PYTHON

Using Python sockets for multiplayer gaming

Discover how to implement multi-player gaming in a Galaxian-style shooter with **Andrew Smith**, in the second part of coding Star-Fighter!

OUR EXPERT

Andrew Smith is a software developer for NHS Digital, has a bachelors degree in software engineering and a master's degree in computer networks.

This month we're going to look at the multiplayer version of *Star-Fighter* that was covered in **LXF282** and played as a single player game (created by Francis Michael Tayag). We'll cover the network programming techniques that make it possible to play this game across a LAN (local area network) and where in the source code the changes have been made.

Both instances of the game (server and client) will be executed on the same machine, even though it's possible to run them on separate machines on the same LAN. Network programming in Python is an advanced topic so if any readers are new to Python, it may be advisable to focus on just the setup and execution of the project. It's also worth pointing out that this tutorial won't cover how parts of the game work or how the game is structured, because this was discussed in **LXF282**. Instead, this tutorial focuses on the networking element of the game (which uses the sockets library) and where changes have been made in the game to make multiplayer action possible.

We're aiming to run two instances of the game on the same device. It's possible that some readers may not have two computers readily available, so for this tutorial just one device is used. Let's begin by setting up our Python development environment.

Installation and setup

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`. To make sure that you're using PyGame version 2.0 (*Star-Fighter* does use a recent version of PyGame), type `python3 -m pip install pygame==2.0.0`. You should now have version 2.0 of PyGame installed. Type the following to check what you've just installed:

```
python3
import pygame
quit()
```

If all's well then your screen should look the same as the screengrab (*facing page, top right*).

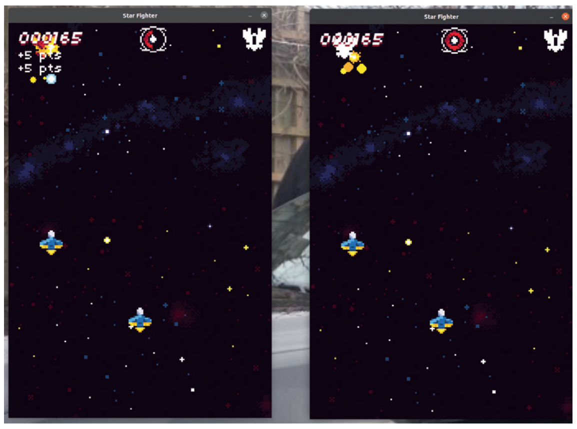
Finally, grab a copy of the *Star-Fighter* project source code by cloning the GitHub repository. Before typing the following to clone (copy) the GitHub repository, move into a folder on your system that you'd like the project to be copied to.

```
git clone https://github.com/asmith1979/starfighter_
multiplayer/
```

As an example, the whole project has been put into a folder called **PythonProjects**, which was created before downloading the project.

To edit and view the source code you can either use a basic text editor or something more specific such as *Notepad++*, *PyCharm* or *VS Code*. In this tutorial, we'll be using *gedit* to view and edit the source files. If and when using this method to view or edit source files, it may be helpful to open up three console windows: one for editing/viewing source files and the other two terminal windows for executing the PyGame code.

QUICK TIP
You may find it useful to use three console windows for this tutorial: one for editing or viewing source code; one for running the server instance; and one for running the client instance.



An example of multiplayer gameplay. Shown are both the server and client instance of the program running on the same device.

Star-Fighter in multiplayer mode

For the multiplayer feature of the game to work on the same device, you'll need to modify the script file **game.py**. Open up **game.py** in your chosen editor and find the property called **multiplayerDemo** – there should be two instances of this. The value should be currently set to False. Change both instances of **multiplayerDemo** to True, and then save.

On both of the open terminals, run *Star-Fighter* by typing the following into each terminal window:

```
python3 game.py
```

The game's music may sound odd with both instances of the game playing it at the same time on the same device. So on both instances of the game, open the Options menu and select the Sound menu. Once there, set the music volume at 0 and then go back into the Options menu. No more distracting music!

Out of the two instances running the game program, one will need to run as the game's server and the other instance will need to run as the client player instance. In each of the instances, go into the Multiplayer menu settings via the Options menu.

When you first access the Multiplayer menu, the default mode that's selected is standalone, which means the game is currently played in single player mode. Pressing the right cursor key selects server mode so the game instance will act as a server. Pressing the right cursor key again puts the game into client mode.

Choose one of the instances to be in server mode and the other game instance to be in client mode. Because both instances are now running with `multiplayerDemo=True`, this acts as an override to using the real IP addresses and both instances use the local host address, 127.0.0.1 instead of the actual IP address of the device and both use different port numbers. The server instance uses port number 20001 and the client instance uses port number 20002.

Once the server mode on the server instance of the game has been selected, choose Accept and Close, and then go back to the main menu and start the game.

In the game instance of the client, accept and go back to the main menu and start the game. This should display a message on the screen that the client has been connected to the server, and the game will begin.

You can play both server and client individually by clicking the executing game instance. Observe the effect of this on the other game instance and vice versa.

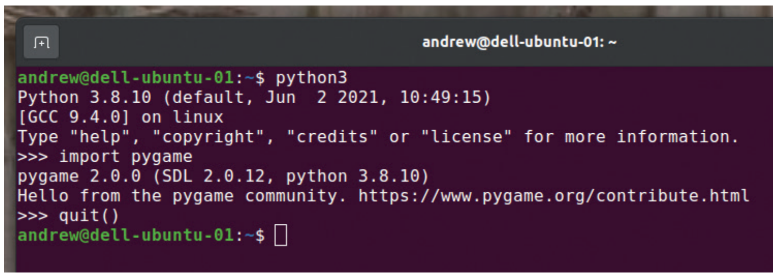
Converting to multiplayer

Before any network code could be implemented or used in the *Star-Fighter* project, various parts of the program had to be altered for the game to accept there was the possibility of being more than one player and if there was more than one player, to handle the extra player equally. The game code has been adapted to handle up to two players at a time.

In **scenes.py** in GameScene class constructor type

```
self.playerCollection = []

# Initialize the player
self.player = Player(PLAYER_IMGS, BULLET_IMG, self.P_Prefs)

self.playerCollection.append(self.player)

# Add second player to screen if multiplayer version is selected
if self.P_Prefs.multiplayer == SERVER_MODE or self.P_Prefs.multiplayer == CLIENT_MODE:
    self.playerTwo = Player(PLAYER_IMGS, BULLET_IMG, self.P_Prefs)
    self.playerTwo.isMultiplayer = True
    all_sprites_g.add(self.playerTwo)
    self.playerCollection.append(self.playerTwo)

all_sprites_g.add(self.player)

# Create a spawner
self.spawner = Spawner(self.playerCollection, self.g_diff, self.P_Prefs.multiplayer)
```

A second player is only created when the game program is either in SERVER_MODE or CLIENT_MODE (a mode of network play) which is selected by the player in the multiplayer menu. If the mode is left as default, STANDALONE, the game continues as a single player game. Note that a property has been added to the



```
andrew@dell-ubuntu-01: ~
andrew@dell-ubuntu-01:~$ python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pygame
pygame 2.0.0 (SDL 2.0.12, python 3.8.10)
Hello from the pygame community. https://www.pygame.org/contribute.html
>>> quit()
andrew@dell-ubuntu-01:~$
```

Here's the PyGame version installed. You should see that version 2.0 of PyGame has been installed.

» IP ADDRESSES AND LOCALHOST

An IP (internet protocol) address identifies a device on the internet or on a local area network (LAN) if an internet connection exists. An IP address can be identified by four numbers separated by three full-stops. For example a valid IP address could be 172.168.1.212. A local host address is always referred to as 127.0.0.1 and is also known as the loopback address.

In Multiplayer Demonstration mode, which this tutorial is based on, only local host addresses are used in both instances so both use 127.0.0.1 with different port numbers so that the communications don't clash on the same device. When the Multiplayer Demonstration mode isn't activated, the IP address of the devices is used and each instance has to run on a separate device for the multiplayer feature of the game to work. You'll notice that there are some multiplayer settings defined in **games.py** that refer to IP addresses.

```
self.clientAddressPort = (['127.0.0.1', 20002])
self.serverAddressPort = (['127.0.0.1', 20001])
self.clientIPAddress = self.clientAddressPort[0]
self.serverIPAddress = self.serverAddressPort[0]
self.clientPort = 20002
self.serverPort = 20001
self.multiplayerDemo = False
```

By default, both client and server IP addresses are set up as the localhost address (127.0.0.1) because at this stage in the program there's nothing to identify otherwise. An address port variable is also set up for both server and client that contains both the IP address and port number being used.

The port numbers in this example are used to receive communications. Note that 20001 is to receive data on the server port and port number 20002 is to receive data on the client port.

»

LXF283.code1_gaming_p18b.indd 89 22/10/2021 15:40

Player class to identify if the player is multi-player identified by `isMultiplayer`, which is a Boolean variable.

Furthermore, functions throughout the game program that expect an instance of a player to be passed in are passed in as a collection of players. This avoids having to add another argument to a function to take an extra player. You'll see in the previous source code example that `self.playerCollection` is used to hold a collection of players in the game.

When playing in multi-player mode, the server is the side that has dominant control, because the scores and both players' health are fed from the server to the client. The position of the client spaceship is the only property passed from the client to the server, which the server then inserts into its current "game model" instance.

At the end of the GameScene constructor, the main communications points are set up so that gameplay data can be transferred between server and client instances of the game program:

```
# Create both server and client stub
self.serverStub =
MultiplayerDataTransferServer(self.P_Prefs)
self.clientStub = MultiplayerDataTransferClient(self.P_
Prefs)

# If acting as server, setup server listening thread
if self.P_Prefs.multiplayer == SERVER_MODE:
    self.threadedServerProcess = threading.
Thread(target=self.serverStub.setupServerListening)
    self.threadedServerProcess.start()

# If acting as client, setup client listening thread
if self.P_Prefs.multiplayer == CLIENT_MODE:
    self.threadedClientProcess = threading.
Thread(target=self.clientStub.clientEndPoint)
    self.threadedClientProcess.start()
```

Even though both server and client points are set up in the constructor, only one is used depending on the game instance being played by the player. If the game instance running is acting as the server, the server stub will be used. If the game instance is running as the client, the client stub will be used. On each instance of the game, there's a thread that's used to listen for incoming communication and to also send that data out again. Threads enable the program to flow naturally while another process can execute alongside it.

Here's the Server player waiting for Client player to connect. Remember to always start the server before the client.

Server-side processing

`MultiplayerDataTransferServer` is the class used for all server-side processing. Once a player of the game program acts as the server in the multiplayer menu, this is the class that manages the communication between the server instance of the program and the client instance. The server class contains the following functions (aside from the constructor):

```
• getServerIPAddress()
• clientInList()
• clientLimitReached()
• getConnectionRequest()
• sendDataToClient()
• setupServerListening()
• selfTermination()
```

The main functions of interest out of this class are `setupServerListening()`, `sendDataToClient()` and `selfTermination()`. The other functions in the class can be regarded as helper functions.

The function, `setupServerListening()`, is the main function of interest in the class because it first sets up a socket and continually listens for incoming communications from the client instance of the program. This is also the function that's threaded on the server side. The function `sendDataToClient()` is used to send game data to client, which includes data for both players as well as game data such as the score. The function `selfTermination` is used to end the server process at the end of the multiplayer game session.

Client side processing

`MultiplayerDataTransferClient` is the class used for all client-side processing and is located in **scenes.py**. The main functions of interest in this class include:

- `clientEndPoint()`
- `sendDataToServer()`
- `selfTermination()`

The other functions present in this class are used as helper-functions. In this class, `clientEndPoint()` is the function that's threaded and is used to receive communication from the server side of the game.

Transferring game data

The transfer of player/game data to and from the client and server is done in the update function of the GameScene class located in **scenes.py**. Data for both the client and server side instances are dealt with here.

There is a class that has been created and used called PlayerData located in **sprites.py** that holds mainly player data, but also some game data. It's a collection of this data that's passed between client and server. The example below shows that the data from the server is processed by the client once it's received.

```
if self.P_Prefs.multiplayer == CLIENT_MODE: # Client
mode
```

```
    # Data Transfer from Server (To Client)
    if len(self.clientStub.objectCollection) > 1:
      self.clientStub.dataTransferInProgress = True
      # Player Two Data (Remote player)
       playerData = self.clientStub.objectCollection[1]

       self.playerTwo.setData(playerData)

       # Get general game data
       self.score = playerData.gameScore
       self.is_gg = playerData.gameEnd

       # Player One Data (Client Player)
       clientPlayerData = self.clientStub.
objectCollection[2]
       self.player.health = clientPlayerData.health
       self.player.isDead = clientPlayerData.isDead
       self.player.gun_level = clientPlayerData.gun_
level
       self.player.prev_gunlv = clientPlayerData.prev_
gunlv
….
```

From studying this code, you'll see that a variable called `dataTransferInProgress` is set to True near the start. This is to tell `clientEndPoint()` that no more data is to be collected until the data has been fully taken from the data structures and put into the game model. Once this is done, it's set to False and the data from the server is then continued to be retrieved. The data is transmitted in a layered way and then processed in a layered way. The first data structure (data layer 0) is used for general information purpose such as connection requests or requests to terminate a service. The following two structures (data layer 1 and 2) that are passed contain player and game data.

The same is also done for the server-side:

```
if self.P_Prefs.multiplayer == SERVER_MODE: # Server-
Mode

  # Data Transfer to Client (From Server)
  # Clear send object collection to reset
  self.sendObjectCollection.clear()
  sendGameDataObject = MultiplayerMessage('GAME
DATAFROMSERVER', '0.0.0.0', 0)
  self.sendObjectCollection.
append(sendGameDataObject)

  # Add server data
  pData = PlayerData()

  # Add Server Player Data
  pData = self.player.getData()

  # Add Server Game Data
  pData.gameScore = self.score
  pData.gameEnd = self.is_gg
….
```

From studying these two sections of code in **scenes.py**, you'll learn that as well as retrieving data, there's also a side of the process that collects data to send back to the other side. The server will collect data from its own game instance, then send it to the client. The client instance will collect the relevant data (spaceship position) to send back to the server for processing.

The original game has been further modified than what's been shown in this tutorial to incorporate a second player, but it's beyond the scope of this article to cover this aspect. We're only focusing on the network coding elements of the game. Further aspects that could be worked on to improve this project include:

> An improved algorithm to generate enemy game characters (located in **spawner.py**)
> An improved algorithm to generate power-ups (located in **spawner.py**)
> Better client authentication (only basic client IP authentication has been implemented)
> Adapt code further so that it can be played over the internet instead of just via a LAN

Have fun with the code and let us know what improvements you create! And remember – when looking for a function or variable name in a script, you can use the IDE's search facility to go directly to it instead of scrolling through all the source code.

THREADS IN PYTHON

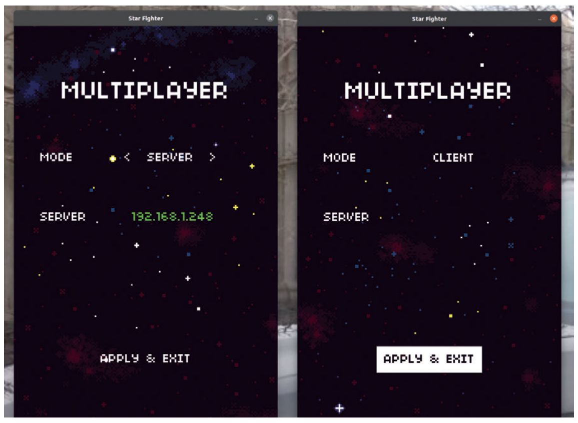
A thread is a separate flow of execution that enables a process to operate alongside the main process of an application or in the case of this project, a video game. To start using the Threading functionality in Python, the threading library needs to imported.

In the Multiplayer *Star-Fighter* project, threads have been set up and used to manage communication between the client and server instances of the game program. At the same time the game can continue as normal.

The first step in using a thread is to set one up. This is done like so:

```
self.threadedClientProcess = threading.Thread(target=self.
clientStub.clientEndPoint)
self.threadedClientProcess.start()
```

As can be seen from the above code segment, the thread is set up to target a function to process as the thread. In this case it's a function called `clientEndPoint()` that will be used as the thread. This function can be found in the `MultiplayerDataTransferClient` class located in **scenes.py**. In addition, threaded functions are usually looped processes so it's important to fully define and be clear on the end condition that will end that process, then end the thread itself. In `clientEndPoint()` there's a Boolean condition that's used to end the looping process



The multiplayer menu. Select to either act as the server or have the client connect to the server.

» GET INTO A KILLING FRENZY AND Subscribe now at http://bit.ly/LinuxFormat