Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

VIDEOJUEGO RPG (ROLE-PLAYING GAME) BASADO EN ALGORITMOS EVOLUTIVOS

Alumno: José Javier Pérez Cruz

Tutor: Víctor Manuel Rivas Santos
Juan Roberto Jiménez Pérez

Dpto: Departamento de Informática

Jaén, junio de 2018

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Departamento de Informática

Don Víctor Manuel Rivas Santos y Don Juan Roberto Jiménez Pérez , tutores del Proyecto Fin de Carrera titulado “Videojuego RPG basado en algoritmos evolutivos”, que presenta Jose Javier Pérez Cruz, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2018

El alumno:

JOSE JAVIER PÉREZ CRUZ

Los tutores:

RIVAS SANTOS VICTOR MANUEL - 29798183G

Firmado digitalmente por RIVAS SANTOS VICTOR MANUEL - 29798183G
Fecha: 2018.06.26 00:02:42 +02'00'

VÍCTOR MANUEL RIVAS SANTOS
JUAN ROBERTO JIMÉNEZ PÉREZ

AGRADECIMIENTOS

A lo largo de mi vida, siempre he creído que era agradecido con lo bueno y lo no tan bueno que me ha ido ocurriendo. Y, aun así, situaciones como esta me hacen sentir que, en realidad, siempre fui un ingrato. Así, me gustaría aprovechar este comienzo para reconocerle la culpa a algunas de las personas que me ayudaron a llegar y más tarde a mantenerme, sabiendo que esta no será la última vez que tenga que hacerlo a lo largo de mi vida.

Quiero agradecer a mis padres el ser los principales artífices de que hoy esté aquí, escribiendo esto. Incluso en las malas, vuestra fe ciega en mi ha sido un motivo más para dar siempre lo mejor.

Quiero dar las gracias a mis hermanos por darle verdadero significado a esa palabra y por esa ayuda que nunca me negaron.

Gracias a Marta, mi pareja, por ser mi principal apoyo desde que nos conocimos. Gracias por creer en mi incluso cuando ni yo mismo podía.

También me gustaría reconocer a mis amigos, con mención especial a mis tortullas. Con vosotros todo es más fácil.

Asimismo, me gustaría valorar la labor y la profesionalidad de mis dos tutores, Roberto y Víctor. Gracias por aguantarme a mi y a mis correos de madrugada los domingos.

Por último, y más por orgullo que por soberbia, me gustaría agradecerme a mí mismo el esfuerzo realizado para llegar hasta aquí.

Al final del día podemos aguantar mucho más de lo que pensamos que podemos”

- *Frida Kahlo*

ÍNDICE

1.- INTRODUCCIÓN AL PROYECTO

1.1.- Definición del proyecto

A lo largo de esta memoria, vamos a describir el proceso de análisis, diseño y desarrollo de un videojuego. En él haremos uso de algunos conceptos de computación evolutiva, realizando en el proceso un estudio sobre la aplicación óptima de estos para la resolución de problemas derivados.

El videojuego a desarrollar es un RPG con un sistema de combate por turnos peculiar. El jugador tomará el rol de entrenador o conciencia de un personaje en el juego para indicarle la estrategia que él considere mejor para ganar el combate en las condiciones en las que se encuentre al iniciar su turno. Posteriormente, será el personaje virtual el que decidirá, a través de un algoritmo genético, cuál es la mejor solución posible encuadrada dentro de la estrategia que le señalaron.

Por ejemplo, si el jugador señalase que su personaje debe atacar, este buscará realizar acciones que le permitan realizar el máximo daño posible. Mientras que, si, por ejemplo, el jugador indica que debe adoptar una estrategia defensiva, el personaje buscará minimizar los posibles daños que pudiera sufrir en el siguiente turno rival.

El desarrollo del videojuego se realizará utilizando el motor de videojuegos Unity, con lenguaje de programación C# y se seguiré un enfoque ágil para el correcto desarrollo del proyecto.

1.2.- Motivación

La industria de los videojuegos compone uno de los sectores que más beneficio económico está obteniendo en la actualidad. Grandes empresas invierten ingentes cantidades de dinero para servir al consumidor un producto software diseñado y desarrollado por enormes equipos de profesionales. Mientras tanto, también surgen desarrolladoras independientes, casos en los que, en ocasiones, la ilusión y el tiempo de sus creadores es la única inversión para arrancarlos.

El principal propósito de este trabajo es obtener un videojuego cuya jugabilidad gire en torno al uso de algoritmos evolutivos, concretamente algoritmos genéticos. Se busca aprovechar una de las metaheurísticas más estudiadas en las ciencias computacionales y utilizarla como piedra angular de un programa que debe ejecutarse en tiempo real como es un videojuego. A partir de esto, estudiaremos el impacto de la aplicación de la computación evolutiva en software de este tipo y su influencia sobre el rendimiento de la aplicación y el dinamismo de la misma.

Se pretende demostrar que, en proyectos independientes, en este caso unipersonal, también es posible aprovecharse de las ventajas de complejas metaheurísticas acompañadas de un enfoque estructurado y profesional.

2.- ALGORITMOS GENÉTICOS

Vamos a describir las particularidades de la resolución de problemas usando computación evolutiva. Profundizaremos sobre la teoría de **Algoritmos Genéticos (AG)**, que será el paradigma de estrategia evolutiva sobre el que centraremos la Inteligencia artificial del personaje.

2.1.- Inteligencia Artificial

La inteligencia Artificial es la rama de las ciencias computacionales encargada de estudiar modelos de cómputo capaces de simular procesos de inteligencia humana por parte de máquinas. Estos procesos suelen incluir aprendizaje, razonamiento y autocorrección.

Los elementos actuadores en cualquier interacción de este ámbito son el **medio** y el **agente**. El agente es una entidad capaz de percibir su entorno usando sensores, procesar tales percepciones y actuar para producir cambios sobre este de manera racional mediante actuadores. El medio, por otra parte, es todo aquello que rodea al agente, incluyendo otros agentes externos. En conclusión, el medio se considera el problema a resolver por el agente, que es quien aporta una solución racional.

El **aprendizaje** dentro de la inteligencia artificial es un apartado muy importante. En muchas ocasiones el agente se encontrará en entornos desconocidos o con incertidumbre. En estos casos, lo ideal es conseguir un agente que sea capaz de utilizar sus percepciones y los resultados cosechados para conseguir mejores soluciones en el futuro. Un buen mecanismo de aprendizaje es capaz de mejorar el rendimiento de los agentes a medio y largo plazo.

El **razonamiento** consiste en, una vez obtenidas todas las percepciones externas, ser capaz de traducir los datos obtenidos en algún tipo de información al que podamos aplicar una serie de operadores y reglas que nos permitan obtener conclusiones, y por ende soluciones, racionales.

Otro aspecto clave en este apartado es la **representación del conocimiento,** ya que de la representación que escojamos puede depender mucho la calidad de las soluciones aportadas y del agente en general. La representación del conocimiento

significa elegir una serie de convenciones para describir objetos, relaciones y procesos en el mundo del agente.

Para diferenciar los diversos tipos de Inteligencia Artificial que pueden darse, Stuart Russell y Peter Norvig, informáticos con gran reputación en el campo de la IA, señalan estas cuatro categorías **[2]**:

- **Sistemas que piensan como humanos**
    - Emulación completa del comportamiento humano. Automatizan actividades vinculadas al comportamiento humano como el aprendizaje, la toma de decisiones o la resolución de problemas.
- **Sistemas que piensan racionalmente**
    - Utilizan la lógica para imitar un pensamiento racional. Estudia los cálculos que hacen posible percibir, razonar y actuar consecuentemente.
- **Sistemas que actúan como humanos**
    - Su función es imitar, de la manera más realista posible, el comportamiento humano. La robótica parte como principal valedor de esta parcela. Se trata de estudiar cómo agentes pueden realizar tareas que, por el momento, el ser humano hace mejor.
- **Sistemas que actúan de forma racional**
    - Emulan de forma racional el comportamiento humano. Se basa en el diseño de agentes inteligentes.

Las diversas técnicas que componen la inteligencia artificial tienen aplicaciones muy variadas. Hoy en día, debido a los numerosos avances, pueden ser útiles para aspectos de seguridad, sanidad, educación e incluso aplicaciones más cotidianas como en mucho software que utilizamos en el teléfono móvil o en nuestro ordenador.

La Inteligencia Artificial es sin duda uno de las ramas de la informática que mayor crecimiento ha sufrido en los últimos años. Este crecimiento, y las posibilidades que otorga, lo ha convertido en uno de los campos con mayor demanda de profesionales. Tanto es así, que reputados conocedores del tema y consultoras como Gartner **[5]** han incluido este ámbito entre las mayores tendencias tanto en el presente como en los años venideros.

De cara al sector de los videojuegos, tenemos que tener en cuenta que esta fue una de las primeras aplicaciones prácticas de la inteligencia artificial en software. En nuestro

proyecto, hacemos uso de la Inteligencia Artificial para configurar el comportamiento interno tanto del personaje propio del jugador como el de los enemigos.

2.2.- Metaheurísticas

En ciencias computacionales, el objetivo es encontrar un algoritmo que sea capaz de dar buenas soluciones con un tiempo de ejecución relativamente bueno. En algunos casos, como en los descritos anteriormente de inteligencia artificial, optimizar simultáneamente la calidad de las soluciones y el tiempo de ejecución es imposible.

Una **heurística** es un algoritmo que sacrifica una de las dos variables para tratar de mejorar la otra. Es decir, se limita a encontrar soluciones medianamente buenas en un tiempo de ejecución razonable. Su aplicación es útil cuando encontrar la solución óptima al problema costaría demasiados recursos computacionales y podemos conformarnos con una solución aproximada o no existe solución óptima al problema descrito. La mayoría de los problemas de la IA se resuelven mediante heurísticas por definición.

Una vez conocido esto, podemos definir qué es una metaheurística. Una **metaheurística** es un algoritmo de propósito general con una serie de procedimientos iterativos capaces de guiar a una heurística subordinada para buscar soluciones factibles en dominios en los que la tarea es muy dura o consume demasiados recursos. Es decir, se trata de una heurística que resuelve problemas genéricos a partir de unos datos proporcionados sobre unos procedimientos generales y abstractos **[7]**.

Las metaheurísticas suelen ser utilizadas cuando el problema no tiene una heurística específica que dé una solución satisfactoria sin consumir más recursos de los disponibles. Para problemas con heurísticas ya definidas, la metaheurística suele ser menos eficiente.

Al ser un campo de investigación con numerosos investigadores y muchas publicaciones en revistas, existen gran cantidad de metaheurísticas, clasificadas atendiendo a varios criterios como su fuente de inspiración, o si hacen uso de memoria. Incluso, hay quien ve patrones de diseño como GRASP dentro de la definición de metaheurística **[6]**

A la hora de escoger la metaheurística que vamos a aplicar, tenemos que tener en cuenta que algunas funcionan mejor que otras según el problema. Por tanto, es importante tener claros los criterios de selección de una metaheurística u otra dependiendo del problema, y, una vez elegida, saber adaptarla a este de manera correcta.

2.3.- Metaheurísticas basadas en Poblaciones

Una clasificación muy habitual de las metaheurísticas es saber si se basan en poblaciones o no. A grandes rasgos, una población es un conjunto de datos descritos con la misma representación, aunque dentro de cada metaheurística su función pueda ser diferente.

Las metaheurísticas basadas en poblaciones, también llamadas P-Metaheurísticas, se pueden ver como un proceso de mejora iterativa de una población de soluciones:

- Se **inicializa** la población.
- Se **genera** una nueva población, que se integra en la anterior mediante un proceso de **selección.**
- Se repite el proceso hasta cumplir una condición de parada.

Las fases de generación (inicio de la población) y reemplazo pueden utilizar memoria (almacenar información útil) o no (basarse solo en la población actual).

Un aspecto importante a tener en cuenta dentro de este tipo de algoritmos es conseguir una población inicial lo suficientemente diversificada para no caer en convergencia prematura. Esto es, buscar una población lo suficientemente repartida a lo largo del espacio de búsqueda como para conseguir no caer demasiado pronto en un óptimo local y tener mayores posibilidades de conseguir mejores soluciones.

Por otra parte, también es importante definir correctamente el criterio de parada para acotar la ejecución lo mejor posible. Este criterio puede ser estático (conocido previamente → numero de iteraciones) o adaptativo (no conocido a priori → número máximo de generaciones sin mejora, solución óptima alcanzada...)

2.4.- Algoritmos evolutivos

Para este proyecto hemos escogido trabajar con una P-Metaheurística inspirada en la naturaleza. En concreto, trabajaremos dentro del marco de la computación evolutiva, llamada así por seguir y modelar los principios de Darwin sobre la evolución natural.

Para entender la computación evolutiva es necesario saber cuál es la base que se tomó para su creación. En este caso, nos referimos a los estudios publicados por Charles Darwin en 1859 en el libro titulado *El Origen de las Especies* (*On the Origin of Species*) **[8]**. A lo largo de esta publicación, no exenta de polémica, se detalla el proceso de evolución que propuso Darwin basándose en dos principios:

- Modificaciones en la descendencia.
    - Con el paso de las generaciones, los rasgos de los individuos que mejor se adapten a las condiciones que favorezcan su supervivencia se vuelven más comunes en la población, haciéndola evolucionar.

- Selección natural
    - La naturaleza selecciona las especies mejor adaptadas para sobrevivir a su entorno y reproducirse en él.

A partir de los años 60, varias corrientes de investigación independientes comenzaron a aplicar a la programación estos principios para formar los pilares de lo que hoy conocemos como computación evolutiva. Una de las motivaciones para estudiar este tipo de computación fue imitar el cerebro humano, pues este era el mejor “buscador de soluciones” hasta la fecha. Sabiendo que el cerebro fue construido a partir de un proceso evolutivo natural, se trató de imitar este proceso para optimizar las soluciones y el tiempo invertido en encontrarlas.

Dicho esto, podemos asumir que los algoritmos evolutivos son modelos bioinspirados, basados en poblaciones y estocásticos (no deterministas), pues igual que en la naturaleza hay una serie de individuos variados que pueden reproducirse entre ellos y en los que el azar interviene para definir sus características y las de sus descendientes.

Además de los principios ya comentados, la evolución natural se rige por una serie de mecanismos. Aunque aún no son totalmente conocidos, sí que debemos conocer algunas de sus características:

- El proceso de evolución opera sobre los **cromosomas** que componen y codifican las estructuras de vida.
- La selección natural es el enlace entre los cromosomas y la actuación de sus estructuras decodificadas.
- El proceso de **reproducción** es el punto donde la evolución toma actúa.
- **La evolución biológica no tiene memoria.**

Los algoritmos evolutivos son una emulación que simplifica un proceso muy complejo con estructuras muy difíciles de estudiar cómo es la evolución natural. No obstante, aprovecha gran cantidad de términos biológicos ya definidos para nombrar muchos de los elementos que formarán los algoritmos de optimización. La tabla I muestra a continuación algunos ejemplos de términos algorítmicos asemejados a términos evolutivos.

Elemento algorítmico	Metáfora evolutiva
Problema	Entorno
Optimización	Evolución
Individuo	Solución al problema
Objetivo	Fitness / Adaptación
Elemento de la solución	Gen
Valor del elemento	Alelo

Tabla 1: *Comparación entre los elementos algoritmos y las metáforas evolutivas*

También es muy importante conocer los conceptos de Genotipo y Fenotipo y saber diferenciarlos. El **Genotipo** representa la codificación de la solución en forma de cromosoma, sobre él actúan los operadores de variación que se efectúan durante la reproducción entre individuos. El **Fenotipo** representa el valor de la solución. Este valor será el que utilice la función de adaptación para comprobar las habilidades de supervivencia de un individuo en su entorno.

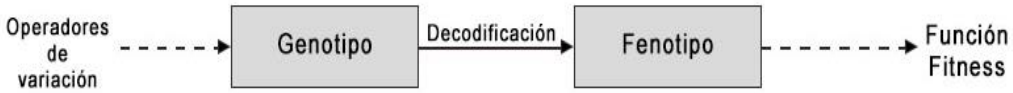


***Ilustración 1**: Correspondencia entre Genotipo y Fenotipo*

Si bien ya se han definido correspondencias entre los términos más comúnmente utilizados, en la ilustración 2 podemos ver un esquema básico de un algoritmo evolutivo. A continuación, explicaremos con más detalle en qué consiste cada uno de los elementos y procesos participantes en la ejecución del algoritmo.

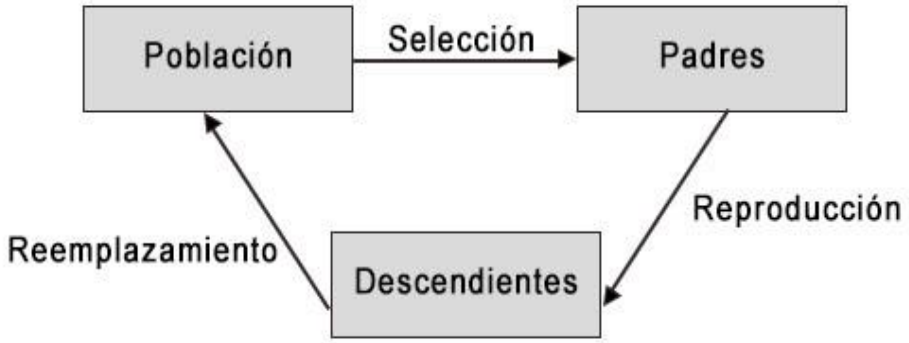


***Ilustración 2:** Esquema básico de un algoritmo evolutivo*

A la hora de realizar la **selección** dentro de la población, es necesario tener en cuenta que, al igual que en la evolución natural, probablemente haya individuos de la población con mayor habilidad para reproducirse que otros. No obstante, para favorecer el concepto de diversificación, todos los individuos dentro de la población deben tener posibilidades para reproducirse.

Durante el proceso de reproducción, existen dos posibilidades para generar descendientes a partir de los padres: **mutación** y **cruce** o recombinación. En la mutación, aplicamos un cambio con aleatoriedad sobre uno o varios genes dentro del cromosoma, configurando un nuevo descendiente que nos da más posibilidades de diversificar el

espacio de búsqueda. En el cruce, combinamos los genes dos padres para generar uno o varios descendientes. Con este método, aseguramos que los “mejores” genes tengan más posibilidades de persistir en el tiempo.

En el reemplazamiento, los descendientes sustituirán a una serie de individuos de la población inicial. Es tarea de diseño escoger entre las diversas posibilidades de reemplazo. En general, las técnicas más utilizadas son reemplazar aquellos individuos con el peor fitness, reemplazar a los padres o reemplazar aleatoriamente entre el conjunto global de individuos.

2.4.1.- Tipos de algoritmos evolutivos

Hasta ahora hemos visto las características generalizadas de la computación evolutiva en conjunto. Se trata de conseguir un algoritmo basado en una población de soluciones, con un valor fitness, que evolucionen en el tiempo para explorar el espacio de búsqueda hacia mejores soluciones.

A partir de ese esquema general, podemos señalar cuatro paradigmas principales **[9]** con los que clasificar los algoritmos evolutivos **[Ilustración 3]**

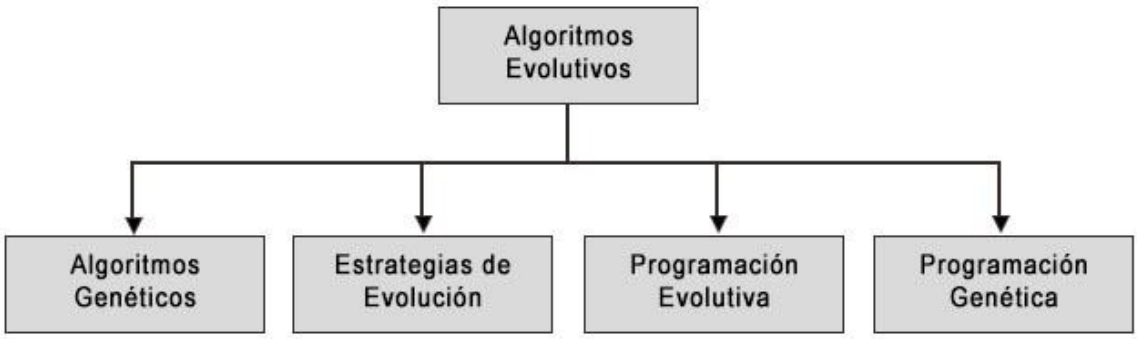


***Ilustración 3**: Paradigmas de la computación evolutiva*

2.4.1.a. Estrategias de Evolución

Enfatizan los cambios de comportamiento al nivel de los individuos. Cada individuo de la estrategia evolutiva posee la siguiente estructura:

$$I = \langle x_1, x_2, \ldots, x_n, \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$$

Donde los x_i representan variables del problema y los σ_i, los parámetros de la estrategia evolutiva. Se tratan de técnicas de optimización cuya principal característica es su capacidad de auto-adaptación. Se caracterizan por dar la misma probabilidad de ser padres a todos los individuos durante el proceso de reproducción, además de ser considerado un proceso determinista.

Programación Evolutiva

Enfatizan los cambios de comportamiento al nivel de las especies. Son utilizados para resolver problemas de optimización y aprendizaje automático mediante máquinas de estados. Los individuos se presentan, de forma general, como ternas cuyos valores representan los estados de un autómata finito, aunque actualmente no hay considerada ninguna representación fija.

Estas técnicas muchas veces se confunden con los Algoritmos Evolutivos anteriormente comentados debido a sus similitudes. No obstante, se diferencian en que la PE es un método estocástico que trabaja sobre la población, mientras que los AE son deterministas y trabajan cada iteración sobre los individuos.

Programación Genética

Evoluciona expresiones representadas como árboles. En la Programación Genética se trata de desarrollar programas informáticos automatizados. Los individuos son tradicionalmente representados como estructuras de árboles que pueden ser fácilmente explorados de forma recursiva.

Un aspecto muy interesante de la PG es el operador de cruce. Como los individuos de la población suelen representarse como árboles, el operador de cruce puede

intercambiar nodos o ramas entre los padres, dando lugar a individuos muy variados que aportan más posibilidades al hacer la optimización.

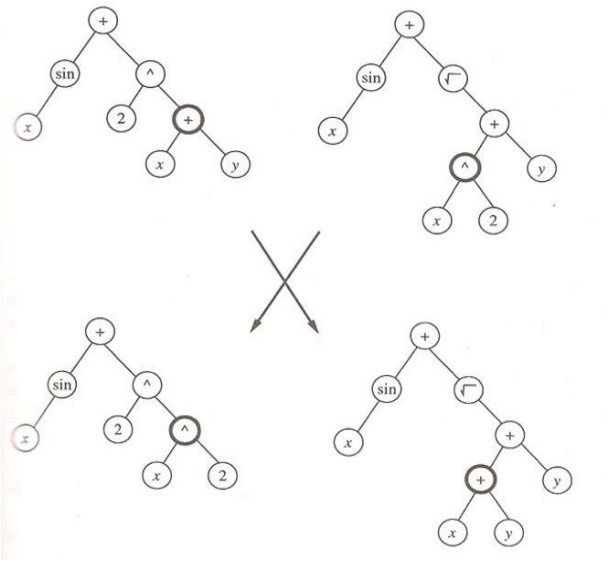


Ilustración 4: *Ejemplo de cruce en un árbol de programación genética.*

Por contra, la mutación sí suele considerarse más compleja, pues debe tener en cuenta sobre qué nodo trabaja y tener algún tipo de seguro ante posibles fallos. Esto hace que en muchas de las aplicaciones de esta técnica se considere como una operación innecesaria.

2.5.- Algoritmos genéticos

Los algoritmos genéticos (AG) son algoritmos de optimización, búsqueda y aprendizaje inspirados en el proceso de evolución genética **[1]**. Fueron oficialmente propuestos por John H. Holland en 1975, aunque hasta ese momento ya se había comenzado a trabajar con alguna de las técnicas que él incluyó. A día de hoy, es el paradigma de computación evolutiva más usual.

Su esquema es el mismo que hemos visto en el estudio de los algoritmos evolutivos **[10]**. El conjunto de individuos en la población se codifica a través de estructuras de datos que llamaremos cromosomas formados por genes. Un cromosoma con valores binarios (el gen puede tomar como valor 0 o 1) se denomina genotipo. A partir de ahí, se hace evolucionar la población de cromosomas a través de un proceso de competición y una serie de variaciones controladas por los operadores de mutación y cruce.

2.5.1.- Modelos

Dentro del paradigma de los Algoritmos Genéticos, nos encontramos con dos formas de clasificarlos muy fáciles de diferenciar y cuya diferencia es bastante importante a la hora de desarrollar el algoritmo.

- **Modelo Generacional**
    - Durante cada iteración, se genera una nueva población completa de individuos que reemplaza a la original. Este modelo facilita el diseño de la operación de reemplazo, aunque puede hacernos descartar soluciones interesantes.
- **Modelo Estacionario**
    - Durante cada iteración, se escogen dos padres de la población a los que se le aplican los operadores genéticos. A partir de ahí, se genera uno o varios descendientes que reemplazan al mismo número de cromosomas en la población original. Puede producir una presión selectiva alta (convergencia rápida hacia óptimos locales) cuando elegimos reemplazar a los peores cromosomas de la población.

2.5.2.- Construcción

Ahora que ya hemos tratado los conceptos generales relativos a los algoritmos evolutivos, vamos a estudiar el proceso que hay que llevar a cabo para diseñar y construir el algoritmo.

2.5.2.a. Diseño de la representación

El primer paso será definir el mecanismo mediante el cual podremos codificar cualquier posible solución en un cromosoma, y decidir si esta representación será binaria o no.

Será uno de los pilares durante el diseño de nuestro algoritmo, pues es una decisión muy importante. Una buena representación puede facilitar mucho el diseño del resto de operadores para un problema concreto.

Algunas de las representaciones más comunes son:

- **Representación binaria**
  - Da lugar a Genotipos con genes que pueden tomar valor 0 o 1. Es una representación muy sencilla y con la que es fácil trabajar en pasos posteriores, aunque hay casos en los que no se puede aplicar
- **Representación con números reales**
  - Se trata de representar a los individuos como vectores de valores de números reales. Es bastante utilizada. La función de evaluación opera para dar al cromosoma un valor de evaluación real tal que: f: $R^n \rightarrow R$
- **Representación de orden**
  - Los individuos son permutaciones dentro de un sistema. Es una forma de representación muy útil para problemas de secuenciación. Los operadores de cruce y mutación suelen ser un poco más complicados de implementar, pues deben asegurar que sigue tratándose de una permutación.

2.5.2.b. Definir cómo inicializar la población

Se puede realizar de manera uniforme por el espacio de búsqueda, de manera que haya valores muy dispersos por este que nos permitan valorar el máximo número de soluciones posibles sin caer en la convergencia prematura.

Otra opción frecuentemente utilizada es la de utilizar los valores generados previamente por otra heurística para inicializar nuestra población de soluciones.

2.5.2.c. Definir la correspondencia entre Genotipo y Fenotipo

En ocasiones, esta correspondencia será trivial y podremos realizarla fácilmente. Para otros problemas, nos encontraremos en la necesidad de utilizar alguna función que, tras aplicarla sobre los valores del cromosoma, nos devuelva el valor asociado del fenotipo para el problema concreto.

2.5.2.d. Diseñar la evaluación de individuos

Durante la ejecución del algoritmo, será necesario saber la adaptación de todos los individuos de la población. Como ya hemos visto, los individuos mejor adaptados deben tener mayores probabilidades de reproducirse, o lo que es lo mismo, de trasmitir sus genes a próximas generaciones. Para calcular dicha adaptación, utilizamos una función *fitness* que nos permita unificar en torno a un criterio los valores de adaptación de los individuos. Es un paso muy costoso computacionalmente.

2.5.2.e. Diseñar la estrategia de selección de padres

El objetivo de este paso es garantizar que los individuos con mayor adaptación al medio teniendo representación en futuras generaciones. No obstante, hay que permitir que individuos no tan buenos puedan reproducirse. Un proceso de selección muy duro para los individuos “malos” aumentará la presión selectiva que define la reproducción, conduciendo con mayor hábito a óptimos locales. Esto puede ser útil si lo que necesitamos es llegar a una solución que, aunque no sea tan buena como otras, sea obtenida rápidamente.

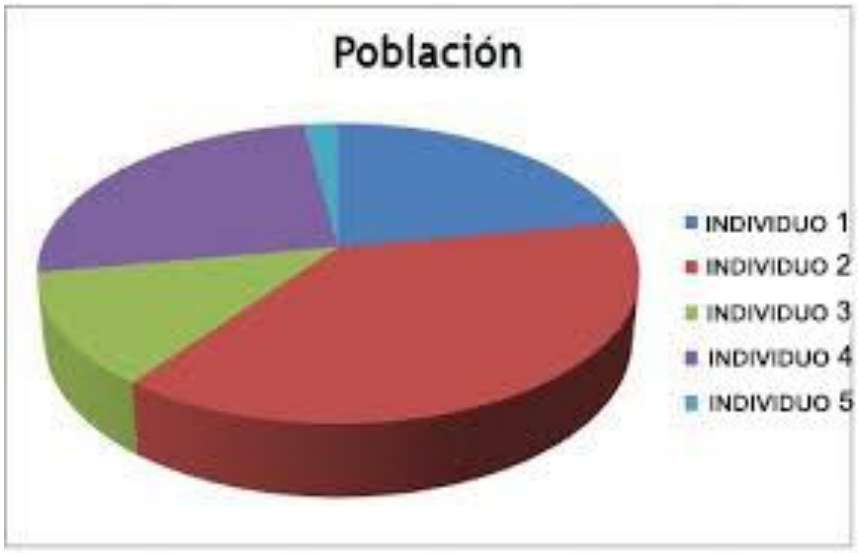
Existen varios mecanismos conocidos para definir estrategias de selección. Comentaremos algunos de los más comúnmente usados:

SELECCIÓN PROPORCIONAL

Con este método se le proporciona a cada individuo una posibilidad de reproducirse directamente proporcional al valor de su función de adaptación tal que:

$$ps_i = \frac{f_i}{\sum_j f_i}$$

A partir de ahí, se elige con aleatoriedad uno de ellos atendiendo a sus posibilidades. Esto puede provocar convergencia prematura si los mejores individuos dominan rápidamente la mayor parte de las posibilidades. Este modelo también es llamado como selección por ruleta, pues es fácilmente representable por un gráfico circular como el siguiente.



I

Ilustración 5: *Gráfico circular que representa la elección de padres en una selección proporcional.*

SELECCIÓN POR TORNEO

Se forman grupos de N individuos dentro de la población. Dentro de esos grupos, se escoge el mejor individuo, que pasará directamente a ser padre. Cuanto mayor N, mayor es la presión selectiva. La ilustración 6 representa un ejemplo de selección por torneo.

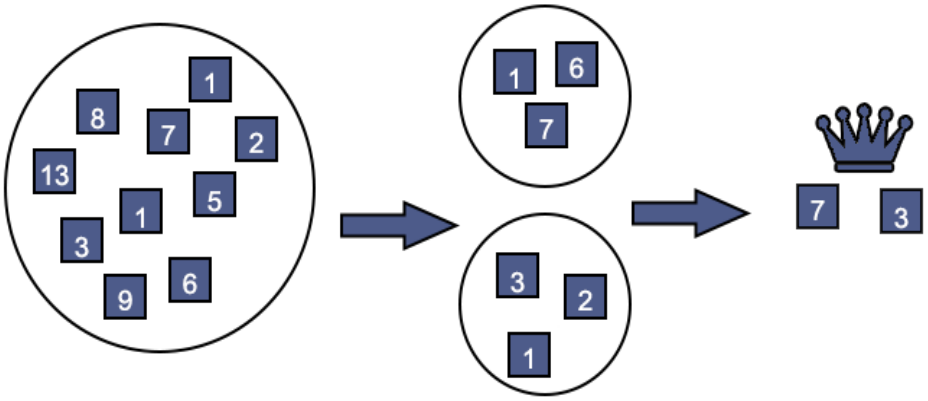


Ilustración 6: *Esquema que ejemplifica un proceso de selección padres por torneo.*

SELECCIÓN ALEATORIA

Se escoge un número de individuos como padres de forma totalmente aleatoria. Reduce la presión selectiva, aunque probablemente tarde más en encontrar una solución óptima.

SELECCIÓN ELITISTA

Se seleccionan los N mejores cromosomas entre padres e hijos. Estos cromosomas se incluyen directamente en la siguiente generación. Cuanto mayor sea N, mayor será la presión selectiva, pues estaremos pasando óptimos locales entre generaciones.

2.5.2.f. Diseñar un operador de mutación

La principal tarea del algoritmo de mutación es favorecer la exploración del espacio de búsqueda en su totalidad. Se trata de una operación con probabilidad baja de ocurrencia que, de darse, modifica uno o varios genes dentro del individuo, provocando variaciones respecto a los descendientes esperados.

La mutación binaria es seguramente la más fácil de aplicar. Se le da una misma probabilidad de mutación a todos los genes del cromosoma y se cambia un 0 por 1 en caso de tener que aplicar el operador. El caso de la mutación sobre cromosomas con valores reales suele realizarse utilizando operadores estadísticos que cambien un valor dentro de un rango predefinido. Por último, en individuos representados como permutaciones, se escogen dos genes aleatorios y se intercambian sus posiciones.

2.5.2.g. Diseñar un operador de cruce

El operador de cruce trata de favorecer la persistencia de las mejores características de la población en futuras generaciones. Para que este operador sea correcto, debe producir cromosomas válidos de acuerdo a la representación y producir hijos que hereden características de ambos padres. La probabilidad de efecto de este algoritmo suele ser relativamente alta.

Su funcionamiento en líneas generales es el de intercambiar las características de los dos padres para producir uno o dos hijos dependiendo del modelo.

2.5.2.h. Definir las estrategias de reemplazo

Una vez hemos generado los hijos, debemos tomar la decisión de si deben ser o no incluidos en la población de la siguiente generación y, en caso positivo, a quién deben sustituir.

En modelos generacionales, los hijos sustituirán al total de la población anterior. No obstante, se recomienda utilizar **elitismo** para que uno o varios de los mejores individuos no puedan ser sustituidos.

En modelos estacionarios, debemos elegir la estrategia de reemplazo con mayor conciencia. Las técnicas de reemplazo estacionario más comunes son:

- **Reemplazar al peor**
- **Reemplazo de torneo restringido**. Se genera un subconjunto de k elementos entre los que se escoge el peor.
- **Reemplazo del padre.** El hijo reemplaza al peor padre. En caso de haber dos hijos en un mismo proceso de reproducción, se reemplaza ambos padres.
- **Reemplazo del peor semejante.** Se escoge un criterio para establecer similitudes entre los individuos. A partir de ese criterio, se reemplaza al peor dentro de los parecidos al nuevo individuo.

2.5.2.i. Decidir la condición de parada

La condición de parada es un aspecto necesario en cualquier metaheurística. Su función es terminar la ejecución cuando la solución es lo suficientemente buena o se cree

que no se merece gastar más tiempo o recursos en una mejor. Algunas de las condiciones de parada más comunes son:

- Encontrar un óptimo
- Definir un numero estático de generaciones
- Tras un número de generaciones sin encontrar mejora.

3.- DISEÑO DE VIDEOJUEGOS

3.1.- Introducción al diseño de videojuegos

Para saber cómo se diseña un videojuego, primero tenemos que saber qué es lo que vamos a diseñar. Dar una definición de lo que es un videojuego es muy difícil, pues se trata de un campo demasiado amplio en cantidad y variedad. Hablamos de crear una definición para un concepto que abarca desde *Asteroids* (1979) hasta superproducciones multimillonarias como el último juego de la saga *Metal Gear*, *The Phantom Pain*.

Una propuesta interesante es el **sistema de formas** que propone Keith Burgun **[22]**, reputado diseñador de videojuegos con dos libros publicados sobre el tema.

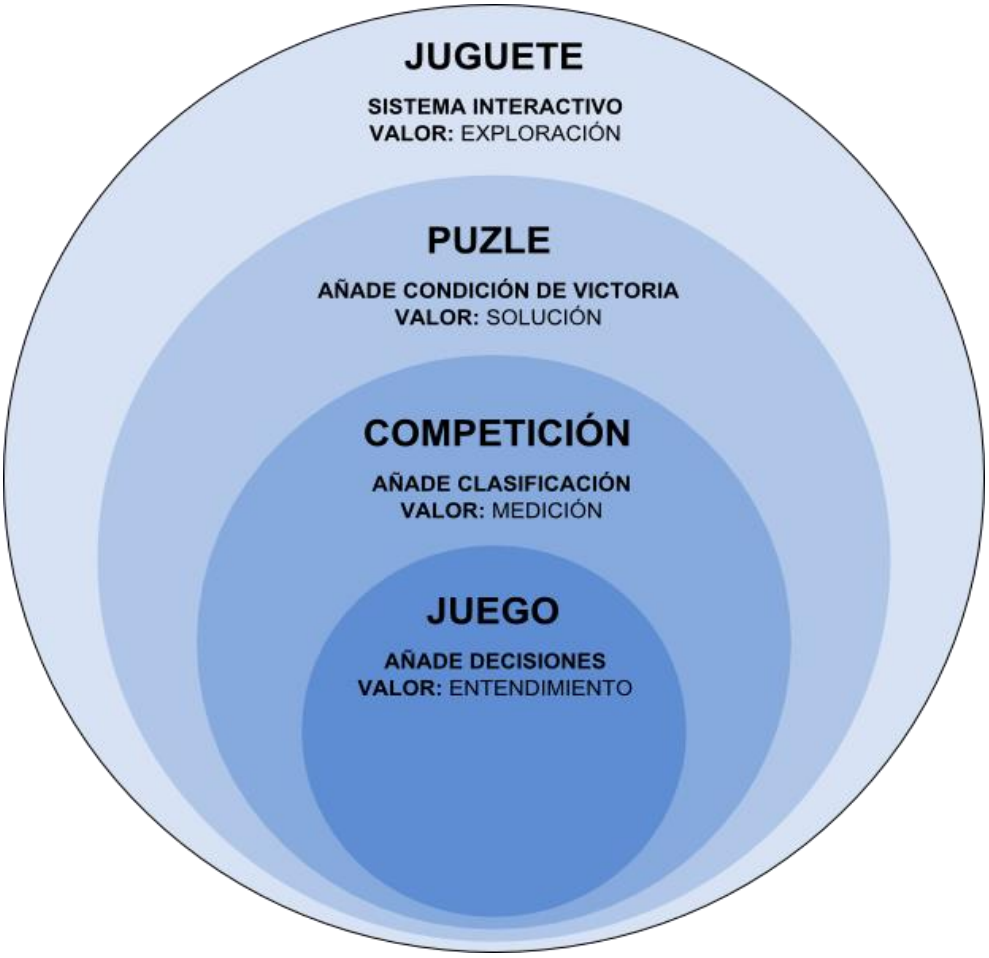


Ilustración 7: *Sistema de formas explicado por Keith Burgun*

Burgun señala que cualquier elemento dentro de cualquiera de las formas puede ser considerado un videojuego, de hecho, asegura que ninguna forma es mejor que otra o asegura que el producto final vaya a terminar siéndolo.

Juguete - Sistema Interactivo

Cualquier cosa que permita al jugador interactuar con ella puede ser considerada dentro de la forma Juguete. Su valor consiste en permitir la exploración, el descubrir algo nuevo mientras interactuamos con el sistema. *Minecraft, Los Sims* y la mayoría de simuladores entran dentro de esta clasificación

Puzle - Sistema Interactivo con condición de victoria

Si además de interactuar, existe una o varias reglas que nos permita ganar (pero nunca perder) durante nuestra interacción con el sistema, se trata de un Puzle. La mayoría de juegos de un solo jugador suelen ser de este tipo. Por ejemplo, en *Uncharted*, existe una condición de victoria, que es completar el juego. No obstante, no podemos ser derrotados, puesto que, si morimos, tendremos que volver a empezar por donde íbamos, pero no hay derrota como tal. El juego no se termina hasta que no alcances la victoria.

Competición - Sistema interactivo con clasificación

La principal diferencia entre el puzle y la competición, es que aquí, además de vencedores, también hay vencidos. La mayoría de juegos de competición pasan a la siguiente forma, pues es difícil ver un juego de este tipo en el que no tengamos que tomar decisiones. *Guitar Hero* es un buen ejemplo dentro de esta clasificación, pues podemos ganar o perder, pero no tenemos que tomar decisiones durante el juego, solo pulsar los botones requeridos en el instante adecuado.

Juego - Sistema interactivo con clasificación y toma de decisiones

Para saber bien a qué nos referimos en esta forma, tenemos que tener claro qué es una decisión. Cuando llega el momento de hacer una elección, pueden darse tres posibilidades: Tener toda la información posible sobre las posibilidades, en cuyo caso solo elegimos entre perder y ganar; no tener ningún tipo de información sobre las posibilidades,

donde lo estaríamos jugando al azar; o tener cierta información, pero también parte de incertidumbre sobre cada elección, en cuyo caso sí estaríamos tomando una decisión. En resumen, una decisión debe ser una elección con consecuencias parcialmente conocidas.

Dicho esto, a la hora de diseñar nuestro videojuego es conveniente saber en cuál de estas categorías (o dentro de otras clasificaciones similares) se encontraría clasificado. Para poder llegar hasta esta conclusión, tenemos que definir la jugabilidad que tendrá nuestro videojuego con el máximo detalle posible, además de comenzar a definir las mecánicas con las que guiaremos al jugador.

3.2.- Jugabilidad

La jugabilidad (o *gameplay*) es el conjunto de reglas de funcionamiento que definen la calidad del videojuego **[18]**. En resumen, define a qué se va a jugar, de qué manera, qué acciones podrá realizar el personaje y cómo va a reaccionar el entorno a estas.

Muchos expertos recomiendan, antes de detallar las mecánicas que guían la jugabilidad, definir el personaje y sus características: herramientas, habilidades y posibilidades que tendrá el jugador para manipularlo dentro de su entorno. De esta forma, conoceremos la gran mayoría de las posibilidades que tendrá el jugador para entonces diseñar el juego de cara a entretenerle.

En otras publicaciones, se recomienda realizar un diseño de niveles, y de los retos dentro de estos niveles, para posteriormente diseñar un personaje que pueda adaptarse a cómo estos están construidos, y que pueda hacer las acciones necesarias para superarlos.

La jugabilidad es considerada el corazón del videojuego. Todo lo demás debe girar en torno a una experiencia de juego divertida e interesante para el jugador o el espectador. Por tanto, debemos anteponer la jugabilidad ante otras decisiones durante el diseño.

3.3.- Mecánicas

Dentro de un videojuego, llamamos mecánicas a las reglas que dirigen las posibles interacciones entre los agentes del videojuego. Es decir, lo que los jugadores realizarán y sufrirán durante el tiempo de juego. La labor del diseñador de videojuegos será conseguir un conjunto de reglas lo suficientemente equilibradas como para favorecer el disfrute de los jugadores. El objetivo al que se pretende llegar con las mecánicas es nutrir y motivar lo diseñado para la jugabilidad

Por ejemplo, si queremos desarrollar un *videojuego* con temática *Survival Horror*, género donde los recursos para el jugador deben ser escasos y debe suponerle un reto el avanzar, no podemos incluir entre nuestras mecánicas aspectos como un guardado automático cada minuto, un uso ilimitado de los objetos del personaje o una barra de vida que disminuya muy poco. Para este caso, las mecánicas deberían provocar en el jugador cierto nerviosismo o ansiedad por la alta posibilidad de morir y las consecuencias que conllevará dicho fracaso, como, por ejemplo, tener que volver desde puntos de guardado relativamente lejanos.

Todos los componentes como mecánicas, niveles, narrativa y efectos audiovisuales deben diseñarse para estar al servicio de la jugabilidad y favorecer al máximo la experiencia de juego del usuario.

3.4.- Tema

El tema, tanto en el cine, como en la literatura, como en los videojuegos, es el todo alrededor del cual hemos de crear nuestro universo. En el caso de los videojuegos, tenemos que tener en cuenta la importancia del tema para diseñar el resto de aspectos: modelos, música, guion, niveles, dificultad y, lo más importante, jugabilidad. Es necesario tener claro este concepto, pues hemos de dirigir el resto de elementos como subordinados de este.

En un videojuego, todos los elementos deben buscar favorecer y potenciar la experiencia de juego buscada **[21]**. El tema podría considerarse una condensación o un sumario de la experiencia de juego buscada. Por ejemplo, en el videojuego *Alan Wake* (2010), el tema es el miedo a la oscuridad. Partiendo de ahí, se diseña una jugabilidad complicada, en la que si tu personaje muere te “duele” porque tienes que rehacer bastante camino, escenarios muy poco iluminados, con pocos focos de luz y una trama al nivel de los mejores thrillers de Hollywood. Por esto es importante tener claro desde un principio cual va a ser el tema de nuestro juego y, a partir de ahí, que experiencia de juego queremos conseguir sobre el jugador.

3.5.- Motor de Videojuego

Al igual que en el resto de productos software, el desarrollo debe ser un paso posterior al diseño. En el caso de los videojuegos, la implementación suele ser la etapa más larga y difícil, tanto es así que gran parte de los proyectos que acaban fracasando lo hacen en esta etapa por escasez de recursos para continuar su desarrollo. Durante muchos años, el desarrollo de videojuegos fue una tarea casi impracticable para desarrolladores independientes y de bajo presupuesto.

Como respuesta a este problema y gracias al crecimiento del sector, con el paso de los años se ha ido facilitando el desarrollo de las tareas más frecuentes dentro del desarrollo de videojuegos. Con esta idea, surgen los motores gráficos o motores de videojuegos.

Un motor de videojuego **[23]** es un software con una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Se trata de herramientas muy útiles y frecuentemente utilizadas en el sector, pues suelen abaratar enormemente el gasto de tiempo y recursos que supone la implementación del juego.

Los motores de videojuego, también llamados motores gráficos, tienen como función facilitar al equipo desarrollador un motor de renderizado para gráficos 2D o 3D, un motor físico o detector de colisiones y una gran cantidad de utilidades extras que le permitan ahorrar etapas de desarrollo.

Un videojuego se diseña, normalmente, como un sistema diseñado por capas o particiones cerradas. En las etapas de mayor nivel se encuentran, usualmente, elementos de la interfaz a disposición del jugador. Más visuales y menos difíciles de comprender para el jugador. Las capas más bajas incluyen la comunicación con el hardware, que puede ser consolas, ordenadores e incluso dispositivos móviles o tabletas, cada uno con su sistema operativo propio. El motor de videojuegos suele ser el encargado de gestionar esas tareas, ahorrando un trabajo muy duro para el programador, pues debe traducir todas las entradas a un “lenguaje” que el hardware sea capaz de interpretar correctamente.

Por todo esto, es usual ver como estudios independientes e incluso profesionales, hacen uso en sus proyectos de este tipo de herramientas. A la hora de desarrollar un videojuego, la elección del motor de videojuegos también es importante. Necesitamos tener en cuenta aspectos como si se tratan gráficos en 2 dimensiones o en 3, si necesitamos un

acabado más profesional o por el contrario necesitamos un motor sencillo y fácil de usar, si necesitamos o no que su uso sea gratuito, etc. A continuación, vamos a hacer un repaso de los motores más comunes del mercado actual, tratando de diferenciarlos a partir de sus características.

3.5.1.- UNREAL ENGINE 4

Uno de los motores gráficos más utilizados en el mercado a nivel profesional es *Unreal Engine 4* **[13]**. Desarrollado por *Epic Games,* es la cuarta versión lanzada por la empresa estadounidense. Ha sido usado en juegos de gran prestigio y presupuesto por empresas de alta reputación como *Bioware o 2kGames.*

Este motor da soporte a casi todas las plataformas del mercado actual, incluyendo juegos de red, ordenadores, móviles, consolas y sistemas de realidad virtual. Soporta y facilita el desarrollo con formas 3D complejas, de hecho, es su potencia gráfica lo que lo convierte en un motor altamente competitivo. Se puede descargar gratuitamente desde la página oficial, no obstante, cualquier juego lanzado al mercado que supere los 3000 dólares en ventas tendrá que ceder a *Epic Games* el 5% del dinero ingresado.

El principal aspecto negativo de *Unreal Engine 4* es la dificultad para obtener cierto manejo con la plataforma al comenzar a trabajar con esta. Además, está basado en C++, lenguaje que puede resultar bastante difícil de utilizar para personas que deseen realizar un videojuego con poco o ningún conocimiento en programación. En conclusión, *Unreal* se considera un motor muy profesional por las posibilidades que nos aporta gráficamente a cambio de las dificultades que pueda suponer su aprendizaje y su uso generalizado.

3.5.2.- CRYENGINE V

CryEngine es un motor de juego desarrollado por *CryTek* que actualmente es propiedad de la empresa Ubisoft **[14]**. Se trata de un motor 3D bastante potente que permite programar en C# y C++. Actualmente no está entre los motores 3D profesionales más utilizados y son pocas las grandes empresas que terminan de utilizarlo. No obstante, es interesante el realismo que permite en los personajes y sus animaciones y el hecho de incluir terreno y vegetación por defecto. Además de

incluir entre sus prestaciones muchos elementos interesantes y que dejan un muy buen acabado.

3.5.3.- RPG MAKER

Aunque no es considerado un motor de videojuegos en su definición, es una herramienta muy interesante para realizar videojuegos RPG. Creado por *ASCII*, es propiedad de *Enterbrain*, lleva existiendo (en diferentes versiones) desde 1995, por lo que esta aplicación tiene cierta reputación a sus espaldas **[12]**. Lo más interesante de *RPG Maker* es su facilidad de uso, ya que permite realizar juegos sin conocimiento alguno en programación, pero también tiene un complejo editor de scripts, que hasta la última versión venía preparado para el lenguaje de programación *Ruby* (Actualmente *Javascript*). Su amplia comunidad (muy activa) es otra gran ventaja, aunque suele salir perjudicado por su poca potencia gráfica, que además solo admite 2D.

3.5.4.- GAME MAKER

Game Maker: Studio es otra de las herramientas más recomendadas para aquellos que quieren empezar en el desarrollo de los videojuegos **[11]**. No es necesario aprender ningún lenguaje de programación, aunque incluye un lenguaje de edición de scripts propio (*Game Maker* Lenguaje *GML*). Se basa en un sistema de “Drag & Drop” muy intuitivo, pero limitado en cuanto a la potencia del resultado final.

3.6.- UNITY 5

Unity es un motor de videojuego multiplataforma creado por Unity Technologies y disponible para Windows, OS X y Linux, con soporte de compilación para la mayoría de las plataformas del mercado actual, incluyendo los campos de realidad virtual. **[24]**

Se trata de un motor con bastante reputación y una potencia más que suficiente para un acabado muy realista y profesional. Además, su principal ventaja respecto a la mayoría de competidores es la facilidad para aprender a trabajar con la herramienta y la posibilidad de ver resultados casi desde el primer momento. Esto permite que desarrolladores independientes y con pocos conocimientos puedan conseguir videojuegos con muy buen acabado. Además, el uso de Unity es totalmente gratuito para aficionados o incluso empresas de muy bajo presupuesto o recién formadas. Para empresas de mayor facturación, Unity solicita el uso de otros dos planes, Plus (35 $/mes) y Pro (125 $/mes). Es, por todo esto, uno de los motores de videojuego más utilizados para aprender sobre Diseño y/o Desarrollo de videojuegos en el sector académico.

Además, dispone de gran cantidad de contenido de terceros que pueda facilitar el desarrollo, y una muy amplia documentación con muchos ejemplos y fácil de entender en la página oficial. Unity, que permite desarrollar en 2D y 3D, tiene dos modos: Personal (Gratuito) y Profesional (Para empresas).

Todas las características comentadas anteriormente han provocado que nos decantemos por el uso de Unity para la consecución del prototipo que queremos realizar. Pues, además de cumplir con todos los requerimientos para desarrollar el proyecto, exportar desde esta plataforma a otras donde poder terminar el producto en un futuro no es demasiado complejo. Utilizaremos C# para los scripts ya que es un lenguaje soportado por Unity no muy complejo y con bastantes seguidores, lo que puede facilitarnos encontrar y corregir errores que se puedan ir presentando durante el desarrollo.

4.- ALG. EVOLUTIVOS EN VIDEOJUEGOS

4.1.- La inteligencia artificial en los videojuegos

En la gran Mayoría de videojuegos, independientemente del género, existen una o varias inteligencias artificiales que modelan el comportamiento de los diversos personajes o incluso del mundo, tratando de dotarles de una “conciencia” que pueda sumergir al jugador en la narrativa o simplemente divertirle más.

Podemos encontrar ejemplos de esto en casi cualquier videojuego del mercado actual, desde simuladores deportivos que utilizan una IA para controlar a los jugadores y tratar de suponerle un reto de dificultad variable al jugador hasta videojuegos de mundo abierto donde las acciones qué realizan (y cómo las realizan) también debieron ser programadas. Por tanto, es posible encontrar números estudios sobre el comportamiento a seguir por los diferentes agentes que componen el mundo en un videojuego.

Respecto a esto, es importante tener en cuenta que la implementación de las inteligencias artificiales en los videojuegos es frecuentemente adhoc (solución para un problema preciso no generalizable a otros), por tanto, distintos juegos tendrán distintas heurísticas, de manera que tratarán de adaptarse a las mecánicas de juego concretas **[15]**. Incluso, en muchos juegos nos encontramos con la necesidad de diseñar e implementar varias Inteligencias Artificiales que modelen de manera diferente la conducta de dos personajes distintos. Esto tiene sus pros y contras, por un lado, nos permite aportar gran realismo al mundo y al comportamiento de los personajes, como en la realidad, distintas personas actuarán de manera diferente en la misma situación; como inconveniente, esta decisión aumentará las horas de trabajo y, por tanto, el coste del desarrollo.

En producciones antiguas de videojuegos, las herramientas para la creación de videojuegos no disfrutaban de los avances que tenemos a día de hoy. Tanto a nivel hardware, como software, realizar un videojuego con una inteligencia artificial avanzada o unos gráficos potentes era un imposible. Por esto, y teniendo en cuenta que los ingresos no eran ni la mitad de los que consigue una gran producción hoy, los desarrolladores estaban obligados a reducir los costes en este tipo de proyectos al máximo. Al programar el comportamiento de los diversos enemigos, se limitaban a programar una sola IA que definiera el comportamiento de todos, lo cual acababa resultando en videojuegos monótonos que a la larga podían ser aburridos para el jugador. Uno de los ejemplos más

significativos en este aspecto es el juego *PacMan* (1980), donde el jugador se enfrenta a una serie de fantasmas que tratan de darle caza en un mundo muy limitado en cuanto a posibilidades.

Visto el éxito de los juegos Arcade en los 80s, y tras comprender el problema que suponía la monotonía en el comportamiento de la CPU, surgieron dos grandes estrategias como solución. La primera, aumentar la complejidad de las heurísticas usadas por la CPU, que aumentaría el reto para el jugador y daría mayor complejidad a sus acciones. Juegos como *Street Fighter II* (1991), que supuso un éxito a nivel mundial, utilizaron esta estrategia para divertir al jugador con un gran roster de personajes contra los que combatir. La otra solución fue incluir algún tipo de variable aleatoria en los algoritmos, permitiendo a la CPU hacer movimientos impredecibles y proponiendo un reto bastante más duradero al jugador.

También es interesante comentar la IA “perfecta” desarrollada para el juego de lucha en 2D *Mortal Kombat* (1992). Se llama así debido a que es la máxima dificultad desarrollada para la CPU del juego en la que está siempre actúa tratando de optimizar los resultados: bloquea o esquiva todos los ataques bloqueables, evita saltar o agacharse si pueden suponer un impedimento a un bloqueo y conoce y ejecuta todo contra-ataque posible. A pesar de todo esto, es posible, aunque muy difícil, conseguir ganarle, lo que supone un reto incluso para los usuarios de mayor habilidad.

Durante los años siguientes, la industria del videojuego seguiría un crecimiento exponencial en cuanto a número de ventas e ingresos, que se traduciría también en una mayor inversión en cualquier elemento que pudiera mejorar un producto. Esto incluye la programación de inteligencias artificiales mucho más avanzadas y realistas. En este punto, es necesario mencionar dos juegos: *Half Life* (1998) y *Halo* (2001). En estos dos juegos del género *first person shooter* (FPS) nos enfrentamos a enemigos que son capaces de sorprendernos, reaccionar ante nuestros movimientos e, incluso, cooperar con otras inteligencias artificiales para acorralarnos.

A día de hoy, las grandes empresas del sector disponen de presupuestos mucho mayores y de plantillas más amplias con personal especializado el desarrollo de videojuegos. Además, el software disponible para el desarrollo de videojuegos, como por ejemplo los motores gráficos anteriormente vistos, han evolucionado para ser mucho más útiles y potentes. Esto, evidentemente, supone muchísimas mejoras en el producto, como en los apartados gráfico, narrativo y el que aquí nos incumbe, el relacionado con la

programación de comportamiento. Ejemplos muy llamativos de este crecimiento pueden ser *Far Cry 3*, donde incluso los animales tienen inteligencias artificiales avanzadas que les permiten cazar, huir o esconderse dependiendo de su entorno; *Skyrim* o *FallOut,* con gran cantidad de *NPCs* (Personajes no manejables) que se comportan de una u otra forma dependiendo de la situación; e incluso simuladores deportivos como la franquicia *FIFA* de *EA Sports*, que provee un complejo sistema inteligente para que tanto los rivales como los jugadores de tu equipo se adapten a la situación dentro del partido.

4.2.- Contenido procedural en videojuegos

En el anterior apartado hemos estudiado la evolución de la complejidad del comportamiento de los diversos agentes que forman parte del videojuego, pero también es importante estudiar todo lo que le rodea. Dentro de este apartado podemos incluir el mundo en el que se mueven, los objetos que pueden utilizar, los obstáculos que se puedan encontrar y cómo todo esto interacciona entre sí.

Durante los primeros años del sector se desarrollaba un número limitado de contenido, lo cual se traducía en pocas horas de jugabilidad o un juego que acabara resultando repetitivo. Gracias al 'Boom' del sector en los 80s y a los avances en hardware, los estudios pudieron comenzar a permitirse desarrollar más contenido y de mayor calidad: Mayor número de escenarios, que también serían más grandes; más personajes, mejor desarrollados y con más posibilidades; más armas u objetos utilizables por el jugador e incluso mayor número de eventos a partir de las acciones del jugador.

Con el paso de los años, aparecerían aspectos que dieron un gran plus a la calidad de estos productos. Entre ellos, el "mundo abierto". Un juego es considerado de mundo abierto si ofrece al jugador la posibilidad de explorar un mundo relativamente grande y sin (demasiadas) barreras. Dos buenos ejemplos y estandartes de este tipo de videojuego son *Final Fantasy VII* (1997) y *Grand Theft Auto III* (2002). Otro aspecto que surgió y que a día de hoy se sigue utilizando bastante fue la personalización de contenidos, incluyendo aquí al propio jugador. Cambiar los colores o elegir entre diferentes posibilidades (cambiar peinado o ropa del personaje) permitirían hacer crecer el concepto "Player Story" (o la historia que el jugador crea en su mente mientras juega). El mejor ejemplo que podemos encontrar aquí es *Los Sims* (2000), donde el jugador tiene plena libertad para crear el mundo a su manera.

Pero al final, el problema de estos juegos seguía siendo el mismo: El contenido era finito, por lo que al final un jugador podría a sentir que ya había conseguido todo lo que podía conseguir. Una solución muy adoptada ante este problema es la de mantener un desarrollo del videojuego "permanente", incluso después de realizar el lanzamiento, para proporcionarle más contenido al jugador de manera continua a través de Internet. El problema de esto es que, a pesar de haber acabado un producto, tenemos que continuar invirtiendo en el desarrollo de contenido, lo cual puede ser un riesgo. La otra solución es utilizar un desarrollo de contenido procedural.

El desarrollo de contenido procedural consiste en no delegar únicamente la creación de los contenidos del videojuego en el equipo de desarrollo, sino ayudarte de una serie de algoritmos para generar, siguiendo unas directrices, un número ilimitado de contenido, extendiendo bastante la vida del producto. Esto permite la sensación de estar en un videojuego infinito, con un universo creado alrededor del jugador que aparentará ser único e irrepetible.

Comúnmente, se asocia la creación de contenido procedural a la creación de escenarios, niveles y mapas. No obstante, también puede ser un elemento determinante a la hora de crear personajes, enemigo o herramientas únicas, además de distribuir como se desee estos.

4.3.- Videojuegos y algoritmos evolutivos

Tras hablar de inteligencia artificial y contenido proceduralmente generado en videojuegos, vamos a tratar de qué manera puede ayudarnos la computación evolutiva en el desarrollo de videojuegos, y a exponer algunos ejemplos en los que juegos del mercado utilizaran algún paradigma evolutivo.

En el caso de la inteligencia artificial de los personajes, nos encontraremos con muchos casos en los que necesitamos que el personaje en cuestión actúe como una persona, sea capaz de razonar y aprender. Para este caso, y como ya hemos visto, la computación evolutiva es muy útil, pues puede simular bastante bien el proceso de aprendizaje.

En el caso del contenido, los algoritmos evolutivos pueden ser muy útiles para la elaboración de mapas e incluso de algunas de las mecánicas del videojuego. Además, la computación evolutiva puede ser muy útil para comprobar cuál es la solución óptima para completar un nivel, puesto que podemos tomar simulaciones del nivel como individuos de la población y evolucionarlos para conseguir la mejor solución.

A continuación, vamos a comentar algunos ejemplos de videojuegos que incluían computación evolutiva:

GALACTIC ARMS RACE (Evolutionary Games – 2010)

Se trata de un *shooter* en el que manejamos a una nave por el espacio cuyo objetivo es eliminar a sus enemigos sin ser eliminada. Hasta ahí, todo normal. Lo curioso de este videojuego es que utiliza la generación de contenido procedural para conseguir fabricar armas únicas para el jugador. Además, contiene un algoritmo evolutivo que le permite potenciar la fabricación de armas parecidas a las que el jugador más ha utilizado.

WARNING FOREVER

Warning Forever es un *shooter* en 2D desarrollado por *Hikware* en 2003. Aquí, el jugador maneja una pequeña nave que debe destruir grandes naves enemigas. Cada vez que el jugador elimina a uno de los enemigos, se genera otro más avanzado que, basándose en cómo se eliminó al jefe anterior, se adapta para suponer un reto mayor al jugador. El juego, de producción independiente y gratuito al público, no tuvo demasiado éxito, aunque es un objeto de estudio muy interesante.

BOSSES FOREVER

Bosses Forever es un adictivo juego desarrollado por la compañía independiente *Too DX*. Consiste en un escenario en 2D en el que podremos jugar solos o acompañados de un amigo (Modo cooperativo) para tratar de eliminar a un jefe disparándole. El jefe tiene una inteligencia artificial que le permite aprender sobre cómo recibió los ataques y qué ataques suyos fueron más efectivos, para protegerse y atacar con mayor efectividad contra el usuario. Cada vez que eliminamos a un jefe, nos enfrentamos a otro de mayor dificultad, siendo un verdadero reto incluso para los más habilidosos.

ANGELINA

Angelina no es exactamente un videojuego, sino una inteligencia artificial capaz de hacer infinitas instancias de estos. Michael Cook, de *College London University*, creó un sistema capaz de crear de manera completamente autónoma un videojuego que pueda ser disfrutado por humanos. Angelina crea, de forma aleatoria, la disposición del nivel, los enemigos y su comportamiento, los criterios de éxito y las distintas recompensas que se puedan tener. Posteriormente, simula que el nivel puede ser completado y define la curva de dificultad entre niveles. Por el momento, Angelina requiere de colaboración humana para incluir gráficos complejos o música al videojuego, aunque es una IA muy interesante e innovadora.

5.- METODOLOGÍA DE TRABAJO

El objetivo de este TFG es conseguir el prototipo de un software clasificado como un videojuego. Por tanto, es necesario conocer las etapas en las que suelen dividirse los proyectos de creación software y las metodologías de trabajo más comúnmente utilizadas.

5.1.- Etapas del desarrollo de software

En el ámbito de la ingeniería de software **[26]**, es común dividir el proceso de desarrollo global del producto en etapas, de manera que podamos llevar a cabo un enfoque estructurado que nos permita obtener un producto de calidad con un número de recursos finito. En adelante, se presentan las etapas a través de las cuales desarrollaremos el videojuego.

ANÁLISIS

Durante la etapa de análisis se extraen los requisitos del producto software. En esta primera etapa, es necesaria mucha comunicación entre el cliente y el desarrollador, ya que el objetivo es conseguir la mayor cantidad de información posible sobre qué se quiere y qué se necesita. Es importante obtener una idea clara de los conceptos, llegando a conclusiones precisas con el cliente. En algunos casos, como para este proyecto, no hay un cliente completamente definido y externo al equipo de desarrollo. En esta situación, el principal objetivo de esta etapa será el de diferenciar los principales casos de uso y las clases conceptuales asociadas a estos.

DISEÑO

Tras obtener una información global, pero poco detallada de lo que se pretende conseguir con el sistema, pasamos a diseñar el producto software. Durante esta etapa, tendremos que tomar todas las decisiones importantes que nos permitan desgranar, con el máximo detalle posible, las partes de las que se compondrá el software para permitir su realización física. Es una etapa de gran importancia, pues un buen diseño aumentará la calidad del producto y facilitará la corrección de problemas que pudiéramos encontrarnos en etapas siguientes.

IMPLEMENTACIÓN

A grandes rasgos, se trata de traducir el diseño en código. Un buen diseño debería permitirnos identificar sin dificultades las tareas a realizar y el orden en el que hay que realizarlas.

PRUEBAS

Tras completar la implementación, deberíamos tener un software capaz de funcionar bajo las condiciones esperadas. A partir de ahí, tenemos que comprobar si realmente funciona correctamente, cumple con lo esperado por el usuario y no tiene fisuras en su funcionamiento al cambiar las condiciones. Hay varios métodos para completar esta etapa, aunque frecuentemente se colabora con el usuario final para comprobar que el software funciona correctamente.

DOCUMENTACIÓN

A lo largo del desarrollo del proyecto se deberían realizar dos informes: Un manual de usuario y un manual técnico o memoria. El manual de usuario consistirá en definir las posibilidades de funcionamiento del software: Qué cosas puede hacer y cómo pueden hacerse. El manual técnico será una memoria que contendrá todo el estudio del análisis y las decisiones de diseño que hemos tomado, con el fin de facilitar el mantenimiento futuro y/o cualquier ampliación sobre lo construido. No es una etapa en sí, sino un proceso que se va realizando durante todas las etapas.

MANTENIMIENTO

Es aconsejable realizar dos tipos de mantenimiento: uno correctivo, que consiste en resolver los errores que vayan surgiendo y un mantenimiento evolutivo, que permita añadir y mejorar funcionalidades.

5.2.- Metodologías de desarrollo software

A continuación, comentaremos dos de los enfoques de desarrollo más utilizados para el desarrollo de productos software.

5.2.1.- Desarrollo Tradicional

En el desarrollo tradicional de software, se van encarando las etapas de manera secuencial. Tienen que detallarse tanto el análisis y el diseño del software que se va a desarrollar en etapas tempranas del proyecto. Es necesaria mucha comunicación con el cliente durante las primeras etapas, pues solo se desarrollará aquello que se detalle antes de comenzar la implementación. Esto significa, que un análisis con ambigüedades o un mal diseño puede hacer fracasar el proyecto, incluso cuando pueden haberse invertido gran cantidad de horas en etapas posteriores.

Este modelo de desarrollo puede resultar útil para proyectos en los que el equipo de desarrollo tiene muy claros los requisitos que, además, no variarán durante el tiempo que tarde el proceso en desarrollarse o en proyectos de corta duración sobre los que no se pueda aplicar un enfoque ágil adecuado.

En los primeros años del desarrollo de software, era la metodología utilizada para desarrollarlo, aunque, tras comprobar sus deficiencias en proyectos de mucho presupuesto que fallaron por la poca adaptación a los cambios de esta estrategia, surgieron otras metodologías.

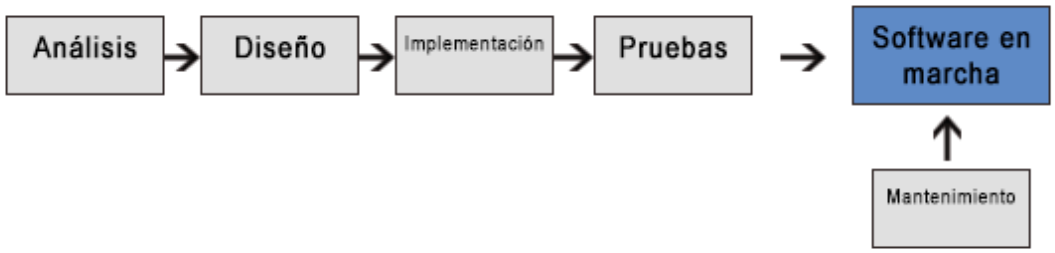


Ilustración 8: *Diagrama de desarrollo secuencial*

5.2.2.- Desarrollo Ágil

La respuesta a las deficiencias de la metodología tradicional surgió con la publicación de una serie de técnicas denominadas ágiles para el desarrollo de software. Aunque están compuestas por las mismas etapas por las que se compone la metodología tradicional, tienen una diferencia muy llamativa.

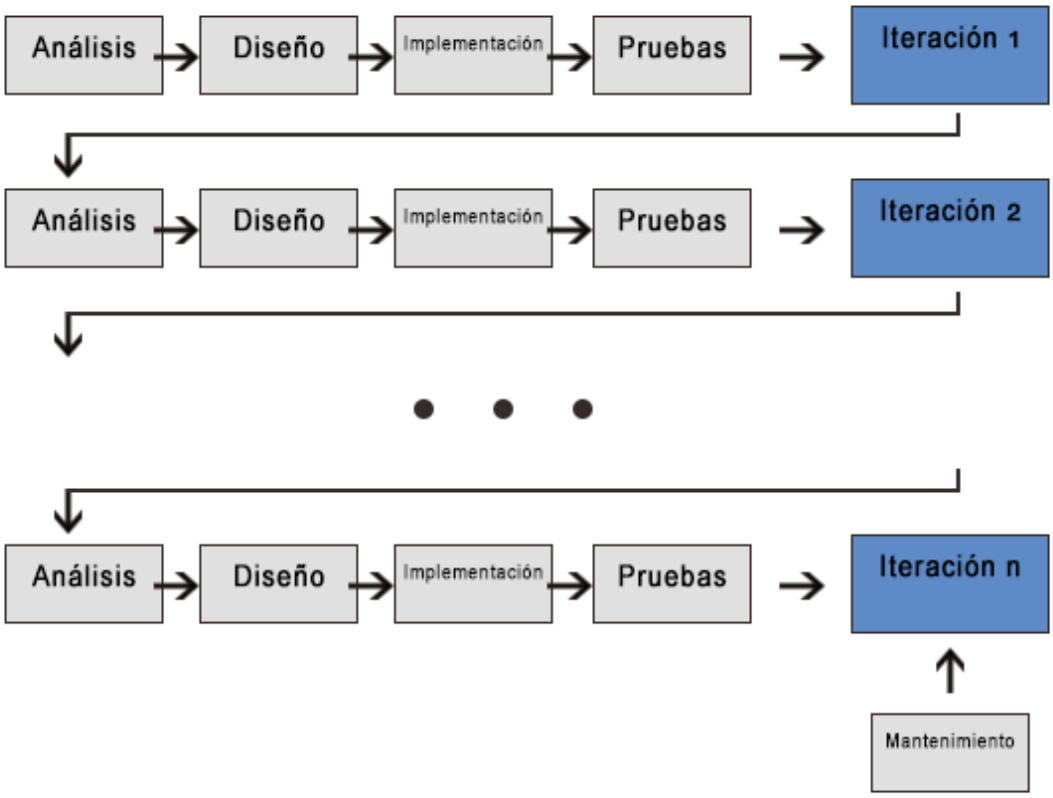


Ilustración 9: *Diagrama de desarrollo iterativo*

En el caso de las metodologías ágiles, se divide el proyecto en iteraciones, que podríamos ver como subproyectos de corta duración (2-4 semanas). Al principio del proyecto, se realiza un análisis muy básico al principio, obteniendo únicamente los requisitos de alto nivel y sin detallarlos demasiado. Entonces, ordenaremos los requisitos según la importancia que estimemos, y los asociaremos a las diversas iteraciones. De este modo, y para cada iteración, realizaremos un análisis y un diseño más profundo sobre los requisitos incluidos en la iteración para posteriormente implementarlos y realizar las pruebas durante esta.

Con un enfoque ágil, planificaremos con mucho detalle los “subproyectos” o iteraciones que se encuentren más cercanos en el tiempo, mientras que dejaremos para después el estudio y la planificación de iteraciones más tardías. Esta manera de trabajar favorece la inclusión de cambios entre iteraciones y posibilita añadir requisitos nuevos con el proyecto en marcha. Por ello, es necesario replanificar el proyecto conforme avanzamos en las iteraciones, pues tenemos que adaptar el planteamiento inicial a los cambios, desajustes o fallos en las estimaciones que hayan podido surgir **[27]**.

Por esto, es importante mantener varias reuniones al término y al inicio de cada iteración, de manera que podamos favorecer la comunicación entre el equipo de desarrollo y poder comprobar que el proyecto se está desarrollando al ritmo esperado, sin problemas, y cumpliendo lo prometido al cliente. Es importante llevar al equipo hacia los objetivos esperados y adaptarse a las exigencias que el cliente pueda ir mostrándonos.

Al favorecer la adaptación durante el desarrollo del proyecto, permite reducir muchos de los riesgos que aparecerían en proyectos tradicionales, como no cumplir lo esperado, consumir muchos más recursos de los planteados inicialmente, ya que normalmente se trata de no superar un tiempo previamente determinado durante el desarrollo.

5.2.3.- Conclusión

Para este proyecto software, se ha decidido trabajar con una metodología tradicional de desarrollo. La falta de experiencia del desarrollador en la implementación y el diseño de videojuegos requiere un buen análisis y un buen diseño de lo que se quiere implementar, puesto que, de otra forma, es muy probable llegar a conclusiones equivocadas sobre el alcance del proyecto. Además, parece difícil definir con buen criterio el número de puntos de historia que se le asignen a cada tarea, además de la velocidad estimada de desarrollo, por lo que se corre el riesgo de subestimar la dificultad de tareas como la realización de interfaces, aspecto realmente importante en los videojuegos, o la obtención de conclusiones a través de pruebas que realmente puedan permitirnos sacar el máximo partido al algoritmo genético desarrollado.

A lo largo de la memoria, se detallará el estudio inicial realizado para definir de manera clara el proyecto que se pretende desarrollar junto con un análisis que buscará favorecer una plena comprensión de los requisitos del producto software. Posteriormente, pasaremos a la etapa de diseño para identificar las mejores soluciones posibles a los

problemas encontrados durante la anterior etapa. Finalizado esto, realizaremos el desarrollo del software basándonos en lo anterior y concluiremos con el desarrollo de pruebas.

6.- DISEÑO DEL VIDEOJUEGO

6.1.- Visión general

El juego a desarrollar será un RPG (*Role Playing Game*) basado en un sistema de combate por turnos en el que el jugador tomará el rol de conciencia del personaje. Comenzará el jugador cuyo personaje tenga un atributo mayor de velocidad. Al jugador, en su turno, se le muestra la situación (posición de los personajes, barras de vida, posibles estados…) y se le presentan varias estrategias posibles para elegir el comportamiento al que deberá amoldarse su personaje.

Una vez que el jugador ha escogido estrategia, el control pasa por completo al personaje, que elegirá qué acciones realizar entre las posibles y en qué orden realizarlas. El número de acciones que puede realizar el personaje vendrá limitado por su cansancio. Cada acción consume un número variable de cansancio. Así, moverte en cualquier dirección puede suponer un gasto de cansancio menor que atacar, por ejemplo.

El mundo en el que ocurre la batalla estará dividido en secciones de igual tamaño, de tal modo que solo un personaje puede ocupar una sección concreta a la vez. Cada acción puede afectar a diferentes secciones, tomando siempre como punto de referencia la posición del jugador. Por ejemplo, una flecha en línea recta podría afectar a 3 secciones en esa dirección, mientras que un ataque de corto alcance solo a la sección adyacente. De igual manera, cada movimiento se realiza en secciones, haciendo así un movimiento “discreto”. Esto puede favorecer bastante la elección de la estrategia por parte del jugador, ya que tendrá una visión más clara del entorno antes de tomar la decisión. Además, es posible que algunas secciones favorezcan cierto tipo de ataque, reduzcan la velocidad del personaje temporalmente o puedan provocar cierto estado al personaje que se encuentre encima.

Cada personaje tiene una barra de vida con un valor que difiere del resto. Cuando la barra de vida baja a 0, el personaje es eliminado. En combates individuales, gana el personaje que elimine al resto. En combates por equipos, gana el equipo que elimine a todos los personajes del equipo contrario.

Cada personaje tendrá habilidades de dos tipos: activas, que se traducirán en movimientos propios durante su turno, y pasivas, que afectarán de una manera u otra

durante el desarrollo del combate. Una habilidad activa podría ser un tipo determinado de ataque o la habilidad de recuperar vida utilizando un movimiento. Una habilidad pasiva, podría recuperar cierta cantidad de vida cada turno sin necesidad de utilizarla o fortalecer los ataques cuando la vida del personaje caiga por debajo del 50%, por ejemplo.

6.1.1.- Propósito del juego

Se busca conseguir un juego dinámico, con partidas rápidas, que enganche al jugador y le permita jugar en sus ratos libres, bien con el fin de mejorar su personaje y sus estrategias o de simplemente divertirse. Es un juego mayoritariamente casual (Arcade) y con vistas a convertirse en multijugador. La audiencia esperada será de jugadores casuales jóvenes e interesados en juegos de estrategia.

El juego contendrá cierta “violencia”, pues se trata de un sistema de combate, al fin y al cabo, aunque al combatir contra seres irreales y teniendo en cuenta el aspecto poco realista de los modelos *low-poly*, probablemente mantendría el criterio PEGI en torno a los 12 años, por lo que en principio este dato encajaría dentro del segmento de público esperado.

6.1.2.- Juegos similares

El modelo de combate podría llegar a ser considerado por el jugador como parecido al de juegos como *Fire Emblem* o *Final Fantasy Tactics*, aunque el hecho de manejar estrategias y no el comportamiento directo de los personajes provoca una variación increíble en lo que a jugabilidad se refiere. Se trata de un juego de estrategia donde no sólo tenemos que tener en cuenta las acciones del enemigo al realizar las nuestras, sino también las posibles acciones finales de cada personaje.

No existen muchos juegos en el mercado que utilicen los algoritmos evolutivos de forma tan explícita y dejando el resto de mecánicas como subordinadas al control de estos.

6.2.- Tema unificador

Al realizar el diseño de este juego, se ha tenido en cuenta como tema unificador el concepto de estrategia. Puede resultar confuso que se utilice la estrategia como núcleo duro del juego cuando este deberá colaborar con la pseudo-aleatoriedad de la que parten los algoritmos genéticos. No obstante, y según el modelo de cuatro formas interactivas de Burgun **[22]**, esta colaboración entre una toma de decisiones cuya situación actual es totalmente conocida, aunque la situación tras la decisión es incierta, clasifica nuestro videojuego dentro de la forma más completa de todas.

Por este motivo se decide utilizar un escenario de dos dimensiones con una cuadrícula y unos movimientos fácilmente identificables. En el momento de ejecutar la acción, la distancia hacia el rival, su estado y el tuyo deben ser datos conocidos casi por completo. La incertidumbre en la decisión corre entonces por cuenta del algoritmo genético que, dependiendo de la fiabilidad y la experiencia de nuestro personaje, podrá fallarnos en mayor o menor medida respecto a lo planeado. Por tanto, debemos centrar esfuerzos en implementar lo mejor posible el algoritmo genético, ya que de la fiabilidad de este dependerá la experiencia de juego.

6.3.- Personajes

Para el prototipo, se han desarrollado dos personajes similares de cara a equilibrar la lucha lo máximo posible de cara a las distintas pruebas que se vayan a realizar para favorecer el desarrollo. No obstante, la idea es permitir mayor variedad de personajes que hagan más divertida la jugabilidad y otorgue más posibilidades al jugador.

En un principio, se creará una lista de personajes definidos con un nombre, unos atributos y una serie de habilidades asignadas. De cara a un futuro se buscará permitir la creación y la personalización de personajes por parte de los jugadores. Como cualquier juego de rol, y más aun con vistas a ser multijugador, es necesario permitir y favorecer la *Player Story* **[21]**, es decir, permitir al jugador identificarse y crear una historia subjetiva alrededor de su personaje. Esto obliga a que cada personaje deba ser distinto al resto. Así, se podrán dar valores distintos para los puntos de vida, el factor de curación, el ataque, la defensa, el cansancio, la velocidad y demás atributos; además de dar distintas posibilidades en cuando a las acciones a realizar, distintas estrategias e incluso distinto comportamiento frente a estas estrategias.

También se pretende que los personajes que el jugador escoja sean bastante más débiles durante las primeras partidas, aumentando sus atributos y su capacidad de escoger la mejor posibilidad de ejecución de la estrategia con el paso de las partidas.

6.4.- Diseño de niveles

Los niveles tendrán gran peso visual y a nivel jugable durante la partida, por lo que un diseño profundo será requerido en un futuro. Los niveles serán escenarios rectangulares con una cuadrícula de n por m casillas donde los personajes podrán moverse. En algunas de estas cuadrículas podría haber potenciadores o reductores de alguno de los parámetros de los personajes. Ningún personaje puede salirse del rango de casillas predefinido.

La temática de los escenarios se decidirá con el proyecto más avanzado, aunque posiblemente haya varios niveles, cada uno con alguna característica distintiva respecto a los demás.

6.5.- Interfaz de usuario

La primera escena incluida en el juego es la interfaz dedicada a la pantalla de inicio, donde, a través de botones, se da opción al jugador de iniciar una partida (Un jugador o Versus), acceder al panel de opciones, o salir si lo desea.

La interfaz de usuario con mayor importancia será la visible durante las partidas. El jugador deberá conocer antes de elegir su estrategia, el estado de su personaje y su situación dentro del entorno en ese instante. Así, la interfaz tendrá que mostrar la cantidad de vida restante respecto a la vida total del personaje, la cantidad restante de escudo o puntos defensivos, los efectos positivos y/o negativos que tenga como consecuencia de alguna habilidad y su situación y la de el/los adversario/s sobre el escenario.

Además, la interfaz de usuario debe permitir la elección entre las distintas estrategias al inicio de cada turno y diferenciar correctamente estas opciones para evitar confusiones que pudieran alterar la experiencia de juego buscada.

6.6.- Mecánicas

A continuación, se detallan las reglas que gobiernan la interacción entre las entidades que forman parte del juego.

- La cámara se mantendrá sobre el escenario a una distancia concreta del centro (modelo 3D) para permitirle al jugador ver la situación completa. Permitirá el giro alrededor del centro del escenario utilizando el ratón.
- El juego se manejará con el ratón (PC) o con una pantalla táctil en dispositivos móviles si se diera el caso.
- Durante la partida, el jugador solo tendrá que elegir la estrategia que el personaje va a seguir en cada turno pulsando sobre uno de los botones en pantalla.
- Los personajes tendrán habilidades activas, que tendrán que ser utilizadas durante su turno y habilidades pasivas, que serán propias del personaje y tendrán efecto durante toda la partida.
- Las habilidades activas pueden ser de varios tipos (Ataque, defensa, movimiento…) afectando de diferente manera al entorno personaje y su entorno al utilizarse.
- La partida se divide en etapas. Una etapa está compuesta por la secuencia de turnos de todos los personajes activos en la partida.
- El orden para la ejecución de los turnos de cada etapa depende de la velocidad de los personajes. El más rápido ejecutará su turno primero.
- Cada personaje tendrá un cansancio que determinará el número máximo de acciones que pueden llegar a ser ejecutadas durante su turno antes de que acabe.
- Algunas habilidades solo afectarán a un rango determinado de cuadrículas dependientes de la posición desde las que son ejecutadas.
- Para el cálculo del daño que efectúa la habilidad dependerá el daño propio de la habilidad, el ataque del personaje que la efectúa y la defensa del que recibe el ataque.

- En caso de que un personaje tenga puntos de escudo al recibir el daño, este no le afectará directamente a la vida, sino que primero le eliminará los puntos de escudo para posteriormente reducir la vida.
- Los puntos de escudo se reinician a 0 al empezar el turno.
- Algunas habilidades no son compatibles con algunas estrategias, por lo que no estarán disponibles para estas.
- Algunas habilidades potenciarán los atributos del personaje y otras disminuirán los del rival. Algunas zonas del escenario también podrán tener este tipo de efectos.
- En algunas ocasiones, se establecerá un número máximo de etapas de duración del combate al término de las cuales ganará el personaje con más vida tras la última etapa.

6.7.- Criterios de éxito y fracaso

Como la jugabilidad se basa en la disputa entre varios jugadores en partidas, tenemos que definir cuál es realmente el final de la partida. Para esto, tenemos que saber cuándo gana uno de los dos personajes.

En este caso, dicha decisión de diseño es muy sencilla: ganará el personaje que reduzca antes la vida de el/los otros a 0. Por tanto, los criterios, tanto de éxito como de fracaso, depende del lado por el que se miren. Cuando hay uno o varios ganadores, tiene que haber uno o varios vencidos, sean manejados por humanos o por una CPU.

Hay un caso específico que no se ha planeado implementar en el prototipo que es el caso de los combates con límites de etapas. Esto significa que solo se realizarán un número determinado de turnos durante el combate. En ese tipo de combates, es posible que alguno de los dos rivales no vea su vida reducida por debajo de 0, por lo que habría que elegir otro criterio. Aquí, ganará el combate el personaje con mayor vida al término de la última etapa en el caso de que ninguno de los adversarios hubiese sido derrotado antes.

Criterios de éxito

+ Eliminar a el/los rival/es (Reduciendo su vida a 0).
+ En combates individuales con límite de etapas, tener más vida que el rival al final de la partida.

Criterios de fracaso

— La vida del personaje queda reducida a 0.
— En combates individuales con límite de etapas, el rival tiene más vida al final de la partida.

6.8.- Flowboard

El flowboard es considerado como uno de los diagramas que mejor documentan la estructura de un videojuego. Es una combinación entre un diagrama de flujo y un *storyboard* y trata de representar, de forma simplificada, los posibles canales de comunicación entre las pantallas que tendrá el videojuego.

En la Ilustración 10, podemos ver el flowboard que se ha realizado para el videojuego que estamos desarrollando, y puede servir como una buena muestra de las pantallas que necesitará el juego para funcionar.

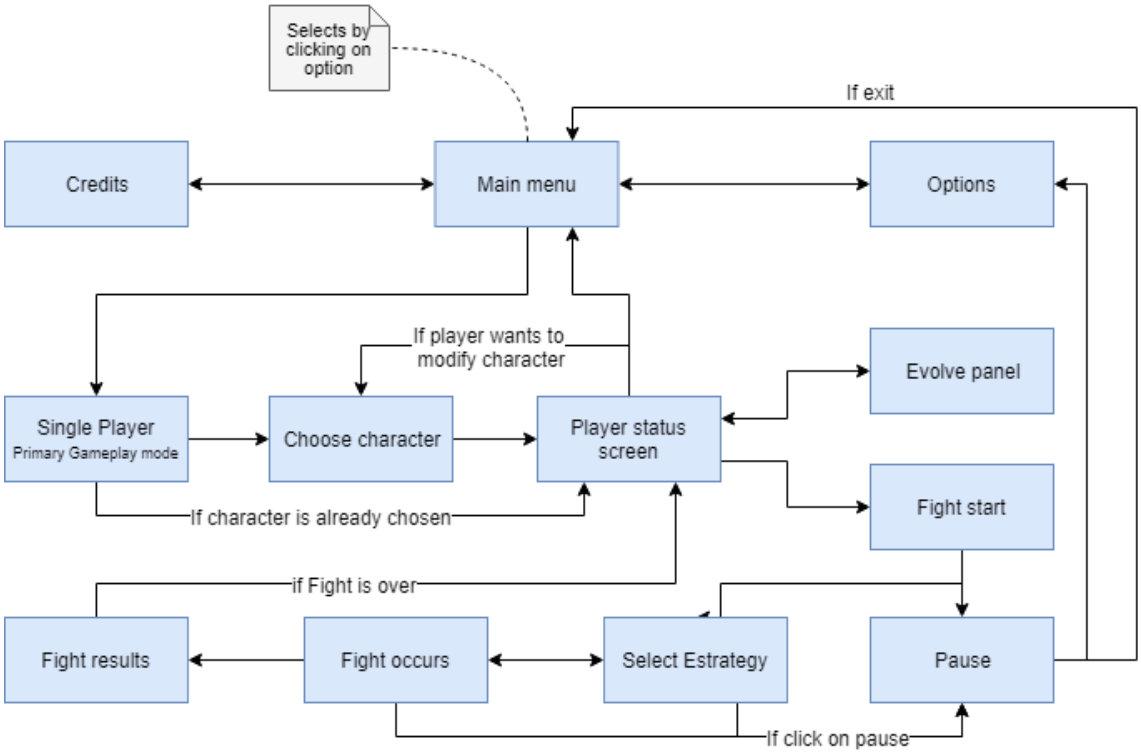


Ilustración 10: *Flowboard para el juego desarrollado*

Como podemos ver en el Flowboard, el menú principal debería permitirnos llegar hasta pantallas: Pantalla de opciones, créditos y pantalla de partida de un jugador. Antes de comenzar la partida, deberíamos poder escoger personaje, ver sus atributos, evolucionarlo y acceder a la lucha. En la partida, y hasta que esta acabe, los jugadores deberán elegir estrategia y esperar hasta que vuelva a ser su turno.

7.- ANÁLISIS DEL PROYECTO

Durante el anterior capítulo nos hemos dedicado a realizar el diseño del producto como videojuego y a definir su jugabilidad y otros aspectos que se consideran esenciales en el diseño de videojuegos. Este análisis previo podríamos considerarlo como lo que el "cliente" espera como resultado de nuestro desarrollo. Es decir, lo documentado anteriormente da las directrices de lo que se pretende conseguir.

El hecho de que el diseño del videojuego ha sido realizado por la misma persona que se va a dedicar a las tareas de desarrollo, nos ahorra tareas laboriosas como estudiar el ámbito, la terminología, la situación o los objetivos del proyecto. Además, no corremos el riesgo de desarrollar algo distinto a lo que se pretendía en ese estudio previo. Si el diseño del videojuego hubiese sido realizado por otro equipo distinto al de desarrollo, algo de lo más habitual, entonces sí necesitaríamos una etapa de análisis más extensa para entender el documento de diseño de videojuego y eliminar las posibles ambigüedades que pudieran surgir.

7.1.- Definición Inicial de Requisitos

Un requisito es una condición o capacidad que debe exhibir o poseer un sistema para satisfacer un contrato, estándar, especificación u otra documentación formalmente impuesta. En este caso, los requisitos deben amoldarse a la experiencia de juego pretendida. A continuación, realizamos una definición de los requisitos que hemos encontrado tras el estudio previo de lo que debe hacer el software atendiendo a las mecánicas anteriormente desarrolladas.

7.1.1.- Requisitos funcionales

- El sistema permitirá al jugador elegir una estrategia cuando sea el turno de su personaje
    - Permitir al jugador que elija entre varias estrategias durante su turno para indicar al personaje qué debe hacer. Estas estrategias deben llamar a un algoritmo evolutivo que, atendiendo a una serie de variables, dirija el comportamiento del personaje durante su turno.
- El sistema integrará correctamente las condiciones de éxito y fracaso y terminará la partida cuando alguna de estas se cumpla.

    - Aplicar unas condiciones de satisfacción a la partida, de manera que sea posible distinguir entre ganador y perdedor para cada partida.

- El sistema permitirá partidas entre dos jugadores humanos, cada uno controlando a un personaje.
    - Se dará la posibilidad de jugar partidas en multijugador local entre dos jugadores humanos, uno contra otro.
- El sistema implementará variedad de acciones que permitan una jugabilidad variada.
    - Para que la jugabilidad sea variada y se base realmente en la estrategia, es necesario que haya cierta variedad de acciones que permitan esto.
- El sistema permitirá partidas para un solo jugador
    - Se deberá poder jugar únicamente un jugador contra otro controlado por la CPU.
- El sistema debe permitir la persistencia de acciones y personajes.
    - Tanto las acciones como los personajes deben poder ser guardados y leídos desde un fichero externo.
- Tras la elección de estrategia, el personaje deberá realizar movimientos con cierto sentido atendiendo a la estrategia elegida.
    - Para que la experiencia de juego sea correcta, es necesario que el jugador sienta que el personaje tiene en cuenta la estrategia que él ha elegido. Si el personaje “no le hace caso”, la partida se decantaría por aleatoriedad, dejando de ser un juego de estrategia.

7.1.2.- Requisitos no funcionales

- La ejecución del algoritmo genético debe ser relativamente rápida, siendo casi invisible para el jugador la ejecución de este.
    - El jugador no debería quedarse esperando a que el personaje actúe después de tocar el botón de estrategia. Esta acción debería ser casi inmediata para cumplir con el ritmo del juego.
- El videojuego deberá poder ser ejecutado en ordenador y/o dispositivos *Android*.
    - Se creará un ejecutable que funcione en al menos un tipo de los dispositivos mencionados sin tener en cuenta las características de estos.

7.2.- Modelo de clases conceptuales

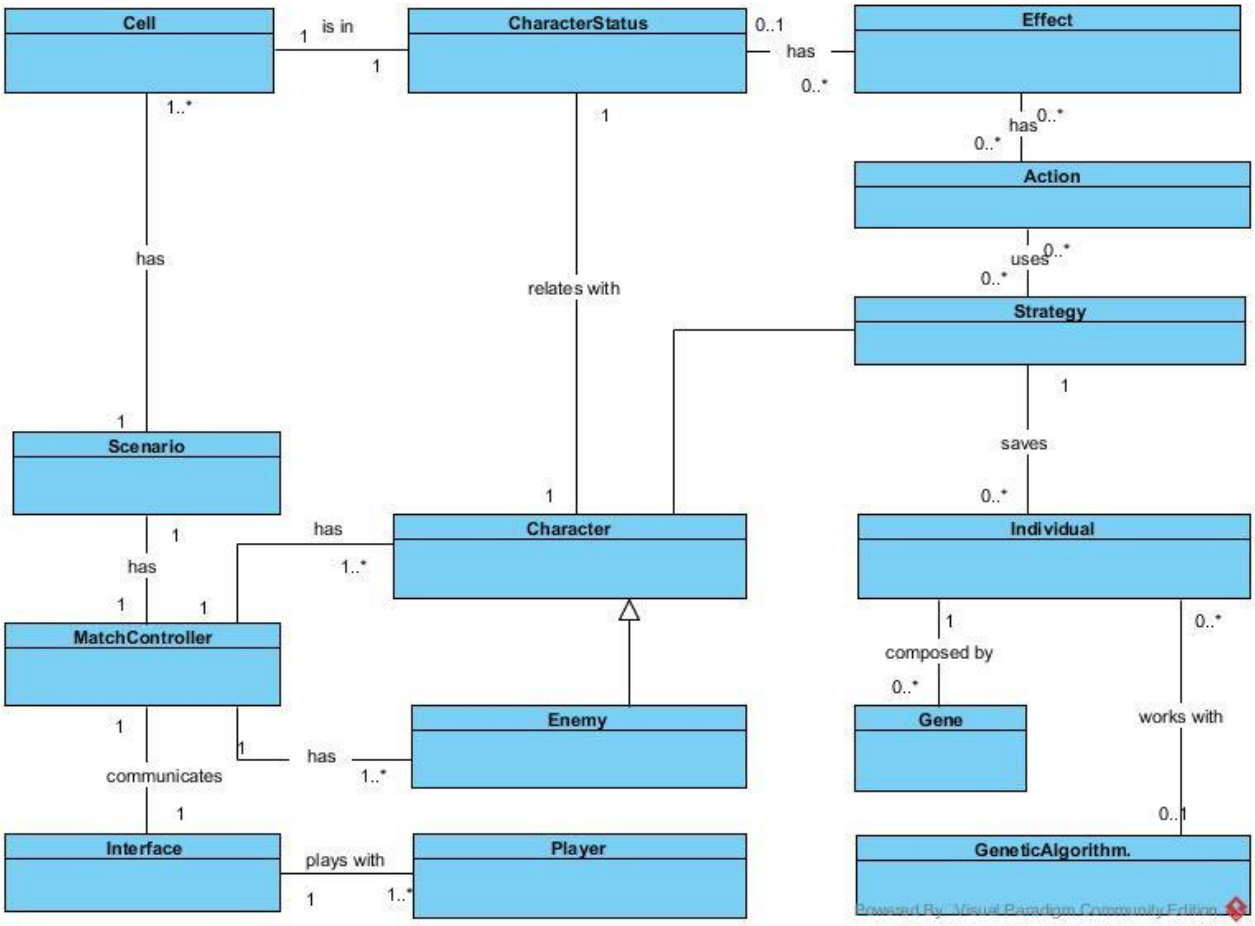


Ilustración 11: *diagrama de clases conceptuales*

7.3.- Planificación de costes

Vamos a trabajar sobre un proyecto de desarrollo independiente, Esto quiere decir que no tenemos un presupuesto inicial demasiado alto acorde a las cifras que se mueven en la industria. Por tanto, es necesario ajustar correctamente el presupuesto desde el principio.

Este prototipo va a ser realizado por un único desarrollador, que trabajará desde el mes de febrero hasta junio a razón de 5 días a la semana de lunes a viernes, 4 horas cada día. Dado que la experiencia del desarrollador en proyectos de este tipo es baja, se podría considerar un sueldo de 729€ al mes. Además, hay que contar con el desgaste del equipo informático sobre el que se va a desarrollar. Teniendo en cuenta que un equipo correcto para el desarrollo de este tipo de proyectos puede costar unos 600€, con una vida media de 3-4 años, o lo que es lo mismo, 42 meses, el gasto por el equipo será unos 15 euros cada mes. Incluimos otros 15€ para gastos de luz y material de papelería. En total, 759€ cada mes.

Si tenemos en cuenta que usaremos el plan gratuito de Unity3D y que, de necesitar algún elemento de la tienda de Unity, buscaremos que este sea gratuito, podemos cerrar el costo planificado en 759€ * 4meses = **3036€.**

Esto quiere decir que necesitaríamos una inversión inicial de esa cantidad como mínimo para hacer frente económicamente al proyecto de desarrollo del prototipo.

8.- DISEÑO DEL SOFTWARE

Como ya hemos dicho anteriormente, la elección de estrategia es el tema central de nuestro juego, y, a su vez, el algoritmo genético será el encargado de que estas elecciones tengan realmente peso en el videojuego que estamos desarrollando.

Es necesario hacer un diseño realmente bueno del algoritmo genético, ya que su funcionamiento puede ser el más determinante y a la vez inesperado dentro del juego. Una mala implementación del algoritmo genético es un riesgo para el proyecto, pues si al final la ejecución de este fuese muy lenta o no alcanzase soluciones medianamente válidas para la estrategia planteada, las probabilidades de que el juego terminase en fracaso aumentarían terriblemente.

Es por esto que comenzaremos con el diseño del algoritmo genético de la manera en la que estudiamos en capítulos anteriores.

8.1.- Diseño del algoritmo genético

Para el diseño del algoritmo genético, vamos a seguir los pasos que comentamos durante el estudio previo de los pasos a tomar para la construcción de estos.

8.1.1.- Diseño de la representación

En primer lugar, definiremos la representación de los individuos de la solución y trataremos de justificarla. Este probablemente será el paso más difícil y a la vez más importante, ya que basaremos el resto del diseño en las decisiones que tomemos aquí.

Una vez elegida una estrategia para un personaje, que tiene un número finito de acciones posibles, el algoritmo genético tiene que encargarse de aportar una solución relativamente válida atendiendo a la situación propia del personaje y del medio en el que se encuentra **[Ilustración 12]**. Por tanto y a grandes rasgos, el individuo o solución será una lista con las acciones que debe realizar el personaje, definiendo también de alguna forma el orden en el que debe realizarlas. De aquí obtenemos dos preguntas: Cómo se definirán dichas acciones y cómo vendrá definido el orden de realización para el personaje.

Para responder a la primera pregunta, tenemos que conocer qué necesitamos para representar cada acción: Qué habilidad realizará, cuánto cansancio consume, cuál es su rango de efecto, cuál es la posición del jugador y en qué dirección de entre las 4 posibles

la va a realizar. Asumiendo que el algoritmo genético se ejecuta para el personaje, que a su vez tendrá un listado de habilidades disponibles con parámetros que suponemos fijos, sólo nos faltaría por conocer la dirección en la que se realizará la acción. Así, cada alelo podría ser una dupla con dos parámetros: la posición en la lista de habilidades del personaje y la dirección en la que se realizará la acción **[Ilustración 13]**.

Ilustración 12: *Dibujo básico de posibles direcciones en las que se realizarán las acciones.*

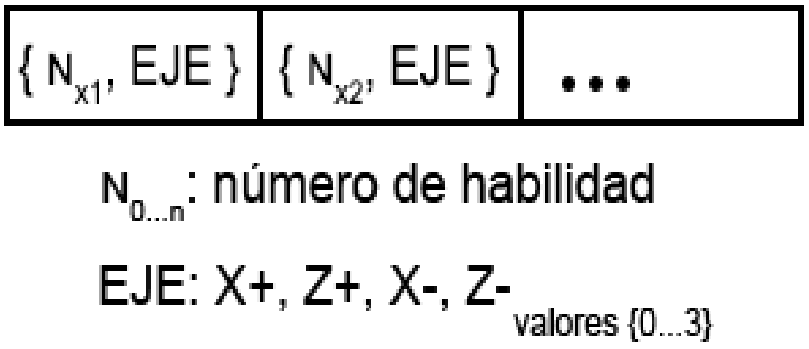


Ilustración 13: *Representación básica de cada gen del individuo.*

Partiendo de esta representación, tomaremos la respuesta más trivial a la decisión correspondiente al orden en el que se ejecutan las acciones del individuo. En este caso, el personaje seguirá la secuencia de acciones según vienen representadas en el cromosoma.

8.1.2.- Definir como inicializar la población

La manera más sencilla de inicializar la población de individuos es conseguir una secuencia aleatoria de habilidades compatibles con la estrategia elegida y posteriormente darles un valor aleatorio entre 0 y 3 que defina la dirección en la que va a realizar la estrategia.

Para aportar cierto aprendizaje al proceso, el personaje tendrá, para cada estrategia, un registro con las mejores soluciones encontradas en ejecuciones anteriores. Estas soluciones se incluirán a la población inicial para favorecer la exploración por regiones que dieron buenos resultados en otros contextos. Es tarea de iteraciones futuras conocer el número de individuos que se añadirán.

8.1.3.- Definir la función de adaptación

La adaptación del individuo al medio dependerá de la estrategia escogida por el jugador en ese turno, pudiendo variar por completo los criterios. Por ejemplo, para una estrategia de ataque, el fenotipo será el daño que acaba realizando la secuencia de acciones de acuerdo a la simulación, mientras que, para una defensa, el objetivo puede ser reducir posibles daños, alejarse del enemigo e incluso recuperar vida.

Para este prototipo, no definiremos el monto total de estrategias que podrían existir en el videojuego, sino tan solo los tipos suficientes para realizar pruebas tanto del algoritmo como de la jugabilidad que nos permitan obtener datos concluyentes al respecto.

8.1.4.- Definir la estrategia de selección de padres

Debido a la cantidad de posibilidades que tiene el personaje para decidir las acciones a realizar, la secuencia en la que las realiza y los 4 ejes en los que puede realizarlas, es fácil tener muchas soluciones totalmente “inútiles”. Esto, sumado a que estamos trabajando sobre un videojuego y las decisiones se deben tomar con relativa fluidez, nos obliga de alguna forma a converger rápidamente hacia óptimos locales y explotar el espacio de búsqueda cercano a ellos.

Por ejemplo, en una estrategia de ataque, es posible que muchas de las soluciones nos lleven a ni siquiera alcanzar al enemigo. Si en las primeras generaciones se encuentra algún individuo en el que se provoque cierto daño sobre el enemigo, eso quiere decir que el enemigo está en esa dirección y esa es una posible forma de hacerle daño.

Por todo esto, parece conveniente utilizar una estrategia con alta presión selectiva y en la que podamos utilizar elitismo entre generaciones. Teniendo en cuenta que es probable que en muchos individuos la adaptación pueda ser 0 según la estrategia (en una estrategia de ataque, por ejemplo, es fácil que en muchos individuos de la población inicial ni siquiera se llegue a alcanzar al enemigo). Probaremos una selección por torneo sobre la que estudiaremos la variación del factor tamaño del torneo..

8.1.5.- Diseño del operador de mutación

Implementaremos los siguientes operadores de mutación:

- **Mutación sobre la habilidad** – Cambiar el número que marca el índice de la habilidad por otro válido de manera aleatoria y comprobar que no se sobrepase el cansancio.
- **Mutación sobre la dirección** – Cambiar el número que marca la dirección en la que se va a ejecutar la acción por otro aleatorio entre 0 y 3.

Dada la necesidad de alcanzar rápidamente buenas soluciones, el operador de mutación deberá tener una probabilidad de ocurrencia baja.

8.1.6.- Diseño el operador de cruce

El cruce entre padres será una operación trivial dada la representación que hemos escogido. Se escogerán una o varias acciones (duplas) de cada padre y se intercambiarán entre ellas. De ahí obtenemos dos hijos en los cuales tendremos que comprobar que su cansancio se mantiene dentro del rango válido. Para el caso concreto, diseñamos tres operadores de cruce:

- Reemplazando los n primeros cromosomas de cada padre, siendo n un número aleatorio entre 1 y la mitad del tamaño del cromosoma.
- Reemplazando los n últimos cromosomas de cada padre, siendo n un número aleatorio entre 1 y la mitad del tamaño del cromosoma.
- Reemplazando los n primeros cromosomas de un padre por los m últimos de otro, siendo n y m un número aleatorio entre 1 y la mitad del tamaño del cromosoma en cada caso.

8.1.7.- Diseño de la estrategia de reemplazamiento

En un principio, y tras lo descrito anteriormente, parece mejor optar por una estrategia de reemplazamiento generacional que mantenga el elitismo. En este caso, se mantendrá una porción de la población en la siguiente generación, mientras que el resto serán eliminadas, dando paso a los hijos que surjan de la reproducción.

8.1.8.- Diseño de la condición de parada

Para la condición de parada utilizaremos dos condiciones. La primera y más fácil de implementar es definiendo un número máximo de iteraciones que, cuando llegue el algoritmo, terminará su ejecución. Esto nos permite controlar la velocidad que pueda tener al ejecutarse. Para mejorar la eficiencia del proceso, añadiremos también como condición de parada un número máximo de iteraciones sin cambios. Esto permite que, de darse la solución óptima en generaciones tempranas, no tengamos que llegar hasta el final, acelerando el proceso.

8.2.- Diseño de la elección de estrategia

Una vez que hemos hablado de la forma en la que van a ejecutarse las estrategias. Para la ejecución del algoritmo genético, tendremos una clase *GeneticAlgorithm* que almacenará los datos necesarios para su ejecución teniendo en cuenta la situación actual: estado del personaje, estado del rival y estrategia a la que deben adaptarse los individuos; y los métodos necesarios para crear la población inicial de individuos, reproducirla correctamente para obtener nuevas generaciones, implementar las condiciones de parada y obtener el mejor individuo al final de la ejecución.

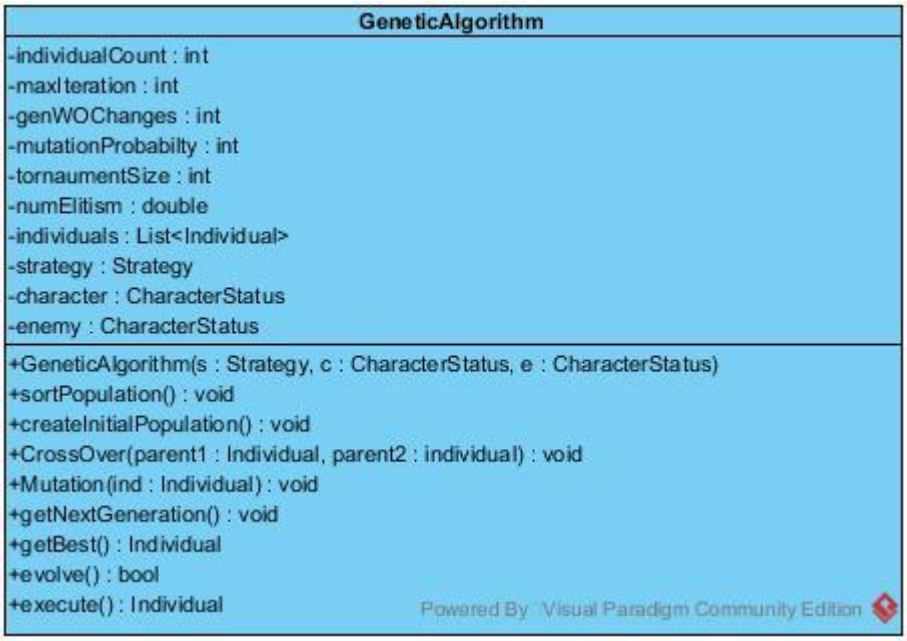


Ilustración 12: *diagrama de clase GenethicAlgorithm*

El evento que llame a ejecutarse a esta clase debe ser la elección de estrategia por parte del jugador. El cómo se hace esto debería venir dado por el diseño de interfaz. Como cada algoritmo genético ha de valorar situaciones diferentes para estrategias que pueden ser diferentes, Necesitaremos una clase que, con acceso a los datos de los personajes y del escenario, cree y ejecute un objeto de la clase Algoritmo genético y pase el mejor individuo generado al personaje al que le toca su turno para que lo ejecute y muestre el resultado final por pantalla.

Para describir de manera más formal y detallar aún más un proceso tan importante como este, se ha realizado un diagrama de secuencia con las acciones que ocurren por orden desde que el jugador indica la estrategia a seguir hasta que su personaje ha terminado de actuar. **[Ilustración 15]**

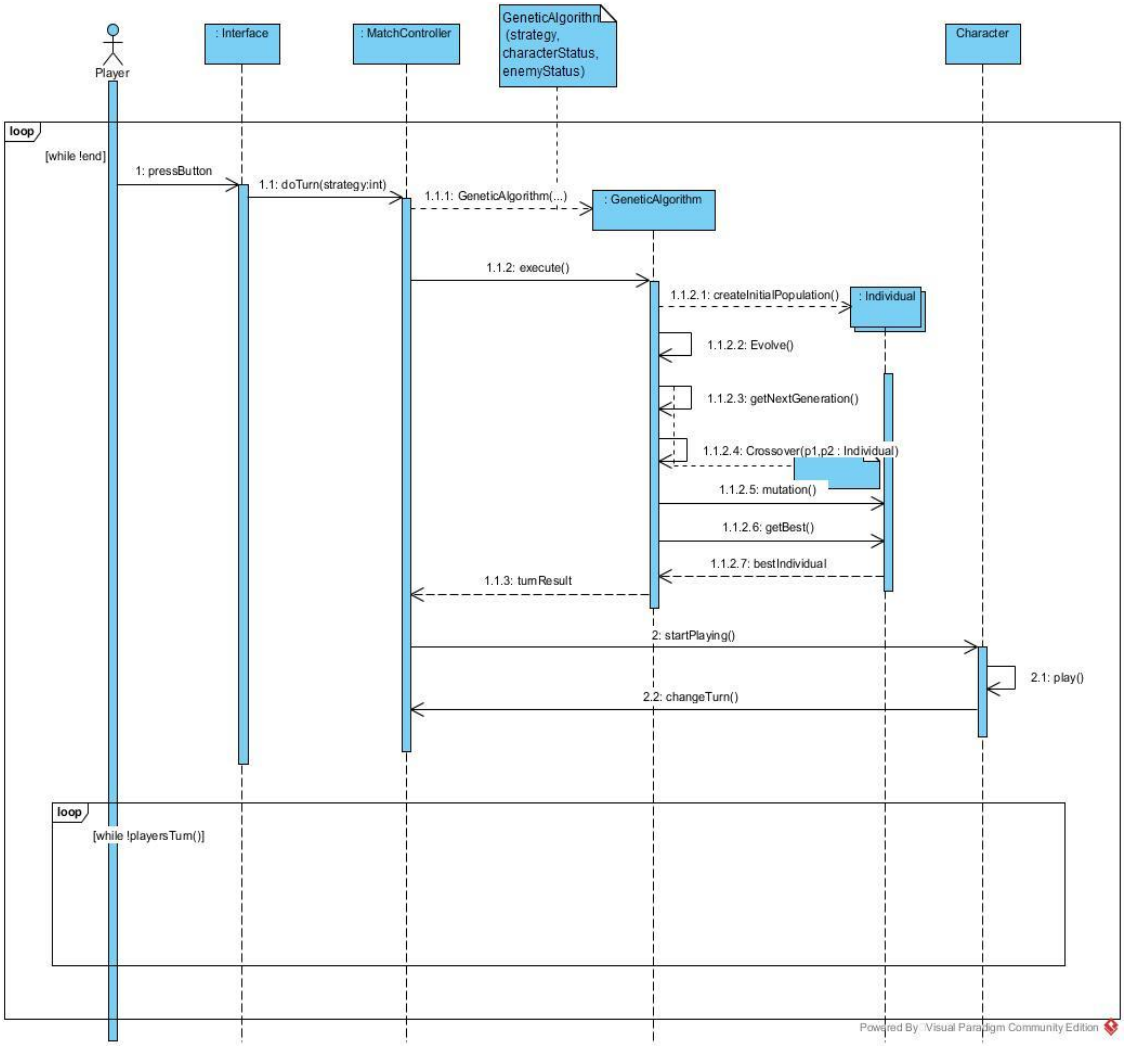


Ilustración 13: *Diagrama de secuencia que representa lo que ocurre desde que el jugador elije estrategia hasta que esta se ejecuta.*

Como podemos observar, igual que el controlador de partida (*MatchController*) podría ser considerado como una factoría de estrategias, pues es el único encargado de crear y ejecutar los objetos de la clase *GeneticAlgorithm,* a su vez podríamos considerar al Algoritmo genético como factoría de Individuos (*Individual*), que, a su vez, deberán ser factoría de Genes para conformar el cromosoma

Es importante destacar que el jugador solo tiene que interactuar con el videojuego al principio de su turno, para elegir la estrategia. Esto puede resultar un sistema de combate aburrido para algunos jugadores, por lo que hemos de tratar de hacer la ejecución de los turnos rápida y amena, con efectos visuales que puedan atraer los suficiente a los jugadores.

8.3.- Diagrama de clases de diseño

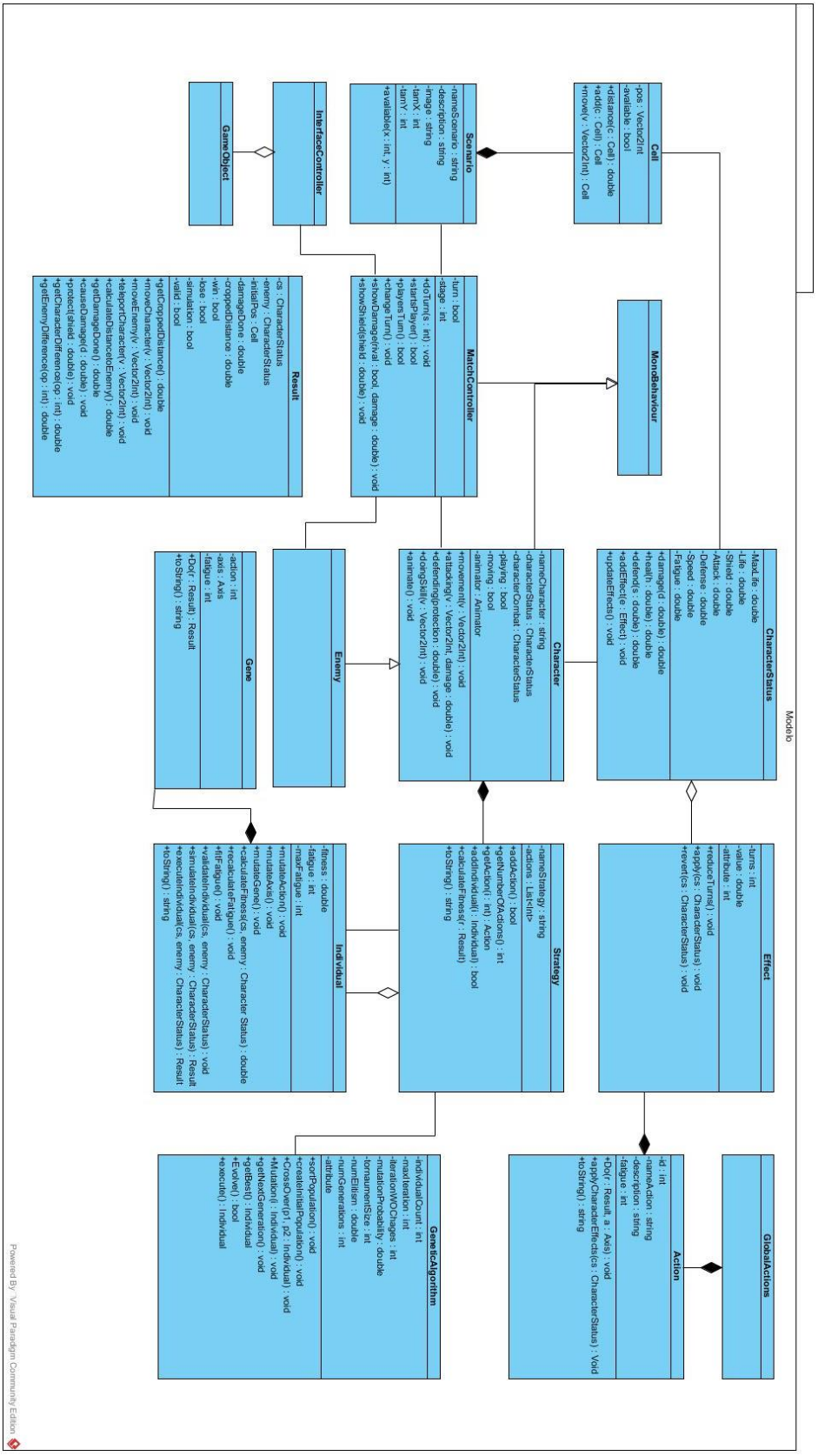


Ilustración 14: *Modelo del dominio del software*

8.4.- Diseño de las acciones

Dentro del modelo de dominio, podemos ver como existe una clase *GlobalActions* que actúa como contenedor de objetos de tipo *Action*. Esta decisión de diseño se ha tomado con la idea de facilitar la persistencia posteriormente, ya que esta clase contendrá todas las acciones posibles en un diccionario, de manera que las estrategias solo tendrán una referencia a la ID de la acción, en vez de a la acción por completo.

Con esto favorecemos la cohesión entre las acciones. Por ejemplo, si decidiésemos modificar el cansancio de una acción para equilibrar el juego, solo tendríamos que modificarla en *GlobalActions*, en vez de en cada estrategia de cada personaje que pueda tener esa acción entre sus posibilidades.

Tomando eso como referencia, solo queda estructurar la clase *Action* para que permita almacenar gran cantidad y variedad de acciones, tal y como se especifica en uno de los requisitos comentados durante el análisis. Para esto, se ha decidido implementar la clase *Action* como una clase abstracta de la que heredarán tipos de acciones más específicos. **[Ilustración 17]**

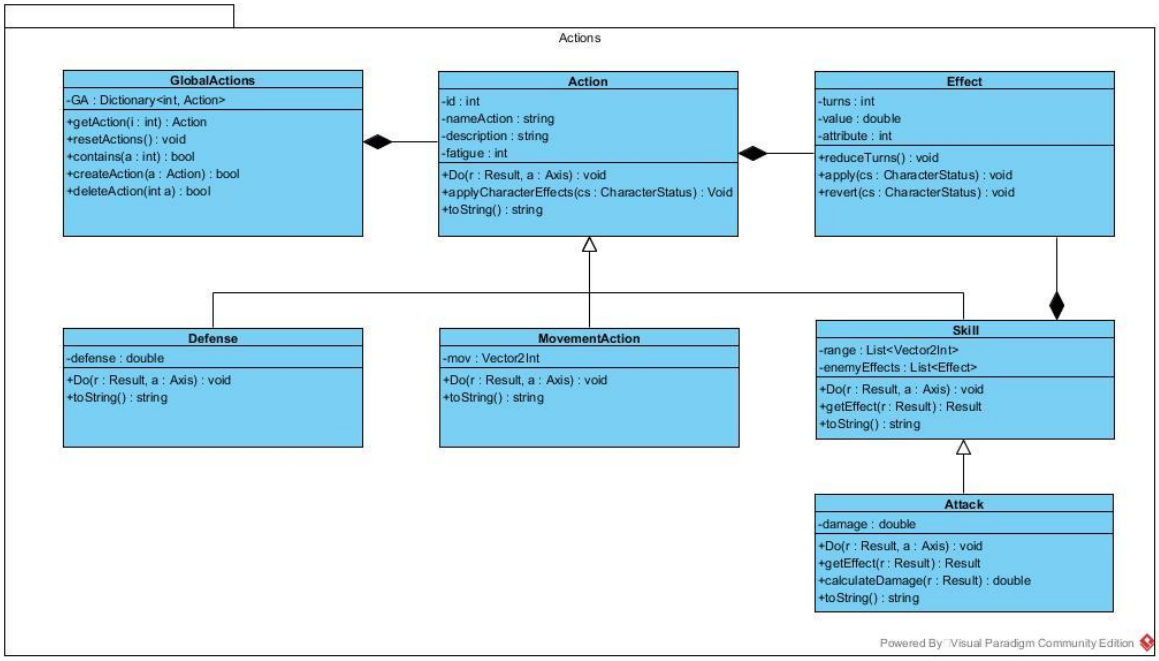


Ilustración 15: *Modelo del dominio ampliado para las acciones*

8.5.- Diseño de la estrategia

Otra decisión de gran importancia es cómo encarar el diseño de las estrategias. Por muy bueno que sea el algoritmo genético y los operadores que lo componen, de la estrategia depende que las acciones tengan el sentido que el jugador esperaba cuando seleccionó una u otra estrategia.

Cada personaje tendrá un conjunto de instancias de la clase 'Estrategia'. En cada una de estas instancias, tendremos una lista de las posibles acciones que puede realizar el personaje. Como ya comentamos antes, no almacenará la referencia a la acción, sino una lista con los Identificadores almacenados como enteros en *GlobalActions.*

La estrategia también será la encargada de calcular la adaptación de cada individuo, ya que dependiendo del objetivo que se busque en esta un individuo puede ser malo o bueno. Esto lo implementaremos utilizando varias clases Estrategia que hereden de la interfaz Estrategia e implementen su método abstracto para el cálculo de la adaptación de un individuo.

Para esto, crearemos una clase abstracta *Strategy* y utilizaremos la herencia para sobrecargar el método encargado de calcular la adaptación del individuo al medio. Esto quiere decir que tendremos una clase por cada tipo de estrategia que queramos. **[Ilustración 18]**

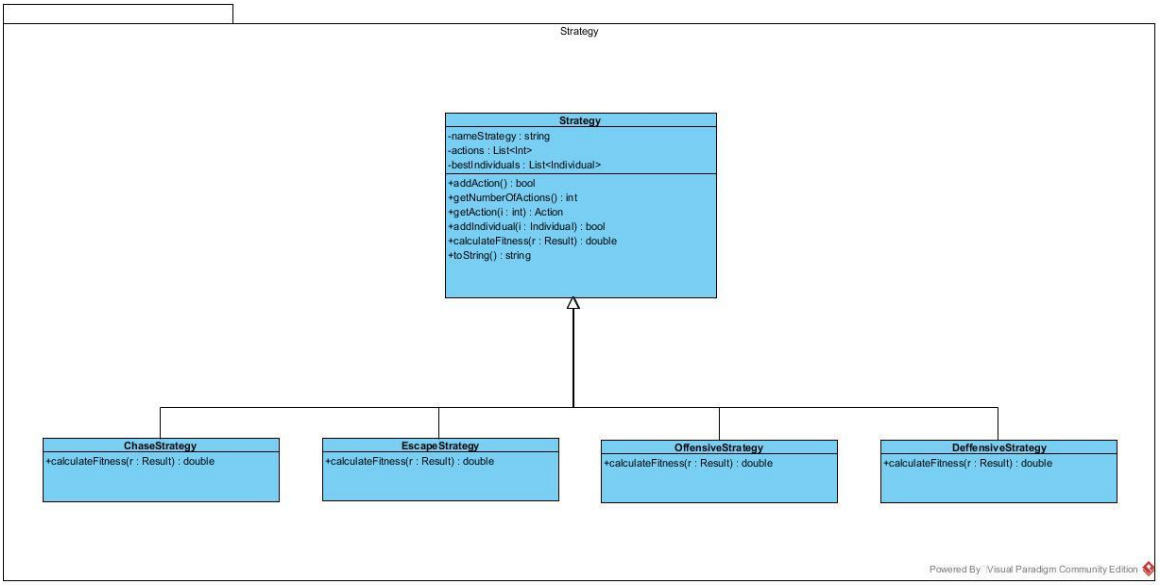


Ilustración 16: *Modelo de clases representativo para el diseño de estrategias*

9.- IMPLEMENTACIÓN

Como ya se ha comentado anteriormente, la implementación del videojuego se realizará sobre el entorno *Unity3D*, con lenguaje de programación orientada a objetos C#.

Para la implementación, comenzamos representando de forma muy básica la escena, todo de acuerdo a los prototipos que se mostraron en anteriores capítulos. Un cuadrado por cada casilla en la escena y dos cilindros representando a cada personaje. También incluiremos un botón como interfaz para dar comienzo a la ejecución del algoritmo genético. A partir de eso, comenzaríamos la creación de scripts en C#.

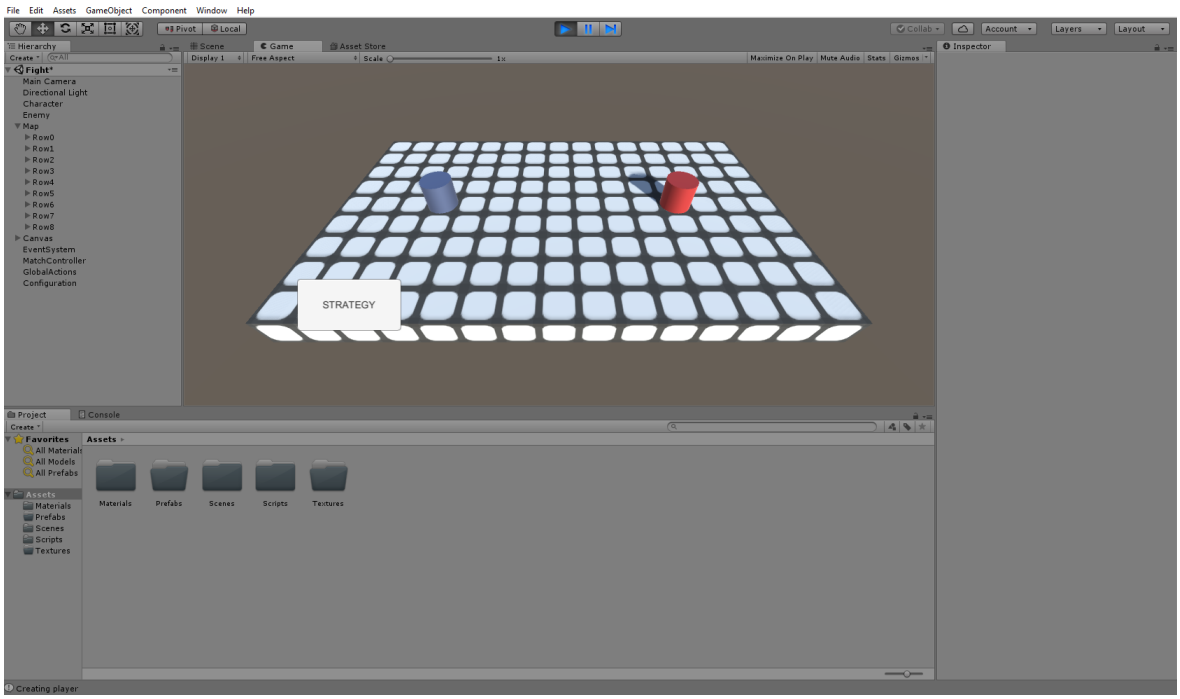


Ilustración 17: *Primeros pasos del prototipo. Inicio de la escena.*

9.1.- Implementación del algoritmo genético

Al igual que como hicimos en el diseño, comenzaremos esta etapa trabajando en el algoritmo genético, tirando de ese hilo para producir todas las clases necesarias (y ya diseñadas) para poder ejecutarlo y comprobar que produce el resultado esperado.

Para facilitar el comienzo de esta etapa, utilizaremos como acción el movimiento hacia una casilla adyacente, con una única estrategia implementada,

que buscará acercar la posición del personaje azul a la del rojo. Evitaremos mover al adversario hasta que podamos comprobar que el genético funciona correctamente. Una vez que el algoritmo genético se ejecute, se mostrará por pantalla la nueva posición de los personajes y se sacará por consola los movimientos realizados.

9.1.1.- Atributos

La clase algoritmo genético almacena todos los parámetros necesarios para controlar su ejecución. Estos son:

- Número de individuos
  - El número de individuos que tendrá la población en cada generación.
- Número máximo de iteraciones
- Número máximo de iteraciones sin mejora
- Probabilidad de mutación
  - Entero de 0 a 100.
- Tamaño del torneo
  - Tamaño del subconjunto de padres escogido entre los que se realizará el torneo. A mayor tamaño del torneo, mayor velocidad de convergencia.
- Número de elitismo
  - Número de individuos que serán afectados por el elitismo, es decir, que pasarán directamente entre generaciones sin cruce ni mutación.

Además, también ha sido necesario incluirle una lista de individuos que representan la generación actual, otra lista para representar los individuos que pasan a la siguiente generación, una referencia a la estrategia, al estado del personaje y del enemigo respecto a los que se ejecuta y una instancia de la clase *StreamWriter* de C# **[19]** utilizada para escribir sobre ficheros el resultado de cada ejecución (Muy útil para futuras pruebas).

9.1.2.- Métodos

La clase *GenethicAlgorithm* tiene un constructor parametrizado al que se le pasa la estrategia elegida **[Ilustración 20]** y la situación del personaje y el enemigo. Los parámetros que manejan el algoritmo se cargan desde una clase estática *Configuration* de la que hablaremos más tarde. Se inicializan el resto de atributos.

```
public GeneticAlgorithm ( Strategy _strategy, CharacterStatus _character, CharacterStatus _enemy ) {

    strategy            = _strategy;
    character           = _character;
    enemy               = _enemy;

    individualCount     = Configuration.individualCount;
    maxIteration        = Configuration.maxIteration;
    genWOChanges        = Configuration.genWOChanges;
    mutationProbability = Configuration.mutationProbability;
    tornaumentSize      = Configuration.tornaumentSize;
    numElitism          = Configuration.numElitism;

    numGenerations      = 0;

    individuals         = new List<Individual> ();
    nextGeneration      = new List<Individual> ();

    sw                  = new StreamWriter ("genetico.txt");
}
```

Ilustración 18: *Constructor parametrizado de la clase Algoritmo Genético*

Otro método de gran importancia es el método *execute()* ya que será el método desde el que se llamará a la ejecución del algoritmo genético, sin necesidad de interacción externa con el resto de operadores.

```
//MAIN GENETIC ALG. FUNCTION
//Invokes the other parts of the algorithm
public Individual execute () {
    CreateInitialPopulation ();
    geneticToString ();
    Evolve ();

    strategy.addIndividual (getBest ());
    geneticToStringEnd ();
    sw.Close ();
    return getBest ();
}
```

Ilustración 19: *Método* execute() *del algoritmo genético. Realiza la llamada al resto de operadores.*

El resto de operaciones (Generación de población, cruce y mutación). se realizan de acuerdo al diseño establecido anteriormente. Para conocer tanto los operadores de cruce, que trabajan con el cromosoma del individuo, como los operadores de mutación, que realizan llamadas a métodos de la clase Individuo, vamos a estudiar un poco el resultado obtenido de la implementación de la clase *Individual*.

9.1.3.- Implementación de los individuos

A la hora de la implementación del individuo, lo más difícil fue tener en cuenta que trabajamos con cromosomas de longitud variable. Esto requirió un mayor esfuerzo de cómputo a la hora de realizar algunas operaciones, pues se incrementaba el riesgo de tratar de acceder a posiciones no existentes en el array.

Además, por cada inserción en el cromosoma, es necesario validar, primero que su cansancio no es mayor que el cansancio máximo del personaje concreto para ese turno, elegir una estrategia para “arreglarlo” en caso de que no lo fuera, y después obtener y validar el valor de adaptación para cada individuo **[Ilustración 22].**

```
public void recalculateFatigue(){
    fatigue = 0;
    for (int i = 0; i < chromosome.Count; i++) {
        fatigue += chromosome [i].getFatigue ();
    }
    this.fitFatigue();
}

public bool fitFatigue(){
    int excess = fatigue - maxFatigue;

    if ( excess <= 0 ) return true;
    else {
        while( excess > 0 ) {
            for (int i = chromosome.Count-1; i >= 0; i--) {
                if ( chromosome[i].getFatigue() == excess) {
                    chromosome.RemoveAt (i);
                    fatigue = maxFatigue;
                    return true;
                }
            }
            fatigue -= chromosome [chromosome.Count - 1].getFatigue();
            chromosome.RemoveAt (chromosome.Count - 1);
            excess = fatigue - maxFatigue;
        }
        return true;
    }
}
```

Ilustración 20: *Cálculo y validación del cansancio del individuo.*

La intención con la función *fitFatigue()* es eliminar del cromosoma el número mínimo de genes posible hasta que el cansancio del individuo sea válido.

La clase *Gene*, representada en la ilustración 23, contiene cada valor que en conjunto con otra forma el cromosoma, contiene, como se dijo durante el diseño, la acción que realiza, almacenada como un entero que referencia al ID de la acción en la clase estática G*lobalActions; el eje en el que va a realizarla y la fatigue que consume dicha acción.*

```
public class Gene {

    protected int action;
    protected axis axis;
    protected int fatigue;
```

Ilustración 21: *atributos de la clase que representa el Gen.*

9.1.4.- La clase *Result*

Una clase no relacionada directamente con el algoritmo genético, pero muy importante para la correcta ejecución de este, es la clase *Result*. En esta clase se almacena la situación relevante de la partida antes, durante y después de la simulación o ejecución de cada individuo.

También contiene aspectos como el daño realizado o la distancia recortada en el turno. Esto facilita la implementación de la función que determina el valor de adaptación al medio del individuo a través de la estrategia.

Además, permite conocer si un individuo es válido o no. Un individuo no será válido si conduce a error o posiciona al personaje sobre una casilla inválida, ocupada o fuera del rango.

9.2.- Implementación de estrategias y acciones

Como ya comentamos antes, las estrategias serán aspectos de gran impacto en la jugabilidad y, más concretamente, en el algoritmo genético. De la buena implementación de las estrategias dependerá la obtención del mejor resultado.

Hasta ahora, y con el objetivo de poder implementar con cierta eficiencia el algoritmo genético, hemos incluido dos estrategias en el sistema: Perseguir (*ChaseStrategy*) y escapar (*EscapeStrategy*). No obstante, si queremos que el prototipo realmente muestre las bases de la jugabilidad deberemos incluir, como mínimo, dos estrategias más: una que muestre un comportamiento ofensivo (*OffensiveStrategy*) y otra con un comportamiento defensivo (*DeffensiveStrategy*).

Además, será necesario crear acciones que puedan permitirnos comprobar el correcto funcionamiento de estas nuevas estrategias.

Es necesario recordar, que la clase Estrategia (*Strategy*), era una clase abstracta, con lo cual, por definición, no se puede realizar ninguna instancia de esta. Tendremos que utilizar la herencia para definir diferentes tipos de estrategia.

9.2.1.- Perseguir y escapar

Estas dos estrategias han sido implementadas inicialmente por varios motivos: son básicas en la jugabilidad, son las más fáciles de implementar y es sencillo comprobar si el turno se ha ejecutado siguiendo la estrategia.

```
public class ChaseStrategy : Strategy {

    public override double CalculateFitness (Result individualResult) {
        return individualResult.getCroppedDistance ();
    }
```

Ilustración 22: *Estrategia de perseguir, incluyendo la función de cálculo de adaptación.*

En el caso de la estrategia de escape, la adaptación del individuo será el contrario a la distancia recortada.

```
public class EscapeStrategy : Strategy {

    public override double CalculateFitness (Result individualResult) {
        return  (0.0 - individualResult.getCroppedDistance ());
    }
```

Ilustración 23: *Estrategia de escapar.*

9.2.2.- Estrategia ofensiva

Una estrategia ofensiva debe buscar producir daños sobre el enemigo, aun a costa de descuidar la defensa durante el turno de combate. En base a esta descripción, podríamos elegir varios criterios por los que juzgar el individuo.

```
public class OffensiveStrategy : Strategy {

    public override double CalculateFitness (Result individualResult) {
        double fitness = (individualResult.getDamageDone () * 4) + (individualResult.getCroppedDistance () * 2) +
            (individualResult.getCharacterDifference(1)) - (individualResult.getEnemyDifference(0));

        return fitness;
    }
```

Ilustración 24: *Estrategia ofensiva*

En este caso, buscamos darle la mayor importancia posible al hipotético daño realizado al enemigo durante el turno, pues, al fin y al cabo, es de lo que trata el juego, de reducir la vida del rival. El segundo parámetro con más importancia es la distancia recortada, es decir, cuánto se acerca. En caso de no poder realizar daño al adversario, la estrategia ofensiva debe buscar acercarse si eso le permite atacar en próximos turnos. También interesa que el personaje haya incrementado su ataque o que el enemigo haya sufrido otro tipo de daño.

La estrategia seguida para multiplicar los diferentes parámetros por un número ha sido multiplicar por 2 el parámetro más importante. Si de entre los parámetros restantes, hay otro que es más importante que el resto, multiplicar por 2 ese y todos los parámetros de mayor importancia.

9.2.3.- Estrategia defensiva

Una estrategia defensiva debe buscar evitar o mitigar los posibles daños que pueda sufrir por parte del enemigo. Se podría considerar el planteamiento totalmente contrario a la descrita anteriormente.

```
public class DeffensiveStrategy : Strategy {

    public override double CalculateFitness (Result individualResult) {
        double fitness = ((individualResult.getCharacterDifference(5)*4 + individualResult.getCharacterDifference(0)*2 +
            individualResult.getCharacterDifference(2)*2 - individualResult.getCroppedDistance ()) );
        return fitness;
    }
```

Ilustración 25: *Estrategia defensiva.*

En este caso, se utilizan como parámetros diferenciales la variación en el escudo, y en la vida del personaje, la variación en el parámetro de defensa y, por último, la distancia que consigue alejarse.

9.2.4.- Las funciones de diferencia

Una función que hemos utilizado en las dos estrategias son las funciones *getCharacterDifference(i:int)* y *getEnemyDifference(i:int)* de la clase *Result*. Estas funciones reciben como parámetro el índice asociado al atributo sobre el que se quiere conocer la diferencia y devuelve el cambio producido, en personaje y enemigo respectivamente, desde justo antes de la simulación del individuo hasta el momento en el que se llama a la función. Es un método muy interesante para implementar mayor variedad de estrategias de forma sencilla.

Como se puede apreciar en la ilustración 28, el cálculo diferencial entre el estado actual y el inicial de cada atributo, se realiza accediendo a la variable *CharacterStatus* que está guardada en el controlador de partida.

```
public double getCharacterDifference ( int op ) {

    Attributes att = (Attributes) op;

    switch (att) {

        case Attributes.LIFE:
            return (cs.getLife ()    - MatchController.getPlayer().characterCombat.getLife());

        case Attributes.ATTACK:
            return (cs.getAttack ()  - MatchController.getPlayer().characterCombat.getAttack());

        case Attributes.DEFENSE:
            return (cs.getDefense () - MatchController.getPlayer ().characterCombat.getDefense ());

        case Attributes.SPEED:
            return (cs.getSpeed ()   - MatchController.getPlayer ().characterCombat.getSpeed ());

        case Attributes.FATIGUE:
            return (cs.getFatigue () - MatchController.getPlayer ().characterCombat.getFatigue ());

        case Attributes.SHIELD:
            return (cs.getShield () - MatchController.getPlayer ().characterCombat.getShield());

        default:
            return (cs.getAttack ()  - MatchController.getPlayer().characterCombat.getAttack());

    }
}
```

Ilustración 26: *Función getCharacterDifference(op : int).*

9.2.5.- Acciones

Para que la implementación de las estrategias pueda comprobarse, es necesario crear acciones que permitan jugar y recibir una salida esperada. La clase *Action* es una clase abstracta de la que heredarán el resto de posibles acciones. Entre sus atributos están su id, el nombre y la descripción de la acción y el cansancio (*fatigue*) que consume realizarla. Además, también puede tener efectos sobre el jugador.

```
public abstract class Action {

    protected int id;
    protected string nameAction;
    protected string description;
    protected int fatigue;

    protected List<Effect> characterEffects;
```

Ilustración 27: *atributos de la clase Action*

Si recordamos el diseño de esta clase y de sus subclases planteadas, la herencia es un poco más “compleja” que, en el caso de las estrategias, ya que será necesario incluirle diferentes atributos como el rango, el daño o la cantidad de escudo que recuperan. No obstante, son clases más genéricas, por ejemplo, de la clase *Attack*, no sale un solo ataque, sino tantos como queramos, con distintos valores para el daño, el rango (o casillas a las que afecta), cansancio, etc.

Todas las instancias de acciones en el videojuego se han guardado en una clase *GlobalActions*, asociada a un *GameObject* en la escena. Esta clase contiene un *IDictionary<int, Action>* **[19]** que hace que la búsqueda de una acción por su id sea muy rápida. La idea de tener esta clase, es aumentar y facilitar la cohesión en las habilidades, de manera que, si modificamos una acción en *GlobalActions*, esta será modificada para cualquier individuo o estrategia que la referencie.

También es necesario comentar la clase *Effect.* Una acción puede tener asociado uno o varios efectos sobre el personaje que la ejecuta sobre el enemigo. Dichos efectos incrementarán o reducirán un atributo durante una cantidad determinada de turnos o, en su defecto, hasta la finalización de la partida. Al comienzo de cada turno, se actualizan los efectos y se eliminan si ya ha “caducado”.

9.3.- Implementación de la animación

Uno de los mayores problemas a los que me he enfrentado durante la implementación de este proyecto ha sido la animación de los personajes y del videojuego en general. Hasta ahora, siempre que había realizado un algoritmo de estas características, solo me había tenido que preocupar por obtener un procesamiento de los datos lo más rápido posible para, una vez obtenido el resultado, mostrarlo por pantalla de forma instantánea. En este caso, para la implementación del videojuego, tuve que cambiar la forma de pensar. Aquí también es necesario obtener y procesar los datos lo más rápido posible, no obstante, una vez que el algoritmo genético devuelve el resultado, es necesario mostrarlo en una secuencia de tiempo correcta. Esto es: **animar la escena.**

Para esto, una vez que el algoritmo genético devolvía el mejor individuo obtenido, realizamos una simulación final de ese individuo -*En realidad, una ejecución del individuo*-, almacenando los pasos que se daban en una clase *Step*, cuya función es almacenar únicamente lo que debe hacer el individuo cuando llegue a ese paso del turno, incluyendo la animación que debe ejecutar el modelo 3D del personaje.

```
public class Step {

    public TypeOfSteps type;
    public string      animation;

    public Vector3     startPosition;
    public Vector3     endPosition;
    public float       moveSpeed;

    public Vector3     objective;
    public double      value;
```

Ilustración 28: *atributos de la clase Step.*

Instancias de esta clase son almacenadas en una cola (Clase *Queue<T>* **[19]**) en la clase Personaje, de forma que, cuando se de paso a la ejecución, se empieza a retirar de la cola por el mismo orden por el que entraron, y se procesan los pasos para ejecutar la animación con la dirección y el sentido esperado, mostrando por pantalla el resultado de una forma secuencial en la que el jugador entienda lo que está pasando en la escena durante cada turno.

Por el momento, la velocidad del paso no interfiere, el movimiento, o de forma más genérica, la acción, se ejecuta en un tramo de exactamente un segundo, independientemente del número de frames reales a los que se ejecute el juego. Para esto, utilizamos, en la función *Update()* de la clase abstracta *MonoBehaviour,* de la que hereda *Character.*

9.4.- Implementación de la interfaz

En el ámbito de los videojuegos, la interfaz es una parte de gran importancia. El videojuego tiene que responder de la mejor forma posible ante lo que espera el jugador, sin dar lugar a ningún tipo de confusión. Es conveniente recordar que la interfaz será la única vía de comunicación entre el universo ficticio del videojuego y el universo real, en el que se encuentra el jugador. Por todo esto, debemos indicar lo mejor posible la situación en la que se encuentra la escena o, al menos, las distintas partes de la situación que necesite el usuario para tomar las decisiones. Por otra parte, la interfaz debería mostrar cómo podría llegar hasta dichas decisiones. En este caso concreto, y como ya comentamos durante el diseño del videojuego, la intención es que el jugador tenga el mayor flujo de información posible sobre la situación actual, ya que la incertidumbre de la decisión vendrá dada por la aleatoriedad de ejecución del algoritmo genético.

En este caso, Unity permite al desarrollador utilizar gran cantidad de opciones para crear interfaces totalmente funcionales de una forma sencilla. Para la construcción de la interfaz, arrastramos los elementos que necesitemos (botones, paneles de texto, imágenes…) a un *Game Object* propio de Unity, un *Panel* **[20]**.

Por ejemplo, en la siguiente imagen **[Ilustración 31]**, podemos visualizar cómo queda la interfaz de la pantalla de inicio tras añadirle el logo, los botones y algunas imágenes más para estilizar la pantalla. Tanto la barra superior, como cada uno de los dibujos de cromosomas que aparecen en la pantalla son *Game Objects* dinámicos, que se mueven a lo largo de la pantalla, y están controlados por scripts.

Ilustración 29: *Pantalla de inicio*

Dentro de la partida, la interfaz coge aún más peso, pues se trabaja con datos que varían durante el transcurso de esta y que, además, deben variar en el momento adecuado, justo cuando se ejecuta la animación **[Ilustración 32]**.

Ilustración 30: *En todo momento, el jugador debe conocer su vida y su cantidad de escudo, al igual que la de su enemigo.*

De esta forma, en un instante de la partida entre la ejecución de un turno y otro, habrá dos realidades: la existente tras finalizar el turno, que se habrá obtenido tras ejecutar el individuo para asegurar su validez y calcular su adaptación y la que se muestra por pantalla, que se debería actualizar de manera secuencial con el devenir de las acciones. Además, se comprobará que factores como la vida o el escudo no muestran valores inválidos en ningún momento, como valores por debajo de 0 o por encima del máximo permitido.

Además, es necesario controlar los botones, desactivando los de los jugadores cuando no sea su turno **[Ilustración 33]** e indicar por pantalla un mensaje alertando de la finalización de la partida y la consiguiente victoria de uno de los dos personajes **[Ilustración 34]**. Además, para facilitar el entendimiento de lo que pasa, se mostrará por pantalla el daño hecho / la vida recuperada / el escudo añadido en cada acción del personaje a través de un indicador posicionado sobre el personaje que lo recibe **[Ilustración 35]**.

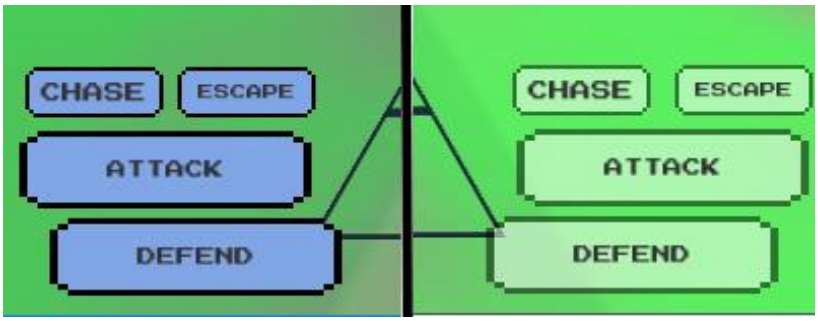


Ilustración 31: *Cuando es el turno del jugador, los botones tienen un color mate. Cuando no es su turno, son blancos con alta opacidad.*

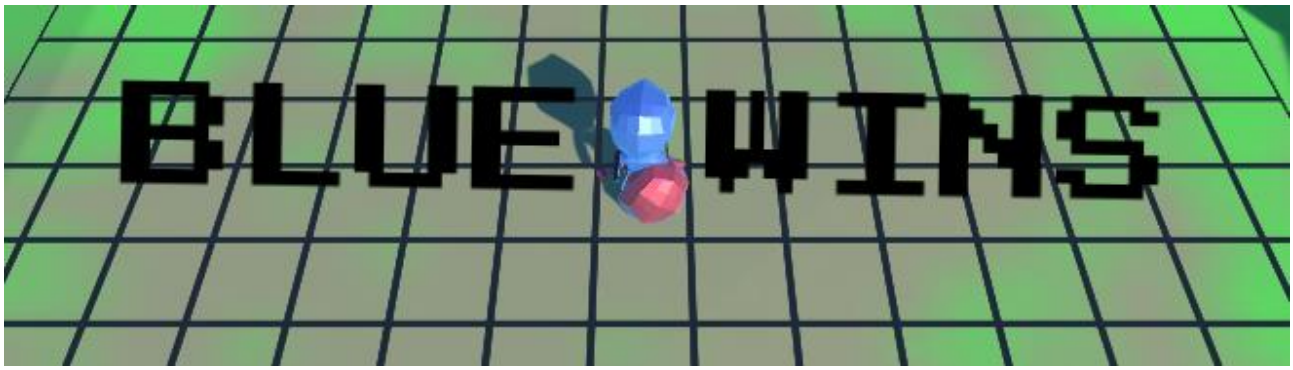


Ilustración 32: *Mensaje de finalización del combate tras ganar el personaje azul.*

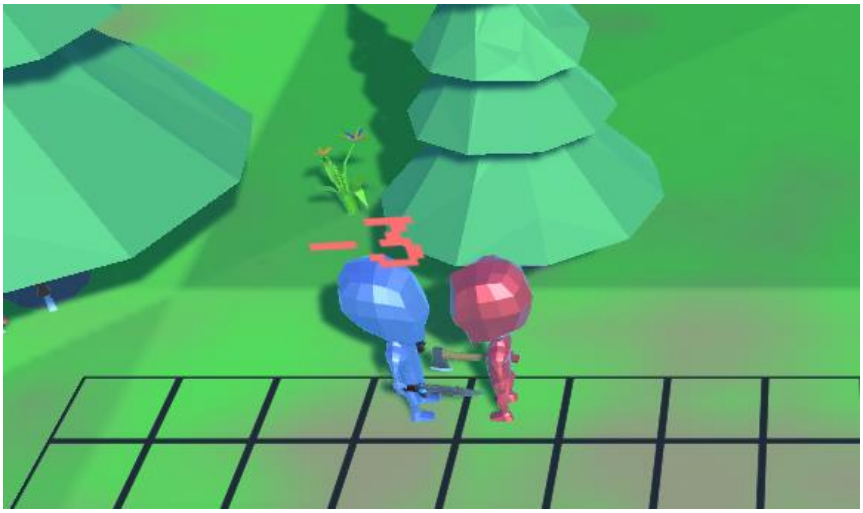


Ilustración 33: *El personaje azul acaba de sufrir daño.*

9.5.- Modo un jugador

Para el modo un jugador, la mayor dificultad fue implementar una pequeña inteligencia artificial que decidiera, en el turno del enemigo, cuál era la mejor estrategia que podía realizar. Además de esto, fue necesario eliminar los botones del segundo jugador de la interfaz en dicha escena, y controlar el cambio de turno para obtener respuesta automática por parte de la CPU.

```
public override Strategy chooseStrategy(){
    List<int> def = new List<int> ();
    List<int> off = new List<int> ();
    CharacterStatus enemy = MatchController.getEnemy ().getCharacterCombat ();

    for (int i = 0; i < strategys.Count; i++) {
        Strategy s = strategys [i];
        if (s.getBalance() < 0)     def.Add (i);
        else                    off.Add (i);
    }

    def.Sort ();
    off.Sort ();

    if (enemy.getLife () < characterCombat.getFatigue ()) {
        if (off.Count > 0) {
            return strategys [off [off.Count - 1]];
        }
    } if (characterCombat.getLifePercentage() < 0.4) {
        if (def.Count > 0) {
            if (characterCombat.getFatigue () < enemy.getFatigue () ||
                characterCombat.getDefense () <= enemy.getAttack ()) {
                return strategys [def [def.Count - 1]];
            } else
                return strategys [def [Random.Range (0, def.Count)]];
        }
    } else {
        if (characterCombat.getPosition ().distance (enemy.getPosition ()) >= 3) {
            if (off.Count > 0)
                return strategys[off[Random.Range(0,off.Count)]];
        } if (off.Count > 0) {
            return strategys [off [off.Count - 1]];
        }
    }

    return strategys [Random.Range (0, strategys.Count)];
}
```

Ilustración 34: *Método chooseStrategy(). El enemigo elige cuál podría ser la mejor estrategia dividiéndolas entre si son ofensivas o defensivas.*

Esta pequeña inteligencia artificial debería cambiar dependiendo del enemigo. Es decir, el jugador debería enfrentarse a enemigos donde la CPU debiera elegir más “alocadamente”, otros donde la estrategia a elegir fuera netamente defensiva e incluso, algunos donde solo tuvieran una estrategia disponible, por lo que no habría opción a elegir.

9.6.- La clase *Configuration*

Como durante la implementación del juego hemos decidido no centrar nuestros esfuerzos en trabajar la persistencia de datos, he decidido crear una clase con atributos estáticos que almacene todos los valores respectivos a los personajes, sus estados iniciales y los parámetros que dirigen el comportamiento del algoritmo genético. Así, durante futuras pruebas para establecer los mejores valores posibles para cada parámetro, solo tendremos que modificar los atributos de esta clase.

```
public class Configuration : MonoBehaviour {

    //PLAYER ATTRIBUTES
    public static string playerName    = "BLUE";
    public static int    playerLife    = 20;
    public static int    playerHealF   = 1;
    public static int    playerAttack  = 1;
    public static int    playerDefense = 1;
    public static int    playerSpeed   = 2;
    public static int    playerFatigue = 4;
    public static int    playerPosX    = 3;
    public static int    playerPosY    = 4;

    //ENEMY ATTRIBUTES
    public static string enemyName     = "RED";
    public static int    enemyLife     = 20;
    public static int    enemyHealF    = 1;
    public static int    enemyAttack   = 1;
    public static int    enemyDefense  = 1;
    public static int    enemySpeed    = 1;
    public static int    enemyFatigue  = 4;
    public static int    enemyPosX     = 11;
    public static int    enemyPosY     = 4;

    //GENETIC ALGORITHM VARIABLES
    public static int    individualCount     = 100;
    public static int    maxIteration        = 40;
    public static int    genWOChanges        = 20;
    public static int    mutationProbability = 25;
    public static int    tornaumentSize      = 3;
    public static double numElitism          = 0.25;

    public static int    individualsSaved    = 10;

    //SCENARIO VARIABLES
    public static int   tamX = 15;
    public static int   tamY = 9;
```

Ilustración 35: *Atributos de la clase Configuration*

Así, tanto el algoritmo genético como los personajes obtendrán los datos de entrada desde un *GameObject* asociado a este script **[Ilustración 19]**.

Dos métodos que pueden resultar interesantes, y que he implementado para esta clase son *fromString y toString*. El primero coge una cadena de caracteres de entrada y

trata de inicializar sus variables a partir de este. En el caso del segundo, devuelve todos los valores de la instancia.

Para que la cadena de texto sea lea correctamente, debe tener el mismo formato que en que exporta los datos la función *toString*. Esto es, separando las secciones de jugador, enemigo, genético y escenario por comas, y los valores dentro de él por espacios.

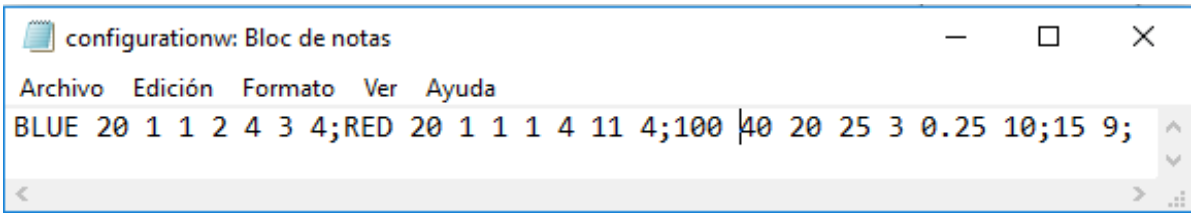


Ilustración 36: *formato en el que se exporta la clase Configuration*

9.7.- Implementación de la cámara

Para la cámara, se ha probado entre varias, y finalmente se ha elegido una situada ligeramente sobre la escena y antes de la primera fila. La intención es poder observar lo que está pasando en cada momento. También se ha dotado a la cámara de un script que permite girarla sobre el punto central del escenario. Para esto, es necesario hacer clic derecho con el ratón y moverlo hacia los lados.

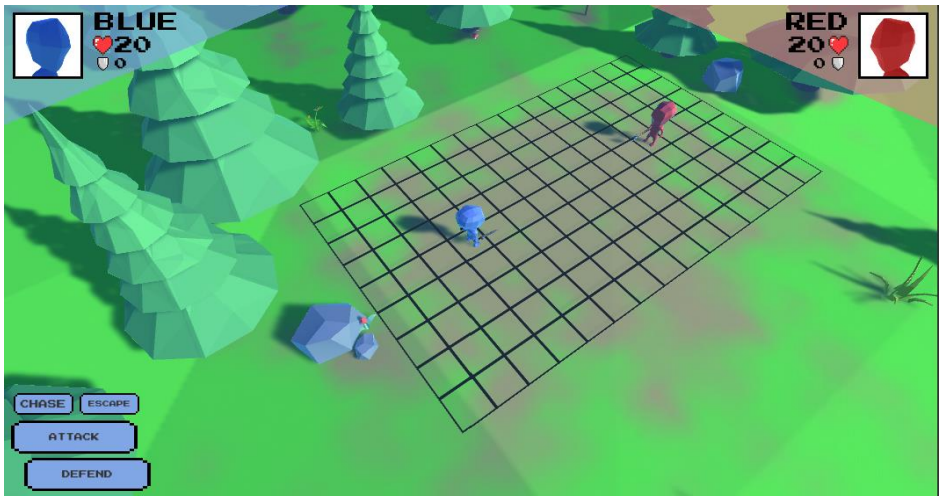


Ilustración 37: *Escena Single Player en 16:9 con la cámara ligeramente rotada.*

10.- Pruebas

10.1.- Introducción

A continuación, vamos a tratar de explicar las pruebas realizadas, tanto de la jugabilidad como de las funcionalidades implementadas. Además, durante esta etapa, realizaremos un estudio del funcionamiento del algoritmo genético en general y de qué variaciones produce la modificación de uno y varios parámetros. Esto nos permitirá ajustar los valores del algoritmo genético a los que puedan ser más adecuados.

Durante la etapa de implementación, ya nos hemos encontrado con la necesidad de ejecutar varias veces el proyecto para comprobar si el código de las diferentes partes era correcto y desempeñaba las funciones que se esperaban del mismo. Hasta ahora, estas ejecuciones se han realizado directamente sobre el programa Unity, que permite ejecutar el juego además de conocer el estado gráfico en cada momento.

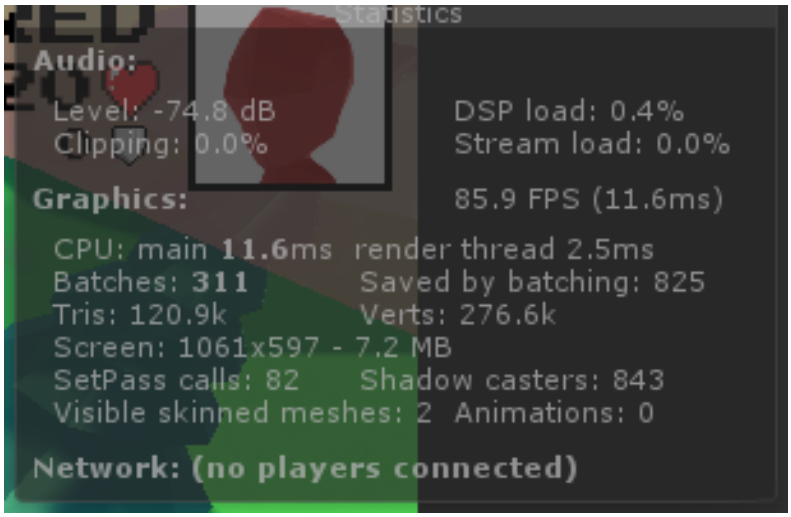


Ilustración 38: *Estado del audio y los gráficos durante una ejecución del juego.*

A partir de estas ejecuciones y sus estadísticas se pueden sacar algunas conclusiones. Por ejemplo, el juego suele ejecutarse de forma fluida en el tiempo, en ningún momento se bajan de los 30 frames.

Además, a través de tareas de testeo de videojuegos, se ha comprobado que los individuos no válidos se descartan correctamente y que los personajes siguen las estrategias con bastante sentido. La vida, el daño y el escudo también se ejecutan siempre correctamente.

Gracias a estas pruebas también se consiguieron solucionar todos los fallos obtenidos relativos a intentar acceder a posiciones del cromosoma que pudieran ser inválidas. En principio, el juego no debería ver su ejecución terminada antes de tiempo por ningún tipo de excepción no controlada.

Las funcionalidades del juego están muy bien delimitadas por el momento. El jugador, tiene pocas opciones de hacer algo inesperado para el sistema, pues solo dispone de 4 botones, cuyo correcto funcionamiento ya hemos comprobado, y un leve giro de cámara, poco relevante en la jugabilidad. Es por eso que, en mi opinión, no se requiere un plan de pruebas extenso para comprobar la funcionalidad actual del prototipo, aunque es probable que, en etapas más avanzadas del producto, en las que entren en juego aspectos como persistencia, personalización o evolución, si vaya a ser imprescindible un plan de pruebas que garantice la calidad del producto y la cohesión entre sus partes.

El juego también ha sido probado por más de 50 posibles usuarios, de los que se han obtenido y analizado respuestas sobre la posición de la cámara y de los botones, el color y la forma de estos últimos o algunas posibles funcionalidades que tal vez no se le ocurrieron al diseñador al inicio del proyecto.

10.1.1.- Equipo utilizado

Todo el desarrollo descrito hasta ahora se ha realizado sobre el mismo equipo. Se trata de un Asus VivoBook del año 2010 con un procesador Intel® Core™ i5-3317U de 1,70GHz, una memoria RAM de 4GB y una tarjeta gráfica NVIDIA GEFORCE GT 635M de 2 GB. El sistema optativo es un Windows 10 de 64 bits.

Para las pruebas, nos basaremos, principalmente, en los resultados dados por este equipo, aunque también referenciaremos a ejecuciones hechas en otros distintos equipos informáticos.

10.2.- Pruebas del algoritmo genético

Para que el algoritmo genético se ejecute con la mayor eficiencia y eficacia posible, es necesario realizar un estudio para evaluar la calidad los operadores que se han implementado y los posibles valores que deberían tener los parámetros que manejan su comportamiento.

Es necesario, entre otras cosas, dar un equilibrio al número de individuos, número máximo de iteraciones (con y sin cambios), probabilidad de mutación, número de elitismo y tamaño de torneo para que el algoritmo genético trabaje en armonía y garantice buenos resultados en tiempos de ejecución válidos.

Además, trataremos de comprobar cómo funcionan estos aspectos en consonancia con el cansancio del personaje. Es necesario tener en cuenta que el cansancio del personaje es al final el factor que determina el tamaño de cada individuo, por lo que, a mayor cansancio, mayor número de posibilidades, mayor complejidad del problema y mayor dificultad para encontrar la mejor solución. No obstante, en el diseño del videojuego se plantea que cada personaje podría tener un valor diferente para el cansancio, por lo tanto, no deberíamos buscar cuál es el mejor valor para esto. Sobre el cansancio, deberíamos comprobar que, a mayor valor de cansancio para el mismo individuo, se obtienen soluciones iguales o mejores. De lo contrario, no tendría sentido incluir este apartado.

10.2.1.- Consideraciones Iniciales

Es necesario señalar que Unity dificulta en gran medida la medición de tiempos que se ejecutan sobre el mismo hilo que el juego en general. Por esto, realizaremos una medición subjetiva. Dado que es un videojuego, y que debe estar adaptado para gran cantidad de dispositivos en gran cantidad de entornos, nos basaremos en si el tiempo afecta o no a la experiencia del juego, y a si realmente, sobre el equipo sobre el que se ejecuta, el juego se considera que está funcionando lento.

Dada la gran cantidad de pruebas con otros jugadores y ordenadores realizada, me parece una medida con cierta fidelidad.

10.2.2.- Funcionamiento del algoritmo genético

En primer lugar, debemos comprobar que realmente se está aplicando el proceso genético correctamente. Para responder a esto, ejecutaremos el algoritmo genético con una serie de valores de prueba y sacaremos los datos obtenidos y los individuos de cada generación a un fichero de texto.

Este paso es clave para poder sacar conclusiones realmente válidas durante las siguientes pruebas. Las respuestas se basarán en la observación de los ficheros por parte del que escribe. En el prototipo jugable, se incluirá la funcionalidad de exportar los datos del genético a un fichero de texto (.txt) si se desea comprobar la veracidad de estas conclusiones.

10.2.2.a. Funcionamiento general

El algoritmo genético se ejecutará teniendo en cuenta los parámetros de la tabla a continuación.

ATRIBUTO	VALOR
Número de individuos	100
Número máximo de iteraciones	40
Número máximo de iteraciones sin cambios	10
Probabilidad de mutación	25%
Tamaño del torneo	4
Número de elitismo	25%
Cansancio del jugador	5
Cansancio del enemigo	5

***Tabla 2**: Parámetros iniciales del algoritmo evolutivo.*

Ejecutaremos el algoritmo genético 10 veces. En las 10 ejecuciones, se debería poder responder afirmativamente a todas las preguntas que vamos a realizar.

- **¿Se generan el número de individuos adecuado en la primera generación?**

  Sí, en cada ejecución se generan 100 individuos en la generación 0.

- **En las siguientes generaciones, ¿Sigue habiendo el número de individuos adecuado?**

  Sí, en todas las ejecuciones comprobadas siempre hay 100 individuos en cada generación

- **¿En cada iteración, pasan el número correcto de individuos por elitismo? ¿En caso afirmativo, pasan siempre los mejores?**

  Sí, siempre pasan los 25 mejores individuos de la anterior generación.

- **En algún caso, ¿La ejecución termina antes del número máximo de iteraciones sin cambios?**

  No, no ha ocurrido, aunque en una de las ejecuciones terminó justo en dicho número. No obstante, esto no es incorrecto, puesto que en la primera generación ya apareció el mejor individuo encontrado.

- **¿Se devuelve siempre el mejor de todas las generaciones o se ha encontrado algún individuo mejor que el devuelto finalmente?**

  Como hay elitismo, y este se aplica correctamente, siempre se devuelve el mejor individuo aparecido en el total de las generaciones.

- **¿Se generan torneos con el número adecuado de elementos? De ser así, ¿ganan el torneo los mejores individuos?**

  Esta información, debido a que era muy amplia y destrozaba la construcción y entendimiento del fichero genético, se mostró únicamente por consola. Los torneos siempre eran de 4 elementos que además se ordenaban correctamente.

- **Si variamos el cansancio del personaje ¿Varía también la longitud media de los cromosomas?**

Sí, de hecho, el individuo siempre busca “rellenar” el cansancio restante si queda. Por tanto, el cansancio y el tamaño del cromosoma son directamente proporcionales.

10.2.2.b. Funcionamiento de los operadores de Cruce

Para comprobar el funcionamiento de los métodos implementados para el cruce, hemos sacado por pantalla los padres y los hijos tras aplicarse el operador para los 3 casos. Se ha comprobado que el cruce es efectivo, aunque algunas veces provocaba la obtención de individuos no válidos que acababan siendo descartados, por ejemplo, individuos que trataban de alcanzar una casilla fuera del mapa.

```
before crossOver from start 1;5;5;5;0,0 1,0 0,3 0,3 0,2
after  crossOver from start 1;5;5;5;1,3 1,0 0,3 0,3 0,2
```

***Ilustración 39**: Ejemplo de aparición de crossover, Podemos ver cómo sobre el mismo individuo ha cambiado el primer gen.*

También se ha comprobado el funcionamiento de este operador con varias longitudes de cromosoma. De longitud 1 en adelante, funciona correctamente, aunque en el primer caso la operación es obvia, sustituye todo el cromosoma.

10.2.2.c. Funcionamiento de los operadores de mutación

En esta parte tenemos que comprobar varios aspectos:

i. Que el operador se aplica atendiendo a la probabilidad seleccionada.
ii. Que se ejecutan todos los operadores indistintamente
iii. Que cada uno de los operadores se aplica según lo esperado.

(i) Las dos primeras pruebas que podemos realizar para saber si realmente se está valorando la probabilidad es igualar la probabilidad de mutación a 0, donde no debería ocurrir en ningún caso, y a 100, donde siempre que haya una operación de cruce debería darse. Efectivamente, ambas pruebas se cumplen.

Posteriormente, deberíamos dar un rango intermedio para comprobar que los datos no se desvían. Como la operación no es muy compleja (Obtener un número aleatorio en un rango de 0 a 100 y comprobar que está por debajo de la probabilidad

determinada) y las pruebas anteriormente realizadas fueron concluyentes, bastará con una serie de ejecuciones que se mantengan en el rango para afirmar este punto.

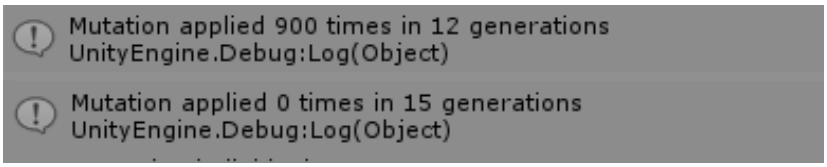
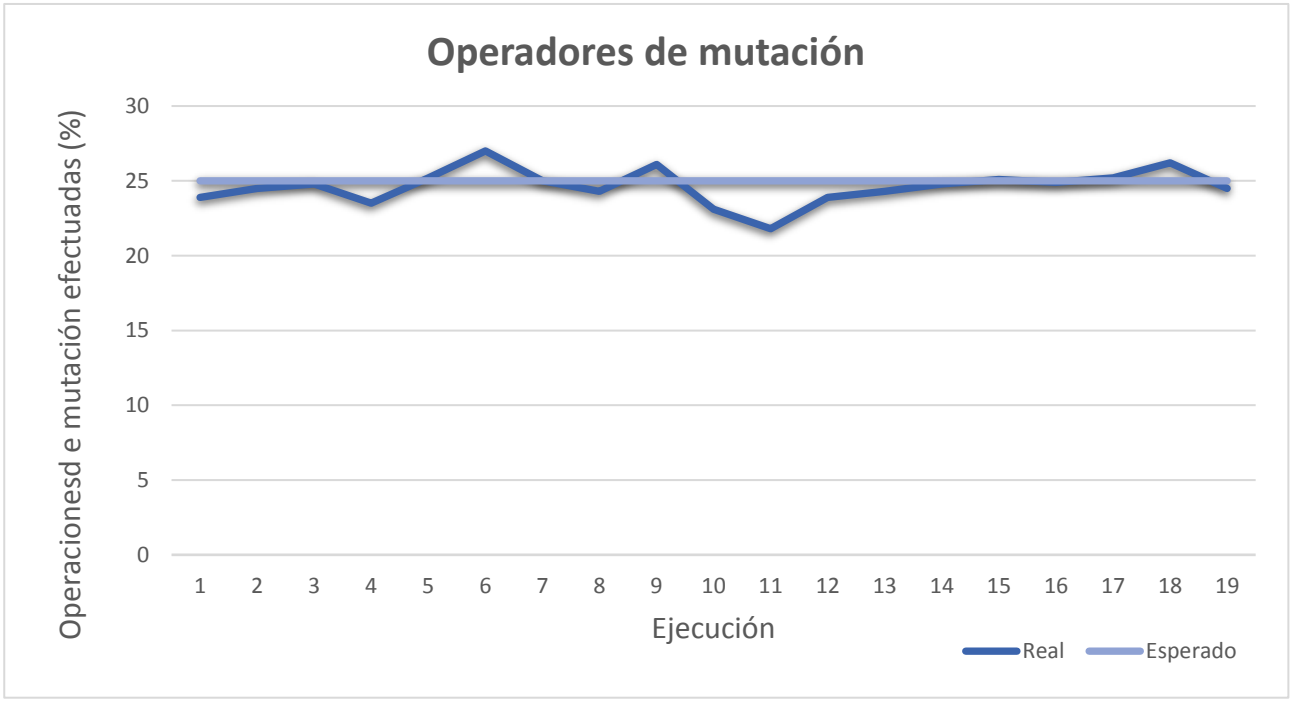


Ilustración 40: *Primera prueba para el operador de mutación*



De acuerdo al gráfico anterior, la desviación del número de operadores efectuados respecto a la línea que marca donde deberían estar no es muy grande, por lo que podemos verificar que este punto es correcto.

(ii) Este punto es muy probable que sea correcto, ya que la operación que redirige a la mutación es un switch al que se accede con un entero aleatorio. No obstante, sacaremos por consola cada operador de mutación cada vez que se ejecute. Tras realizar esto, podemos comprobar que todos aparecen con una frecuencia similar **[Ilustración 43]**.

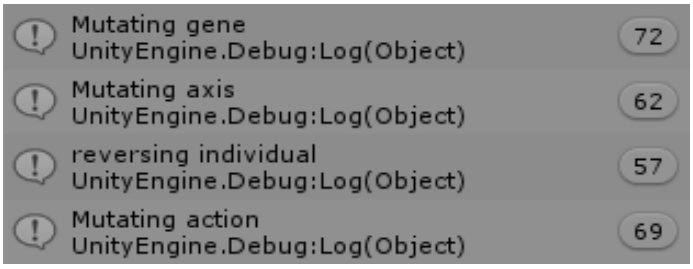


Ilustración 42: *Numero de apariciones de cada operador de mutación*

(iii) Probar que el operador funcione como se espera es algo más complejo. Primero, debemos saber que operador de mutación se está ejecutando y, posteriormente, ver qué cambio realiza sobre el individuo. Comprobamos los cambios sobre los individuos con cada operador de mutación **[Ilustración 4]** y verificamos también este tercer punto.

```
before axis mutation -1;5;5;5;0,01,20,12,31,1
after axis mutation  -1;5;5;5;0,01,20,12,31,2
before gene mutation -0.246211051940918;5;5;5;0,21,11,10,10,2
after gene mutation  -0.246211051940918;5;5;5;0,21,31,10,10,2
```

Ilustración 41: *Ejemplo de mutación sobre dos individuos.*

Con los tres puntos verificados, podemos confirmar que el operador de mutación tiene su funcionalidad bien implementada. Para este operador, solo quedaría estudiar cuál es la probabilidad de mutación más favorable.

10.2.3.- Parametrización del algoritmo genético

Para decidir cuáles son las mejores combinaciones posibles de parámetros para utilizar el algoritmo genético en este caso, realizaremos las pruebas de la siguiente manera:

- En primer lugar, inicializaremos todos los parámetros a valores “no extremos”, es decir que en principio no supongan una diferencia bruta para ninguno de los dos rangos en los que debería moverse.
- Posteriormente, jugaremos con cada uno de los parámetros modificándolo hasta dar con el que mejor resultado pueda darnos, quedándose dicho valor “guardado” para las pruebas realizadas con el resto de variables.

- Una vez hayamos hecho esto con todos los atributos, repetiremos el proceso, aunque con menor detalle, para terminar de refinar el estudio.

Como valores “no extremos” nos pueden valer los utilizados durante la anterior etapa de pruebas, ya que nos hemos encontrado con resultados aceptables y tiempos de ejecución bastante rápidos.

Además, para decidir cuál es la mejor parametrización en cada caso, aumentaremos el cansancio del personaje al valor que nos permita obtener el rango de valores de adaptación más alto posible desde el primer turno. Esto lo realizaremos calculando el menor camino posible para eliminar al enemigo partiendo de las posiciones iniciales. Partiendo siempre de dicha configuración inicial, podremos basar nuestro estudio en resultados objetivos.

El valor mínimo de fatiga para eliminar a un enemigo de vida 10 con las estrategias, acciones, ataques y movimientos implementados hasta ahora es de 13. Por lo tanto, le daremos esa fatiga al personaje azul, que será quien comenzará actuando en la partida.

A partir de aquí, lo que haremos será ejecutar 20 veces el algoritmo genético en una misma ejecución para cada configuración. Estos datos los sacaremos por pantalla y los guardaremos a mano en una tabla.

En esta tabla estará contenido el valor de la adaptación del mejor individuo obtenido tras la ejecución del algoritmo genético. Su celda estará coloreada en base al tiempo de ejecución de forma subjetiva. De esta manera, se usará el verde para las soluciones obtenidas más rápidamente, el verde claro y el amarillo para tiempos considerables y el rojo para tiempos malos o inaceptables por el usuario.

10.2.3.a. Número de individuos

Partiendo de los datos anteriormente registrados. Realizaremos las ejecuciones con un rango de individuos amplio.

		Número de individuos					
		25	50	100	200	400	800
Número de ejecución	1	18,17	19,17	20	20	49	54
	2	20	19,17	20	37	49	68
	3	20	19,17	20	48	49	68
	4	20	19,17	20	49	49	68
	5	20	20	20	49	49	68
	6	20	20	49	49	49	68
	7	20	20	49	49	49	68
	8	20	20	49	54	49	68
	9	20	20	49	54	54	68
	10	20	43	49	54	54	68
	11	20	43	49	54	54	96
	12	42	43	49	54	54	96
	13	42	43	49	54	54	96
	14	42	43	49	54	54	96
	15	42	43	49	54	54	96
	16	42	43	54	54	54	96
	17	42	43	54	68	54	96
	18	42	43	54	68	54	96
	19	42	43	54	68	96	96
	20	43	43	54	68	96	96
	MED	**29,86**	**32,48**	**43**	**52,95**	**56,2**	**81,3**

***Tabla 3**: Comparativa del número de individuos basándonos en el valor de adaptación.*

Según los resultados de la tabla anterior, hasta los 100 individuos casi no se nota cambio en la velocidad de ejecución del algoritmo. En el caso de los 200 individuos ya sí podemos encontramos con cierta ralentización que, dependiendo del equipo y la situación, podría ser molesta. Por tanto, **el número de individuos estará entre 100 y 200 individuos.** Aumentar los individuos por encima de ese rango, supondría demasiado impacto en la experiencia de juego, por lo que queda descartado.

10.2.3.b. Número de iteraciones

Partiremos de los 150 individuos, para ver el número de iteraciones que mejor resultado dan. Eliminaremos la restricción de iteraciones sin cambios para estudiar este caso.

		Número máximo de iteraciones					
		5	10	20	40	80	100
Número de ejecución	1	13,79	16	20	49	20	92
	2	13,79	18,17	20	49	54	96
	3	14,67	19,17	20	49	54	96
	4	16	20	20	49	73	96
	5	18,52	20	20	49	82	96
	6	20	20	20	54	82	96
	7	20	20	20	54	82	96
	8	20	20	20	82	87	96
	9	20	38	20	82	96	96
	10	20	38	68	82	96	96
	11	20	38	68	87	96	96
	12	20	38	68	87	96	96
	13	20	38	68	87	96	96
	14	20	38	68	87	96	96
	15	20	54	68	92	96	96
	16	20	54	92	96	96	96
	17	20	72	92	96	96	96
	18	20	72	92	96	96	96
	19	20	72	92	96	96	96
	20	20	72	92	96	96	96
	MED	**18,84**	**38,87**	**52,4**	**75,95**	**84,3**	**95**

***Tabla 4**: Comparativa del número máximo de iteraciones basándonos en el valor de adaptación.*

Tras ver los resultados obtenidos, creo que merece la pena buscar un número máximo de iteraciones cercano a las **40 iteraciones**. Si reducimos esa cifra, perdemos gran calidad de la solución, aunque bien es cierto que conseguimos una ejecución algo más rápida.

10.2.3.c. Número de iteraciones sin mejora

Para acelerar la obtención de la solución del apartado anterior, podemos estudiar cómo nos favorece usar un número de iteraciones sin mejora.

		Número máximo de iteraciones sin mejora					
		3	5	10	15	20	30
Número de ejecución	1	20	20	20	20	42	68
	2	20	20	20	20	78	82
	3	20	20	20	20	78	92
	4	20	20	20	20	78	96
	5	20	20	54	20	78	96
	6	20	20	54	20	78	96
	7	20	20	54	20	78	96
	8	20	20	54	20	78	96
	9	20	20	54	20	92	96
	10	20	20	54	82	92	96
	11	20	20	54	82	92	96
	12	20	20	54	82	96	96
	13	20	20	54	82	96	96
	14	37	20	54	82	96	96
	15	37	20	54	82	96	96
	16	37	20	54	82	96	96
	17	37	20	54	82	96	96
	18	43	20	54	82	96	96
	19	43	20	54	82	96	96
	20	43	20	54	82	96	96
	MED	**26,85**	**20**	**47,2**	**54,1**	**86,4**	**93**

***Tabla 5**: Comparativa del número de iteraciones sin mejora basándonos en el valor de adaptación.*

Como la mejora en el tiempo conseguida con las **20 iteraciones** es buena y la solución obtenida de media está cerca de la mejor, aplicaremos dicho criterio.

10.2.3.d. Probabilidad de mutación

Tras establecer tres parámetros básicos que marcan en gran nivel la velocidad de ejecución, ahora pasaremos a estudiar la probabilidad de mutación.

		Probabilidad de mutación					
		0	10	25	50	75	100
Número de ejecución	1	20	20	20	20	20	48
	2	20	20	49	49	54	48
	3	20	20	49	49	54	48
	4	20	20	54	49	54	48
	5	20	49	54	54	54	48
	6	20	49	54	54	54	48
	7	20	49	54	54	54	48
	8	20	49	54	54	54	48
	9	20	49	78	54	54	48
	10	20	49	78	54	54	48
	11	20	49	78	68	54	54
	12	20	49	78	68	54	54
	13	20	49	78	68	54	54
	14	20	49	78	68	54	54
	15	20	49	78	68	54	54
	16	20	49	92	68	54	54
	17	20	49	92	68	54	73
	18	20	49	92	82	87	73
	19	20	49	96	82	87	73
	20	20	49	96	82	92	73
	MED	**20**	**43,2**	**70,1**	**60,65**	**57,5**	**54,8**

***Tabla 6**: Comparativa de la probabilidad de mutación basándonos en el valor de adaptación.*

Como hemos podido comprobar, mayor probabilidad de mutación no nos asegura una mejor solución promedia. El rango entre el **25% y el 50%** parece el más adecuado para este problema.

10.2.3.e. Tamaño del torneo

Otro aspecto clave dentro del proceso genético es el proceso de selección de los padres. El tamaño del torneo regula las posibilidades de un individuo con un "mal" fitness de avanzar entre generaciones

		Tamaño del torneo					
		2	4	8	12	16	32
Número de ejecución	1	31	20	20	20	20	82
	2	31	20	20	49	82	92
	3	43	20	20	49	87	92
	4	43	49	20	49	92	92
	5	43	49	49	54	92	96
	6	43	49	49	54	92	96
	7	43	49	77	54	92	96
	8	43	49	77	54	92	96
	9	43	49	77	54	92	96
	10	43	54	77	54	96	96
	11	43	54	78	68	96	96
	12	43	54	96	68	96	96
	13	43	54	96	68	96	96
	14	43	54	96	68	96	96
	15	43	54	96	68	96	96
	16	43	54	96	68	96	96
	17	43	54	96	68	96	96
	18	48	54	96	82	96	96
	19	48	54	96	82	96	96
	20	48	54	96	82	96	96
	MED	**42,55**	**47,4**	**71,4**	**60,65**	**89,85**	**94,7**

Tabla 7: *Comparativa del tamaño de torneo basándonos en el valor de adaptación.*

Probablemente, esta evaluación sea la que más me ha llamado la atención, puesto que no esperaba para nada que aumentar tanto el tamaño de torneo para buscar una rápida convergencia fuera a darnos tan buenos resultados. Más, teniendo en cuenta la gran cantidad de posibilidades que puede tener el personaje. Para cubrirnos las espaldas, escogeremos el **tamaño de torneo 16.**

10.2.3.f. Número de Individuos afectados por elitismo

El elitismo, recordamos, se daba cuando un individuo pasaba entre generaciones por ser, simplemente, el mejor. Definiremos un porcentaje de individuos que pasan entre iteraciones por elitismo.

		Elitismo					
		0	0.10	0.20	0.40	0.60	0.80
Número de ejecución	1	20	54	77	20	20	20
	2	20	54	87	20	82	49
	3	20	54	92	43	87	49
	4	20	54	92	43	92	54
	5	20	54	92	43	92	54
	6	20	54	92	54	92	54
	7	43	54	92	54	92	54
	8	49	54	92	54	92	54
	9	49	54	92	54	92	77
	10	49	54	96	78	96	87
	11	49	54	96	78	96	87
	12	54	54	96	78	96	87
	13	54	54	96	78	96	92
	14	54	54	96	78	96	92
	15	54	54	96	78	96	92
	16	61	54	96	78	96	92
	17	77	54	96	92	96	92
	18	77	54	96	92	96	92
	19	77	54	96	92	96	92
	20	96	54	96	92	96	92
	MED	**48,15**	**54**	**93,2**	**64,95**	**89,85**	**73,1**

Tabla 8: *Comparativa del número de elitismo basándonos en el valor de adaptación.*

Tanto el rango cercano a 0.2 como el rango 0.6 parecen muy adecuados para obtener muy buenas soluciones en tiempos bastante rápidos. Por empate, desempatamos con dos criterios: La peor solución en el 20% es mejor que en el 60% y la convergencia debería ser menor. Por tanto, **tomaremos un 20% de elitismo entre generaciones.**

10.2.3.g. Resto de parámetros

Nuestro algoritmo genético tiene otro atributo, el número de individuos que se cargan desde la memoria de la estrategia asociada al personaje. No obstante, al no tener la persistencia de dichos objetos completamente implementada, es prácticamente imposible hacer las pruebas de funcionamiento o hacer un estudio de los resultados en función de la variabilidad del parámetro.

10.2.3.h. Funcionamiento completo de parámetros

Tras todas estas pruebas, podemos ver en la siguiente tabla la configuración que hemos obtenido para nuestro algoritmo genético.

ATRIBUTO	VALOR
Número de individuos	150
Número máximo de iteraciones	40
Número máximo de iteraciones sin cambios	20
Probabilidad de mutación	30%
Tamaño del torneo	16
Número de elitismo	20%

Tabla 9: *Valores obtenidos tras el estudio.*

Como decisión de diseño de videojuegos, tomaremos como cansancio del jugador como del enemigo para las pruebas y el prototipo con un valor de 5. Este valor se ha tomado en base a pruebas de jugabilidad hechas con anterioridad, en las que se descubrió que tanto la duración de los combates como la velocidad de los mismos era propicia con este intervalo de cansancio o similares.

No obstante, y como ya se ha explicado anteriormente, el cansancio de los personajes no es fijo, por lo que no hay que caer en el error de no tener un algoritmo genético preparado para dichas variaciones.

Es por esto, que, a la hora de tomar decisiones para la parametrización del algoritmo, os hemos decantado por soluciones que exigiesen mayor exploración y mayor variación de la población de la que se parte. Es posible que estos parámetros no sean los que obtienen la mejor solución ni los más rápidos, no obstante, proponen una configuración interesante sobre la que plantear la jugabilidad.

11.- LANZAMIENTO

11.1.- Introducción

Aunque se trata de un prototipo, a estas alturas de la planificación ya puede ser conveniente tener algún plan de actuación para cuando llegue el momento de lanzarlo al mercado y tratar de competir con gigantes y no tan gigantes de la industria.

No podemos olvidar que todo el tiempo invertido en el desarrollo del videojuego debe ir orientado a buscar cierta remuneración. No obstante, lanzar un producto en un sector tan competitivo como es el de los videojuegos no es para nada fácil. En el caso de los estudios independientes es una tarea titánica conseguir un hueco en el mercado.

En base a esto, señalaremos los pasos a seguir, y las opciones a estudiar, para conseguir sacarle partido económico al producto que hemos estado desarrollando. Es necesario tener en cuenta que estas decisiones se basan parcialmente en la subjetividad de quién las toma, en su ética empresarial y en estudios de marketing y diseño gráfico más avanzados de los que podemos incluir en este documento. Por tanto, solo nos dedicaremos a introducir algunas de las posibilidades más recurrentes del sector.

11.2.- Marca e identidad

Para conseguir que nuestro producto sea reconocido tenemos que conocer el término **diferenciación**. Se trata de conseguir un producto que, por un motivo o por otro, destaque sobre el resto de competidores; y, obviamente, que destaque por algo positivo. Se debe buscar un videojuego de calidad, pensado para conseguir una grata experiencia de juego en el consumidor más que para vaciarle los bolsillos. Cuando hablamos de grandes empresas o de empresas independientes, pero con más experiencia en el lanzamiento de videojuegos, estas ya suelen tener un plan de actuación, una reputación formada y cierta masa social de seguidores.

Sin embargo, cuando eres un estudio pequeño y sin experiencia, es necesario replantearse cómo y por qué quieres que te conozcan y a partir de ahí, diseñar tu producto y tu empresa para recibir ese reconocimiento. Este proceso es el conocido como ***branding.***

Para este trabajo concreto, haremos una breve descripción del proceso de branding del videojuego, ya que, en mi opinión, sacar valor al producto que hemos desarrollado también es importante.

En primer lugar, trataremos el nombre del videojuego. Un aspecto con bastante importancia sobre todo en algunas plataformas, ya que la búsqueda se hace a través del nombre. Por ejemplo, en la *PlayStore* de *Android*, la mayoría de las aplicaciones incluyen en el nombre las palabras por las que quieran aparecer en las búsquedas. Como la plataforma no está clara, y esto es algo que podríamos cambiar después, buscaremos un nombre que de alguna forma refuerce el tema unificador y el diseño del juego.

Tras pensar varios nombres y encuestar entre mis conocidos para detectar cual podría ser mejor, el ganador fue ***PLOYn.*** *Ploy*, en inglés, significa estrategia y le añadimos una ‘n’ al final de la palabra para diferenciarlo y hacerlo más sonoro.

Como el diseño visual del videojuego ha buscado simplificar las formas y los movimientos a través de modelos *low-poly* y una interfaz con gran impacto de temática pixel, tanto el logo **[Ilustración 45]** como el resto de aspectos de diseño de interfaces, imágenes y *merchandising* deberían seguir esta línea. A partir de ahí, las decisiones se basarán en aspectos más subjetivos y propios del diseño gráfico.

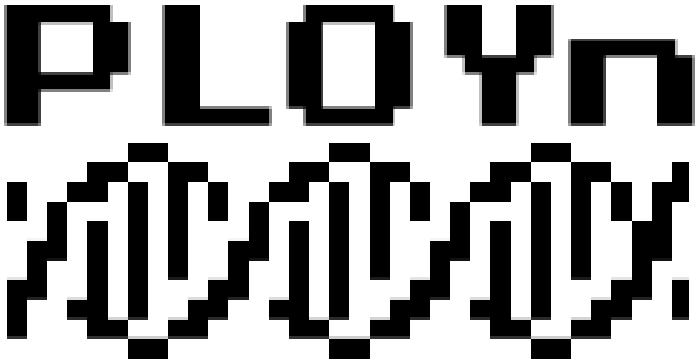


Ilustración 43: *Prototipo de logo para PLOYn.*

Una vez obtenemos el prototipo para el logo, pasamos a elegir el color con el que queremos que asocien a nuestro juego. Probamos con varios **[Ilustración 46]**, siempre teniendo en cuenta el significado que puedan atraer.

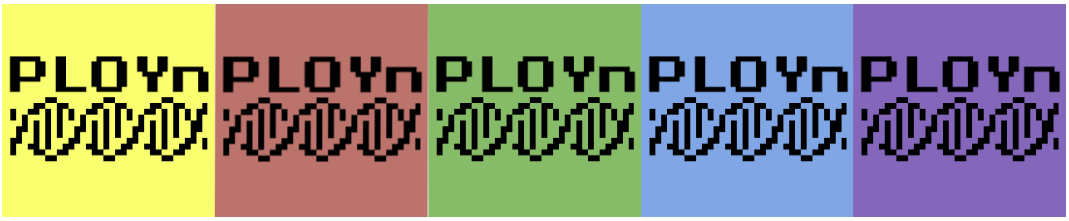


Ilustración 44: *Ejemplos de colores para el logo.*

Decidimos quedarnos con el azul (#80A6E5), que será el color con el que intentaremos promocionar el juego **[Ilustración 47]**. A partir de aquí, aplicaremos estas directrices para diseñar el aspecto visual de las distintas interfaces, la pantalla de inicio y resto de imágenes utilizadas para marketing **[Ilustración 48]**.

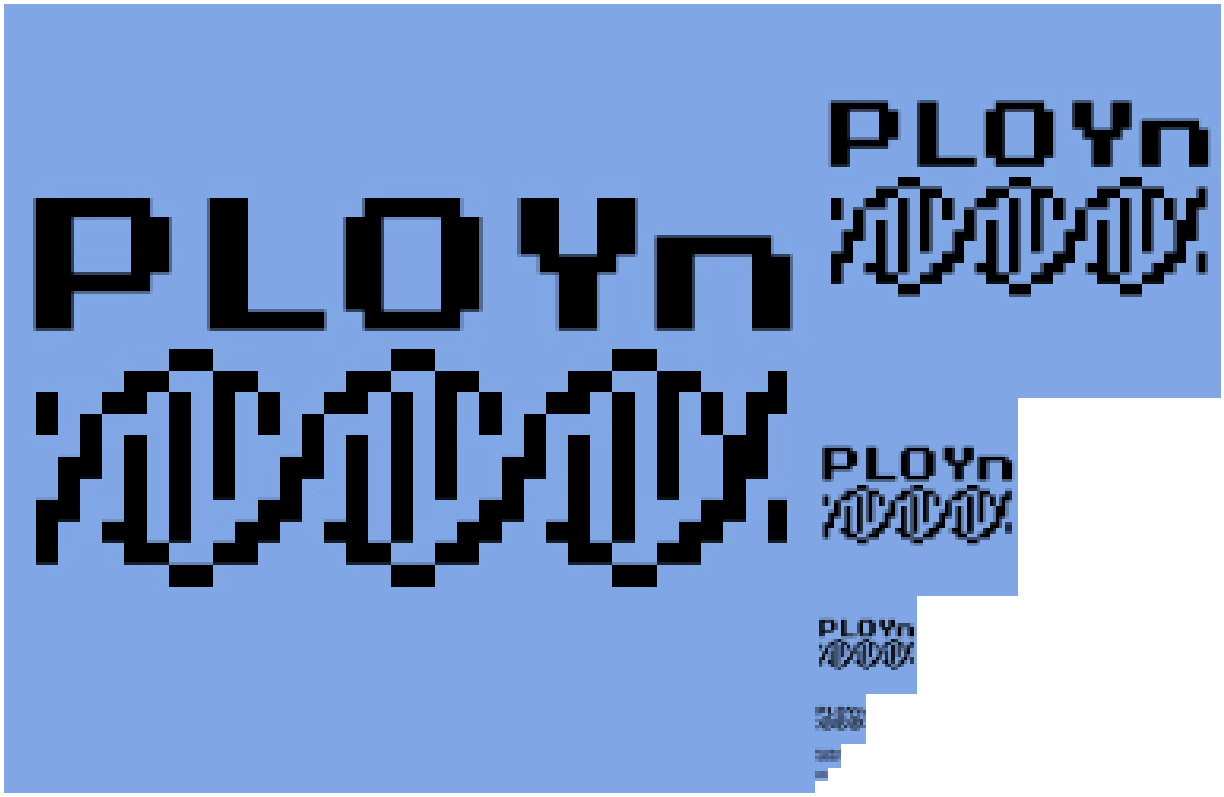


Ilustración 45: *Mipmap del prototipo de logo.*

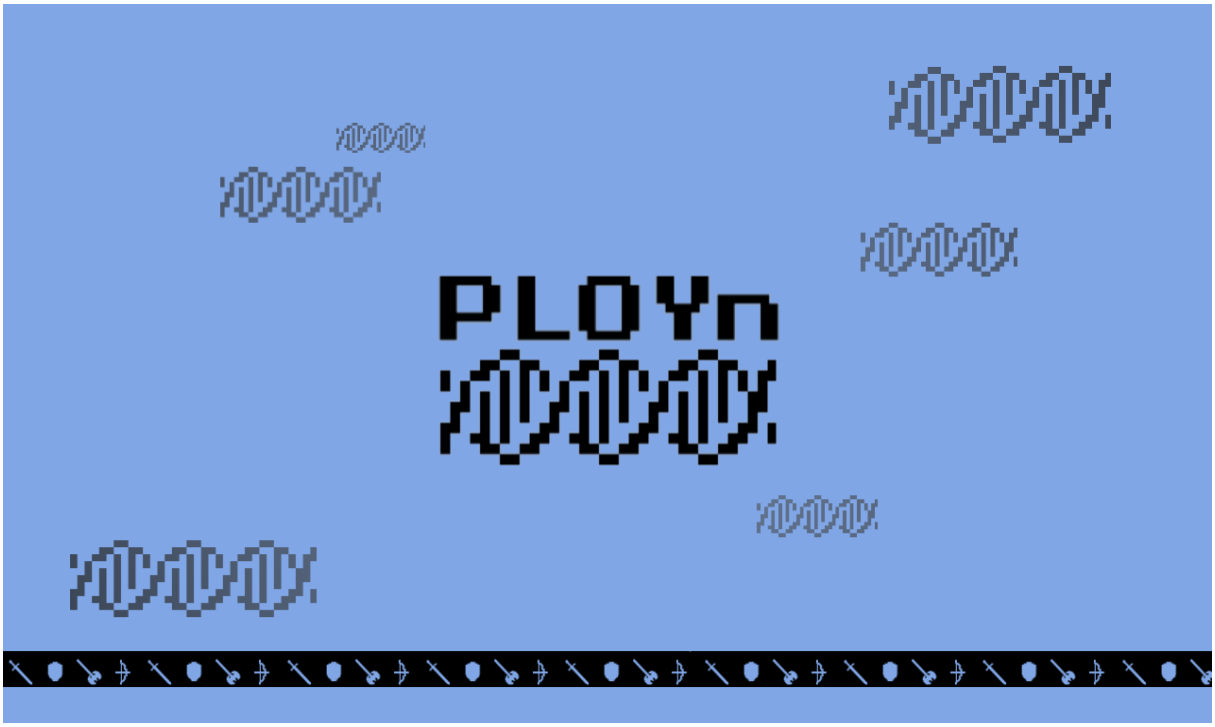


Ilustración 46: *Prototipo de fondo de pantalla*

11.3.- Monetización

La monetización es una parte indispensable al lanzar un videojuego. Al fin y al cabo, se trata de sacar dinero del producto sobre el que hemos pasado gran parte de nuestro tiempo desarrollando. Hay varias formas de ganar dinero a través de los videojuegos, queda en manos de la empresa y de su ética el explotar alguna, varias o todas las posibles.

- Anuncios
    - Es posible exigir el visionado de un anuncio para continuar jugando o para desbloquear cierta parte de la jugabilidad que te permita avanzar. Es una buena forma de desentendernos de la explotación comercial del videojuego, aunque los ingresos dependen del número de usuarios, independientemente de que acaben fidelizados o no.
- Venta de contenido jugable
    - Permitir al jugador que pague por contenido del juego. Este contenido puede ser posible obtenerlo invirtiendo horas de juego o no, dependiendo de su importancia para la jugabilidad.
- Personalización
    - Permitir al jugador personalizar sus personajes, enemigos e interfaz a través del pago directo por estos elementos.
- Eliminar anuncios
    - Al igual que incluimos los anuncios, es posible obligar al jugador usual a pagar si desea desactivarlos.
- Compra directa del producto
    - Incluir un precio al videojuego y obligar al usuario a pagarlo para jugarlo.
- *Merchandising*
    - Si se espera o se encuentra cierto seguimiento, es posible hacer *merchandising* y ponerlo a la venta para sacar más partido económico al producto.

El estudio sobre qué monetización es preferible para este videojuego concreto parece más conveniente realizarlo sobre iteraciones más tardías, utilizando test A|B y testeos de la jugabilidad hasta dar con la tecla.

11.4.- Publicación

Tras confirmar que todo funciona como debe y planear las diferentes estrategias de diseño, marketing y comercialización, llega el momento de la publicación del videojuego.

Esta etapa, y el coste que puede suponer, depende de nuestros medios y de la plataforma elegida durante el desarrollo. Publicar juegos para consolas y/o grandes plataformas de ordenador como *Steam* **[26]** suele resultar más caro que en el caso de juegos para móviles (En la *PlayStore*, por ejemplo, es gratis) o para navegadores web. No obstante, la publicación en consolas como *Xbox, PlayStation* o *Nintendo*, puede aumentar exponencialmente el prestigio de la desarrolladora y su producto, además de darle un gran impulso en cuanto a número de ventas.

La decisión sobre la plataforma en la que lanzaremos el juego es una buena tarea futura teniendo en cuenta que aún no se ha decidido con total seguridad los dispositivos sobre los que el juego podría tener más éxito.

12.- CONCLUSIÓN

12.1.- Conclusiones obtenidas del proyecto

Al término de esta memoria, se ha conseguido un prototipo de un videojuego. En el argot del sector, estaríamos hablando de un juego en fase alfa, pues pese a mostrar una buena parte de los rasgos de su jugabilidad, no se trata de una versión abierta al público ni su principal función es la búsqueda de errores o funcionalidades nuevas.

En el momento en el que comenzó el proyecto, uno de mis mayores miedos era compaginar la ejecución de un algoritmo genético efectivo en un videojuego, donde cada segundo cuenta para encantar al jugador. A pesar de las no pocas advertencias sobre la posibilidad de que la ejecución del primero fuera lenta y afectase en gran medida a la jugabilidad, creo que se ha obtenido una inteligencia artificial, si se me permite usar dicho término, que trabaja a la velocidad esperada y con la eficacia requerida. En mi opinión, crear y configurar las heurísticas que guían el comportamiento me hecho, no solo aprender, sino interesarme mucho más sobre este ámbito de computación.

Además, el haber seguido todas las etapas de diseño de software en un proyecto de estas características creo que ha sido muy favorable tanto para la calidad del mismo como para mi experiencia en el desarrollo de similares.

Desde que era pequeño, mi deseo siempre fue dedicar mi vida laboral al diseño y desarrollo de videojuegos, no como producto, sino como arte. Tras el desarrollo de este proyecto, ese deseo, lejos de mitigarse, ha sido encumbrado. Tras tantas horas dedicadas a este proyecto siento que este proyecto, que ha sido tanto personal como académico/profesional, no ha hecho más que empezar.

12.2.- Posibles mejoras

Para finalizar la redacción de este documento, me gustaría dejar constancia de algunas mejoras que entiendo que podrían ser interesantes para el futuro del proyecto. No obstante, y al tratarse de un videojuego, dichas mejoras quedan sujetas a la subjetividad del autor como diseñador de videojuegos.

1. Añadir la posibilidad de realizar partidas multijugador. Cada jugador tomaría el control de un personaje en batallas 1 contra 1. Dada la naturaleza arcade que ha terminado por tener el videojuego, parece un punto muy necesario si se pretende el lanzamiento y posterior monetización del mismo.
2. Crear de una base de datos para el almacenamiento de todos los datos persistentes, tales como acciones, personajes, animaciones, modelos 3D, historial de mejores individuos escenarios, etc. Este punto me parece básico e imprescindible si se quiere incluir el anterior.
3. Dar la posibilidad al jugador de crear y personalizar personajes. Favorecer el máximo este punto elevaría conceptos como la *Player Story* o *la Player Agency* **[21]** y daría una mejor experiencia de juego.
    a) Personalizar los modelos 3D y las características del personaje sería la mejor opción.
    b) Dar una lista de personajes entre los que el jugador pueda elegir sería el plan b si el desarrollo de la anterior opción fuese demasiado costoso.
4. Permitir que los personajes suban de nivel con el paso de los combates, aprendan acciones y estrategias y mejoren sus atributos. De no realizarse este punto, hay bastantes posibilidades de que el jugador sienta que se trata de un videojuego demasiado repetitivo o aburrido.
5. Incluir más escenarios, de diferentes dimensiones y características, y dar la posibilidad a los jugadores de poder jugar en todos ellos. También parece un punto necesario antes del lanzamiento del videojuego.
6. Incluir combates múltiples, bien todos contra todos en escenarios de mayor dimensión o de un jugador contra varios enemigos. Aportaría variedad a la jugabilidad.

7. Incluir un menú de opciones desde el que editar aspectos como la posición de la cámara.
8. Exportar el videojuego para plataformas móviles y/o resto de plataformas disponibles, adaptando y probando la interfaz y los parámetros del algoritmo genético en base a la potencia media de los dispositivos. A mi parecer, la naturaleza actual del videojuego aumentaría el porcentaje de éxito en móviles. No obstante, los cambios que se produjeran antes del lanzamiento podrían cambiar este parecer si se aumentase el factor rol.

En general, las mejoras deberían ir dirigidas a mejorar el videojuego como producto, aportándole el mayor valor posible a ojos del jugador, y al fin y al cabo cliente, objetivo.

13.- BIBLIOGRAFÍA

[1] ***Algoritmos genéticos*** (2007). *Jorge Arranz de la Peña y Antonio Parra Truyol*; Universidad Carlos III – http://www.it.uc3m.es/jvillena/irc/practicas/06-07/05.pdf

[2] ***Inteligencia Artificial: Un enfoque moderno*** *(1994) – Peter Norvig y Stuart J. Russel. ISBN 9781292153964*

[3] ***Introducción a la inteligencia artificial***. *Bruno López Takeyas*; Instituto Tecnológico de Nuevo Laredo -- http://www.itnuevolaredo.edu.mx/takeyas/Articulos/Inteligencia%20Artificial/ARTICULO%20Introduccion%20a%20la%20Inteligencia%20Artificial.pdf

[4] ***Wikipedia: Inteligencia artiicial*** – https://es.wikipedia.org/wiki/Inteligencia_artificial

[5] ***Gartner Top 10 Technology Trends****, Gartner Consultory – https://www.gartner.com/*

[6] ***Wikipedia: Metaheurísticas*** – https://es.wikipedia.org/wiki/Metaheur%C3%ADstica

[7] ***Introducción a los algoritmos metaheurísticos*** *(2006).* Francisco Herrera; Grupo de Investigación "*Soft Computing and Intelligent Information Systems*" http://www-2.dc.uba.ar/materias/metah/TutorialHerrera2006.pdf

[8] ***El origen de las especies (1859)****,* Charles Darwin – http://www.rebelion.org/docs/81666.pdf

[9] ***La computación evolutiva y sus paradigmas*** *(2011). Adriana María Iglesias Solano y Andrés Bezaleel Iglesias Carbonell*; Universidad Simón Bolívar.

[10] ***Algoritmos genéticos y computación evolutiva*** *(2004)*. Adam Marczyk – http://the-geek.org/docs/algen/

[11] ***Web del motor GameMaker*** – https://www.yoyogames.com/gamemaker

[12] ***Web del motor RPG Maker*** – http://www.rpgmakerweb.com/

[13] ***Web del motor Unreal Engine*** – https://www.unrealengine.com/en-US/what-is-unreal-engine-4

[14] ***Web del motor CryEngine*** – https://www.cryengine.com/

[15] ***Inteligencia Artificial en videojuegos***. Javier Alcalá; Ciclo de conferencias Game Spirit 2 – http://www.flasentertainment.com/blog/ia.pdf

[16] ***Teoría de juegos: Definiendo los videojuegos*** *(2014)***.** http://blogs.vandal.net/64752/vm/191831522014

[17] ***SLAY THE DRAGON: Writing great video games.*** *(2015)* – Robert Denton Bryant & Keith Giglio. ISBN: 978-1615932290

[18] ***The Art of Game Design*** *(2008)*, Jesse Schell. ISBN: 978-1-4665-9864-5

[19] ***Página de referencia para el lenguaje C#*** – https://docs.microsoft.com/

[20] ***Página de referencia de Unity –*** https://docs.unity3d.com

[21] ***Diseño narrativo y Guion de videojuegos***, Ángel Codón Ramos.

[22] ***The four interactive forms***, Keith Burgun – http://keithburgun.net/interactive-forms

[23] ***Motores de videojuegos: ¿Cómo Funcionan?***, Unity – https://unity3d.com/es/what-is-a-game-engine

[24] ***Web del motor Unity*** – https://unity3d.com/es

[25] ***Generación por procedimientos*** – https://es.wikipedia.org/wiki/Generaci%C3%B3n_por_procedimientos

[26] ***Ingeniería del software, Un enfoque práctico***. Séptima edición, Robert S. Pressman. ISBN: 978-60715-0314-5

[27] ***Web de Scrum en español.*** – https://proyectosagiles.org/que-es-scrum/

Web de Steam Direct – https://partner.steamgames.com/steamdirect

14.- ANEXOS

14.1.- Anexo 1. Material utilizado

El desarrollo de este trabajo ha sido realizado sobre el ordenador personal del alumno, un Asus VivoBook del año 2010 con un procesador Intel® Core™ i5-3317U de 1,70GHz, una memoria RAM de 4GB y una tarjeta gráfica NVIDIA GEFORCE GT 635M de 2 GB. El sistema optativo es un Windows 10 de 64 bits.

Para el desarrollo del software, y tal y como se menciona en la memoria, se ha utilizado la herramienta *Unity3D* junto con *Mono DEVELOP*, herramienta que permite la edición de los scripts en C# que hemos implementado. También hemos hecho uso de la herramienta online *Pixlr* para la creación y edición de imágenes que se utilizan tanto en el videojuego como en la memoria. La redacción de esta última se ha realizado sobre *LibreOffic*e, conjunto de aplicaciones de ofimática creado por *The Document Foundation*.

Para la creación de la escena, se han utilizado los siguientes paquetes de assets gratuitos, descargados desde la tienda de Unity:

- https://assetstore.unity.com/packages/3d/environments/low-poly-pack-94605
- https://assetstore.unity.com/packages/3d/environments/low-poly-free-pack-58821
- https://www.assetstore.unity3d.com/en/?stay#!/content/110455

De estos paquetes se ha utilizado, mayoritariamente, modelos 3D que permitieran crear una escena visualmente atractiva. En algunos casos, se ha utilizado la herramienta *Blender* para reajustar o retocar algunos de estos modelos.

En ningún momento se ha hecho uso de ninguna herramienta, sistema, plan o producto de pago durante la redacción del videojuego ni la memoria.

14.2.- Anexo 2. prototipo a papel

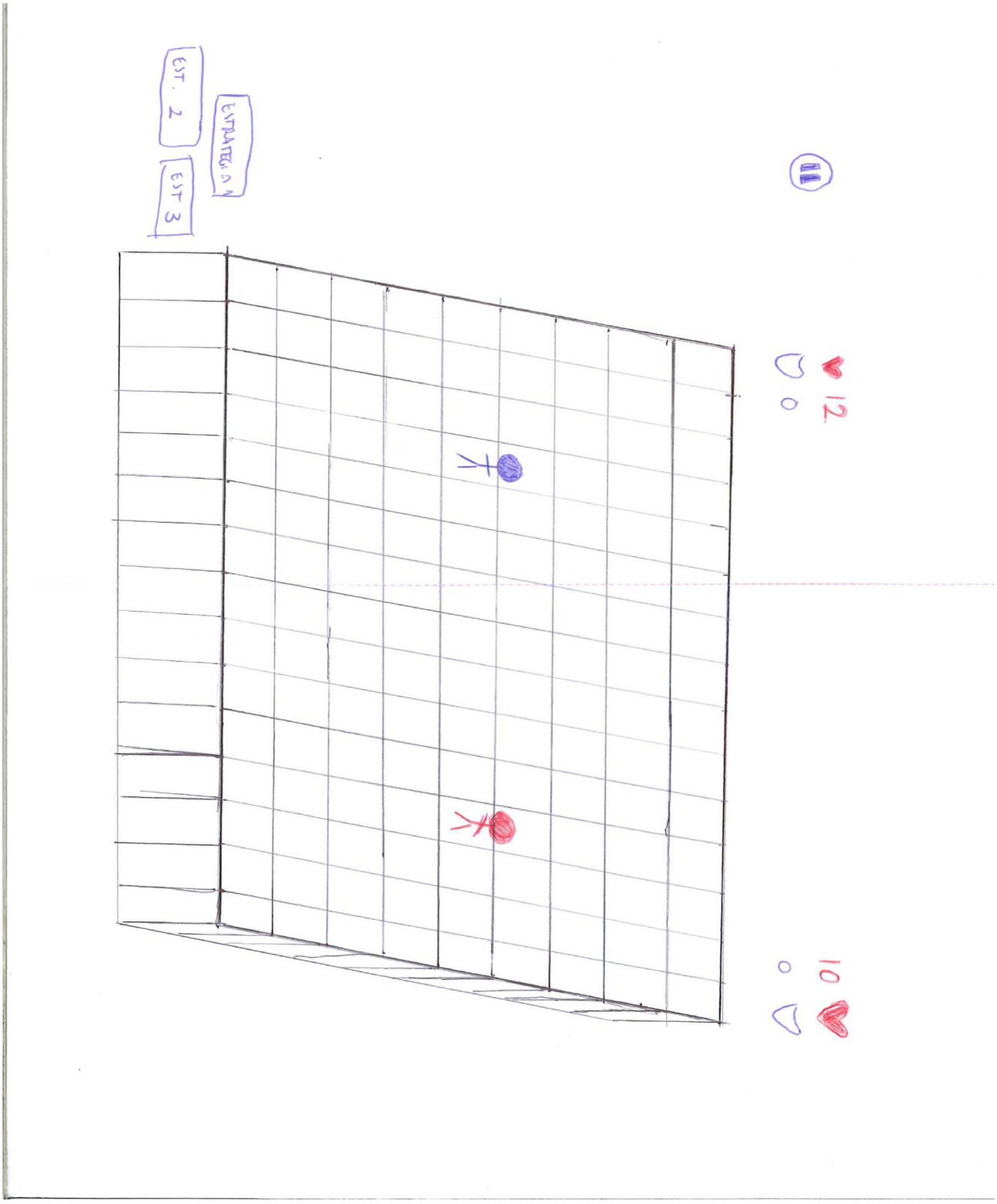


***Ilustración 47**: Dibujo que representa la interfaz esperada.*

14.3.- Anexo 3. Manual de usuario

Para iniciar el videojuego, solo tenemos que hacer clic sobre el ejecutable, que debe encontrarse en la misma carpeta que el fichero UnityPlayer.dll y la carpeta PLOYn_Data.

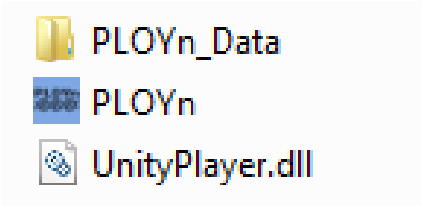


Ilustración 48: *Ficheros necesarios para ejecutar la aplicación.*

Una vez que el juego ha arrancado, nos aparecerá un panel de configuración, donde podremos elegir la resolución de pantalla que más se ajuste a la nuestra entre otros parámetros.

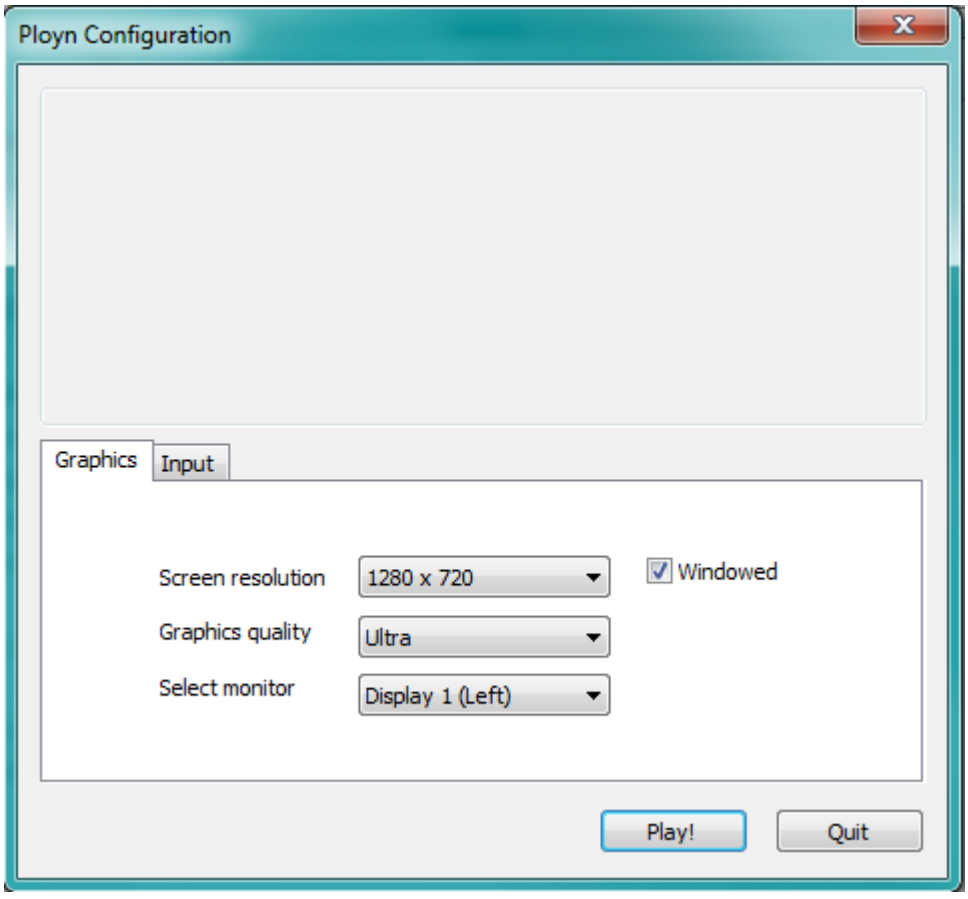


Ilustración 49: *Panel de configuración visible al iniciar el juego*

Una vez configuremos la resolución y el resto de apartados comenzará la ejecución del videojuego. Concretamente pasaremos a la siguiente pantalla de inicio.

Ilustración 50: *Pantalla de inicio del videojuego.*

Desde aquí, podemos elegir el tipo de partida que queremos, tanto si es Para un solo jugador (*Single Player*) como si queremos enfrentarnos a otro jugador en modo local. (*Versus*). En este caso elegimos versus, ya que la versión para un jugador trabaja de manera similar de cara al usuario.

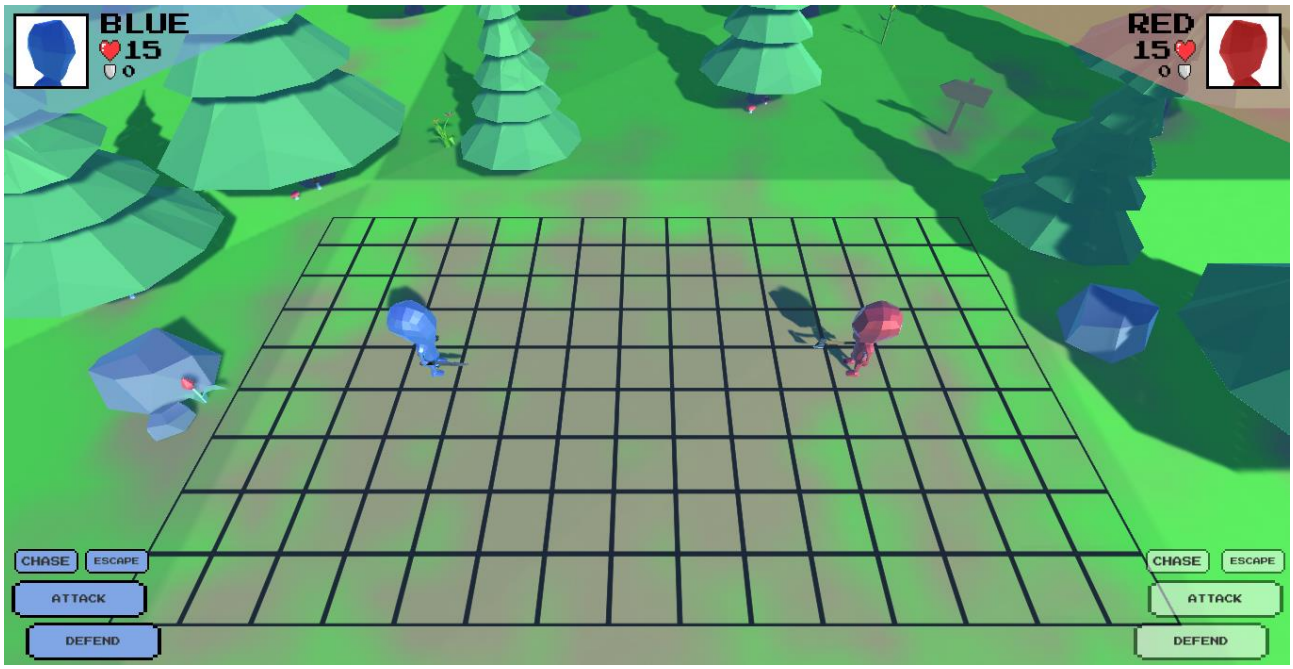


Ilustración 51: *Partida en modo Versus.*

En esta pantalla podemos ver el nombre, la vida y el escudo de nuestro personaje en las esquinas superiores, a la izquierda el personaje 1 y a la derecha el personaje 2. Además, podemos consultar a través del juego la posición de nuestro personaje y el rival en cada momento dentro del escenario. Para elegir estrategia en nuestro turno, tendremos que irnos a la esquina inferior correspondiente a nuestro personaje y hacer clic sobre uno de los botones.

Ilustración 52: *Botones representativos de las estrategias*

Si los botones se muestran en color blanco y con opacidad media (dejando ver en parte lo que hay detrás), entonces es posible que no sea aun nuestro turno o que ya hubiéramos elegido una estrategia y el personaje esté ejecutándola.

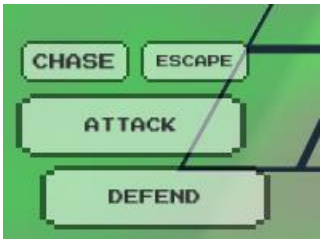


Ilustración 53*: Botones inactivos.*

Si en algún momento de la partida no pudiésemos ver alguna parte de la escena, es posible girar la cámara sobre esta. Para ello, solo tenemos que mantener clic derecho con el ratón sobre cualquier punto y desplazar el puntero horizontalmente.

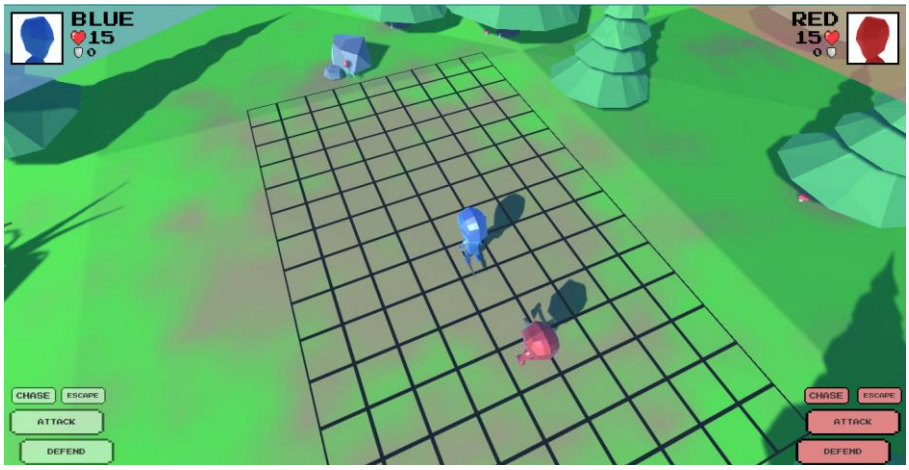


Ilustración 54: *Escena tras girar la cámara.*

La partida proseguiría hasta que uno de los dos cayera derrotado. Tras esto, veríamos un mensaje con el nombre del vencedor.

Ilustración 55: *Pantalla cuando el personaje azul gana.*

Pulsando con el ratón sobre el mensaje del ganador nos permite volver a la pantalla de inicio, donde podríamos elegir volver a jugar o incluso cambiar de modo.