**PROGRAMACIÓN (Por Jaime Calvo Amor y Alexander Matos Teodoro)**

Para poder ofrecer una imagen visual de la disposición de las funciones y todo el script, se ha realizado un diagrama de clases que se mostrará más adelante, a continuación se va a proceder a explicar funcionalidades destacables, el proceso de creación y decisiones de diseño.

**Menús y configuración**

Para controlar el juego, tenemos 3 html básicos, el Menú (MenuCoreDefense.html):

-Se encarga de configurar el juego (audio e idioma), además de detectar si hay que cargar la versión móvil

-Tiene acceso a las instrucciones de cómo se juega

-Hay un apartado de contacto para saber quién ha desarrollado el videojuego web

-Finalmente hay dos opciones de juego, jugar solo o en multijugador (2 personas), estos dos botones llevan a un menú de selección de dificultad, y al seleccionarla, se iniciar el juego

Este menú se encargará de configurar el juego inicialmente. Al darle a una dificultad, llamará a otro html, que cargara Core Defense.html o Core Defense Movil.html dependiendo de si ha detectado que es necesaria la versión móvil.

En cuanto a Core Defense.html y Core Defense Movil.html

Estas páginas, lo primero que hacen es una vez cargado el body llamaran a initGame, que configurará las variables de canvas, su contexto, obtener la dificultad, idioma, multijugador y sonido establecidos, iniciar el mundo (que usara todas las demás clases y objetos) y el mapa de teclas pulsadas que usara el mundo. Además, muestra las vidas del segundo jugador si está habilitado. Finalmente obtiene el archivo json especifico dependiendo de la dificultad elegida y se lo pasa al mundo en su función initMundo.

Tienen funciones para empezar el juego, pararlo, y volver al menú.

La diferencia de la versión móvil, es básicamente que tiene botones específicos (en diferentes canvas) para pulsar y mover los jugadores, que estos al ser pulsados llaman a la función moverJugadorTouch y lo que hace es activar en el mapa la tecla que está siendo pulsada.

**Mundo: Creando el motor de renderizado y el escenario**

Con el propósito de juntar toda la acción y los objetos en escena, se procede a crear la clase mundo. El primer paso para darle forma al juego es crear el motor de renderizado con los objetos del escenario, tiene que tener:

* Casillas: Donde se contienen los bloques del escenario y se indica si los personajes o enemigos pueden moverse por allí.
* Jugadores: En continua comunicación con la clase mundo para poder moverse, renderizarse, y llevar a cabo toda la lógica relacionada.
* Enemigos: Parecidos a los jugadores en funcionalidad y controlados por IA (inteligencia artificial)
* Balas: Lanzadas por los enemigos y los jugadores, únicamente tendrán una posición, una velocidad y una dirección.

La clase mundo es la “central” del juego, es a la única que se llama desde los html, y esta se encarga de llamar a todo lo necesario en su función initMundo.

Esta función básicamente lo primero que hace configurar sus variables y preparar las imágenes que se van a usar en los sprites de enemigos, jugadores y balas. Inicializara también los controladores de las balas de jugadores y enemigos respectivamente.

Escoge aleatoriamente una posición para el jugador o jugadores y los posiciona, además almacena estas posiciones en su variable para usarlas al respawnear jugadores.

Genera los spawns de enemigos en sus posiciones, que son siempre las mismas en todos los mapas.

Ahora generará la board (el array de casillas del mapa), lo que hace es recorrer los datos del json y va metiéndolos por filas y columnas con su respectivo tile o sprite. Para la board es necesario expandirla, ya que los mapas son de 18x18 y el canvas es de 720x720, se expande al tamaño del canvas copiando datos para que, por ejemplo, la casilla 0,0 de la board inicial (es decir la del json), se quedara desde el 0,0 hasta el 40,40 (que es cellSize, el tamaño de una celda del juego), entonces asi la board queda como el canvas, para poder calcular bien las colisiones de todos los objetos entre sí.

**Renderizando y animando:**

Para visualizar los elementos del escenario la clase mundo tiene una función de pintado, que se llama en cada iteración, de esta manera se obtienen todos los sprites que tiene que imprimir en pantalla y los coloca donde hacen falta.

Respecto a las animaciones, la solución que se ha encontrado por razones de optimización, es: Cada objeto animado (jugadores y enemigos) contienen sus secuencias de animaciones, cuando la función de pintado les pida el sprite a pintar, el objeto pasa el siguiente sprite de la secuencia. De esta manera las animaciones tienen como unidad la velocidad de fps que elijamos en el bucle principal del juego. Si queremos ralentizar una animación, es fácilmente controlable. Un ejemplo es el troyano, si queremos hacer que tenga su animación la mitad de rápido, sólo es decir que cambie el índice del sprite cada dos llamadas.

**Implementando la funcionalidad**

Una vez teníamos elementos estáticos y podíamos cargar los escenarios con un JSON, había que crear la lógica de juego para que haya jugadores que se muevan, disparen, mueran, etc.

Todas las acciones de disparo se llevan a cabo por un controlador que gestiona todas las balas en juego, la razón para estructurarlo de esta manera es para poder tener un control sobre dichas balas y llevar a cabo los cálculos de dirección y colisión sobre el conjunto. De manera que además podemos controlar en qué parte del bucle va a tener lugar.

Las funcionalidades para mover los jugadores se encuentran en la clase mundo, que se encarga de decir por donde puede pasar y por donde no.

Los enemigos toman todas sus decisiones mediante una IA. En lo respecto a la inteligencia artificial de los enemigos, tuvimos un problema de diseño inicial:

* Los objetivos de la máquina son: llegar al núcleo, navegando por el escenario disparando y rompiendo las defensas. El enemigo tiene que romper el escenario, no debe abrirse paso por el camino más corto al núcleo, ya que todos irían por el mismo camino desde el spawn, sino que tiene que ir ocupando el escenario y acercarse poco a poco.
* Algoritmos de búsqueda como A\* ofrecían el camino más corto, sin embargo. En un escenario en continuo cambio se necesitaría estar calculando continuamente las rutas, y sería viable para pocos enemigos, pero si tuviéramos que calcular en cada iteración la ruta por cada 5, 10 o 20 enemigos, esta solución no sería tan adecuada.

Con la intención de optimizar los cálculos y conseguir una heurística adecuada, se ha partido la IA en funciones por cada comportamiento:

- Navegar: El enemigo se tiene que mover por el escenario, y cuando haya un bloque en su dirección cambia de dirección. El bucle principal en mundo le hará llamadas para moverse, una función en el virus se encarga de actualizar la posición.

-Direccionarse: Tarde o temprano tiene que apuntar hacia el núcleo y disparar para intentar abrirse paso en la dirección que la máquina piensa que es la buena. Calcular la dirección en la que debe moverse, ya que no hay movimientos diagonales, siempre será a elegir entre Arriba/abajo o izquierda/derecha. Saber cual de las dos es la buena es tan sencillo como hacer una resta entre la posición del enemigo y el núcleo. De esta manera quitamos la complejidad de una búsqueda A\* en realizar sólo dos restas por cada enemigo. Para simular que el enemigo decide girarse y disparar, dicho comportamiento se llama desde la función de disparo, justo antes de disparar.

Los resultados fueron que los enemigos iban arrasando el área donde aparecen, yendo en dirección al núcleo.

**Generando los enemigos:**

Los enemigos tienen que generarse, y se generan desde un spawn. Dicho Spawn establece cómo se van a crear los enemigos y dónde, tenemos 4 en el escenario y cada uno gestiona sus propios monstruos: ejecuta su destrucción, los crea con los valores indicados en función de la dificultad, o actúa de auxiliar cuando sea necesario.

Para definir la frecuencia de aparición se usan algoritmos de lógica difusa (Ejemplo: el virus tiene que aparecer en un rango entre 4 y 7 segundos).

La clase Spawner tiene una función que actúa de auxiliar para el troyano, dicho enemigo cuando acaba su tiempo de vida y ningún jugador lo ha matado, explota y suelta 3 enemigos. En lugar de implementar la lógica de un Spawner en la clase Troyano, el troyano hace 3 llamadas a su Spawner y le pide que cree un en su posición en cada llamada.