How not to code: a guide to concise programming

Updating a 22-year-old game brought Andrew face to face with some very poor coding practices

AUTHOR
ANDREW GILLETT

Andrew Gillett is a tutor and programmer who has worked on ten published games including *RollerCoaster Tycoon 3*, *LostWinds 2*, and *Kinectimals*.

Download the code from GitHub: **wfmag.cc/wfmag48**

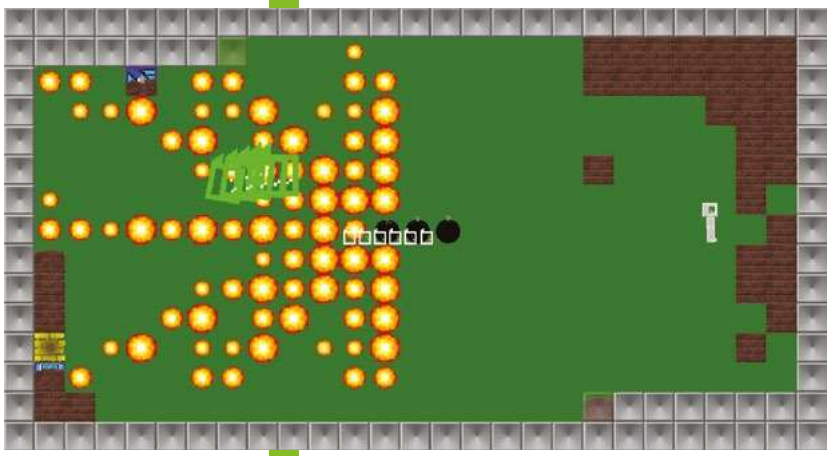
In 1998, at the age of 17, I was learning how to write games in C. My first attempt, the subtly titled *DEATH*, was not going well. The game was my take on *Hardcore*, a 1992 Atari ST game by legendary game developer and sheep enthusiast Jeff Minter, which had been released only as an unfinished five-level demo. The player controlled four gun turrets on the outside of a square arena, into which enemies teleported. While the original game had been enjoyable and promising, my version wasn't much fun, and I couldn't work out why. Making a decent game would also have involved making dozens of levels and many enemy types, which was looking like too big a task, especially as I was finding it hard to understand the intricacies of how the enemies in *Hardcore* moved.

So I abandoned that game and decided to replicate a different one – 1994's *MasterBlaster*, a *Bomberman*-style game on the Commodore Amiga. *MasterBlaster* didn't have a single-player mode or bots, so there was no enemy AI to write. And the level was just a grid with randomly generated walls and power-ups – so there was no real level design involved. With those two hurdles removed, development went fairly smoothly, the biggest challenge being working out some of the subtleties of how character movement worked.

The game, which I named *Partition Sector*, was finished in mid-1999 and spent the next 18 years on my website being downloaded by very few people. In late 2018 I decided to do a quick update to the game and release it on Steam. Then I started having ideas, and ended up working on it, on and off, for two years.

One of the biggest hurdles I came across when writing my first games was how to structure the code. I knew how to write a basic game loop, in which you update the positions of objects within the game, then draw the level and the objects within it, and then loop back to

A series of ultrabombs blowing up a snake.



The 2021 version of *Partition Sector*.

the start, ending the loop when the 'game over' criteria are met or the player has chosen to quit. But for a full game you need things like a main menu, submenus, going through a particular number of rounds before returning to the main menu, and so on. In the end, I was able to come up with something that worked, but looking back on my old code 20 years on, I could see many cases of absolutely terrible practice. While most of my time was spent adding new features, a lot of time was spent rewriting and restructuring old code. I'm going to share some examples from the original code so you don't make the same mistakes!

"I started having ideas, and ended up working on it, on and off, for two years"

AVOID REPETITION

In the original *Partition Sector* code, when the player moved the analogue stick up to move up the screen, there was a section of code which checked for this and then applied the movement. This included checks such as: is the player currently piloting a remote-control bomb, and is there space for the bomb to move up? If so, apply the movement. If the player isn't remote-controlling anything, check to see if there's space to move the player character up. This includes checks for either a wall or a bomb being in the way. If the player has the ghost power-up, they can walk through most, but not all, walls, so we have to check for that too. If the player's able to move, we move them up a small amount, then also do a small movement left or right if they aren't fully aligned with the vertical lane they're moving in. If movement is blocked by a wall or a bomb, and the player has the 'Strength' power-up which allows them to push such things, we apply the push effect, first making sure there's a free space for the wall or bomb to be pushed into. Finally, if the player is blocked by a wall, we might apply what I call secondary movement to the left or right, moving the player in the direction of an available lane. There's also code for updating the walking animation. In total, the code for the player moving up was 155 lines.

MOVEMENT LOGIC

Having written this code, I then proceeded to copy and paste it three times, for the down, left, and right directions. This is extremely bad. The code for moving down was identical to the code for moving up, except that it was checking for blockages and applying movement in the opposite direction. The same was true of the ➔

OTHER LANGUAGES

Each programming language has its own way of giving you access to other code files – in C# and Java you can automatically use public classes from other files within the same project, while in C++ you have to use header files (with include guards or #pragma once) to declare shared functions and classes. When working in Python, note that it won't let you have circular imports – for example, writing `import file2` in **file1.py**, and `import file1` in **file2.py**.

left/right code, except it checked and updated the X-axis instead. One of the biggest issues with this code was that any change to the logic had to be applied four times – once for each direction. Besides the increased workload, every time I made a change, there was a risk that I'd forget to apply it to one of the directions, in which case I'd end up with a character who obeyed different movement rules depending on which direction they were moving.

The solution was to create a function, `move`, with two integer parameters – `xDir` and `yDir`. A value of zero for either of these means 'Don't try to move on this axis', while values of -1 or 1 correspond to moving up/left or down/right. Each frame I get the control inputs and call the function with the appropriate values. Inside `move`, instead of specifically checking for a wall above, below, or to the left or right of the player, I instead work out which grid square the player would be in if they were to move in the specified direction, then check to see if there's a wall or bomb in that square. Writing generic code like this can sometimes be slightly harder than writing separate code for each specific case, but it's a vital skill to practice. Part of what separates experienced and inexperienced programmers is the ability to recognise opportunities for reducing code duplication.

"Experienced programmers recognise opportunities for reducing code duplication"

The code in **Figure 1** and **Figure 2** shows two simplified re-creations of the player movement code in Python. The second example has been split into multiple code files. In each case, grid squares are 64×64 pixels, and the player moves at a speed of two pixels per frame. In the first (bad!) code example, the update method in the `Player` class has separate movement code for each direction. If we take the example of trying to move left, we only allow the player to move if there isn't a wall immediately to their left. We just check a single point, which is halfway down the sprite on the Y-axis, and just past its left edge on the X-axis (in this example, the player's X and Y coordinates represent the top left of the sprite). We convert the selected pixel coordinates to grid coordinates and check for a wall – in the GRID list at the top of the code, each row is represented by a string, where a space represents an empty square, while 'X' represents a wall. If the proposed new position is empty, we apply the movement. We then check to see whether the player is perfectly aligned with their current horizontal lane. If not, we apply a small change to the Y position, moving it in the direction of being centred in the lane.

In **player.py** in the second set of example code, there's only one set of movement code, which deals with all four directions. You'll also notice that constants are used for the grid square size, half the grid square size, and the player movement speed. First, we set the variables `x_dir` and `y_dir` to -1, 0, or 1 to indicate which direction (if any) the player is trying to move on each axis. It uses the `Controls` classes, which are explained later on. The game doesn't allow for diagonal movement, so if the player's →

› Snake Hunter mode.

FIGURE 1

```
import pgzero, pgzrun

WIDTH,HEIGHT = 576,320

GRID = ['XXXXXXXXX',
        'X       X',
        'X X X X X',
        'X       X',
        'XXXXXXXXX']

class Player(Actor):
    def __init__(self,pos):
        super().__init__('player', pos,
anchor=('left', 'top'))

    def update(self):
        if keyboard.left:
            x = int(self.x) - 2
            y = int(self.y) + 64 // 2
            grid_x = x // 64
            grid_y = y // 64

            # The square ahead does not have a wall
            if GRID[grid_y][grid_x] == ' ':
                # Apply movement
                self.x -= 2

                # Lane alignment
                if y % 64 < 64 // 2:
                    self.y += 1
                elif y % 64 > 64 // 2:
                    self.y -= 1

        if keyboard.right:
            x = int(self.x) + 64 + 2 - 1
            y = int(self.y) + 64 // 2
            grid_x = x // 64
            grid_y = y // 64
            if GRID[grid_y][grid_x] == ' ':
                self.x += 2
                if y % 64 < 64 // 2:
                    self.y += 1
                elif y % 64 > 64 // 2:
                    self.y -= 1

        if keyboard.up:
            x = int(self.x) + 64 // 2
            y = int(self.y) - 2
            grid_x = x // 64
            grid_y = y // 64
            if GRID[grid_y][grid_x] == ' ':
                self.y -= 2
                if x % 64 < 64 // 2:
                    self.x += 1
                elif x % 64 > 64 // 2:
                    self.x -= 1

        if keyboard.down:
            x = int(self.x) + 64 // 2
            y = int(self.y) + 64 + 2 - 1
            grid_x = x // 64
            grid_y = y // 64
            if GRID[grid_y][grid_x] == ' ':
                self.y += 2
                if x % 64 < 64 // 2:
                    self.x += 1
                elif x % 64 > 64 // 2:
                    self.x -= 1

player = Player( (64,64) )

def update():
    player.update()

def draw():
    screen.clear()

    for row_index in range(len(GRID)):
        for column_index in range(len(GRID[row_
index])):
            if GRID[row_index][column_index] != ' ':
                x, y = column_index * 64, row_index
* 64
                screen.blit('gridblock',(x,y))

    player.draw()

pgzrun.go()
```

FIGURE 2

```
import pgzrun, pygame
from shared import *
from player import Player
from controls import KeyboardControls,
JoystickControls

WIDTH,HEIGHT = 576,320

controls = JoystickControls(0) if pygame.joystick.
get_count() > 0 else KeyboardControls()
player = Player( (64,64), controls )

def update():
    player.update()

def draw():
    screen.clear()

    for row_index in range(len(GRID)):
        for column_index in range(len(GRID[row_
index])):
            if GRID[row_index][column_index] != ' ':
                x, y = column_index * GRID_SQ_SIZE,
row_index * GRID_SQ_SIZE
                screen.blit('gridblock',(x,y))

    player.draw()

pgzrun.go()
```

Figure 1: An example of bad code. Sure, it works, but look at all that duplication.

Figure 2: An example of good code. Look how much more compact it is than the previous sample.

DIFFERENT CLASS

An abstract class is one that cannot be instantiated – or in other words, you can't create an object based on it. Its purpose is to serve as a base class for other classes, which implement the specified abstract methods.

pressing keys for both axes at once, we ignore the up and down keys. If either `x_dir` or `y_dir` is non-zero, the player is trying to move. The variable `horizontal` exists purely for readability, and is set to `True` or `False` depending on whether the player is trying to move on the X-axis. We calculate the centre position of the player, then work out which grid position we're going to check for a wall, based on the axis and direction. The function `get_grid_pos` is used to convert from pixel to grid coordinates. If no wall is present, we apply the movement by updating both the X and Y coordinates based on `x_dir`/`y_dir` multiplied by `SPEED`. Although we only want to move on one axis, the variable for the other axis will be zero, so no movement will take place, without the need for an `if` statement. Finally, for lane alignment, we work out the position we'd like the player to be aligned with, and the `move_towards` function, defined, above the `Player` class, ensures that they will move in the necessary direction, without overshooting the target position.

SPLIT CODE INTO FUNCTIONS

The game's original C code contained a function named `playTheGame`, which was over 1200 lines long. This function was called when the player selected Start Game from the main menu, and handled setting up and playing each round of the game. Within this function, the largest section of code was the main game loop, which included updating each player (primarily dealing with movement and the dropping of bombs), updating bombs and explosions, updating the walls which shrink in towards the centre of the level, and drawing everything to the screen.

```
updatePlayers();
updateUncarriedFlags();
updateKOTH();
updateBombs();
updateExplosions();
updateDeath();
updateShrinking();
updateBattleRoyaleTeams();
updateRespawningWalls();
updateSnakeHunterSpawning();
```

› Moving code into separate functions makes it easier to see the big picture.

› Don't forget that a finished game needs menus.

It also checked to see if the round needed to end (because a player had won, for example). Apart from the code which updates bombs and explosions, none of this had been separated into separate functions. In the new version, the code and data for the game logic have been moved into a class named `Game`, and that class contains methods such as `updatePlayers`, `updateBombs`, `updateShrinkingWalls`, and so on. The code for the main game loop largely consists of calls to these new methods, making it shorter and more readable.

SPLIT YOUR CODE

The code for the 1999 version of the game consisted almost entirely of a single C++ file which was over 5500 lines long. At the time, I didn't know how to split my code into multiple files. These days I try to keep code files as short as possible – preferably no more than 1000 lines, although this isn't always possible. Ideally, each file should implement one class or module.

The improved version of the Python code shows how to split your code into multiple files. The main file is **maze_good.py** which runs the game, and uses the `from`/`import` statements to load in other Python files. The `Player` class is contained in **player.py**, the keyboard and gamepad controls are contained in **controls.py**, and **shared.py** contains constants and functions which need to be accessible from multiple files.

AVOID GLOBAL VARIABLES

The original code featured many variables that needed to be accessed from multiple functions. For example, there was an array of players, mainly accessed from the main game function, but also used in functions for getting the sprite for the current animation frame, drawing the players on the screen, and several others. My solution at the time was to use global variables, which can be accessed from any function. Beginner programming courses advise you to minimise your use of global variables, as they

King of the Hill mode features a scoring zone with flashing disco tiles.

can make it difficult to keep track of where variables are changed. Another problem with this approach is that it can lead to old data persisting. When I created a new variable, I had to remember to reset it each time a new game is started, otherwise, the value of that variable from the previous game would carry over to the new game. Usually this led to obvious bugs which were easily fixed, but occasionally it led to more subtle bugs which weren't noticed for some time. Encapsulating these variables inside the `Game` class solved this issue, as the game object is re-created at the start of each game, ensuring that all variables contained within it are automatically set to their default values.

BEHAVIOUR AND CLASSES

Each player needs one or more variables indicating which controls they are using. For example, one player may be using a particular controller; another player may be using the arrow keys; a third player might be using different keys. In the original code was a variable indicating if the player was using a keyboard or a controller, and a variable specifying their controller or keyset number. Each time I wanted to check whether a player was pressing their controls in a particular direction, I would first have to check which type of controls they were using, and then had to check the input for the relevant control type.

A better approach is to have a class named `Controls`, which exists purely as a base class for other classes that implement specific control methods. The `Controls` class itself isn't intended to be instantiated: it doesn't represent any particular control method, just controls in general. It specifies methods that its subclasses must override. In **controls.py**, the `Controls` class inherits from Python's `ABC` (abstract base class), and uses the `@abstractmethod` attribute to specify that any class inheriting from `Controls` must implement the methods `get_x_dir` and `get_y_dir`. The `KeyboardControls` and `JoystickControls` classes inherit from `Controls` and provide their own implementations of those methods. In the main file **maze_good.py**, we assign the player an instance of either `KeyboardControls` or `JoystickControls` depending on whether any controllers are connected.

The beauty of this system is the player code doesn't need to know anything about the different control methods, or even that a control method exists. All it cares about is having an object that it can call `get_x/y_dir` on. When I switched the game to using Steam's own input system for controllers, I created a new class called `SteamInputControls`. No changes needed to be made to the player code, I just had to give the players `SteamInputControls` objects instead of `JoystickControls` ones.

"The 1999 version of the game consisted almost entirely of a single C++ file"

Writing bad code is inevitable when you're starting out – even experienced programmers sometimes do it. You don't have to write everything perfectly the first time, but if you learn from my mistakes and get into good habits early, you'll make your life a lot easier.

Partition Sector is out now on Steam.

A list of C++ code files in the new version of *Partition Sector*.