Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Una metodología basada en patrones de diseño de software para una óptima implementación de mecánicas en videojuegos.

Agosto 2023

Tesis para obtener el grado de Licenciatura en Ingeniería en Ciencias de la Computación

Autor:
Erick Blas Ruiz Ponce
Director de tesis:
Juan Carlos Conde Ramírez

Índice general

1. Introducción **5**

2. Estado del arte **8**

2.1. El juego y las reglas 8

2.2. Mecánicas famosas en juegos 2D 9

2.3. Mecánicas famosas en juegos 3D 13

2.4. UML 15

2.5. Patrones de diseño de software 15

3. Marco teórico **17**

3.1. Metodologías usadas para el desarrollo de videojuegos 17

3.1.1. Scrum 17

3.1.2. XP 18

3.1.3. Espiral 19

3.1.4. Cascada 20

3.2. Géneros de videojuegos 21

3.2.1. Shooters 21

3.2.2. Role Playing Game (RPG) 23

3.2.3. Pelea 24

3.2.4. Deportivos 24

3.2.5. Plataformas 24

3.2.6. Aventura 25

3.2.7. Terror 25

3.3. Tipos de jugadores 25

3.4. Curva de dificultad 27
3.5. Recursos y entidades 29
3.6. Mecánicas 29
3.7. Mecánicas centrales 30
3.8. Mecánicas secundarias 31
3.9. Gameplay Periférico 31
3.10. Tipos de mecánicas 32
3.10.1. Acciones del jugador 32
3.10.2. Economía interna 32
3.10.3. Progresión 34
3.10.4. Condiciones de victoria y derrota 36
3.11. Machinations 37
3.12. Diagramas UML 41
3.13. Patrones de diseño de software 44

4. Patrones de diseño de software aplicados en el desarrollo de videojuegos 46
4.1. Patrones de creación 46
4.1.1. Prototype 47
4.1.2. Singleton 52
4.2. Patrones estructurales 55
4.2.1. Flyweight 55
4.2.2. Spatial Partition 59
4.3. Patrones de comportamiento 64
4.3.1. Command 64
4.3.2. Game Loop 67
4.3.3. Observer 68
4.3.4. State 72
4.3.5. Update 80

5. Metodología propuesta 82
5.1. Identificar tipo de mecánicas acorde al tipo de juego 85
5.2. Determinar los tipos de jugadores objetivo 86

5.2.1. Delimitar los tipos de jugadores 89
5.3. Diseño de las mecánicas 89
5.3.1. Planteamiento de las mecánicas centrales 89
5.3.2. Planteamiento de las mecánicas secundarias 91
5.3.3. Identificar los recursos y entidades para las mecánicas planteadas 92
5.4. Prototipado 92
5.5. Selección de patrones de diseño de software 94
5.6. Programación 96
5.7. Testeo y correcciones 96

6. Ejemplo **98**
6.1. Identificar tipo de mecánicas acorde al tipo de juego 98
6.2. Determinar los tipos de jugadores objetivo 99
6.3. Diseño de las mecánicas 99
6.3.1. Planteamiento de mecánicas centrales 99
6.3.2. Planteamiento de mecánicas secundarias 102
6.3.3. Identificar los recursos y entidades para las mecánicas planteadas 104
6.4. Prototipado 106
6.5. Selección de patrones de diseño 109
6.6. Programación 110
6.6.1. Diagrama UML 111
6.6.2. Mecánicas 114
6.6.3. Patrones de diseño de software 125
6.6.4. Assets usados para el ejemplo 130
6.7. Testeo y correcciones 130

7. Conclusiones y trabajo futuro **132**

8. Anexo: Herramientas de Unreal Engine **134**

9. Referencias **136**
Bibliografía 136

Enlaces de referencia 137

Capítulo 1

Introducción

El juego es de las primeras actividades realizadas por el ser humano y una de las más importantes para su desarrollo en las primeras etapas de la vida. El psicólogo suizo Jean Piaget clasificó el juego en tres etapas, en relación con el desarrollo de la inteligencia; siendo la segunda y la tercera las que más nos incumben para este trabajo. Las principales características de cada etapa son las siguientes:
La primera de ellas es conocida como "Juego Simbólico". En ésta, el niño se enfoca en juegos con elementos de ficción para adaptar a su mundo infantil los rasgos del entorno que desconoce y será hasta que empiece a jugar con otros niños que logrará desplazarse cada vez más hacia la realidad. Dicho avance permitirá al pequeño entrar al siguiente estrato, el "Juego Reglado": el uso y respeto por las reglas demuestran que el niño ha internalizado lo social, viéndose capaz de disfrutar de ello. Su uso le permitirá pasar de jugar al juego, es decir, tendrá las bases y fundamentos necesarios para una convivencia libre de conflictos; conocimiento de las acciones permitidas, límites, condiciones para poder ser el ganador y, lo más importante, diversión a través de la creación y alcance de objetivos que requieran de cooperación, estrategias o enfrentamientos pacíficos.[FA99]
En los juegos creados formalmente, existe lo que vendría siendo la evolución de las reglas: las mecánicas. Las mecánicas son el conjunto de reglas llevadas al máximo nivel de detalle, estas le indican al jugador las acciones que puede realizar, la forma en que interactuará con el mundo que lo rodea, su economía,

el comportamiento de los NPC (Non Playable Character), enemigos, entre otros aspectos.

Para que pueda existir la interacción entre los jugadores y las mecánicas, el jugador deberá conocer las acciones que debe realizar con el control/teclado junto con lo que ocurrirá dentro del juego en respuesta a su acción, por otro lado, los diseñadores junto con los programadores se encargarán de tomar en cuenta todos los factores necesarios para que dentro del juego todo ocurra adecuadamente. Por ejemplo, dentro del juego Crash Bandicoot el jugador puede oprimir un botón para provocar que el protagonista realice un salto, a la par que puede usar el joystick para hacer que el salto se dirija a una dirección en específico, pero, para que esto ocurra, antes hubo un trabajo de diseño y programación que permite responder a las preguntas ¿Cuánto podrá saltar el personaje? ¿De cuánto será la gravedad en la zona donde se encuentra? ¿Podrá cambiar de dirección mientras se encuentra en el aire? ¿Con cuánta fuerza realizará el salto? ¿Podrá correr para saltar una distancia mayor?, entre otras preguntas.

Este tipo de preguntas junto con muchas otras posibles requieren de un proceso de planificación para poder ser identificadas, esto debido a que se requerirá de tener una idea bastante clara de lo que se desea realizar, el público objetivo, los elementos y personajes junto con la forma en que interactuarán entre sí, entre otros aspectos. Con este objetivo en mente, se deberá seguir una metodología de trabajo que permita a los diseñadores tener una guía bien estructurada de los pasos a seguir para detallar sus ideas, identificar los posibles errores dentro de su diseño y no dejar cabos sueltos, esto a fin de facilitarle a los artistas y programadores el trabajo de plasmar las ideas en el juego y poder obtener resultados satisfactorios.

Tener unas mecánicas bien detalladas será de increíble ayuda para lograr la diversión en los jugadores, pero no es la única parte importante de la planeación que debe ser tomada en cuenta. Aunque tengamos las mejores mecánicas de la historia plasmadas en papel, hará falta una buena ejecución de estas por parte de los programadores para que lleguen a ser satisfactorias; esto debido a que, si se genera un *gameplay* con bajones de FPS (Frames per second), lleno de *bugs* (errores dentro del juego) o con fallos en la interacción de los elementos del mundo virtual, el jugador se frustrará y terminará teniendo una mala experiencia.

Una de las herramientas más útiles para poder optimizar los procesos en cuanto a tiempo de desarrollo y recursos computacionales a la hora de ejecutar el juego son los patrones de diseño. Estos brindan soluciones habituales y óptimas a problemas que pueden ser encontrados comúnmente a la hora de programar software, estas recomendaciones sobre cómo solucionar los problemas no brindan las mismas herramientas que una función de una librería, sirven como guía para que el programador sea capaz de tener una base con la cual empezar a trabajar.

Tomando en cuenta lo anteriormente planteado, se espera que este trabajo brinde una metodología de trabajo con la cual cualquier equipo de desarrollo sea capaz de planear y diseñar mecánicas de juego de una forma óptima, a la vez que aprenden sobre los patrones de diseño de software más usados en el desarrollo de juegos y su correcta implementación, dando como resultado un sistema mejor estructurado y un código más eficiente.

Capítulo 2

Estado del arte

2.1. El juego y las reglas

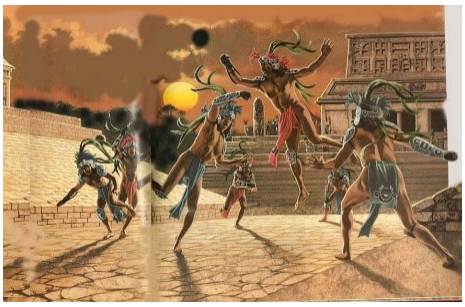
El juego no es una acción humana que surgió en los últimos años, se han encontrado indicios de que el hombre del paleolítico realizaba juegos con motivos religiosos, de entretenimiento y para mejorar las destrezas físicas necesarias para la supervivencia en esa época. Tiempo después, estos juegos empezaron a evolucionar en cuanto a complejidad y estructura, lo que los llevó a generar una de las partes más importantes del juego, las reglas. [AMA]

Aproximadamente en el año 3000 A.C, los juegos habían evolucionado al punto que apareció el que se considera el primer juego de tablero, el juego Real de Ur (Figura 2.1a), este ya tenía sus propias reglas y podía requerir de un alto nivel de estrategia para obtener la victoria. Siglos más tarde, algunas civilizaciones como los mayas o los egipcios desarrollaron sus propios juegos de pelota, con reglas, equipos, canchas e inclusive tenían árbitros para regular algunos de los sucesos que podían ocurrir durante los juegos. [A.T]

Las reglas planteadas permitían a los jugadores tener un convenio sobre las condiciones de victoria, lo que estaba y no estaba permitido hacer e inclusive los castigos en caso de no cumplir con estos convenios durante el juego. Retomando el ejemplo del juego de pelota maya (Figura 2.1b), podemos encontrar la regla de que los jugadores solamente debían golpear la pelota con la cadera, tenían la

restricción de no dejar caer la pelota al suelo y la condición de victoria de meter la pelota dentro de los aros designados.

(a) Juego de Ur



(b) Juego de Pelota Maya

Figura 2.1: Comienzos del juego

2.2. Mecánicas famosas en juegos 2D

En 1958, William Higginbotham creó el que se considera como el primer videojuego de la historia, Tennis for two (Figura 2.2a), este simulaba dentro de un osciloscopio una partida de tenis en las que los jugadores debían golpear la pelota con un botón y esto provocaría su cambio de movimiento dependiendo del ángulo y la velocidad de esta. Con este título comenzó la época de los videojuegos en dos dimensiones y con ella la creación de varias de las mecánicas más importantes de la historia de los juegos. Algunas de estas mecánicas destacables de esta época son:

- El movimiento: Parte fundamental de casi cualquier videojuego, esta mecánica podemos observarla desde el famoso juego Pong de Atari o en Space Panic.
- El salto: Otra de las mecánicas que más se utiliza hasta día de hoy es la de saltar para alcanzar distintas zonas de los niveles, derrotar enemigos o esquivar diversos obstáculos. Una de las primeras apariciones de esta mecánica se puede observar en Donkey Kong de 1981.

- Combate cuerpo a cuerpo: Las mecánicas de combate cuerpo a cuerpo permiten a los jugadores y a los enemigos enfrentarse a una distancia corta, dependiendo del juego, esto puede ser por medio de algún arma o con alguna parte de su cuerpo.

- Combate a distancia: Permite al jugador y a los enemigos realizar ataques, aunque no se encuentren cerca unos de los otros, esto se consigue por medio de arcos, armas, magia o algún poder con el que cuente el personaje.

- Puntuación: Consiste en incrementar o decrementar una cantidad numérica o dar una nota en base a la habilidad demostrada por el jugador a lo largo de un segmento del juego.

- Medidor de salud: Esta mecánica está representada generalmente por una barra o un contador y permite visualizar el daño que podría resistir nuestro protagonista o algún enemigo. Al llegar la vida del jugador a cero, este tendría una penalización como regresar a cierta zona del nivel, perdería una vida en caso de que el juego cuente con una cantidad de vidas o perdería algunos objetos dependiendo del juego.

- Vidas: Representa con un valor numérico la cantidad de intentos que tiene el jugador para poder intentar una zona del juego, el jugador generalmente puede conseguir más vidas al tomar algún objeto dentro de los niveles y, al perder todas, sería penalizado regresando al comienzo del juego o al comienzo de algunas zonas en concreto del mismo.

- Recolección de objetos: Esta mecánica consiste en que el jugador puede recoger elementos repartidos en los niveles para así poder conseguir ventajas como un aumento de salud, aumento en la cantidad de vidas, consumibles o elementos de la economía interna del juego que le permitiría intercambiarlos por otros elementos dentro del mismo.

- Temporizador: Mecánica que, al dar un tiempo límite para realizar cierta acción o llevar un registro del tiempo que fue requerido para completarla, agrega dificultad y un grado de competitividad al juego. Este tiempo permite hacer un top de los jugadores, premiar al jugador dependiendo del

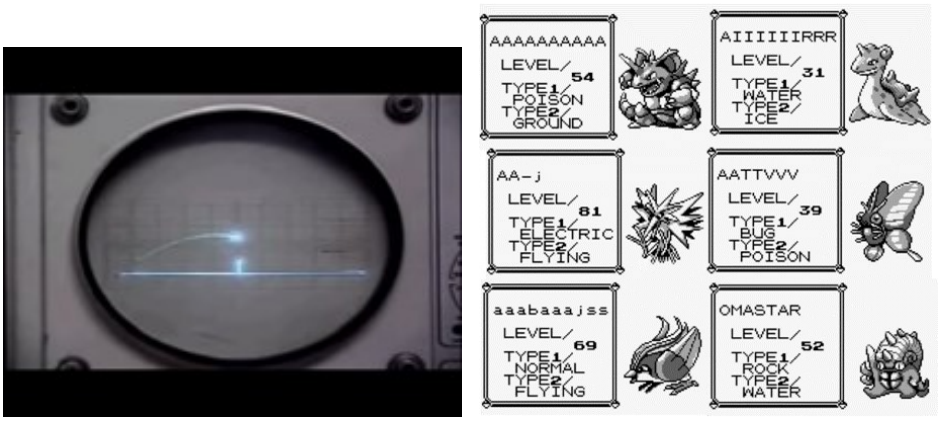
tiempo que requirió o castigarlo en caso de no terminar de realizar una acción en el tiempo planteado.

- Progresión de niveles: Una mecánica común en los géneros RPG y JRPG con la que se les permite a los jugadores aumentar los atributos de los personajes en base a un valor numérico que resume "el nivel de poder" del mismo. Estás mecánicas son muy comunes en juegos como Earthbound, Final Fantasy, Dragon Quest o Pokémon (Figura 2.2b).

- Cambios en valores de los atributos: Esta mecánica está muy ligada a la progresión de niveles, ya que, es común que cuando un personaje aumente de nivel, también aumenten algunos de sus atributos, como la vida, el ataque, la defensa, resistencia a la magia, etc. Otras maneras en las que comúnmente se aumentaban o disminuían los valores de los atributos es por medio de potenciadores, objetos y habilidades de personajes con las que el jugador podría darle ventajas a sus personajes o desventajas a los enemigos o personajes de otros jugadores.

- Combos: Mecánica que consiste en oprimir una serie de botones (que podría incluir el movimiento de un joystick) en un orden específico para así realizar distintos movimientos dependiendo de los personajes. Esto permite aumentar enormemente las posibilidades del *gameplay* a la vez que les permite a los jugadores más experimentados hacer jugadas más complicadas. Los combos son una mecánica muy común en juegos beat'em up y de pelea, como es el caso de Street Fighter, Streets of Rage o The King of Fighters.

En la Figura 2.3 se puede observar el comienzo del juego Super Mario Bros, en este juego podemos observar varias de las mecánicas que fueron mencionadas, como es el caso del movimiento, salto, puntuación, vidas, recolección de objetos y temporizador.



(a) Tennis for two

(b) Niveles en Pokemon

Figura 2.2

Figura 2.3: Super Mario Bros

2.3. Mecánicas famosas en juegos 3D

El salto a los videojuegos a la tercera dimensión fue un proceso largo y escalonado, esto podemos observarlo en los diversos intentos de crear la sensación de estar jugando un juego en 3D por medio de imágenes en dos dimensiones usando efectos visuales o algunas técnicas como el Raycasting, pero, no fue hasta la aparición de juegos como Virtua Fighter o Descent que se empezó a utilizar polígonos que generaban juegos que realmente estaban en tres dimensiones [4]. Esta nueva dimensión trajo consigo una evolución en la mayoría de las mecánicas anteriormente mencionadas junto con algunas nuevas, como lo son:

- Movimiento de cámara: Una de las mecánicas más comunes en juegos en tres dimensiones es la que permite mover la cámara para así poder observar distintas partes del escenario, resolver acertijos, luchar contra enemigos, entre otras cosas.

- El sistema de cobertura: Permite al jugador moverse rápidamente entre distintos elementos del escenario para poder cubrirse de los disparos enemigos, este sistema fue popularizado por la saga de Gears of War (Figura 2.4a).

- Escalar: Mecánicas que podrían considerarse una evolución del salto en plataformas y le permite al jugador escalar por distintas superficies del nivel y realizar saltos más complejos al agregar la tercera dimensión. Estas mecánicas son comunes en juegos como Tomb Raider, Assassin's Creed o Uncharted.

- Combate cuerpo a cuerpo en primera persona: Evolución del combate cuerpo a cuerpo en dos dimensiones que le da al jugador una inmersión más grande al poder observar las manos del personaje y las armas de este como si las estuviera viendo en persona.

- Z-Targeting: Una mecánica nacida en el juego The Legend of Zelda Ocarina of time (Figura 2.4b) que les permite a los jugadores poder fijar algún objetivo para poder concentrar su atención en él, esto le permite al jugador atacar o defenderse de ese enemigo en concreto de una manera más

sencilla.

- Disparos en tres dimensiones: Con la inclusión de esta nueva dimensión se aumentaron las posibilidades a la hora del combate a distancia debido a la habilidad de mover la cámara en distintas direcciones y el poder mover a los enemigos en más direcciones.
- Hogueras: Esta mecánica permite al jugador descansar en zonas especiales del juego para poder recuperar la salud, aumentar niveles y demás acciones a cambio de causar la reaparición de los enemigos comunes de la zona.
- Vuelo en tres dimensiones: La tercera dimensión permitió crear mejores experiencias en las mecánicas de vuelo al permitirle a las naves o personajes con la habilidad de volar la posibilidad de aumentar las direcciones posibles para sus movimientos.
- Perseguidores: Se trata de un enemigo o conjunto de enemigos que son imposibles de eliminar por el jugador hasta cierto punto de la historia, estos estarán constantemente acechándolo y obligándolo a cambiar su estrategia de juego. Esto se popularizó principalmente por juegos como Resident Evil 2 y posteriormente por la saga Outlast.

(a) Sistema de cobertura en Gears of War (b) Z-Targetting en The Legend of Zelda

Figura 2.4

2.4. UML

El Lenguaje de Modelado Unificado (Unified Modeling Language) es un lenguaje que permite a los desarrolladores crear representaciones gráficas que les ayuden a visualizar las clases que requerirán a la hora de desarrollar software, los atributos con los que contarán, las funciones que realizarán y las relaciones que habrá entre las clases.
En 1997 el lenguaje UML fue propuesto por James Rumbaugh, Grady Booch e Ivar Jacobson; esto con el objetivo de crear un estándar unificado debido al gran número de lenguajes de modelado que ya existían en esa época. Esta propuesta dio origen a la que fue la versión 1.0 de UML y que, con el paso de los años, fue evolucionando hasta su versión 2.5.1 que es la que se utiliza actualmente (y que cuenta con una gran cantidad de variaciones adaptadas a distintos enfoques). [Won22]

2.5. Patrones de diseño de software

En el año de 1994 un conjunto de cuatro programadores llamados Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm publicaron el libro Design Patterns: Elements of Reusable Object-Oriented Software [EGV94] basándose en la idea que tuvo un arquitecto llamado Cristopher Alexander en los años setenta. El objetivo de este libro fue dar a conocer posibles soluciones para problemas comunes dentro del área de la programación de forma similar a lo que hizo Alexander en el área de la arquitectura.
Dentro del libro Design Patterns se escribieron veintitrés patrones de diseño de software divididos dentro de tres secciones dependiendo de la funcionalidad del patrón, las cuales fueron llamadas Patrones Creacionales, Patrones Estructurales y Patrones de Comportamiento.
Desde entonces han sido inventados más patrones de diseño en base a necesidades que han surgido con el paso de los años y algunos de los patrones ya existentes han ido aumentado o disminuyendo en su uso, algunos de ellos se han vuelto tan populares que se usan cotidianamente dentro de distintos ámbitos sin saber que lo que se está haciendo tiene un nombre.

Actualmente existen diversos libros y páginas web que hablan sobre patrones de diseño, algunos de estos incluso se enfocan en el uso de patrones en áreas específicas de la programación, como es el caso de Game Programming Patterns de Robert Nystrom [Nys14]. En este libro se mencionan varios de los patrones de diseño de software que pueden ser aplicados dentro de los videojuegos tanto por principiantes que apenas están comenzando en esta área como por programadores más avanzados que buscan optimizar sus juegos.

Capítulo 3

Marco teórico

3.1. Metodologías usadas para el desarrollo de videojuegos

Dentro de la industria de los videojuegos es común utilizar metodologías aplicadas en otras áreas del desarrollo de software; es por esto que es necesario mencionar algunas de estas para determinar qué elementos de las mismas pueden ser de utilidad para la metodología que será planteada dentro de este trabajo.

3.1.1. Scrum

Esta metodología ágil se caracteriza por dividir las tareas en determinada cantidad de días de trabajo a las que se les llama "sprints". Los sprints pueden durar desde un día hasta cinco semanas dependiendo de la etapa en la que se encuentre el juego, esto permite a los desarrolladores tener las llamadas "Sprint Meetings", las cuales se desarrollarán entre el final de un Sprint y el comienzo del siguiente, esto con el objetivo de poder identificar lo que se logró, los problemas que surgieron y las tareas a realizar en el Sprint siguiente. [Gar]
Las principales ventajas de esta metodología es que permite a los desarrolladores ir cambiando algunos aspectos a lo largo del desarrollo de acuerdo a los problemas que pueden ir surgiendo a lo largo de los distintos sprints, y que, probablemente no podrían haber sido identificados a la hora de comenzar con la

planeación. También puede ser de mucha utilidad al aplicar la técnica de "Divide y vencerás" para distribuir el trabajo en tareas más pequeñas que además de facilitar el trabajo hace que el producto final parezca algo menos difícil de alcanzar.

3.1.2. XP

La metodología XP (o programación extrema) plantea cuatro variables para los proyectos: el tiempo, el alcance, el costo y la calidad; de estas cuatro variables, el equipo de desarrollo deberá ser capaz de determinar los valores para una de estas variables y el resto deberá ser planteada por los factores externos (el jefe del proyecto o los clientes). Por ejemplo, si los clientes plantean el alcance y el tiempo de desarrollo del juego y el jefe del proyecto plantea la calidad, el equipo de desarrollo podrá plantear el costo por la realización del proyecto.
Al igual que en la metodología Scrum, la metodología XP divide el trabajo en tareas más pequeñas y requiere de hacer juntas constantes para poder tener una buena comunicación entre los integrantes del equipo, esto permite que todos los miembros del equipo conozcan el estado actual del proyecto.
La metodología XP también suele dividir el proyecto en cuatro fases:

1. Planificación: En esta etapa el equipo identifica las características con las que debe de contar el juego, además de que define la prioridad de cada una de estas.

2. Diseño: El equipo comienza con la preproducción del juego, en esta etapa se definen cómo deberán quedar varios aspectos del mismo para que los programadores y artistas tengan claros los elementos que deberán crear.

3. Codificación: Se comienza el proceso de escribir el código para el juego, esto se hará en parejas de trabajadores frente a una misma computadora, esto permitirá que a la vez que uno de los programadores escribe código el otro va revisando la sintaxis para poder evitar errores y que ambos conozcan el comportamiento del código por si es necesario realizar cambios para corregir problemas u optimizar los procesos.

4. Pruebas: Al tener una versión jugable del producto se comenzará a hacer pruebas para poder identificar si los resultados que se están obteniendo son satisfactorios, encontrar errores que no fueron identificados al momento de codificar o sugerir cambios para mejorar la jugabilidad.

3.1.3. Espiral

Esta metodología le permite al equipo dividir el proyecto en varios "espirales", estos serán un conjunto de ciclos que se irán repitiendo en varias ocasiones y que estarán divididos en las siguientes etapas:

1. Determinar los objetivos: En esta etapa el equipo tendrá que identificar los objetivos que se esperará conseguir en el espiral correspondiente, las maneras en que podrían ser conseguidos los objetivos y se establecerá un cronograma de actividades.

2. Análisis de las alternativas: Durante esta etapa, el equipo se reunirá para ver la viabilidad de cada una de las propuestas, se seleccionará una de ellas y se identificarán los posibles riesgos de seguir esta propuesta. Una vez identificados los posibles riesgos, se realizará un prototipo con el que se tratará de ver los efectos de los riesgo y las maneras en que pueden ser resueltos.

3. Desarrollo: Se trata de cumplir los objetivos decididos al comienzo del espiral haciendo uso del prototipo creado, durante esta etapa se estarán haciendo pruebas para verificar el correcto funcionamiento.

4. Evaluación: Se le muestra al cliente los resultados obtenidos en esta espiral para recibir una retroalimentación y saber si el proyecto va por buen camino.

Una vez terminada la cuarta etapa del espiral, se reiniciará el proceso para poder comenzar un nuevo espiral.
Esta metodología brinda algunas ventajas como el hecho de que el equipo de desarrollo evalúa constantemente los riesgos a los que puede tener que enfrentarse, con la experiencia que va obteniendo el equipo puede realizar espirales

cada vez más rápido y al estar en una comunicación constante con el cliente pueden ir recibiendo retroalimentaciones para tener un resultado que le guste a ambas partes.

3.1.4. Cascada

Esta metodología le propone al equipo de desarrollo dividir el trabajo en distintas etapas enumeradas para realizarlas en ese orden. Al terminar cada una de las etapas de desarrollo el equipo deberá reunirse para identificar si los resultados obtenidos fueron satisfactorios, en caso de que se identifiquen errores o algún detalle a cambiar, el equipo deberá hacer los cambios necesarios antes de avanzar a la siguiente etapa del proyecto. [Mon]
La metodología cascada divide el proyecto en las siguientes etapas:

- Requisitos: El equipo se reúne para poder identificar los elementos con lo que deberá de contar el juego y se documentarán para tener un registro de lo que se buscará obtener.
  En esta fase también se identificarán los costos que tendrá la realización del juego y se identificará quién realizará ciertas acciones.

- Diseño: En esta etapa se empieza a trabajar para crear la arquitectura de software, interfaces y librerías que serán necesarios para el desarrollo del juego.

- Implementación: Se utiliza los resultados obtenidos en la fase anterior para poder crear el juego. El resultado de esta fase será una versión jugable que será probada en la etapa posterior.

- Prueba: El juego será probado en expertos para poder detectar si los resultados obtenidos dan una buena experiencia de juego y se buscarán errores que no hayan sido detectados por los desarrolladores.

- Entrega y mantenimiento: Parte final del desarrollo del juego, en esta etapa se entrega el juego para poder ser distribuido a los usuarios. Una vez empezada la venta, el equipo deberá seguir trabajando en el mantenimiento del juego creando actualizaciones que corrijan los posibles errores que puedan ir apareciendo a lo largo del tiempo.

Es importante identificar cuándo es mejor usar la metodología cascada en vez de una metodología ágil.La metodología cascada se caracteriza por ser muy rígida y que al usarla es difícil realizar cambios una vez avanzado el proyecto, es por esto que, la metodología cascada es recomendable para aquellos juegos en los que una vez empezado el desarrollo no se vayan a realizar grandes cambios. También es importante mencionar que en con esta metodología los proyectos suelen requerir de un costo menor y que, en caso de que algún miembro del equipo de desarrollo se vaya del proyecto, no será tan complicado para el nuevo integrante entender el proyecto gracias a que todo ya está bien documentado.

3.2. Géneros de videojuegos

Al igual que en otros artes como pueden ser la música o el cine, en los videojuegos se han creado una gran variedad de géneros y subgéneros para poder cumplir con los gustos de la mayoría de los jugadores posibles, así como dejar volar la imaginación de los desarrolladores y hacer realidad los juegos que tienen en mente. Es vital para el desarrollo de un juego tener claro el género objetivo, esto permitirá tener una mejor idea de las mecánicas que se requerirán, la historia que se buscará contar, aspectos artísticos, entre otros elementos del juego.
Algunos de los ejemplos de géneros más comunes que se pueden encontrar dentro de la industria son:

3.2.1. Shooters

En los juegos shooter, el jugador hace uso de diversas armas de fuego para derrotar a los enemigos con los que se encuentre durante el camino, a la vez que, intenta no recibir daño por parte de los mismos. Dentro de estos juegos es común brindarle al jugador maneras de reponer munición y recuperar salud para que así logre llegar al final de los niveles. Debe haber una buena velocidad de movimiento de los personajes y los enemigos, posiciones en las que el jugador pueda cubrirse de los disparos y (en la mayoría de los casos) una forma de combatir cuerpo a cuerpo

Desde principios del siglo XXI, con la llegada de juegos como Halo, la mayoría de estos juegos cuenta con opciones de juego multijugador en línea para que los jugadores puedan enfrentar a sus amigos u otros jugadores desconocidos sin importar la distancia entre ellos.
Algunos de los subgéneros que se pueden encontrar dentro de este género son:

- **FPS:** Son las siglas para First Person Shooter y estos se caracterizan por colocar la cámara de forma que el jugador pueda ver lo que estaría viendo el personaje que está manejando. Algunos de los juegos más famosos de este subgénero son Halo, Doom, la saga Call of Duty, Wolfenstein o Killzone.

- **TPS:** Estas siglas significan Third Person Shooter y, a diferencia del género anterior, estos colocan la cámara alejada de la espada del personaje que se está controlando. Algunos ejemplos de estos juegos son Gears of War, The Division, Mafia, GTA, Uncharted o Mass Effect.

- **Shooter sobre rieles:** En este subgénero el jugador no tiene la capacidad de mover al personaje debido a que este se moverá en determinados momentos y en la dirección que ya está fijada por los desarrolladores, por lo que, el jugador se encargará de dispararle a los enemigos lo más rápido posible para evitar recibir daño y terminar el nivel con la mayor puntuación posible. Ejemplos de estos juegos son House of the Dead, Resident Evil Umbrella Chronicles y Darkside Chronicles.

- **Shoot 'em up:** Este subgénero se caracteriza por enfrentar al o a los jugadores contra un gran número de enemigos haciendo uso de armas de fuego. Algunos de los juegos más conocidos de este género son Metal Slug, Contra y Hotline Miami.

Dentro de las sensaciones que se buscan generar dentro del jugador hay una gran variedad dentro de los shooters, ya que no todos buscan las mismas sensaciones todo el tiempo, por ejemplo, las versiones actuales de Doom realizadas por ID Software se caracterizan por dar la sensación de acción desenfrenada junto con la de un alto nivel de movilidad, por otro lado, en juegos como Gears of War, contamos con un juego que contiene un alto grado de acción pero al mismo

tiempo le brinda al jugador la sensación de estar atrapado en una cobertura buscando la estrategia más óptima para salir de esa situación.

3.2.2. Role Playing Game (RPG)

Los RPG se caracterizan por brindarle al jugador una aventura con un mundo por explorar a la par de que siguen una historia principal, en estos juegos es común encontrar misiones secundarias que den a conocer más sobre el mundo en el que se encuentra, la historia de los personajes o simplemente ayuden a conseguir recursos o experiencia. También es común que los personajes no tengan una personalidad tan definida para permitir que el jugador se sienta identificado con el protagonista, permiten mejorar las estadísticas de los personajes por medio de subidas de nivel, habilidades u objetos mejores y, en algunos juegos, permiten cambiar el rumbo de la historia por medio de las decisiones u acciones que haya realizado el jugador a lo largo del juego. Algunos de los subgéneros dentro del RPG son:

- **MMORPG:** Son las siglas para Massive Multiplayer Online Role Playing Game y se caractetizan por permitirle a varios jugadores vivir una aventura en un mismo mundo por medio de la conexión online, las acciones de los jugadores puede perjudicar o ayudar al resto de ellos ya sea al atacarlos o ayudarse mutuamente durante misiones. Algunos juegos icónicos dentro de este subgénero son World of Warcraft, Fallout 76 o The Elder Scrolls Online.

- **RPG tácticos:** El objetivo en este subgéneros es mover y realizar acciones con un conjunto de unidades dentro de su correspondiente turno para poder cumplir con los objetivos de la misión, ya sea, tomar una zona, derrotar unidades o defender una ubicación . Dentro de este subgénero podemos encontrar juegos como Valkyria Chronicles, Fire Emblem y XCOM.

- **ARPG:** Son las siglas de Action Role Playing Game y, a diferencia del subgénero anteriormente mencionado, una de sus características es el combate en tiempo real. En este género se encuentran juegos como Fallout, The Elder Scrolls: Skyrim y Scarlet Nexus.

3.2.3. Pelea

En estos juegos es común que dos personajes (ya sea controlados ambos o sólo uno por un jugador) se enfrenten en batallas donde el objetivo es bajar a cero la vida del rival, sacarlo de una determinada área o dejarlo inconsciente el tiempo suficiente. Para conseguir este objetivo, los jugadores deberán hacer uso de un conjunto de movimientos y combos que disminuirán la vida del rival al ser conectados. Estos juegos se caracterizan por tener medidores de salud de cada uno de los personajes, distintos personajes con formas de pelea distintas entre sí y movimientos especiales y finales que pueden ser activados bajo determinadas circunstancias. Algunos ejemplos de juegos de pelea son la saga de Street Fighter, Mortal Kombat, The King of Fighter, Soul Calibur, Killer Instinct y Smash Bros.

3.2.4. Deportivos

Los juegos deportivos tratan de simular actividades deportivas que se realizan en la vida cotidiana o toman elementos de los mismos para crear sus propias versiones, es por esto que, dependiendo del deporte en el que se esté basando el juego, las características van cambiando. Dentro de los juegos que tratan de emular deportes reales podemos encontrar la saga FIFA y PES dentro del fútbol soccer, Madden NFL en el fútbol americano, MLB The Show en el béisbol, entre otros.
Dentro de los juegos deportivos que no tratan de representar como tal el deporte en que están basados, sino que, crear su propia versión del mismo, podemos encontrar juegos como Mario Strikers, Rocket League, Mario Kart, entre otros.

3.2.5. Plataformas

Género caracterizado por hacer uso de diversos movimientos y habilidades con los que cuenta el protagonista para así poder derrotar enemigos y avanzar por el mundo esquivando todos los obstáculos que estén en el camino. Este género se popularizó en gran medida con la llegada de Super Mario Bros y de ahí fueron apareciendo otros juegos emblemáticos como es el caso de Sonic, Crash Bandicoot, Banjoo-Kazooie, Donkey Kong, entre otros.

3.2.6. Aventura

En estos juegos el jugador tendrá un mundo por explorar a la vez que sigue una historia principal y que, a menudo, viene acompañada de misiones secundarias y coleccionables que buscar a lo largo de este mundo. Dentro de este género aparecen juegos como la saga Tomb Raider, Uncharted, The Witcher, Assasin's Creed, entre otros.

3.2.7. Terror

En este género se caracteriza por buscar causarle tensión y sacarle algunos sustos a los jugadores haciendo uso de entornos tenebrosos, criaturas extrañas, fantasmas, silencios y otros elementos que puedan causar estas sensaciones. Algunos de los juegos más conocidos dentro de este género son Amnesia, Outlast, Alien Isolation, entre otros.
Dentro de este género podemos derivar el género del Survival Horror, en el que, el jugador deberá administrar correctamente un conjunto de recursos como la munición o medicamentos para poder derrotar a las criaturas que puedan ser encontradas. Claros ejemplos de este género son las sagas de Resident Evil, Silent Hill, The Evil Within, Dead Space o Metro.

3.3. Tipos de jugadores

Otro factor fundamental a la hora de diseñar las mecánicas de nuestro juego es identificar a qué tipo de jugadores estamos tratando de llegar con el mismo. Dentro del mundo de los videojuegos se han realizado varias clasificaciones para los jugadores, estas están basadas en distintos aspectos como puede ser la habilidad que tienen dentro de los juegos, la cantidad de dinero que están dispuestos a pagar dentro de ellos, la cantidad de horas que dedican a jugar, las plataformas en las que juegan, el tipo de contenido que buscan dentro de los juegos, entre otros.
Una de las formas más conocidas de clasificar a los jugadores es la taxonomía de Bartle (llamada así por su creador, Richard Bartle); esta clasificación coloca en una plano de dos ejes cuatro tipo de jugadores, los triunfadores, los explorados,

los socializadores y los asesinos. Esta clasificación es la más adecuada dentro de esta metodología dado que nos ayuda a clasificar en base a lo que los jugadores buscan dentro de los juegos y sus mecánicas.

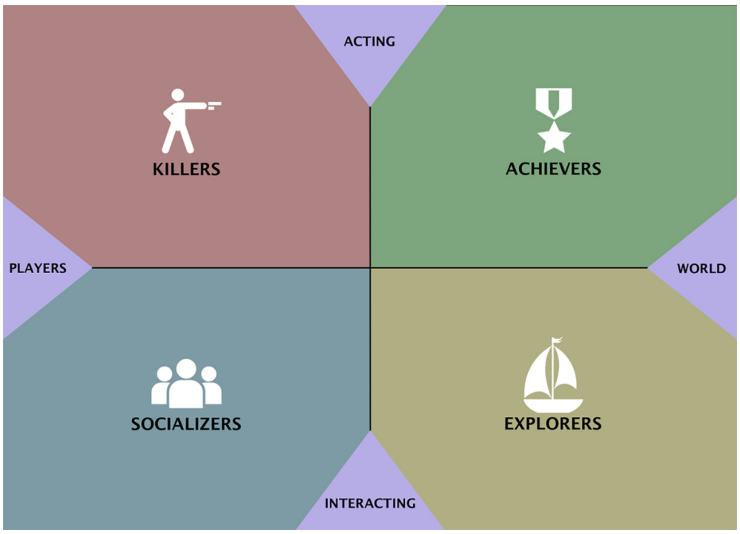


Figura 3.1: Taxonomia de Bartle

Como se puede apreciar en la figura 3.1, cada tipo de jugador se divide por ciertas características y objetivos dentro del juego, las cuales son:

- Triunfadores: Este tipo de jugador se caracteriza por completar lo más posible del mundo en el que se encuentran, buscan conseguir todos los logros o trofeos (dependiendo de la plataforma en la que jueguen), obtener todos los coleccionables, equipo o mejoras de los personajes, a la vez de que tratan de ser los mejores dentro de los tableros de puntuación o clasificación.

- Exploradores: Estos jugadores buscan interactuar con el mundo para así poder explorar cada rincón de este, encontrar todos los *easter eggs* [1] escondidos por los desarrolladores e interactuar con todos los NPC que sean posibles.

[1] Secretos o curisidades que requieren determinadas acciones para ser encontrados

- Socializadores: Los jugadores socializadores disfrutan de interactuar y jugar con otras personas, ya sea amigos o gente desconocida dentro del juego. Para este tipo de jugadores, los demás jugadores son compañeros con los cuales conseguir un objetivo y vivir aventuras.
- Asesinos: Este tipo de jugadores buscan usar a los demás jugadores para poder obtener sus objetivos, esto generalmente sucede al matarlos dentro del juego o robando las pertenencias de los demás jugadores. Este tipo de jugadores es de los que más plantean un reto a la hora de diseñar un juego, esto debido a que muchas veces son necesarios para la experiencia de los demás jugadores, pero, si hay un exceso de los mismos, pueden ahuyentar a los jugadores que se cansan de ser atacados.

Además de esta clasificación de jugadores, hay otros dos elementos que deben ser tomados en cuenta, que tan accesible será el juego. Además de la dificultad (elemento que se tomará más a detalle en la sección siguiente) se debe de pensar si buscamos que nuestro juego sea capaz de ser jugado fácilmente por personas que no tienen mucha experiencia jugando videojuegos o si buscaremos dar una jugabilidad orienta al público más exigente.

3.4. Curva de dificultad

La curva de dificultad es una gráfica que representa la relación entre el aprendizaje del jugador y los desafíos a los que deberá enfrentarse. Si el jugador lleva poco tiempo de juego y se le presenta un desafío demasiado complicado considerará que el juego es demasiado injusto y lo dejará con una sensación de frustración, sin embargo, si tiene una habilidad demasiado alta a comparación de los desafíos que se le presentan, este considerará que el juego es demasiado fácil y aburrido.

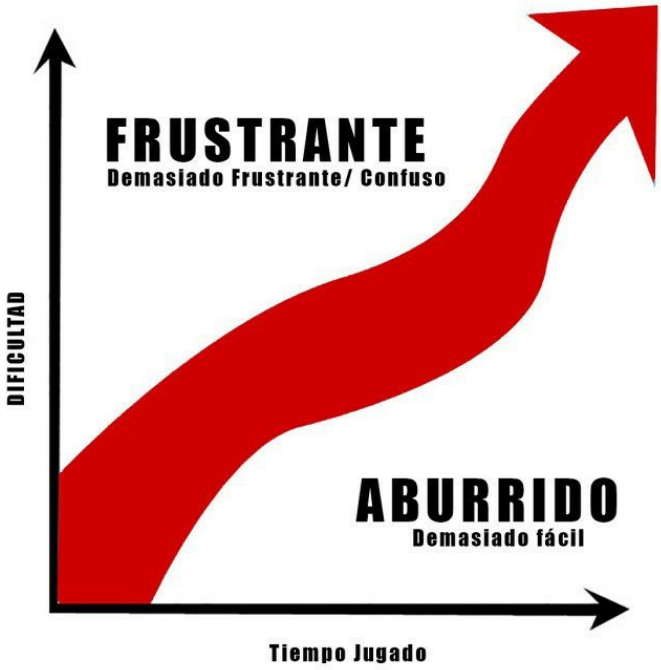


Figura 3.2: Curva de dificultad

Es por esto que, a la hora de desarrollar las mecánicas se debe tomar en cuenta que deben poder ser aplicadas para presentar desafíos sencillos que con el paso del tiempo vayan escalando en dificultad para poder darle una buena experiencia al jugador durante todo el tiempo en que este se encuentre jugando. Mantener la atención del jugador no es una tarea sencilla, cada jugador tiene un nivel de habilidad distinto al del resto y, por lo tanto, lo que para algunos puede ser un reto casi imposible para otros puede ser un desafío nivelado. En algunos juegos este problema puede ser solventado por medio de enemigos de distintos niveles repartidos por distintas zonas, en caso de que el jugador no pueda eliminar a esos enemigos puede ir a enfrentar a algún otro para ir mejorando sus habilidades y posteriormente enfrentar a los que no podía derrotar antes. Otra forma más común es aplicar distintos niveles de dificultad para que el jugador seleccione el que considere adecuado en base a la experiencia con la que cuenta, aunque, esto muchas veces solo logra generar una dificultad artificial en la que

los enemigos tienen una cantidad de salud demasiado alta o demasiado baja o causen una cantidad de daño demasiado alto o demasiado bajo.
Estas dos son de las formas más comunes de solventar este problema pero, hay que aclarar que no hay una forma definitiva para plantear la dificultad dentro de un juego, debe ser algo que se que se defina por medio del sistema que se considere más adecuado para el mismo en base a su género, las mecánicas y varios testeos que permitan encontrar esos puntos exactos con los que la mayoría de los jugadores tenga una buena experiencia.

3.5. Recursos y entidades

La diferencia entre los recursos y las entidades puede llegar a ser un poco confusa, pero, puede explicarse pensando en que los recursos son el concepto del objeto y las entidades son el objeto físico o los estados en los que se puede encontrar.
Por ejemplo, si en nuestros juegos contamos con flechas, el recurso es la flecha y las entidades son cada flecha con las que nuestro personaje o algún enemigo cuenta (sin importar la cantidad). Dentro de la entidades de las flechas, también debemos contar con los estados en las que estas se encuentran, ya sea que el jugador la está recargando, la está apuntando o ya se encuentra disparada; por otro lado, dentro de nuestros recursos de flechas, ¿ahora contamos con una flecha más? ¿una flecha menos? ¿Esos cambios en los estados alteraron la cantidad de nuestros recursos?.
Es fundamental que al diseñar un recurso se tome en cuenta todos estos aspectos que determinarán las funciones que tendrán los recursos, la forma en que estos podrán ser obtenidos, cómo se podrán gastar o perder y la interacción que pueda tener con otros recursos a la hora de juntarlos para crear otros o inclusive intercambiarlos.

3.6. Mecánicas

Como ya se ha mencionado, las mecánicas son las reglas del juego llevadas a su máximo nivel de detalle, estas regirán las acciones que el jugador podrá

realizar a lo largo del juego, la economía interna del mismo, las acciones que podrán ser realizadas por los NPC y la forma en que se comportará el mundo.
Las mecánicas pueden comenzar a ser concebidas como ideas poco detallas para posteriormente ser detalladas, por ejemplo, podemos pensar en un personaje que combate haciendo uso de una espada, para poder crear una mecánica sería necesario definir varios de los aspectos necesarios para el combate, como es el caso de los tipos de ataque que tendrá el personaje, el alcance de la espada, el daño que haría con cada ataque, las posibles combinaciones de ataque que podrán realizarse, entre muchos otros aspectos. Dentro de las mecánicas podemos encontrar muchas clasificaciones dependiendo de la importancia que tengan dentro del juego y las funciones que estas estén cumpliendo. Es importante conocer estas clasificaciones para así saber lo que estamos diseñando, el énfasis que se le debe dar a cada una, si las mecánicas que tenemos en mente realmente aportan algo al *gameplay* y valdrá la pena invertir tiempo y recursos para su desarrollo. [Fab] [EA12] [Fra]

3.7. Mecánicas centrales

Las mecánicas centrales son aquellas que regirán gran parte de lo que podrá realizar nuestro jugador y las forma en que se comportará el mundo que lo rodea. Es por esto que, es vital para el desarrollo del juego definir correctamente las funciones de estas mecánicas en base al tipo de juego que se esté desarrollando (por lo que es fundamental tener claramente identificado el género de videojuegos objetivo) y pensando en el desarrollo que se le dará a estas para mantener el interés del jugador.
Para conseguir mantener el interés del jugador se deberá crear las mecánicas centrales pensando en cómo pueden trabajar en conjunto con otras mecánicas para lograr plantear desafíos iniciales que le permitan al jugador aprender lo básico sobre estas y poco a poco ir aumentando el nivel de dificultad, es decir, tomando en cuenta la curva de dificultad.

3.8. Mecánicas secundarias

Las mecánicas secundarias son aquellas que tienen un papel menos importantes que las centrales pero que están muy ligadas a estas, las mecánicas secundarias también están presentes en menos ocasiones durante el juego. Es importante recordar que no se debe saturar el *gameplay* con demasiadas mecánicas que puedan llegar a ser innecesarias, por lo que, se deben pensar en estas como un complemento que enriquezca y complemente a las mecánicas centrales.
Una de las formas en que las mecánicas secundarias pueden cumplir con este objetivo es al aplicar mejoras en un arma, obstáculos que cumplan con ciertos patrones de movimiento o algún mecanismo que permita abrir puertas; esto sin caer en la creación de *gameplay periférico*, del cual hablaremos en el siguiente tema.

3.9. Gameplay Periférico

El *gameplay* periférico son aquellas secciones dentro de nuestro juego que requieren de un diseño y programación de mecánicas que solamente serán utilizadas en un pequeño pedazo del juego. Este debe evitarse a menos de que estas secciones puedan enriquecer en gran medida la experiencia del usuario o al menos que un conjunto de estas mecánicas sean parte de minijuegos característicos del género que en el que se está trabajando.
Un ejemplo de *gameplay* periférico puede ser cuando se está desarrollando un juego de disparos en tercera persona y de la nada se agrega una sección en la que los personajes juegan fútbol, esto puede ser considerado gracioso por el equipo de desarrollo y quizá a muchos de los jugadores les pueda gustar, pero, se debe tomar en cuenta que estas mecánicas deben de ser pulidas y requerirán de una gran cantidad de tiempo y recursos (los cuales podrían ser aprovechados para pulir elementos del juego que si se requieran en la mayor parte del mismo).
Algunas de las veces en que este tipo de *gameplay* puede ser aplicado es en sagas como Yakuza en las que los minijuegos como el karaoke, arcades o centros de bateo son una características dentro de la saga aunque no son parte del género principal del juego (Acción/ Beat'em up). Un punto a tomar en cuenta dentro

de este ejemplo es el hecho de que, aunque son minijuegos muy pulidos, son en su mayoría "simples" y no son algo que tome una cantidad de tiempo de desarrollo que pueda afectar en gran medida al juego principal.

3.10. Tipos de mecánicas

Todas las mecánicas a diseñar para el juego deben cumplir con alguna función para enriquecer la jugabilidad dentro del mismo, por lo que, será necesario clasificar las mismas en base a sus funciones para poder llevar un mejor control de la relación que habrá entre las mismas y los objetivos que deberán cumplir.

3.10.1. Acciones del jugador

En todo videojuego debe haber una interacción entre el programa y los jugadores, es por esto que, diseñar y programar las posibles acciones del jugador dentro del juego es de las tareas más importantes y vitales dentro del juego.
Dentro de estas mecánicas podemos definir el tipo de movimiento que tendrá el jugador, los posibles ataques que pueda realizar, las interacciones con el entorno, la forma en que saltará, la forma en que podrá mover la cámara, entre muchos otros.

3.10.2. Economía interna

Todo juego conlleva una economía interna que va mucho más allá de lo que la mayoría piensa, ya que, no solo abarca la moneda que puede existir dentro del mismo para comprar objetos. La economía interna incluye la munición que el jugador puede usar para disparar un arma, las vidas o la salud con la que cuenta, los objetos que pueden ser usados, las tropas que puede manejar e inclusive el tiempo puede ser parte de la economía interna dependiendo del juego en cuestión. [EA12]
En este tipo de mecánicas el prototipado (que será explicado a profundidad dentro de la metodología) es indispensable a la hora del diseño para poder obtener los valores más óptimos dentro de las posibles combinaciones.
Por ejemplo, si tenemos un juego donde el oro es uno de los elementos más va-

liosos y podemos construir en determinado momento una edificación que ayude a generar más del mismo, si, el índice de producción de esta edificación es demasiado alto, el oro perderá su valor dentro del juego en el momento en el que el jugador tenga suficiente de este para generar todas las demás edificaciones/tropas/mejoras y aún así tenga oro de sobra. En ese momento, el oro pasaría a ser algo inútil en el segundo plano, pero, ¿qué pasaría si el índice de producción es demasiado pequeño?. En caso de que el jugador pueda generar u obtener oro de una manera demasiado lenta a comparación de la cantidad que necesita sentiría que no está progresando y podría frustrase hasta el punto en que deje el juego debido al mal diseño de la economía del mismo.
Dentro de las mecánicas que gestionan la economía interna del juego y que debemos nivelar se encuentran:

- Productoras: Se encargan de generar más elementos como dinero, recursos, enemigos, aliados, etc. Estos aparecerán "de la nada" sin necesidad de que el jugador dé algo a cambio. Un ejemplo podría ser una caja de munición que aparece cada cierto tiempo durante una batalla con un jefe con el objetivo de que el jugador no se quede sin munición y, por ende, pierda la batalla.

- Consumidoras: Las consumidoras serán los encargados de disminuir cierta cantidad de recursos en base a una acción ocurrida o el paso del tiempo. Un ejemplo podría ser un juego en el que el jugador debe administrar una franquicia de tiendas, por lo que, cada día deberá pagar el salario de sus empleados, renta, electricidad, etc. La mecánica consumidora sería esta disminución del dinero con el que cuenta el jugador con el paso de cada día dependiendo de los empleados y sucursales.

- Convertidoras: Estas mecánicas se han vuelto famosas por los juegos en los que se debe hacer un "crafteo" [2], consisten en convertir uno o varios recursos en algún otro elemento con una funcionalidad distinta, el jugador pierde los recursos pero recibe algún objeto o construcción a cambio del mismo (el objeto recibido no existía previamente). Podemos encontrar

[2] Crear un objeto en base a determinados materiales

ejemplos de estas mecánicas en el famoso videojuego *Minecraft*, uno de estos ejemplos es la clásica conversión de cuatro bloques de madera en una mesa de trabajo con la que el jugador después puede construir objetos más sofisticados.

- Negociadoras: De forma a similar que en las mecánicas convertidoras, las negociadoras consisten en que el jugador dé una cantidad de determinado recursos a cambio de recibir algún otro objeto u recurso, la diferencia radica en que en las convertidoras se crea un objeto en base a otro y en las negociadoras se entrega un objeto existente del que muchas veces existe una cantidad limitada por parte del "vendedor". Un ejemplo de estas mecánicas puede encontrarse en el juego *Cyberpunk 2077* en el que el jugador puede comprar y vender objetos mientras que los vendedores tengan inventario del objeto a comprar o dinero para comprarle objetos al jugador.

3.10.3. Progresión

Las mecánicas de progresión son aquellas que definen la forma en que el jugador podrá avanzar a lo largo del juego, una de las formas en que se pueden determinar el avance es por medio del "juego en tiempo real" y por turnos.
En la progresión por turnos el jugador tendrá pausas dentro las acciones que realizará y el feedback recibido para, posteriormente, volver a actuar dentro del juego, esto permite darle al juego una importancia mayor a la estrategia o darle más protagonismo a la historia. El combate por turnos es una de las formas de progresión por turnos (dada la redundancia) más conocida, en esta, el jugador tiene una determinada cantidad de movimientos en los que deberá ordenarle a sus personajes lo que harán (ya sea atacar, defenderse, usar un objeto, cambiar de posición, etc.) para tratar de derrotar a los enemigos o resistir los futuros ataques de los mismos; una vez termina el turno del jugador, se le pasará el turno al jugador siguiente o a la IA que se encarga de controlar a los personajes enemigos, esto con el objetivo de hacer los movimientos necesarios para contrarrestar las medidas tomadas por el jugador que tuvo el turno anterior. Una vez que termina esta fase de respuesta, se pasará nuevamente el turno al

jugador que comenzó, lo que hace que este tipo de combate tenga un énfasis mucho mayor en tener una buena estrategia y saber administrar correctamente tanto los personajes que utilizaremos como los objetos y los sets de movimiento con los que contará cada uno. Esta forma de jugar es muy común dentro de JRPG's como el caso de Pokémon, Fire Emblem, los primeros Final Fantasy, Dragon Quest o Persona.

Otro de los géneros donde es muy común encontrar la progresión por turnos puede ser dentro de juegos de toma de decisiones, dentro de estos juegos al jugador se le presentará una historia y en base a los hechos sucedidos deberá seleccionar entre una serie de opciones que determinarán las acciones a tomar por los personajes que controle, estas elecciones afectarán a los hechos subsecuentes y podrán cambiar en gran medida el rumbo que tomará la historia. Aunque en estos juegos no se encuentre una clara diferencia entre los turnos, esta basado en un sistema en que el jugador tendrá que esperar para recibir el feedback de la historia y posteriormente tendrá su turno para elegir, lo que igualmente sucederá de forma cíclica hasta el final del juego. Algunos ejemplos de juegos donde se apliquen estas mecánicas son The Walkig Dead, The Wolf Among Us, Until Dawn o Life is Strange.

Dentro de la progresión en tiempo real podemos encontrar una variedad de géneros mucho mayor que la se encuentra dentro de la progresión por turnos, en este tipo de progresión hay una movilidad constante cuando los jugadores se enfrentan a los enemigos, exploran el mundo o resuelven acertijos, lo que puede generar una mayor sensación de acción o liberad, sin dejar a un lado que también tiene su componente estratégico (aunque con menos tiempo para pensar en los movimientos que se harán). Entre algunos de los juegos que cuentan con este tipo de progresión podemos encontrar Devil May Cry, Halo, The last of us, Bayonetta, entre otros.

Otro tipo de mecánicas relacionadas con las mecánicas de progresión son aquellas que determinan obstáculos que evitan el avance del jugador, más no son parte del objetivo final dentro de la zona. Algunos ejemplos pueden ser trampas u obstáculos que requieran de realizar determinadas acciones para poder avanzar o enemigos que deban ser derrotados para proseguir con el camino.

3.10.4. Condiciones de victoria y derrota

Las mecánicas encargadas de llevar las condiciones de victoria son aquellas que le dirán al jugador lo que debe realizar para poder completar el juego (a largo plazo), completar cierta parte de la historia (mediano plazo) o lo que debe hacer para poder avanzar dentro de una determinada zona (corto plazo). Dentro de los objetivos a corto plazo, es recomendable no decirle exactamente el jugador lo que se debe de hacer todo el tiempo, esto para evitar la sensación de que "se le está llevando de la mano"; en su lugar, es recomendable dejar pistas claras que le permitan al jugador deducir cómo poder seguir avanzando. Para conseguir este objetivo, se debe combinar las habilidades de diseño de mecánicas junto con las diseño de niveles, las mecánicas le brindarán al jugador las herramientas necesarias para poder avanzar dentro de la historia y el diseño del nivel dejará las pistas y los desafíos pertinentes.

Un ejemplo de mecánicas de victoria pueden ser un rompecabezas en el que se deba descubrir cómo alcanzar una zona alta, una puerta bloqueada que requiere de una llave o una tarjeta de acceso, una cantidad de enemigos que deberán ser derrotados, escapar de una determinada zona antes de que termine el tiempo, defender una ubicación, entre otros.

Las condiciones de derrota serán las que determinarán cuando el jugador ha fallado en su misión y le aumentarán la dificultad al juego, algunas de las condiciones de derrota más comunes son el perder la cantidad de vida con la que cuenta el personaje, que se termine la cantidad de tiempo con la que se dispone, que se termine una determinada cantidad de recursos, perder una determinada cantidad de tropas, entre otros.

Es importante mencionar que no es necesario contar todo el tiempo con alguna condición de derrota, pueden existir zonas del juego donde el jugador pueda explorar, hablar con otros personajes, resolver un puzzle sin tener presión, realizar alguna actividad secundaria, crear nuevos objetos o comerciar con algún otro personaje.

3.11. Machinations

Dentro de las herramientas de las que podremos apoyarnos para el desarrollo de nuestras mecánicas se encuentra Machinations [Mac16], una herramienta en línea que permite hacer prototipos de una forma similar a la de los diagramas de flujo. Estos diagramas permiten estudiar la forma en que nuestras mecánicas causarían cambios dentro de la economía interna del juego dependiendo de si hay entradas por parte del usuario, eventos aleatorios o si se cumplen determinadas condiciones. [EA12]
Antes de comenzar a usar Machinations, es necesario conocer la simbología básica que se utilizará a la hora de crear nuestro diagramas, entre los símbolos básico podemos encontrar los siguientes:

- **Pool:** Las pool se representan como un círculo y su función principal será la de contener recursos que serán representados por fichas o una cantidad numérica (en caso de ser demasiados). Las fichas o la cifra representan la cantidad con la que la pool cuenta de un recurso (en caso de que no haya nada la pool estaría vacía). Las pool pueden representar la cantidad con la que un elemento del juego o el jugador cuentan de determinado recurso.
- **Connection:** Las conexiones nos ayudarán a mostrar la dirección en la que fluirán los recursos junto con la cantidad que estaría fluyendo cada vez que estos sean movidos. El número que aparezca en la parte inferior de la flecha denotará la cantidad que será enviada y, en caso de tener la palabra "all", se enviarán todos los recursos posibles.
- **Source:** Las fuentes son representadas por un triángulo equilátero con su punta superior viendo para arriba y su función es la de generar recursos, ya sea porque se cumplió alguna condición o porque estos deban estarse generando todo el tiempo.
- **Drain:** Los drenadores tienen la función opuesta a las fuentes y se representan de forma opuesta (con un triángulo viendo hacia abajo). Estos se encargarán de retirar recursos ya sea porque estos se gasten con determinadas acciones o estén destinados a desaparecer constantemente.

- **Converter:** Los convertidores se representan con un triángulo viendo hacia el lado derecho a la par de que está cruzado por una línea vertical (este también puede ser creado al combinar una fuente con un drenador) y su función es la de retirar una determinada cantidad de recursos para así generar una determinada cantidad de algún otro.

Como podemos observar en la descripción del funcionamiento de estos símbolos básicos, estos cumplen con las mismas características que algunas de las diferentes tipos de mecánicas que fueron mencionados en la parte de la economía interna, por lo que, su uso hasta cierto punto puede ser bastante intuitivo si es que analizamos la clasificación de las mecánicas antes de comenzar a hacer el prototipado.

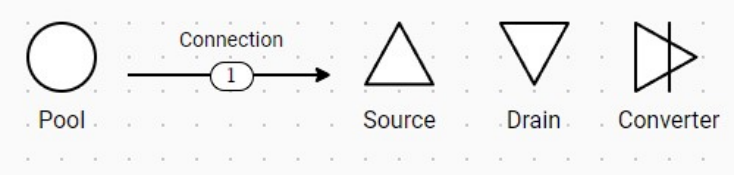


Figura 3.3: Simbología básica

Una vez que tenemos las herramientas básicas, es necesario crear condiciones para que estas se puedan activar en determinado momento, ya que, en un juego nunca ocurrirán los mismos hechos con la misma frecuencia; la mayoría de los cambios dentro del mundo van a ocurrir en respuesta a diversas situaciones que tendrán en mente los diseñadores y programadores a la hora de hacer las mecánicas.
Dentro de Machinations, estas condiciones son conocidas como Triggers y algunas de las formas en que estos pueden se creados son:

- **Passive:** Los trigger pasivos son aquellos que dicen que solamente se realizará una acción cuando otro trigger externo se lo pida (ya se porque algunos de los otros elementos dentro del diagrama hayan sido activados por su respectivo trigger). Los elementos que se encuentran pasivos no tienen ningún cambio en su forma, por lo que se ven igual que como fueron mostrados en la figura 3.3

- **Interactive:** Los elementos que se encuentran en modo Interactive tienen un símbolo de mouse sobre de ellos y se les adhiere un contorno más pequeño dentro de la figura, además de que, se les suma la capacidad de ser activados cuando el diseñador hace clic en ellos.
- **Automatic:** Los elementos a los que se le asigna el trigger Automatic estarán activados en todo momento y en cada paso estarán moviendo los recursos. Se puede identificar cuando está activado el trigger automático debido a la presencia de un asterisco sobre las figuras.
- **Enabling:** Los elementos que hacen uso de este trigger se activarán una vez cuando se de inicio a la simulación y contarán con una figura parecida a un rayo sobre de ellos.

Figura 3.4: Interactive Triggers

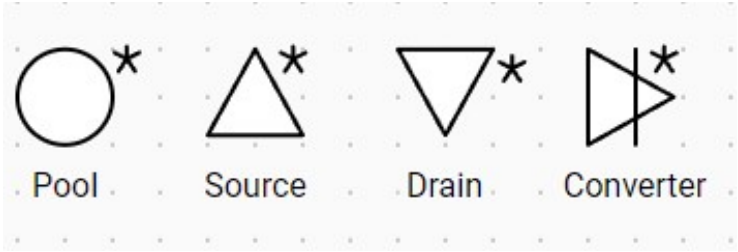


Figura 3.5: Automatic Triggers

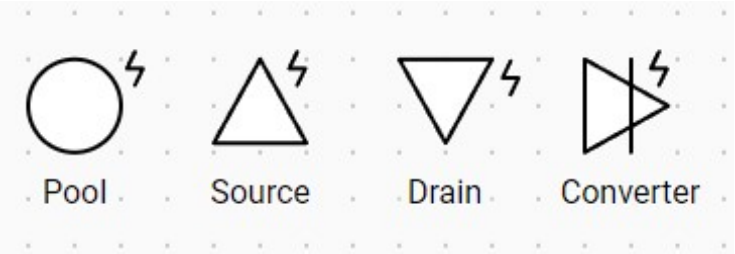


Figura 3.6: Enabling Triggers

Para poder comenzar a hacer uso de Machinations y hacer nuestros prototipos de las mecánicas que diseñamos es necesario ingresar a la página de Machinations, iniciar sesión, ir a la parte de “My Machinations” en tu perfil y seleccionar New. Esto nos generará plano mostrado en la figura 3.7, en el que será posible arrastrar desde el panel izquierdo los elementos que mencionamos anteriormente.

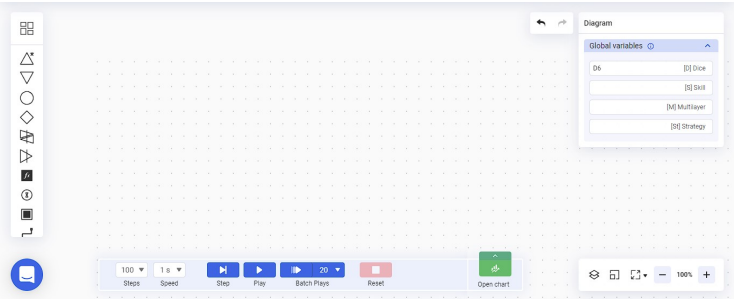


Figura 3.7: New Project

Al seleccionar alguna de las figuras, en la parte derecha de la pantalla aparecerá un menú como el de la Figura 3.8 con el que se podrán modificar distintos aspectos como la cantidad de recursos, labels, tipo de trigger y demás elementos dependiendo de las funciones de cada elemento.

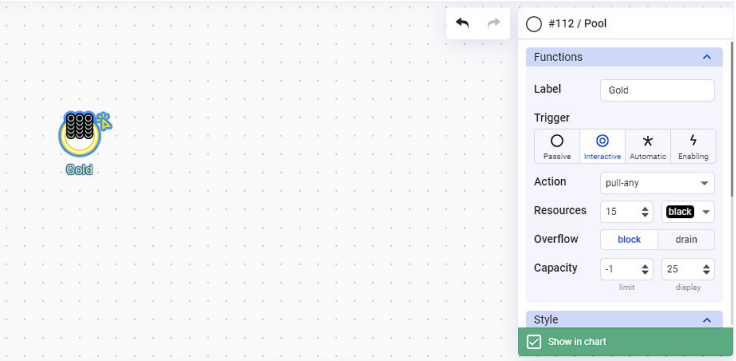


Figura 3.8: Configuration

Una vez que se terminé de crear el prototipo de nuestra mecánica, será necesario darle Play en la parte inferior de la pantalla para comenzar con la simulación.

3.12. Diagramas UML

Dentro de los tipos de diagramas que podemos crear haciendo uso de los diagramas UML se encuentran los diagramas de clases, estos permiten crear una representación visual del comportamiento de las clases que serán utilizadas dentro de un programa junto con la relación que habrá entre ellas.
Para poder hacer uso de este tipo de diagramas UML y facilitar la comprensión de los diagramas que serán presentados en la sección de patrones de diseño de software será necesario conocer la simbología presentada a continuación.

- Cuadro de clase: Contiene tres divisiones, la primera de ellas (contando de arriba para abajo) contendrá el nombre de la clase, la segunda los atributos de la clase y la tercera las funciones de la misma. (Figura 3.9)
- Atributos: Los atributos de la clase deberán contar con un símbolo que represente su visibilidad, el nombre del atributo, dos puntos (:) y el tipo de variable que corresponde al mismo. Los tipos de visibilidad serán representados con los siguientes símbolos. (Figura 3.9)
  - +: Representa que el atributo es público.

- -: Representa que el atributo es privado
- #: Representa que el atributo es protegido
- : Representa que es un package

- Funciones: Al igual que en los atributos estos deberán contar con su respectivo símbolo para representar la visibilidad, luego deberá ser agregado el nombre de la función con sus argumentos y su tipo dentro de un paréntesis y, finalmente, se agregarán dos puntos (:) junto con el tipo de dato de retorno. (Figura 3.9)

- Herencia: Se representa con una línea que termina con una flecha, la línea debe comenzar en la clase hija y la flecha debe apuntar a la clase padre. (Figura 3.10)

- Asociación: Se representa con una línea que conecta ambas clases. (Figura 3.11)

- Composición: Se representa con una línea que comienza en el componente y termina con un rombo coloreado en negro que apunta a la clase a la que pertenece el componente. Sobre la línea deberá anotarse un rango que denote la cantidad de instancias que compone a la clase y se denota de la forma siguiente.3.12

  - 0...1: De ninguna a una instancia
  - n: Cantidad específica de instancias
  - 0...*: De ninguna a muchas instancias
  - 1...*: De una a muchas instancias
  - n...m: Rango específico de instancias

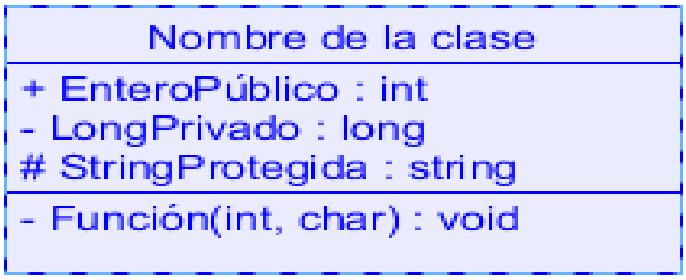


Figura 3.9: Cuadro de clases

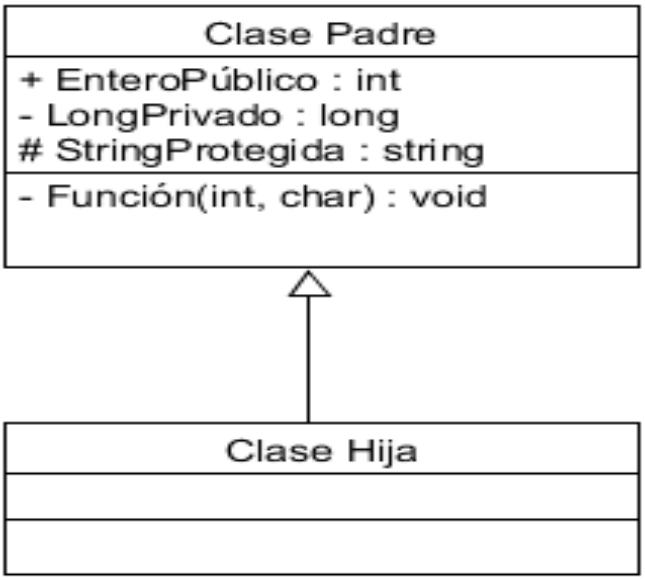


Figura 3.10: Herencia

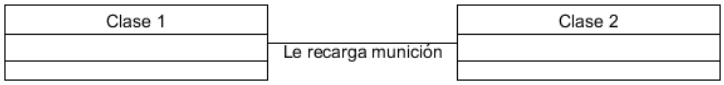


Figura 3.11: Asociación

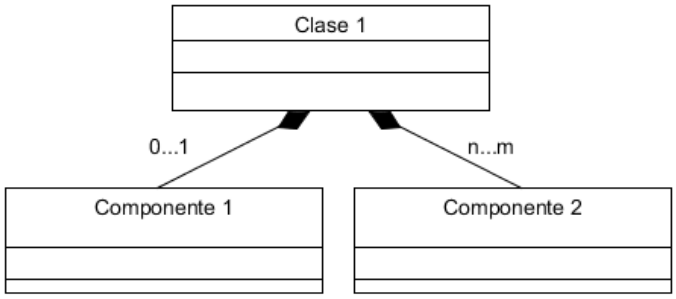


Figura 3.12: Composición

3.13. Patrones de diseño de software

Los patrones de diseño permiten a los programadores conocer posibles soluciones ya establecidas para problemas con los que comúnmente se encuentran al momento de programar. Estos patrones consisten en una descripción del problema a resolver y una estructura común con la que se podría resolver el problema, por lo que, a diferencia de los algoritmos y las librerías, no se trata de líneas de código que se puedes aplicar directamente en nuestro programa, sino de una recomendación de lo que deberá implementar el programador en su código para encontrar la solución. Estos patrones generalmente se dividen en los siguientes tres tipos:

- Patrones de creación: Permiten la creación de los distintos objetos que se utilizarán a lo largo, en este caso, del juego. Parte de los objetivos de estos patrones es separar el código del comportamiento del código de su creación del objeto y evitar que se tenga que estar modificando el patrón con la creación de cada objeto. Algunos ejemplos de estos patrones son Factory Method, Builder, Singleton y Abstract Factory.
- Patrones estructurales: Ayudan a manejar la forma en que se relacionan y comunican los distintos objetos entre sí, estas formas de relacionarse permiten crear estructuras más complejas. Algunos ejemplos de este tipo de patrones pueden ser el caso de Bridge, Decorator, Facade, Composite y Adapter.

- Patrones de comportamiento: Ayudan al programador a llevar un mejor control del comportamiento de los objetos y las distintas interacciones que pueden ocurrir entre ellos. Dentro de este tipo de patrones se encuentran Memento, Mediator, Observer, Strategy, Update o State.

Capítulo 4

Patrones de diseño de software aplicados en el desarrollo de videojuegos

Para la selección de estos patrones de diseño de software que se pueden aplicar dentro de los videojuegos se analizó los mencionados dentro de los libros *Game Programming Patterns* de Robert Nystrom [Nys14] y *Dive into design patterns* de Alexander Shvets[Shv21] junto con la documentación oficial de Unreal Engine (los enlaces a las respectivas partes de la documentación se encuentran dentro de las secciones de los patrones correspondientes). Después de analizar los patrones mencionados en estos libros junto con la documentación de Unreal Engine para verificar si estos patrones han sido aplicados dentro del motor, se llegó a la conclusión de que los siguientes son algunos ejemplos de patrones de diseño de software que pueden ser utilizados dentro del desarrollo de videojuegos.

4.1. Patrones de creación

Los siguientes patrones se podrán utilizar en el desarrollo de juegos para crear de una forma óptima los distintos objetos y actores que serán utilizados

a lo largo del mismo. Crear de una forma adecuada las clases será fundamental para la optimización, esto debido a que estas serán las bases sobre las que muchos de los demás patrones de diseño trabajarán y su mala aplicación o no hacer uso de estos podría provocar un mal funcionamiento tanto al comienzo como a lo largo del juego en caso de que tengamos actores que sean llamados una vez empezada la partida. Por lo que, a continuación estudiaremos algunos de estos patrones que más se pueden encontrar en el desarrollo de videojuegos. [Shv21] [Nys14] [Shv]

4.1.1. Prototype

El patrón prototype nos propone la creación de una clase prototipo que será utilizada como la base sobre la cual se harán "clones", esto con el objetivo de facilitar tareas relacionadas con la creación de instancias similares, es por esto que, este patrón de diseño puede ser muy útil a la hora de crear enemigos dentro de nuestro juego o inclusive al crear objetos como balas o elementos que el jugador podrá recoger.

Para que esto sea posible, nuestra clase prototipo tendrá que contar con una función clone() que después será sobrescrita por sus clases hijas. Cada una de las clases hijas deberá contar con una instancia que servirá como el modelo que le transmitirá a las futuras instancias la información inicial para su funcionamiento, por lo que, está instancia original deberá estar en una zona alejada del jugador para que este no pueda hacer cambios en los valores que serán clonados. Por ejemplo, si nuestro jugador se encontrara con la instancia original de nuestra clase Vampiro y le causa un daño que reduce su salud a la mitad, esto causaría que nuestros futuros vampiros aparezcan con su salud igualmente reducida a la mitad.

Diagrama de clases

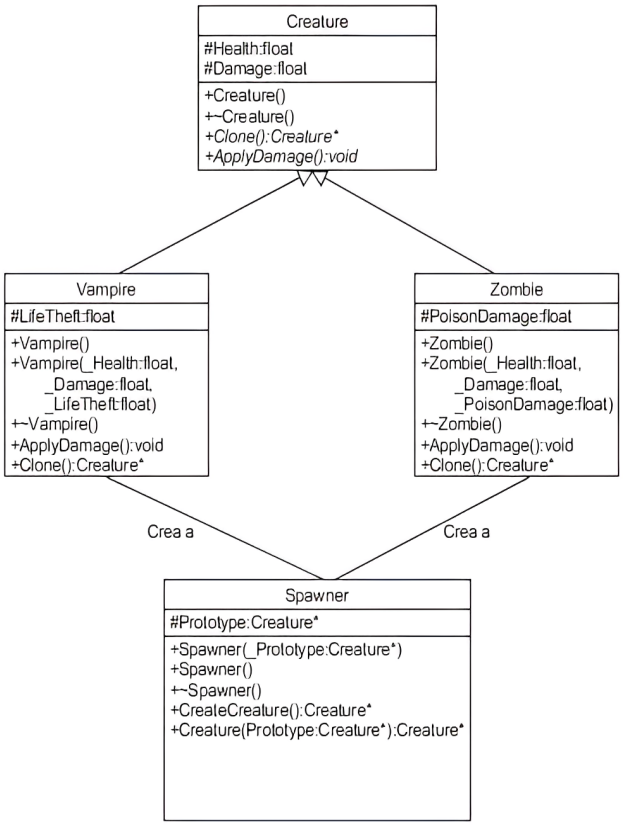


Figura 4.1: Diagrama para los ejemplos de Prototype

Ejemplo 1

Para este ejemplo se utilizará el patrón de diseño Prototype para la creación de una clase Spawner que se encargará de crear distintos tipos de criaturas en lugar de tener que crear un tipo de Spawner para cada una de ellas.
El primer paso será crear una clase Creature que será la clase padre de las clases Zombie y Vampire. Esta clase contará con la función Clone y ApplyDamage que serán sobrescritas por las clases hijas y contará con los atributos Health y Damage que estarán presentes en todas las clases hijas.

```
class Creature:
{
public:
    Creature();
    ~Creature();

    virtual Creature* Clone();
    virtual void ApplyDamage();

protected:
    float Health;
    float Damage;
};
```

Ahora procederemos a crear las clases hijas, para este caso contarán con dos funciones constructoras, la función Clone y un atributo privado que diferencia los efectos del daño de cada clase (el vampiro absorberá vida al dañar y el zombie causará un efecto de veneno). Se debe prestar especial atención en que la función Clone dentro de la clase hija regresa un puntero de la clase Creature pero que en el return tenemos un puntero de la clase hija.

```
#include "Creature.h"

class Vampire : public Creature
{
public:
    Vampire();
    ~Vampire();
    Vampire(float Damage, float Health, float LifeTheft);

    Creature* Clone()
    {
        return new Vampire(Damage, Health, LifeTheft);
    }

    void ApplyDamage();

private:
```

```
    float LifeTheft;
}
```

Una vez que tenemos nuestras clases para las criaturas procederemos a crear la clase Spawner. Para esta clase mostraremos dos versiones que se podrán utilizar en base a los objetivos deseados, la primera versión será en la que crearemos una instancia de Spawner para cada criatura.
Para esta versión nuestra clase Spawner contará con un atributo privado que determinará el tipo de criatura para la que funcionará este spawner, este atributo servirá para retornar el objeto resultante de que se llame a la función Clone del tipo de criatura creada.
Ya que están creadas las clases se crearán los spawners haciendo uso del primer constructor de cada clase, esto permitirá que nuestros spawners guarden la instancia prototipo para hacer los futuros clones.

```
    Spawner ZombieSpawner = Spawner(new Zombie());
    Spawner VampireSpawner = Spawner(new Vampire());
```

La diferencia entre los constructores radica en que el primer constructor hará una lectura en el archivo de configuración para poder obtener los valores iniciales de la instancia original y en el segundo constructor será usado para los clones, en este constructor se recibirán como parámetros los valores que se clonarán de la instancia original.

```
Creature* Clone()
{
    return new Zombie(Health, Damage, PoisonDamage);
}

Vampire::Vampire()
{
    INIReader ConFile("InitialData.ini");

    if(ConFile.ParseError() < 0)
        ConFile.PrintError("Zombie: Confile Failed);

    Health = (float)ConFile.GetInteger("Vampire", "Health", 0);
    Damage = (float)ConFile.GetInteger("Vampire", "Damage", 0);
    LifeTheft = (float)ConFile.GetInteger("Vampire", "LifeTheft",
    0);
```

```
}

Vampire::Vampire(float _Damage, float _Health, float _LifeTheft)
{
    Damage = _Damage;
    Health = _Health;
    LifeTheft = _LifeTheft;
}
```

Creamos las instancias de nuestros zombies y el vampiro haciendo uso de sus respectivos spawners.

```
Creature* ZombieClone = ZombieSpawner.CreateCreature();
Creature* VampireClone = VampireSpawner.CreateCreature();
Creature* ZombieClone2 = ZombieSpawner.CreateCreature();
```

Y llamamos con cada uno su función ApplyDamage para corroborar que aunque cada uno estén guardados como Creatures siguen usando sus respectivas funciones sobrescritas.

```
ZombieClone->ApplyDamage();
VampireClone->ApplyDamage();
ZombieClone2->ApplyDamage();
```

```
Zombie: I've applied 10 of damage and 1 of poison damage.
Vampire: I've applied 5 of damage and I've stolen 10 of health
Zombie: I've applied 10 of damage and 1 of poison damage.
```

Figura 4.2: Resultados del programa

Ejemplo 2

Para esta segunda versión se utilizará una sola instancia de Spawner que servira para crear cualquier tipo de criatura, esto debido a que no se le pasará un prototipo en el constructor y, en cambio, se le pasará este prototipo como parámetro cada que se creará una criatura.

```
Creature* Spawner::CreateCreature(Creature* Prototype)
{
    return Prototype->Clone();
}
```

Ahora se crearán las instancias prototipo de cada clase hija y se quedarán guardadas para que sean enviadas como parámetro cada que deban ser clonadas.

```
Creature* ZombiePrototype = new Zombie();
Creature* VampirePrototype = new Vampire();
Spawner CreatureSpawner = Spawner();

Creature* ZombieClone = CreatureSpawner.CreateCreature(
ZombiePrototype);
Creature* VampireClone = CreatureSpawner.CreateCreature(
VampirePrototype);
Creature* ZombieClone2 = CreatureSpawner.CreateCreature(
ZombiePrototype);
```

El resto del programa trabajará de la misma forma que en la primera versión, lo único que cambio fue el hecho de ahora se crea una instancia Spawner general y en la primera versión se crea una instancia específica para cada criatura.

Ventajas dentro de los ejemplos

Al aplicar este patrón se pudo crear una sola clase Spawner en lugar de tener que crear una clase derivada de Spawner para cada clase derivada de Creature, además de que, se puede evitar un aumento en el número de dependencias si es que se crea una sola clase Spawner en la que se incluya la cabecera de cada clase derivada.
Otra de las ventajas observables es que solamente la instancia original tiene que acceder al archivo de configuración y las instancias clonadas recibirán una copila de los valores iniciales por parte de la instancia original.

4.1.2. Singleton

El patrón de diseño Singleton propone el uso de una única instancia para una clase, a la vez que, se crea un punto de acceso global para que otras clases puedan tener acceso a esta. Este patrón pueden ser de mucha utilidad cuando necesitemos que múltiples clases o instancias tengan acceso a un recurso y puedan a la vez causar que haya cambios en el mismo. Si se tuviera varias instancias capaces de tener acceso a este recurso, podrían generarse problemas debido a que alguna de las instancias quizá no se entere de los cambios que hayan sido

realizados por las demás, es por esto que, al tener una sola instancia podemos controlar mejor lo que está sucediendo.

Algunas de las aplicaciones más comunes que se le puede dar a este patrón dentro de los videojuegos es al crear elementos como un tablero de posiciones en un juego de carreras, el marcador en un juego de disparos o un tablero de puntuaciones.

Una consideración que se debe tener al hacer uso de este patrón, es que pueden conllevar algunos de los mismos problemas generados con el uso de variables globales, como es el caso de la dificultad al encontrar errores debido a la gran cantidad de lugares en las que se puede acceder, en este caso, a la instancia.

Diagrama de clase

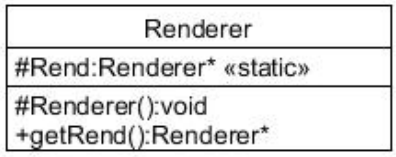


Figura 4.3: Diagrama de clases Renderer

Ejemplo

Un ejemplo de una aplicación de este patrón es al crear una clase Renderer dentro de un juego desarrollado en base a SDL2 y C++, esta clase nos permite presentar y limpiar la pantalla, dibujar figuras, presentarlas en pantalla, incluir imágenes, entre otras funciones de las que se encarga la librería de SDL2. Es por esto que, tener una única instancia de esta clase nos puede ayudar para evitar problemas a la hora de mostrar nuestro juego en pantalla.

Para su implementación debemos comenzar con crear en la parte protected del header file un atributo estático.

```
protected:
    Renderer(void);
    static Renderer* Rend;
```

Posteriormente, se declara en la parte public del header file la función con la que se podrá tener acceso a la instancia creada.

```
class Renderer
{
public:
    static Renderer* getRend();
```

Una vez terminado con el header file, inicializamos Rend con nullptr y creamos la función que se encargará de que solo se pueda tener una instancia y la retorne cada que sea requerida; en este caso, se imprime un mensaje para demostrar que no se deja crear más instancias (esto solo es con fines demostrativos).

```
Renderer* Renderer::Rend = nullptr;

Renderer* Renderer::getRend()
{
    if(Rend == nullptr)
        Rend = new Renderer();
    else
        cour << "No se puede crear otra instancia" << endl;

    return Rend;
}
```

Finalmente, para la creación de la instancia dentro de nuestra función principal se crea llama la función getRend (para demostrar el funcionamiento se tratará de crear dos instancias del Rend)

```
    Renderer* Rend = Renderer::getRend();
    Renderer* Rend2 = Renderer::getRend();
```

Corriendo el juego, la instancia de Renderer se está encargando de mostrar en pantalla todo lo que es generado en cada frame, pero, también podemos observar que no se llamó al constructor una segunda vez y se ha mostrado en pantalla el mensaje de que no es posible crear una segunda instancia.

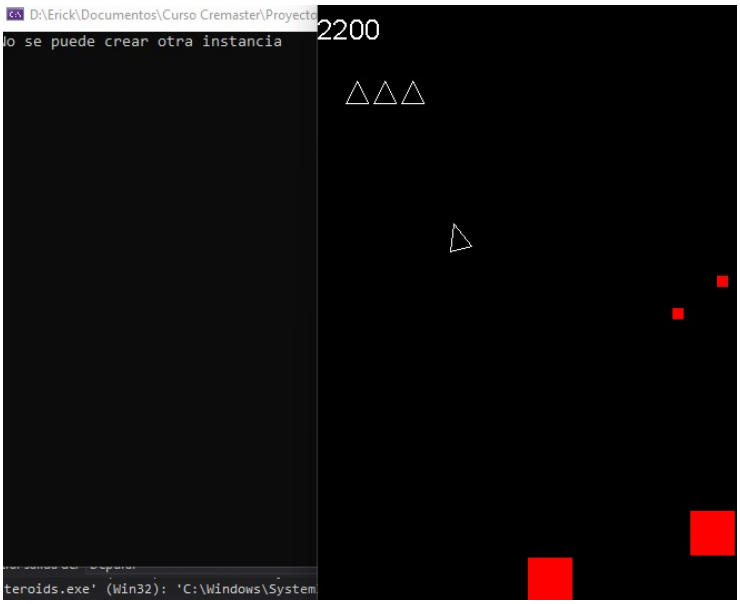


Figura 4.4: Juego corriendo y mensaje de error

4.2. Patrones estructurales

Los patrones de diseño estructurales son aquellos que nos permitirán acoplar las distintas clases entre si para poder crear estructuras que permitan compartir recursos e información entre los elementos que se encuentran en estas mismas.

4.2.1. Flyweight

Este patrón de diseño nos propone una posible solución en situaciones en las que debemos aligerar la carga que hay al crear múltiples objetos que compartan varios elementos en común. Por ejemplo, si estamos diseñando un juego en el que dentro del escenario podamos contar con varias cajas, una muralla conformada por varios segmentos de la misma o un conjunto de rocas, podremos aplicar este patrón.

Para poder hacer esto posible, será necesario dividir cada uno de los atributos del objeto en intrínsecos y extrínsecos. Los intrínsecos serán aquellos que serán iguales todo el tiempo y, por lo tanto, podrán ser compartidos. Por otro lado, los

extrínsecos serán aquellos que estarán cambiando constantemente (como puede ser la posición o incluso la escala).
Una vez que tengamos identificados los atributos extrínsecos e intrínsecos deberemos de dejar estáticos los atributos intrínsecos y crear métodos que hagan posible la manipulación de los atributos extrínsecos. Esto con el objetivo de que solamente se hagan cambios en los atributos que no afectarán a todos los objetos.

Ejemplo

Dentro de Unreal podemos encontrar un tipo de componente que nos puede facilitar esta tarea, se trata de los Hierarchical Instanced Static Mesh Component. Este tipo de componente nos permite crear un array de instancias que directamente compartirán elementos como su mesh, la textura, tipo de colisiones, propiedades físicas, entre otros elementos (estos serían nuestros atributos intrínsecos).

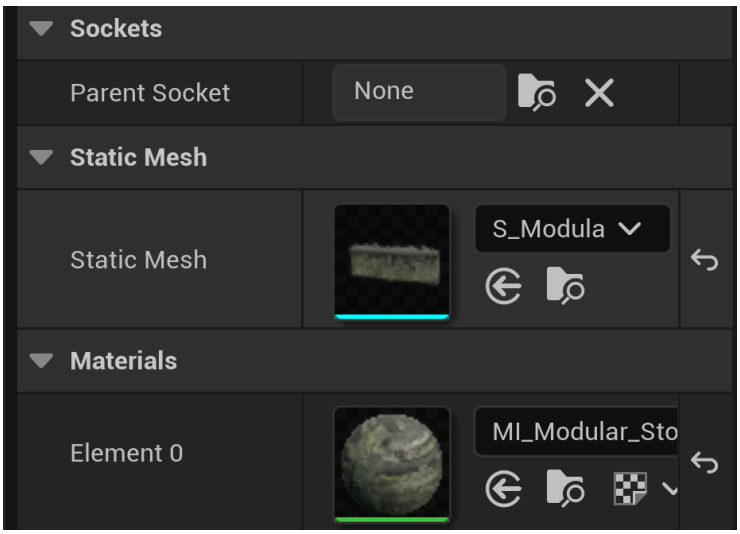


Figura 4.5: Algunos de los atributos intrínsecos

Por otro lado, al agregar elementos al array podemos observar que solamente algunos atributos como la posición, la rotación o la escala puede ser modificada en cada uno de ellos. En la figura 4.6 podemos observar como estos atributos

son independientes en cada uno de los elementos.

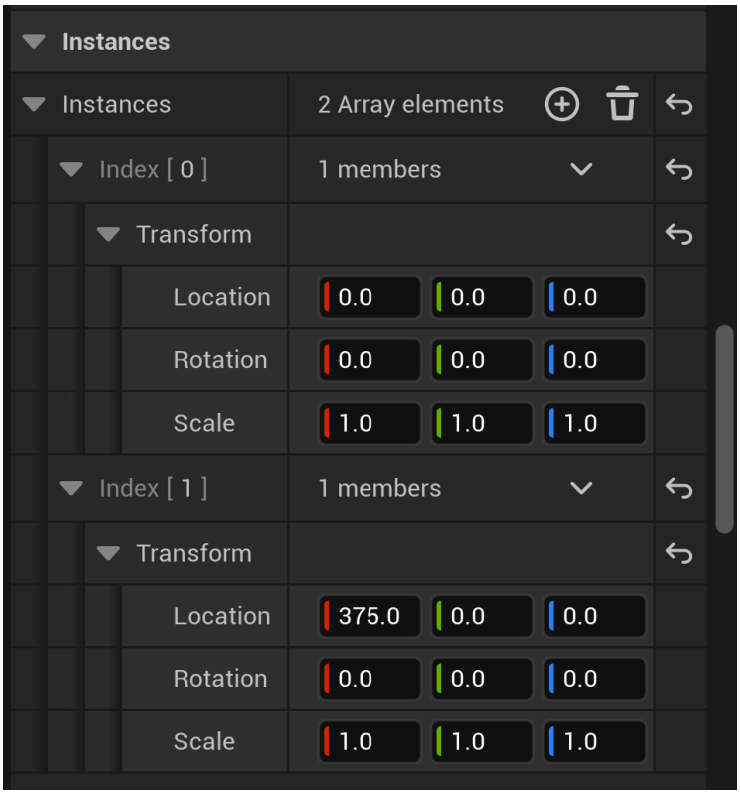


Figura 4.6: Algunos de los atributos extrínsecos

Si la cantidad de elementos que se agregará es demasiado grande es posible crear un ciclo for con el que se agreguen instancias dentro del constructor de nuestro objeto.

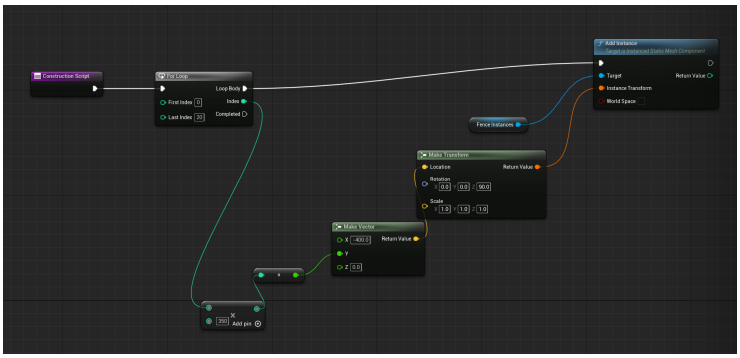


Figura 4.7: Creación de las instancias dentro del constructor

Lo que nos permite crear un conjunto de segmentos de muralla de una forma óptima.

Figura 4.8: Muralla dentro del mapa

Es importante mencionar que además de compartir atributos los elementos de nuestra muralla estas serán dibujadas en una sola llamada, por lo que realizar estos tipos de objetos también ayudará a disminuir el tiempo de dibujado dentro del juego.

Algunos puntos a tomar en cuenta al utilizar los Hierarchical Instanced Static Mesh Component es que si se necesita crear instancias muy lejanas es recomendable que estas estén por separado (no crear todas las instancias similares de un mapa dentro de un mismo Hierarchical Instanced Mesh Component) y que si será requerido trabajar con cada instancia por separado lo más conveniente sería no meterlas dentro del mismo component.

4.2.2. Spatial Partition

Dentro de todo videojuego es vital que nuestro jugador y los NPC (tanto aliados como enemigos) estén constantemente interactuando con otros personajes y objetos dentro del mapa (ya sea explícita o implícitamente), pero, eso no significa que cada uno de ellos deba verificar si hubo interacciones con absolutamente todos los actores y objetos presentes.

Por ejemplo, si nuestro personaje viene al mercado del pueblo más cercano después de una feroz batalla contra un dragón y deseamos vender los objetos que han sido obtenidos, ¿tendría sentido verificar si hubo alguna colisión con un puesto o personaje que se encuentra al otro lado del mercado? ¿tendría sentido dar la opción de comerciar con algún comerciante que igualmente se encuentra en esa zona? Claramente la respuesta es no, y es por esto que, el patrón Spatial Partition nos propone una forma de ahorrar recursos que serían malgastados en operaciones innecesarias.

La propuesta de este patrón consiste en dividir el terreno de nuestro mapa en distintas secciones para que los objetos solamente tengan en consideración a los otros objetos que se encuentran dentro de la misma sección o secciones adyacentes. Al realizar esta separación en secciones podemos evitar hacer comparaciones innecesarias y limitarnos solamente a las necesarias, aunque se debe de ser muy cuidadoso para evitar errores en las intersecciones y se debe de tomar en cuenta todos los posibles escenarios a la hora de seleccionar que secciones se tomarán en cuenta.

Diagrama de clases

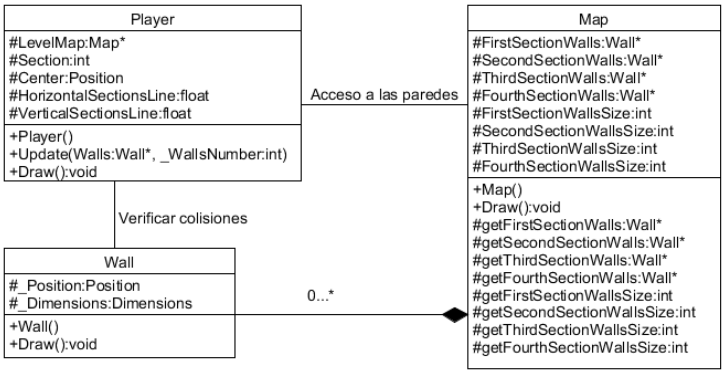


Figura 4.9: Diagrama de clases

Ejemplo

Una forma de demostrar cómo puede ser de utilidad este patrón es con el tablero del juego Pacman, dentro de este juego el jugador debe mover a Pacman dentro de un laberinto con el objetivo de comer todas las píldoras sin ser tocado por los fantasmas (a menos que se tome la pastilla que permite comer fantasmas). Al estar dentro de un laberinto, el jugador no es capaz de colisionar con todas las paredes del mapa, ni es capaz de comer cualquiera de las pastillas, solamente es posible que interactué con los objetos que tiene en la cercanía. Con propósitos demostrativos tomaremos solamente en cuenta para este ejemplo las paredes (aunque para las pastillas sería prácticamente el mismo funcionamiento).
Así es como el jugador ve el tablero:

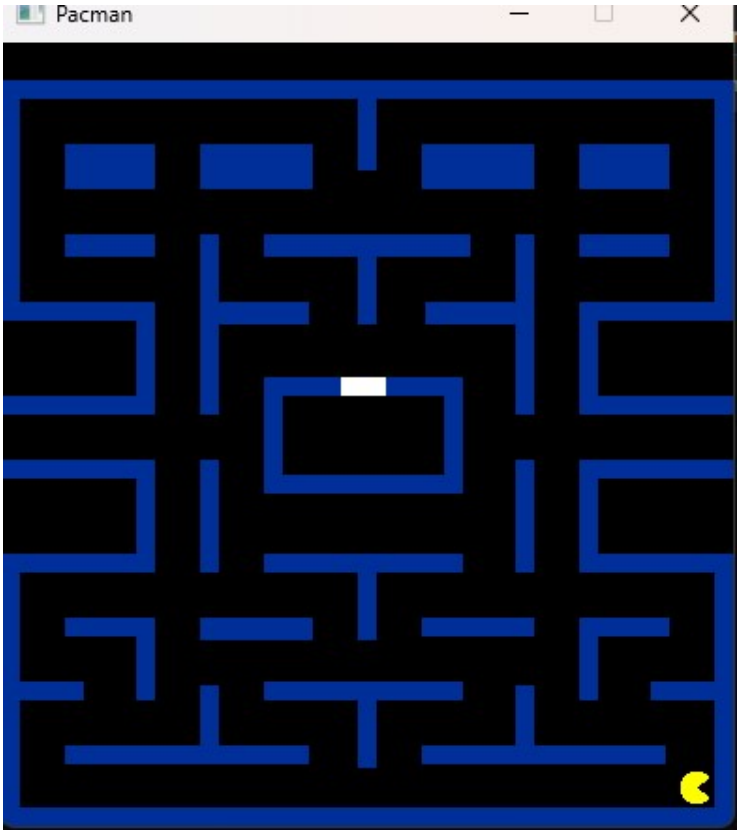


Figura 4.10: Mapa de Pacman

Pero internamente el mismo se encuentra dividido en cuatro sectores.

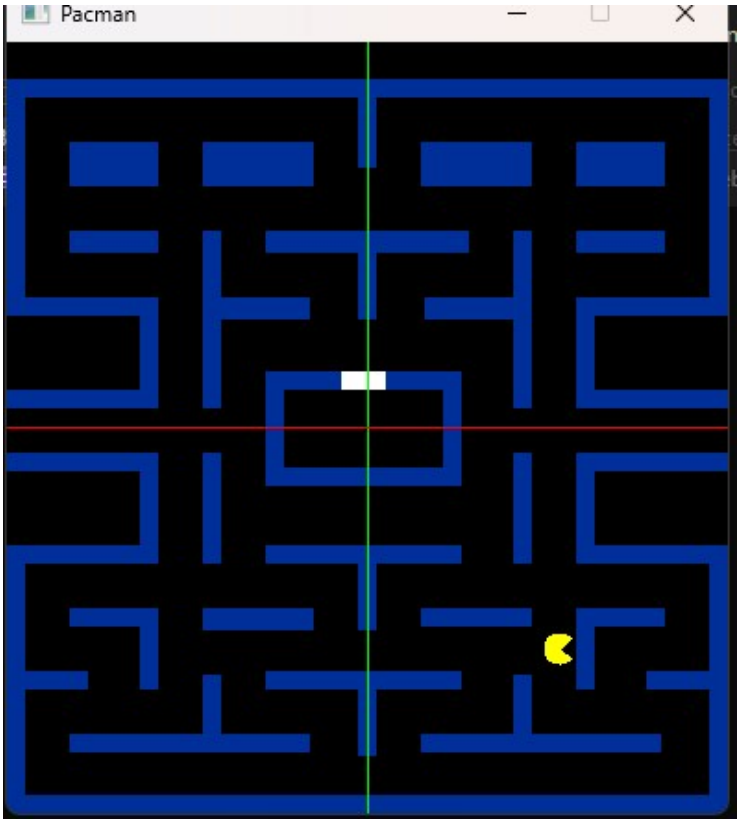


Figura 4.11: Mapa de Pacman con divisiones

Para poder identificar con cuales paredes debe ser revisada la colisión, nuestra clase jugador llama a su función UpdateSection desde el Game Loop, dentro de la cual se encarga de revisar la sección en la que se encuentra en base a sus coordenadas.

```
void Player::UpdateSection()
{
    if(Center.x <= HorizontalSectionsLine)
    {
        if(Center.y <= VerticalSectionsLine)
            Section = 1;
        else
            Section = 3;
    }
    else
```

```
    {
        if(Center.y <= VerticalSectionsLine)
            Section = 2;
        else
            Section = 4;
    }
}
```

Una vez identificada su sección, se utiliza un switch desde el Game Loop para pasarle al jugador el arreglo de direcciones de las paredes que se encuentran en esa sección junto con el número de las mismas y así pueda verifica únicamente si hay colisión con las paredes cercanas.

```
switch(MainPlayer.getSection())
{
    case 1:
        MainPlayer.Update(MainMap.getFirstSectionWalls(),
                    MainMap.getFirstSectionWallsSize());
    break;

    case 2:
        MainPlayer.Update(MainMap.getSecondSectionWalls(),
                    MainMap.getSecondSectionWallsSize());
    break;

    case 3:
        MainPlayer.Update(MainMap.getThirdSectionWalls(),
                    MainMap.getThirdSectionWallsSize());
    break;

    case 4:
        MainPlayer.Update(MainMap.FourthSectionWalls,
                  MainMap.getFourthSectionWallsSize());
    break;
}
```

Una ventaja adicional que podría aprovecharse en caso de que nuestro juego fuera 3D o el mapa fuera tan grande que no todos los objetos podrían aparecer en pantalla a la vez es que este patrón nos añadió un mejor control de los objetos a la hora de dibujarlos en pantalla.

```
void Map::Draw()
```

```
{
    for(i = 0; i < FirstSectionWallsSize; i++)
        FirstSectionWalls[i].Draw();
    for(i = 0; i < SecondSectionWallsSize; i++)
        SecondSectionWalls[i].Draw();
    for(i = 0; i < ThirdSectionWallsSize; i++)
        ThirdSectionWalls[i].Draw();
    for(i = 0; i < FourthSectionWallsSize; i++)
        FourthSectionWalls[i].Draw();
}
```

Como podemos observar dentro de esta función de la clase Map, en caso de que quisiéramos ahorrar recursos solo dibujando lo necesario, sería tan sencillo como poner condicionales para que solamente entren los ciclos que dibujan los objetos requeridos y no sería necesario seleccionarlos de uno en uno.

4.3. Patrones de comportamiento

Estos patrones permitirán dentro del desarrollo de juegos facilitar tareas del comportamiento de las distintas clases con las que vayamos a contar, además de asignar correctamente las responsabilidades que tendrán cada una de estas. Aplicar correctamente estos patrones ayudarán a resolver algunos de los problemas más comunes dentro del desarrollo de videojuegos, algunos de estos están presentes en prácticamente cada uno de estos.

4.3.1. Command

Dentro de la programación muchas veces es necesario que nuestro usuario pueda realizar una misma acción de distintas maneras posibles y los videojuegos no son la excepción, un ejemplo de esto es cuando nuestro juego tiene soporte para gamepad, teclado y ratón; cuando esto sucede, es necesario que tanto al oprimir un botón del teclado o el gamepad o hacer click en una determinada zona (en el caso, por ejemplo, un menú) se realice una misma acción.
Para facilitar esta tarea, el patrón de diseño command nos propone la creación clases command cuyos objetos sean los encargados de "transmitir" ordenes al objeto deseado. Esto puede facilitar acciones como el cambio de input para

ciertas acciones, el uso de varios inputs para las mismas o detonar acciones dentro del escenario.

Diagrama de clases

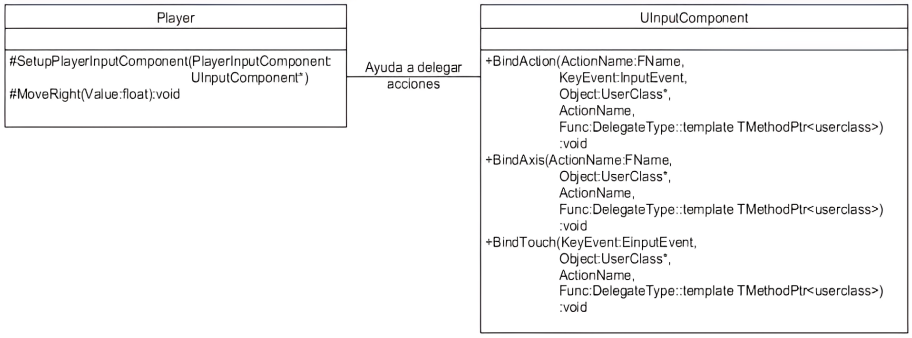


Figura 4.12: Diagrama de clases

Ejemplo

Un posible ejemplo de la aplicación de este patrón lo podemos encontrar dentro de Unreal al definir los inputs dentro de nuestro juego, para esto debemos hacer click en la opción “Edit” de Unreal, posteriormente seleccionar “Project Settings” y finalmente seleccionar “Input”.

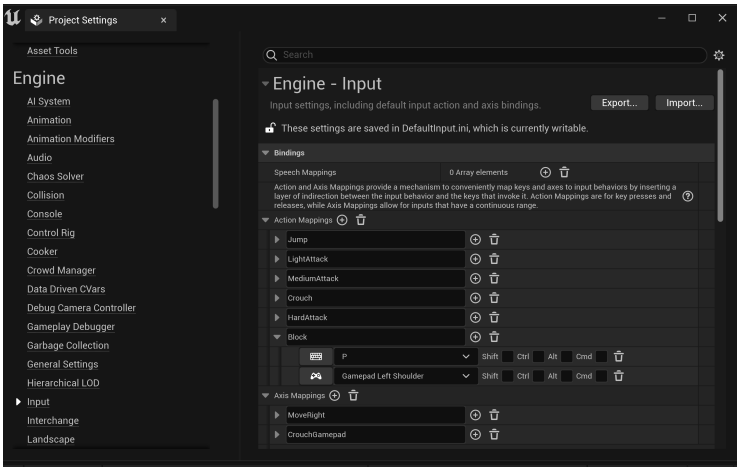


Figura 4.13: Menú de Input

En este menú podremos agregar los comandos que se detonarán al usar las teclas/botones/joysticks seleccionados. Se debe tomar en cuenta que estos vienen divididos en dos secciones, Action Mappings para aquellas acciones que requieran de presionar y soltar una tecla (como en el caso de un ataque o un salto) y Axis Mappings para aquellas que requieren de una entrada constante (como puede ser el caso del movimiento).
Una vez que tenemos configurado adecuadamente nuestros inputs, podremos configurar en nuestras clases que puedan ser poseídas las acciones que se realizaran cuando reciban cada comando. Para esto haremos uso de una función que recibirá una puntero de un objeto de la clase UInputComponent [Eng22k] (una clase ya definida por Unreal encargada de delegar las funciones en base a los input que recibe).

```
void AFightGameCharacter::SetupPlayerInputComponent(class
    UInputComponent* PlayerInputComponent)
{
    //set up gameplay key bindings
    PlayerInputComponent->BindAction("Jump", IE_Pressed, this,
                                &AFightGameCharacter::Jump);
    PlayerInputComponent->BindAction("Jump", IE_Released, this,
                          &AFightGameCharacter::StopJumping);
    PlayerInputComponent->BindAxis("MoveRight", this,
                            &AFightGameCharacter::MoveRight);

    PlayerInputComponent->BindAction("LightAttack", IE_Pressed,
                  this, &AFightGameCharacter::LightAttack);
    PlayerInputComponent->BindAction("MediumAttack", IE_Pressed,
                this, &AFightGameCharacter::MediumAttack);
    PlayerInputComponent->BindAction("HardAttack", IE_Pressed,
                  this, &AFightGameCharacter::HardAttack);
}
```

Dentro de la función haremos uso de las funciones BindAction y BindAxis dependiendo de como hayamos configurando nuestros inputs anteriormente. En caso de usar la función BindAction nuestro primer parámetro será el nombre que le dimos a nuestro comando dentro de Project Settings, el siguiente será uno de los posibles valores del enum EInputEvent [Eng22d] (los más usados

IE_Pressed e IE_Released), una referencia al objeto que realizará la acción y ,finalmente, la función que será llamada.
En caso de usar BindAxis seguiríamos el mismo proceso pero sin agregar el parámetro EInputEvent, esto debido a que nuestra entrada es constante. Otra diferencia respecto a BindAction es que la función delegada podrá recibir un valor flotante que le indicará el valor en el Axis (que tanto estamos moviendo el joystick o si estamos presionando una tecla); este valor nos puede permitir realizar acciones en base a este valor, como por ejemplo, que el jugador se mueva a determinada velocidad al recibir determinado valor o la dirección de este movimiento en caso de que el valor sea positivo o negativo.

```
void AFightGameCharacter::MoveRight(float Value)
{
    ...
}
```

4.3.2. Game Loop

Este patrón de diseño es de los pocos que se aplicarán en todos los videojuegos, esto debido a que un videojuego es un ciclo constante de entradas por parte del jugador (inputs), el procesamiento de las acciones pertinentes y la salida en pantalla (output) con la que el jugador responderá para iniciar nuevamente el ciclo.
Es por esto que al crear nuestro game loop debemos considerar siempre estos tres aspectos indispensables para el funcionamiento de nuestro juego, respetando el orden previamente mencionado.

Ejemplo

Dentro del siguiente ejemplo de un Game Loop podemos observar como iniciamos un ciclo while que estará trabajando mientras que una variable booleana que determina si el juego está corriendo tenga un valor true. Dentro del ciclo contamos con la función que lee las entradas del jugador por medio del teclado, posteriormente se llama a la función que actualiza el juego en base a las entradas y demás elementos dentro del juego, y, finalmente, se llama a la función encargada de dibujar los elementos del juego en pantalla.

```
void Game::RunLoop()
{
    while(mIsRunning)
    {
        ProcessInput();
        UpdateGame();
        GenereateOutput();
    }
}
```

4.3.3. Observer

En los videojuegos es común que se ejecuten determinadas acciones dependiendo de lo que haga el jugador, esto puede ser para actualizar el HUD, desbloquear recompensas tras cumplir determinadas tareas, desbloquear logros o trofeos (dependiendo de la plataforma en la que se esté jugando), entre otras situaciones.

Se podría realizar la parte lógica para determinar si los requisitos han sido cumplidos dentro de la misma parte del código en la que sucede el detonante, pero esto podría causar un gran desorden dentro de nuestro código y que el nombre de nuestras funciones no corresponda del todo con lo que está pasando dentro de las mismas. Por ejemplo, si contamos con un logro que nos pida derrotar 500 ogros, otro que sea por derrotar 1000 enemigos con una lanza, otro por derrotar 5 enemigos en menos de 10 segundos, otro por derrotar 20 enemigos sin haber sido dañado, entre muchos otros dentro del juego, no sería lo más adecuado que al causarle daño a un enemigo su función TakeDamage empiece a realizar todas las preguntas necesarias para saber si alguna de estas condiciones se ha cumplido por completo o se actualice el valor en caso de que el enemigo haya sido eliminado.

Es por esto que, para evitar ese tipo de desorden dentro de nuestro código y mejorar la interacción entre nuestras clases el patrón Observer nos propone el uso de clases encargadas de identificar por medio de notificaciones las situaciones en las que nuestras condiciones deseadas puedan ser cumplidas. Una vez que tengamos nuestras clases podremos notificarles desde nuestras funciones originales que algún evento acaba de ocurrir y así nuestras nuevas clases observer harán

sus respectivos trabajos en sus funciones correspondientes.

Diagrama UML

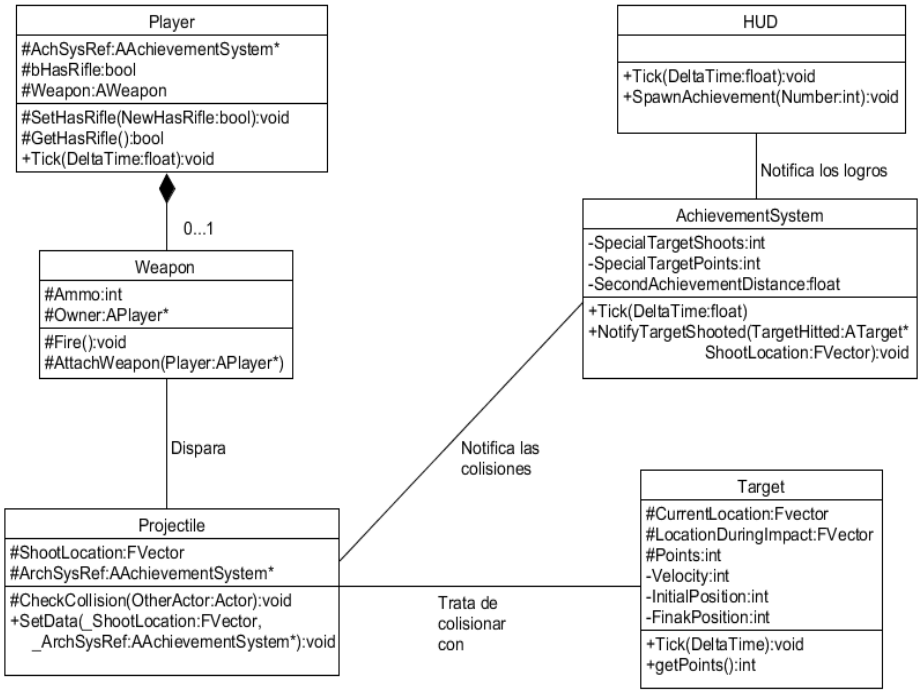


Figura 4.14: Diagrama UML

Ejemplo

Para este ejemplo se creará una clase observer llamada AchievementSystem que se encargará de identificar las situaciones en las que el jugador cumpla con los requisitos para obtener uno de los logros dentro de un juego de disparo al blanco.

El primero de estos logros consistirá en que el jugador haya logrado darle todos los tiros al blanco que es más pequeño y más veloz, por otro lado, el segundo consistirá en dispararle a este mismo blanco pero a partir de cierta distancia.

Lo primero que haremos será crear nuestra clase AchievementSystem y agrega-

remos una función Notify para los casos en los que nuestros proyectiles colisionen con algunos de los objetivos. Esta función recibirá la localización de donde se originó el disparo y una referencia del objetivo que recibió el tiro.

```
protected:
    UFUNCTION(BlueprintCallable)
    void NotifyTargetShooted(ATarget* TargetHitted, FVector
    ShootLocation);
```

En este caso se utilizó como base el template para juegos de disparos en primera persona de Unreal, por lo que, además de las clases creadas para el ejemplo, modificaremos tres de las clases creadas por defecto por Unreal.
La primera clase a modificar será la clase del personaje que usa el jugador, para eso crearemos un atributo dentro de la clase que sea una referencia a nuestra clase AchievementSystem (este valor deberá ser asignado a nuestro personaje después de que nuestra instancia de AchievementSystem sea creada). Dentro de la clase del proyectil agregaremos un atributo para guardar la referencia a nuestra instancia AchievementSystem junto con un FVector en el que guardaremos las coordenadas de donde fue disparada el arma. Además de estos atributos crearemos una función en la que se podrán asignar los valores que se pasarán como parámetros a la hora de crear la instancia de la munición.
Finalmente, se modificará la línea donde se crean los proyectiles dentro de nuestro cpp de la clase del arma, esto con el objetivo de pasarle la información de la localización y la instancia de nuestro AchievementSystem al proyectil.

```
World->SpawnActor<ATargetShootingProjectile>(ProjectileClass,
SpawnLocation, SpawnRotation, ActorSpawnParams)->setData(
Character->GetActorLocation(), Character->getchSysRef());
```

Una vez que se realizaron las modificaciones pertinentes en las clases generadas por Unreal se revisará si la colisión de nuestros proyectiles fue realizada contra alguno de los objetivos y, en caso de que esto suceda, nuestra instancia de AchievementSystem recibirá la información necesaria sobre el proyectil y el objetivo con el que hubo la colisión para así poder verificar si se cumplen las condiciones para alguno de los logros relacionados.

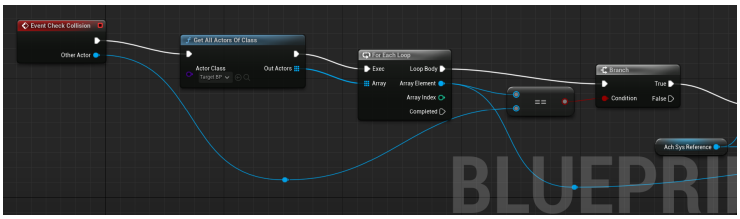


Figura 4.15: Se compara si hubo colisión con algún objetivo

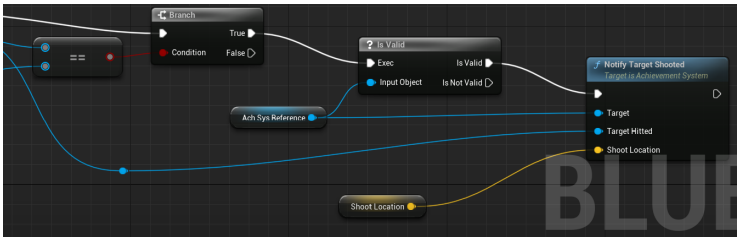


Figura 4.16: Llamado del Notify de AchievementSystem

Dentro de la función Notify se realizarán las operaciones necesarias para verificar si se cumplen las condiciones para alguno de los dos logros relacionados al objetivo especial, dejando así en una sección adecuada del código estas operaciones que no tienen que ver con la función CheckCollision en sí.

```
void AAchievementSystem::NotifyTargetShooted(ATarget* TargetShooted
    , FVector ShootLocation)
{
    if(TargetHitted->getPoints() == SpecialTargetPoints)
    {
        SpecialTargetsShoots--;
        if(SpecialTargetShoots == 0)
            GameModeRef->NotifyAchievementToHUD(1);
        if(CalculateDistance(targetHitted->GetActorLocation(),
                        ShootLocation)<= AchievementDistance)
            GameModeRef->NotifyAchievementToHUD(2);
    }
}
```

Otro de los aspectos que quedarían fuera de nuestra función CheckCollision y es necesaria para nuestro sistema de logros es el llamado de la función que se encargará de desplegar el logro dentro de nuestro HUD.

Figura 4.17: Logro desbloqueado

Ventajas dentro de este ejemplo

Dentro de este ejemplo de un pequeño juego de disparo al blanco podemos observar las ventajas de que al crear cada proyectil no es necesario mencionarle a este cual es el objetivo especial o las veces que se ha golpeado al mismo, en su función para determinar si golpeó a los objetivos no se mezcla el código de la función en si con el código para determinar si se golpeó el proyectil especial, las veces que se ha golpeado a este o la distancia a la cual se le disparó. Otra de las ventajas es que cada proyectil no tiene que contar con una referencia a nuestro HUD, ya que, la tarea de notificar que un logro ha sido desbloqueado queda asignada a nuestra instancia de AchievementSystem.

4.3.4. State

Este patrón de diseño nos propone el uso de "Estados" en los que se va a encontrar algún objeto, por lo que, está estrechamente ligado con el concepto de las máquinas de estado. Un estado dentro de este contexto puede definirse como la situación en la que se encuentra una instancia de una clase o la acción que se encuentra realizando. Dependiendo de este estado, nuestra máquina se encargará de cambiar aspectos de la instancia en cuestión, como puede ser la animación

que realizará, la gravedad que le afectará, el tipo de movimiento que tendrá, entre otros.Dentro del desarrollo de juegos, esto nos puede ser demasiado útil para saber qué es lo que se encuentra haciendo el jugador, un arma, un enemigo o cualquier NPC para posteriormente actualizar su comportamiento.
Para llevar a cabo este patrón, es necesario definir los estados en los que se podrá encontrar el objeto en cuestión, definir la función que se encargará de realizar los cambios dependiendo del estado (probablemente con un switch) y cambiar el estado dentro del código dependiendo de la entrada que se haya registrado por parte del control, de si se ha registrado un ataque al jugador, se ha detectado que se ha entrado a una determinada área (como puede ser el agua o un espacio sin gravedad), entre otros factores que puedan causar el cambio en el estado.
En el caso de estar trabajando dentro de Unreal, hay otra alternativa para la implementación de este patrón que puede facilitar mucho el trabajo, se trata de los Animation Blueprints[1] [Eng22b].

Ejemplo 1

Para crear un Animation Blueprint es necesario dar click derecho sobre el Content Browser de Unreal y posteriormente seleccionar el apartado de Animations y Animation Blueprint.

[1]Herramienta de Unreal que facilita tareas relacionadas con las animaciones de los actores

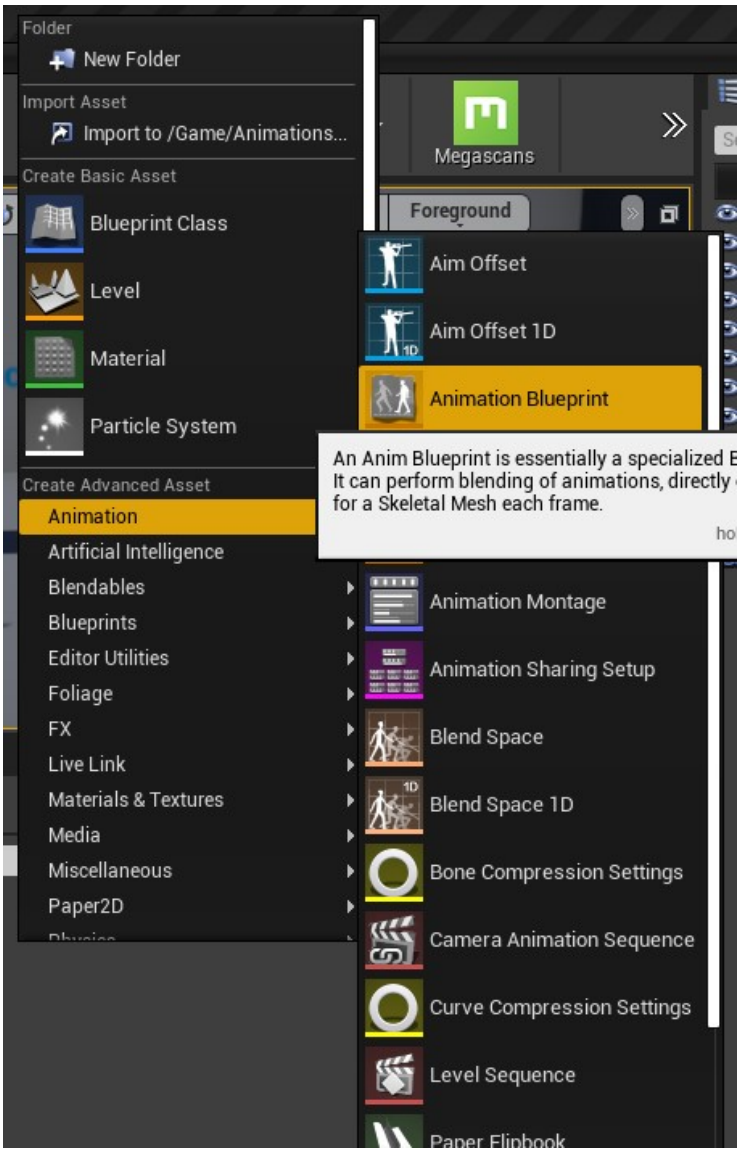


Figura 4.18: Creación de Animation Blueprint

Al comenzar a crear el Animation Blueprint, Unreal nos pedirá seleccionar el esqueleto con el que se estará trabajando.

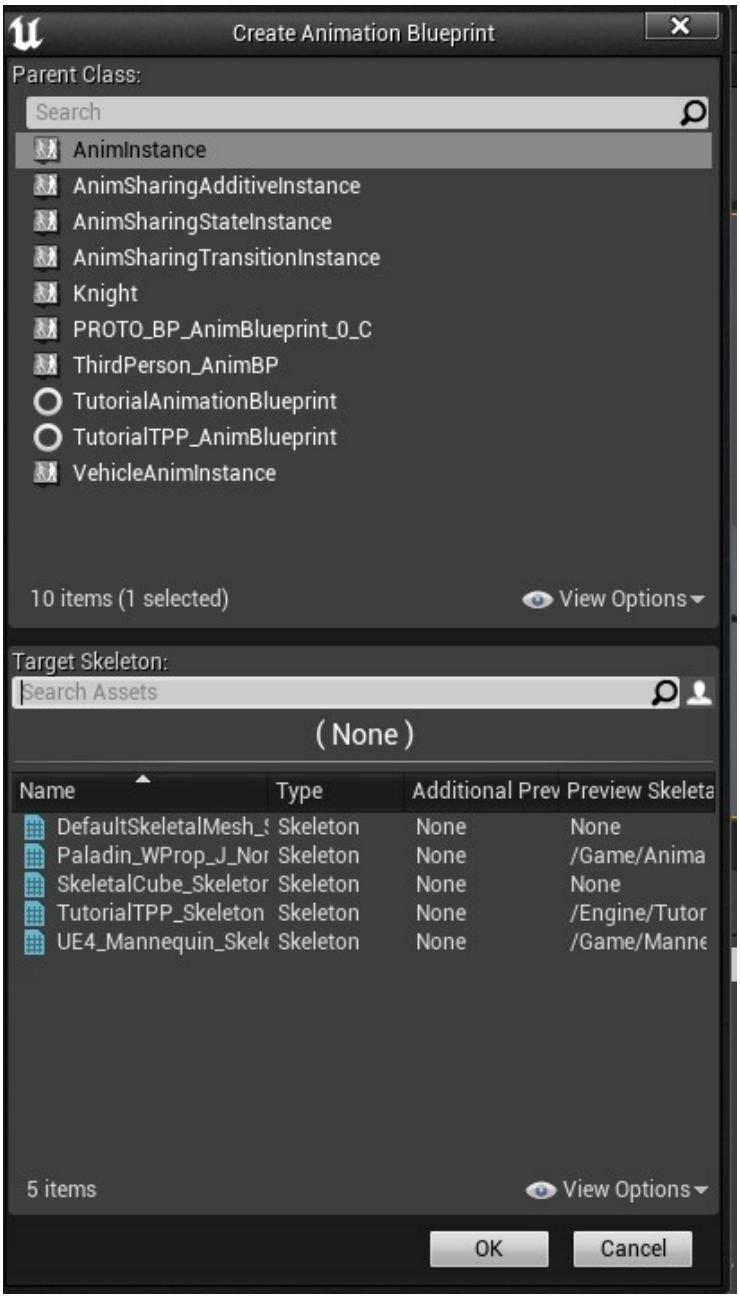


Figura 4.19: Selección del esqueleto

Una vez terminada la etapa de creación, se generará el Event Graph y la

State Machine [Eng22i]; dentro de nuestro Event Graph se podrá hacer uso de los Blueprints para poder activar o desactivar las variables que determinarán el cambio de estado dependiendo de si se revisarán los cambios cada que las animaciones se actualicen o haciendo uso de Notifys.
Para crear un Notify se necesitará seleccionar una de nuestras animaciones que aparecerán en la parte inferior derecha.

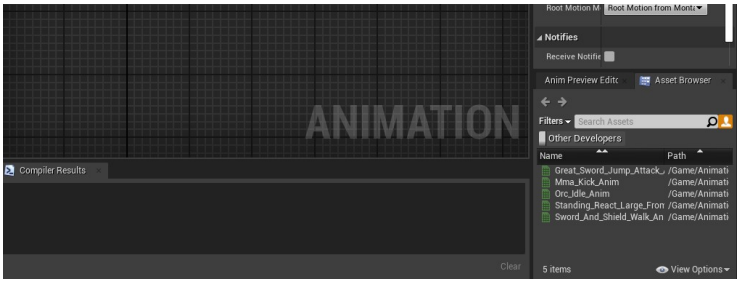


Figura 4.20: Selección de animaciones

Dentro de la visualización de la animación, crearemos una nueva Notify dando click derecho sobre la línea del tiempo y seleccionando Add Notify y New Notify.

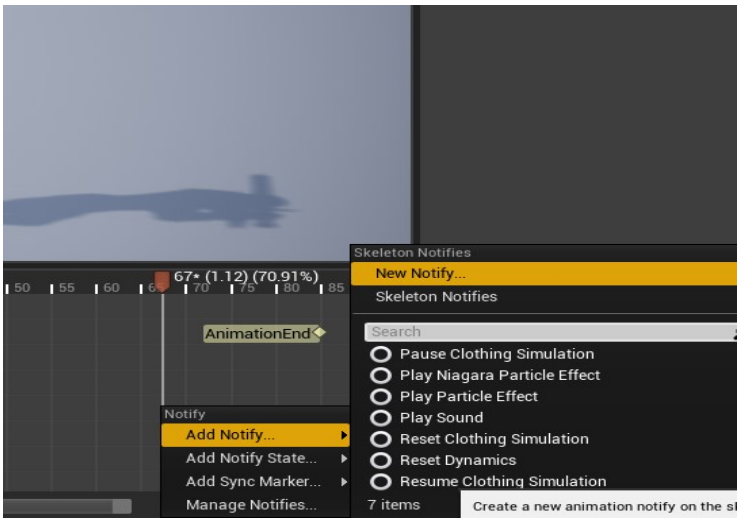


Figura 4.21: Creación de Notify

Ya que hemos creado una Notify, será posible llamarla como si fuera una función dentro del Event Graph para así cambiar las variables o realizar otras acciones dependiendo de si queremos ser notificados cuando la animación termine, acaba de comenzar o ha comenzado determinada acción dentro de la misma. Por otro lado, ya que contamos con las variables que determinarán el cambio del estado, podemos comenzar a trabajar con la State Machine. Dentro de este apartado contaremos con un Entry (el comienzo de la máquina) y deberemos conectar al mismo un State; para crear un State solo hay que dar clic derecho sobre un espacio vacío y seleccionar la opción de Add State. Esto creará un State que al ser conectado con algún otro generará un flecha de flujo junto con otras flechas dentro de un círculo.

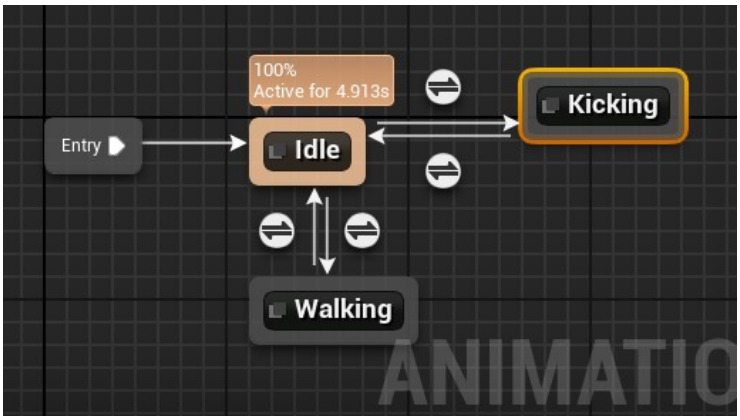


Figura 4.22: State Machine

Las flechas que se encuentran dentro de un círculo blanco nos permitirán (además de muchas otros elementos a configurar) seleccionar lo que debe ser cumplido para que haya un cambio en el estado de nuestra instancia; si le damos un solo clic al círculo, se desplegará un menú en la parte derecha de Unreal que permitirá hacer la transición automáticamente de un estado a otro (recomendable para regresar de un determinado estado a un estado Idle o Inactivo). En caso de que se desee hacer el cambio de animación, al ser pulsado un botón u por medio de otra acción que denote un cambio en una variable, se puede dar doble clic dentro del círculo para poder determinar el cambio por medio de Blueprints. En este sencillo ejemplo podemos observar como el valor de una

variable Bool llamada isWalking nos determinará si el estado Walking será el estado actual.

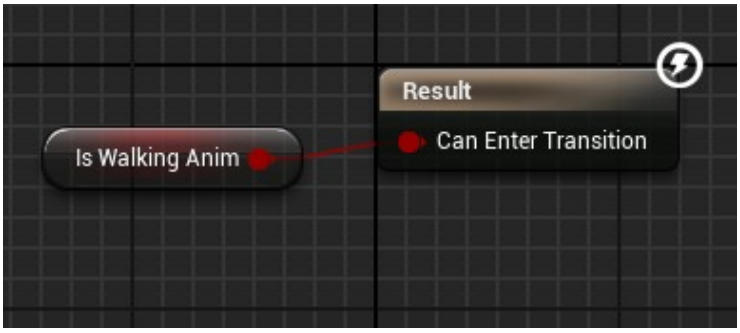


Figura 4.23: Cambio de estado

Finalmente, se deberá seleccionar la animación que será usada dentro del estado correspondiente, para esto, primero se tiene que dar doble clic dentro del state y después se debe arrastrar desde la parte inferior derecha la animación deseada y conectarla al modulo de la animación de salida que será generado por defecto.

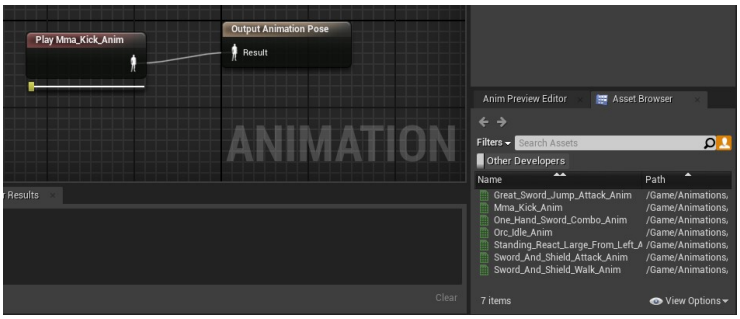


Figura 4.24: Animación que se usará

Ejemplo 2

En caso de que se quiera aplicar este patrón sin la ayuda de las máquinas de estado de Unreal, se puede crear un enum con los estados deseados; en este caso, cada estado nos ayudará a determinar las posiciones y las dimensiones de nuestra Collision Capsule dependiendo de la postura de nuestro personaje.

```
enum class EPosition : uint8
{
    StandUP,
    Blocking,
    Crouched,
    BlockingCrouched
};
```

Una vez que contamos con nuestros posibles estados se deberá seleccionar el estado adecuado dependiendo de la situación, en esta parte del ejemplo nuestro personaje se agachará, aunque antes de poner el estado de agachado, se revisa si el personaje se encuentra bloqueando, por lo que el estado final puede ser el de agachado o agachado bloqueando.

```
    if(canMove)
    {
        isCrouching = true;

        if(!isBlocking)
            CurrentPosition = EPosition::Crouched;
        else
            CurrentPosition = EPosition::BlockingCrouched;
    }
```

Una vez que contamos con el estado actual del personaje se puede hacer uso de un switch que cambie la posición y el tamaño dependiendo del estado de nuestro personaje.

```
void AFightGameCharacter::SetCollidersPosition()
{
    switch(CurrentPosition)
    {
        case EPosition::StandUp:
            CollisionCapsule->SetRelativeTransform(CapsuleTransform
                                                                );
            if(!isFlipped)
                HeadMesh->SetRelativeLocation(HeadHurtboxTransform.
                                                GetLocation());
            else
                HeadMesh->SetRelativeLocation
                                    (FlippedHeadHurtboxLocation);
        break;
```

```
    }
}
```

4.3.5. Update

Se trata de uno de los patrones de diseño fundamentales para el desarrollo de videojuegos y nos propone crear una función que se encargue de actualizar elementos como la posición, las acciones, cambios de variables y otros aspectos de cada instancias de determinadas clases. El uso de esta función para actualizar el estado de las instancias suele ser en cada frame para algunos elementos importantes del juego como es el caso del jugador, las armas que este maneja o algunos enemigos, pero, no debe aplicarse todo el tiempo ni para todos los objetos. Si usamos esta función en desmedida el rendimiento del juego será claramente afectado debido a todo el código a procesar en cada uno de los frames, es por esto que, la función de Update deberá ser aplicada solamente cuando sea necesario actualizar algún aspecto del juego o, por lo menos tener alguna bandera para avisarle a la función que deberá realizar solamente acciones básicas (como presentar el objeto en pantalla).
Algunas de las razones por las que probablemente no necesitamos actualizar en cada frame algunas de las instancias puede ser porque esta se encuentre demasiado lejos o fuera de la vista del jugador, porque esta no tenga ninguna función más que estética o en ese momento no sea necesario que realice función alguna.

Ejemplos

Al generar algunas clases de C++ en Unreal usando clases base, está vendrá por defecto con una variable que determinará si el objeto generado se actualizará en cada frame, en caso de que dejemos la variable en true, esta provocará que el contenido dentro de la función tick sea llamado en cada frame. Cabe a resaltar que, en los mismos comentarios que se generan con el código Unreal, se nos menciona que se puede poner en false la variable (en caso de ser requerido) para mejorar el rendimiento del juego.

```
AHitbox::Ahitbox()
```

```
{
    /*Set this actor to call Tick() every frame.
    You can turn this off to improve perfomance if you don't
    need it*/
    PrimaryActorTick.bCanEverTick = true;
}
//Called every frame
void AHitbox::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}
```

En caso de que queramos crear nuestra propia función Update, es posible usar algunas variables para determinar qué acciones van a ser usadas en cada frame, por ejemplo, en este código escrito usando SDL2 se usa la variable booleana *Pause* para determinar si el jugador será capaz de moverse, disparar, rotar y demás acciones.

```
void Player::Update(bool Pause)
{
    if(!Pause)
    {
        DeltaTime = Rend->getDeltaTime();
        ...
    }
}
```

Sin embargo, se deja fuera el llamado a la función que dibuja al jugador para que este siga apareciendo en pantalla mientras el juego se encuentra en pausa.

```
        CurrentCooldown--;
        DamageCooldown--;
    }
    Rend->DrawTriangle(&FirstPoint, &SecondPoint, &ThirdPoint,
                                        255, 255, 255, 255);
}
```

Capítulo 5

Metodología propuesta

La siguiente metodología fue realizada con el propósito de ayudar a los desarrolladores de videojuegos con las tareas de diseño y programación de mecánicas de juego. Para lograr estos objetivos se abordarán diferentes etapas que ayudarán a dividir las tareas en segmentos más pequeños y que a la vez permitirán llevar una planeación ordenada que ayude a evitar contratiempos a la hora de programar.

Para el diseño de las etapas se utilizó la experiencia adquirida durante la clase de "Desarrollo de Videojuegos" cursada dentro de la Benemérita Universidad Autónoma de Puebla junto con el estudio de otros tipos de metodologías de desarrollo de software y el proceso de investigación sobre diseño de mecánicas de juego y programación de juegos tanto en Unreal Engine como C++ en general.

Antes de comenzar con las fases de diseño es necesario realizar una primera reunión donde se hablará sobre las ideas que se tengan para el juego, los objetivos que se esperan alcanzar con el mismo, generar una lluvia de ideas inicial y conocer las propuestas generales de cada uno de los integrantes del equipo.

Las etapas que se seguirán para la metodología son:

1. Identificar tipo de mecánicas acorde al tipo de juego
2. Determinar los tipos de jugadores objetivo
3. Diseño de las mecánicas

3.1. Planteamiento de mecánicas centrales

3.2. Planteamiento de mecánicas secundarias

3.3. Identificar los recursos y entidades para las mecánicas planteadas

4. Prototipado

5. Selección de patrones de diseño de software

6. Programación

7. Testeo y correcciones

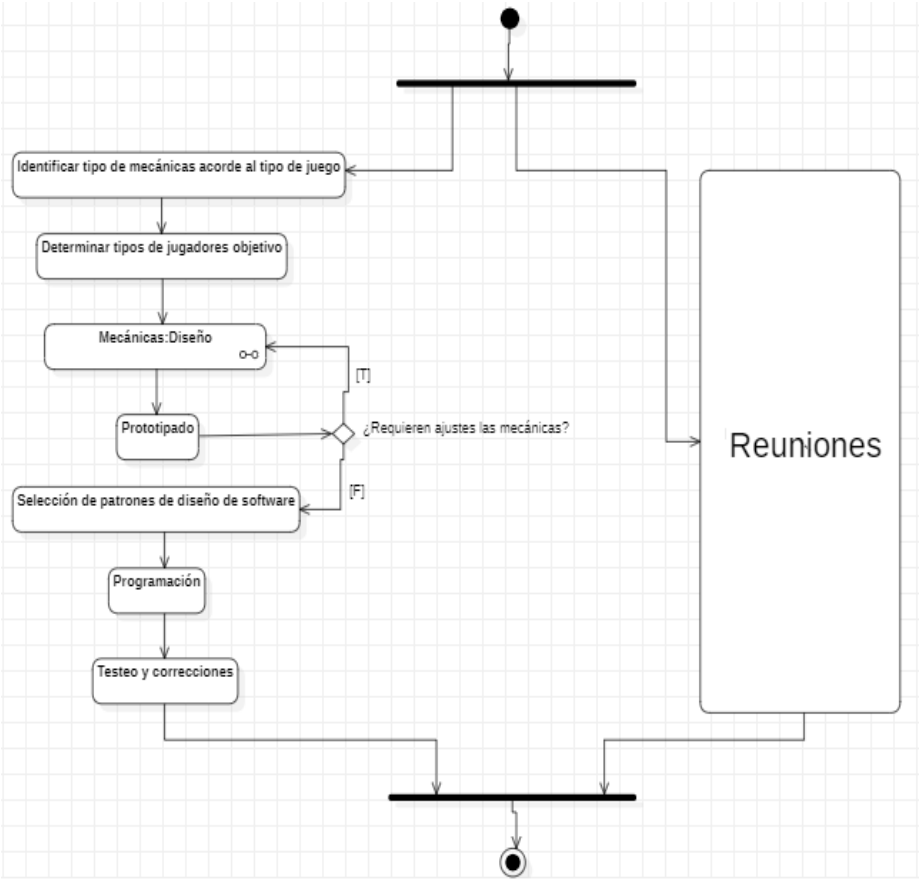


Figura 5.1: Pasos metodología

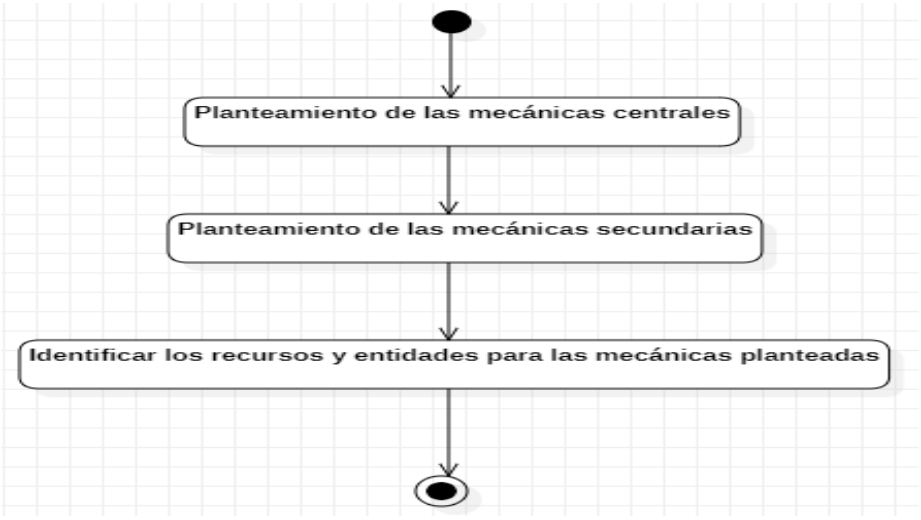


Figura 5.2: Subactividad: Diseño de las mecánicas

Además de la reunión inicial se recomienda seguir un sistema similar al de los "sprints" de la metodología Scrum para realizar reuniones constantes a lo largo del proyecto.

Las reuniones deberán realizarse al inicio de cada semana y servirán para verificar los avances realizados por cada miembro del equipo, asignar nuevas tareas junto con los tiempos esperados para concluirlas y verificar que no haya dudas sobre los pasos a seguir hasta la siguiente reunión, ya que, es vital para el desarrollo de las mecánicas que todos los integrantes estén bien sincronizados en lo que se realizará. Una hora de reunión puede llegar a sonar tediosa pero puede ahorrar varias horas o incluso días de trabajo corrigiendo aspectos que no quedaron claros para todos los integrantes.

Para la asignación de las tareas se recomienda que sean repartidas en conjunto entre el jefe del equipo y los miembros, esto en base a las tareas realizadas anteriormente y la experiencia de cada uno de los miembros. Tomar en cuenta estos dos aspectos puede evitar alargar de más algunas tarea,s ya sea porque sea algo demasiado complicado para uno de los miembros del equipo o porque haya sido asignado a una parte del proyecto con la que no esté familiarizado del todo.

Además de las reuniones semanales se recomienda realizar reuniones entre cada una de las etapas y subetapas para comentar las dudas que se tengan, comentar

ideas y planear lo que se realizará en la etapa correspondiente.
En recomendable que el jefe del equipo haga un resumen de las conclusiones a las que se llegó al final de cada reunión para que así los integrantes se puedan dar cuenta en caso de que no hayan comprendido del todo cuales son los acuerdos comunes a los que se llegó durante la reunión.

5.1. Identificar tipo de mecánicas acorde al tipo de juego

Antes de comenzar a diseñar las mecánicas de nuestro juego, debemos tener bastante claro el género del juego en el que se está trabajando y las posibles habilidades que la historia del mismo le pueda dar a nuestro protagonista. Por ejemplo, en caso de estar creando un juego que trate la historia de una joven atrapada dentro de un edificio junto con un fantasma o demonio que la acecha, no tendría demasiado sentido planificar habilidades para que la chica lance poderes (a menos que así sea requerido por la trama). En este caso, sería más conveniente pensar en mecánicas del tipo esconderse haciendo uso de los entornos, causar ruidos capaces de alertar intencional o accidentalmente a la criatura, acertijos o puzzles que requieran ser resueltos para avanzar, obstáculos que requieran de conseguir algún objeto, entre otros. En resumen, debemos pensar en mecánicas que sean congruentes con el juego que estamos trabajando, a la par de que, permitan divertir y plantearle un reto a los jugadores.
Otra de las razones por las que es importante identificar el tipo de juego en el que estamos trabajando es que esto nos ayudará a saber cuáles mecánicas deben ser priorizadas. Por ejemplo, si estamos haciendo un juego de disparos que tiene ciertas secciones en las que se maneja uno vehículo, aunque las mecánicas de manejo deben estar bien implementadas y pulidas, la prioridad debe de ser las mecánicas de disparo, cobertura, lanzar granadas, movilidad del personaje, etc. Esto debido a que, en este caso, un buen manejo será un punto a tomar en cuenta, pero la prioridad de los jugadores que buscan un juego de disparos son las mecánicas de combate con las armas, mientras que, el manejo de ese vehículo

será un factor secundario.
En esta primera etapa todavía no es necesario definir todos los detalles de las mecánicas, el propósito de esta etapa es tener una idea general de las mecánicas que serán requeridas para desglosar las ideas y especificar los detalles en una de las etapas posteriores.

5.2. Determinar los tipos de jugadores objetivo

Retomando los tipos de jugadores mencionados en la taxonomía de Bartle, es fundamental saber a que tipo de jugadores irá dirigido nuestro juego, ya que, si no conocemos nuestros jugadores objetivo, difícilmente podremos brindarles los elementos que buscan dentro de un videojuego.
En caso de que el juego no contenga elementos multijugador, es más sencillo determinar esto, ya que, podríamos descartar directamente las partes de socializar o eliminar a otros jugadores, pero, en caso de contar con elementos multijugador, tendremos que considerar todas las combinaciones posibles de jugadores.
Es importante recalcar que el juego puede ser diseñado para unir a los cuatro tipos de jugadores anteriormente mencionados. Un ejemplo de este puede ser el juego *Sea of Thieves* (Figura 5.3), en el cual, los jugadores simulan la vida pirata al navegar barcos, explorar islas, buscar tesoros y enfrentarse a otras tripulaciones. Las mecánicas y el diseño del mundo le permiten a los Triunfadores buscar los distintos coleccionables, vencer las misiones de temporada y conseguir todos los cosméticos para su pirata y el barco, los Exploradores pueden navegar y descubrir cada una de las islas junto con los secretos que esconden, los Socializadores pueden reunirse con varios amigos para hacer más fácil la tarea de controlar el barco, cumplir con las tares anteriormente mencionadas y defenderse de los jugadores Asesinos que busquen hundir sus barcos para así quedarse con sus tesoros.

Figura 5.3: Sea of Thieves

Finalmente, debemos determinar el nivel de simulación de nuestras mecánicas tomando en cuenta el público objetivo. Por ejemplo, tomamos dos juegos de béisbol, el juego de béisbol dentro de Wii Sports (Figura 5.4a) y MLB The Show (Figura 5.4b).

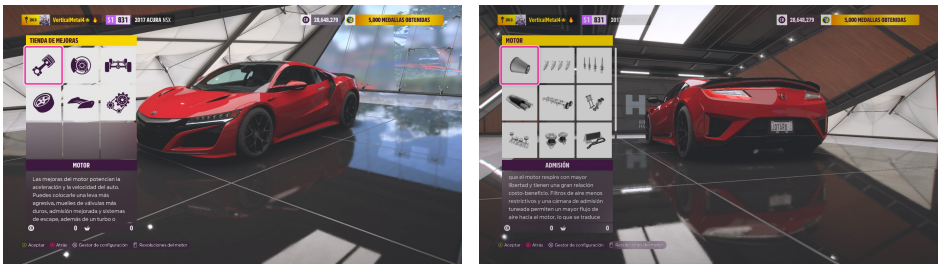
Desde la perspectiva visual, podemos notar como Wii Sports intenta ser un juego menos realista y, si jugamos ambos juegos, podemos notar cómo es mucho menos complicado de jugar que MLB The Show. Esto no es casualidad ni es porque Nintendo haga un esfuerzo menor, Wii Sports logra ser un juego para divertirte con familiares y amigos incluso si no están familiarizados con este u otro videojuego, este está muy bien diseñado para que sea intuitivo y cualquiera con el conocimiento básico de béisbol pueda pasar un buen rato jugando. Por otro lado, MLB The Show intenta simular de mejor manera lo que es jugar y dirigir un partido de béisbol como en la vida real, esto es porque su objetivo son los jugadores más amantes del deporte que quieren dedicarle muchas más horas para poder dominarlo y planear cada detalle que sea posible.

No hay una perspectiva mejor que la otra ni uno es mejor que el otro por estas razones, aunque ambos tratan de un mismo deporte tiene un público diferente, cada uno busca satisfacer distintas necesidades que puedan tener los jugadores y ese es otro de los factores que son vitales para tomar en cuenta a la hora de diseñar nuestro videojuego.

(a) Wii Sports (b) MLB The Show

Figura 5.4

Otro ejemplo de esta situación podemos encontrarlo en el nivel de personalización de vehículos dentro de dos juegos de carreras, Forza Horizon 5 (Figura 5.5) y Mario Kart 8 (Figura 5.6). Aunque ambos agregan la posibilidad para que los jugadores cambien piezas de sus vehículos para así cambiar aspectos de la conducción de los mismos, en Mario Kart se muestra un sistema más ligero en el que el jugador cambia el cuerpo del vehículo, las llantas y el paracaídas para así cambiar aspectos de la conducción, mientras que, por otro lado, en Forza Horizon 5 podemos observar un sistema de personalización en el que jugador puede cambiar varias piezas del motor, de la plataforma de manejo, del tren de transmisión, las llantas y sus rines, entre otros muchos aspectos que requieren de un conocimiento mucho mayor para poder conseguir una buena configuración para la conducción del vehículo (aunque también se da la posibilidad de usar los tuneos de otros jugadores).



(a) Menú principal de personalización (b) Personalización del motor

Figura 5.5

Figura 5.6: Personalización en Mario Kart 8

5.2.1. Delimitar los tipos de jugadores

Si tomamos todos estos factores en cuenta, podemos llegar a la conclusión de que será necesarios dejar claros los siguientes aspectos:

- ¿Mi juego contará con elementos multijugador?
- En caso de tener elementos multijugador, ¿los jugadores trabajarán juntos, se enfrentarán entre si o ambas?
- ¿A cuáles de los cuatro tipos de jugadores quiero atraer?
- ¿Qué tipo de mecánicas debería de implementar para atraerlos?
- ¿A qué nivel de realismo quiero llegar con las mecánicas?

Realizarse esas preguntas pueden permitir al equipo entender mejor el tipo de jugadores al que esperan llegar con su juego e identificar los elementos en los que deberán de enforcarse a lo largo del desarrollo.

5.3. Diseño de las mecánicas

5.3.1. Planteamiento de las mecánicas centrales

Una vez que tenemos claros el juego y el público al que irá dirigido nuestro juego, podremos comenzar a diseñar las que serán nuestras mecánicas centrales.

Unos aspectos muy importantes a tomar en cuenta a la hora de pensar en las mecánicas de nuestro juego son:

- **Deben ser entretenidas e interesantes:** Dependiendo del juego, podemos buscar la diversión del jugador, causar terror, intriga y demás sentimientos posibles, es por esto que, al diseñar nuestras mecánicas es fundamental que estas consigan despertar la atención y el interés por parte del jugador para que así incentivar que siga jugando.

- **Más no es mejor:** Un juego no es mejor en base a la cantidad de mecánicas que son implementadas, todas las mecánicas deben de cumplir con un objetivo dentro del funcionamiento del juego y deben jugar un papel a la hora mejorar la experiencia del jugador. Las mecánicas que no aportan nada relevante al juego, no sirven para mantener despierto el interés del jugador o dejan de tener un valor a medida que avanza el juego solo causarán desagrado y la sensación de que se está perdiendo el tiempo, además de que, a la hora de desarrollar, también serán tiempo y recursos desperdiciados y que pudieron ser mejor utilizados al pulir las mecánicas que son relevantes.

- **Deben ser polivalentes:** Las mecánicas que se diseñen deben de poner evolucionar para combinarse con distintos elementos dentro de los escenarios, deben poder combinarse con otras mecánicas e inclusive realizar modificaciones dentro de la misma para enriquecer la jugabilidad. Si nuestras mecánicas no pueden tener estas evoluciones, fácilmente se volverán repetitivas y aburridas para el jugador.

- **No todas deben aplicarse en todo momento:** No todas las mecánicas deben ser indispensables para todo el juego, algunas de las mecánicas deben poderse pausar durante determinados momentos para evitar que el jugador se canse de ellas y se vean agregadas "a la fuerza". Variar dentro el uso de las mecánicas será fundamental para mantener al jugador interesado en el juego.

Para definir las mecánicas centrales es recomendable generar una tabla en la que se pueda enlistar cada mecánica identificando los tipos de mecánica que

es, su función dentro del juego y los valores iniciales para las posibles variables relacionadas. Esto ayudará a que se tenga un registro que podrá ser revisado por los miembros del equipo y a la vez ayudará a identificar si es necesario agregar o eliminar alguna mecánica en base a las ya existentes.

5.3.2. Planteamiento de las mecánicas secundarias

Como fue mencionado antes, las mecánicas secundarias son aquellas que acompañaran y servirán de apoyo a las mecánicas secundarias a la hora de realizar determinadas acciones y para enriquecer la jugabilidad del jugador al agregar mejoras o desafíos para una determinada zona.
Un ejemplo de esto puede ser al implementar un sistema de miras para las armas del juego, estas brindarán un componente extra de estrategia y facilitará al jugador apuntar a los enemigos que se encuentran en una zona más alejada dentro del mapa; esto permitirá enriquecer más la experiencia pero no será fundamental para que el jugador progrese dentro del juego y no será necesario usarlas durante todo el tiempo de juego (lo que permite que no se vuelva monótono y tedioso).
Cabe recalcar la advertencia de no cruzar la línea entre una mecánica secundaria y la de *gameplay periférico*, recordemos que al crear *gameplay* periférico se puede consumir una gran cantidad de tiempo y de recursos en dejar las mecánicas de esta parte del juego muy pulidas, lo que, posteriormente, puede costar que no se cuente con el tiempo y los recursos suficientes para dejar las mecánicas centrales y secundarias lo más libres de fallas posible. Para esta etapa de la metodología también es recomendable realizar una tabla en donde se puedan enlistar las mecánicas secundarias propuestas, además de que se escriba con cuales mecánicas centrales se encuentran relacionadas, su función y los valores iniciales para las posibles variables relacionadas.

5.3.3. Identificar los recursos y entidades para las mecánicas planteadas

Una vez que se hayan identificado las mecánicas centrales y secundarias que estarán presentes en nuestro juego, será necesario plantear los recursos y entidades con los que trabajarán las mecánicas, la forma en que estas se relacionarán entre sí y su papel que jugará cada uno dentro de las mismas.
En caso de que nuestras mecánicas hagan uso de recursos, también será necesario plantear la forma en que será posible conseguir más de estos recursos, la forma en que se consumirán, si será posible comprarlos, craftearlos, recolectarlos o intercambiarlos por algún otro recurso dentro de una tienda o incluso con algún otro jugador.
Es vital para el desarrollo del juego identificar todos estos elementos antes de comenzar con el proceso de programación, esto debido a que nos permitirá tener en mente desde un comienzo la cantidad de clases que necesitaremos, la forma en que serán creadas, las tareas que deberán ser realizadas por cada una de ellas y la forma en que se tendrán que relacionar.
Conocer todos estos elementos permitirán seleccionar de una forma más adecuada los patrones de diseño durante la etapa de selección dentro de los mismos, además de que, disminuirán la probabilidad de tener que hacer una reestructuración del código en etapas más avanzadas del código debido a que se deban hacer cambios al implementar alguna función o clase adicional para el funcionamiento.
Para organizar estos elementos es recomendable realizar una tabla en la que se enlisten cada uno de los elementos, las mecánicas con las que se relacionan, la formas en que se obtendrán o perderán y si es una entidad y/o un recurso.

5.4. Prototipado

Antes de comenzar con la fase de programación, es necesario hacer prototipos de las mecánicas de nuestro juego, estos prototipos permitirán tener versiones simples de las mismas con las que se podrá detectar errores en el diseño, errores que pueden ir desde formas de romper la economía interna del juego, formas de

hacer que los jugadores se queden estancados, errores en la progresión o que las mecánicas no sean lo suficientemente interesantes.

Algunas de las formas más comunes de hacer prototipos de las mecánicas de juego es por medio de un juego de cartas, un tablero estilo juego de mesa o por medio de Machinations (se recomienda revisar el capítulo en el que se explica su funcionamiento). Dentro de los prototipos realizados con tableros o con cartas, no es necesario hacerlos demasiado elaborados, ya que, solamente serán usados por el equipo interno y no saldrán al público (por lo que hacerlos demasiado vistosos puede ser una perdida de tiempo), es mejor dedicar el tiempo en implementar correctamente las mecánicas dentro del material para que el testeo sea adecuado.

Al estar en esta fase de testeo, el juego deberá ser llevado a cabo por los miembros más experimentados en el testeo de mecánicas y que tengan la mayor capacidad para detectar errores dentro de las mismas, que sean capaces de detectar cómo se podría romper la economía interna del juego, cómo algunos jugadores podrían obtener una ventaja injusta sobre otros jugadores, entre otros elementos que puedan romper con la experiencia. Esto debe ser revisado muy a detalle debido a que, como dijo una vez Joshua Eric Sawyer (conocido por haber trabajado en juegos como Fallout New Vegas, Pillas of Eternity, Pentiment, entre otros juegos), 'Una buena práctica para el diseño de cualquier juego multijugador es asumir que cada jugador es el tramposo más laborioso, el estafador menos escrupuloso y el sádico más malicioso en uno sola persona' [Saw21]. Este consejo también lo recomiendo no solo para juegos multijugador, ya que, los jugadores muchas veces tratarán de encontrar cualquier debilidad dentro del diseño del juego, la explotarán lo máximo posible para su beneficio y, probablemente, se quejarán de este aspecto una vez terminado el juego.

En caso de que se haya encontrado errores durante el testo, se tendrá que realizar los cambios pertinentes para corregir los mismos y se repetirá el proceso de testo y correcciones hasta que podamos decir que las mecánicas son lo más sólidas posibles. Podremos decir que este objetivo se cumplió una vez que ya no encontremos(o sea demasiado difícil de provocar) formas de que el jugador pueda abusar de algún aspecto dentro de las mecánicas para obtener un beneficio o arruinar la experiencia de otros jugadores, ya no puedan causarse situaciones en

las que se rompa la economía del juego o el progreso se vuelva demasiado lento, se cause que algún recurso o mecánica se vuelva inútil y demás situaciones que le dejen un mal sabor de boca a los jugadores.

5.5. Selección de patrones de diseño de software

Antes de comenzar con la etapa de programación de nuestro juego, es debe pensar en los patrones de diseño de software que se implementarán y será necesario que el lector haya revisado el apartado en el que se explican las clasificaciones de los patrones de diseño de software, en que se diferencian estas clasificaciones, los distintos patrones que se incluyeron en cada una de estas y la función de cada uno de estos.
Una vez que se tiene claro estos aspectos sobre los patrones de diseño de software, se tendrá que analizar cuáles podrían ser aplicados dentro del código y si es necesario aplicarlos, ya que, en caso de no ser necesarios o en caso de ser aplicados de una manera incorrecta será contraproducente para nuestro código y el rendimiento del juego.
Para evitar que esto suceda, se recomienda realizarse las siguientes preguntas:

- **¿Qué es lo que necesito hacer?:** Para resolver esta pregunta será necesario analizar específicamente que es lo que será requerido programar para determinado elemento dentro del juego, esto incluye las clases que serán requeridas, la interacción que habrá entre ellas, la forma en que serán creadas, los resultados esperados y la interacción que tendrá con otros elementos del juego.

- **¿Hay un patrón que me permita realizarlo?:** Para determinar los patrones que podrían ser de utilidad se recomienda revisar dentro de las clasificaciones de los patrones y leer el funcionamiento de cada uno para así verificar cuál corresponde mejor con lo que es requerido.

- **¿Es necesario la inclusión del patrón?:** Como fue mencionado anteriormente, no por el hecho de que un patrón coincida con alguna de las funciones requeridas es fundamental usarlo (o aplicarlo en todo momento,

como puede ser el caso del patrón Update). Es por esto que, es fundamental analizar si el patrón tendrá una funcionalidad el tiempo suficiente para poder comenzar a aplicarlo.

- **¿Cómo podría aplicarlo dentro de mi código?:** La forma exacta en la que se aplicará va a variar dependiendo del programa que estemos realizando, debemos recordar que los patrones no son funciones que puedan ser llamadas dentro del código directamente, son sugerencias de cómo resolver determinados problemas que son comunes. Se recomienda revisar ejemplos sobre algunas maneras de implementarlos antes de hacerlo directamente en el código, tomando en cuenta que la forma de aplicarlos no será exactamente igual y tendrán que ser comprendidos y adaptados respectivamente.

Realizar entre los miembros del equipo estas preguntas puede reducir el tiempo que tomará programar algunas de las mecánicas; como caso particular, cuando comencé a programar junto con mis compañeros de equipo el videojuego "Ichiban Bottles of Redemption" para la clase de Desarrollo de Videojuegos, no sabía de la existencia de los patrones de diseño de software. A medida que avanzamos con el desarrollo nos topamos con el problema de que llegó un punto en que nuestro protagonista podía hacer varias acciones pero no sabíamos cómo hacer la transición entre cada una de estas y poder cambiar las animaciones junto con los objetos que aparecerían en pantalla. De haber conocido el patrón de diseño State, podríamos haberlo aplicado desde un principio y de una forma más óptima para reducir el tiempo de desarrollo en esa parte del juego.
Para esta etapa de la metodología se recomienda realizar una tabla en la que se mencione los patrones de diseño de software que se planea utilizar dentro del juego y la función que tendrá dentro del mismo, esto con el objetivo de tener claro los patrones que nos podrán ser útiles antes de empezar a programar y así evitar cambios ya que esté avanzado el código.

5.6. Programación

Antes de comenzar con esta fase es necesario que en la parte de diseño del juego y las mecánicas se hayan cumplido las tres condiciones siguientes:

- Todas las mecánicas que se utilizarán ya han sido planteadas y testeadas.
- Las mecánicas ya son lo suficientemente robustas para todos los casos (pudiendo exceptuar solamente algunos bastante particulares, difíciles de generar y que, en todo caso, no arruinarían la experiencia de juego).
- Se ha definido las clases que se utilizarán, su función que tendrá cada una y los patrones de diseño que será aplicados.

Es casi inevitable que a lo largo de la codificación surjan nuevos retos que no estuvieran contemplados, pero, al asegurar que estos tres aspectos están cubiertos, podemos disminuir la probabilidad de que ocurran ese tipo de imprevistos que provoquen cambios grandes dentro del código y que aumenten el tiempo de desarrollo.
Como fue estudiado en la sección de las metodologías existentes, es de gran utilidad tener una buena planeación antes de comenzar a realizar el código y saltarse los pasos anteriormente mencionados puede causar una gran desorganización y alargar tanto los tiempos como los costos de desarrollo.

5.7. Testeo y correcciones

Como fue mencionado anteriormente, es casi inevitable que aparezcan errores y se tengan que hacer correcciones una vez que se pase del papel a la computadora. Es por esto que, una vez que se termine de programar cada mecánica (no hasta el final del juego) se deberá pasar por una etapa de testeo para detectar los errores que puedan surgir al realizar determinadas acciones. Esto se deberá repetir con cada mecánica para verificar que todas interactúen de forma adecuada y con un rendimiento óptimo.
En caso de que se observen bugs, que las mecánicas no estén lo suficientemente pulidas o que el frame rate sea menor a los 30 FPS (o inclusive los 60 FPS dependiendo del proyecto), será necesario identificar la parte del código que genera

este mal funcionamiento para posteriormente hacer los cambios necesarios para mejorar el funcionamiento.

Esta etapa de la metodología junto con la etapa de la programación se repetirá en ciclos dependiendo de la cantidad de mecánicas y los errores que puedan aparecer durante los testeos. En el desarrollo de videojuegos es vital utilizar todo el tiempo que se tenga disponible en dejar todas las mecánicas lo más pulidas posible y con la menor cantidad de errores, esto para asegurar que el jugador tenga la mejor experiencia de juego posible.

Muchas veces, aunque el juego sea muy bueno y ofrezca mecánicas innovadoras, si el rendimiento es malo, puede causar que el interés del jugador se pierda o que su experiencia se vea altamente afectada, lo que puede causar que el jugador no quiera probar futuros trabajos del equipo y que cuente su mala experiencia con otros jugadores.

Capítulo 6

Ejemplo

Se aplicará la metodología propuesta para así poder crear mecánicas dentro de un pequeño juego de acción en el que se buscará que el jugador elimine a los enemigos en un determinado orden para así poder avanzar al siguiente nivel.

6.1. Identificar tipo de mecánicas acorde al tipo de juego

Dado que nuestro juego será un First Person Shooter de acción y estrategia, requeriremos mecánicas relacionadas con el combate. Además de los ataques, también deberá contar con habilidades que le permitan facilitar la tarea de eliminar a los enemigos en el orden adecuado, parte de estas pueden ser algunas trampas que nos permitan detener a los enemigos en una determinada zona para posteriormente ir por esos enemigos en específico.
Ligado a las mecánicas de combate, requeriremos de mecánicas relacionadas con la salud y, por lo tanto, condiciones de victoria y derrota. En el caso de las condiciones de derrota nos encontraremos con que nuestro personaje pierda toda la salud o no pueda derrotar a los enemigos en el orden adecuado, por otro lado, nuestra condición de victoria constará en que el jugador pueda derrotar a los enemigos en el orden adecuado antes de que se termine el tiempo.

6.2. Determinar los tipos de jugadores objetivo

Este juego no contará con elementos multijugador, por lo que, retomando la taxonomía de Bartle podemos descartar de nuestros jugadores objetivo a los asesinos y a los socializadores. Esto nos deja a los tipos de jugadores exploradores y los triunfadores, nuestro juego no contará con un basto mundo por explorar o secretos a buscar dentro del mapa, por lo que también podemos descartar a los jugadores exploradores de nuestros jugadores objetivo.
Finalmente tenemos a los jugadores triunfadores, dentro de este juego se contará con un sistema de puntuación que aumentará en base que el jugador elimine a los enemigos en el orden adecuado, el tiempo que requirió para terminar el nivel y la cantidad de errores que cometió al eliminar enemigos. Se buscará que el tratar de obtener mejores puntuaciones fomente que este tipo de jugadores intenten dominar las mecánicas y planear mejores estrategias para enfrentar los desafíos. Se espera obtener un gameplay adecuado para los jugadores a los que les gustan los juegos FPS con movimientos que no lleguen a tener elementos fantásticos, pero sin llegar al punto del realismo total (disparos afectados por condiciones climáticas, balas con efecto de caída, etcétera)

6.3. Diseño de las mecánicas

6.3.1. Planteamiento de mecánicas centrales

En base a las ideas para las mecánicas que se identificaron dentro de la primera etapa de la metodología se plantearon las siguientes mecánicas centrales para el funcionamiento del juego.

Nombre de la mecánica	**Tipo de mecánica**	**Función**	**Valores**
Caminar	Acciones del jugador	Moverse dentro del mapa	-Velocidad de movimiento: 700cm/s
Saltar	Acciones del jugador	Evadir obstáculos	-Velocidad de salto: 420 cm/s - Escala de gravedad: 1.0
Disparar	Acciones del jugador y economía interna	Reducir la salud de los enemigos con cada disparo	- Daño: 10.0
Colocar trampas	Acciones del jugador y economía interna	Provocar diversos efectos dependiendo del tipo de trampa a los enemigos que tengan contacto con ellas	- Trampas colocadas: 1
Salud del jugador	Economía interna y condición de derrota	Determina la salud con la que cuenta el jugador y al perder toda su salud perderá la partida	- Salud máxima : 100.0
Cantidad de trampas	Economía interna	Determina la cantidad de trampas de cada tipo con las que cuenta el jugador	- Trampas iniciales de cada tipo: 1
Tipos de enemigos	Progresión	Se crearán distintos tipos de enemigos con cantidades de daño y velocidades diferentes para dificultar la victoria del jugador	Enemigos: - Vampire - Werewolf - Zombie

Cuadro 6.1: Mecánicas centrales

Nombre de la mecánica	**Tipo de mecánica**	**Función**	**Valores**
Daño de los enemigos	Economía interna	Determina la cantidad de salud que disminuirá al jugador cada que un enemigo le conecte un ataque	Zombie: 20.0 Vampire: 10.0 Werewolf: 50.0
Salud de los enemigos	Economía interna	Determina la salud con la que cuenta un enemigo y al perderla toda se considera que ha sido derrotado	Zombie: 60.0 Vampire: 100.0 Werewolf: 200.0
Tiempo	Economía interna y condición de derrota	Determina la cantidad de tiempo con la que cuenta el jugador para terminar el nivel. El tiempo sobrante al cumplir el nivel mejorará la puntuación del jugador y, en caso de terminarse, el jugador será derrotado	Tiempo inicial: 3 minutos
Puntuación	Economía interna	Medida de calificación que determina el rendimiento del jugador dentro del nivel. El tiempo el sobrante y la eliminación de enemigos objetivo aumentará la puntuación, mientras que eliminar los enemigos no objetivo la disminuirá.	Aumento de puntos por: - Enemigos objetivo: 100 - Segundos restantes: 10 por segundo Decremento de puntos: - Enemigo no objetivo: 30

Cuadro 6.2: Mecánicas centrales

Nombre de la mecánica	Tipo de mecánica	Función	Valores
Cajas de recursos	Economía interna	Al ser recogida por el jugador habrá una posibilidad de que aumente la cantidad de alguno de sus recursos	- Probabilidad: Todos los recursos tendrán la misma probabilidad de aparecer, que será la la misma probabilidad de que la caja esté vacía.
Enemigos no objetivo	Progresión	Enemigos que se encargarán de entorpecer el avance del jugador al tratar de disminuir su salud y dificultando la eliminación de los enemigos objetivo	- La cantidad de enemigos no objetivo variará en cada nivel, pero casi siempre será mayor que la de los objetivos
Enemigos objetivo	Progresión y condición de victoria	Serán los enemigos que serán el objetivo en ese momento de la partida, al eliminar a todos los enemigos objetivo se vencerá el nivel	- La cantidad de enemigos objetivo variará en cada nivel

Cuadro 6.3: Mecánicas centrales

6.3.2. Planteamiento de mecánicas secundarias

En base a las mecánicas centrales planteadas en la etapa anterior ahora plantearemos las mecánicas secundarias que servirán de apoyo a las mecánicas centrales. Dentro de de este juego se evitará crear *gameplay* periférico y por ende no se plantearán mecánicas que generen este tipo de *gameplay*.
Las mecánicas secundarias a implementar dentro del juego serán las siguientes:

Nombre de la mecánica	**Mecánicas centrales relacionadas**	**Función**	**Valores**
Trampa de retención	Trampas	Detener a los enemigos para poder eliminarlos más adelante o evitar que sigan al jugador	- Trampas colocadas: 1
Trampa explosiva	Trampas	Reduce la salud de los enemigos cercanos.	- Trampas colocadas: 1 - Daño: 100.0
Aumento de salud	Salud del jugador	Probabilidad de recuperar parte de la salud del jugador al recoger una caja de recursos	- Salud recuperada: 20.0
Recolección de trampas de retención	Cantidad de trampas	Probabilidad de aumentar la cantidad de trampas de retención al recoger una caja de recursos	- Trampas recogidas: 1
Recolección de trampas explosivas	Cantidad de trampas	Probabilidad de aumentar la cantidad de trampas explosivas al recoger una caja de recursos	- Trampas recogidas: 1
Aumentar puntuación	Puntuación	Eliminar enemigos objetivo junto con el tiempo sobrante al terminar el nivel aumentarán la puntuación del jugador	- Eliminar enemigo objetivo: 100 - Tiempo de sobra: 10 puntos por segundo
Disminuir puntuación	Puntuación	Eliminar enemigos que no sean el objetivo disminuirán la puntuación del jugador.	- Eliminar enemigo no objetivo: 30 puntos

Cuadro 6.4: Mecánicas secundarias

Mecánica	Mecánicas centrales relacionadas	Función	Valores
Incremento de tiempo	Tiempo	Posibilidad de aumentar la cantidad de tiempo al recoger una caja de recursos	- Tiempo agregado: 15 segundos
Generación de cajas de recursost	Tiempo, salud del jugador y trampas	Al eliminar un enemigo objetivo se generará una caja de recursos. Esta podrá otorgarle al jugador algún recurso o podrá estar vacía	- Cada resultado posible tiene la misma probabilidad de aparecer
Absorción de salud	Tipo de enemigo	Los vampiros recuperarán salud al dañar al jugador	- Robo de salud: 20.0

Cuadro 6.5: Mecánicas secundarias

6.3.3. Identificar los recursos y entidades para las mecánicas planteadas

Los recursos y entidad que se aplicarán dentro del juego en base a las mecánicas previamente planteadas serán los siguientes:

Nombre	Entidad y/o recurso	Tipos de mecánicas de economía interna	Forma de producción o consumo
Caja de recursos	-Entidad: El objeto soltado por los objetivos al morir	-Productor -Consumidor	-Al morir un objetivo aparecerá una caja de recursos -Al entrar en contacto el jugador con la caja esta desaparecerá

Cuadro 6.6: Recursos y entidades

Nombre	**Entidad y/o recurso**	**Tipos de mecánicas de economía interna**	**Forma de producción o consumo**
Trampa explosiva	-Entidad: El objeto soltado por el jugador -Recurso: La cantidad de trampas explosivas con las que cuenta el jugador	-Productora -Consumidora	- Al tomar una caja de recursos habrá una posibilidad de que aumente la cantidad de recursos de trampas explosivas con las que cuenta el jugador - Cada que el jugador coloque una trampa explosiva contará con menos recursos de la misma
Trampa de retención	-Entidad: El objeto soltado por el jugador -Recurso: La cantidad de trampas de retención con las que cuenta el jugador	-Productora -Consumidora	- Al tomar una caja de recursos habrá una posibilidad de que aumente la cantidad de recursos de trampas de retención con las que cuenta el jugador - Cada que el jugador coloque una trampa de retención contará menos recursos de la misma
Objeto de tiempo	- Recurso: La cantidad de salud con la que cuenta el jugador	- Productora - Consumidora	- El tomar una caja de recursos habrá una posibilidad de que aumente la cantidad de tiempo con la que cuenta el jugador - Cada segundo el jugador perderá recursos de tiempo

Cuadro 6.7: Recursos y entidades

Nombre	**Entidad y/o recurso**	**Tipos de mecánicas de economía interna**	**Formas de producción o consumo**
Salud	- Recurso: La cantidad de salud con la que cuenta el jugador	-Productora -Consumidora	- Al tomar una caja de recursos habrá una posibilidad de que el jugador aumente los recursos de salud con los que cuenta - Cada que el jugador reciba daño por parte de los enemigos se disminuirá los recursos de salud con los que cuenta

Cuadro 6.8: Recursos y entidades

6.4. Prototipado

Para prototipar la economía interna dentro del juego se decidió utilizar la herramienta Machinatios. Este prototipo ayudó a mejorar los valores iniciales que se pensaron al comenzar con el mismo, esto debido a que al hacer las simulaciones se pudo identificar que la cantidad de enemigos objetivo era demasiado pequeña a comparación de la salud del jugador (la simulación terminaba en muchas más ocasiones por la eliminación de enemigos objetivo que por la perdida de salud del jugador). Además de ese cambio, se redujo la cantidad de tiempo para la partida ya que nunca se terminaba por la falta de tiempo.
Cabe destacar que para estas simulaciones se han utilizado probabilidades que no representan el nivel de habilidad de todos los jugadores, los valores finales probablemente cambien dependiendo de la habilidad y de la cantidad de jugadores que prueben cada mecánica.

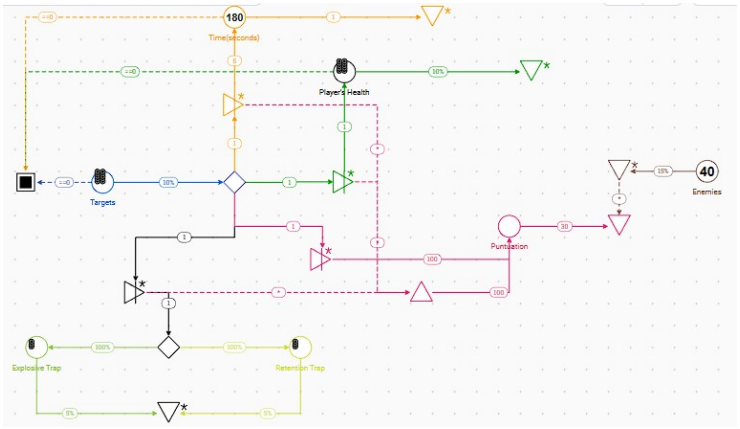


Figura 6.1: Prototipo

Dentro del prototipo se crearon distintas secciones con conectores de un determinado color para representar el recurso que representan, las secciones y sus respectivos recursos son las siguientes:

- Azul: Son los enemigos objetivos que debe eliminar el jugador y al llegar a cero se terminará la simulación. En cada paso de la simulación hay una probabilidad de que uno de estos enemigos sea eliminado y siempre sumarán puntos al hacerlo.

- Naranja: El tiempo con el que cuenta el jugador para superar el nivel, en caso de llegar a cero se terminará la simulación. En cada uno de los pasos de la simulación se drenará "un segundo" del tiempo y cada que se eliminar un enemigo objetivo hay una probabilidad de que aumente el tiempo.

- Verde: Se trata de la salud con la que cuenta el jugador y al igual que con los recursos anteriormente mencionados si llega a ser cero la simulación terminará. En cada paso de la simulación hay una probabilidad de que el jugador reciba daño y se le drene una parte de su salud, por otra parte, cada que un enemigo objetivo es eliminado hay una probabilidad de que el jugador recupere salud.

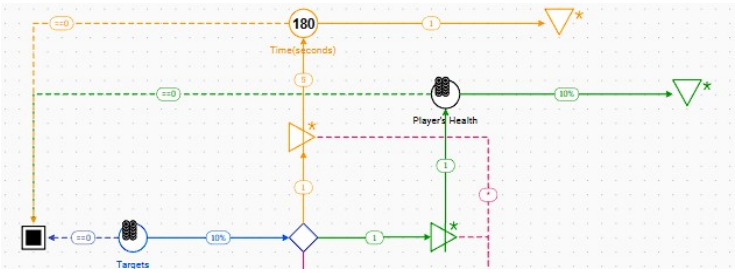


Figura 6.2: Primera sección del prototipo

- Rosa: Representa la puntuación del jugador y será aumentada cada vez que el mismo eliminar a uno de los enemigos objetivo, o por otro lado, en caso de que el jugador elimine a uno de los enemigos comunes está decrementará.

- Café: Se trata de la zona de los enemigos que no son el objetivo. En cada paso de la simulación hay una posibilidad de que se elimine a uno de los enemigos que no son el objetivo y, dado que es algo que debe evitar el jugador, esto causará que se disminuya la puntuación.

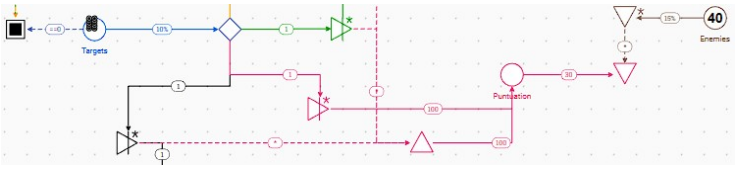


Figura 6.3: Segunda sección del prototipo

- Negro: En caso de que se vayan a generar minas se crea una bifurcación para que haya un cincuenta por ciento de probabilidad para ambos tipos de mina.

- Amarillo: Se trata de las minas de retención y cada que se generan minas hay una probabilidad de que el jugador obtenga más, por otro lado, en cada paso de la simulación hay una probabilidad de que se utilice una de las minas de retención.

- Verde claro: En esta sección de las trampas explosivas hay una posibilidad de que se obtengas más trampas explosivas cuando se generan minas y, al igual que con las minas de retención, en cada paso de la simulación hay una posibilidad de que se disminuya la cantidad de minas de retención.

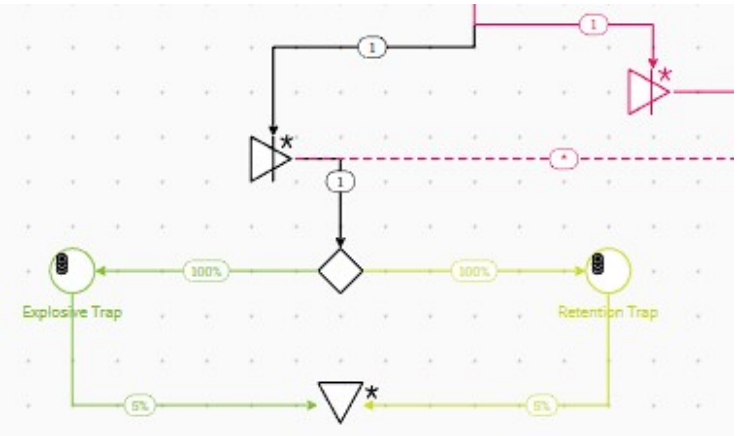


Figura 6.4: Tercera sección del prototipo

6.5. Selección de patrones de diseño

Después de analizar los patrones previamente mencionados se llegó a la conclusión de que los patrones de diseño de software a utilizar dentro del juego serán los siguientes:

Patrón de diseño de software	**Función dentro del juego**
Singleton	Se aplicará en la creación de la clase que llevará el seguimiento de los logros y la clase que verificará si los enemigos eliminados son los objetivos
Command	Será aplicado para llevar el control de los input que le permitirán al jugador control al personaje principal
Game Loop	Se utilizará para mantener el flujo constante del ciclo de inputs del jugador, el procesamiento dentro del juego y las salidas en pantallas en base a los dos pasos anteriores
Observer	Se implementará un sistema de logros dentro del juego para así darle desafíos adicionales al jugador que desee completarlos pero que a la vez no impedirán que los jugadores que no estén interesados en ellos terminen los niveles. Otra de sus aplicaciones será en la clase encargada de verificar si los enemigos eliminados son los correctos
Flyweight	Será utilizado para optimizar la creación de elementos inmóviles dentro del mapa
State	Se aplicará dentro de cada uno de los enemigos para cambiar las animaciones dependiendo de las acciones que estén realizando en cada momento
Update	Permitirá controlar las funciones que realizará tanto el personaje controlado por el jugador como los enemigos y el HUD en cada uno de los frames del juego

Cuadro 6.9: Patrones de diseño de software

6.6. Programación

Para esta parte del caso de estudio se mostrarán partes del código en C++, Blueprint y demás herramientas que forman parte de Unreal para la imple-

mentación de las mecánicas y patrones de diseño de software anteriormente mencionados.

6.6.1. Diagrama UML

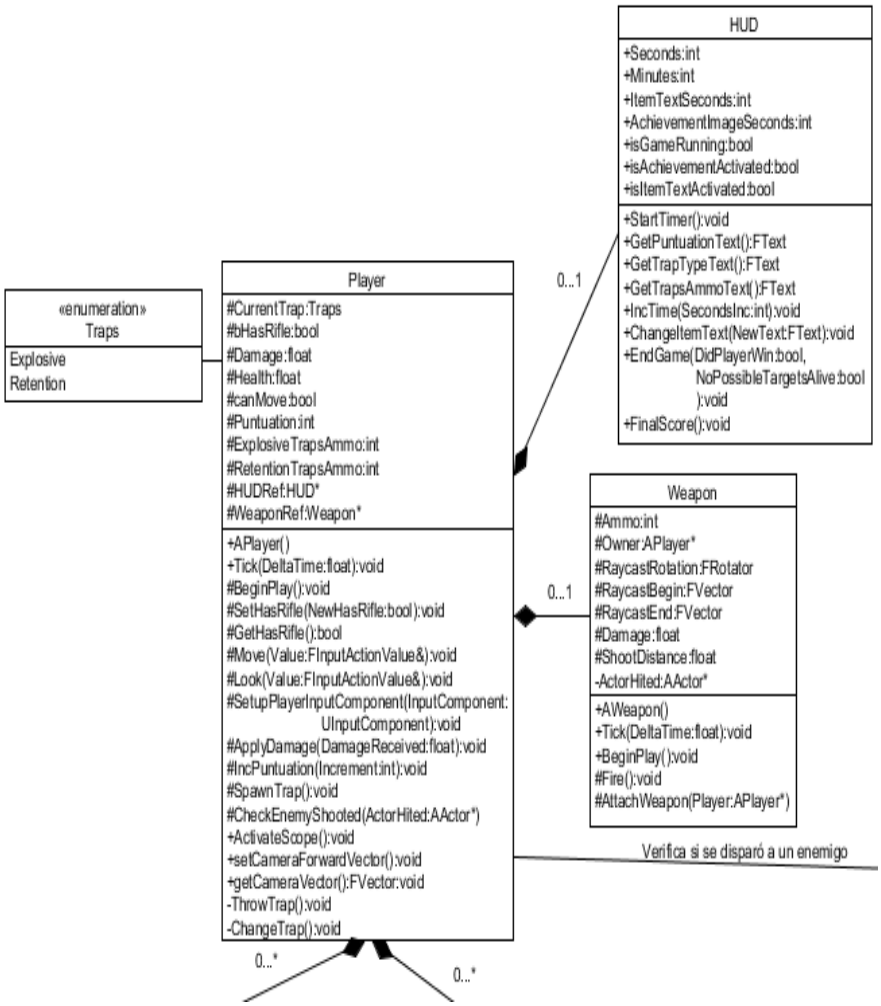


Figura 6.5: Clase del jugador, HUD y arma

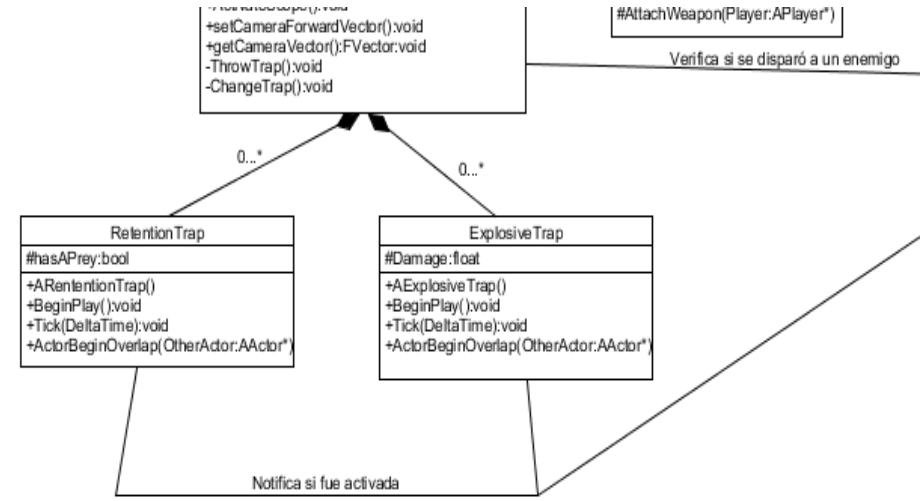


Figura 6.6: Clases para las trampas

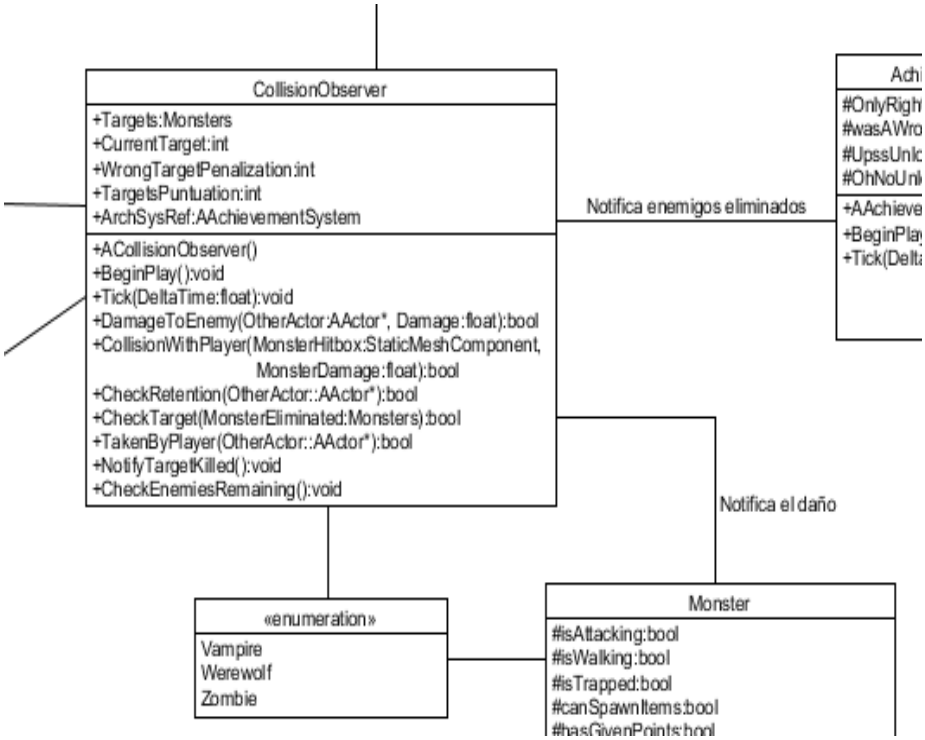


Figura 6.7: Detector de colisiones

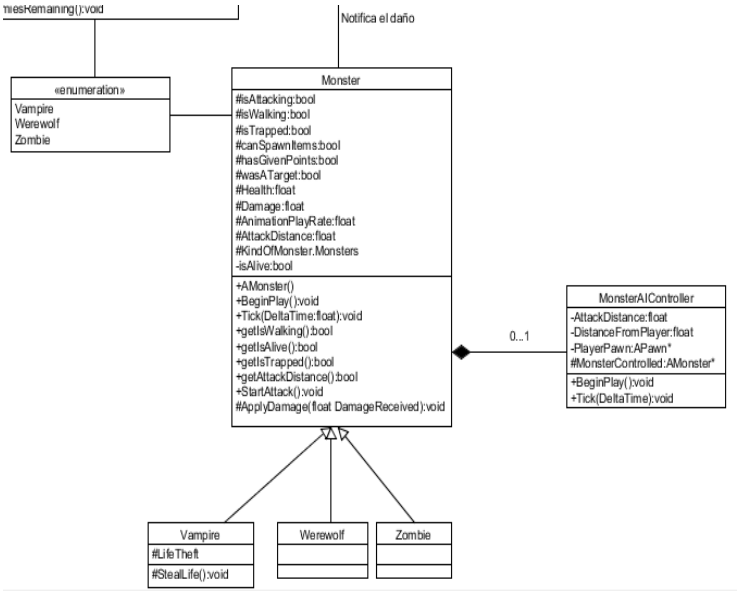


Figura 6.8: Clases de los monstruos

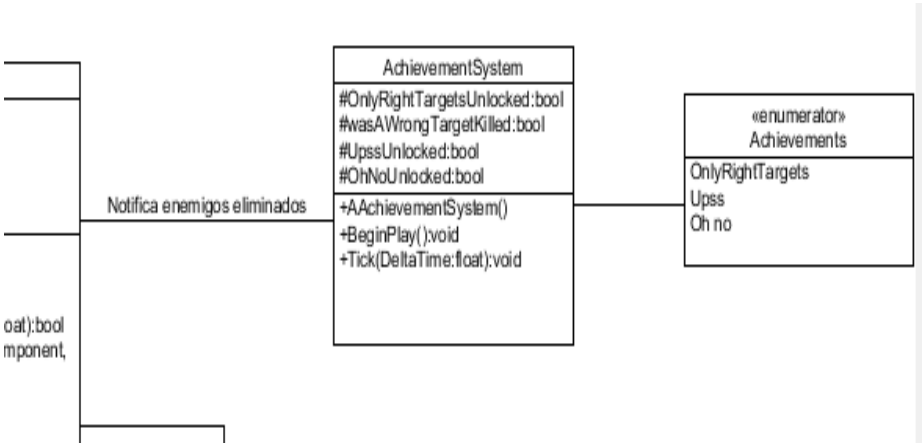


Figura 6.9: Clase para el sistema de logros

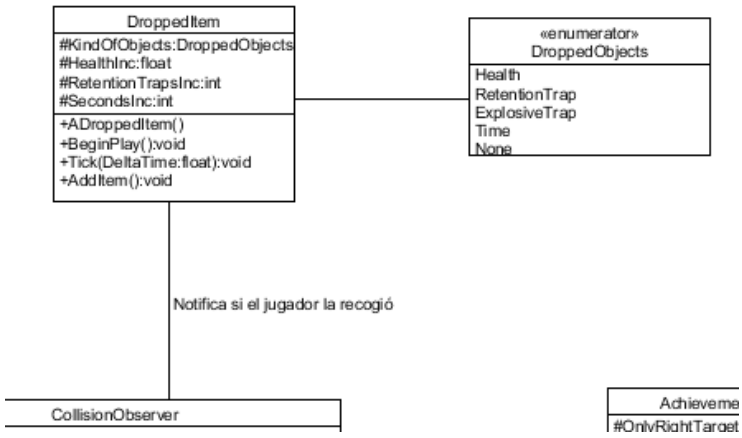


Figura 6.10: Clase para las cajas de objetos

6.6.2. Mecánicas

Trampa explosiva

La trampa explosiva está físicamente compuesta por su mesh [1] y una caja de colisión para detectar si algún actor se superpone con la misma.

[1] Modelo en tres dimensiones que se incorpora dentro de un actor con el objetivo de darle una representación física dentro de los niveles

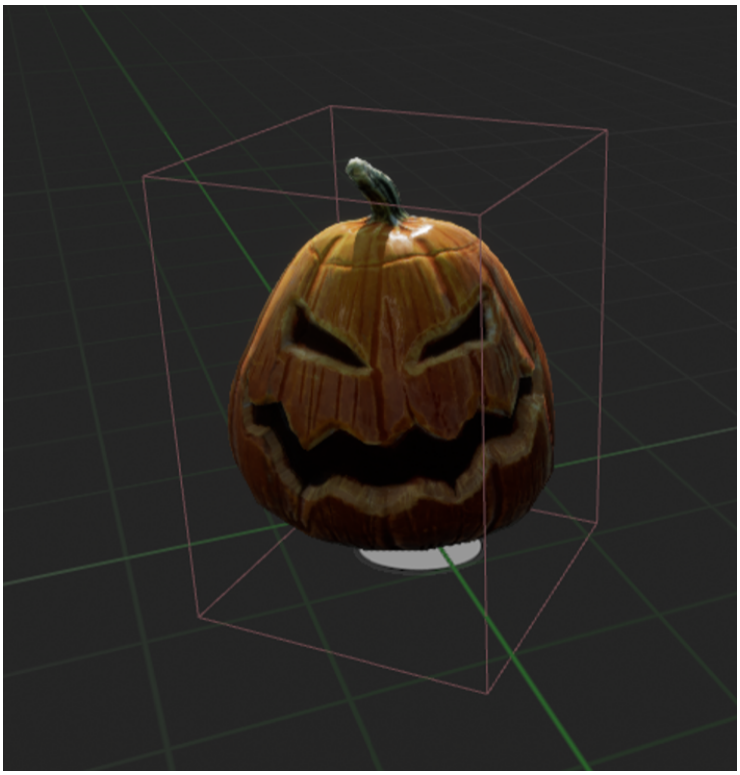


Figura 6.11: Trampa explosiva

En caso de que se cumpla la superposición la trampa accederá al observer de las colisiones por medio del Game State para así hacer uso de la función que le permitirá detectar si el actor que activó el trigger es un monstruo, y en caso de ser así, se active la trampa y se aplique el daño correspondiente. Una vez activada la trampa estará creará el efecto de explosión para posteriormente destruirse.

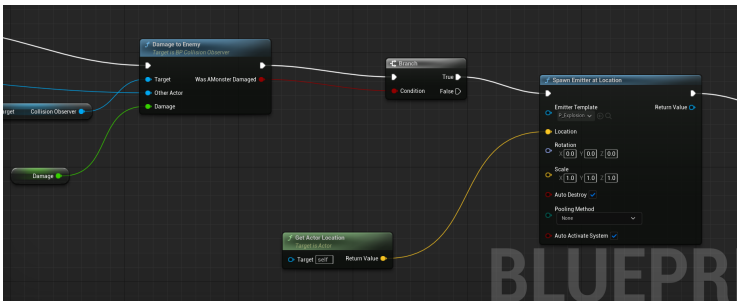


Figura 6.12: Blueprints al activar el trigger de trampa explosiva

Trampa de retención

Físicamente la trampa tiene los mismos elementos que la trampa explosiva pero con un mesh distinto. De forma similar a la anterior trampa, la trampa de retención llamará al detector de colisiones para saber si un monstruo ha activado el evento, en caso de ser así, se determinará que la trampa ya tiene una presa y se creará el el círculo que denotará que la trampa ya se encuentra activa.

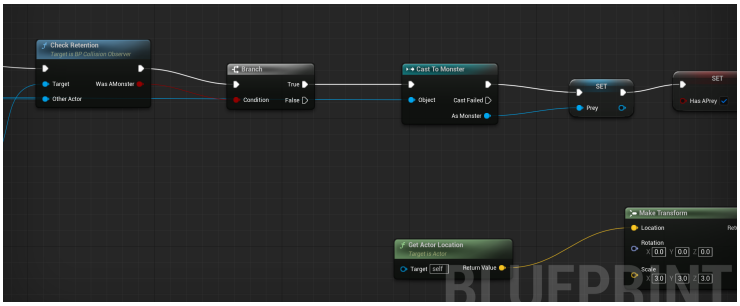


Figura 6.13: Blueprints al activar el trigger de la trampa de retención

En caso de que un monstruo haya activado la trampa su movilidad será anulada y la trampa verificará en cada tick si la salud del monstruo atrapado es mayor a 0, de lo contrario, la trampa destruirá el círculo que la rodea y se destruirá a sí misma.

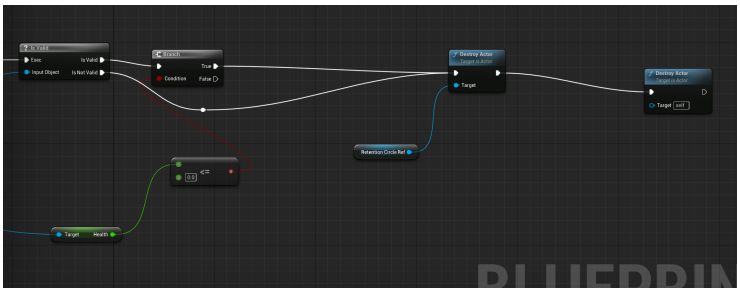


Figura 6.14: Trampa verificando si la presa sigue viva

Figura 6.15: Trampa activada

Caja de recursos

Las cajas de recursos son creadas dentro del mapa cada que un enemigo objetivo es eliminado y estará presente hasta que el jugador la recoja.

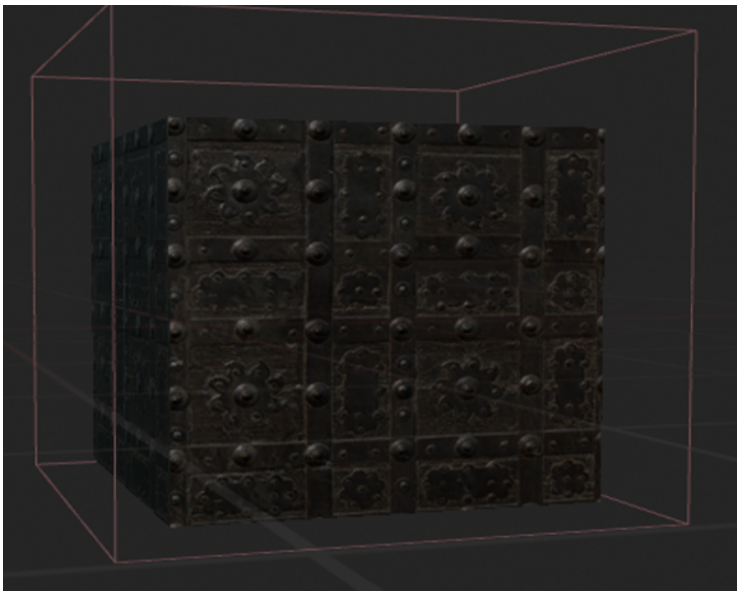


Figura 6.16: Caja de recursos

Para hacer esto posible la caja cuenta con un collider que cuando detecta que alguien ha entrado dentro de él hace uso del Game State para acceder al detector de colisiones y verificar si ha sido el jugador quien activó el trigger. En caso de ser así, la caja llamará a su función AddItem que probablemente le de un beneficio al jugador para posteriormente destruirse.

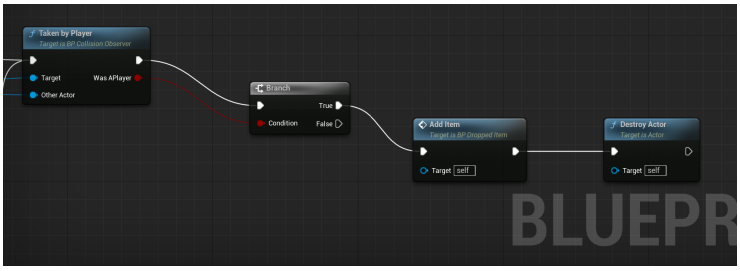


Figura 6.17: Blueprints al activar el trigger de la caja

Dentro de la función AddItem se seleccionará aleatoriamente el contenido de la caja (que puede ser ninguno) para después otorgarle el contenido al jugador.

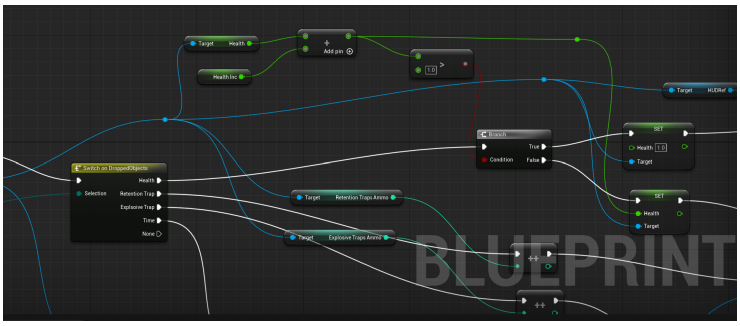


Figura 6.18: Otorgamiento de beneficios

HUD y visualización de recursos

El jugador cuenta con un HUD con el que puede verificar la cantidad de salud con la que cuenta, el tiempo que tiene disponible, el tipo de trampas que tiene equipado en el momento junto con la cantidad de las mismas con las que cuenta, su puntuación actual y el tipo de enemigo objetivo actual.

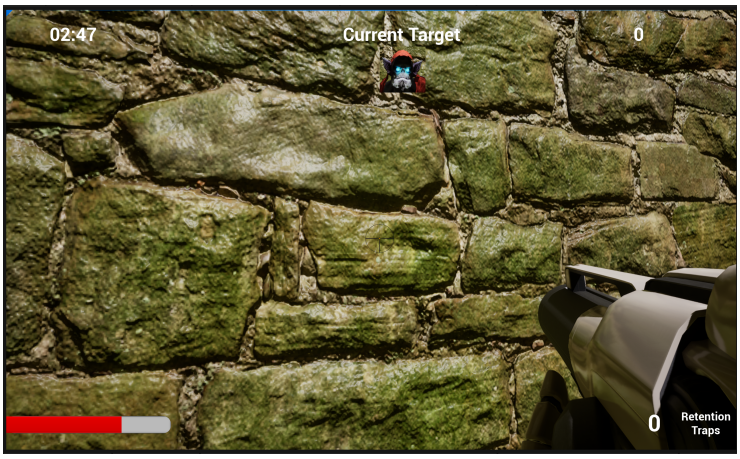


Figura 6.19: HUD

Para hacer esto posible el HUD cuenta con funciones que le permite mantener estos valores actualizados en pantalla por medio de una referencia al jugador y un contador de tiempo interno.

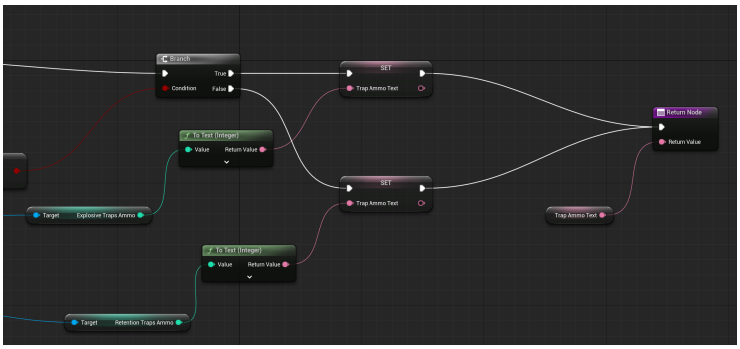


Figura 6.20: Ejemplo de función con la que el HUD obtiene la cantidad trampas

Disparo del arma

Para determinar si el jugador le ha disparado a alguno de los enemigos se utilizan vectores, puntos de impacto y la clase obsever que detecta colisiones. Todo comienza con calculando el inicio del vector por medio de un punto de disparo (el arma del jugador) y el punto final que es calculado haciendo uso del vector directo de la cámara, la distancia que puede viajar el disparo y el origen del disparo.

```
RaycastEnd = SpawnLocation + Character->getCameraVector() *
             ShootDistance;
```

Una vez está calculado el que será el principio y el final del vector de disparo se creará el vector dentro del mundo y detectará si encuentra un punto de impacto. En caso de que se detecte un impacto nuestra clase del jugador hará uso del detector de colisiones para identificar si el punto de impacto ocurrió en un monstruo.

```
if(GetWorld()->LineTraceSingleByChannel(Hit, SpawnLocation,
    RaycastEnd, ECollisionChannel::ECC_GameTraceChannel2))
{
    ActorHited = Hit.GetActorLocation();

    if(ActorHited)
        Character->CheckEnemyShooted(ActorHited);
}
```

Cambio y lanzamiento de trampas

Cada que el jugador pulsa la tecla que le permite cambiar de trampas se verifica cual es el tipo de trampa actual y se cambia la misma.

```
if(canMove)
{
    if(CurrentTrap == Traps::Explosive)
        CurrentTrap = Traps::Retention;
    else
        CurrentTrap = Traps::Explosive;
}
```

Si el jugador pulsa la tecla con la que soltará una trampa, primero se verificará el tipo de trampa, la cantidad de trampas de ese tipo con las que cuenta y, en caso de que tenga suficientes, se llamará a la función que creará la trampa dentro del mapa.

```
if(canMove)
{
    if(CurrentTrap == Traps::Explosive)
    {
        if(ExplosiveTrapsAmmo > 0)
        {
            ExplosiveTrapsAmmo--;
            SpawnTrap();
        }
    }
    else
    {
        if(RetentionTrapsAmmo > 0)
        {
            RetentionTrapsAmmo--;
            SpawnTrap();
        }
    }
}
```

Dentro de la función que es llamada en blueprints se revisará el tipo de trampa que está utilizando el jugador en el momento y se creará un objeto de la clase adecuada dependiendo del tipo de trampa.

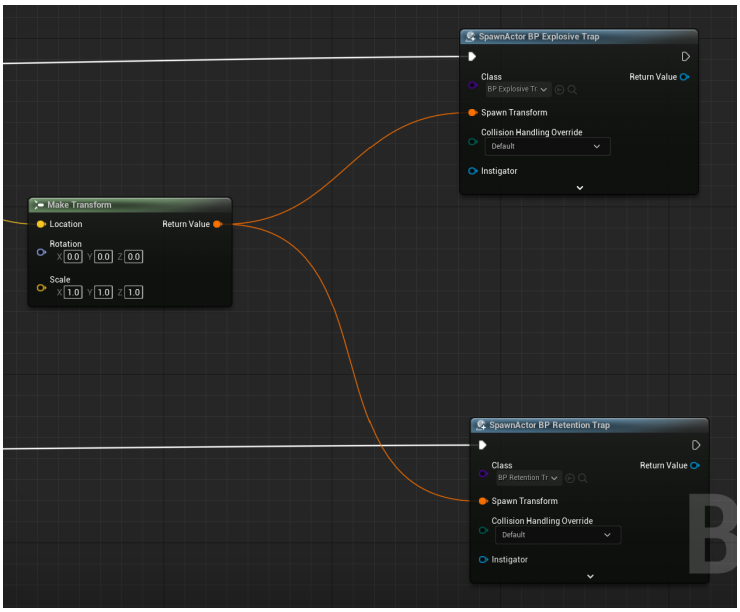


Figura 6.21: Lanzamiento de trampa en blueprints

Daño al jugador

Para poder aplicar daño al jugador este tiene que estar en el rango de ataque de los monstruos, es por esto que, dentro de la clase que se encarga de manejar la IA de los monstruos se revisa si el jugador se encuentra en este rango, en caso de no estarlo seguirá buscando al jugador y, en caso de estarlo, comenzará la animación de ataque del monstruo correspondiente.

```
if(MonsterControlled->GetisAlive() && !MonsterControlled
                                        ->getisTrapped())
{
    if(MonsterControlled->GetisWalking())
    {
        if(MonsterControlled->GetDistanceTo(PlayerPawn) >
                                            AttackDistance)
            MoveToActor(PlayerPawn);
        else
            MonsterControlled->StartAttack();
    }
}
```

Dentro de la animación de ataque de cada monstruo hay dos eventos que se activan durante distintos momentos de la animación, uno con el que se activa la caja que detectará si el jugador recibió el ataque y otro con el que se desactivará la caja al terminar el ataque.

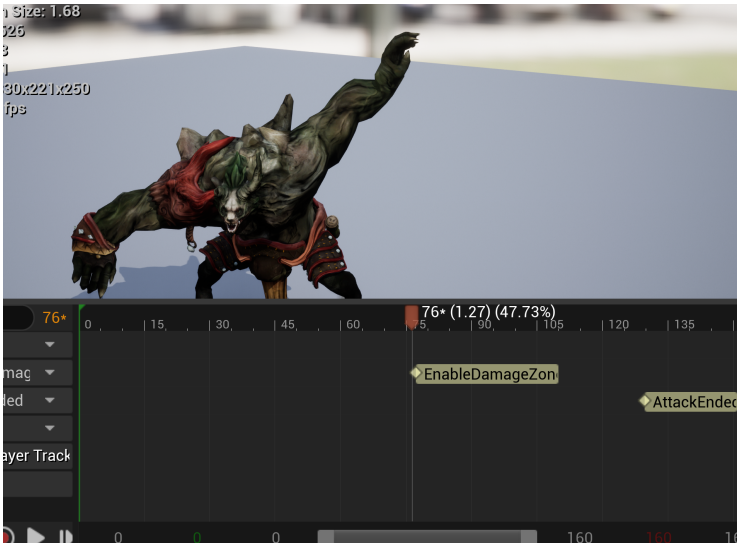


Figura 6.22: Eventos dentro de la animación

Una vez que se activa el evento del comienzo del ataque se obtiene una referencia al detector de colisiones para que verifique si el jugador recibió el ataque y, de ser así, se le aplicará el daño correspondiente. Dentro de la captura se puede ver los Blueprints para el ataque del vampiro, este cuenta con la habilidad especial que además de dañar al jugador este recuperará salud al hacerlo.

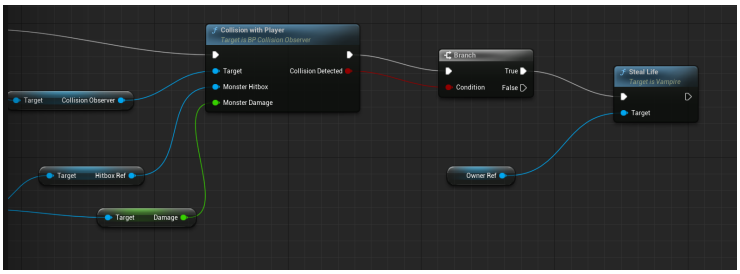


Figura 6.23: Blueprints para el daño

Enemigos objetivo

Cada vez que un enemigo es eliminado se llama a la función para detectar si ese enemigo era el objetivo, en caso de ser así se retornará true para que se le pueda aumentar la puntuación al jugador y, de lo contrario, se le disminuirá la puntuación.

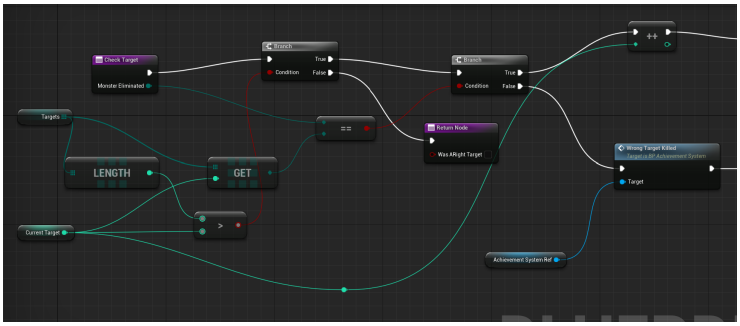


Figura 6.24: Verificación de si el enemigo era un objetivo

Condiciones de victoria y de derrota

Cada que se elimina un enemigo objetivo se llama a la función que verifica si ya se han eliminado todos los objetivos, en caso de ser así se le notifica al Game State y al HUD que el juego ha terminado con una victoria.

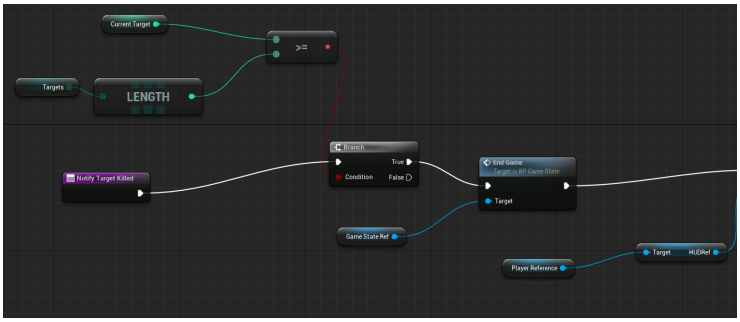


Figura 6.25: Blueprints para verificar si han eliminado a todos los enemigos objetivo

Cada que un enemigo es eliminado se verifica si aún hay enemigos del tipo del objetivo dentro del mapa, en caso de que no se cumpla esa condición el

jugador perderá la partida.

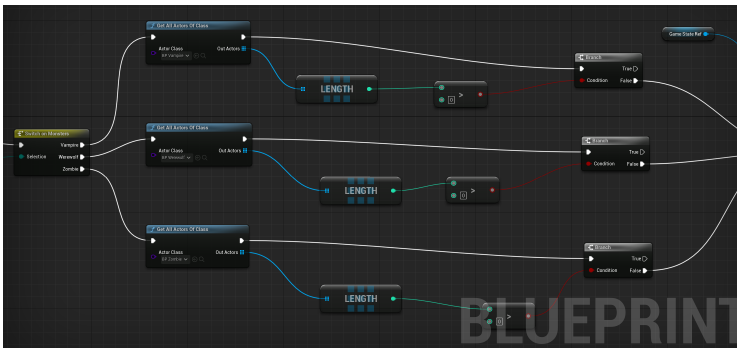


Figura 6.26: Función que verifica si se puede eliminar un enemigo objetivo

En caso de que el tiempo se haya terminado, el jugador ya no cuente con salud o suceda el caso anteriormente mencionado se notificará al Game State y al HUD que el jugador ha perdido.

6.6.3. Patrones de diseño de software

Observer

Dentro de este ejemplo se crearon dos clases observer, una encargada de las colisiones y otra encargada de los logros. Dentro de las clase encargada de verificar las colisiones se encuentran las funciones que se mencionaron anteriormente para detectar si los enemigos golpearon al jugador, si el jugador disparó a algún enemigo, si el jugador recogió una caja, entre otras.

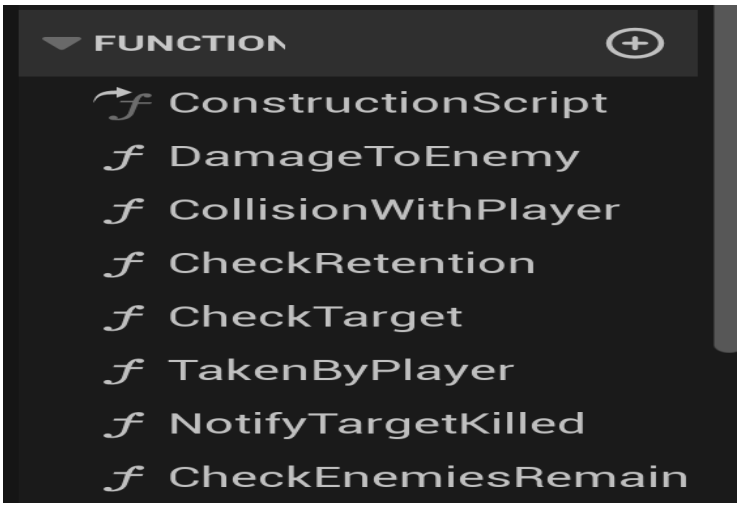


Figura 6.27: Funciones del detector de colisiones

Aplicar este patrón permitió tener funciones mucho más claras al no mezclar tareas dentro de las mismas, un ejemplo de esto es con la función DamageToEnemy, la cual es llamada por clase de la trampa explosiva y el personaje controlado por el jugador; en ambos casos, estas clases solamente reciben la respuesta que requieren para su funcionamiento, la cual es si se ha dañado a un enemigo. Mientras estas clases no tienen que realizar la lógica para saber esta respuesta, nuestra clase observer verifica dentro de su función si el actor que activó el respectivo trigger es un enemigo y si es un enemigo objetivo, lo cual permite que en esa función se notifique al observer del sistema de logros si mató a un enemigo erróneo o no y, a la vez, le puede notificar al enemigo que ha recibido daño. Como podemos observar en la figura 6.28 la función en cuestión es algo grande y el tenerla dentro de una clase aparte nos permite mejorar el código dentro de las clases que hacen uso de esta función, además de que se evita el tener código duplicado.

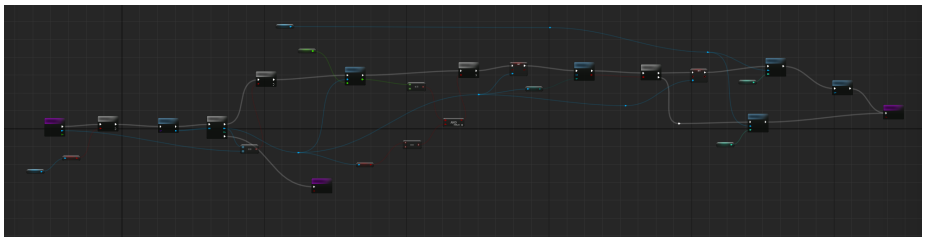


Figura 6.28: Función Damage to Enemy

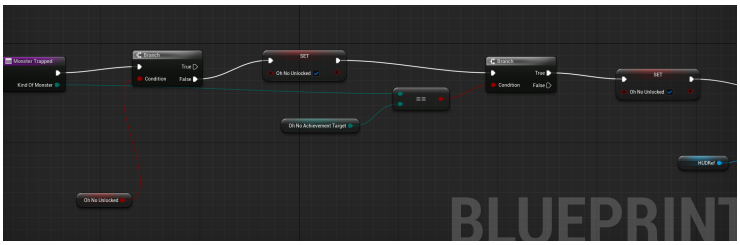


Figura 6.29: Sistema de logros verificando si las condiciones para uno de ellos fue cumplida

Figura 6.30: Logro por ganar una partida sólo matando a los objetivos correctos

Flyweight

Este patrón permitió crear varias instancias de las lápidas que sirven de decoración dentro del mapa, estas lápidas comparten una misma Mesh y las mismas texturas, por lo que eran el objetivo ideal para aplicar el patrón. Para aplicar este patrón se creó una clase que contiene un Hierarchichal Instanced Static Mesh Component y se agregó instancias dentro de un array. Aunque cada instancia comparte elementos en común tienen rotaciones, localizaciones y tamaños diferentes entre cada una de ellas.

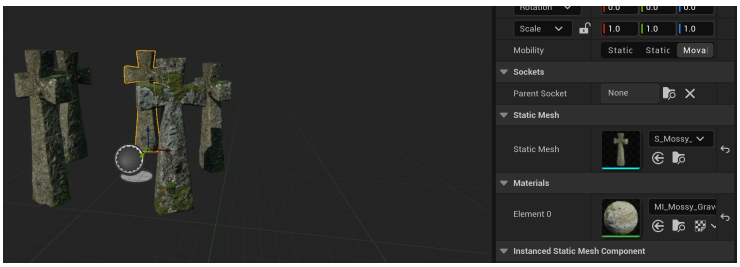


Figura 6.31: Grupo de tumbas

State

Cada uno de los monstruos dentro del juego cuenta con una máquina de estados para cambiar sus animaciones y activar eventos dependiendo de lo que está sucediendo en cada animación.

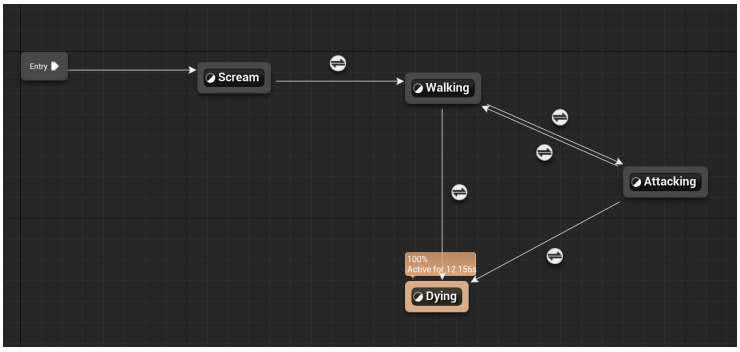


Figura 6.32: Máquina de estados del vampiro

Hacer uso de estas máquinas de estado no solo permitió hacer uso del cambio

de animaciones, también permitió realizar parte de la lógica del comportamiento de los monstruos que debía ser activada en momentos concretos de las animaciones, como por ejemplo, la activación y desactivación de las cajas de daño de los enemigos mencionados anteriormente.

Singleton

Para poder asegurar que no se crearán más instancias de clases que requieren ser únicas, como puede ser el sistema de logros (tener más de una instancia podría causar que un logro apareciera más de una vez durante el juego), se tuvo que agregar un sistema en el constructor de las clases Blueprint para que al crear una nueva clase se verifique si ya hay una instancia existente y, en caso de que sea así, se destruya la nueva instancia.

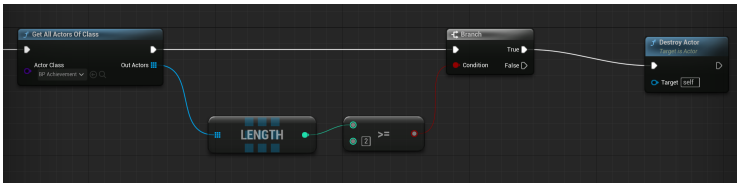


Figura 6.33: Constructor del sistema de logros

Patrones directamente aplicados por el motor

Los siguientes patrones vienen implementados por defecto por el motor, por lo que se requirió realizar poco o ningún cambio para su funcionamiento.

- Command: Dado que se utilizó el sistema de Inputs de Unreal, solamente fue requerido agregar algunos comandos como el uso de la tecla E para soltar una trampa o R para hacer el cambio entre tipos de trampas.
- Game Loop: Es el ciclo que está vigente mientras el juego se encuentra corriendo, este es activado directamente al seleccionar play dentro del editor y termina cada vez que se pulsa Esc o se cierra la ventana generada.
- Update: Cada que se generó alguna de las clases requeridas para el ejemplo estas generaron automáticamente un método Tick (debido a que fueron en su mayoría clases derivadas de la clase Actor o Character), este método

funcionó como Update de cada clase (excepto por algunas en las que por optimización fue desactivado).

6.6.4. Assets usados para el ejemplo

Cabe mencionar que algunos de los elementos para mejorar visualmente esta implementación fueron obtenidos de forma externa, a continuación se mencionan estos elementos junto con los enlaces para obtenerlos.

- Las texturas para el suelo [Meg22d], paredes [Meg22a]y las cajas de recursos [Meg22b] fueron obtenidas gracias a Megascans
- Las lápidas usadas para el escenario igualmente fueron obetnidas en Megascans [Meg22c]
- Los modelos de las calabazas usados para las trampas fueron obtenidos del Marketplace de Epic Games [Dev22]
- El creador de círculos que fue utilizado para la trampa de retención fue obtenido del Marketplace de Epic Games [Mot20]
- Los personajes y las animaciones utilizados para los enemigos fueron obtenido en Mixamo [Mix22]

6.7. Testeo y correcciones

Una vez terminado el proceso de programación se realizaron algunas pruebas con las que se identificaron los siguientes elementos para hacer cambios dentro del juego:

- Aumento en la velocidad de los enemigos: Se identificó que con las velocidades con las que contaban los enemigos era demasiado sencillo para el jugador tener mucha distancia respecto a ellos, lo que facilitaba bastante el evitar el daño.
- Mejoras dentro de la inteligencia artificial: Se identificó que favorecería a la jugabilidad hacer mejoras dentro de la inteligencia artificial de los

enemigos para generar pequeñas variaciones dentro de los caminos a seguir y así evitar grandes aglomeraciones al estar siguiendo varios de ellos el mismo camino.

- Cambios en la distancia entre las lápidas del escenario: Durante las pruebas se observó que al hacer determinados movimientos el jugador puede quedar en el centro de ciertos elementos del escenario impidiendo que pueda moverse a menos que los salte. Es por esto que se disminuyó ciertas distancias entre estos elementos para evitar que exista este tipo de espacios.

Capítulo 7

Conclusiones y trabajo futuro

Tener una correcta planeación dentro del desarrollo de cualquier tipo de software es vital para evitar complicaciones que conlleven a un aumento en la cantidad de tiempo y costos, es por esto que, seguir una metodología como la propuesta puede ser de gran utilidad para los desarrolladores de, en este caso, videojuegos.

Las pruebas realizadas dentro del ejemplo del uso de la metodología basada en patrones de diseño de software demostraron que aplicarla puede facilitarle a los desarrolladores las tareas de diseño de mecánicas y programación de las mismas al tener contemplado desde una etapa temprana tanto los elementos que conforman sus mecánicas, las relaciones entre ellas y los patrones de diseño de software que pueden aplicar para programarlas.

Dentro de los patrones de diseño de software que fueron tratados durante el trabajo es importante recalcar que varios de ellos ya son implementados directamente dentro de motores de juego que se usan a nivel profesional dentro de la industria, como es el caso de Unreal Engine. Es por esto que tener conocimiento sobre los mismos será de utilidad para que los programadores entiendan mejor las herramientas brindadas por el motor e incluso puedan facilitarles la obtención de un trabajo a nivel profesional dentro del futuro, ya sea porque co-

nocen mejor el motor o tengan los conocimientos que puedan ayudar a la hora de desarrollar un motor propio.

Se espera poder presentar dentro de algunos estudios de videojuegos la metodología planteada para así poder seguir probando su efectividad y recibir sugerencias por parte de los trabajadores de los mismos. En base a las sugerencias recibidas será posible hacer aún más eficiente la metodología, ya sea por medio de nuevas etapas o subetapas que puedan sugerirse, consejos sobre cómo mejorar el diseño de las mecánicas, otros patrones de diseño de software que suelen utilizarse en la industria, entre otros posibles aspectos.

También se espera poder presentar la metodología a desarrolladores que están comenzando dentro de este campo para que así puedan tener una guía que les ayude tanto a saber cómo desarrollar mecánicas, a conocer patrones de diseño de software y algunas de las herramientas con las que pueden trabajar en Unreal Engine para facilitar tareas dentro del desarrollo.

Capítulo 8

Anexo: Herramientas de Unreal Engine

- **Blueprints:** Lenguaje de programación visual diseñado para trabajar orientado a objetos dentro del motor Unreal Engine. Este lenguaje de programación trabaja en base a nodos que son conectados para trabajar de forma secuencial y cuenta con tipos de variables, funciones, condiciones, ciclos, entre otros elementos comunes dentro de los lenguajes de programación. [Eng22c]
- **Animation Blueprints:** Blueprints que permiten facilitar tareas relacionadas con las animaciones de los personajes, como pueden ser las transiciones entre las animaciones, activar eventos en base a las animaciones, combinar animaciones, entre otros. [Eng22b]
- **Actor:** Son todos los objetos capaces de ser añadidos dentro de un nivel, tanto al hacerlo directamente dentro del editor o al crearlos con C++ o Blueprints. [Eng22a]
- **State Machines:** Herramienta que puede se creada dentro de un Animation Blueprint para programar transiciones entre las animaciones en base a condiciones que determinarán el "estado" actual del personaje. [Eng22i]

- **Skeletal Mesh:** Conjunto de polígonos que dan forma a objetos o personajes que requieren de movimiento. Estos cuentan con un conjunto de "huesos" interconectados para servir como vértices sobre los cuales se podrán generar los movimientos. [Eng22h]
- **Static Mesh:** Forma geométrica creada en base polígonos que no cuenta con movimiento entre sus vértices y que permite darle una representación física dentro del juego a un objeto. [Eng22l]
- **Material:** Define la forma en que una superficie se comportará al ser dibujada dentro del mundo, esta contiene valores como el color, la transparencia, la reflectividad, entre otros. [Eng22f]
- **Hierarchical Isntanced Mesh Component:** Componente que permite la creación de varios Static Mesh que comparten elementos en común pero que pueden tener valores diferentes para atributos como la escala, la posición o la rotación. [Eng22j]
- **Pawn:** Clase que puede ser controlada tanto por un jugador como por una IA para realizar acciones dentro del mundo. [Eng22g]
- **UInputComponent:** Clase que le permite a un Pawn delegar acciones en base a inputs cuando ésta es controlada por un jugador. [Eng22k]
- **FText:** Tipo de dato usado para añadir texto en los componentes que serán observados por el usuario. [Eng22e]

Capítulo 9

Referencias

Bibliografía

[EGV94] Ralph Johnson Erich Gamma Richard Helm y John Vlssides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[FA99] Inés Sánchez de Bustamante Francisco Aquino. *Algunas reflexiones sobre acerca del juego y la creatividad desde el punto de vista constructivista*. Universidad Autónoma del Estado de México, 1999.

[EA12] Joris Dormans Ernest Adams. *Game Mechanics: Advanced Game Design*. New Riders, 2012.

[Nys14] Robert Nystrom. *Game Programming Patterns*. GB, 2014.

[Shv21] Alexander Shvets. *Dive Into Design Patterns*. 2021.

[Fab] Carlo Fabricatore. *Gameplay and game mechanics design: A ket to quality in videogames*.

[Fra] Albert T. Franch. *Introducción al diseño de videojuegos*. Universidat Oberta de Catalunya.

Enlaces de referencia

[Mac16] Machinations. 2016. URL: https://machinations.io/.

[Mot20] Dragon Motion. *Magic Circle Creator*. 2020. URL: https://www.unrealengine.com/marketplace/en-US/product/magic-circle-creator.

[Saw21] Josh Sawyer. 2021. URL: https://twitter.com/jesawyer/status/1453793728015962140?t=tiDoUPNAbA_zUzrkEoBOAg&s=19.

[Dev22] Pack Dev. *Halloween Pumpkins / 150+ Variations*. 2022. URL: https://www.unrealengine.com/marketplace/en-US/product/halloween-pumpkins-150-variations.

[Eng22a] Unreal Engine. *Actors in Unreal Engine*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/actors-in-unreal-engine/.

[Eng22b] Unreal Engine. *Animation Blueprints*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/animation-blueprints-in-unreal-engine/.

[Eng22c] Unreal Engine. *Blueprints Visual Scripting*. 2022. URL: https://docs.unrealengine.com/5.1/en-US/blueprints-visual-scripting-in-unreal-engine/#:~:text=The%20Blueprint%20Visual%20Scripting%20system%20in%20Unreal%20Engine,object-oriented%20%2800%29%20classes%20or%20objects%20in%20the%20engine..

[Eng22d] Unreal Engine. *EInputEvent*. 2022. URL: https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/Engine/EInputEvent/.

[Eng22e] Unreal Engine. *FText*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/ftext-in-unreal-engine/.

[Eng22f] Unreal Engine. *Materials*. 2022. URL: https://docs.unrealengine.com/5.1/en-US/unreal-engine-materials/#:~:text=Materials%20in%20Unreal%20Engine%20define%20the%20surface%20properties,should%20interact%20with%20the%20light%20in%20your%20scene..

[Eng22h] Unreal Engine. *Skeletal Meshes*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/skeletal-meshes/.

[Eng22i] Unreal Engine. *State Machines*. 2022. URL: https://docs.unrealengine.com/5.1/en-US/state-machines-in-unreal-engine/.

[Eng22j] Unreal Engine. *UHierarchicalInstancedMeshComponent*. 2022. URL: https://docs.unrealengine.com/5.0/en-US/API/Runtime/Engine/Components/UHierarchicalInstancedStaticMesh-/.

[Eng22k] Unreal Engine. *UInputComponent*. 2022. URL: https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Components/UInputComponent/.

[Eng22l] Unreal Engine. *UStaticMesh*. 2022. URL: https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Engine/UStaticMesh/.

[Meg22a] Quixel Megascans. *Quixel Castle Wall*. 2022. URL: https://quixel.com/megascans/purchased?assetId=tc2kbiqt.

[Meg22b] Quixel Megascans. *Quixel Medieval Iron Door*. 2022. URL: https://quixel.com/megascans/purchased?assetId=ub1gfbwew.

[Meg22c] Quixel Megascans. *Quixel Mossy Gravestone*. 2022. URL: https://quixel.com/megascans/purchased?assetId=wjuvdeuqx.

[Meg22d] Quixel Megascans. *Quixel Mossy Ground*. 2022. URL: https://quixel.com/megascans/purchased?assetId=peiqm20.

[Mix22] Mixamo. *Mixamo*. 2022. URL: https://www.mixamo.com/#/.

[Won22] WonderShare. *¿Qué es UML?* 2022. URL: https://www.edrawsoft.com/es/uml/.

[AMA] Gema Sáez Rodríguez Antonio Monroy Antón. *Evolución del juego a lo largo de la historia*. URL: https://www.efdeportes.com/efd143/evolucion-del-juego-a-lo-largo-de-la-historia.htm.

[A.T] A.T. *Historia de los juegos de mesa: desde los orígenes hasta nuestros tiempos*. URL: https://www.tiratu.com/noticias-juegos/historia-juegos-mesa.

[Gar] Mauricio García. *Scrum y desarrollo de videojuegos*. URL: https://ceoindie.me/2018/03/18/introduccion-metogologia-scrum-desarrollo-videojuegos-1/.

[Mon] Luis Ángel Monge. *Metodología Casca en los Videojuegos*. URL: https://www.ingenioteka.com/metodologia-de-cascada-en-los-videojuegos/.

[Shv] Alexander Shvets. *Design Patterns*. URL: https://refactoring.guru/design-patterns.