Credit: https://github.com/jtmfam/Gh0stenstein

Create 3D gaming worlds with Python

Understanding the basics of rending a 3D world is a great way to understand game code. Lock and load, says **Andrew Smith**…

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.

During the early to mid-90s, some of the most popular video games to play on a PC were video games such as *Wolfenstein 3D*, *Doom*, *Quake* and *Unreal Tournament* to name a few. In this edition of *Linux Format* we're going to look at the construction and mathematics involved in creating a 3D game world, similar to that of *Wolfenstein 3D* (1992) or *Doom* (1993). In this first part of the tutorial, we'll cover how a 3D game world can be created using *PyGame* and look at some of the mathematics used to navigate around the game world (using functions from the Python mathematics library) as well as some of the collision detection techniques.

This tutorial is based on a project that was created over 10 years ago in 2011 and has recently been adapted to work on the latest Python for the benefit of this tutorial. The original project was written to work on Python 2.7. The original source code and resources that this tutorial is based on can be found at **https://github.com/jtmfam/Gh0stenstein**.

Python 3.10 has recently been released, so we'll install and set up Python 3.10 for the benefit of this tutorial. For those that have Python and *PyGame* already installed, Python 3.8+ should be suitable for use with this tutorial. Type the following to install Python 3.10 and *PyGame*.

Released back in 1992, Wolfenstein 3D was the first example of a first-person shooter using ray casting.

CREDIT: Wolfenstein 3D 1992, id Software.

CREDIT: Doom 1993, id Software.

Doom, created by id Software, was another example of a first-person shooter that used ray casting and featured 2D sprites.

```
$ sudo apt-get install python3.10
$ sudo apt-get install python3-pip
$ python3.10 -m pip install pygame
```

Check both the python and *pygame* versions.

Next, `git clone` from repository.

```
$ git clone https://github.com/asmith1979/
lxf286_3DWorld/
```

As an example, the whole project has been put into a folder called **PythonProjects**, which was created before downloading the project. Alternatively the source code and project can be retrieved from the **LXF286 linuxformat.com/archives**. This tutorial will focus on the source code located in the folder called **Gh0stein\src**. If it's not already in that folder type `cd Ghostein` to get into the folder and gain access the Python source code.

To edit and view the source code you can either use a default text editor installed on your flavour of Linux 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. It may be helpful to open up two console windows: one for editing/viewing source files and the other terminal window for executing the *PyGame* code.

When you've cloned the git repository, you'll find all the files that are needed for this tutorial in the **src** folder. In this folder you'll find the source files **main.py** and **worldmanager.py.** There's also a folder called **pics** that contains additional folders for the images for the game objects and other game content to create the 3D world. **WorldManager.py** is the file mainly responsible for loading and generating the 3D world environment.

To run in the **src** folder type the following:

```
$ python3.10 main.py
```

Use the cursor keys to move and look around the 3D world that's been generated as well as changing weapon by using the numeric keys (1-6) at the top of the keyboard (not the numberpad). When you've explored the 3D environment, press the Escape key to end the program and continue with this tutorial.

Enter the 3D world

Using your editor of choice, open and view **main.py** in the **src** folder. Close to the top of the file, you'll find a 2D array (24x24) called **worldMap**. This data structure is used to define the contents of the 3D world.

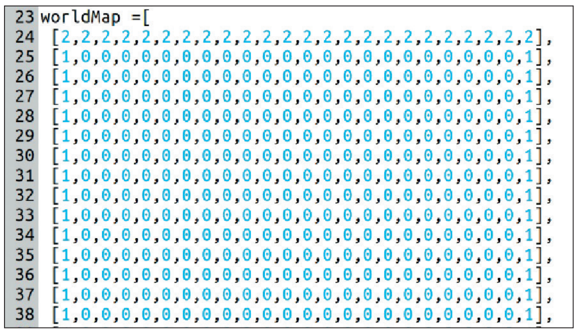
Around the outer edges of the data structure, you'll currently see that a number 1 is defined, which is used to tell the program which wall texture to use. The following wall textures are available to use and try out. Just replace 1 with one of the following numbers, save and then re-run the program.

```
1 – Eagle wall texture
2 – Purple-stone wall texture
3 – Red-brick wall texture
4 – Greystone wall texture
5 – Blue-stone wall texture
6 – Mossy wall texture
7 – Wood wall texture
8 – Colour-stone wall texture
```

All the textures for the walls are 64x64 pixel PNG files located in **src/pics/walls**. All the texture are loaded in by the **WorldManager** class located in **worldManager.py**. Open **worldManager.py** to look at how this is done.

In the constructor of the **WorldManager** class, you'll see that all textures – both for the walls and the game objects – are loaded into the program's memory. All textures for game objects are loaded into a collection called sprites. All textures for the walls are loaded into a collection called images. In addition, all images are loaded by a custom created function called **load_image** which takes in various arguments to define how the image is loaded. A short example of how this is done is shown in the source code example below:

```
...
self.images = [
    load_image(pygame.image.load("pics/walls/eagle.png").convert(), False),
    load_image(pygame.image.load("pics/walls/redbrick.png").convert(), False),
    load_image(pygame.image.load("pics/walls/purplestone.png").convert(), False),
    load_image(pygame.image.load("pics/walls/greystone.png").convert(), False),
...
```



```
worldMap =[
 [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
```

Screenshot of the world map, a 2D array representation of the 3D world that can be moved around in. The 0s represent an empty space.

The first person...

The view of the first person is represented by the Camera class implemented in **WorldManager.py**. In the generated 3D world that's seen when you run the program, the x-position of the player becomes the z-position (forward and backward) and the y-position becomes the new x-position, so in effect you set the position by (z,x) by how the 3D game world is generated by the *draw* function in the **WorldManager** class.

An instance of the Camera class is created in the constructor of **WorldManager**, passing in first the x- and y-positions that translate into the z- and x-positions, as well as the direction the player is facing.

```
class Camera(object):
  def __init__(self,x,y,dirx,diry,planex,planey):
    self.x = float(x)
    self.y = float(y)
    self.dirx = float(dirx)
    self.diry = float(diry)
    self.planex = float(planex)
    self.planey = float(planey)
```

Moving around the 3D world

So far in this tutorial we've covered how to run the game program and how the 3D world is initially created by

» OBJECT ORIENTATED PROGRAMMING

Object orientated programming techniques have been used in this project. The **worldmanager.py** script file contains a collection of classes used to create and construct the view of the 3D world. These classes include the **WorldManager** and **Camera** class. The **WorldManager** class consists of a class constructor and a method called *draw()* which is used to render the 3D world.

Each class in Object Orientated Programming employs a constructor that's used to initialise values and perform initial operations when and instance of that class is created or called. For example, the constructor in the **WorldManager** class is used to load images both for the objects that can be shown when the program is running and also the texture images for the wall. The constructor of the **Camera** class is used to just initialise values for later use in the program. Both of the constructors used in the script file take in arguments to the constructors, so that values can be passed in from the **main.py** script file.

Look at the source code from **main.py**:

```
wm = worldManager.WorldManager(worldMap,sprite_positions, 20, 11.5, -1, 0, 0, .66)
```

From the above code example, it can be seen that passed into the **WorldManager** constructor is the 2D array that represents the 3D world (`worldMap`), sprite positions, player position and direction. As part of the **WorldManager** constructor operation the camera view is set up for the player's first-person view.

»

QUICK TIP
Instead of scrolling down lines of text looking for a variable or function name, use the IDE's search facility to go directly to what you're looking for.

using a collection data structure. Now we'll go further in detail, looking at the mathematics involved in making it possible to move around the 3D world.

When the Up cursor key is pressed to move forward, the following code is executed:

```
...
  moveX = wm.camera.x + wm.camera.dirx +
moveSpeed
  if worldMap[int(moveX)][int(wm.camera.y)] == 0:
    wm.camera.x += wm.camera.dirx * moveSpeed

  moveY = wm.camera.y + wm.camera.dir * moveSpeed
  if worldMap[int(wm.camera.x)][int(moveY)] == 0:
    wm.camera.y += wm.camera.diry * moveSpeed
...
```

The above code first processes movement along the z-axis first (forward and backward) along the world map by calculating the distance to move in the direction the player is facing and then checks to see if it corresponds to a 0 value on the **worldMap**. The 0 value on the **worldMap** represents a free space the player can move to. If the space is free, the player (or camera view) is moved to that space.

Following on from this, the movement for the x-axis is processed, represented as y. Again, the distance in the direction is first calculated and checked to see if the resulting movement is in a free space on the **worldMap**. If it is then the player (camera view) is moved there. The above method can be summarised for each axis as

1 Get the move value
2 Check that the move value corresponds with 0 on the **worldMap**
3 If it is a free space, move player forward to that space

To show the effect on the movement when the above code is altered, comment out the following lines of code with a preceeding # and run the program again.

```
 #if worldMap[int(wm.camera.x)][int(moveY)] == 0:
 #    wm.camera.y += wm.camera.diry * moveSpeed
```

When the code is run, you'll only be able to move forward along the z-axis and not the horizontal axis because the mathematics processing has only been done for one axis. Comment back in the code so the program can be run normally.

When the Down cursor key is pressed, a similar operation happens apart from the camera position value is decreased instead of increased. See below.

```
 if(worldMap[int(wm.camera.x - wm.camera.dirx *
 moveSpeed)][int(wm.camera.y)] == 0):
   wm.camera.x -= wm.camera.dirx * moveSpeed

 if(worldMap[int(wm.camera.x)][int(wm.camera.y -
 wm.camera.diry * moveSpeed)] == 0):
   wm.camera.y -= wm.camera.diry * moveSpeed
```

When the Right cursor key is pressed to turn the player to the right (or rotate the player in a clockwise direction), the following code is executed:

```
oldDirX = wm.camera.dirx
wm.camera.dirx = wm.camera.dirx * math.cos(-
rotSpeed) - wm.camera.diry * math.sin(- rotSpeed)
wm.camera.diry = oldDirX * math.sin(- rotSpeed) +
wm.camera.diry * math.cos(- rotSpeed)

oldPlaneX = wm.camera.planex
wm.camera.planex = wm.camera.planex * math.cos(-
rotSpeed) - wm.camera.planey * math.sin(- rotSpeed)
wm.camera.planey = oldPlaneX * math.sin(- rotSpeed)
+ wm.camera.planey * math.cos(- rotSpeed)
```

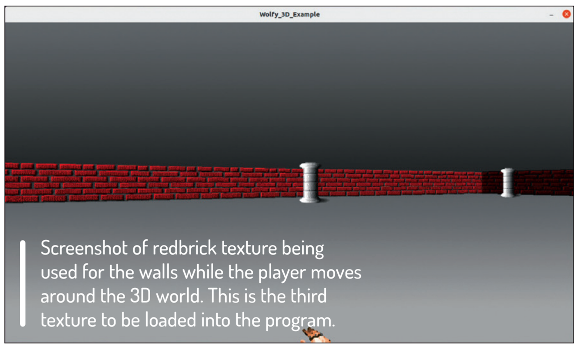
For everything to be consistent and work correctly, both the camera direction and camera plane must be rotated at the same time. To experiment with the above code, comment out one of the sections as we did before by using #, save and re-run the program.

You may notice that not only is the 3D effect compromised, the movement is also compromised. Comment back in the section of code you've commented out and save so the program works correctly again.

When the Left cursor key is pressed to turn the player to the left (or rotate the player in an anti-clockwise direction), the similar code is executed as above. However, the value of **rotSpeed** is passed into the trigonometric functions with a positive value rather than negative. Notice that the – sign isn't present in the code that's executed when the Left cursor key is pressed. It's also worth mentioning that the angle value is processed in radians and not degrees. The rotational speed is set earlier in **main.py** (see below):

```
rotSpeed = frameTime * 2.0 # the constant value is in
radians / second
```

As can be seen from the above code segment, the rotational speed is set to two radians a second offset by how long the frame takes to process. One radian equals 57.2958 degrees, two radians equals 114.5916 degrees, so each time the left or right cursor key is pressed the player view is rotated 114.5916 degrees.



Screenshot of redbrick texture being used for the walls while the player moves around the 3D world. This is the third texture to be loaded into the program.

Face the enemy

From the values that are generated from the above code, it is possible to take these values and use them to determine which direction the player is facing. For the benefit of this tutorial, the following values have been worked out depending on which direction the player has turned to face.

Front Direction Facing values (which are the default starting values):

```
wm.camera.dirx = -1.0
wm.camera.diry = 0.0
wm.camera.planex = 0.0
wm.camera.planey = 0.66
```

Right Direction Facing values:

```
wm.camera.dirx = -0.25204831631892727
wm.camera.diry = 0.9677146512483924
wm.camera.planex = 0.6386916698239392
wm.camera.planey = 0.16635188877049237
```

Rear Direction Facing values:

```
wm.camera.dirx = 0.9981760309968467
wm.camera.diry = -0.06037061489986301
```

» PYTHON MATHEMATICS LIBRARY

Python has a powerful mathematics library that has a range of uses. In regards to the project featured in this tutorial, the trigonometric mathematical functions have been used such `math.cos` and `math.sin` and also `math.abs`. The Python math library can be included by writing this code in a Python script:

```
import math
```

The functions from the maths library that have been used in this project are:

`math.cos(x)` : the cosine of x radians
`math.sin(x)` : the sine of x radians
`math.abs(n)` : the absolute value of the specified number

In the Python script file, **main.py**, two of the three trigonometric mathematical functions have been used when the player turns either left or right in the game environment. See this code example from the project to learn how this has been achieved:

```
oldDirX = wm.camera.dirx
wm.camera.dirx = wm.camera.dirx * math.cos(- rotSpeed) - wm.camera.diry * math.sin(- rotSpeed)
wm.camera.diry = oldDirX * math.sin(- rotSpeed) + wm.camera.diry * math.cos(- rotSpeed)
oldPlaneX = wm.camera.planex
wm.camera.planex = wm.camera.planex * math.cos(- rotSpeed) - wm.camera.planey * math.sin(- rotSpeed)
wm.camera.planey = oldPlaneX * math.sin(- rotSpeed) + wm.camera.planey * math.cos(- rotSpeed)
```

In the Python script file, **WorldManager.py**, the `abs` function is used when rendering the 3D world, as shown here:

```
if (side == 0):
  perpWallDist = (abs((mapX - rayPosX + (1 - stepX) / 2) / rayDirX))
else:
  perpWallDist = (abs((mapY - rayPosY + (1 - stepY) / 2) / rayDirY))
```

```
wm.camera.planex = -0.03984460583390901
wm.camera.planey = -0.658796180457919
```

At the moment, when the program starts it uses the front facing values as shown above. Now, as part of this tutorial, we're going to change the facing direction of the player, to face the right direction. Change the following code located in **main.py**

```
wm = worldManager.WorldManager(worldMap,sprite_positions, 20, 11.5, -1, 0, 0, .66)
```

to the following:

```
wm = worldManager.WorldManager(worldMap,sprite_positions, 20, 11.5, -0.25204831631892727,
0.9677146512483924, 0.6386916698239392,
0.16635188877049237)
```

It may be helpful to remember the input structure of the **WorldManager** constructor in that it takes in the following arguments in the following order:

```
- worldMap data structure
- Sprite positions, passing in data structure sprite_positions
- Camera X position (Z-axis)
- Camera Y position (X-axis)
- Camera Direction X
- Camera Direction Y
- Camera Plane X
- Camera Plane Y
```

Save the code and then run the program to see the result. You should now start facing the right direction in the 3D world. Feel free to continue to move round the 3D world as before to see that everything still works normally, before pressing the Escape key to exit the program and continue with the tutorial.

Now that you've done this for the right-facing direction, now have a go at changing the code so that the player faces the rear of the 3D world using the above code change as a guide. When changed, save and run the program again to see the result.

The above values were generated from pressing the Right cursor key continually so the player faces different directions. The camera direction and camera plane values that are needed next are for when the player faces to the left of the 3D world from continually turning to the right. To help you identify these values, place the following code under the condition where the Space bar is pressed in **main.py**, see below:

```
elif event.key == K_SPACE:
   print(wm.camera.dirx)
   print(wm.camera.diry)
   print(wm.camera.planex)
   print(wm.camera.planey)
```

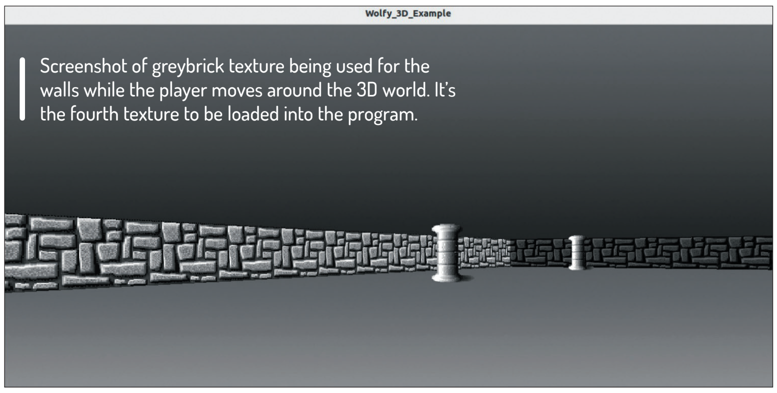
Now, when ever you press the Space bar when the program is running, it'll output the values in the terminal window for **wm.camera.dirx**, **wm.camera.diry**, **wm.camera.planex** and **wm.camera.planey**,which should now help you get the values for when the player is facing left in the 3D world.

Even though the program works in radians, we've been able to slightly manipulate the value output and set the direction of the player to 90-degree facing angles. For those that want to further improve on this, try obtaining values for every 45 degrees-facing turn. There'll be eight instances of values to retrieve (45 goes into 360 degrees eight times).

In this tutorial we have covered how the 3D world was first defined by a 2D array (**worldMap**), looked at how the images have been loaded in to be used by the program, gained a basic understanding of the mathematics used and introduced the Python mathematics library. We'll build on this in part two. See you next month! LXF

QUICK TIP

What's a radian then? One radian is the angle create when the radius distance is wrapped to a circle. It's roughly 57 degrees aka 180 / Pi. To go from radians to degrees: multiply by 180, divide by Pi. To go from degrees to radians: multiply by Pi, divide by 180.



Screenshot of greybrick texture being used for the walls while the player moves around the 3D world. It's the fourth texture to be loaded into the program.