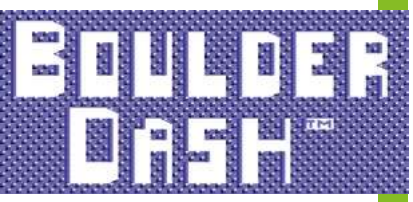
Source Code DX: make a Boulder Dash level editor in Python

In a Source Code special, Mark shows you how to create an entire Boulder Dash construction kit from scratch

AUTHOR
MARK VANSTONE

Mark Vanstone is the technical director of TechnoVisual, author of the nineties educational game series, *ArcVenture*, and after all this time, still can't resist game coding. **education.technovisual.co.uk**

Download the code from GitHub: **wfmag.cc/wfmag56**



Boulder Dash was a popular video computer game in the mid-eighties, and in Wireframe issue 30 (**wfmag.cc/30**), we showed you how to code your own miniature remake. This time, we'll expand on that program to make a level editor, which you can then use to design your own puzzles for other players to navigate.

Before you get started, be sure to have a look through that previous Source Code article to familiarise yourself with how the program works – you can get the code and assets for it from that issue's GitHub: **wfmag.cc/wfmag30**. As a quick reminder of how the game works, it saw intrepid hero Rockford dig his way through underground caves to find gems, all the while avoiding the falling rocks. Those rocks not only fall downwards if there's nothing to hold them up, but they'll also roll down onto other rocks if there's nothing to the left or right of them. Rockford's controlled with the cursor keys, and the aim is to collect all the gems to complete the level.

› The original *Boulder Dash* had its own level editor called the Construction Kit.

We'll continue writing our code in Pygame Zero based on the original program, but to incorporate an editor section, we change our window size to add an extra 200 pixels to the width. As before, our game screen is defined by a two-dimensional list which, in our original program, we filled with random items. For this version, we'll set the play area to a 'default' layout of all soil blocks with wall blocks around the outside to stop Rockford from going off the screen. Each location in the list matrix has a name: either `wall` for the outside boundary, `soil` for the diggable stuff, `rock` for a round, moveable boulder, `gem` for a collectable jewel, and finally `rockford` to denote our hero. Rockford is also defined as an Actor, as this makes things like switching images and tracking other properties easier.

The first thing to do to our program is to add a switch to turn the editor on or off. To do this, we'll define a variable called `editorState`. If this variable's set to True, we open the program with an extra 200 pixels on the right-hand side of the play area using `WIDTH = 1000`. When this area is

When we first load the program, we fill the game area with just soil and a wall border.

shown, we'll need to draw all the elements we need to see in the editor. First, to keep things tidy, we make a function called `drawEditor()` which will be called from the `draw()` function. We probably want to display a title for the area – something like 'EDITOR' will do the job – and then underneath, print 'ON' if the `editorState` variable is True. We can toggle the `editorState` variable with a key press such as the **SPACE** bar. We do this by defining the `on_key_down(key)` function and then capturing the key press value. If the **SPACE** bar is pressed, we set the `editorState` to the opposite of what it is. The code below shows you how we do this.

```
def on_key_down(key):
    global editorState
    if key == keys.SPACE and WIDTH > 800:
        editorState = not editorState
```

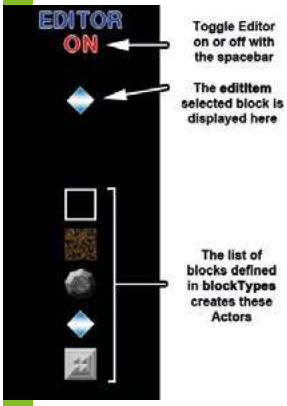
To write an editor for the game, we need to be able to change the blocks that are in the play area. So here's the plan: we'll create a visual list of the available blocks which are clickable. When the block's clicked on, it becomes the currently selected item. Then if we click in the play area, it will change the block to be the item we've selected. All we need to do to change what's in the play area is to change the item name in the items list and the display will update. We'll be using a small set of blocks for this example, but you can add new ones yourself. If you want to create new images for any new blocks, you'll need an image editing program. If you don't have Photoshop, you can create new graphics with programs like GIMP or even Microsoft Paint. There are also free online alternatives such as Photopea, which can export graphics as a PNG file – this is what you'll need for any new blocks. The block images are 40 pixels by 40 pixels in size and should be saved in the images directory alongside your program file.

"We need to be able to change the blocks that are in the play area"

BLOCK FALL

To create our visual list of blocks, we can create a list of Actors which will mean we can test them for mouse clicks. We'll call the list `blockTypes` and define them at the start of the program. Notice how we can define a screen position for them so we can have them neatly arranged in a line. If we had more blocks then we could arrange them in a grid format. You can add your own ➔

The editor panel. This could start to look a lot busier once you start adding your own block types.



Our earlier Source Code *Boulder Dash* game from issue 30 had randomly generated blocks.

blocks here if you want to expand the game, and you can define new blocks in the same way with their name being the file name of the image without .png at the end. To display our list on the screen, we'll need to add a **draw()** command for each item on the list. See the code below to learn how the list is defined and how our **drawEditor()** function is shaping up.

```
blockTypes = [
    Actor('blank', center=(900, 250)),
    Actor('soil', center=(900, 300)),
    Actor('rock', center=(900, 350)),
    Actor('gem', center=(900, 400)),
    Actor('wall', center=(900, 450))
    ]

def drawEditor():
    screen.draw.text("EDITOR", center = (900, 20),
owidth=0.5, ocolor=(255,255,255), color=(0,0,255)
, fontsize=40)
    if editorState: screen.draw.text("ON", center
= (900, 50), owidth=0.5, ocolor=(255,255,255),
color=(255,0,0) , fontsize=40)
    for b in range(0, len(blockTypes)):

        blockTypes[b].draw()
```

We should now see the editor panel (with the 'ON' indicator) and a column of blocks. Next, we need to make them clickable. To do this, we'll need to define an **on_mouse_down(pos)** function. In this function, we'll check we're in the right **editorState** (True) and then check each of the **blockTypes** on the list with the **collidepoint(pos)** function to see if the mouse down event was over the block in the editor section. If it was, then we can set a variable to represent the currently selected item called **editItem**. This variable will be defined at the top of the program and set as the name of the block that was clicked. As things stand, we won't have any visual indicator of which block is currently selected, so we can remedy this by drawing a copy of the **editItem** block in the editor above the list with **screen.blit(editItem,(880,100))**.

We should now have an editor with a list of blocks which can be clicked to set the currently selected item, which is then displayed above the list. Once we've selected a block, we then want to be able to place it in the game area so it changes the map. To do this, we need to check the mouse click position to see if it's over the play area. Then we need to work out which square on the map has been clicked and change that item in the data to be our **editItem** value. Each of the blocks on the map are 40 pixels by 40 pixels, so we can find the position we need in the items list by dividing the mouse position by 40. However, the game area's displayed starting at 40 pixels down the

EARTH SHAKING

If you owned a ZX Spectrum in the early 1990s, you may have encountered one of the best *Boulder Dash* clones ever made: *Earth Shaker*, programmed by Michael Batty and given away as a cassette on the cover of Your Sinclair magazine. With large scrolling levels and a complex array of additional block types – such as a Gravity Stick, which made rocks and other items float for a period – it was a slick and polished iteration of a familiar classic. A few months later, Your Sinclair published a level editor, which, like the one introduced here, allowed readers to design, save, and play their own maps in the original game. You had to type in reams of BASIC, though, but thanks to the wonders of the internet, you can now download it from World of Spectrum: **wfmag.cc/earthshaker**.

screen (to give room for information prompts), so we subtract 40 from the mouse y position before we do the division. The code below shows you how this calculation and testing for clicks on the blocks in the editor is written in the `on_mouse_down(pos)` function.

```
def on_mouse_down(pos):
    global editItem
    if editorState:
        c = int(pos[0]/40)
        r = int((pos[1]-40)/40)
        if r > 0 and r < 14 and c > 0 and c < 20:
            if editItem != "blank":
                items[r][c] = editItem
            else : items[r][c] = ""
        else:
            for b in range(0, len(blockTypes)):
                if blockTypes[b].collidepoint(pos):
                    editItem = blockTypes[b].image
```

GRID LOCKED

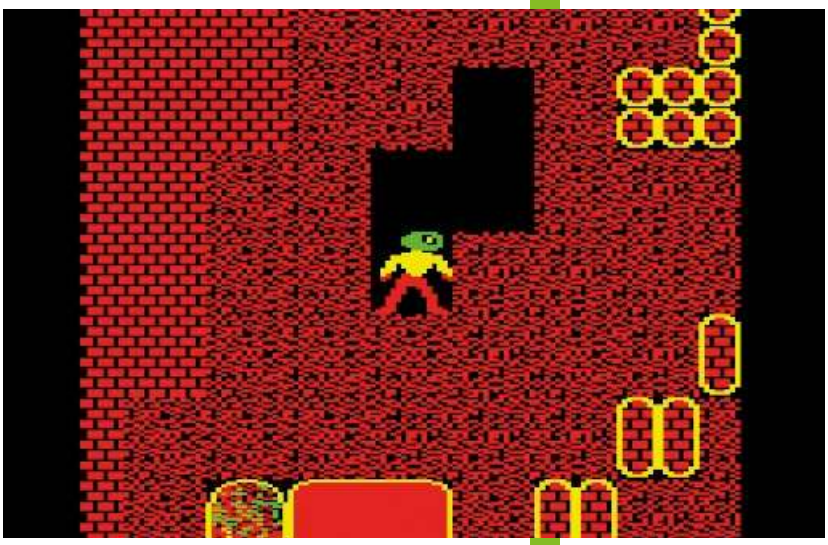
If you've added some extra blocks to the list, you should be able to select and place them on the map at this point. If you want them to behave differently than other blocks in the game, though, you'll need to add some code. The code you write will depend what you want the blocks to do. For example, if you wanted to add a fire block which will sizzle Rockford if he walks over it, you'd need to put some code to test the block directly under Rockford to see if it's a fire block; if it is, set `gameState` to 1. You'd need to put that code in the `moveRockford(x,y)` function. If you want to add extra blocks, have a look at the full listing at the end of this tutorial to see how different types of blocks are dealt with in that function.

So now we should be able to go from a default play area with just soil, border wall, and Rockford to generating a set of boulders and walls with gems for the player to collect without getting squished.

There are many ways to arrange the boulders and walls to make it difficult for the player to get the gems without clearing the soil or moving boulders in the right order. (Of course, you'll need to make sure that it's actually possible to collect the gems.)

Once you've laid out some blocks on the game area, you can test the level by hitting the **SPACE** bar to switch the editor mode off and then start moving Rockford around the play area to see how the boulders react. The only problem with this situation at the moment is that as we move Rockford around, he's changing the map we have made. The rocks start moving, the gems get collected, and the only way to get the original map back is to make it again in the editor. What we need is a way of saving and loading maps. Let's make a couple of buttons to load and save maps, then. We can position these down at the bottom of the editor. They'll be Actors and respond to a mouse click like our blocks, but when clicked, we'll call functions `saveMap()` and `loadMap()`. There are several ways we can save data from our program; if the data was more complicated, we might want to look at the JSON format to save our game maps, or we could use a comma-separated text format. In this case, though, a really effective way of saving this data is to use a library called Pickle. This provides data serialisation functions, which means translating structured data into and from a suitable file format. By opening a file and then calling `pickle.dump`, we can take a lot of the headache out of saving our maps. Conversely, when we want to load our map back in, we just open the file and call `pickle.load` and the data is read back into our items list. Have a look at the code overleaf to see the basics of our save and load functions. ➔

"If you want to expand the game, you can define new blocks"



Repton: a lot like *Boulder Dash*, but set in a posh boarding school. (We may have made that last bit up.)

```
import pickle

def loadMap():
    global items
    with open ('mymap.map', 'rb') as fp:
        items = pickle.load(fp)

def saveMap():
    with open('mymap.map', 'wb') as fp:

        pickle.dump(items, fp)
```

LOAD AND SAVE

You'll see that we're using a fixed file name for saving the map. If you wanted to have the user change the file name, you might want to have a look at the filedialog part of the tkinter library to provide a load or save dialog window to enable choosing a file name for your map. For the purposes of this article though, we will stick with a fixed file name for our map.

Currently when our save function is called, there's no feedback to the user that anything has happened, which may be a bit disconcerting for some. It's probably wise to add some messages into our routines, then: we'll want to have a confirmation that the file has been saved or if there was a problem saving it. We can make a simple messaging system by having a global variable **editorMessage** and a countdown variable **editorMessageCount** to display the message for a period of time and then stop displaying it. If we set the **editorMessage** variable to something like 'MAP SAVED' and the **editorMessageCount** variable to 200, then we can check to see if this variable is greater than zero in the **drawEditor()** function, then, if it is, display the message on the screen using **screen.draw.text()**. After displaying the text, we decrement the **editorMessageCount** variable by 1. This will mean that after 200 cycles of the **draw()** function, our message will disappear.

What if the saving operation failed? There could be all kinds of reasons why this might happen, and it's always a good policy to check when files are loaded or saved that the data transfer actually happened. To check that our file save didn't encounter an error, we can use a **try:** and then an **except IOError:** structure. Underneath the **try:** command we open our file, use Pickle to dump the data to the file, and then set our **editorMessage** to confirm the file's saved. Then we use the **except IOError:** command, and under that, we set our **editorMessage** to display an error message. This means that if an error occurs while saving, we'll see an error message; otherwise, we'll see the confirmation and know that our map file has been saved. Look at the code below to see the updated **saveMap()** function with error checking.

Earth Shaker was one of the best *Boulder Dash* clones on the ZX Spectrum.


```
def saveMap():
    global editorMessage, editorMessageCount
    try:
        with open('mymap.map', 'wb') as fp:
            pickle.dump(items, fp)
        editorMessage = "MAP SAVED"
        editorMessageCount = 200
    except IOError:
        editorMessage = "ERROR SAVING MAP"
        editorMessageCount = 200
```

Our save function is now complete, so we'll turn our attention to how we load the map back in. We have the basics of the loading routine using the **pickle.load()** function, but what happens if we haven't created a map yet? The loading routine would fail and we wouldn't have any map data to work from. We can use the same technique we used with the **saveMap()** function to catch an error if it can't load the file. By using **try:** and **except IOError:** again, we can display a message to say the map has loaded if no error

The finished editor. See how devious you can make your own puzzles!

occurs, and if the map isn't loaded, fill our items list with the default map layout (just soil and boundary walls) and display a message to say that the default map has been loaded. Having put all this in place, we can then add a call to `loadMap()` when the program first runs. If we have an existing map file, the program will load it, and if not, it will load the default map. This means we don't need to generate the default items list at the beginning of the program as the `loadMap()` function will do it for us.

"Now we have an editor that will load the last map we saved"

FILE HANDLING

Now we have an editor that will automatically load the last map we saved or make a default map, allowing us to edit all the blocks in the game area, save the map, and then test it with Rockford. If we test our puzzle layout and find that Rockford gets squished, then at the moment all we can do is close the program and restart it to get back to the saved map. That's going to get very tedious if we're testing over and over. What we need is a reset key. We can check for the escape key in the `on_key_down()` function, and when that's detected, we need to set our `gems`, `collected`, and `gameState` variables all to zero, redefine the Rockford Actor to be back in the top corner, and then call `loadMap()`. This will set everything back to the way it first loads in.

Our editor's nearly finished now, with just one more thing to do. When we've made and tested our fiendish map, we'll want to challenge our friends, family, or random passers-by to solve it. In other words, we want to let them play the game without the editor section. All we need to do is change the `editorState` at the top of the program to False (this will mean the editor section will not be shown) and add a new variable, `editorEnabled` (also set to False), which we will check before letting the **SPACE** bar switch modes. The game is then playable by a non-editing user.

You now have a fully functional *Boulder Dash* editor! Have a look at the full listing to see how everything fits together. You could, of course, expand this to add more block items for Rockford to deal with, or enable multiple levels by loading in different maps as the player completes each one. You could add more tools for the editor, such as file load and save dialogs so you can choose the file names you use for your maps, but we'll leave you to have fun adding those extra features.

Boulder Builder

Here's Mark's code for a full-featured *Boulder Dash* construction kit. To get it running on your system, you'll need to install Pygame Zero. Full instructions are available at **wfmag.cc/pgzero**.

```
# Boulder Dash Editor
import pgzrun
import pickle

editorState = True
editorEnabled = True

if editorState:
    WIDTH = 1000

gameState = count = 0
editItem = "blank"
editorMessage = ""
editorMessageCount = 0

blockTypes = [
    Actor('blank', center=(900, 250)),
    Actor('soil', center=(900, 300)),
    Actor('rock', center=(900, 350)),
    Actor('gem', center=(900, 400)),
    Actor('wall', center=(900, 450))
    ]

loadButton = Actor('load', center=(850, 580))
saveButton = Actor('save', center=(950, 580))
items = [[] for _ in range(14)]
gems = collected = 0
rockford = Actor('rockford-1', center=(60, 100))

def draw():
    screen.fill((0,0,0))
    if gems == 0 and collected > 0: infoText("YOU COLLECTED ALL
THE GEMS!")
    else: infoText("GEMS : "+ str(collected))
    for r in range(0, 14):
        for c in range(0, 20):
            if items[r][c] != "" and items[r][c] != "rockford":
                screen.blit(items[r][c], ((c*40), 40+(r*40)))
    if gameState == 0 or (gameState == 1 and count%4 == 0):
rockford.draw()
    drawEditor()

def update():
    global count,gems
    mx = my = 0
    if count%10 == 0:
        gems = 0
        for r in range(13, -1, -1):
            for c in range(19, -1, -1):
                if items[r][c] == "gem":
                    gems += 1
                if items[r][c] == "rockford":
                    if keyboard.left: mx = -1
                    if keyboard.right: mx = 1
                    if keyboard.up: my = -1
                    if keyboard.down: my = 1
                if items[r][c] == "rock": testRock(r,c)
        rockford.image = "rockford"+str(mx)
        if gameState == 0 and editorState == False:
moveRockford(mx,my)
    count += 1


def on_mouse_down(pos):
    global editItem
    if editorState:
        c = int(pos[0]/40)
        r = int((pos[1]-40)/40)
        if loadButton.collidepoint(pos): loadMap()
        if saveButton.collidepoint(pos): saveMap()
        if r > 0 and r < 14 and c > 0 and c < 20:
            if editItem != "blank":
                items[r][c] = editItem
            else : items[r][c] = ""
        else:
            for b in range(0, len(blockTypes)):
                if blockTypes[b].collidepoint(pos):
                    editItem = blockTypes[b].image

def on_key_down(key):
    global editorState, gameState, rockford, collected, gems
    if key == keys.SPACE and editorEnabled:
        editorState = not editorState
    if key == keys.ESCAPE:
        gems = collected = gameState = 0
        rockford = Actor('rockford-1', center=(60, 100))
        loadMap()

def infoText(t):
    screen.draw.text(t, center = (400, 20), owidth=0.5,
ocolor=(255,255,255), color=(255,0,255) , fontsize=40)

def moveRockford(x,y):
    global collected
```

```
    rx, ry = int((rockford.x-20)/40), int((rockford.y-40)/40)
    if items[ry+y][rx+x] != "rock" and items[ry+y][rx+x] !=
"wall":
        if items[ry+y][rx+x] == "gem": collected +=1
        items[ry][rx], items[ry+y][rx+x] = "", "rockford"
        rockford.pos = (rockford.x + (x*40), rockford.y + (y*40))
    if items[ry+y][rx+x] == "rock" and y == 0:
        if items[ry][rx+(x*2)] == "":
            items[ry][rx], items[ry][rx+(x*2)], items[ry+y][rx+x]
= "", "rock", "rockford"
            rockford.x += x*40

def testRock(r,c):
    if items[r+1][c] == "":
        moveRock(r,c,r+1,c)
    elif items[r+1][c] == "rock" and items[r+1][c-1] == "" and
items[r][c-1] == "":
        moveRock(r,c,r+1,c-1)
    elif items[r+1][c] == "rock" and items[r+1][c+1] == "" and
items[r][c+1] == "":
        moveRock(r,c,r+1,c+1)

def moveRock(r1,c1,r2,c2):
    global gameState
    items[r1][c1], items[r2][c2] = "", items[r1][c1]
    if items[r2+1][c2] == "rockford": gameState = 1

def drawEditor():
    global editorMessageCount
    screen.draw.text("EDITOR", center = (900, 20), owidth=0.5,
ocolor=(255,255,255), color=(0,0,255) , fontsize=40)
    if editorState: screen.draw.text("ON", center = (900, 50),
owidth=0.5, ocolor=(255,255,255), color=(255,0,0) , fontsize=40)
    for b in range(0, len(blockTypes)):
        blockTypes[b].draw()
    if editItem != "":
        screen.blit(editItem,(880,100))
    loadButton.draw()
    saveButton.draw()
    if editorMessageCount > 0:
        screen.draw.text(editorMessage, center = (400, 300),
owidth=0.5, ocolor=(255,255,255), color=(0,0,255) , fontsize=40)
        editorMessageCount -= 1

def loadMap():
    global items, rockford, editorMessage, editorMessageCount
    try:
        with open ('mymap.map', 'rb') as fp:
            items = pickle.load(fp)
        editorMessage = "MAP LOADED"
        editorMessageCount = 200
    except IOError:
        editorMessage = "DEFAULT MAP LOADED"
        editorMessageCount = 200
        for r in range(0, 14):
            for c in range(0, 20):
                itype = "soil"
                if(r == 0 or r == 13 or c == 0 or c == 19): itype
= "wall"
                items[r].append(itype)
        items[1][1] = "rockford"

def saveMap():
    global editorMessage, editorMessageCount
    try:
        with open('mymap.map', 'wb') as fp:
            pickle.dump(items, fp)
        editorMessage = "MAP SAVED"
        editorMessageCount = 200
    except IOError:
        editorMessage = "ERROR SAVING MAP"
        editorMessageCount = 200

loadMap()
pgzrun.go()
```

Enter the code shown here (or download it from our GitHub) and you'll be designing your own cunning stages in no time.