den Arduino übertragen wurde, liegt der Grund in folgender Anweisung in der setup()-Funktion des Demo-Sketches:

```
display.begin(SSD1306_SWITCHCAPVCC, 0x3D);
```

Hier wird das Display initialisiert und die hexadezimale Zahl 0x3D als ID des anzusprechenden I²C-Geräts verwendet. Die überwiegende Mehrheit der getesteten Displays hatte aber die ID 0x3C. Oft ist die ID auch auf der Platine des Displays vermerkt. Aber Achtung: Dort stehen in der Regel 8-Bit-Adressen, während der Ardu-

```
FirstStepsFlash

#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "textutils.h"

const uint8_t I2C_ADDRESS_DISPLAY = 0x3C;
const uint8_t OLED_RESET = 4;

Adafruit_SSD1306 display(OLED_RESET);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, I2C_ADDRESS_DISPLAY);
  display.clearDisplay();
  pmem_print_center(16, 4, PSTR("Make"));
  display.drawRect(0, 0, 127, 63, WHITE);
  display.display();
}

void loop() { }
```

ino die 7-Bit-Adresse verwendet. Auf dem eingesetzten Display liest man beispielsweise die Adresse 0x78. Schiebt man die um ein Bit nach rechts, ergibt das 0x3C.

Sobald die richtige ID eingetragen und der Sketch auf den Arduino übertragen wurde, sollte es auf dem Display jede Menge zu sehen geben. Das Studium des Demo-Sketches ist in jedem Fall sinnvoll, aber aufwendig. Einfacher ist es, mit einem kleinen Beispiel anzufangen und dieses sukzessive zu erweitern.

Erste Schritte

Unser FirstSteps-Sketch (siehe Listing auf Seite 128) demonstriert die Initialisierung des Displays und gibt sowohl einen Text („Make“) als auch ein Rechteck aus. Die gesamte Action findet in der setup()-Funktion statt, in der zuerst das globale Objekt display mit der begin()-Funktion initialisiert wird. Der anschließende Aufruf von clearDisplay() ist notwendig, weil der Display-Speicher sonst das Logo der Firma Adafruit enthält.

Die folgenden beiden Anweisungen setzen die Textfarbe auf weiß und die Textgröße auf 4. Der Basis-Zeichensatz hat eine Breite von 6 Pixeln und die Textgröße 1. Die weiteren Textgrößen sind jeweils Vielfache von 6, also haben Zeichen mit der Textgröße 4 eine Breite von 24 Pixeln.

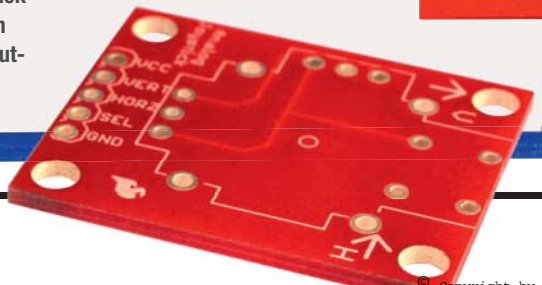
Der Aufruf von setCursor() positioniert den Cursor pixelgenau auf eine x-y-Koordinate und print() gibt schließlich den Text aus. Die Funktion drawRect() zeichnet ein Rechteck und erwartet die Koordinaten der linken oberen Ecke und die Breite und Höhe des zu zeichnenden Rechtecks. Mit der display()-Funktion wird der Inhalt des internen Grafik-Puffers an das Display übertragen und ausgegeben. In der loop()-Funktion ist anschließend nichts zu tun, weil das Display den einmal übertragenen Inhalt so lange anzeigt, bis er verändert wird.

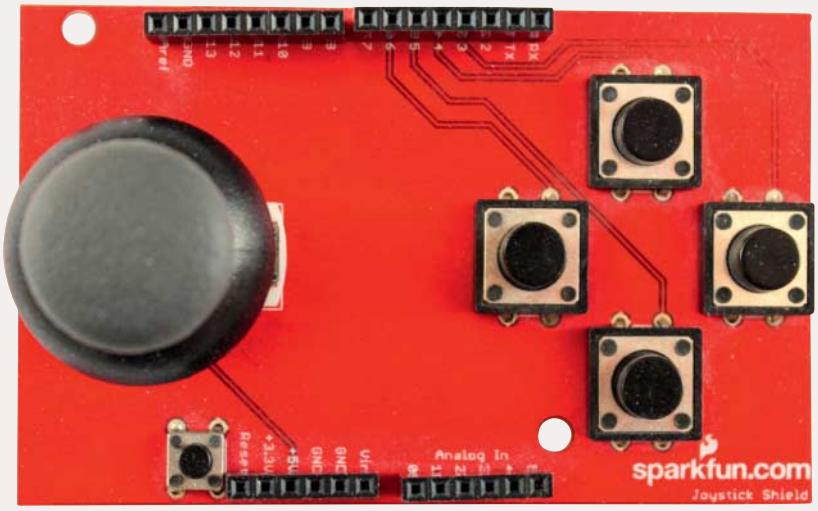
Texte ausgeben

Die Adafruit-Bibliotheken lassen in Bezug auf die Ausgabe von Texten kaum Wünsche offen. Texte können pixelgenau positioniert werden und es stehen verschiedene Schriftgrößen zur Verfügung. Zu lange Texte werden automatisch abgeschnitten oder auf Wunsch umgebrochen.

Ein kleines Problem gibt es aber doch, denn die Ausgabe von Texten, die sich im Flash-RAM befinden, ist nicht ohne Weiteres möglich. Das ist für Anwendungen wie

Den Mini-Joystick muss man noch auf das Breakout-Board löten.





Alternativ zum Joystick und den Drucktastern kann auch ein fertiges Joystick-Shield eingesetzt werden.

Starfield

```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

const uint8_t I2C_ADDRESS_DISPLAY = 0x3C;
const uint8_t OLED_RESET = 4;
const uint8_t MAX_STARS = 10;

struct Star { int8_t x, y, vx; };

Star starfield[MAX_STARS];
Adafruit_SSD1306 display(OLED_RESET);

void init_starfield() {
  for (uint8_t i = 0; i < MAX_STARS; i++) {
    starfield[i].x = random(display.width());
    starfield[i].y = random(display.height());
    starfield[i].vx = random(3) + 1;
  }
}

void move_stars() {
  for (uint8_t i = 0; i < MAX_STARS; i++) {
    starfield[i].x -= starfield[i].vx;
    if (starfield[i].x < 0) {
      starfield[i].x = display.width();
      starfield[i].y = random(display.height());
      starfield[i].vx = random(3) + 1;
    }
  }
}

void draw_stars() {
  for (uint8_t i = 0; i < MAX_STARS; i++) {
    display.drawPixel(starfield[i].x, starfield[i].y, WHITE);
  }
}

void setup() {
  randomSeed(analogRead(A0));
  init_starfield();
  display.begin(SSD1306_SWITCHCAPVCC, I2C_ADDRESS_DISPLAY);
  display.display();
  delay(1000);
}

void loop() {
  display.clearDisplay();
  move_stars();
  draw_stars();
  display.display();
}
```

Spiele aber zwingend notwendig, weil auch viele kurze Texte recht schnell viel Platz im kostbaren SRAM belegen. Daher sollten Texte tunlichst im Flash-RAM abgelegt werden.

Das PSTR-Makro verlagert einen Text ins Flash-RAM. Leider funktioniert die folgende Anweisung nicht wie gewünscht:

```
display.print(PSTR("Make"));
```

Der Mangel ist aber schnell behoben und die zum Download bereitgestellten Dateien textutils.h und textutils.cpp enthalten Funktionen, mit denen sich Texte aus dem Flash-RAM ausgeben lassen. pmem_print_center() gibt Texte sogar gleich horizontal zentriert aus.

Die eigentliche Arbeit erfolgt in pmem_print(), denn hier wird der Text mit der Funktion pgm_read_byte() Byte für Byte aus dem Flash-RAM gelesen und ausgegeben.

Das Beispiel FirstStepFlash zeigt, wie die Funktionen eingesetzt werden.

Effekthascherei

Die bisherigen Beispiele waren allesamt statisch, das heißt, auf dem Bildschirm bewegte sich nichts. Das lässt sich schnell ändern und das animierte Sternenfeld im Hintergrund von Shootduino ist ein dankbares Demonstrationsobjekt. Es verleiht dem Spiel ein gewisses Weltraumflair und lässt sich mit nur wenigen Zeilen Code implementieren. Als in sich abgeschlossenes Demoprogramm gibt es das Sternenfeld im Ordner namens Starfield in unseren Downloads, im fertigen Shootduino-Spiel ist der Code in starfield.h und starfield.cpp zu finden.

Jeder Stern wird durch die Struktur Star repräsentiert. Die Struktur enthält die aktuellen Bildschirmkoordinaten des Sterns und seine aktuelle Geschwindigkeit vx in x-Richtung. Das Sternenfeld wird durch ein Array solcher Strukturen dargestellt und durch drei Funktionen manipuliert.

Die Funktion init_starfield() belegt die Attribute aller Sterne mit zufälligen Werten. move_stars subtrahiert die aktuelle Geschwindigkeit von der x-Koordinate eines jeden Sterns. Dadurch hat es den Anschein, der Stern bewege sich von rechts nach links. Verlässt der Stern den Bildschirm auf der linken Seite, wird er durch einen neuen Stern ersetzt. Schließlich gibt draw_stars() alle Sterne als einzelne Pixel auf dem Bildschirm aus.

Auf mehrfarbigen Displays simuliert ein einfacher Trick übrigens deutlich mehr räumliche Tiefe. Dazu müssen die Sterne lediglich in verschiedenen zufälligen Grautönen dargestellt werden. Dunklere Sterne erscheinen dann automatisch weiter weg.

Grafiken erzeugen ...

1024 monochrome Pixel können schwerlich als Grundlage für ein opulentes Grafikfeuerwerk dienen. Trotzdem sind sie ausreichend für unterhaltsame Spiele, wobei knackige Bitmaps ordentlich zum Spielspaß beitragen.

Prinzipiell eignet sich jedes Grafikprogramm der Welt zur Erstellung der Mini-Bitmaps, die das Display darstellen kann. Allerdings haben moderne Anwendungen wie GIMP oder Photoshop einen völlig anderen Fokus und ihre Werkzeuge sind nicht in erster Linie für die Bearbeitung kleinster Grafiken konzipiert. Das gilt im

Besonderen für die Erzeugung von Animationen.

Hier sind spezielle Grafikprogramme deutlich im Vorteil und für Shootduino fiel die Wahl auf Piskel (siehe Link am Ende des Artikels). Piskel läuft in jedem modernen Web-Browser und ist frei verfügbar. Es bietet die Standardwerkzeuge zur Erstellung und Bearbeitung von Pixel-Bitmaps und -Animationen. Darüber hinaus erlaubt Piskel den Export von Grafiken in die Formate GIF und PNG.

Mit den begrenzten Ressourcen, die auf dem Arduino zur Verfügung stehen, ist es aber nicht sinnvoll beziehungsweise unmöglich, PNG- oder GIF-Dateien direkt zu verarbeiten. Die exportierten Dateien müssen daher noch in C/C++-Code umgewandelt werden, der mit dem Rest des Spiels übersetzt und automatisch im Flash-RAM gespeichert wird.

… und ausgeben

Für diesen Umwandlungsprozess gibt es eine Reihe von Optionen. Ganz Hartgesottene können zum Beispiel die Pixel per Hand in Binärzahlen umwandeln und dabei ein wenig in Erinnerungen an die achtziger Jahre schwelgen. Fortschrittlichere Geister setzen eher auf eine automatische Lösung und schreiben ein kleines Konvertierungsprogramm oder sie setzen auf bereits bestehende Lösungen. Davon stehen sowohl online als auch offline eine ganze Menge zur Wahl. Unter anderem gibt es Image2Code, das Bestandteil der Adafruit GFX-Bibliothek ist. Allerdings kommt es nicht mit allen Grafikdateien zurecht.

Für Shootduino kam daher das eigens in der Programmiersprache Ruby entwickelte gif2cpp zum Einsatz, dass ebenfalls in unserem Download-Paket enthalten ist. Es wandelt GIF-Dateien automatisch in C++-Code für den Arduino um. GIF ist in diesem Fall die sinnvollste Wahl, weil Piskel es unterstützt und weil GIF im Gegensatz zu PNG auch Animationen repräsentieren kann.

gif2cpp setzt eine Installation von Ruby und von ImageMagick voraus. Ferner wird die Ruby-Bibliothek rmagick benötigt, die wie folgt installiert wird:

```
$ gem install rmagick
```

Gegebenenfalls ist dem Kommando noch ein `sudo` voranzustellen.

Weil gif2cpp eine Kommandozeilen-Anwendung ist, eignet es sich auch gut zur Automatisierung.

Die Funktionsweise von gif2cpp ist einfach. Die Funktion `Image::read()` liest alle Bilder, die in einer Datei gespeichert sind, und liefert sie als Liste zurück. Bei einer GIF-Datei ohne Animationen enthält die Liste nur ein Element.

Die anschließende Schleife iteriert über alle Zeilen und Spalten eines jeden Bildes und gibt die Pixel als Binärzahlen aus. Eine 1 steht für ein weißes Pixel und eine 0 für ein schwarzes. Bei den zu erwartenden Bildgrößen hat die binäre Ausgabe den Vorteil, dass der Bildinhalt sogar im Quelltext zu erkennen ist. Beispielsweise wird aus der Bitmap des Raumschiffes der folgende Code:

```
const unsigned char spaceship_bmp[] PROGMEM =
{
  B00110000, B00000000,
  B00111000, B11100000,
  B01111100, B10000000,
  B11111111, B10000000,
  B01111111, B11000000,
  B00111111, B11100000,
  B00111111, B11111000,
  B00111111, B11100000,
  B01111111, B11000000,
  B11111111, B10000000,
  B01111100, B10000000,
  B00111000, B11100000,
  B00110000, B00000000,};
```

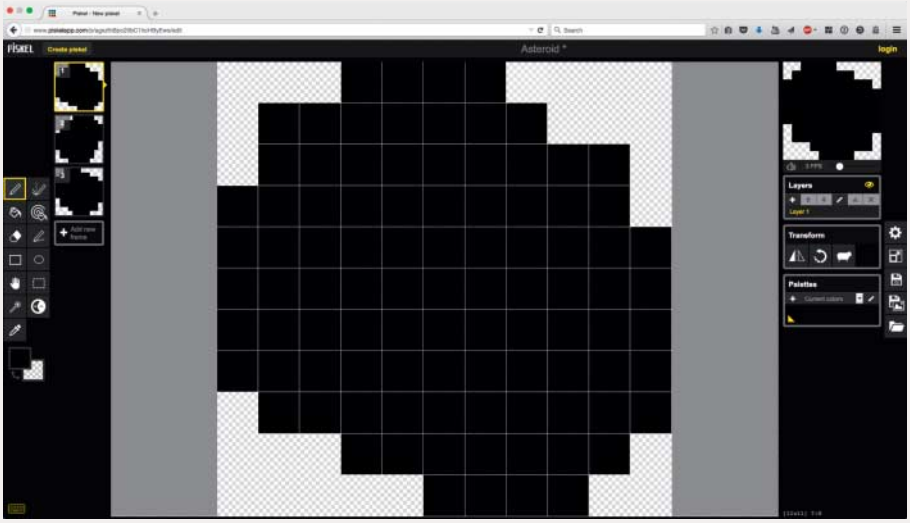
In einer weiteren Ausbaustufe könnte gif2cpp mehrere Dateien auf einmal verarbeiten und auch noch die Header-Datei generieren.

Kontrolle erlangen

Zentrales Element fast aller Videospiele ist die Steuerung – gerade bei Action-Spielen ist sie enorm wichtig. Für die Realisierung der Steuerung gibt es einige Alternativen. Beispielsweise könnte sie komplett aus Drucktastern bestehen, denn vier Drucktaster reichen für Bewegungen in alle Himmelsrichtungen aus; zwei weitere sind genug für Aktionen.

Shootduino setzt für Aktionen auf zwei Drucktaster, aber die Steuerung der Bewegung erfolgt über einen Analog-Joystick. Die sind mittlerweile günstig zu haben und einfach abzufragen. Typischerweise belegt ein Analog-Joystick zwei analoge Pins, nämlich einen für die horizontale und den anderen für die vertikale Richtung. Die meisten Modelle haben sogar noch einen eingebauten Taster, das heißt, sie lassen sich nach unten drücken.

Wer es ganz komfortabel mag, kann alternativ ein Joystick-Shield kaufen. Das Modell von Sparkfun hat beispielsweise einen klickbaren Analog-Joystick und vier Drucktaster. Mit dem Display lässt es sich allerdings nicht direkt verbinden. Dazu bedarf es wieder eines Breadboards oder einer direkten Kabelverbindung.



Mit dem kostenlosen Webdienst Piskel zeichnet man Mini-Grafiken Pixel für Pixel – auch animierte GIFs.

Mehr Flexibilität und Pioniergeist verspricht in jedem Fall der blanke Analog-Joystick. Um ihn mit einem Breadboard zu verbinden, fehlt aber noch ein Breakout-Board. Solche Boards haben die Anbieter von Analog-Joysticks häufig auch im Programm und sie kosten nicht viel. Allerdings kommen Board und Joystick in der Regel separat an und müssen noch per Lötkolben verbunden werden.

Die Software zur Abfrage des Joysticks und der Drucktaster ist nicht weiter kompliziert. Ein paar Präprozessor-Direktiven (#ifdef) machen sogar ein Modul möglich, das sowohl mit dem Joystick-Shield als auch mit dem blanken Analog-Joystick zurechtkommt. Die Schnittstelle des Moduls deklariert die Datei joystick.h. Sie definiert erst einmal Konstanten für die Pins, mit denen die Drucktaster verbunden sind. Ist das Präprozessor-Makro JOYSTICK_SHIELD gesetzt, werden zwei Konstanten mehr definiert, weil das Joystick-Shield über zwei Drucktaster mehr verfügt.

Anschließend definiert die Datei die Struktur JoystickState, die den Zustand des Joysticks und aller Taster jeweils mit einem Bit repräsentiert. Um zum Beispiel abzufragen, ob der Spieler den Joystick gerade nach links bewegt, muss geprüft werden, ob das Attribut left in der Struktur den Wert 1 hat.

Außerdem gibt es noch zwei öffentliche Funktionen: init_joystick() macht die Pins, die mit den Drucktastern verbunden sind, zu Eingabe-Pins und aktiviert die internen Pull-up-Widerstände. Auf diese Weise „flackern" die Taster nicht, sondern liefern immer ein klares Signal. Ferner bestimmt die Funktion die Werte für die Ruheposition des Joysticks in beide Richtungen. init_joystick() wird einmalig in der setup-Funktion() aufgerufen.

Die Funktion update_joystick() liest den aktuellen Stand des Analog-Sticks und der Drucktaster. Sie muss regelmäßig von der loop()-Funktion aufgerufen werden. In der Funktion gibt es eine Besonderheit bei der Ermittlung der up- und down-Werte, denn die Wertebereiche des Joystick-Shields und des blanken Analog-Sticks verlaufen entgegengesetzt.

Die Game-Schleife

Sobald die Hardware gebändigt ist, ist die eigentliche Spielprogrammierung halb so wild. Im Wesentlichen geht es um die Verwaltung diverser Zustände. Der Code ist zu umfangreich, um ihn hier im Heft abzudrucken, deshalb werden im Folgenden nur ein paar interessante Details erläutert.

Eine zentrale Rolle spielt die Struktur Game, die in der Datei game.h definiert wird. Sie enthält unter anderem die aktuelle Anzahl an Leben (lives) und die Anzahl der Asteroiden, die bisher verfehlt wurden (asteroids_missed).

Darüber hinaus verwaltet sie wichtige Informationen fürs Timing. Das Attribut ticks enthält die Anzahl der Millisekunden, die seit dem Start des Arduino verstrichen sind. bullet_fired enthält den Zeitpunkt (in Millisekunden), an dem der letzte Laser abgefeuert wurde. Mit diesen Informationen ist es möglich, die Schussfrequenz der Laserkanone einzustellen. Analoges gilt für asteroid_started. Hier wird vermerkt, wann zuletzt ein Asteroid auf die Reise geschickt wurde.

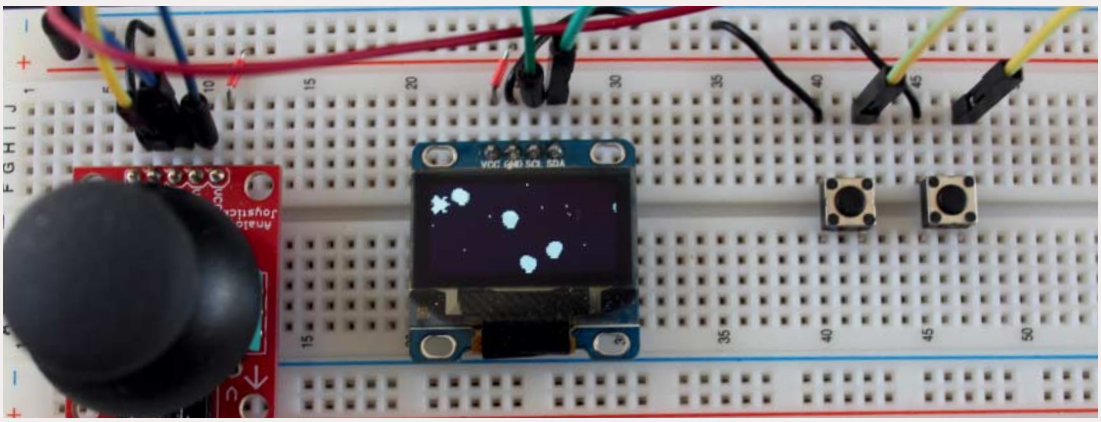
Für die Kontrolle der Spiellogik ist state verantwortlich, denn hier wird der aktuelle Zustand des Spiels gespeichert:

```
enum GameState : uint8_t {
  INTRO, RUNNING, PAUSED, LOST_LIVE,
  DONE, ENTER_HS, SHOW_HS };
```

Hat der Zustand den Wert INTRO, wird der Titelbildschirm des Spiels angezeigt. Dieser Zustand kann sich auf verschiedene Arten ändern. Beispielsweise wird er in den Zustand RUNNING überführt, wenn der Spieler den rechten Drucktaster betätigt. Der Zustand ändert sich aber auch nach sieben Sekunden Inaktivität und wird dann zu SHOW_HS. In diesem Zustand zeigt Shootduino die aktuelle High-Score-Liste an. Die Details zu den Zustandsübergängen finden sich in den Funktionen, die für die jeweiligen Zustände verantwortlich sind und die wiederum in der loop()-Funktion aufgerufen werden:

```
void loop() {
  shootduino.ticks = millis();
  update_joystick();
  display.clearDisplay();
  switch (shootduino.state) {
    case INTRO:      intro();           break;
    case PAUSED:     pause_game();      break;
    case RUNNING:    update_game();     break;
    case LOST_LIVE:  lost_live();       break;
    case SHOW_HS:    show_highscores(); break;
    case ENTER_HS:   enter_highscore(); break;
    case DONE:       game_over();       break;
  }
  display.display();
}
```

Das ist eine klassische Game-Loop, wie sie übersichtlicher nicht sein könnte. Jedem möglichen Zustand ist exakt eine Funktion zugeordnet und ansonsten werden nur das Timing, die Steuerung und das Display aktualisiert.



Im nächsten Sekundenbruchteil wird der von rechts heranrasende Asteroid das Raumschiff zerschmettern – hätte der Spieler doch seine Hände nicht vom Joystick (links) und vom Feuerknopf (ganz rechts) genommen …

Der Titel-Screen des Beispielspiels namens Shootduino

Drückt man den linken Taster, schaltet man das Spiel in den Pausenmodus.

Auch die Implementierung der einzelnen Zustandsfunktionen ist übersichtlich. Beispielsweise zeigt `pause_game()` neben der Nachricht „Pause" die aktuelle Punktzahl und die noch verbleibenden Leben an. Wenn der Spieler den linken Drucktaster drückt, wechselt das Spiel in den Zustand `RUNNING` und die Weltraumschlacht geht weiter.

Aufwendiger ist nur noch die Verwaltung der beweglichen Objekte auf dem Bildschirm, also des Raumschiffs, der Asteroiden und der Laserstrahlen. Raketenwissenschaft ist das aber auch nicht und wird in den Dateien game_objects.h und game_objects.cpp realisiert.

Jedes bewegliche Objekt (mit Ausnahme der Sterne) wird mit der Struktur `GameObject` repräsentiert. Neben den Koordinaten und der aktuellen Geschwindigkeit in x- und y-Richtung enthält die Struktur auch noch Informationen über den Objekttypen und die aktuelle Animationsphase. Damit ist es einfach möglich, den Typen und die Darstellung eines Objekts zu ändern. Zum Beispiel kann aus einem Asteroiden eine Explosion werden, wenn er von einem Laserstrahl getroffen wurde.

Die gesamte Verwaltung aller Spielobjekte wird von den drei Funktionen `init_objects()`, `draw_objects()` und `move_objects()` übernommen. Sie arbeiten alle nach demselben Prinzip: Sie iterieren über eine Liste von Objekten und behandeln dann alle Objekte, die aktiv sind, also deren `is_active`-Flag den Wert `true` hat. Diese Vorgehensweise vermeidet die dynamische Verwaltung von Speicher, denn wenn ein neues Objekt benötigt wird, ersetzt es einfach ein inaktives.

Nichts ist für die Ewigkeit?

Die bisher beschriebenen Funktionen sind absolut notwendig, aber kein Arcade-Shooter wäre komplett ohne eine Liste der High-Scores. Stilecht bietet auch Shootduino den besten Weltraumpiloten die Möglichkeit, sich mit dreistelligen Initialen und ihrer Punktzahl zu verewigen. Dabei ist „verewigen" durchaus ernst gemeint, denn die High-Scores werden im EEPROM gespeichert und überdauern ein Abschalten des Arduino.

Der Arduino Uno bietet 2 KByte EEPROM und die Arduino-Umgebung enthält eine Standardbibliothek, um Daten im EEPROM zu manipulieren. Der EEPROM hat keinerlei vorgegebene Struktur und ist für ein Arduino-Programm nichts weiter als eine Liste von 2048 Bytes. Diese Bytes können über einen Index von 0 bis 2047 adressiert werden.

Zur Speicherung von High-Scores sind das ausgezeichnete Voraussetzungen, denn pro Eintrag sind drei Buchstaben (drei Bytes) und eine Punktzahl in Form eines Integer-Werts ohne Vorzeichen (zwei Bytes) zu speichern. Shootduinos High-Score-Liste enthält drei Einträge und belegt somit magere $3 \times (3 + 2) = 15$ Bytes im EEPROM.

Eine Sache ist allerdings noch zu beachten: Wenn Shootduino zum ersten Mal auf einem Arduino installiert wird, muss die Liste der High-Scores initialisiert werden. Bei dieser Initialisierung werden alle Initialen auf AAA und alle Punktzahlen auf 0 gesetzt. Diese Initialisierung darf aber nicht bei jedem Neustart des Arduino erfolgen, denn dann würde die Liste der High-Scores jedes Mal überschrieben. Shootduino speichert daher im ersten Byte des EEPROM, also an der Adresse 0, den Wert 42, nachdem die Liste initialisiert wurde. Beim Start prüft das Programm den Wert an Adresse 0 und unterlässt die Initialisierung, wenn es dort eine 42 findet.

Die Dateien highscores.h und highscores.cpp enthalten alle Strukturen, Variablen und Funktionen, die Shootduino zur Verwaltung der High-Scores benötigt. Die Struktur `HighScoreEntry` repräsentiert einen einzelnen Eintrag in der High-Score-Tabelle.

EEPROM im Zugriff

Die EEPROM-Bibliothek des Arduino bietet zwei unterschiedliche Zugriffsmöglichkeiten. Die Funktionen `EEPROM.read()` und `EEPROM.write()` operieren auf Byte-Ebene. Mit ihnen können Bytes an beliebigen Adressen gelesen und geschrieben werden. Komfortabler sind die Funktionen `EEPROM.get()` und `EEPROM.put()`, denn mit ihnen lassen sich beliebige Datentypen- und Strukturen lesen und schreiben. Beispielsweise kommen sie in den Funktionen `get_entry()` und `set_entry()` zum Einsatz, um `HighScoreEntry`-Objekte zu lesen und zu schreiben.

Das reduziert den Aufwand zur Verwaltung des EEPROM erheblich und so dienen nur die ersten knapp 100 Zeilen in highscores.cpp der Manipulation des EEPROM. Den Löwenanteil machen dabei die Initialisierung `init_highscores()` und die Funktion `insert_entry()` zum Einfügen eines neuen High-Scores aus.

Alle weiteren Funktionen kümmern sich um die Ein- und Ausgabe der High-Scores während des Spiels. Bei der Eingabe der Initialen kann der Spieler den Joystick nach oben beziehungsweise unten bewegen, um zum nächsten beziehungsweise zum vorhergehenden Zeichen zu wechseln. Der Wechsel zur nächsten Stelle in den Initialen erfolgt über den linken Drucktaster. Abgeschlossen wird die Eingabe mit dem rechten Drucktaster.

Die Funktionen zur Eingabe der Initialen enthalten keine großen Besonderheiten. Lediglich die Konstante `INITIALS_LETTERS` ist erwähnenswert. Sie liegt im Flash-RAM und enthält alle Buchstaben, die in den Initialen erlaubt sind. Statt die aktuellen Buchstaben zu speichern, speichern die Eingabe-Funktionen jeweils den Index, den der Buchstabe in `INITIALS_LETTERS` hat.

EIGENE SPIELE

Wenn Sie selbst eine Spielidee umgesetzt oder einen Klassiker auf unsere Arduino-Konsole portiert haben, freuen wir uns über eine Mail mit dem Code an pek@make-magazin.de!

Auslastung

Der Quelltext von Shootduino besteht aus sechzehn Dateien mit insgesamt weniger als 1000 Zeilen Code (darin sind die generierten Zeilen für die Bitmap-Grafiken enthalten). Das ist für eine Arduino-Anwendung nicht gerade wenig, aber immer noch sehr überschaubar. Der tatsächliche Speicherbedarf sieht wie folgt aus:

– 17 888 Bytes Flash-RAM (55 %)
– 1541 Bytes SRAM (75 %)
– 16 Bytes EEPROM (2 %)

Die Bilanz kann sich durchaus sehen lassen und der verbleibende Speicher lässt noch jede Menge Spielraum für Erweiterungen. Allerdings gibt es auch keinen Anlass für Übermut, denn die verbleibenden 507 Bytes im SRAM können auch schnell aufgezehrt werden. Das muss nicht einmal das Produkt als solches beeinflussen, sondern kann schon während der Entwicklung ein Problem werden.

Beispielsweise sind Ausgaben auf der seriellen Schnittstelle beim Aufspüren von Programmfehlern sehr hilfreich. Die Serial-Bibliothek benötigt aber 173 Bytes SRAM und kann ab einem bestimmten Punkt nicht mehr verwendet werden.

Aus diesem Grund ist es wichtig, stets ein Auge auf den Speicherbedarf neuer Objekte und Bibliotheken zu haben. Insbesondere sollten Datentypen nie unnötig groß gewählt werden. Auch kann übermäßige Generalisierung den Speicher schnell schrumpfen lassen. Aus diesem Grund gibt es zum Beispiel die globale Variable player_hit und kein Attribut was_hit in der Struktur GameObject.

Mit einfachen Tricks lässt sich ein wenig Platz sparen, ohne dass der Quelltext dadurch unübersichtlicher würde. Häufig lohnt sich der Einsatz von Techniken, die zwar nicht unbedingt jedem geläufig sind, aber zum offiziellen Sprachumfang von C/C++ gehören.

Zum Beispiel macht die Struktur JoystickState von Bit-Feldern Gebrauch:

```
struct JoystickState {
  bool left : 1;
  bool right : 1;
  bool up : 1;
  bool down : 1;
  bool joy_button : 1;
  bool left_button : 1;
  bool right_button : 1;
};
```

Dadurch belegt jedes Attribut nur ein Bit im Speicher, sodass die ganze Struktur mit einem Byte auskommt. Leider klappt dieser Trick nicht immer. In der Struktur Star, die zur Implementierung des Sternen-Felds dient, käme das Attribut vx mit nur zwei Bits aus. Weil der Compiler aus Gründen der Effizienz aber automatisch Füll-Bytes (Padding) in die Struktur integriert, bringt es nichts, die Struktur wie folgt zu definieren:

```
struct Star { int8_t x, y, vx : 2; };
```

Im Gegenteil: Der Bedarf an SRAM bleibt in diesem Fall gleich und das Programm verbraucht sogar etwas mehr an Flash-RAM, weil der Compiler zum Zugriff auf das Attribut vx weniger effizienten Code generieren muss.

Ein weiterer Trick betrifft Enumerationen. Moderne C++-Compiler erlauben nämlich enum-Deklarationen mit Typ-Deklarationen:

```
enum GameObjectType : uint8_t
```

Diese Deklaration legt fest, dass eine Ausprägung der Enumeration GameObjectType vom Typ uint8_t sein soll, also nur ein Byte benötigt. Per Voreinstellung wären es sonst zwei.

Wie geht es weiter?

Shootduino ist nicht nur spielbar, sondern macht durchaus Spaß. Trotzdem kann es noch ein paar Erweiterungen und Verbesserungen vertragen. Auf der Software-Seite ließe sich beispielsweise ein Power-Up-System einfügen und der Schwierigkeitsgrad variieren. Auch gäbe ein Demo-Modus dem Spiel einen professionelleren Touch.

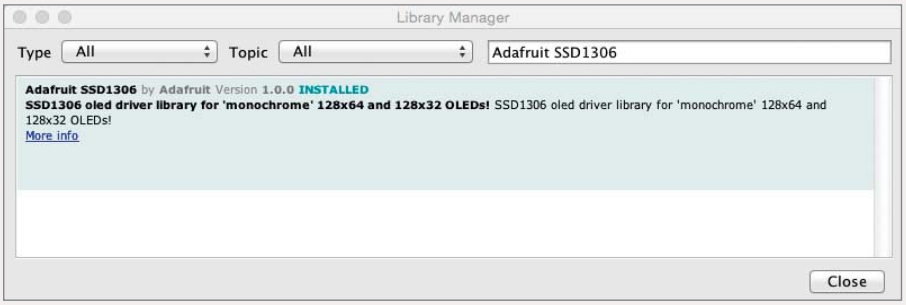
Viele Teile des Shootduino-Codes eignen sich auch für andere Spiele. Der einzige Aspekt, der bisher nicht berührt wird, ist Scrolling, aber das wird vom Display sogar per Hardware unterstützt und sollte daher keine Probleme bereiten.

Auf der Hardware-Seite sind der Fantasie kaum Grenzen gesetzt. Ganz sicher würde eine Audio-Ausgabe die kleine Konsole enorm aufwerten. In Form eines eigenen Shields ließe sich das Ganze sogar in eine portable Konsole mit SD-Kartenslot, Akku und Gehäuse verwandeln. Selbstverständlich könnte das Display auch durch ein Farb-Display ersetzt werden.

Der Arduino ist kein Kraftpaket, aber für ein paar flotte Spielchen hat er genug MIPS unter der Haube. Besonders attraktiv wird er als Spieleplattform durch die günstigen und guten Displays, die mittlerweile überall zu haben sind und sich dank guter Bibliotheken kinderleicht einbinden lassen.

Die Arduino-Umgebung selbst ist vielleicht nicht die komfortabelste der Welt, aber sie hat sich über die Jahre gemausert und wurde insbesondere in den letzten Monaten ständig verbessert. Gerade die Unterstützung moderner C/C++-Features vereinfacht die Programmierung enorm.

Bei aller Euphorie übers Zocken sollte aber nicht vergessen werden, dass sich die Displays auch für seriöse Zwecke eignen und auch beim Debugging eine gute Alternative zum seriellen Monitor sind. Ideen für weitere Einsatzzwecke finden Sie online – einfach den Link unten in den Browser tippen. *—pek*



Die benötigten Adafruit-Bibliotheken werden offiziell von der Arduino-Bibliothek unterstützt.

Links und Foren
make-magazin.de/x4ef