DWARF PROJECT

Managing game objects with Python

Tired of being called small and hairy, **Andrew Smith** grabs his double-headed axe and sets out to create a world full of Dwarfs.

OUR EXPERT

Andrew Smith is a software developer at NHS Digital, has a degree in software engineering and an MSc in computer networks (mobile and distributed).

This issue we're going to add to an already developed game platform, fully created in the Python scripting language using the *PyGame* library module. The theme of the project was inspired by an isometric video game in the early 1990s called *HeroQuest* (**http://bit.ly/lxf277dwarf**). This project first started off as an experiment in video game construction over one weekend when the question was asked: What if the game world moved around the main game character instead of moving the main game character around the game world?

The *Dwarf Game Project* that we'll be modifying as part of this month's tutorial consists of a constructed game map made up of interconnected rooms, game objects and enemy game characters (hostile dwarfs). The player of the game will control the main game character (a green dwarf), around the map disposing of all the enemy players to complete the game or until the main game character is killed by any of the enemy dwarf characters.

To get started, we'll need a few things: Python, *PyGame* and the *Dwarf Game Project*.

To install Python, open a terminal window (Ctr-Alt-T) and type `sudo apt-get python3` followed by `sudo apt-get install pip3`. Then install the *PyGame* module by typing `pip3 install PyGame`.

Finally, grab a copy of the *Dwarf Game Project* from the author's GitHub repository:

```
mkdir lxf278dwarfproj
cd lxf278dwarfproj
git clone https://github.com/asmith1979/dwarfgame
```

The source code and project can also be retrieved from the *Linux Format* archive. Once cloned,you'll notice the following folder structure: **design**, **images**, **sfx** and **src**. The **src** folder contains the Python source code and is also where the source code is executed.

```
cd src
python3 ./dwarfgame.py
```

If all goes well, you should obtain the following output of the *Dwarf Game Project* (*see screenshot, above right*):

Before continuing any further, feel free to get to know the game by moving the game character around the

» PYGAME: THE MAIN LOOP

In every *PyGame* program there needs to be a main loop that renders the graphics used in the program and that also controls input for the game via keyboard and/or mouse depending on the *PyGame* application being developed.

Let's take a look at the main loop implemented in this *Dwarf Project*.

```
while not programEnd:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      programEnd = True
    if event.type == pygame.KEYDOWN
and event.key == pygame.K_ESCAPE:
      programEnd = True
```

```
...
```

As can be seen from the code, the main `while` loop ends on a boolean condition to identify when the program has finished completely. A variable called `programEnd` is used to identify this, which initially is set to a False boolean value.

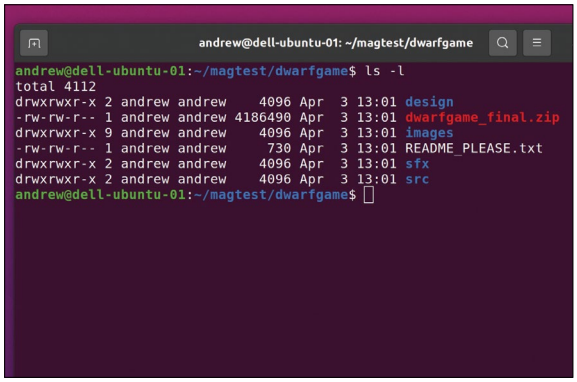
In this particular main `while` loop, you'll notice as you look through the code in the `while` loop that first the input to control the game is dealt with via the keys on the keyboard. The Escape key is used to exit the program, the cursor keys enable directional movement of the player and the Space bar invokes the attack move of the player.

Continuing on from this is the processing that's carried out for both human and computer players, as well as screen rendering operations to process graphics for the game itself and everything in it.

Finally after doing all this we come to the final part of the loop, as shown below:

```
...
  clock.tick(100)
  pygame.display.flip()
```

From the above code we determine our framerate, which is set to 100, and then update the contents of the entire display.



Here's the Dwarf Game Project folder structure that includes design, images, sfx and src.

game world that's been created. The cursor keys are used for movement and the Spacebar key is used for attack. The Escape key ends the game program. When you have become used to the gaming environment, press the Escape key to exit the game and continue with this tutorial.

The **sfx** folder is where all the sound effects are stored. The **images** folder contains all the visuals used in the game such as the artwork used for game characters, floor layouts for the rooms, game objects and so on. The design folder contains documentation created in *MS Visio* but can be viewed in *LibreOffice Draw*, which shows various process flows and most importantly, a structural design showing how the players of the game are created (class diagram).

Before we walk through the code, it's worth pointing out that you can view and edit the source file by typing `gedit ./dwarfgame.py` while in the **src** folder or you can use an IDE called *Notepad++*. This can be downloaded from the Ubuntu software download tool. This tutorial will use *Notepad++* to show various parts of the source code and will be the assumed editor when editing the source code. Any other IDE or editor can also be used such as *VS Code* or *PyCharm*. Note if you're viewing or editing the source code via *gedit* from a Terminal, then it may be wise to open up two terminal windows: one for editing and viewing source code, and the other for executing the game program.

Welcome to Dwarf.py

From viewing the source code, you may notice that the source code is very well commented throughout, to explain what various parts are there for and do. For this reason, an in-depth view of the source code won't be covered here. Instead, we'll give just a brief overview of the most important and relevant parts. A video tutorial can be seen here: **https://youtu.be/7f0Amsrh3vY**.

At the very top of the source code you'll notice as with any other Python script, the libraries for the script to work are declared.

```
import pygame
import os
import threading
import time
from pygame.locals import *
from time import sleep
```

Something that may need to be changed straight away is the screen resolution that the game runs under, because not everyone will be running the script off the same device with similar display capabilities. Scroll down to where you see the following or use the IDE's search facility:

```
HORIZ_RESOLUTION = 1600
VERT_RESOLUTION = 1200
```

Please adjust the screen resolution to what you feel works best for you, unless it's acceptable running under the current settings.

To increase the difficulty of the game and vary the ease at which the enemy players can kill the main game character, you can make an adjustment in a function called `enemyAttackProcessing()` which you can get to quickly by using the Find function of your IDE. In this function you'll notice that there's a locally declared variable called **enemyAttackRatio**, which is currently set to the value of 4. What this means is that for every four attacks of an enemy player, the human player's energy reduces by calling a function named `reduceHumanDwarfEnergy()` that you may also wish to alter, even though you don't have to if you don't want to. Again this function can also be quickly accessed by using the Find functionality of the IDE or editor that you're using.

QUICK TIP

If you want to jump to a specific function in the code, instead of scrolling through lines and lines of code, just use the IDE's search function (if it has one). It's usually invoked by pressing (Ctrl+F). Type in your search term and then click the Search Next button.

Adding existing objects

Before continuing any further, it's expected that you would have already launched the game and become familiar with the controls. You should know how to move around the game and interact with the enemy game characters and game objects that are scattered throughout the game environment.

We're going to add a `BOX_CRATE` object to the room that we're starting in, which you may have noticed is called `PLAYROOM`. An image for our box-crate has already been loaded into memory at the start of the

Here's a successful execution of the Dwarf Game Project from a Terminal window.

program, so scroll down the source code until you reach a section commented on as GAME OBJECTS. You should get to the following:

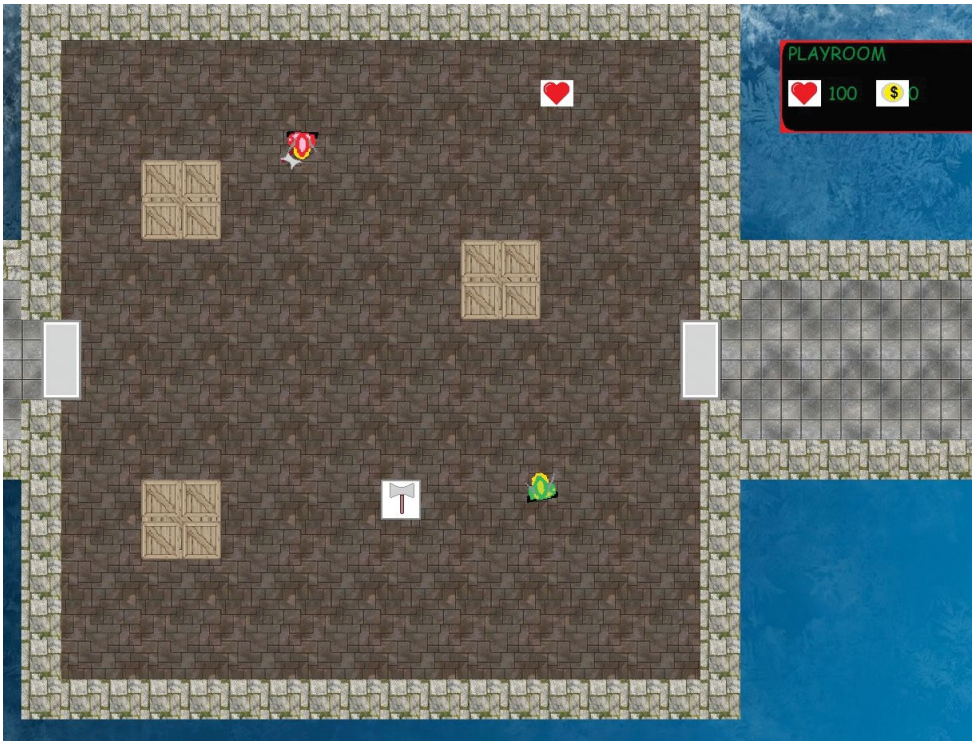
```
gameObjectCollection = [ ] # Stores game objects

# Room Description, Object Type, Active / Non-active, X-position, Y-position
gameObjectCollection.append(['PLAYROOM', 'BOX_CRATE', True, 300, 600])
gameObjectCollection.append(['PLAYROOM', 'BOX_CRATE', True, 300, 200])
gameObjectCollection.append(['PLAYROOM', 'HEART_ICON', True, humanDwarf.x, humanDwarf.y-500])
gameObjectCollection.append(['PLAYROOM', 'DWARF_AXE', True, 600, 600])
gameObjectCollection.append(['THRONE_ROOM', 'DWARF_SHIELD', True, 1400, 1000])
gameObjectCollection.append(['THRONE_HALLWAY', 'HEART_ICON', True, door_x2_throne-50, door_y2_throne+250])
```

Scroll to just after where the last game object has been declared and type

```
gameObjectCollection.append(['PLAYROOM', 'BOX_CRATE' , True, 700, 300])
```

Notice that when you add a game object to the game, there's a certain structure to adhere to: Location, Object Type, Active Status, X-Position, Y-Position. After typing the above and running the program again, the room will contain a third crate (*see screenshot, below*).

Now that you've added a box-crate object, feel free to introduce other objects to the game world. For example, you could add heart objects to various locations of the game to boost the player's energy. Remember that when adding objects to the game, ensure the object you're placing is in the correct location. For example, earlier we specified the location of the box-crate to be PLAYROOM, and the x- and y-location of that box-crate is within the playroom. To get to know the room locations of the game world, it's expected that you would take the time to explore the game for yourself.



Here's is the successful execution of the project after a third box-crate object has been added.

Adding new objects

This section of the tutorial is targeted at more experienced Python coders. Of course, there's nothing stopping anyone from putting in the time and effort to gain a solid understanding of how things have progressed and work in the game.

To add a new game object that doesn't exist in the game is a little bit more involved. First of all, you need to decide what the new game object is going to do and how you want it to behave in the game environment. For example, is it going to be a game object to increase a player's energy, or perhaps it's going to generate an obstacle for the player to navigate around such as a box-crate.

If you remember earlier, we mentioned a project structure where all images for game objects are stored in the location of either **images/GameObjects** or **images/misc**. In these folders, you'll find images that are either 50x50 pixels or 100x100 pixels in size. This is the recommended size to create game objects, given the context and scope of the game. The images can be created in any image manipulation or creation tool you like, such as *GIMP*. This can be regarded as the very first stage in adding a new game object. A guide on how to use *GIMP* is beyond the scope of this tutorial, though.

Once the image has been created and is in one of the folders mentioned above, now is the time to load the image into memory so that it can be used. Search for a section in the code that's commented 'Load Images'.

Notice that as a safety measure, a `try/except` is used when attempting to load in the image just in case the image isn't found for whatever reason. Scroll down to where the game objects has been loaded and follow the code to load in the image for the game object that you've created. Before continuing, check that this now runs without causing an error.

Scroll again to the Game Objects section of the code as you did when adding an existing game object and type the code in to add your object to the existing **gameObjectCollections**. Remember that there's a certain structural declaration to adhere to, as explained previously.

There are four functions you'll need to pay attention to next: `moveFloorUp()`, `moveFloorDown()`, `moveFloorRight()` and `moveFloorLeft()`. These functions are used to move the game environment including game objects and enemy players in respect of the human player. For example, look at the `moveFloorUp()` function.

As you look further into this function you'll reach a section that deals with game objects. These have a while loop that's used to scroll through each of the game objects in the **gameObjectCollection**. To add

Human Dwarf armed now with axe and shield

your own object, just add an additional if statement and repeat in the previously mentioned three functions. Note that subscript [4] refers to the y-coordinate and that the subscript [3] refers to the x-coordinate, which corresponds with the defined data structure earlier explained.

Collision detection

Next, there are another four functions that will need code adding to, as follows: `detectbcl_leftside_col()`, `detectbcl_rightside_col()`, `detectbcl_bottomside_col()` and `detectbcl_topside_col()`. All these functions are used to detect object collision with the player (the green dwarf) of the game. For example, `detectbcl_rightside_col()` is used to see if the human player has collided with the right side of the object. Again, these functions can reached straight away by searching on their name.

As an example, we'll take a brief look at one of the functions, `detectbcl_rightside_col()`.

```
def detectbcl_rightside_col():
  global humanDwarf

  # A separate collection for each object in the game
  boxCrateCollection = [ ]
  axeCollection = [ ]
  heartCollection = [ ]
  shieldCollection = [ ]

  # Separate objects out into their separate collections
  for gameobject in gameObjectCollection:
    if gameobject[1] == 'BOX_CRATE':
      boxCrateCollection.append(gameobject)
    if gameobject[1] == 'DWARF_AXE':
      axeCollection.append(gameobject)
    if gameobject[1] == 'DWARF_SHIELD':
      shieldCollection.append(gameobject)
    if gameobject[1] == 'HEART_ICON':
      heartCollection.append(gameobject)

# Reset counter
  cCounter = 0
  # Check to see if there has been a collision with any of the box crates
  while cCounter < len(boxCrateCollection):
    if boxCrateCollection[cCounter][0] == humanDwarf.location:
      boxx = boxCrateCollection[cCounter][3]
      boxy = boxCrateCollection[cCounter][4]

      if boxx > (humanDwarf.x - 122) and boxx < (humanDwarf.x+10) and (humanDwarf.y+18) >= boxy and humanDwarf.y <= (boxy+106):
        return True

    cCounter = cCounter + 1
  # Reset Counter
  cCounter = 0

  ...
```

Notice that the game objects are broken down into further collections (using the for loop) and then after this is done, each of the collections is checked to see if there has been a collision with the human player's character. Note that you may well need to add to the existing code to include your new game object for example, by adding an additional collection and check as above.

Finally you'll need to make changes to a function called `renderGameObjects()` which takes a screen handler as an argument. This function as the name suggests, and is used to render all game objects in the **gameObjectsCollection**. There's a while loop implemented that's used to scroll through all the game objects in the collection. Adding your own game object to this should be no more complex than adding an indented if statement to the while loop. LXF

» OBJECT ORIENTATED PROGRAMMING

Object orientated programming (OOP) is a paradigm based on the concept of classes, objects and associations that can contain both data and code, which determines how the objects behaves in the software system being created. Before an object is created, a template is first created to then instantiate an object. A template of an object is called a Class which, by using a concept called inheritance, can inherit other data and operations so that they don't have to be re-written each time.

Object Orientated Programming has been implemented in the *Dwarf Game Project* to implement the game characters for the game. There is a base class called Player that includes much of the properties and operations which are applicable for both the human player and computer-based players. Notice that in the base class there's a method called `processWalk_Anim()`, which is used to process the walk animation of dwarf, whether it be the player's dwarf or an enemy dwarf. After the base class Player, there are two derived classes called DwarfPlayer and DwarfAIPlayer, which both inherit the functionality and properties of the Player class. Both of the derived classes DwarfPlayer (used for the human player) and DwarfAIPlayer used for the computer player continue to define properties and operations unique to their application in the *Dwarf Game Project*.