



# Entrega final

## *Space Invaders*

Entornos Virtuales

Pedro Manuel Gómez-Portillo López

[gomezportillo@correo.ugr.es](mailto:gomezportillo@correo.ugr.es)

14 de mayo de 2019

# Índice

1. Introducción.....	3
2. Creación del menú principal.....	4
3. Creación del juego.....	9
3.1. Movimiento de la nave espacial.....	9
3.2. Rotación de la nave espacial.....	11
3.3. Generación de enemigos.....	14
3.4. Disparar y destruir a los enemigos.....	16
3.5. Vida y puntuación.....	18
3.6. Game over.....	24
3.7. Disparos enemigos.....	28
3.8. Jefe final.....	30
3.8. <i>Splash screen</i> y créditos.....	37
3.9. Sonidos.....	39
3.11. Fondo parallax.....	41
3.13. Generar ejecutable.....	43
3.14. Trabajo futuro y conclusiones.....	44

## 1. Introducción

Blender es un programa *open source* y multiplataforma dedicado especialmente al modelado, iluminación, renderizado, animación de modelos tridimensionales.



Logo de Blender

Blender incluye Blender Game Engine (BGE a partir de ahora), un motor de juegos que se utiliza para crear contenido interactivo en tiempo real. El motor del juego fue escrito desde cero en C++ como un componente mayoritariamente independiente e introducido en la versión 2.37. Incluye soporte para funciones como las secuencias de comandos Python o el motor de físicas Bullet.

Además, también permite crear entornos interactivos en los que el usuario pueda controlar elementos de la escena utilizando el ratón y el teclado como interfaz.

En esta práctica, la entrega final de la asignatura de Entornos Virtuales, se hará uso de todo lo aprendido a lo largo de la asignatura para desarrollar un juego del estilo del mítico *Space Invaders*, un arcade diseñado originalmente por Toshihiro Nishikado y lanzado al mercado en 1978 que ha inspirado toda clase de juegos<sup>1</sup>. Aún así no será exactamente el mismo juego, ya que en el nuestro se navegará horizontalmente y las naves irán apareciendo poco a poco.



Logo del *Space Invaders* original

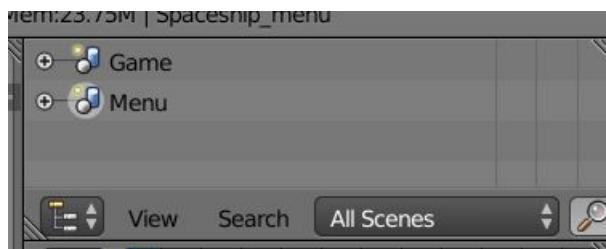
---

<sup>1</sup> [https://es.wikipedia.org/wiki/Space\\_Invaders](https://es.wikipedia.org/wiki/Space_Invaders)

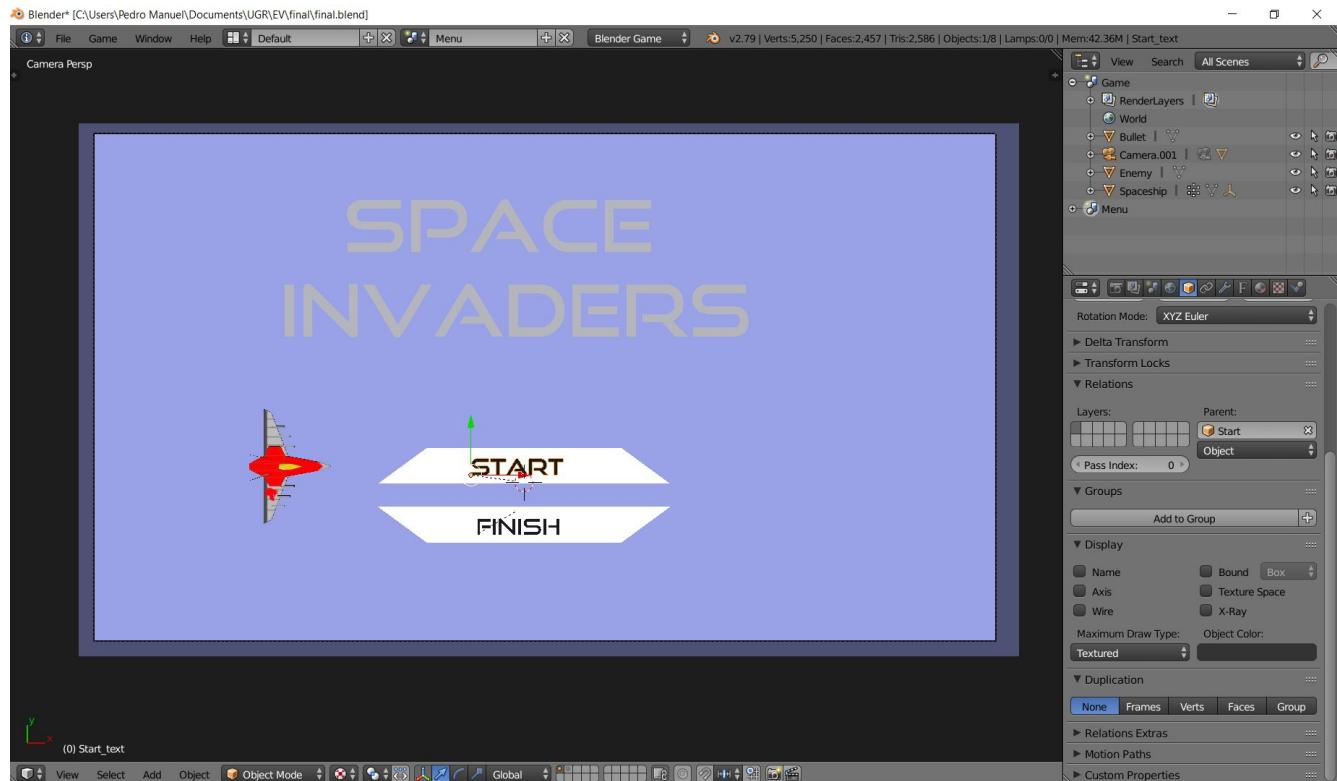
## 2. Creación del menú principal

Se pretende que este sea un juego completo, por lo que tendremos que crear un menú inicial en el que el jugador pueda empezar la partida. Como tendremos varios entornos diferentes (un menú, el juego en sí, la pantalla de game over...) usaremos **varias escenas** en las que encapsularemos cada uno de estos entornos.

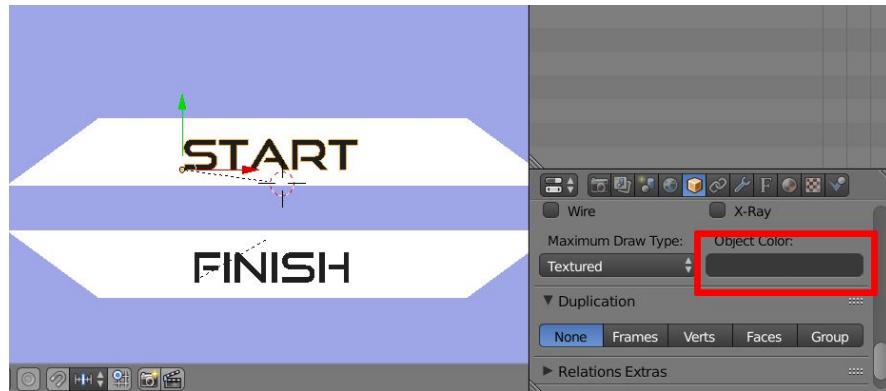
Como partiremos del ejercicio 7 de prácticas, lo primero que haremos será crear una nueva escena, a la que llamaremos *Menú*, y renombraremos la ya existente a *Juego*.



Ahora podemos empezar a modelar el menú. Usaremos un diseño muy sencillo en el que el aparezca el logo del juego en grande y dos botones, *Empezar* y *Salir*, que el jugador pueda seleccionar moviendo una nave.



Si queremos cambiar el color del texto en Blender Game no funcionará con cambiar el material, sino que debemos ir a la pestaña *Object>Display* y cambiar el color del objeto.



Se han utilizado un plano para hacer el fondo, dos planos para hacer los botones y texto para escribir su contenido. Este texto se ha emparentado con los botones para que si movemos uno se mueva con él. Además, se ha añadido otro texto con el nombre del juego en grande y la nave que se utilizará para seleccionar la acción a realizar. La fuente usada para las letras es *Good times*<sup>2</sup>

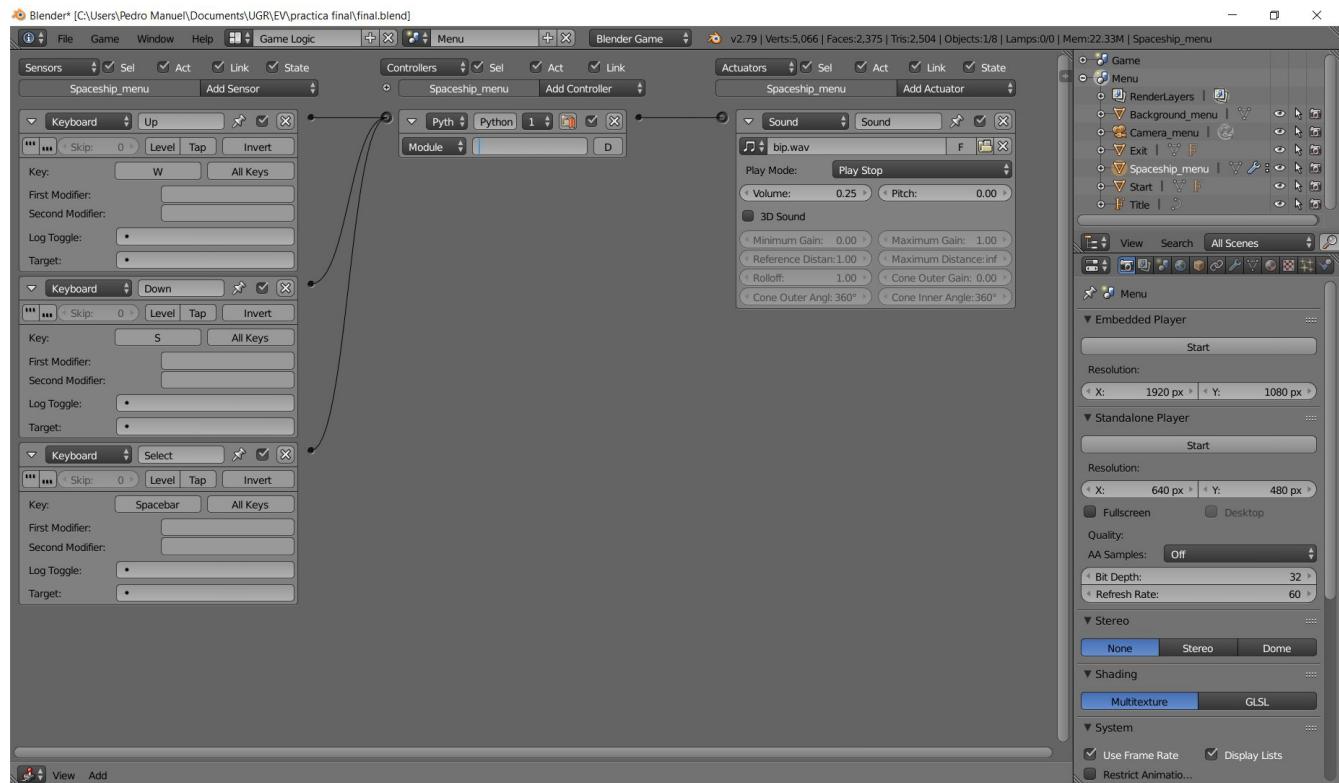
Aunque los colores parecen un poco apagados, al empezar el juego cambian.



2 <https://www.dafont.com/es/good-times.font>

Ahora daremos vida a la nave. Para ello, lo primero que deberemos hacer es abrir la vista de *Logic editor* y agregar tres sensores de tipo *Keyboard*, uno que controlará que la nave suba (*W*), otro que baje (*S*), y otro que permita que el jugador haga su selección (*Espacio*). Para poder acceder a ellos desde el código, al primero le llamaremos *Up*, al segundo *Down* y al tercero *Select*.

Conectaremos estos sensores a un controlador Python y, como actuador, añadiremos un de tipo *Sound* en el que seleccionaremos un pequeño pitido para que suene cada vez que el jugador mueva la nave que se encuentra en la ruta *sounds/bip.wav*.



Si intentamos ejecutar el juego ahora veremos que no hace nada, ya que debemos crear el script de Python que controle la lógica que queremos implementar.

Se ha creado un script en la ruta *scripts/game.py* con el siguiente código. Lo que hace es lo siguiente; tras importar la librería necesaria, primero recuperamos los sensores con los nombres que hemos usado y el objeto que tendrá el script (*owner*), que en este caso será la nave espacial. Tras definir las posiciones de la nave para que esté alineada con el botón de iniciar y con el de salir y guardarlas en variables, podemos empezar a realizar las comprobaciones pertinentes para que,

- Si la nave está alineada con el botón de inicio y pulsamos *Down* baje al botón de salir y activamos el sonido que hemos seleccionado antes.
- Si la nave está alineada con el botón de salir y pulsamos *Up* suba al botón de iniciar y activamos el sonido.

- Si pulsamos *Select* y la nave está en el botón de inicio, cambiamos la escena a la que se llama *Game*.
- Y si pulsamos *Select* y la nave está en el botón de salir terminamos el juego.

```
import bge

def menu(controller):
    controller = bge.logic.getCurrentController()
    scene = bge.logic.getCurrentScene()

    up      = controller.sensors['Up']
    down    = controller.sensors['Down']
    select  = controller.sensors['Select']

    sound = controller.actuators['Sound']

    spaceship = controller.owner
    start_y_position = 0
    exit_y_position  = -2

    if down.positive and spaceship.worldPosition.y == start_y_position:
        spaceship.worldPosition.y -= 2
        controller.activate(sound)

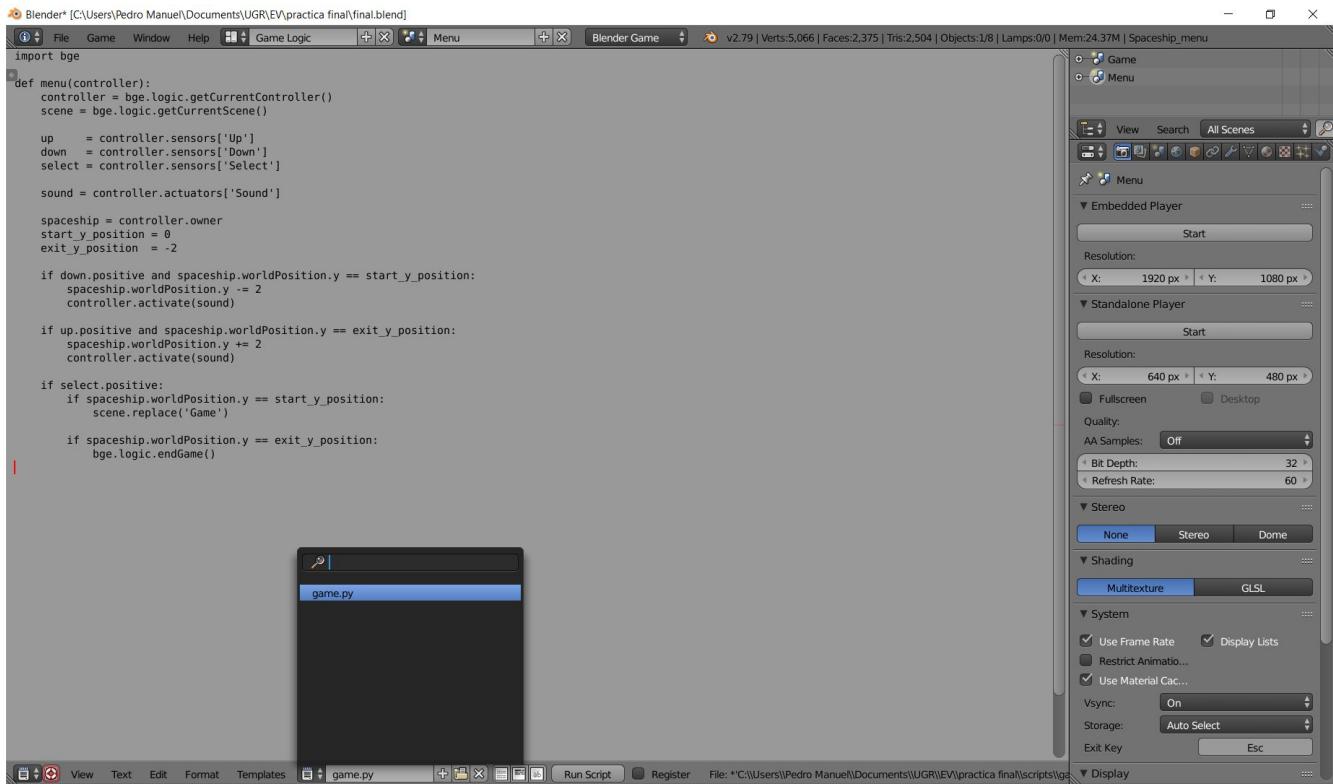
    if up.positive and spaceship.worldPosition.y == exit_y_position:
        spaceship.worldPosition.y += 2
        controller.activate(sound)

    if select.positive:
        if spaceship.worldPosition.y == start_y_position:
            scene.replace('Game')

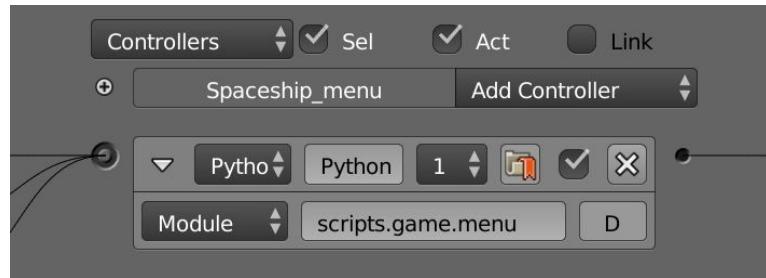
        if spaceship.worldPosition.y == exit_y_position:
            bge.logic.endGame()
```

Ahora tenemos que decirle a Blender que utilice este código como controlador. Podemos, o bien cargarlo en el editor de Blender, como se muestra en la imagen inferior, o simplemente indicar la ruta a Blender del método que queremos cargar, para lo que separaremos la ruta con puntos como se hace en Python.

En este caso por ejemplo, como queremos cargar el método *menu* del archivo *scripts/game.py* tendremos que indicar *scripts.game.menu* como ruta. Algunas de las imágenes están mal porque a mitad del desarrollo se movieron todos los scripts, que estaban en el directorio raíz del proyecto, a la carpeta *scripts/*.



Ahora ya podemos seleccionarlo desde el *Logic editor*. Para ello, seleccionamos *Module* como método de ejecución y *game.menu* para que Blender sepa a qué módulo de la clase *Game* tiene que utilizar.



Con esto ya podemos comenzar el juego. Ahora podemos movernos con *W* y *S* y seleccionar con *espacio*. Al seleccionar *Start* seremos llevados a la escena *Game* que contiene los avances realizados hasta el ejercicio 7 en el que podíamos mover la nave con *WASD* y hacer que disparara pulsando *espacio*.

### 3. Creación del juego

En este capítulo se explicará cómo se ha desarrollado el juego propiamente dicho.

#### 3.1. Movimiento de la nave espacial

Aunque en la práctica número 7 ya se dotó de movimiento a la nave espacial, pero como es necesario limitar este movimiento para evitar que el jugador salga de los límites de la cámara es necesario desecharlo y rehacerlo de manera programática.

Para ello, crearemos un script llamado *scripts/player.py* en el que iremos escribiendo la lógica del jugador. Para hacer que se mueva, primero tendremos que recoger los sensores (que previamente habremos llamado *W*, *S*, *D* y *A*), y tras comprobar que el jugador se encuentra dentro del rango del espacio en el que puede moverse lo desplazaremos modificando su posición global en 0.1 unidades.

```
import bge

def move(controller):
    spaceship = controller.owner

    w_key = controller.sensors['W']
    s_key = controller.sensors['S']
    d_key = controller.sensors['D']
    a_key = controller.sensors['A']

    STEP = 0.1

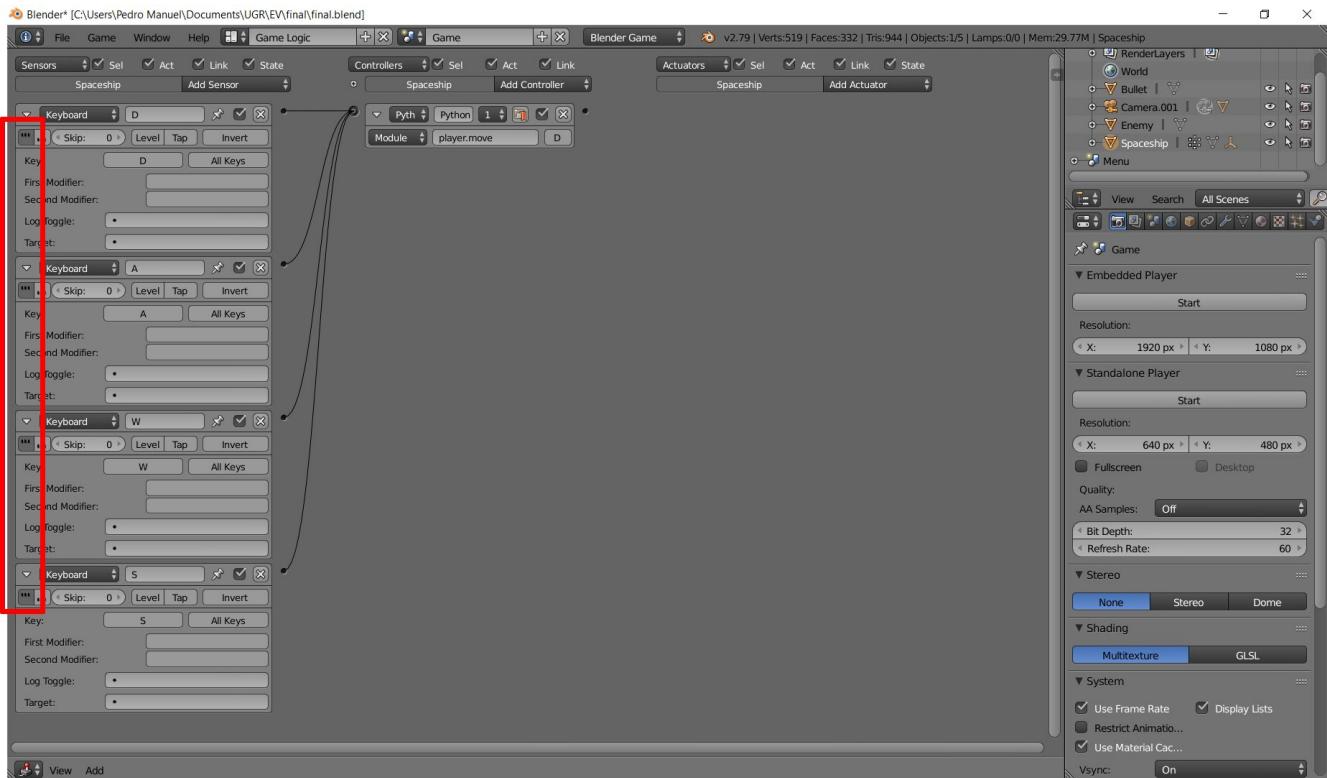
    if w_key.positive and spaceship.worldPosition.z < 3.5 :
        spaceship.worldPosition.z += STEP

    if s_key.positive and spaceship.worldPosition.z > -3.5:
        spaceship.worldPosition.z -= STEP

    if a_key.positive and spaceship.worldPosition.x > -6:
        spaceship.worldPosition.x -= STEP

    if d_key.positive and spaceship.worldPosition.x < 4:
        spaceship.worldPosition.x += STEP
```

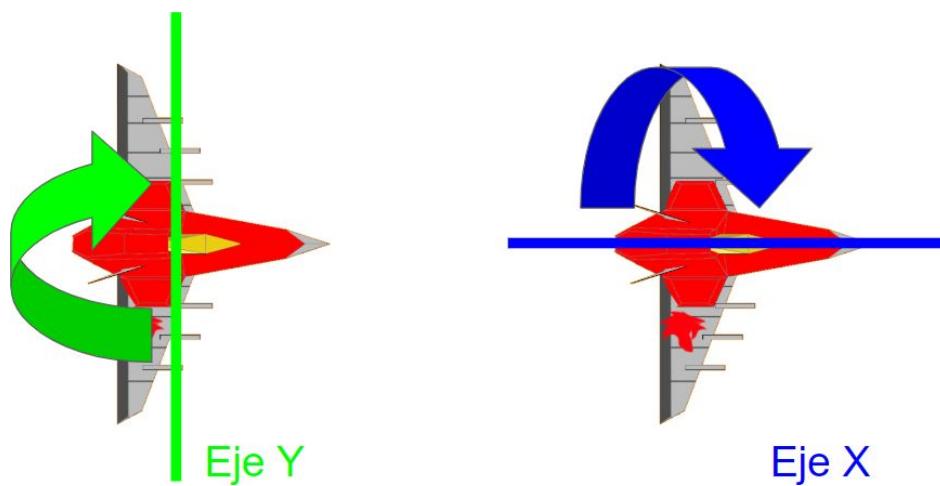
Ahora seleccionaremos a la nave en el archivo Blender y siguiendo el proceso explicado en el capítulo anterior relacionaremos este método como los sensores *Keyboard*.



Es importante que seleccionemos el botón que he marcado en rojo para que el sensor esté mandando la señal de la tecla siempre que esté pulsada; si no, simplemente mandaría la señal cuando la pulsáramos y tendríamos que levantar el dedo y volver a pulsar la tecla para que la nave volviera a moverse 0.1 unidades.

## 3.2. Rotación de la nave espacial

El actual problema con la nave es que su movimiento es muy plano y no es especialmente satisfactorio moverla. Para solucionar esto, haremos que la nave rote ligeramente siguiendo el movimiento que hace, en sus eje Y al moverla hacia delante y atrás y en su eje X al moverla arriba y abajo como muestra la figura inferior.



El siguiente código es el resultado de más de una hora de implementación y pruebas hasta encontrar los rangos de rotación máxima y cuánto rotar la nave cada vez hasta dar con unos valores satisfactorios, por lo que no lo comentaré en detalle sino aquellas cosas que considero más interesantes. Aunque en el código final está incluido con el código que hemos desarrollado antes para mover la nave, en el fragmento interior esa parte ha sido obviada para hacerlo más leíble.

Lo que hace, en resumen, es comprobar si la nave se está moviendo hacia delante o hacia atrás atrás para rotarla acordemente hasta un giro máximo y lo mismo con arriba y abajo, y si la nave no se está moviendo resetea los ejes de giro poco a poco.

Una de esas cosas es que para obtener la rotación de un objeto en Blender debemos usar la función *worldOrientation*<sup>3</sup>, que nos devolverá una matriz 3x3 con los ejes de giro. Para editarla, deberemos obtener un vector con la rotación euler con la función *to\_euler()*, que trabaja con radianes en vez de con grados, y como es más natural trabajar con los segundos usaremos la función de Python *math.radians(DEGREES)* para trabajar con ellos convirtiéndolos desde grados. Al final del todo, cuando hayamos actualizado por completo el vector con la rotación euler del objeto, la volvemos a transformar a matriz con *spaceship.worldOrientation = euler\_rotation.to\_matrix()*.

---

3 [https://docs.blender.org/api/2.78a/bge.types.KX\\_GameObject.html#bge.types.KX\\_GameObject.worldOrientation](https://docs.blender.org/api/2.78a/bge.types.KX_GameObject.html#bge.types.KX_GameObject.worldOrientation)

Además, para que el script se active debe recibir la señal de algún actuador, y paradógicamente para que los ejes de giro se reseteen no debe llegar ninguna señal de activación de sus respectivas teclas, por lo que se ha añadido un actuador de tipo *Always* que uno de cada dos frames (para que no consuma tantos recursos) envía una señal al controlador para que se active.



```
def move(controller):
    [...]
    # Rotation
    matrix_rotation = spaceship.worldOrientation
    euler_rotation = matrix_rotation.to_euler()
    ROTATION_STEP = math.radians(4)
    MAX_ROTATION = math.radians(25)

    if w_key.positive and spaceship.worldPosition.z < 3.5 :
        if euler_rotation.x > -MAX_ROTATION:
            euler_rotation.x -= ROTATION_STEP

    if s_key.positive and spaceship.worldPosition.z > -3.5:
        if euler_rotation.x < MAX_ROTATION:
            euler_rotation.x += ROTATION_STEP

    # reset the X rotation
    if not s_key.positive and not w_key.positive and euler_rotation.x != 0:
        if euler_rotation.x > 0:
            if euler_rotation.x < ROTATION_STEP:
                euler_rotation.x = 0
            else:
                euler_rotation.x -= ROTATION_STEP
        else:
            euler_rotation.x += ROTATION_STEP

    if a_key.positive and spaceship.worldPosition.x > -6:
        if euler_rotation.z > -MAX_ROTATION:
            euler_rotation.z -= ROTATION_STEP

    if d_key.positive and spaceship.worldPosition.x < 4:
        if euler_rotation.z < MAX_ROTATION:
            euler_rotation.z += ROTATION_STEP

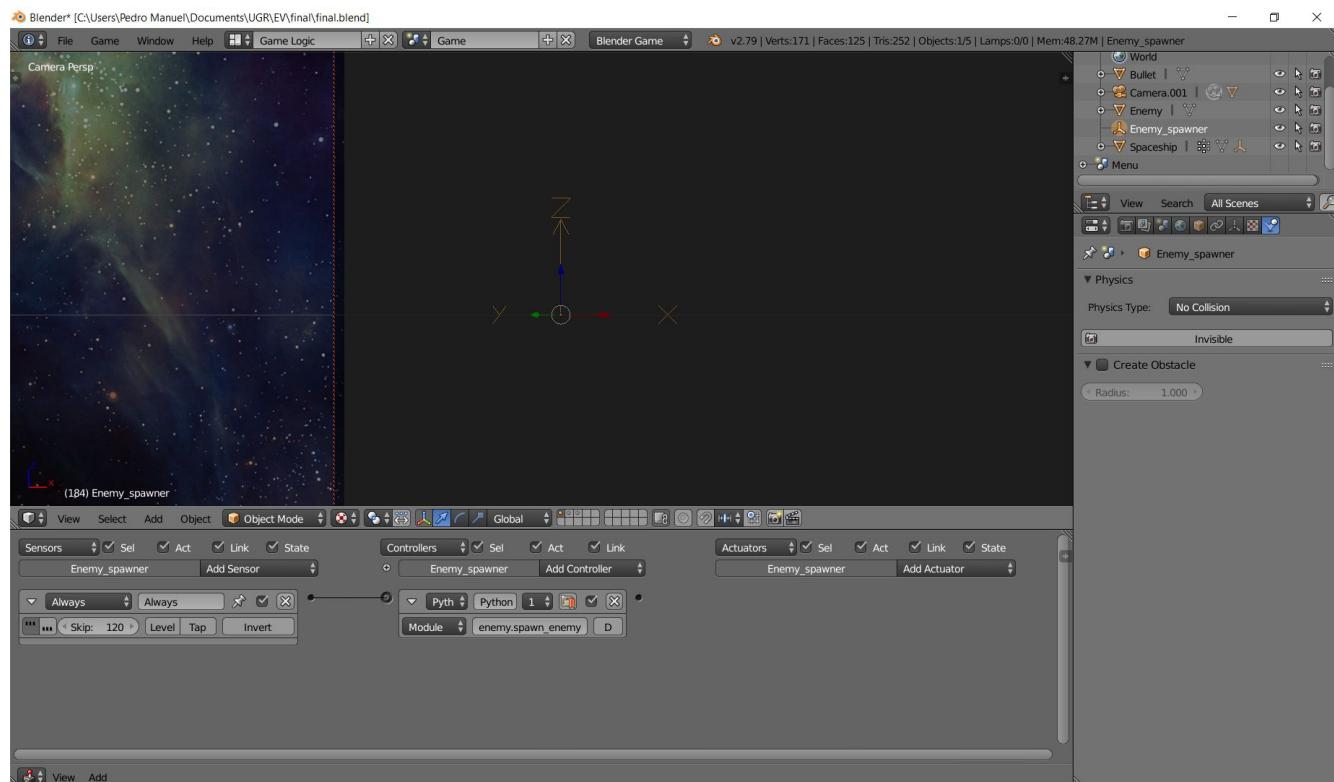
    # reset the Z rotation
    if not a_key.positive and not d_key.positive and euler_rotation.z != 0:
        if euler_rotation.z > 0:
            if euler_rotation.z < ROTATION_STEP:
                euler_rotation.z = 0
            else:
                euler_rotation.z -= ROTATION_STEP
```

```
else:  
    euler_rotation.z += ROTATION_STEP  
  
spaceship.worldOrientation = euler_rotation.to_matrix()
```

### 3.3. Generación de enemigos

Lo primero que se ha hecho para añadir enemigos es mover el modelo del enemigo, *Enemy*, a la tercera capa del archivo Blender y añadir un *empty* a la primera, fuera del campo visual de la cámara. Este *empty* será el *spawner* o el punto desde el que aparecerán los enemigos.

Como queremos que la altura a la que aparecen los enemigos sea aleatoria deberemos hacerlo de forma programática; para ello, le añadiremos primero un sensor tipo *Always* que mande una señal cada 120 frames (si asumimos 60fps, cada dos segundos) unido a un controlador Python desde el que cambiaremos la posición Z de nuestro *spawner*. Se ha creado otro archivo, *enemy.py*, que contendrá toda la lógica de los enemigos. Su primer método será *spawn\_enemy*, encargado de generar enemigos.



A continuación se muestra el código utilizado para esto. Se ha utilizado la función *random.uniform()*<sup>4</sup> de Python para generar la altura pseudo-aleatoriamente siguiendo una distribución uniforme.

4 <https://docs.python.org/3/library/random.html#random.uniform>

```

import bge
import random

random.seed(123)

def spawn_enemy(controller):
    spawner = controller.owner
    scene = bge.logic.getCurrentScene()

    # random Z position
    spawner.worldPosition.z = random.uniform(3, -3)

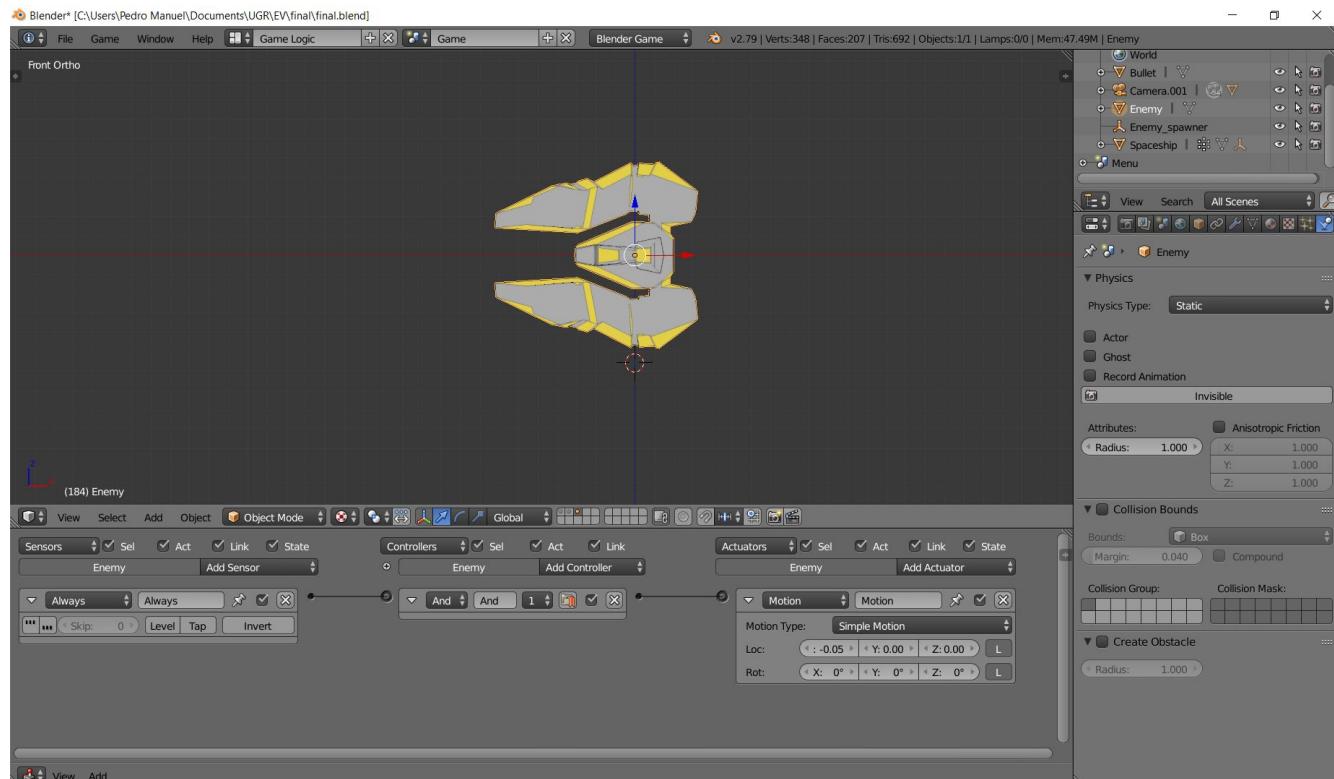
    scene.addObject("Enemy", "Enemy_spawner", 350)

```

El tercer parámetro de la función `scene.addObject()` indica el tiempo de vida del objeto, lo que nos viene genial porque no tendremos que eliminar a mano las naves que el jugador no destruya y salgan por la parte opuesta de la pantalla, ya que con darles un tiempo de vida suficiente para que el objeto salga Blender lo eliminará automáticamente.

Ahora los enemigos aparecerán aleatoriamente pero no se moverán; para conseguirlo basta con hacer que el enemigo original, el que se crea continuamente, se mueva para que sus copias lo hagan.

Para ello añadiremos un sensor *Always* unido a un actuador *Motion* que lo mueva 0.05 unidades en el eje -X.

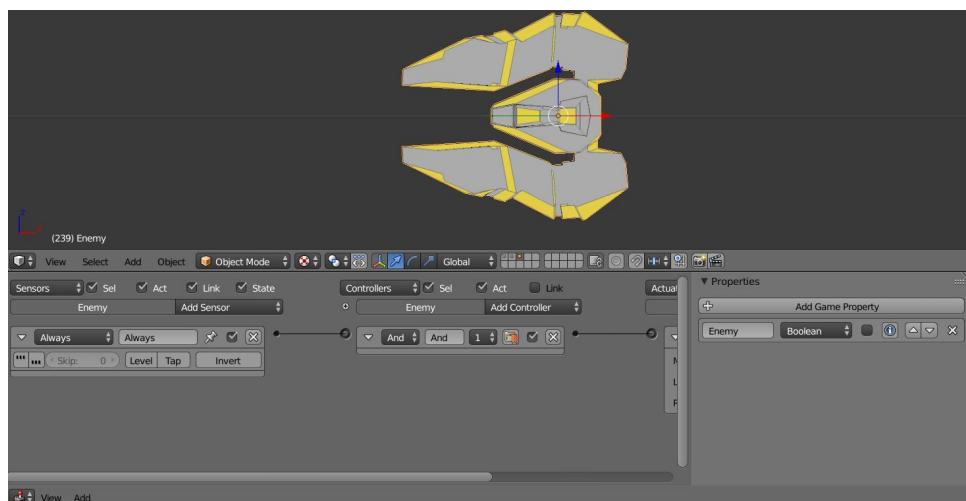


### 3.4. Disparar y destruir a los enemigos

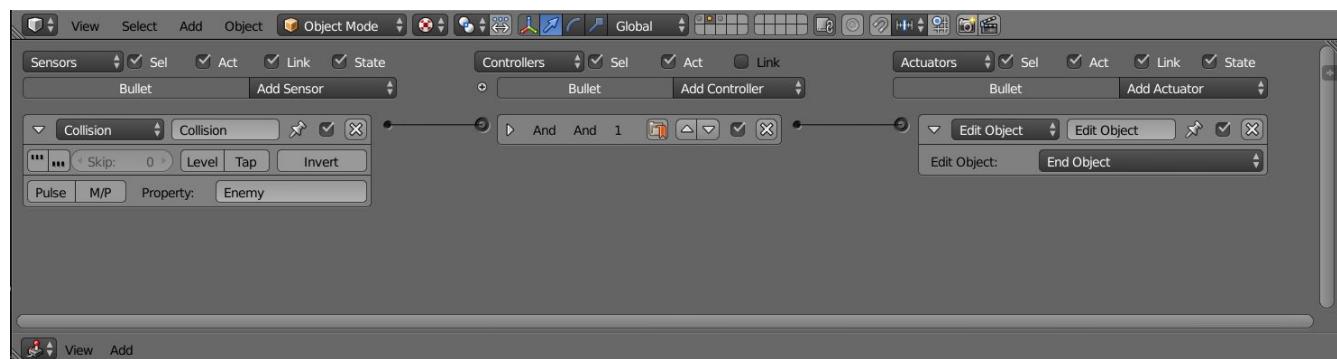
Ahora que el juego va cogiendo forma y ya podemos disparar a los enemigos, lo primero que haremos será hacer que cuando un disparo colisione con un enemigo, ambos desaparezcan.

Para hacer que la bala desaparezca, le añadiremos un sensor tipo *Collision* unido con un actuador tipo *Edit object* con *End object* seleccionado. Con esto más o menos bastaría, pero como queremos que solo desaparezca en caso de que colisione con un enemigo, usaremos las *game properties*<sup>5</sup>, que son variables que podemos aplicar a los objetos y utilizar para, por ejemplo, recuperarlas con los sensores.

Para añadir una propiedad a un modelo debemos seleccionarlo con la vista *Logic editor* abierta y, pulsando N si no aparece, acceder al panel de propiedades y añadir una personalizada; en este caso se llamará *Enemy* y será de tipo **booleano**, ya que solo puede **ser o no ser** enemigo.

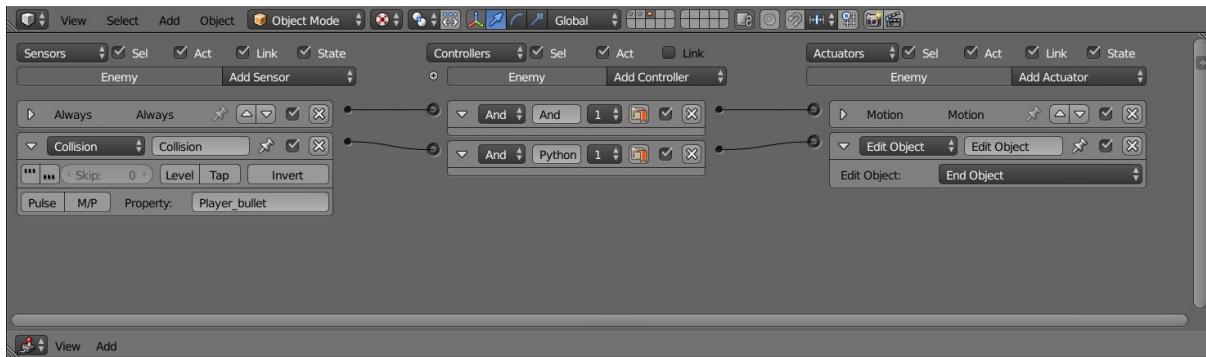


Ahora escribiremos el nombre de esta propiedad en el sensor de la bala para hacer que solo se active al colisionar con un objeto con esta propiedad.



5 [https://docs.blender.org/manual/en/dev/game\\_engine/logic/properties.html](https://docs.blender.org/manual/en/dev/game_engine/logic/properties.html)

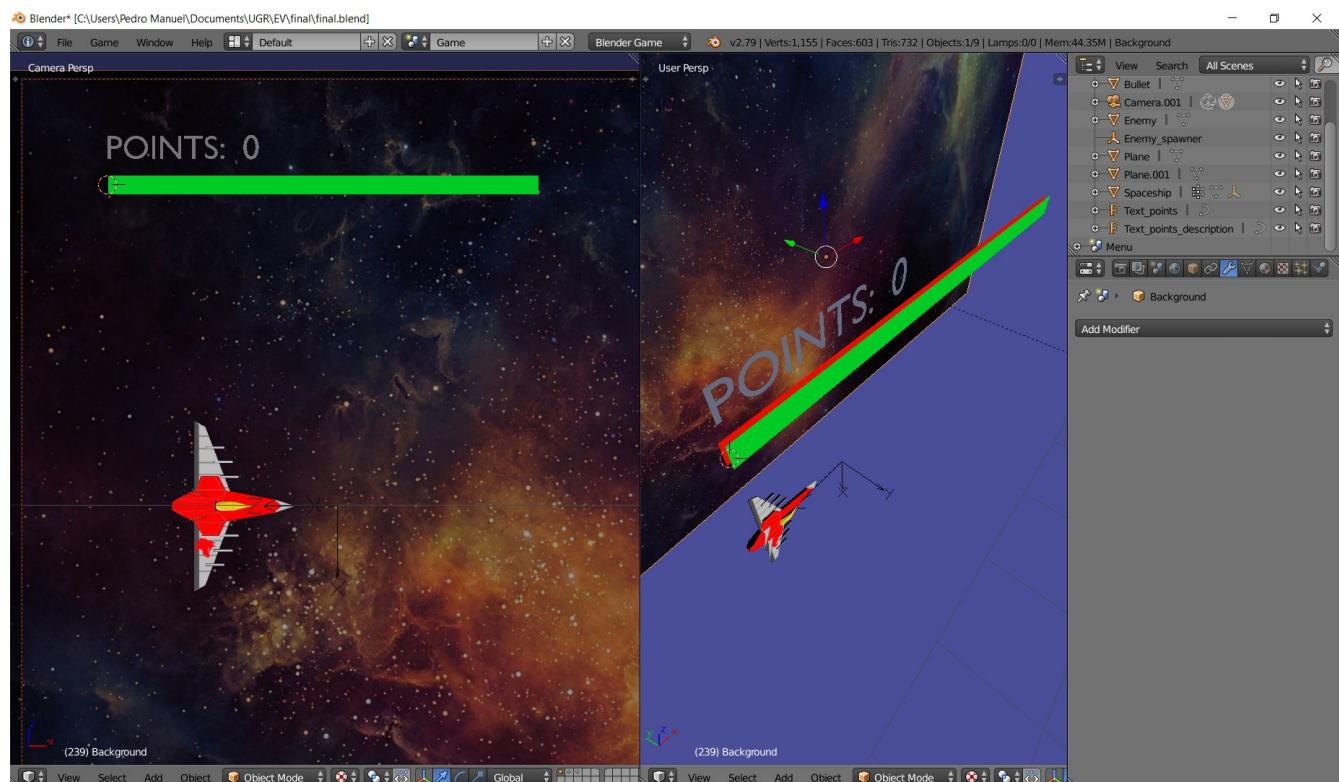
Ahora que ya eliminamos las balas, pasemos a hacer que desaparezcan los enemigos, y lo haremos de forma similar. Lo que haremos será, como en el caso anterior, añadir un sensor que detecte cuando colisionamos con una bala del jugador y que mande una señal a un actuador de tipo *Edit object>End object*.



Y con esto ya podríamos disparar y destruir naves enemigas indefinidamente.

### 3.5. Vida y puntuación

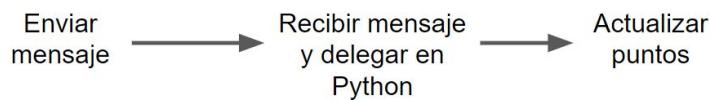
A continuación, lo que haremos será dotar de una barra de vida y un cuadro de texto con la puntuación numérica del jugador, valiendo cada nave destruida 10 puntos. Para ello, añadiremos tanto la barra de vida verde, hecha con un plano, y una barra roja detrás para que parezca que la verde se convierte en roja, como do textos diferentes, uno con la palabra *POINTS*: y otro con los puntos del jugador.



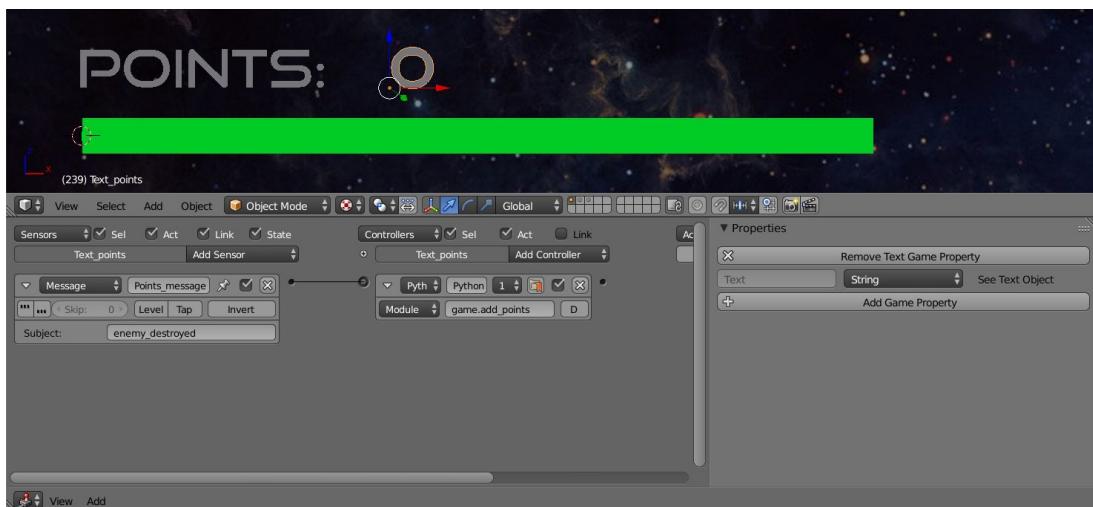
Ahora usaremos una de las herramientas más potentes que ofrece BGE; el **envío de mensajes**<sup>6</sup>. Con esta herramienta podemos crear sensores de mensaje que reciban mensajes de otros actuadores o desde un script de Python. Nosotros usaremos la primera.

En definitiva, lo que haremos es hacer que la nave envíe un mensaje al texto de puntuación y este use Python para leer su contenido y actualizarse a sí mismo.

6 [https://docs.blender.org/api/blender\\_python\\_api\\_current/bge.logic.html#bge.logic.sendMessage](https://docs.blender.org/api/blender_python_api_current/bge.logic.html#bge.logic.sendMessage)

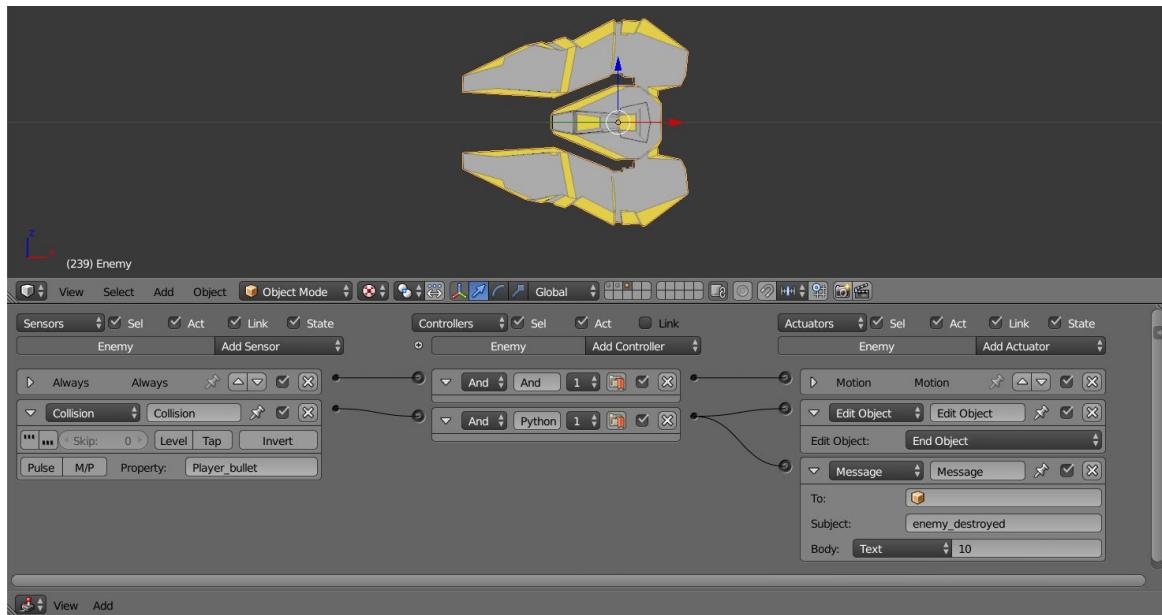


Lo primero que haremos será aumentar el contador de puntos con cada nave enemiga destruida; para ello, empezaremos añadiendo un sensor tipo *Message* al texto llamado *Points\_message* con *enemy\_destroyed* como *subject*. Ahora uniremos este sensor con un script de Python con el método *add\_points* del archivo *game*. Además, como los textos no pueden editarse desde código de manera nativa<sup>7</sup> tendremos que usar las propiedades para hacerlo. Por ello, añadimos la propiedad *Text game* que solo está disponible para los textos, de modo que editando esta propiedad se edita el texto.



Ahora debemos llamar a este sensor cada vez que un enemigo se destruya, por lo que añadiremos un actuador tipo *Message* al enemigo que se activará cuando impacte con una bala con el *subject* que hemos definido antes, *enemy\_destroyed*, y haremos que mande los puntos que tiene que sumarse el jugador por destruirlo en el cuerpo del mensaje como un campo de texto, que en este caso son 10.

<sup>7</sup> <https://blender.stackexchange.com/questions/6288/can-python-be-used-to-edit-a-text-object>



Ahora que ya recibimos el mensaje en el método `add_points` podemos leer el contenido del mensaje para actualizar el texto con la nueva puntuación. Para ello, accedemos al sensor a través de su nombre y leemos el cuerpo del último mensaje recibido, lo que nos da los puntos que debemos sumar al jugador.

```
def add_points(controller):
    text_points = controller.owner

    message = controller.sensors["Points_message"]
    points_to_add = int(message.bodies[0])

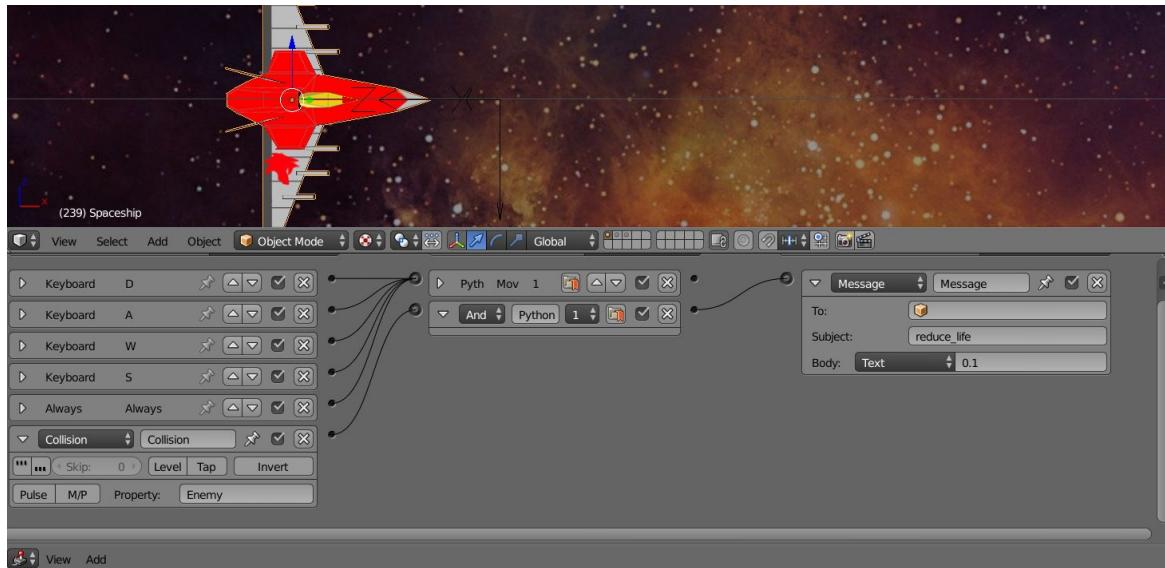
    current_points = int(text_points['Text'])
    current_points += points_to_add
    text_points['Text'] = current_points
```

Ahora cada vez que una nave enemiga sea destruida aumentará la puntuación del jugador en 10.

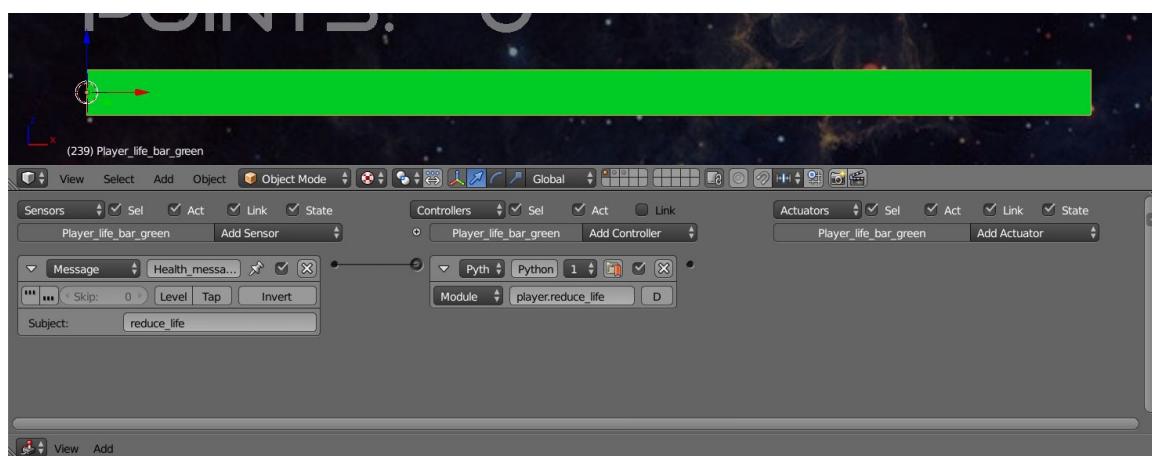


Para disminuir la barra de vida utilizaremos un método parecido, en el que la nave mandará un mensaje a la barra de vida cuando colisione con un enemigo indicándole qué porcentaje de vida debe perder.

Lo primero que haremos será añadir un actuador tipo *Message* con el tema *reduce\_life* y un porcentaje del 10% cada vez que el sensor de colisión que hicimos antes detecte a un enemigo.



Ahora añadiremos un sensor *Message* a la barra de vida con el mismo tema, *reduce\_life*, que llame a un script de Python, concretamente al método *reduce\_life()* del archivo *player.py*. Cabe destacar que, como ahora vamos a cambiarle la escala en el eje X, se ha hecho que el origen de la barra de vida sea su lado izquierdo.



El script quedaría de la siguiente forma; como puede verse, lo que se hace es recuperar el mensaje como en el caso anterior, ver qué porcentaje de vida ha de perder el jugador y reducirlo al porcentaje actual. Tras ello, se actualiza la barra de vida con su nuevo tamaño.

```
def reduce_life(controller):
    health_bar = controller.owner

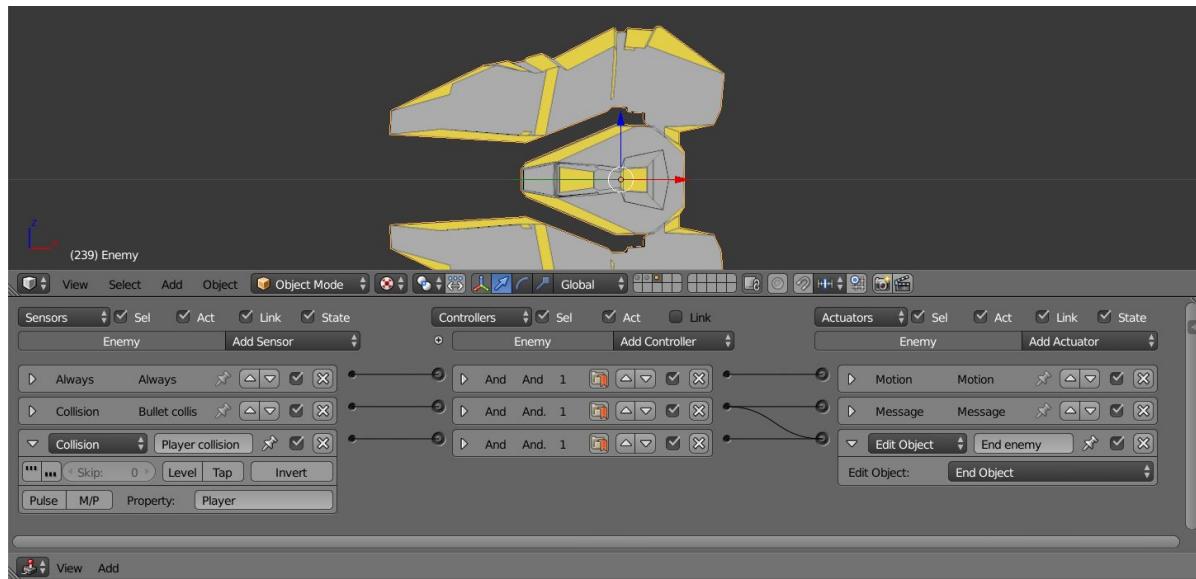
    message = controller.sensors["Health_message"]

    lost_life = float(message.bodies[0]) # between 0 and 1

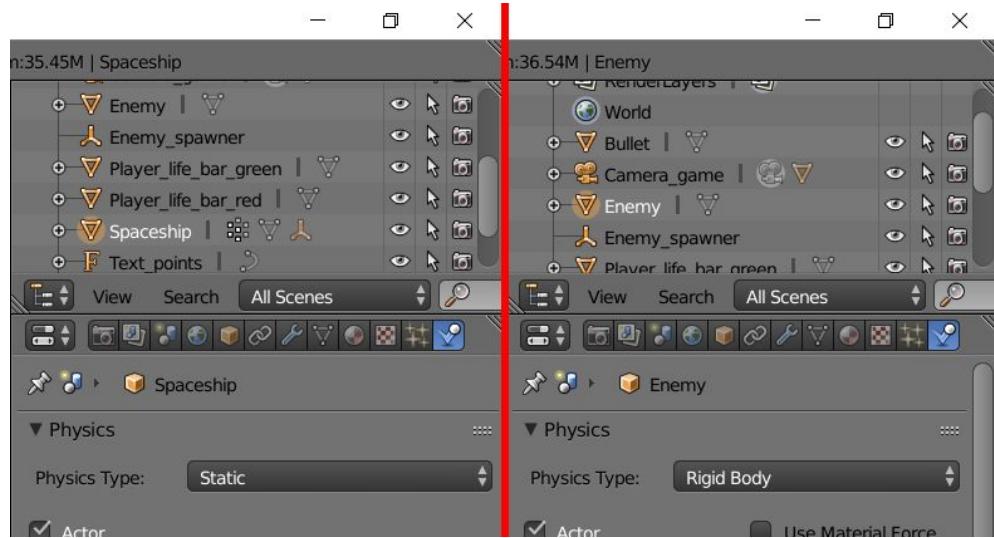
    x_scale = float(health_bar.localScale.x) # between 0 and 1
    new_x_scale = x_scale - lost_life

    if new_x_scale > 0:
        health_bar.localScale.x = new_x_scale
    else:
        health_bar.localScale.x = 0
        # game over
```

Por último solo queda hacer que los enemigos se destruyan al impactar con el jugador, para lo que basta con añadir un sensor *Collision* unido a un *Edit object>End object*. Reutilizaremos el que hicimos antes pero no mandaremos el mensaje para aumentar la puntuación del jugador.



Pero hay un problema, y es que como es un *Rigid body* la nave espacial es afectada por la inercia de los enemigos, por lo que cambia su posición y rotación al impactar contra ellos. Para evitar esto, y como en BGE los objetos estáticos no interactúan por defecto con otros estáticos<sup>8</sup> haremos que la nave sea *Static* y los enemigos *Rigid body*.



Ahora el jugador perderá un 10% de su vida cada vez que golpee a un enemigo.



---

8 <https://blender.stackexchange.com/a/14644/38294>

### 3.6. Game over

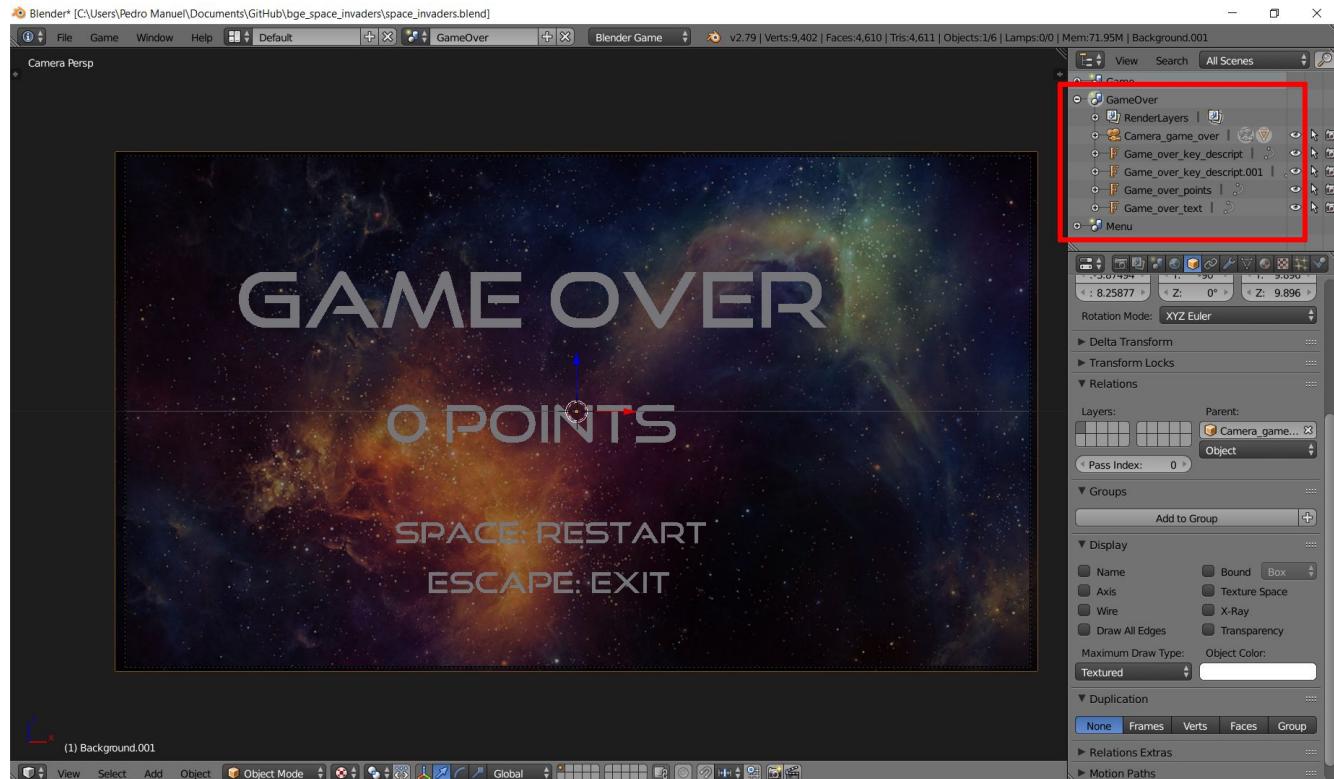
Como ya hemos visto en el script de Python para reducir la vida del jugador, es fácil saber cuándo el jugador se ha quedado sin vida y, por lo tanto, ha perdido. Cuando esto pase, se le mostrará un mensaje con su puntuación final y se le dará la opción de reiniciar la partida.

Para mostrar el mensaje de final de partida podemos hacer principalmente dos cosas,

- Usar la misma escena del juego para que, cuando el jugador pierda, se eliminen todos los objetos y aparezca el mensaje. Esta será la más fácil, ya que la función `Scene.restart()`<sup>9</sup>
- Usar una escena diferente en la que mostrar el mensaje y la puntuación final, a la que habrá que pasar la puntuación obtenida por el jugador.

Como la segunda parece más complicada, será la que implementemos.

Lo primero que haremos será crear una escena, llamada *GameOver*, que tendrá la misma imagen del espacio de fondo con un texto indicando al jugador que ha perdido, su puntuación y las teclas para reiniciar o salir del juego.



9 [https://docs.blender.org/api/blender\\_python\\_api\\_current/bge.types.KX\\_Scene.html#bge.types.KX\\_Scene.restart](https://docs.blender.org/api/blender_python_api_current/bge.types.KX_Scene.html#bge.types.KX_Scene.restart)

Aunque el texto esté descentrado, al iniciar el juego cambia ligeramente de posición y hace que se centre correctamente en pantalla.

Para pasar a esta escena cuando el jugador pierda, basta cargarla cuando el jugador se queda sin vida en el método *reduce\_life()* del archivo *player.py*, abajo en negrita.

```
def reduce_life(controller):  
    [...]  
  
    if new_x_scale > 0:  
        health_bar.localScale.x = new_x_scale  
    else:  
        health_bar.localScale.x = 0  
  
        # game over  
        scene = bge.logic.getCurrentScene()  
        scene.replace('GameOver')
```

Ahora que ya cargamos la escena, le pasaremos la puntuación final de jugador para que la muestre, para lo que usaremos la variable *globalDict*<sup>10</sup>, que es un diccionario compartido que puede ser accedido desde todas las escenas.

Empezaremos guardaremos la puntuación del jugador en este diccionario cuando pierda.

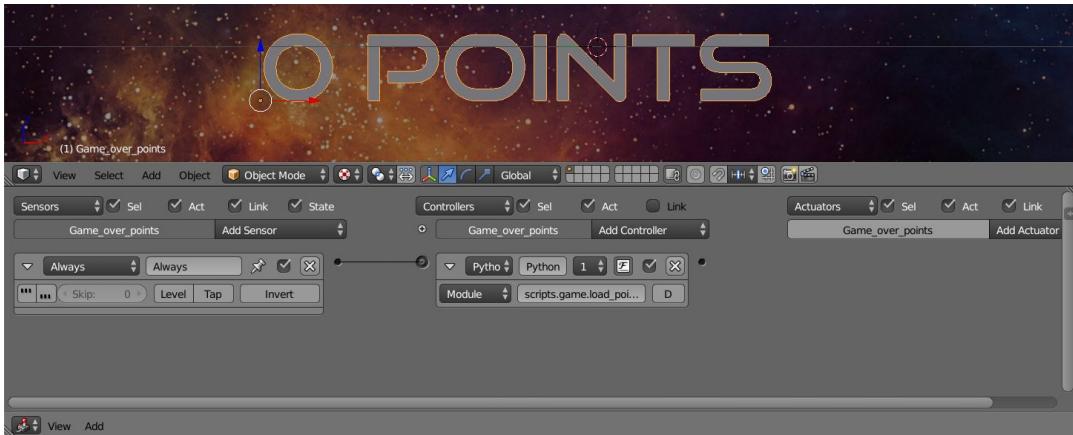
```
def reduce_life(controller):  
    [...]  
  
    # game over  
    scene = bge.logic.getCurrentScene()  
  
    object_list = scene.objects  
    text_points = object_list['Text_points']  
    bge.logic.globalDict['Points'] = text_points['Text']  
  
    scene.replace('GameOver')
```

Ahora pasamos a acceder a este diccionario desde la otra escena cuando cargue. Para ello se necesita que un script de Python se ejecute la escena. Para ello se ha añadido un sensor *Always* sin ningún skip seleccionado unido a un script de Python marcado como *startup script* al texto con la puntuación, como se indica en<sup>11</sup> y<sup>12</sup>.

10 [https://docs.blender.org/api/blender\\_python\\_api\\_2\\_61\\_0/bge.logic.html#bge.logic.globalDict](https://docs.blender.org/api/blender_python_api_2_61_0/bge.logic.html#bge.logic.globalDict)

11 <https://blender.stackexchange.com/a/52888/38294>

12 <https://blender.stackexchange.com/a/91680/38294>

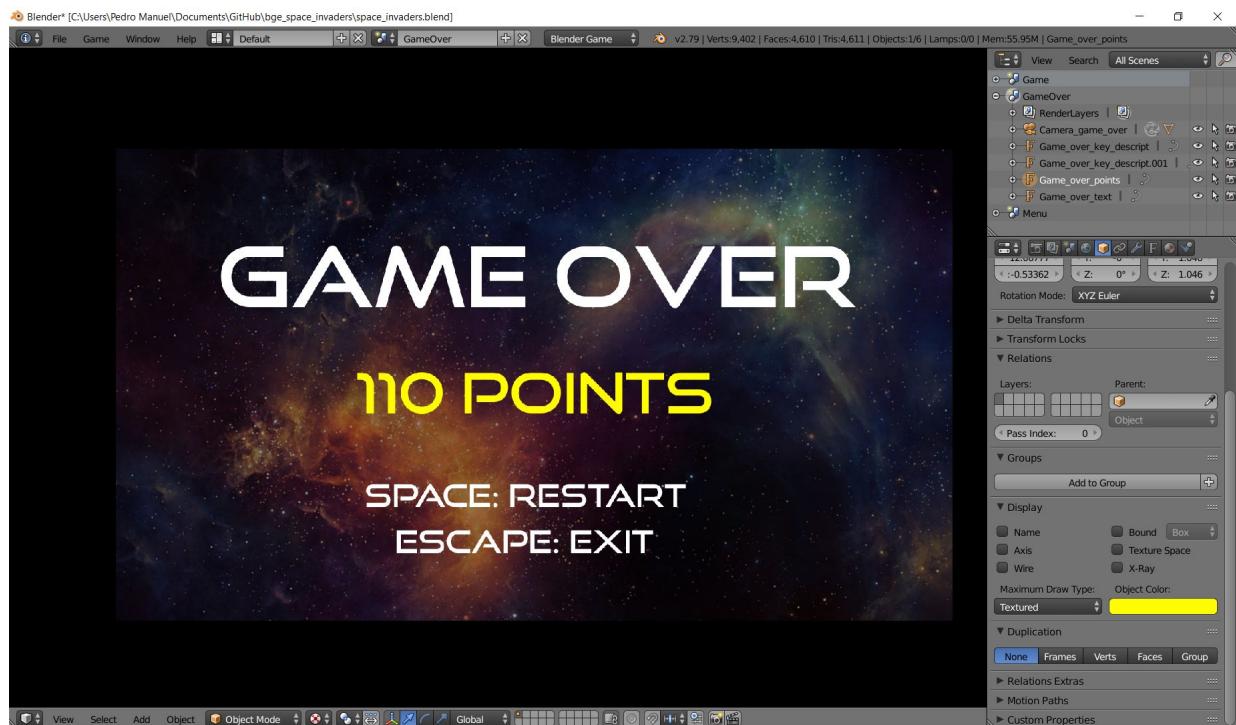


El script, en el archivo `scripts/game.py` se llama `load_points` y es el siguiente. Se comprueba que la clave `Points` exista en el diccionario para que no salte un error si se carga la escena desde Blender en lugar de hacerlo desde la escena del juego.

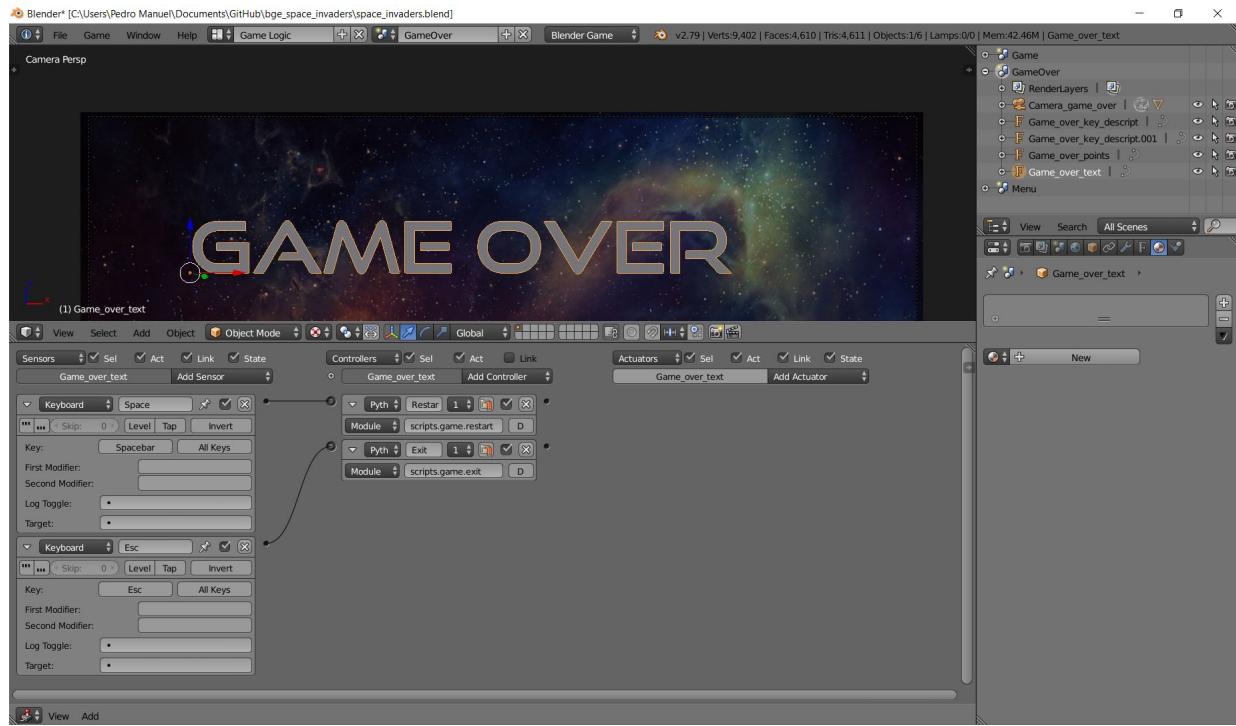
```
def load_points(controller):
    text_points = controller.owner

    if 'Points' in bge.logic.globalDict:
        text_points['Text'] = bge.logic.globalDict['Points'] + ' POINTS'
```

Y con esto ya podríamos ver nuestra puntuación al terminar el juego.



Por último haremos que el juego se reinicie pulsando espacio y se termine pulsando escape. Aunque no tenía mucho sentido desde un punto de vista de diseño que ninguno de los objetos que hay en la escena controlara las teclas pulsadas, se ha decidido que lo haga el texto con *GAME OVER* escrito.



Aunque no hubiera sido necesario sobrescribir la tecla *Escape* para terminar el juego, ya que Blender ya lo hace por defecto, se ha decidido hacerlo para aprender a terminar de manera programática. A continuación se muestran los dos métodos, *restart()* y *exit()*.

```
def restart(controller):
    bge.logic.getCurrentScene().replace('Game')

def exit(controller):
    bge.logic.endGame()
```

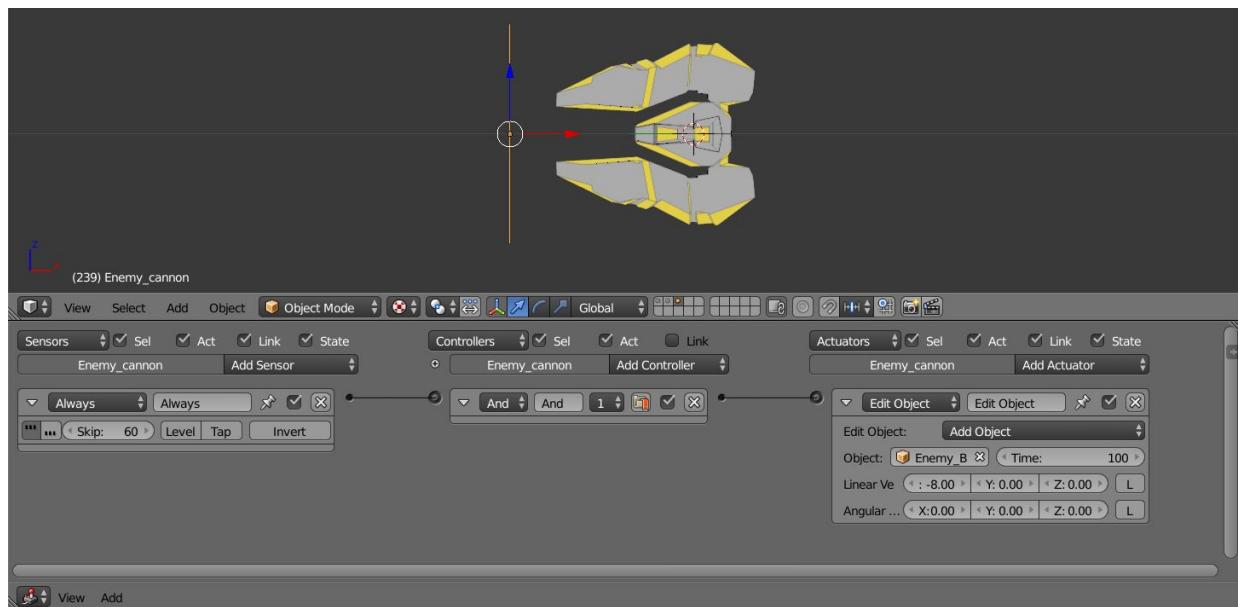
Con esto ya podemos reiniciar y terminar el juego.

### 3.7. Disparos enemigos

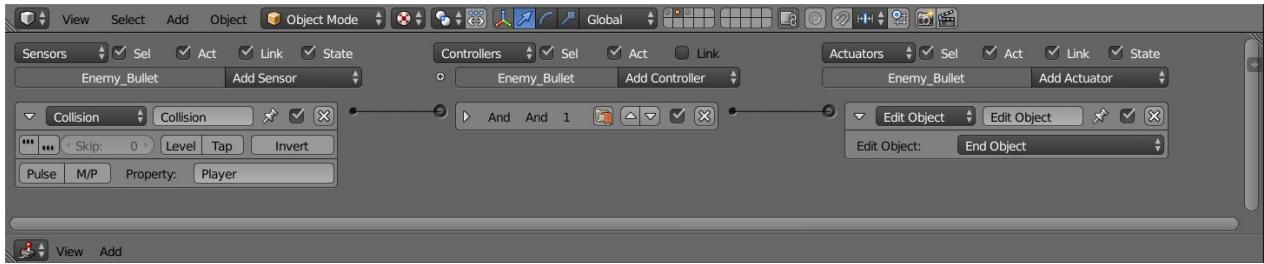
Ahora se implementarán los disparos enemigos, ya que de otra forma el juego sería demasiado sencillo y aburrido.

Lo primero que se hará será crear las balas enemigas, que serán simplemente unas esferas con un material rojo a la que se les ha añadido la propiedad booleana *Enemy\_bullet* para poder detectarlas más adelante.

Tras ello se añadirá un *empty* delante de la nave enemiga y emparentada con ella para que se muevan juntos, llamado *Enemy\_cannon*. Este *empty* será el encargado de generar los disparos enemigos, para lo que añadiremos un sensor *Always* que genere una señal cada 60 frames unido a un actuador tipo *Edit object>Add object* que genere una de las balas enemigas con una velocidad lineal inicial de 8 unidades en el eje -X y un tiempo de vida de 100 frames para que desaparezcan al poco tiempo.

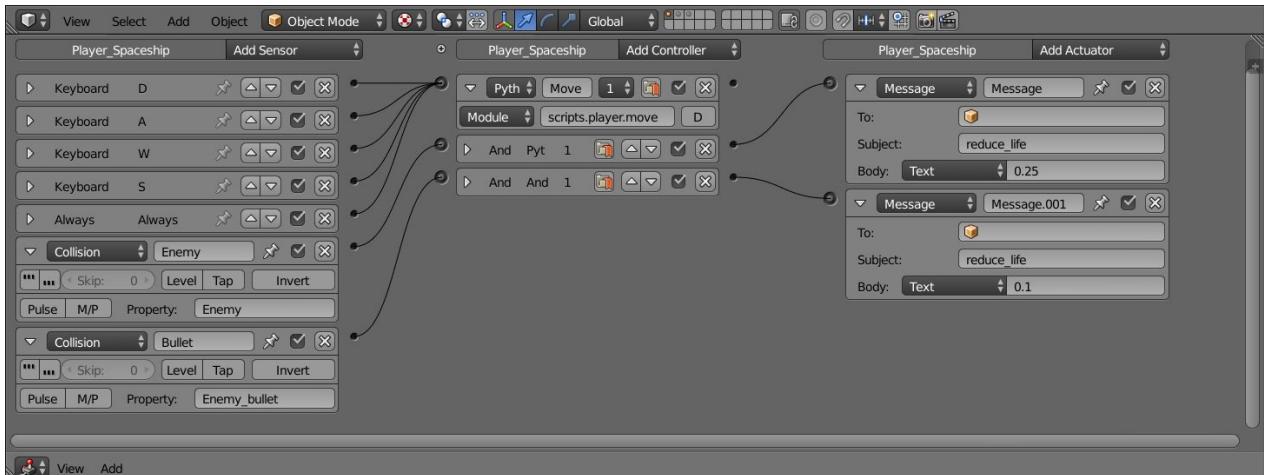


Ahora que los enemigos disparan haremos que las balas desaparezcan al alcanzar al jugador, para lo que se ha añadido un sensor *Collision* con la propiedad *Player* (que solo la tiene el jugador) unido a un actuador *Edit object>End object*.



Aunque inicialmente se pensó hacer que desaparecieran también al impactar con las balas del jugador, es algo que pasa bastante poco y que genera una situación curiosa en la que ambas balas cambian su trayectoria, por lo que se ha decidido dejarlo.

Ahora haremos que el jugador pierda vida al impactar contra las balas, para lo que se reutilizará el mensaje *reduce\_life* creado anteriormente, solo que reduciendo menos vida. Se han cambiado los valores para que colisionar con una nave enemiga quite el 25% de la vida y un disparo enemigo el 10%.



Con esto quedaría terminado la parte de los disparos enemigos.

### 3.8. Jefe final

Como este juego es un *endless*, es decir, solo termina cuando el jugador muere, aparecerá un jefe cada vez que el jugador consiga 300 puntos, y mientras que el jefe esté vivo **no** se generarán naves enemigas. Por tanto, se comprobará la cantidad de puntos cada vez que el jugador destruya una nave y, si es múltiplo de 300, se dejarán de generar enemigos y el jefe aparecerá.

Se ha creado el archivo *boss.py* que contiene toda la lógica de los jefes; crearlos, moverlos, reducir su vida, destruirlo.... .

Empezaremos generando un jefe cuando el jugador consiga una cantidad de puntos múltiplo de la cantidad de puntos necesarios para un jefe. El archivo *game.py* queda de la siguiente manera.

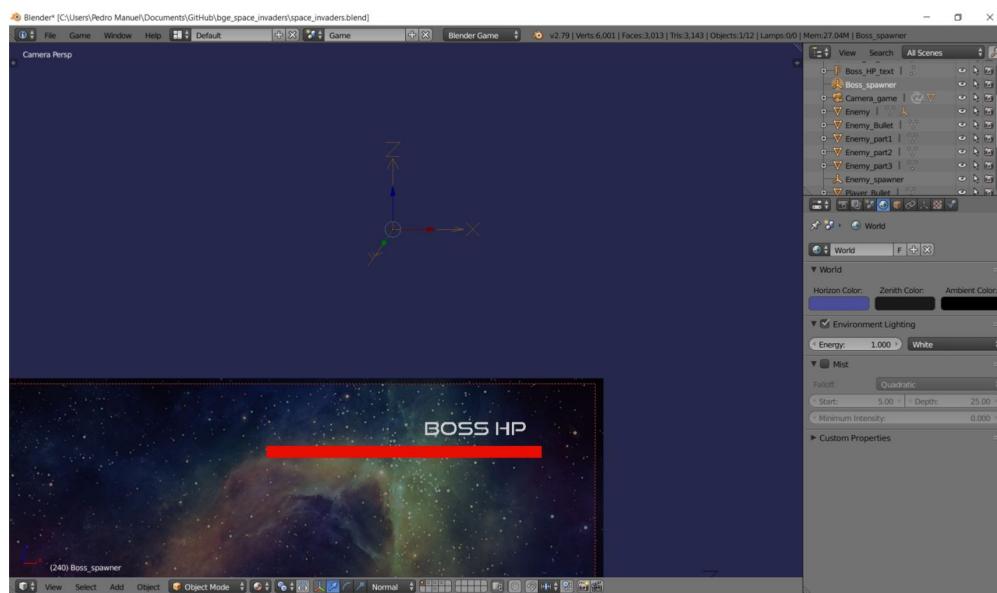
```
from . import boss

POINTS_TO_BOSS = 300

def add_points(controller):
    [...]
    if current_points % POINTS_TO_BOSS == 0:
        scene = bge.logic.getCurrentScene()
        boss.spawn(scene)
```

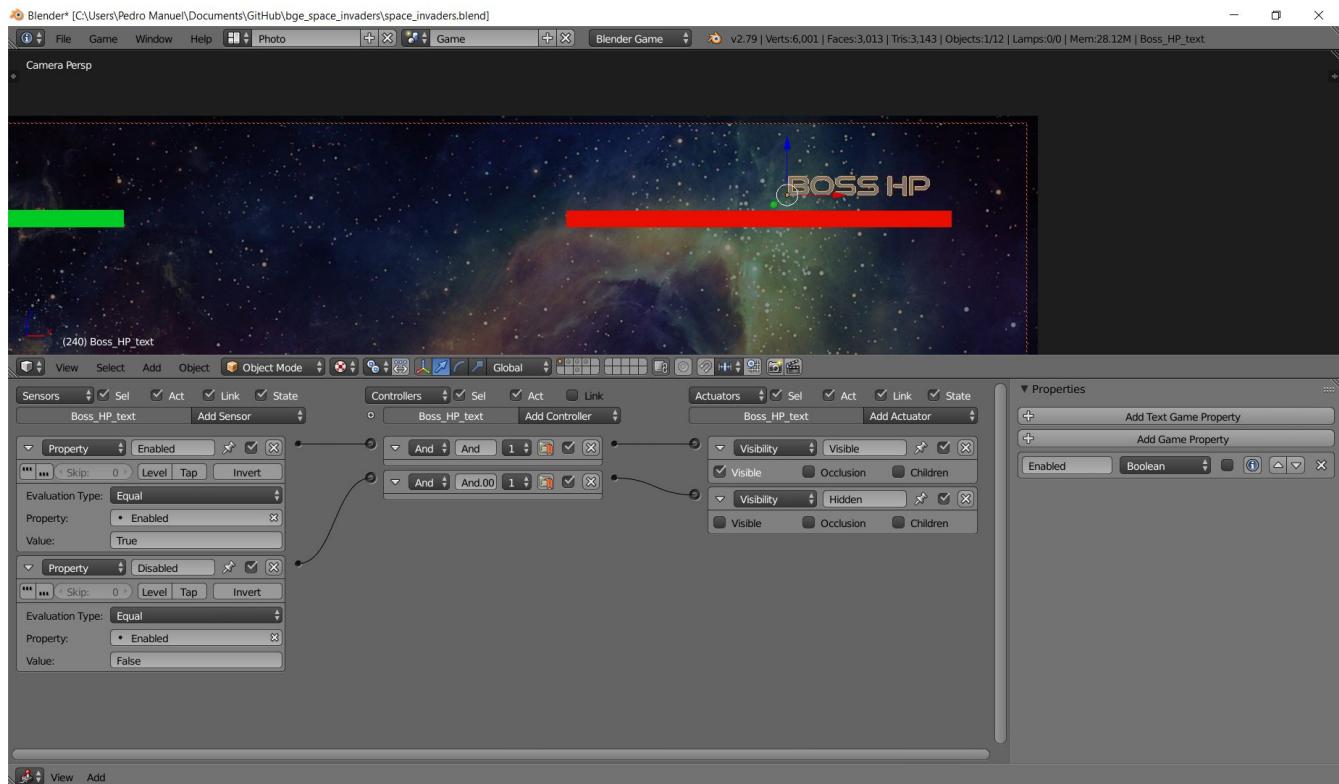
Ahora pasaremos a generar el jefe. Se han añadido 3 elementos nuevos en el archivo Blender.

- Un *empty* llamado *Boss\_spawner* en la parte superior de la pantalla, que será el punto desde el que aparezca el jefe.
- Una barra de vida roja, que será la del jefe, y que aparecerá y desaparecerá cuando el jefe se genere y se derrote, respectivamente.
- Encima de la barra de vida hay un texto con *BOSS HP* escrito para que el usuario sepa qué es la barra roja.



Para hacer aparecer y desaparecer la barra de vida y el texto programáticamente, como no puede hacerse de manera nativa, se utilizará una *property* booleana llamada *Enabled* que por defecto es *False*.

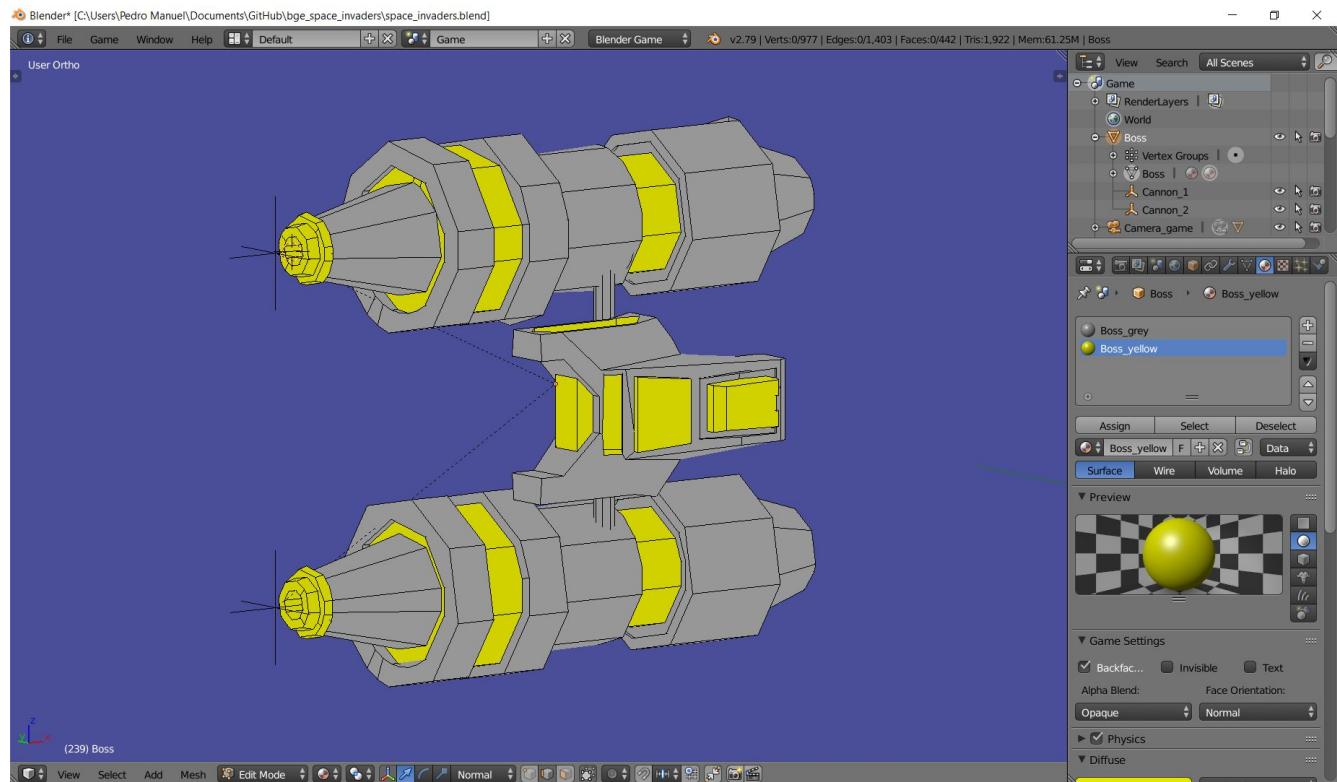
Se ha unido un sensor *Property* a un actuador *Visibility* de manera que cuando la propiedad *Enabled* sea verdadera la visibilidad también y viceversa como se indica en<sup>13</sup>.



13 <https://blender.stackexchange.com/questions/28219/how-do-i-toggle-an-objects-visibility-in-blender-game-engine>

Se ha añadido el mismo esquema de propiedades, sensores y actuadores tanto a la barra de vida como al texto. De este modo, al modificar en código esta propiedad se actualizará la visibilidad de los objetos.

Para el jefe e ha reutilizado y adaptado el modelo de un antiguo proyecto al que se le han añadido dos *empties* que funcionarán como cañones, que dispararán una *Enemy\_bullet* cada 60 frames.



Ahora ya podemos añadir el jefe y mostrar estos objetos. En la función *spawn()* del archivo *boss.py*, que llamábamos al conseguir los puntos necesarios, quedará de la siguiente manera.

```
def spawn(scene):

    if not 'Boss' in scene.objects:
        scene.addObject('Boss', 'Boss_spawner')

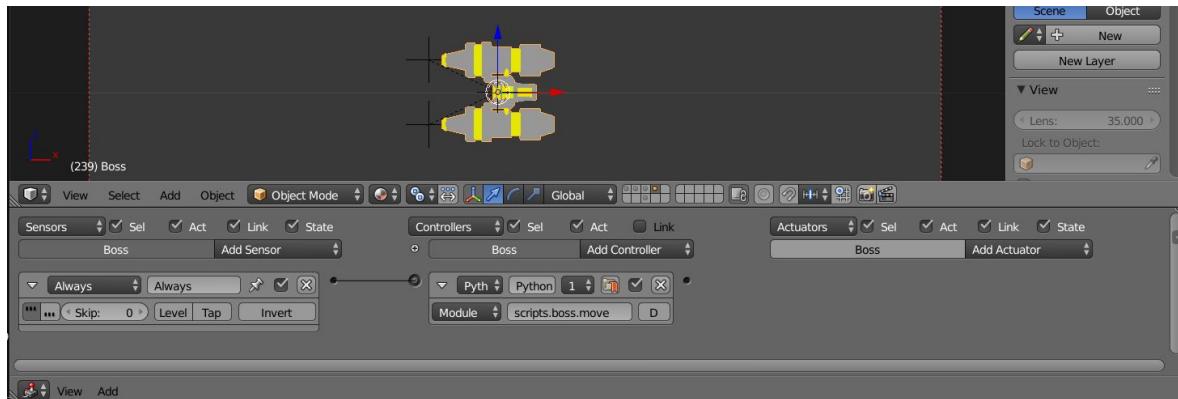
    object_list = scene.objects
    object_list['Boss_HP_bar']['Enabled'] = True
    object_list['Boss_HP_text']['Enabled'] = True

    bge.logic.globalDict['Boss_phase'] = True
```

Se utiliza el diccionario global para anunciar que empieza una fase de jefe, para que esto sea sabido en el archivo *enemy.py* en el que se generan los enemigos para que paren de aparecer.

```
def spawn_enemy(controller):
    if not 'Boss_phase' in bge.logic.globalDict or
        bge.logic.globalDict['Boss_phase'] == False:
        # Generate enemy
        [...]
```

Tras ello, añadiremos al jefe un sensor tipo *Always* con un *skip* de 0 unido al método *move()* del archivo *boss.py*, que será el encargado de manejar la lógica del movimiento del jefe.



```
import bge

MOVEMENT_STEP = 0.025
bge.logic.globalDict['Boss_direction'] = 'DOWN'

def move(controller):
    boss = controller.owner

    if bge.logic.globalDict['Boss_direction'] == 'DOWN':
        if boss.worldPosition.z > -2.2:
            boss.worldPosition.z -= MOVEMENT_STEP

    else:
        bge.logic.globalDict['Boss_direction'] = 'UP'

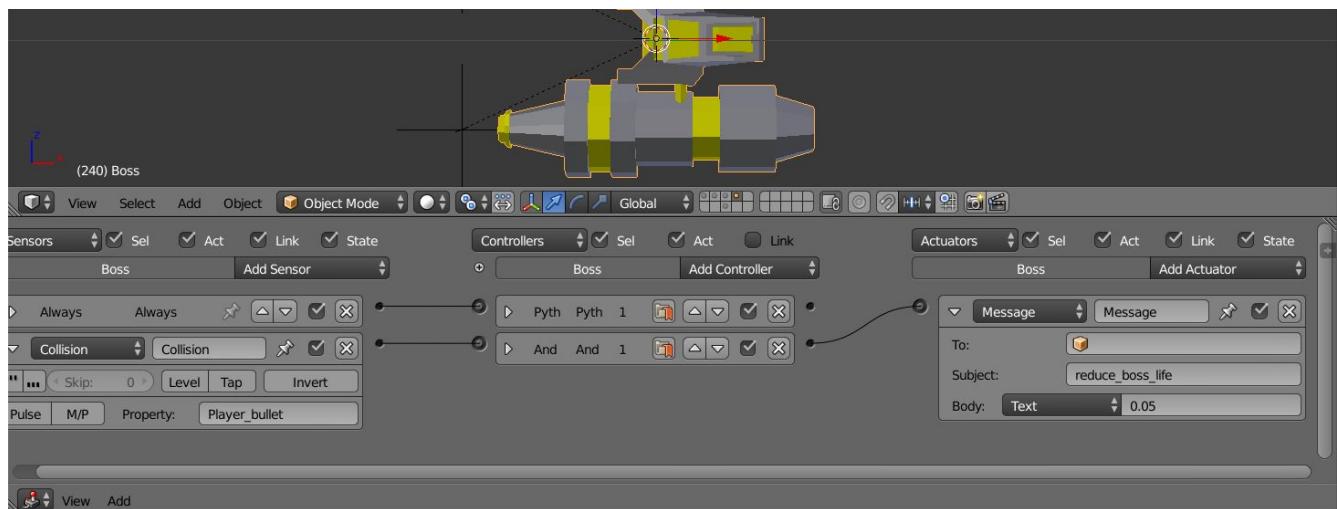
    else:
        if boss.worldPosition.z < 2.2:
            boss.worldPosition.z += MOVEMENT_STEP

    else:
        bge.logic.globalDict['Boss_direction'] = 'DOWN'
```

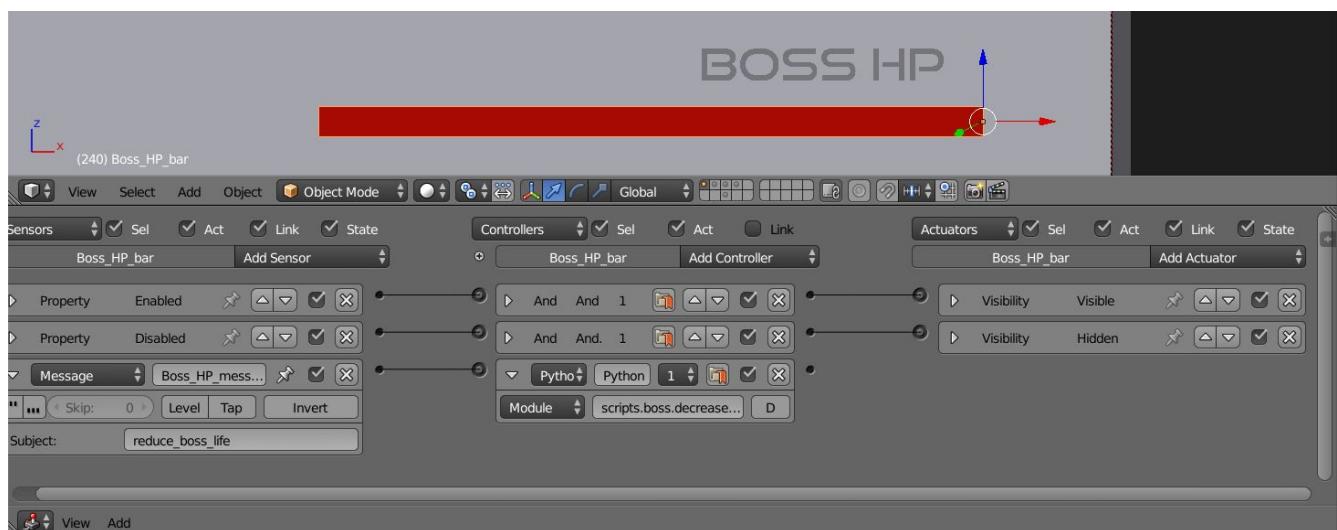
Como puede verse, se ha implementado una pequeña máquina de estados que controla si el jefe está bajando o subiendo comprobando su posición, y lo mueve en consecuencia. Mientras tanto, como sus cañones están activos, seguirá disparando. Se ha usado el diccionario global porque no se ha conseguido hacerlo funcionar con variables globales de Python.

Ahora ya aparece, se mueve y dispara indefinidamente, así que pasaremos a gestionar su vida, que se verá reducida cada vez que impacte con una bala del jugador. Se utilizará un modelo similar al de la vida del jugador, en la que cuando el jefe impacte con una bala del jugador mande un mensaje con el tema *reduce\_boss\_life* a su barra de vida para que este se reduzca un 5% y ésta delegue en un script de Python para actualizarse.

La lógica del jefe.



La lógica de la barra de vida.



Y el método *decrease\_life()* del archivo *boss.py*.

```
def decrease_life(controller):
    boss_health_bar = controller.owner

    message = controller.sensors["Boss_HP_message"]

    if len(message.bodies) > 0:
        damage = float(message.bodies[0])
        x_scale = float(boss_health_bar.localScale.x)
        new_x_scale = x_scale - damage

        if new_x_scale > 0:
            boss_health_bar.localScale.x = new_x_scale
        else:
            boss_health_bar.localScale.x = 0
            destroy(controller)
```

Como vemos, cuando la barra se reduce a 0 se llama al método *destroy()* pasándole el controlador. Este método será el encargado de ocultar el texto y la barra de vida y de resetear su escala para que vuelva a estar completa para el siguiente jefe. Además, eliminará al jefe y establecerá el estado del juego como no jefe para que se vuelvan a generar enemigos normales.

```
def destroy(controller):
    scene = bge.logic.getCurrentScene()

    object_list = scene.objects

    boss_hp_bar = object_list['Boss_HP_bar']
    boss_hp_bar['Enabled'] = False
    boss_hp_bar.localScale.x = 1

    object_list['Boss_HP_text']['Enabled'] = False

    object_list['Boss'].endObject()

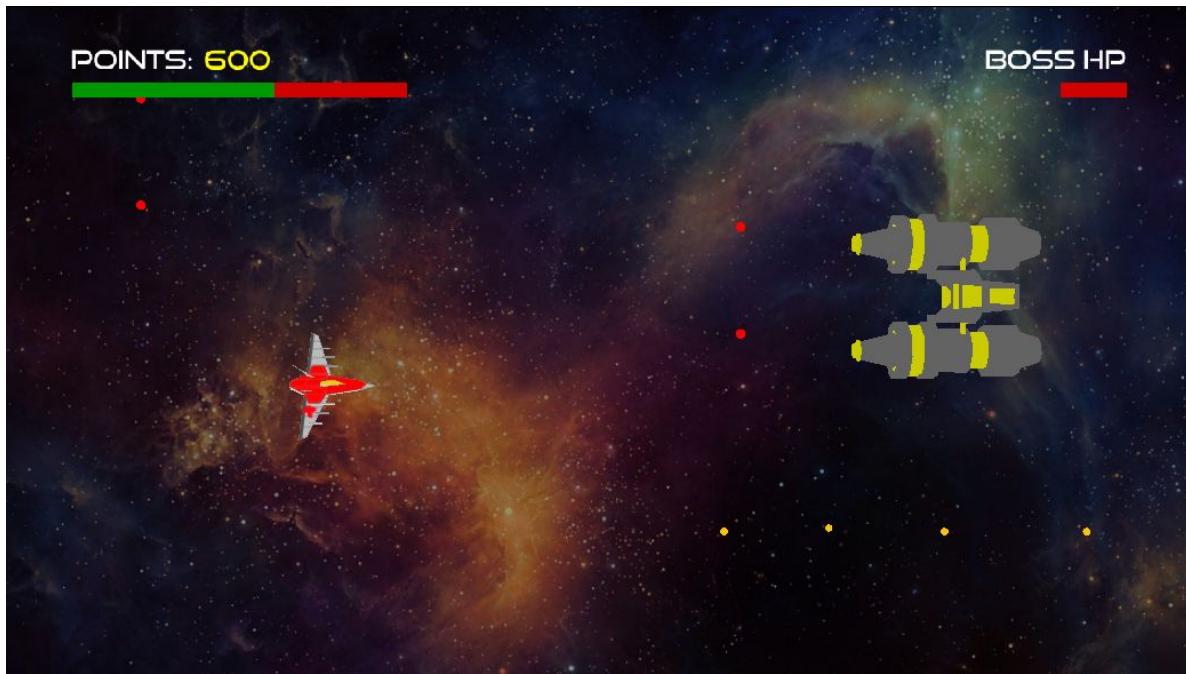
    bge.logic.globalDict['Boss_phase'] = False
```

Por último, se actualizará la puntuación del jugador añadiéndole 50 puntos, 5 veces la puntuación de un enemigo normal. Como ya tenemos un sensor de mensajes en el texto con la puntuación, basta usar la función *sendMessage()*<sup>14</sup> con el tema *enemy\_destroyed* y la puntuación para actualizarla. Por tanto, al final del método *destroy()* se ha incluido la siguiente línea.

```
bge.logic.sendMessage("enemy_destroyed", '50')
```

14 [https://docs.blender.org/api/blender\\_python\\_api\\_current/bge.logic.html#bge.logic.sendMessage](https://docs.blender.org/api/blender_python_api_current/bge.logic.html#bge.logic.sendMessage)

Con esto ya hemos terminado al jefe, que aparecerá periódicamente.

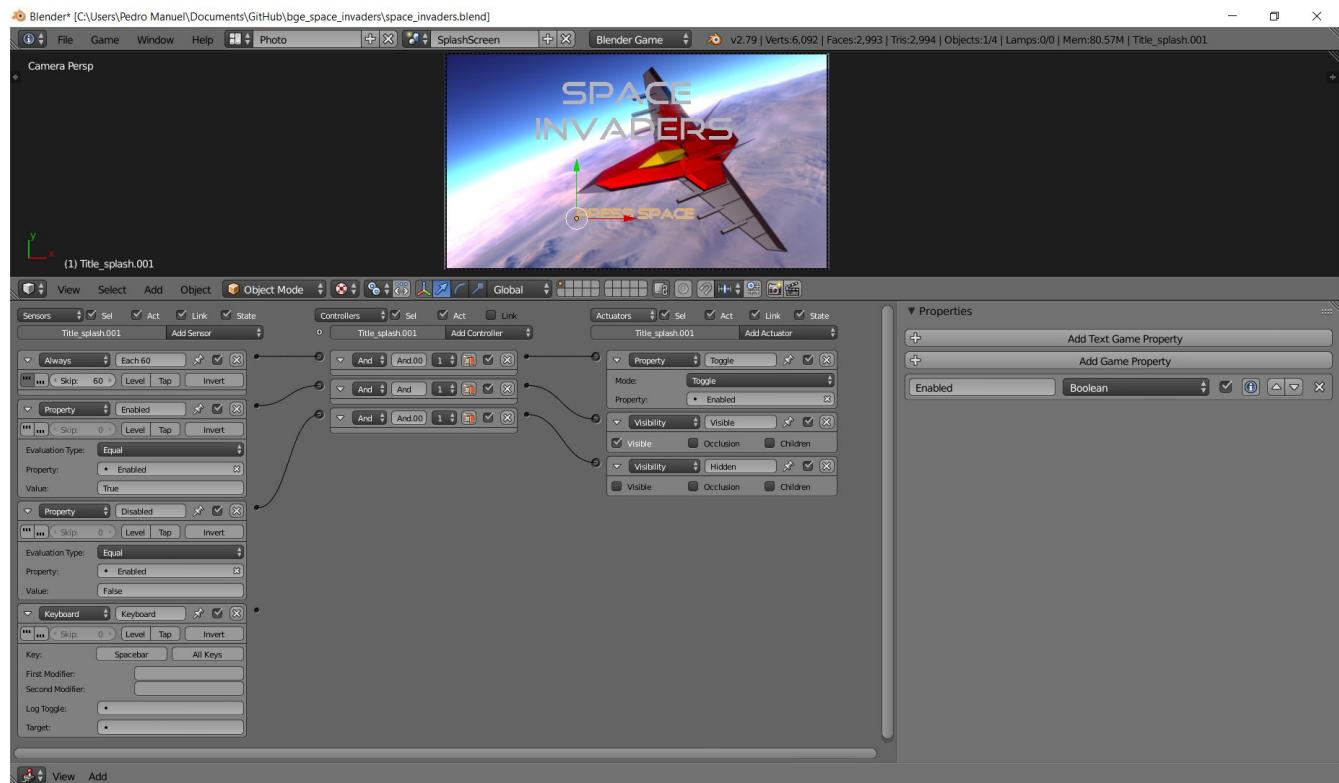


### 3.8. *Splash screen* y créditos

El juego en sí ya estaría terminado, aunque en un intento por mejorarlo se han añadido pequeños retoques e incrementos en la funcioanlidad.

Para dotar de más personalidad al juego se ha incluido una *splash screen* antes de poder acceder al menú principal, para lo que se ha añadido otra escena y un plano con uno de los renders hechos para la primera entrega de teoría que hace las veces de fondo.

Además, para hacer que el texto **PRESS SPACE** parpadee se ha utilizado una metodología parecida a la de ocultar la barra de vida del enemigo; se ha relacionado la visibilidad del texto con una propiedad de tipo *boolean* y se ha añadido un sensor *Always* que se activa cada segundo e invierte esta variable. Así, un segundo será visible y el siguiente no.



Para pasar al menú principal simplemente se ha añadido un sensor *Keyboard* a la tecla *espacio* que llama a un script de Python que llama al método *load\_menu()* del archivo *game.py* y carga la escena *Menu*.

```
def load_menu(controller):
    bge.logic.getCurrentScene().replace('Menu')
```

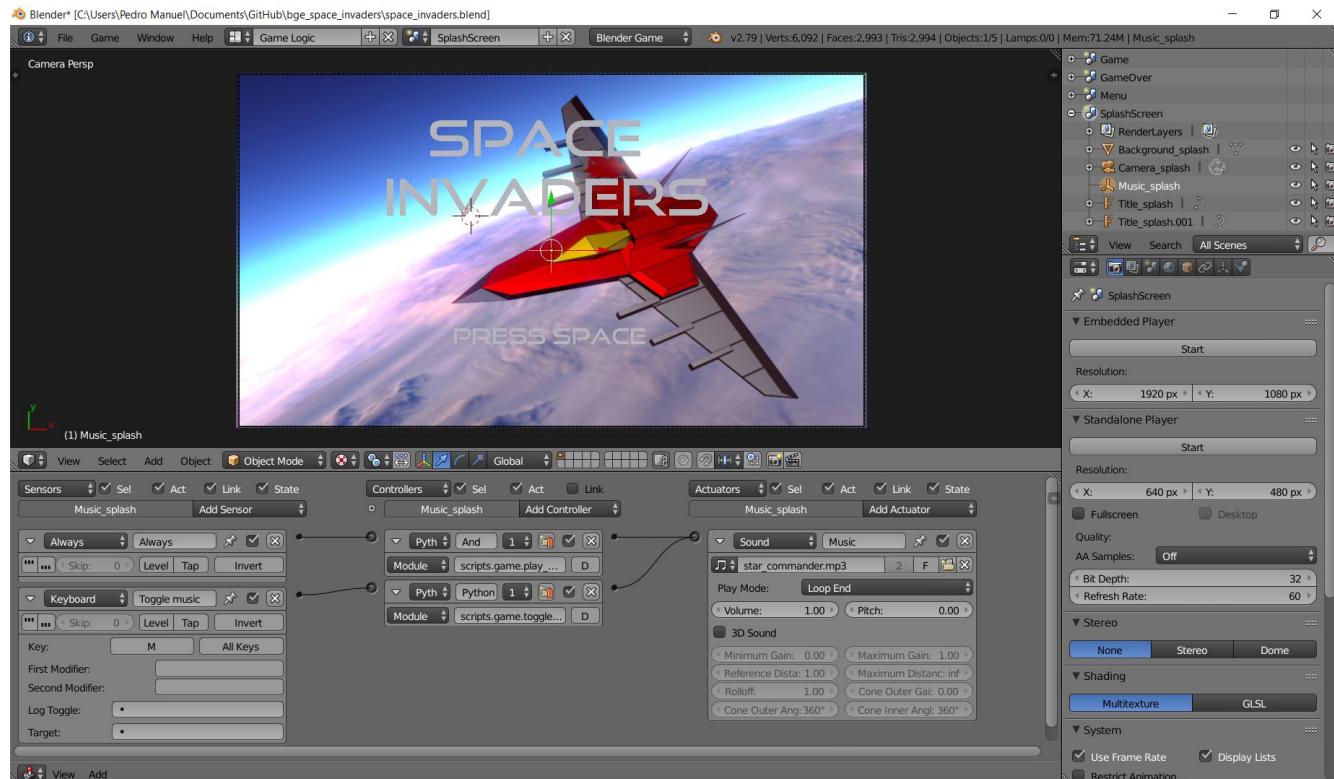
Además, se ha rehecho la pantalla del menú para eliminar el título duplicado y añadir créditos.



### 3.9. Sonidos

Aunque el juego pueda estar terminado es muy plano y no hay mucho *feedback* entre lo que el usuario hace y el juego le muestra, así que para arreglar esto vamos a añadir efectos de sonido.

Lo primero será añadir una canción que suene en bucle a lo largo del juego, para lo que se ha seleccionado el archivo de audio *star\_commander.mp3*. Se ha añadido un sensor *Always* unido al script Python *game.play\_music()* y a un actuador con la canción para que se reproduzca al empezar. Además, para que el usuario pueda pausarla y reproducirla se ha añadido un sensor *Keyboard* a la letra *M*.



```
def play_music(controller):
    if bge.logic.globalDict['Music'] == True:
        music = bge.logic.getCurrentController().actuators['Music']
        music.startSound()
```

```

def toggle_music(controller):
    music = bge.logic.getCurrentController().actuators['Music']

    bge.logic.globalDict['Music'] = not bge.logic.globalDict['Music']

    if bge.logic.globalDict['Music']:
        music.startSound()

    else:
        music.stopSound()

```

Los nodos lógicos han sido aplicados a un *empty* que se ha copiado a la escena *Splash screen*, a la del menú y a la del juego. De esta manera, como se usa el diccionario global el estado de la música se guardará a través de las escenas.

En total se han añadido los siguientes sonidos, cuyos archivos pueden encontrarse en el directorio *music/sounds*. Como es un proceso prácticamente idéntico al de la música de fondo se obviará la explicación de cómo se ha hecho cada uno.

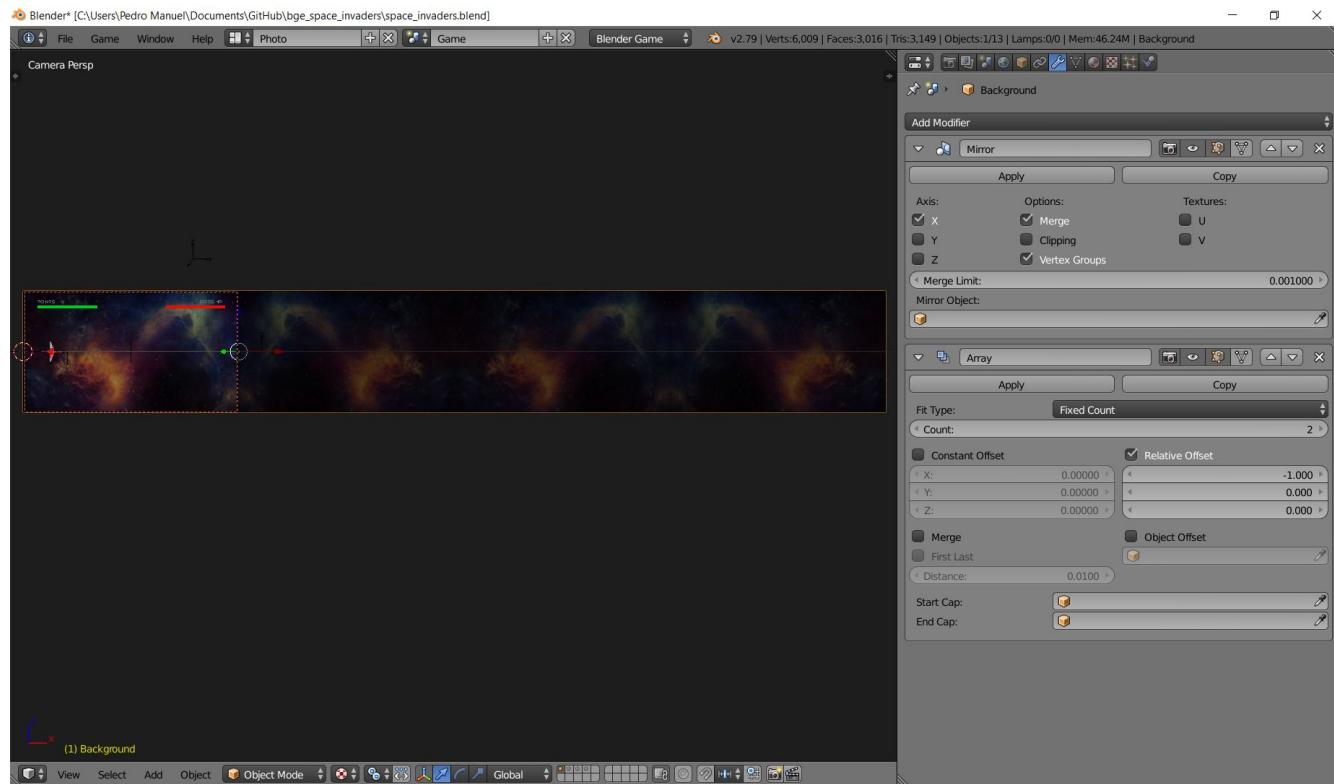
- Explosión de las naves enemigas al ser destruidas.
- Sonido de perder al entrar en la escena *GameOver*.
- Rugido mecánico al aparecer un jefe.
- Gruñido cuando el jugador recibe un disparo o choca con un enemigo.
- Disparo cuando el jugador lanza una bolita.
- Sonido de victoria al destruir a un jefe.

### 3.11. Fondo parallax

Como último añadido, como el fondo estático podía resultar un poco repetitivo, se ha animado para que se mueva. No es exactamente un parallax, ya que para este efecto se necesitarían varias capas, pero el proceso para implementarlo es prácticamente el mismo.

Aunque hay varias maneras de hacerlo, como aún no se ha trabajado con animaciones en este juego es lo que se ha utilizado para el fondo.

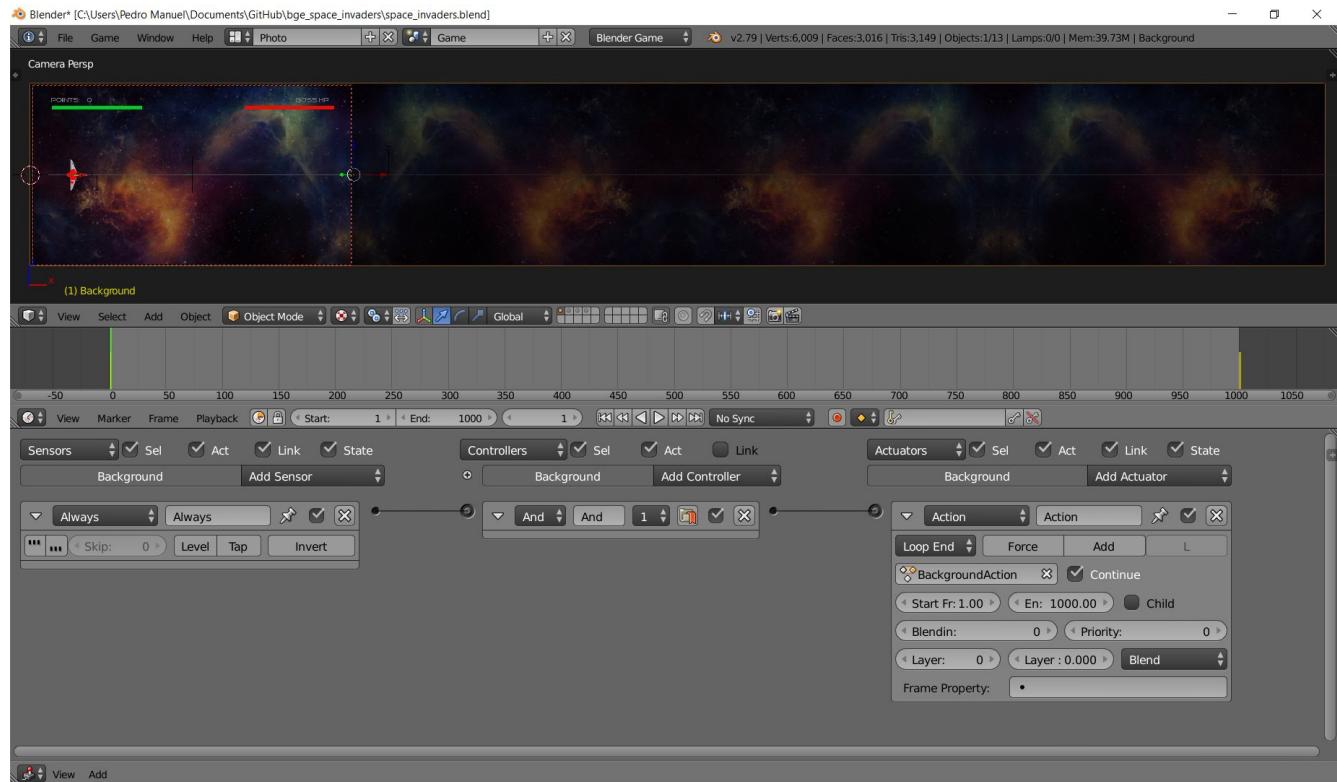
Como el fondo se moverá hacia la izquierda y la imagen que estamos usando no es *seamless* la haremos a mano. Lo primero que se ha hecho es colocar el cursor sobre el vértice derecho de la imagen del fondo con *Ctrl+S>Cursor to selected* y establecerlo como su origen con *Ctrl+Alt+Shift+C*. A continuación se ha añadido un modificador *Mirror* en el eje X, de manera que el fondo queda duplicado en forma de espejo. Luego se ha añadido un modificador *Array* en el eje -X, con lo que ya podemos mover el fondo.



Ahora se quiere que el movimiento dure 1000 frames, por lo que primero se han insertado dos *keyframe* de posición en el frame 1 y en el frame 1001 con la posición original. Como se quiere que el **movimiento sea constante** se ha seleccionado la **interpolación lineal**.

Ahora, para colocar el fondo en su posición correcta para el frame 1000, se ha situado el cursor 3D en el lado izquierdo de la imagen. A continuación, tras activar el poder seleccionar la geometría de los modificadores se ha seleccionado el lado derecho de la tercera repetición de la imagen y se ha movido hasta coincidir con el cursor 3D y se ha insertado otro *keyframe*, con lo que la animación estaría completa.

Ahora, para reproducirla durante el juego se ha añadido un sensor *Always* en el fondo unido a un actuador *Action>Loop end* con la animación seleccionada y 1 como frame inicial y 1000 como frame final.



Ahora al jugar el fondo se moverá ligeramente y reiniciará su posición cada 1000 frames de manera imperceptible para el jugador.

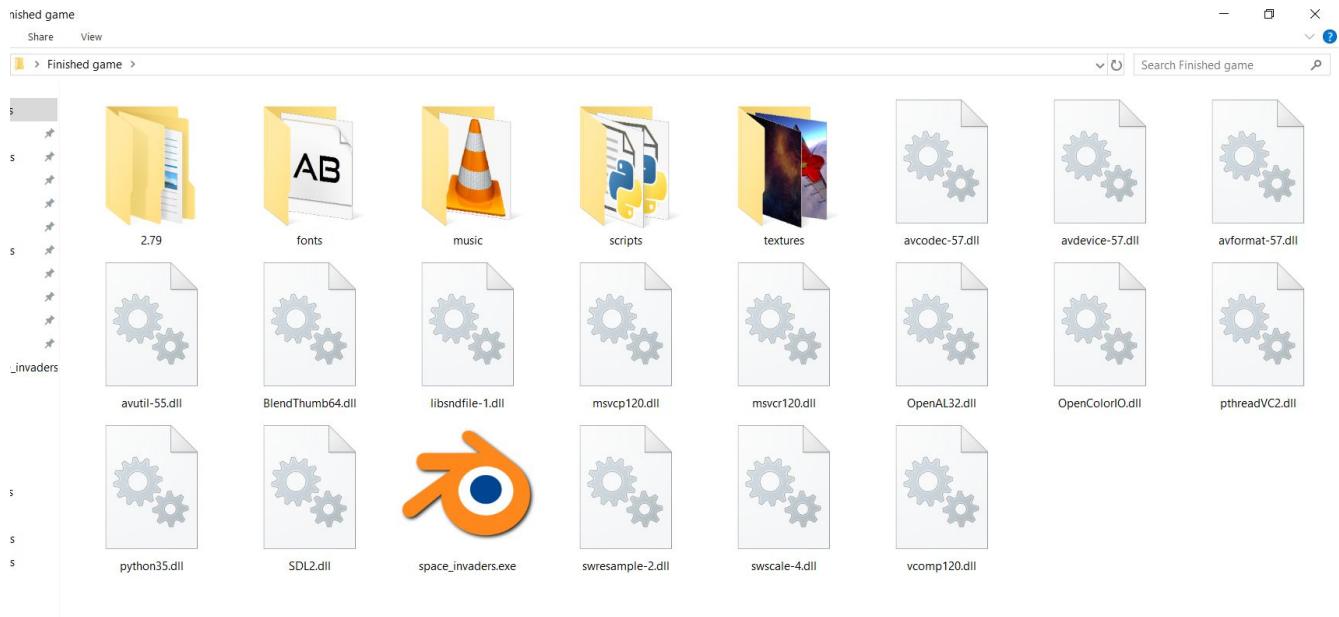
### 3.13. Generar ejecutable

Por último, como todo juego debería poder ser distribuido, generaremos un .exe que los jugadores puedan ejecutar con facilidad.

Lo primero que haremos será activar el addon para exportar a .exe, para lo que iremos a *File>User preferences>Add-ons>Game engine* y activaremos *Save as game engine runtime*.

Ahora ya podemos exportar el juego, para lo que nos situamos en la escena *SplashScreen* y vamos a *File>Export>Save as game engine runtime* y la guardamos en una carpeta.

Si estuviéramos trabajando con todos los assets comprimidos en el archivo .blend también se habrían exportado, pero como no deberemos hacerlo a mano, para lo que copiaremos todas las carpetas necesarias (fuentes, música, texturas, scripts...) del directorio de trabajo a la carpeta donde hemos exportado el juego, que quedaría así.



Ahora ya podemos ejecutar el .exe y jugar.

### **3.14. Trabajo futuro y conclusiones**

Como trabajo futuro, queda pendiente realizar una escena a la que se pueda acceder desde el menú principal con las puntuaciones más altas obtenidas. Inicialmente iba a implementarlo, pero a lo largo del juego ya he trabajado con cambio de escenas, edición de texto usando propiedades, ocultar objetos,... y aunque sería interesante para el jugador no creo que aportara mucho valor extra al juego como proyecto. Además, es más una tarea de programación con Python que de Blender, por lo que finalmente se ha optado por no hacerlo.

En conclusión, la realización de este proyecto me ha servido para aprender aún más de un programa que me encanta y con el que llevo años trabajando, a pesar de no haber tocado nunca el motor de juego que trae consigo. Estoy muy contento de haber realizado finalmente el trabajo que me propuse a principio de curso.