

BEN EVERARD

ARDUINO & PYTHON: BUILD ROBOTIC WEAPONRY

Amass a drone battalion armed to the teeth with foam darts – and take over the world!

WHY DO THIS?

- Control hardware with the Python programming language
- Learn robotics in a semi-practical context
- Take over the world!

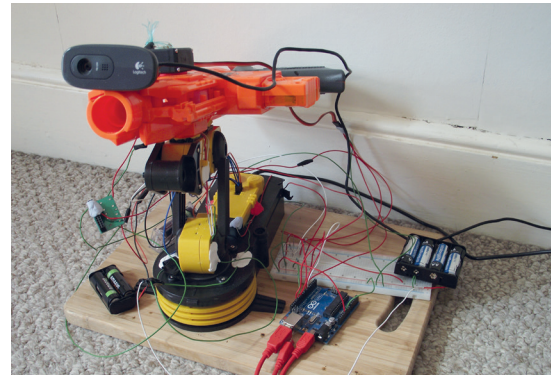
There's something incredibly geeky about Nerf guns. Perhaps it's because they give us a safe way to live out sci-fi fantasies without the risk of actually getting shot by a phaser, or perhaps it's because they're built in such a way that they're easy to take apart and hack.

We've taken one of these geek toys and fitted it out with tracking software and mounted it on a robotic arm. Now it automatically targets any humans that should stray into its range. That should send a message to anyone who tries to break into LV Towers!

The most important part of any such weaponry is the gun. We used a Nerf N-Strike Elite Stryfe Blaster, because this model has a semi-automatic firing system that makes it easier to control electronically, though there are others that could work.

The semi-automatic firing system has two parts. The automatic part consists of two spinning discs that accelerate the foam dart down the barrel. The manual part is a lever assembly that pushes the foam dart forward into these spinning discs.

With the gun picked, we just needed a way of aiming it, and that meant we had to mount it on something that the computer could move. The only real specifications for the mount were that it had two degrees of freedom (so that it could aim both horizontally and vertically), and that it could support the weight of the gun. We used a generic Robotic Arm with PC USB interface from Maplin (www.maplin.co.uk/p/robotic-arm-kit-with-usb-pc-interface-a37jn).



The weapon of mass distraction, ready to strike fear into anyone who trespasses into our geek lair.

To give our killer robot sight, we added a USB webcam. This, when coupled with the OpenCV library and a bit of Python, enables our bot to automatically target people's faces.

As well as the gun, mount and webcam, we needed a few pieces to bring it all together. These were:

- 1 x servo
- 1 x Picoborg motor controller
- 2 x 4AA battery holders
- 4 x 6.3kΩ resistors
- 4 x 200Ω resistors
- 3 x sachets Sugru
- 1 x Arduino Uno R3

1 THE BUILD

There are three basic types of Nerf gun: manual, semi-automatic and fully-automatic. Our semi-automatic gun needed an additional mechanism needed to pull the firing pin forward about two inches. This pushes the dart forwards far enough for the spinning discs to pick it up and fling it forward. Normally this is done by the trigger. However, we took out almost all of the trigger assembly, including a couple of mechanical safeguards that prevented the trigger from firing when there weren't bullets in the chamber. These were all screwed in place, so could be removed easily. We left only the final lever, which was also used to hold the firing pin in place.

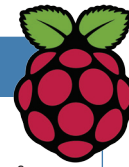
Linear actuators are electrically controlled devices for pushing things forwards. However, they're heavy and expensive, so we opted for a simpler method of

tying a string to an arm on a servo and rotating the servo 160 degrees to pull the string forward. The other end of this string is attached to the firing pin. Servos are motors that are geared and have a feedback potentiometer. This means that rather than just rotate them, you can set them to move to a defined position.

The trigger mechanism

The trigger requires a reasonably hard pull to fire and this could be more than what many small servos can provide. We used a Tower Pro MG995 servo, which is fairly powerful and good value (usually around £10). We mounted this on the outside of the Nerf gun using a blob of Sugru (though any strong glue would do), and cut a section out of the side of the gun to allow it to access the firing mechanism.

Raspberry Pi



We initially tried to write this project to run off the GPIO pins of a Raspberry Pi, but for several reasons, it just didn't work. The most demanding part of controlling this hardware is generating the pulses to communicate with the servo. Turning them on and off very quickly in software is possible, but not really viable when there's so much other stuff demanding CPU time. The solution to this is Pulse Width Modulation (PWM), a hardware feature that enables you to control rapid pulses without much CPU intervention. The Pi does support PWM, though it isn't available in the popular **RPi.GPIO** Python module, so we used the RPIO GPIO module instead (<http://pythonhosted.org/RPIO>).

However, when we tried to use a Pi to power the first soldier in our robot army, we found that it became very unstable. We strongly suspect that this is because of the high power requirements of running both the CPU-intensive OpenCV software, the GPIOs, and the PWM. Even with all the USB peripherals on a powered USB hub, we found it unreliable. It may be possible to get around this with some tweaking.

The way around this is to offload all the input and output functions to a powered expansion board. This board needs to support driving a servo and have at least five GPIO pins. In our opinion, the best expansion for this is an Arduino, and we would recommend using the same hardware setup on a Raspberry Pi or similar small computer. The only necessary change is to the scaling factors for the images in the OpenCV detection. This will make it easier for the Pi to process the data. Of course, this does mean that the image recognition would be less capable. The trade-off is between the frame rate of the video (and detection), and the accuracy of the face-tracking.

Since this method of using an Arduino doesn't use any of the GPIOs, it should be possible to run it on almost any Linux-capable board (the OpenCV Python module is quite portable), and something like the Odroid U3 might be a better (though more expensive) option than the Pi. Alternatively, the Udoo computers include an Arduino built in, so they should allow you to run the entire control from a single board.

To avoid damaging either the servo or the gun, we tied the servo to a coiled elastic band, and then tied this coiled elastic band to the trigger mechanism. This provided a small amount of stretch in the event that we should accidentally set the servo to pull beyond

the distance that the firing pin is supposed to move. Before fully assembling the hardware, we got all of the control circuits and software working, because it would be a lot harder to change things once it was all stuck together.

2 CONTROL

To control the gun and get feedback from the arm, we used an Arduino Uno Rev3. The Uno is our microcontroller of choice because of its ease of use and the number of support and code examples that exist for it online.

Arduinos have a programmable processor and loads of input and output pins (the Uno has 14 digital input/output pins and six analogue input pins). The processor is far simpler than the sort of CPU you'd find in a normal computer, so doesn't support anything like a normal operating system. Instead, you write your programs on a normal computer and upload them onto the Arduino.

The Arduino program for this project has to handle two elements: it has to listen to the sensors that we built to detect excessive movement and pass this on to the computer, and it has to take commands from the computer and control the spinning cylinders and the servo accordingly. These requirements mean that we have to shuffle information back and forth between the computer and the Arduino. The easiest way to do this is to send text over a USB serial connection, which is established automatically when you plug the Arduino into a USB port.

The letters R, G, U and D are sent by the Arduino to let the computer know that the arm has moved as far as it can. R and G are for anti-clockwise and clockwise respectively (we used red and green wires). U and D are for the up and down end stops.

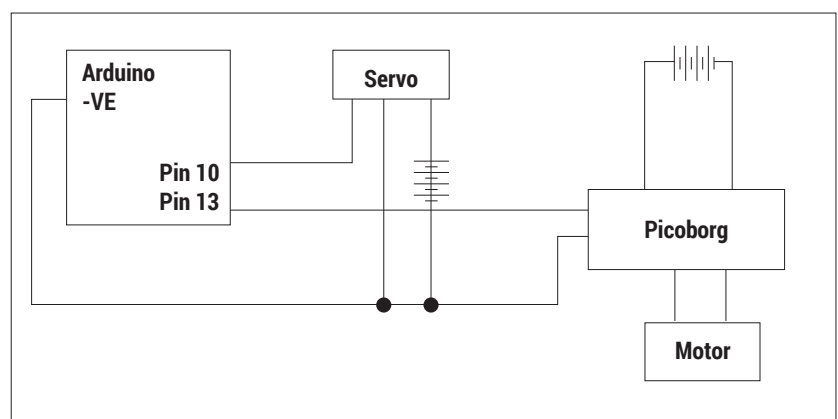
To avoid confusion, the commands from the computer to the Arduino are the numbers 1 to 4:

- 1 reset the trigger
- 2 pull the trigger
- 3 stop the spinning
- 4 start the spinning. Using this simple protocol, we could control the hardware as we needed.

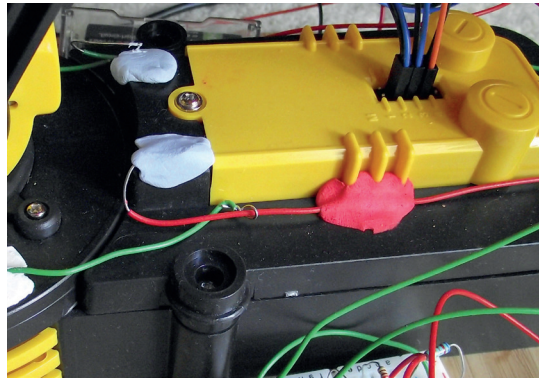
We have learned nothing from Skynet

Now let's take a look at how the hardware connects together. Almost all servos have three wires to control them: a positive, a negative and a control. The control wire can be connected directly to a pin on the Arduino. The positive and negative wires need to be connected to a 5–6V voltage source. The Arduino does have a 5V voltage pin, but it can't supply enough current to drive the servo. Instead, we used a 4 AA battery holder. In order for both the control signal and the drive current to be able to flow properly, you also need

The circuit diagram for how the motor and servo are connected to the Arduino.



The rotation end stops are mounted on opposite sides of the base and held in place with Sugru.



to connect the negative output on the battery to the Arduino ground.

Servos hark from the days before microcontrollers became common, and so their control mechanism is a little unusual. The instructions that tell the servo what position to put the arm in are sent as a series of pulses. These pulses can either be very short or quite long, and the length of the pulse denotes the position the arm should be in. Fortunately, you don't need to worry about any of this as there are libraries to control the pulse frequency and duration for just about every hardware platform that supports servos.

The firing mechanism

For the Arduino, that library is called **Servo**, and it comes as standard. You only need to create a servo object that's attached to the correct pin, and then write the value to it that corresponds to the position you want it in. A couple of examples called **knob** and **sweep** detail all the basic usage and come with the Arduino IDE.

Once the servo has pulled the dart forwards, it's picked up by spinning discs that accelerate it. These are powered by a simple DC motor that needs 5 or 6V applied across it. In order to get to the wires that power the motor, we needed to open up the gun. Inside there was a simple circuit that consisted of a trigger switch for the motors, two safety switches, and a cut-off. We removed all this, and were just left with two wires (one red and one black) heading forwards to the disk motors in the front of the gun. These are what we needed to supply power to in order to shoot.

As with the servo, the Arduino can't deliver enough power for them to run, so instead we need to use an Arduino output to switch a larger current. This is when a small current from a controller is used to turn a switch on or off, and this switch connects or disconnects a more powerful source of power (in our case four AA batteries) to the motors. We used a separate set of batteries to the ones driving the servo. In principle, you could try to set up a single power source to drive all of the motors on this project. However, we kept them all separate both to prolong the battery life and to avoid any awkward power-supply related issues.

There are a huge array of motor drivers available, and plenty of them can attach directly to the Arduino

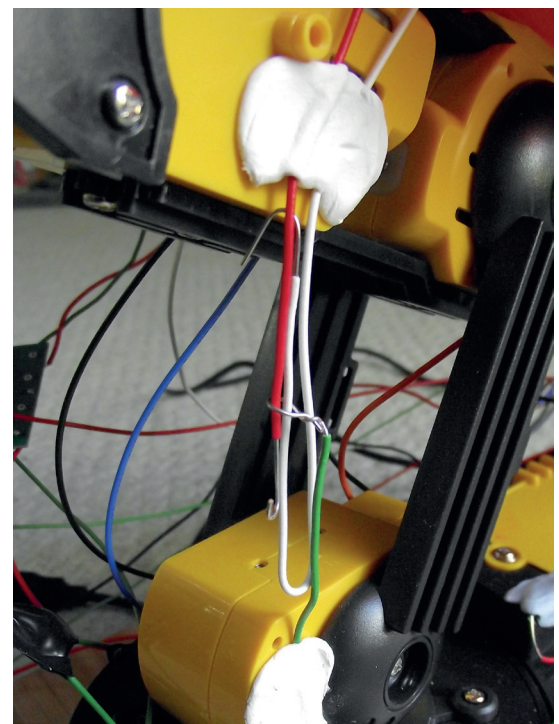
as 'shields' that slot into the headers of the board. Some motor controllers have speed controllers, or direction controllers, or the ability to electronically brake the motor – all features that are often needed, but completely unnecessary for us.

We didn't happen to have an Arduino motor controller in the LV workshop (and this definitely wasn't because someone forgot to order one). We did have a Picoborg motor driver that's designed for the Raspberry Pi. Since all this does is attach the pins from the Raspberry Pi header to Field Effect Transistors (FETs) that are used to drive the motors, we can easily use the board with our Arduino. All we need to do is use a wire to attach one of the Arduino pins to the appropriate place for the Raspberry Pi GPIO pin (in this case pin 4), and similarly connect the ground to the right pin. Once this is connected, the power supply and motor output wires need to be soldered in place, and then turning the pin on or off on the Arduino will turn the motor on and off.

Control the servo

The code to control the trigger servo and the motors is as follows (an extract from the **loop()** function):

```
if (Serial.available()) {
  data_in = Serial.parseInt();
  if (data_in == 1) {
    myservo.write(155);
  }
  if (data_in == 2) {
    myservo.write(25);
  }
  if (data_in == 3) {
    digitalWrite(spinPin, LOW);
  }
}
```



The vertical end stops are combined into a single unit.

```

if (data_in == 4) {
  digitalWrite(spinPin, HIGH);
}
}

```

This simple code enables the Python program to control the gun as it needs.

We won't go into details of how we built the arm because we simply followed the instructions that came with it. However, once it was built, we did have to make some modifications. In its raw state, it had only one-way communication with the computer. That meant that the computer could tell it to move, but it didn't feedback any information about its position. For the general operation of the gun, this wasn't a huge problem; however, it did mean that the computer had no way of knowing if the arm had reached its limit of movement in any one direction.

Protect the mechanism

We built some simple sensors out of wire with a hook bent in the end, and another wire looped around it. As the robot moves, the loop slides up and down the wire. Here the wire is insulated by plastic, so there isn't a connection, but when the loop reaches the hook, there isn't any wire, so the circuit is completed, and this signals the microcontroller.

A couple of resistors (one pull-down and one protection) are needed to make sure that the microcontroller reads the input correctly. See figure 3, below, for the circuit diagram. We built this simple circuit on a breadboard.

These readings are then sent over the serial connection with the following (from the main loop):

```

if(rcount > 0) { rcount--; }
if(gcount > 0) { gcount--; }
if(ucount > 0) { ucount--; }
if(dcount > 0) { dcount--; }

if(digitalRead(rPin) == HIGH && rcount == 0) {
  Serial.write("r\n");
  Serial.write("r\n");
}

```

Safety

We know someone will ask this, so yes, this same method would work with a BB gun, paintball gun, pistol, assault rifle or rocket launcher, but please, PLEASE, don't do it. This machine doesn't think before it pulls the trigger, it simply reacts to an image recognition that's prone to misclassification. The consequences of a poorly aimed, poorly timed foam dart are quite small. The same cannot be said of BBs and paintballs (and surely we don't need to spell out why it's a bad idea to attach a lethal weapon to a computer – just watch Terminator!).

The foam darts fired by Nerf guns are fairly safe, and should be safe for most children old enough to assemble such a project (Hasbro, the maker of Nerf guns, says they're safe for ages 8 and up). That said, it's still a good idea to wear eye protection, especially while testing.

```

rcount = 20000;
}

if(digitalRead(gPin) == HIGH && gcount == 0) {
  Serial.write("g\n");
  Serial.write("g\n");
  gcount = 20000;
}

if(digitalRead(uPin) == HIGH && ucount == 0) {
  Serial.write("u\n");
  Serial.write("u\n");
  ucount = 20000;
}

if(digitalRead(dPin) == HIGH && dcount == 0) {
  Serial.write("d\n");
  Serial.write("d\n");
  dcount = 20000;
}

```

This works in a slightly unusual way. It writes the value to the serial line twice to make sure that it is sent properly, because there isn't much error checking on a serial connection.

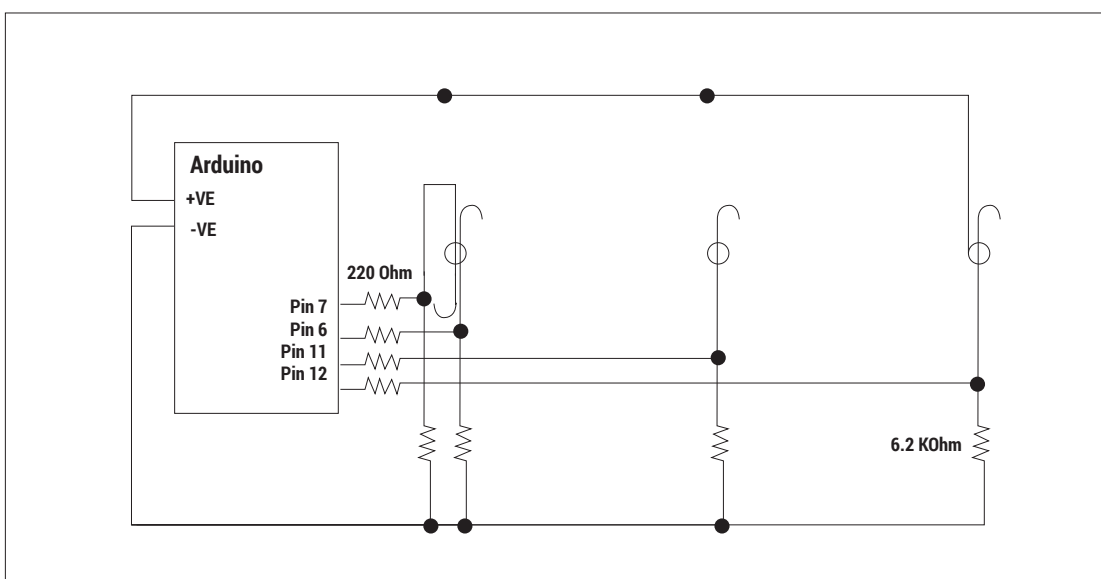
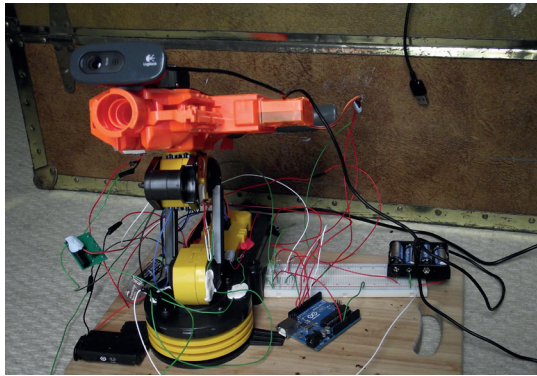


Figure 3. The two sets of resistors make sure that the pin reads correctly when the circuit is open and closed.

The fully assembled weapon primed and ready to fire. It needs long wires to allow it to move freely, but this can lead to it looking a bit like a bird's nest.



It also uses a loop counter to stop it sending the message too frequently. This loop will run much quicker than the loop in the control program that reads the data line. By using these counters, which limit the message to once every 20,000 loop cycles, we ensure that we won't clog up the main program with thousands of duplicate messages, but that we'll still keep sending it frequently enough to make sure that everything runs smoothly.

The arm is controlled via the USB port, so unlike the rest of the hardware, we can send instructions to it directly from our main Python program and not have

to use the Arduino. There isn't a Linux driver for the device we used, but the folks at www.MagPi.com have decoded the USB instructions needed to move the arm, and so we programmed it by sending the necessary commands via the PyUSB module. We'll only cover the commands we need, but for more information, see the article at www.themagpi.com/issue/issue-3/article/skutter-write-a-program-for-usb-device/.

First you need to download PyUSB, the module we'll use to send commands to the arm (<http://sourceforge.net/projects/pyusb>). Unzip this and move into the directory it created and install it with:

```
sudo python setup.py install
```

The arm can then be controlled with code such as:

```
import usb.core, usb.util, time
```

```
RoboArm = usb.core.find(idVendor=0x1267, idProduct=0x0000)
```

```
RoboArm.ctrl_transfer(0x40,6,0x100,0, [16,0,0], 1000)
```

The `ctrl_transfer()` call sends the instruction to the arm. The numbers in the square brackets specify the exact movement, as you'll see in the final code.

If you're using a different mount or arm, you may need to control it via the Arduino. This should be fairly easy to do by extending the serial commands to include extra ones to move the arm.

3 TRACKING

We've now covered everything we need to control the hardware, so we need to create the software that will actually target people who happen to pass by.

Our control software will run in a loop that goes as follows:

- 1 Check for serial communication from the Arduino
- 2 Grab a new frame and look for a face
- 3 If there's a face in the frame: make sure the disc motor is on. Otherwise, turn the motor off
- 4 Calculate how far the face is from the centre of the frame
- 5 Turn the gun towards the face!
- 6 If the face is in the centre of the frame: shoot.
- 7 Repeat

This runs over and over again until the user stops it. Here's the first part (which reads the serial connection):

```
while (ser.inWaiting() > 0):
    serdata = ser.readline()
    if serdata == "r\n":
        stop_r = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "g\n":
        stop_g = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "u\n":
        stop_u = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "d\n":
```

```
stop_d = True
```

```
RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)
```

As the Arduino sends out commands, this reads them in using the Python serial module (see the full code, which you can grab from www.linuxvoice.com/wp-content/uploads/code/lv06-gun.tar.gz for how to initialise this). The `if` statements here match exactly with ones in the Arduino code.

If any of these pieces of data are found, the software sends the arm an instruction to stop moving. It also sets a variable to tell the software not to continue moving in that direction. The variables are used at the end of the loop to make sure that the arm doesn't continue to move even if it's trying to aim in that direction in the following code (from later in the main loop):

```
if correction_x < -100 and stop_r == False and drawn == True:
    stop_g = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,1,0], 100)

if correction_x > 100 and stop_g == False and drawn == True:
    stop_r = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,2,0], 100)

if correction_y < -100 and stop_u == False and drawn == True:
    stop_d = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [16,0,0], 100)

if correction_y > 100 and stop_d == False and drawn == True:
    stop_u = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [32,0,0], 100)
```

These **if** statements means that when the arm moves in one direction, it resets the stop value for the opposite direction.

The variables **correction_x** and **correction_y** hold the distance between the face and the centre of the image. As you can see, this isn't a precision machine, so anywhere with a hundred pixels is close enough. There are a couple of reasons for this. The assembly isn't particularly stiff (so it's prone to wobbling slightly) and the movements of the arm aren't very fine. 100 pixels is an arbitrary amount, so you could increase or decrease it should you build such a weapon, but we found it was accurate enough to hit a person most of the time, and loose enough that it stopped the gun constantly over-correcting. When we used smaller units, the arm became prone to getting stuck in a loop as it moved from too-far one side to too-far the other.

Facial recognition

Facial recognition is quite a complex area that requires specialist algorithms and lots of training images to teach the computer what a face looks like. Fortunately, all the hard work has been done and packaged into OpenCV. This is a cross-platform library that can be used with many popular languages. In Python, the **cv2** module provides us with the facilities we need. Most distros have this in their package manager. For example, on Debian-based distros, you can install it with:

```
sudo apt-get install python-opencv
```

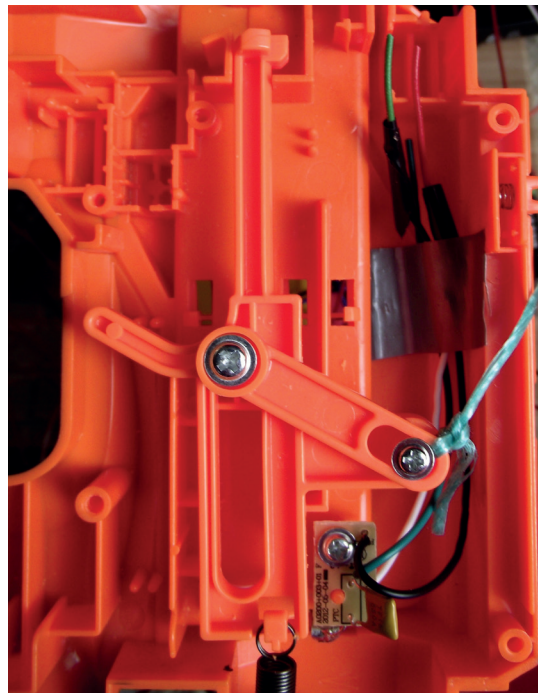
If it's not in your distro's repositories, you'll have to install it via the instructions at http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html.

Once it's installed, you should be able to import **cv2**. This module enables you to use an image recognition method known as Haar cascade to detect items in an image using the Cascade Classifier object.

The exact details of how **cv2** identifies faces is quite complex, so we won't deal with it here. Instead, we'll just look at how you can use it in this software.

CascadeClassifier is a type of object that's included in the **cv2** module. In order to create one, you need a Haar cascade data file. These are XML files that include all the data the classifier needs to find a particular type of object. Each different object to be recognised needs a different Haar cascade.

Your OpenCV installation should have included some useful Haar cascades such as hands, eyes and



The inside of gun with the string tied to the trigger assembly. This is all that was left after we removed superfluous parts of the mechanism.

smiles. We'll use one for detecting faces called **haarcascade_frontalface_default.xml**. If you can't find it in your installation, you can download it from <https://github.com/Itseez/opencv/tree/master/data/haarcascades>.

Before we get to the main loop, we need to create the cascade with:

```
import cv2
face_data = cv2.CascadeClassifier('/home/ben/haarcascade_
frontalface_default.xml')
```

The code inside the loop is:

```
return_val, frame = capture.read()

gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
small_gray_frame = cv2.resize(gray_frame, (0,0), fx=factor_
down, fy=factor_down)
faces = face_data.detectMultiScale(small_gray_frame, 1.5,5)

drawn=False
for (face_x, face_y, face_width, face_height) in faces:
    cv2.rectangle(frame, (face_x*factor_up, face_y*factor_up),
        (face_x*factor_up + face_width*factor_up,
        face_y*factor_up+face_height*factor_up),
        (255,0,0),2)
    if not drawn:
        face_middle_x = factor_up * (face_x + (face_width / 2))
```

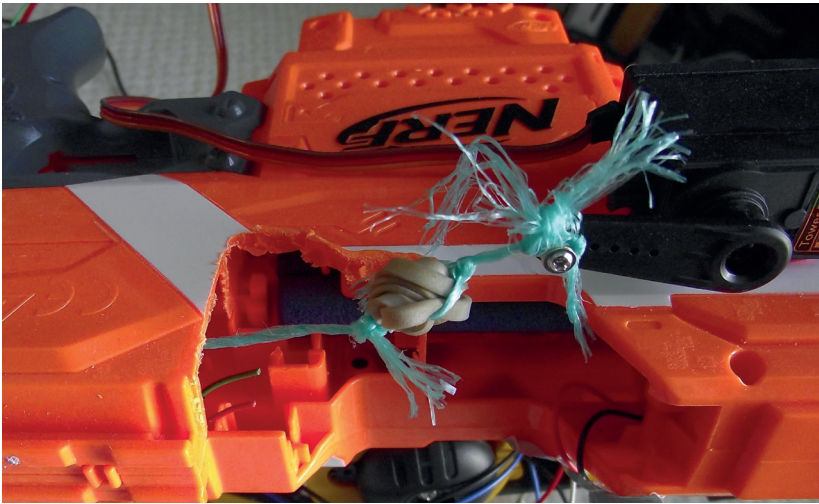
Taking things further

There are plenty of things you could do to make this project better. So far, we've only used one of the standard Haar cascade data files. However, you can create these yourself to recognise specific objects. There are details of how to do this at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

With a bit of practise, you could get it to not shoot at you, or recognise people wearing specific sports-team's shirts.

If you're feeling adventurous, you could even have a go at robotic clay-pidgeon shooting (although it would probably be best to use something a little slower, such as balloons).

This sort of face-tracking doesn't have to be used for evil though. You could use exactly the same hardware (minus the gun) to allow you to have a video chat while wandering about the room, or to video a lecturer who walks about on stage, or even for a more advanced wildlife camera.



The outside of the gun showing the servo and the cutaway that allows it to pull the firing pin forwards.

```

face_middle_y = factor_up * (face_y + (face_height / 2))
drawn = True

correction_x = (real_width/2) - face_middle_x
correction_y = (real_height/2) - face_middle_y
    
```

The higher the resolution on an image, the more accurate the object detection will be. However, it will also take more time to process. The best trade-off will vary depending on your computer, so we've created two variables (**factor_up** and **factor_down**) that can be used to resize the image before and after

processing so that a nice large image can be displayed, but only a smaller one processed. The values of these are set at the start of the program. We found

“The software will try to target the middle of the camera, not what the gun is pointing at.”

that 3 and 0.33 worked well for a moderately powerful computer, but you may wish to vary this depending on what you're running on. It also converts it from colour to greyscale for the same reasons.

The `detectMultiScale()` method is then used to pick out all the faces in the image. The `for` loop then draws a blue box around every detected face, but only one face (the first one) is targeted at any one time. This is then used to calculate the values of **correction_x** and **correction_y** that we used earlier.

Guns before butter

You may notice that this will aim right in the middle of the face. That's a little unfriendly, and not completely safe. Although the darts are soft, so are eyes. A couple of things make this a bit safer. Firstly, the software will try to target the middle of the camera, not what the gun is pointing at. We angled our camera up slightly which meant that the gun was pointing below the face when the face was in the centre of the image. Secondly, we wore eye protection (sunglasses) when getting everything set up, and ideally all the time when the gun is on. Remember that your computer doesn't feel guilt or compassion, so will shoot you straight in your eye and not feel a drop of remorse.

It is possible to calculate the values of **correction_x** and **correction_y** in different ways. For example, you could change them to target a half head's distance below the head (upper-chest) by changing the calculation to:

```
correction_y = (real_height/2) - face_middle_y - face_height
```

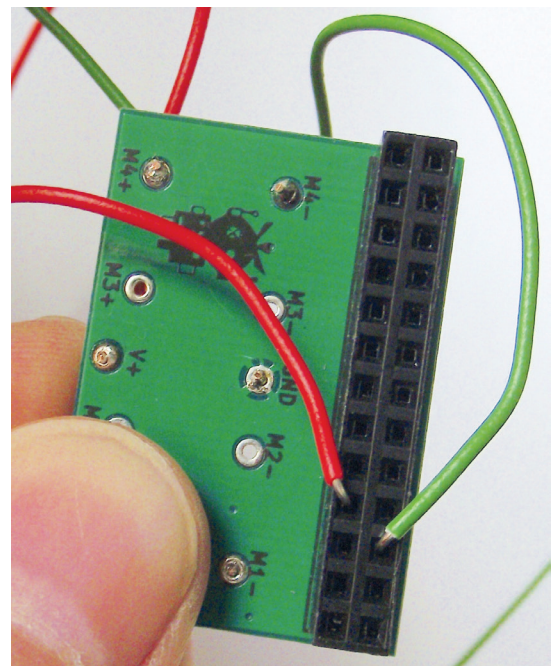
However, this can cause problems in close-quarters combat, because the head takes up a large proportion of the image. By moving the camera upwards, it may push the face off-camera and therefore not recognise it and fail to shoot.

The final part of the control is the part that handles the shooting. This is actually the most complex part of the code because it has to handle a few timing problems. The disc motors need to have time to spin up to speed before firing, but we don't want them to spin permanently because they will just deplete the batteries. Therefore, we turn them on as soon as we detect a face, but have to wait a little while before shooting.

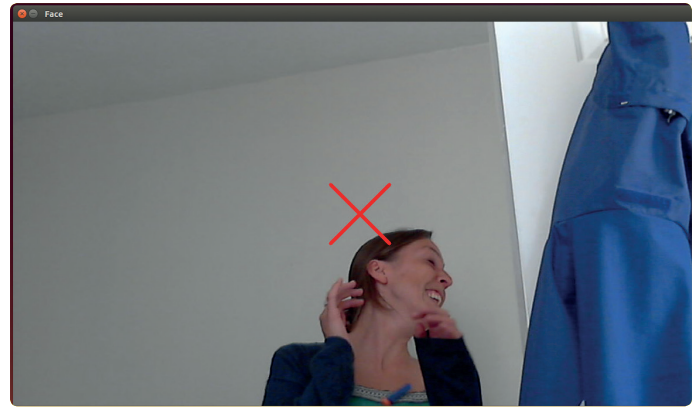
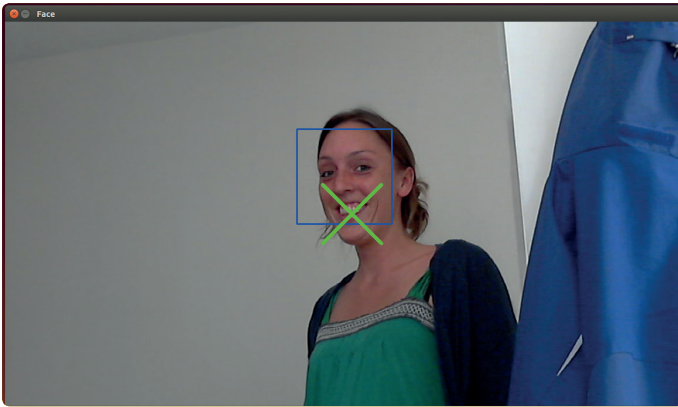
We want to turn the motors off when the face leaves the image, but not immediately because the face might still be there, just not recognisable for a few frames, and we always want to be primed and ready to fire.

The firing sequence goes: if the face is in the middle of the frame and the trigger is reset and the motors are running, then send the Arduino the message to move the servo, then wait until you're sure the servo has moved, then reset the servo.

We can't use the normal sleep functions for all the waiting, because we still want the software to keep running through the loop and targeting. Instead, we're going to use counters that make sure at least a certain number of iterations of the loop are run each



The Picoborg motor controller is designed to go on a Raspberry Pi, but we can co-opt it for use with the Arduino.



time. The code that controls this is:

```

if drawn == True:
#continue to send the message periodically in case there's an
error in transmission
if spin_count%20 == 0:
    ser.write('4\n')
if spin_count < 20000:
    spin_count = spin_count + 1
else:
    spin_count = 20
if drawn == False:
    not_spin_count = not_spin_count + 1
if not_spin_count > 30 and triggered == False:
    spin_count = 0
    ser.write('3\n')
    not_spin_count = 0
if triggered_count > 30 and triggered == True and resetting ==
False:
    ser.write('1\n')
    reset_count = 0
    resetting = True
if reset_count > 30 and triggered == True and resetting ==
True:
    triggered = False
    resetting = False
if reset_count < 31:
    reset_count = reset_count + 1
if triggered_count < 31:

```

Warranty

If you're following this tutorial, you're doing stuff with hardware that it wasn't designed to do. There's no point in taking a dismantled and sawn Nerf gun back to the shop you bought it from if it breaks. They'll laugh you out of the store.

We've built an automatic robot-controlled gun, and it worked for us, but we can't guarantee it'll always work. You might have received a Nerf gun from a different batch (and they don't have published tolerances), or a servo with a bit more power. We don't think you're likely to end up with a smoking heap if you follow the tutorial, but we can't say for sure that you won't. Such is the nature of hardware hacking.

In the course of this project, we managed to burn out two pins on our Arduino (fortunately, the rest of the board still works). It was a lesson to us in checking our wiring before powering on, and a reminder that electronics are fragile.

We've written this as a guide only. For those brave enough to attempt it: good luck.

triggered_count = triggered_count + 1

We've covered all the mechanics, but there are a few more bits of code needed to get everything set up. The full code is at www.linuxvoice.com/wp-content/uploads/code/lv06-gun.tar.gz.


Once the software's fully tested, the only thing to do is stick everything together properly. We used electrician's tape to attach the webcam to the gun, since this allows us to easily remove it once we want to move on to the next project. The joint between the gun and the arm needs to be more solid. Here, we used Sugru, which worked well, but hot glue would also do the job. Actually, most strong glues that stick plastic should work well. We used blobs of Blu-Tack to hold the wires in the Picoborg and servo connectors.

Physical computing

This project is all about physical computing – that is, getting computers to interact with the real world. It takes some inputs (the end-stops and the camera images), performs some processing on them, and generates some outputs (turning the gun and firing the bullets).

This is quite a complex project, but physical computing doesn't have to be. If you want to explore some simpler projects, the Arduino Uno is an excellent place to start. It connects to your Linux PC via the USB port and lets you turn pins on or off, or get input from them.

By unloading this onto an external board, if you accidentally make a mistake, the worst it can do is fry the Arduino, leaving your computer intact. Most people start with projects that turn LEDs on and off, get input from switches, and build up to incorporating sensors into their projects, although there's nothing to stop you jumping in at the deep end, and building a Nerf gun controller for your first project.

The Arduino board and the software it uses are both open source, so there are loads of re-mixed designs with all sorts of features built in or taken out. There's a great community around the Arduino, and loads of hardware available. <http://playground.arduino.cc> is a great place to see what's going on. 

Ben Everard doesn't feel pity, or remorse, or fear, and he absolutely will not stop, ever – at least not until tea time.

The gun successfully defending the desk of the author from an interloper attempting to distract him from his work.