Coding: Make

You asked for more, so here we go: **Mike Saunders** puts pedal to the metal with a car-based driving romp in just under 100 lines of code. Vroom!

LINUX FORMAT On the DVD
» Tutorial code
» SoundTracker 0.6.8

Two issues ago, we ran a one-off programming tutorial in which we wrote a mini *Space Invaders* clone called *PyInvaders*. The feedback has been fantastic: from the forum posts and emails that we've received, it's clear that lots of you have enjoyed learning new programming techniques and enhancing the code. Many thanks! This month we're kicking off a three-part series continuing the game programming theme – and first in the line is a top-down racer.

As before, we're going to use *PyGame*, a combination of the effortlessly readable Python language and the SDL multimedia libraries. Python's syntax is as simple as you can get, unladen with curly brackets and other clutter that you typically see in other languages. This makes it ideal if you're fairly new to coding: these tutorials don't aim to teach programming from scratch, but if you've played around with a bit of code before, you won't find them hard to follow.

If you've never seen a line of Python before, grab a copy of **LXF110** (see back issues on page 106) which includes a quick primer. This month we're going to expand on our *PyGame* knowledge in three key areas:

» **Mouse handling** Whereas *PyInvaders* solely used the keyboard, here we'll demonstrate how to track the mouse position to manipulate the player.

» **Playing music and sounds** What's a game without some toe-tapping tunes and crisp sound effects? Pretty drab, that's what. We'll use *PyGame* to get your speakers doing their job.

» **Using text and fonts** We only used sprites in *PyInvaders*, but in many games you'll want to incorporate an on-screen score display. Thankfully, *PyGame* includes some excellent routines to create and manipulate text in just a few lines of code.

Our expert

Mike Saunders once hacked the NetBSD kernel to provide Alt+cursor key virtual terminal switching. Nowadays he mainly works on his operating system at **http://mikeos.berlios.de**.

Part 1 Getting the right files

To make *PyRacer*, our top-down view driving game, we're going to need some audio and graphics files. If you're feeling in a creative mood (especially likely after you've read our cover feature!), you can craft these files yourself using the guidelines below. If not, see the **Magazine/PyRacer** section of the DVD for a tarball containing example files.

PyRacer will be contained in a single source file, **pyracer.py**, with a directory called **data** next to it. In **data** you'll need:

» **car_player.png** A 40x100-pixel image of the player's car (overhead view with the car facing north). Use PNG transparency to cut out non-car bits, so it's not one solid rectangle and the road will be visible beneath.

» **car_enemy.png** As above, but flipped vertically.

» **whiteline.png** A 20x80-pixel line, and solid white (or textured if you're going for eye candy). As you can guess, this will be used for the markings in the middle of the road.

» **tree.png** A 65x110-pixel image of, yep, a tree! This will be repeated and moved down the side of the road to give a sensation of speed. As with the car images, use transparency outside of the trunk and leaves so that the background will show through.

» **background.mod** The background music in MOD format, as generated by, for example, *SoundTracker* (see the Audio section of the DVD). You can easily create your own MOD tunes, or search online for more (beware of copyright though!).

» **crash.wav** A lovely, crunching sound effect to be played when the player's car hits the enemy car.

As mentioned, these are all supplied on the DVD if you don't want to create your own. Or why not recruit someone else to handle the media side while you concentrate on the code? If the kids are bored, throw them a copy of *Gimp* and tell them you want some top-class sprite images on your desk by the morning, or you're changing the locks next time they're out.

» **In LXF110** We made a complete Space Invaders clone in just a few lines of code.

a racing game

Part 2 Start your engines

Let's move on to the code. From here onwards you'll see the code that's in the tarball on the LXFDVD, split up into chunks for our explanations. Note that Python code is indented with tabs – if the indents look smaller than tabs here, that's just to stop the code wrapping around multiple lines of print where necessary. Also remember that you can add comments to your code using hash (**#**) marks, which is useful for marking reminders or explanations.

```
from pygame import *
import random
```

The first two lines set up our Python environment, saying that we want to use all the functions of the *PyGame* library (hence the asterisk wildcard), and also random-number functionality. Without these lines, we'd only be able to use the bare functionality supplied with Python, so importing extra modules is essential.

```
class Sprite:
	def __init__(self, xpos, ypos, filename):
		self.x = xpos
		self.y = ypos
		self.bitmap = image.load(filename)
	def render(self):
		screen.blit(self.bitmap, (self.x, self.y))
```

You may remember a chunk of code similar to this in the *PyInvaders* tutorial. A class, in object oriented programming terms, is a description for a chunk of data and associated routines. Imagine it as a blueprint for a box, explaining what the box can store, and how it can be used. The **class Sprite** definition here just defines the blueprint – it doesn't do anything yet, as we have to create real boxes (instances of the class) later on.

Get started, innit?

The **__init__** routine is called when we create a new instance of the Sprite class, and as you can see, it takes four parameters. Actually, we can forget about **self**, as that's internal to the class – so when creating a new Sprite object, we pass its initial X and Y positions, plus a filename that's used to load the sprite image. This class also includes a render routine which displays the sprite on the screen; we'll see how that's used later on.

```
def Intersect(s1_x, s1_y, s2_x, s2_y):
	if (s1_x > s2_x - 40) and (s1_x < s2_x + 40) and (s1_y > s2_y - 
40) and (s1_y < s2_y + 40):
		return 1
	else:
		return 0
```

You may also remember this beast from *PyInvaders*. This is the collision detection routine, and looks almost unfathomable at first, but it's not actually that complicated. Basically, it takes four parameters: the X and Y positions of one sprite, and the X and Y positions of another. It then determines whether or not those sprites overlap at all on the screen, returning 1 if so or 0 if not. You can see many references to the number 40 – that's the width of the car sprites. (So yes, those numbers are hard-coded and in a bigger game you'd want this routine to be independent of sprite sizes, but we're striving for simplicity here.)

```
init()
screen = display.set_mode((640,480))
display.set_caption('PyRacer')
```

Next up we tell *PyGame* to get out of its chair and start working with the **init()** routine, and then set the video mode and caption that will appear in the window manager's title bar.

```
mixer.music.load('data/background.mod')
mixer.music.play(-1)
```

Here's our first foray into the world of sound. *PyGame* includes a module called **mixer**, which handles music and sound effects, and in the first line here, we're loading the MOD file that will play in the background. The second line sets the music playing, and you can pass it a parameter stating how many times you want the piece of music to be played. If you pass **-1**, as we're doing here, it means "Play the music in a loop forever" (well, until we quit the game).

```
playercar = Sprite(20, 400, 'data/car_player.png')
enemycar = Sprite(random.randrange(100, 500), 0, 'data/car_
enemy.png')
tree1 = Sprite(10, 0, 'data/tree.png')
tree2 = Sprite(550, 240, 'data/tree.png')
whiteline1 = Sprite(315, 0, 'data/whiteline.png')
whiteline2 = Sprite(315, 240, 'data/whiteline.png')
```

»

Quick tip

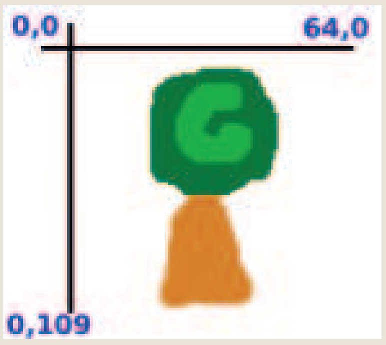
You can make your Python programs run without having to type in **python program.py** by making the file executable (**chmod +x program.py**, for example) and adding a line at the top of the code: **#!/usr/bin/env python**. Now you can just run **./program.py** or double-click it in a file manager.

Co-ordinate yourself

Working with the screen and sprites can be a bit perplexing for novices, because it takes a certain way of thinking to address particular points on the screen. When we consider graphs, for instance, we're used to them starting in the bottom-left, with increasing values in both axes moving our plotting points up and to the right.

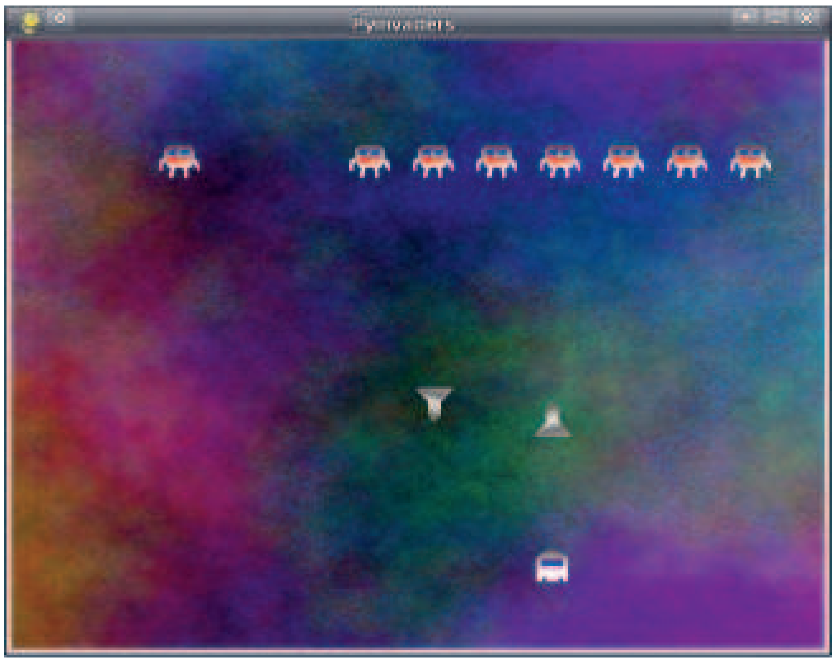
With computer graphics (most of the time!) we work from the top-left, moving downwards and rightwards. So, if you've set up a game window that's 640x480 pixels, position 0,0 will be zero pixels across and zero pixels down. That's the top-left. If you move to 0,240, that's zero pixels across and 240 down – ie halfway down the screen. Position 320,240 is the middle of the screen, and 639,479 is the furthest bottom-right pixel.

Hang on... why 639 and not 640? Well, in typical computing practice, we start counting from 0, so 639 is the same as 640 if we were counting from one. It's these little things that always keep you on your toes when hacking!



› **The top-left of our tree sprite is addressed by 0,0 and the bottom-right by 64,109 (we're counting from zero).**

» **If you missed last issue** Call 0870 837 4773 or +44 1858 438795.



› In LXF110 we made this funky *Space Invaders* clone – see page 106 to get your hands on a copy.

Now we start to use the Sprite class that we defined at the start of the code. In line 1 we say that we want a new Sprite object called **playercar** with an initial X position of 20 (ie 20 pixels across the game window), an initial Y position of 400, and using **car_player.png** in the data directory as a sprite image.

We do the same for the enemy car – the car that's coming from the opposite direction. We only need to create one enemy car, because when it goes past and off the bottom of the window, we can start it from the top again at a different position. Then we create two tree sprites, one for each side of the screen, and two white-line sprites that will scroll down the middle of the screen to enhance the illusion of movement.

```
scorefont = font.Font(None, 60)
crasheffect = mixer.Sound('data/crash.wav')
```

The last two things we need to set up are our font for the score display, and a sound effect for collisions. For the former, we call on *PyGame*'s 'font' routines to create a new font object, passing 'None' to say "Give me a generic font" (you could be more specific, but it might impede portability of the game to other platforms). The '60' specifies the font size. For the latter, we create a new sound object called **crasheffect** using the **data/crash.wav** file.

```
score = 0
maxscore = 0
quit = 0
```

Before we kick off the main game loop, we'll set up a few variables. Note that in Python, you're not forced to initialise variables in this way and can often just start using them when required, but doing this makes the code much easier to understand – you know what's coming up. **Score** is the current score, **maxscore** holds the highest score attained as the game progresses (it's not reset when the cars collide), and **quit** changes to 1 when the game window receives a close event (ie the user clicks the X in the title bar).

```
while quit == 0:
  screen.fill((0,200,0))
  screen.fill((200,200,200), ((100, 0), (440, 480)))
```

So, here's the start of our main loop. While **quit** is set to zero, we execute the following indented code. At the beginning of every game loop, we draw the background, first filling the entire screen with a green colour. 0,200,0 is an RGB (red, green, blue) triplet, so here it's set to zero red, a fairly strong green (200) and zero blue. (The maximum value for each of these is 255.)

The green, green grass

The second **screen.fill** line draws the grey road on top of the grassy green background. Because the road doesn't fill up the entire screen, we pass two extra parameters – the starting point (100 pixels across and 0 pixels down), and the finishing point (440 pixels across and the bottom of the game window).

```
  tree1.render()
  tree1.y += 10
  if (tree1.y > 480):
    tree1.y = -110
  tree2.render()
  tree2.y += 10
  if (tree2.y > 480):
    tree2.y = -110
```

Now we draw the tree objects: one on the left-hand side of the road, and one on the right. When we initialised these objects earlier in the code, we gave **tree1** a starting vertical position of 0 pixels and **tree2** a position of 240. So, one starts at the top of the screen, and the other starts in the middle, to add a bit of variety to the scenery.

In each game loop, we call **render()** on the tree objects, and increment their vertical positions down the screen by 10 pixels. When the trees move off the bottom of the screen, we reset their positions to above the top of the game window – hence -110 rather than 0. If we used 0, the trees would magically appear in full at the top of the window, instead of being introduced gradually.

```
  whiteline1.render()
  whiteline1.y += 10
  if (whiteline1.y > 480):
    whiteline1.y = -80
  whiteline2.render()
  whiteline2.y += 10
  if (whiteline2.y > 480):
    whiteline2.y = -80
```

This is very similar to as the tree code, except that it handles the white lines. We reset the positions to -80 when the white lines go off screen, as that's the height of the images in pixels.

```
  enemycar.render()
  enemycar.y += 15
  if (enemycar.y > 480):
```

Power up with new features

Got to grips with the code? Excellent – it's time to start adding more! Try your hand at these extra features...

» **Easy** Random speeds. Right now the oncoming cars always move at the same speed (15 pixels per game loop). You could set up a new variable before the main game loop, then, in play, set it to a random value and update the enemy car accordingly. When the enemy car goes off screen, you set up a new random accelerator value for the next car.

» **Medium** Damage indicator. Maybe resetting the score on collision is a bit harsh? Perhaps the car should be able to take a bit of damage before the score counter is zeroed? This isn't too hard to add, but there are many different ways you could implement it. Try adding a collision indicator to the top-right of the screen that counts up from 0% to 100%, based on the text routines in the game.

» **Hard** More cars. Having multiple enemy cars on the screen at any given point would really flesh out the game, but you have to be careful. You don't want the enemy cars to overlap (it'd look very strange), nor do you want situations to arise where the game is impossible, with too many cars and no escape path.

Let us know how you get on at **www.linuxformat.co.uk/forums**!

```
    enemycar.y = -100
    enemycar.x = random.randrange(100, 500)
```

Here's the snippet of code that handles the oncoming cars. We first draw the car at its current position, then increment its position down the screen by 15 pixels (so it's moving faster than the trees and white lines). If the enemy car moves off the bottom of the screen, we set it to above the top, so that it can gracefully move back into view, and set its horizontal (X) position to a random place between 100 and 500 pixels on the road.

```
  x, y = mouse.get_pos()
  if (x < 100):
    x = 100
  if (x > 500):
    x = 500
  playercar.x = x
  playercar.render()
```

Time for some mouse handling. To get the mouse coordinates in *PyGame*, simply call **mouse.getpos()**, which returns two values – the horizontal position and the vertical position. In our case, the player car stays at the bottom of the screen, so we're only interested in the X (horizontal) value.

Also, when the player is moving the car with the mouse, we want to keep the car on the road (unless you fancy adding some tree collision routines!). So we limit the X position to being at a minimum 100 pixels from the left edge of the game window, and 140 from the right edge. Remember that the car sprite is 40 pixels wide, which means the right side of the car sprite can be at X position 540 according to our limits.

Word up

```
  scoretext = scorefont.render('Score: ' + str(score), True,
 (255,255,255), (0,0,0))
  screen.blit(scoretext, (5,5))
```

Writing text to the screen is very easy in *PyGame*. In the first line, we create a new bitmap image called **scoretext**. Using **scorefont.render()**, we pass four parameters: the text we want to print, whether it should be anti-aliased, the colour of the text, and the background colour. Note that we've joined the word 'Score:' with the actual value of our score numeric variable by converting the latter to a string using **str(score)**.

The second line here simply renders our **scoretext** object to the screen at the specified position. We've indented the score by five pixels horizontally and vertically because it looks a bit purdier.

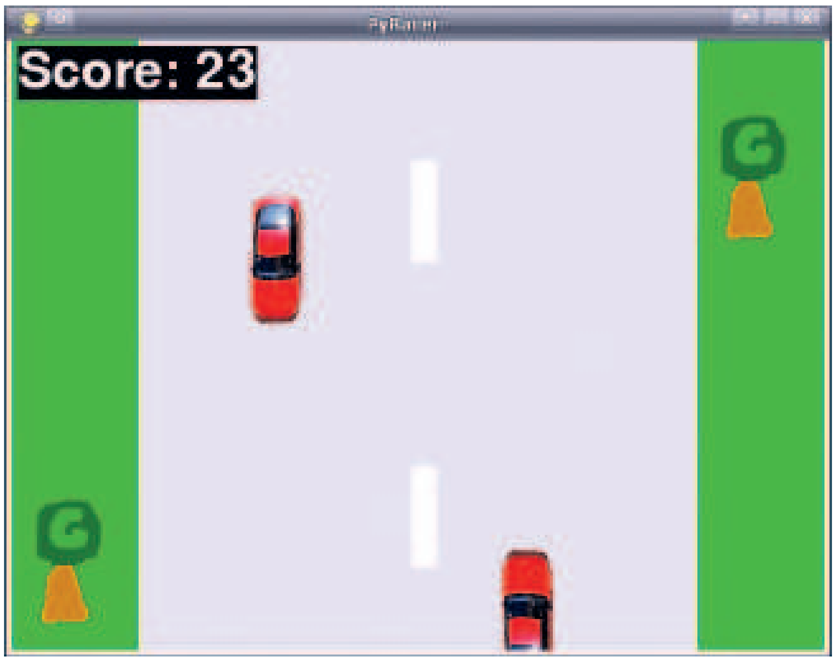
```
  if (Intersect(playercar.x, playercar.y, enemycar.x, enemycar.y)):
    mixer.Sound.play(crasheffect)
    if (score > maxscore):
      maxscore = score
    score = 0
```

Here's where the **Intersect** routine that we defined earlier has its fun. It checks to see if the player car and enemy car sprites overlap in any way; if so, it plays the crash sound effect, checks to see if the current score is the maximum score, and updates **maxscore** if so. Then it resets the current score and continues the game loop.

```
  for ourevent in event.get():
    if ourevent.type == QUIT:
      quit = 1
```

When *PyGame* programs are running, they can receive events from the window manager and keyboard. Here we check to see if there are any events waiting to be processed, and if so, check to see if it's a **QUIT** event (ie the player has tried to close the window). If so, we set our quit variable to 1, which will end the main loop.

```
  display.update()
```



› Check out this leaked screenshot of the next *Gran Turismo* game! Japes aside, graphics aren't a big concern when you're coding – you can prettify things later.

```
  time.delay(5)
  score += 1
```

display.update() is a crucial function in *PyGame* – puts into place all the background and sprite work that we did earlier in the game loop, so without it, nothing on the screen will be rendered. Then we add a delay, so that it won't run insanely quickly on fast machines, and increment the score counter. You'll notice that this is the end of the code indentation, so it's back to the start of the main loop (the **while** bit)!

```
 print 'Your maximum score was:', maxscore
```

And the final line simply spits out the maximum score to the terminal window.

To use the code from the DVD, copy **pyracer.tar.gz** from the **Magazine/PyRacer** section to your home directory, open a terminal window and enter:

```
 tar xfvz pyracer.tar.gz
 cd pyracer
 python pyracer.py
```

That extracts the **.tar.gz archive**, switches into the resulting directory and runs the code. Now you can edit **pyracer.py** to add new features as described in the box, or beautify the sprites in the data directory – happy hacking! LXF

Make it with Mike

Hopefully you've got a lot from this tutorial, and we still have two more to come. The game project in **LXF113** is shrouded in secrecy – as we go to press, only the five richest kings of Europe know what it's going to be about. But stay tuned next month when all will be revealed!

For **LXF114**, the final tutorial, we're opening the floor for suggestions. Tell us what kind of game you'd like to see, and we'll get Mike to write the code with detailed explanations in the mag! The game has to be simple; we can't recreate *Grand Theft Auto IV* in 150 lines of code. It'd also be good to make use of all the techniques we're practising – keyboard handling, mouse co-ordinates, fonts, sounds and so forth.

So, if you've got a corking idea that can be realised in under 150 lines of code, go to **www.linuxformat.co.uk/gamemaking** and post a message in the forum thread. We'll try to reply to every suggestion.

» **Next month** More gaming fun to look forward to, as we expand our PyGame skills.