Proyecto final: Inkzanity Unleashed

Sergio Jiménez Romero, Carlos Martínez Cuenca,
Alberto Velasco Rodríguez

11 de enero de 2025

Índice

1. Reporte del Proyecto

1.1. Diseño inicial

Vamos a empezar por recordar la idea de diseño inicial que teníamos. En nuestro UML había cuatro managers (Sound, Scene, Data y Game) y pensábamos usar los patrones factory, bridge, decorator y state.

En cuanto a funcionalidades, en nuestro UML hablábamos de:

- Dinero.
- Puntuación.
- Varios tipos de armas.
- Una tienda.
- Pintura y superficies pintables.
- Equipos.
- Enemigos.
- Personajes.

En nuestras historias de usuario mencionamos:

- Movimiento del personaje.
- Cámara y movimiento de cámara.
- Barra de vida.
- Power ups.
- Límite de tiempo para la partida.
- Pintar la superficie.
- Transformación en calamar.
- Recargar munición como calamar.
- Armas y poder pintar superficies.
- Eliminar enemigos disparándolos.
- Equipos.
- Guardado de progreso.
- Desbloquear armas.
- Un menú.

1.2. Dificultades encontradas

El principal impedimento ha sido el tiempo. La dificultad fue más alta de lo esperado en algunas partes (detalladas a continuación), sumado a la realización de otros dos proyectos. Esto nos ha llevado a rebajar algunas de nuestras expectativas. La filosofía de desarrollo ha sido priorizar lo esencial para tener un juego funcional y luego añadir detalles.

1.2.1. Dificultades concretas

Pintura La característica principal de Splatoon, la pintura, ha sido el principal desafío. Esto incluye:

- Necesidad de un shader propio para cada superficie pintable.
- Funciones auxiliares como detección de color del jugador, cálculo de áreas pintadas, pintar en tiempo real...

El proceso es largo y complicado, involucrando partículas, colisiones y texturas en tiempo real. Se ha intentado respetar los principios vistos en clase, pero la implementación puede que no sea la mas eficiente por el simple hecho de que desconocemos si esta bien o no. Nos hemos apoyado en dos videos (Mix and Jam y Guinxu) para orientarnos.

Modelo del jugador y animaciones Otro de los problemas principales ha sido el modelo del jugador, pero sobre todo el que se ejecuten las animaciones correctas. No venían bien implementadas en el modelo de la tienda de Unity y tuvimos que trabajar bastante para conseguir que todo se ejecutara correctamente, en especial para las animaciones de disparar mientras que corres. Otro problema que costo resolver es como equipar las armas. El modelo venia con un arma predeterminada, pero como queríamos tener más armas teníamos que cambiarla. Esto resulto ser muy complicado, pues el arma estaba juntada al esqueleto que controla las animaciones del personaje. Finalmente la solución que adoptamos fue forzar al arma a seguir la posición de la mano derecha del personaje con unos ciertos offsets de rotación y posición para que pareciera que el personaje la estaba cogiendo.

Problemas de código Tuvimos bastantes problemas con las corrutinas, deteniendo las incorrectas, lo que causaba comportamientos anómalos (muerte al poco tiempo de reaparecer, quedarse sin munición...).

Uso de Git No tuvimos problemas con el tamaño de archivos (no tuvimos que usar git LFS), pero una persona instaló versiones distintas de paquetes de Unity, causando conflictos en cada merge. Usamos cuatro ramas (main, dev/sergio, dev/Alberto y dev/Carlos).

Problemas de diseño El principal problema fue la arquitectura del jugador. Tuvimos que usar el patrón builder, ya que una factory no ofrecía la flexibilidad necesaria. Esto se discute en profundidad más adelante

2. Comparación de funcionalidades propuestas frente a realizadas

- **Dinero**: no se ha realizado.
- **Varios tipos de armas**: implementados estéticamente, con código estructurado para poder añadir balas, cadencias y daños distintos.

- **Puntuación**: calculada pero no mostrada (la puntuacion es el numero de pixeles de la textura de un color, es un número difícil de interpretar).
- **Pintura y superficies pintables**: se han realizado la inmensa mayoría de las funcionalidades que tiene la pintura en splatoon. La principal funcionalidad no implementada es que el calamar "nade" por pinturas en paredes, por lo que la superficie pintable solo es el suelo. La pintura de cada equipo puede ser de cualquier color, el suelo se pinta del mismo color de la pintura que dispara el jugador, las superficies se pueden pintar y repintar, se puede saber de que color es la pintura sobre la que estas, se puede saber cuanta parte del mapa esta pintada...
- **Tienda**: no se ha realizado.
- **Enemigos**: se ha creado una IA simple que juega contra el jugador. Uno de nuestros planes originales era tener también un modo de oleadas con enemigos no humanoides (slimes por ejemplo), que se correspondían con los enemigos de nuestro UML original. Decidimos priorizar el desarrollo de la versión normal del juego sobre esto y al final no se ha implementado. La IA del modo que se ha implementado usa NavMesh para moverse y una maquina de estados para decidir qué acción hace en cada momento
- **Personajes**: implementación de un personaje similar al de Splatoon.
- **Movimiento del personaje**: el personaje se puede mover pero no saltar. Decidimos que no saltara porque no existía animación de salto. El movimiento se ha realizado con físicas rigidbody para poder cumplir con el punto del proyecto que decía de usar físicas rigidbody
- **Cámara y movimiento**: implementado en tercera persona permitirendo apuntar a donde miras.
- **Barra de vida**: implementada.
- **PowerUps**: código hecho para uno de velocidad, pero no implementado (daría un boost de velocidad en un mapa pequeño y cerrado, lo que dificulta el movimiento).
- **Límite de tiempo**: configurable desde Unity, por defecto 90 segundos.
- **Transformación en calamar**: implementada excepto nadar por paredes.
- **Recargar munición como calamar**: implementado.
- **Armas**: tres implementadas con menú de selección.
- **Eliminar enemigos**: implementado. También se sufre daño por pintura rival.
- **Equipos**: implementado, numero de miembros configurable desde Unity (3 por defecto).
- **Guardado de progreso**: parcialmente implementado. Como las funcionalidades que iban a usarlo no se llegaron a implementar se abandono su desarrollo a la mitad. El codigo esta en el repositorio
- **Menú**: implementado, pero la sección de opciones no está completa. Nos hubiera gustado haber puesto la elección de color.
- **Desbloquear armas**: no implementado. No tiene sentido sin una tienda

3. Patrones implementados

- **Singleton**: se implementa a través de una clase genérica tanto para objetos no Monobehaviour como para componentes que si heredan de Monobehaviour (Singleton¡T¿y MonobehaviourSingleton¡T¿respectivamente). En nuestro código lo han usado GameManager, PaintManager, ColisionHandler. Al final no hemos incluido un SceneManager (solo hay dos escenas y la transición de una a otra es clara en momentos muy definidos, por lo que pensamos que no era necesario crear una clase dedicada, que lo de Unity era suficiente), SoundManager (no hemos incluido sonidos más allá de los que puedan incluir los assets de Unity y el Data manager no lo llegamos a implementar aunque estaba planeado con el sistema de guardado y carga de datos
- **Bridge**:implementado en muchos lugares del código (jugador tiene armas, jugador y armas tienen color, armas tienen partículas...) Probablemente el patrón más utilizado en todo el proyecto nos ha servido para separar la abstracción de la implementación en muchos lugares.
- **Flyweight**:Utilizado para que todas las armas compartan las texturas y la textura de la pintura, solo cambiando el color. Se ha implementado a través de los prefabs, que ya hacen este trabajo por nosotros
- **Command**: se ha implementado este patrón en el código, usado para disparar, mover al jugador y convertirlo en calamar. Con esto hemos conseguido abstraer el input de la acción, lo que nos ha permitido implementar el control de la IA muy fácilmente a través de la interacción con el componente player controller.
- **State**: usado tanto para la lógica del jugador (cuando debe recargar, sufrir daño por estar encima de la pintura y cuando no debe pasar nada) y para determinar el comportamiento de la IA (cuando debe moverse, disparar o recargar)
- **Observer**: utilizado para lanzar eventos. Le dice al jugador cuando recargar y cuando curarse además de iniciar toda la lógica de inicio y final de partida. También esta presente en los menus, aunque a través de funcionalidades del editor de Unity
- **Builder**: este patron se ha utilizado para crear e instanciar el jugador. El motivo se va a discutir en la parte de decisiones de diseño en más profundidad

4. Principios SOLID

A lo largo de todo el proyecto se ha tratado de respetar al máximo los principios SOLID. Hemos decidido incluir un ejemplo de una aplicación de cada uno.

- **Single Responsibility**: el Paint manager delega el calculo de colisiones al colisión handler. El Paint manager se encarga de todo lo que tenga que ver con pintar y saber que áreas están pintadas (modificar el shader). Si también calculara colisiones haría dos cosas, incumpliendo este principio.
- **Open/Close**: para pasar datos de una escena a otra se han creado los Information Container. Se ha implementado su lectura en el GameManager de tal forma que cada cosa que el GameManager tenga que conocer de escenas anteriores se pueda cargar sin tener que modificar su código (se crean IContainerLoaders para cada caso especifico) de tal forma que si quisiéramos pasarle los datos de color o numero de jugador, podríamos crear un InformationContainer para ese tipo y un ContainerLoader que cargue esos datos sin modificar el código del game Manager.

- **Liskov Substitution**: aunque al final no hemos implementado los enemigos no humanoides, su controller es un ejemplo de donde hemos aplicado liskov (ya que programamos el controller del jugador pensando en que sí que los haríamos). El controller del jugador inicialmente heredaba de un base controller que tenía un método shoot. Nos dimos cuenta de que, como el jugador disparaba, si usábamos el mismo controller para los enemigos tendríamos enemigos que disparan (es una funcionalidad que no tienen, modifica el comportamiento de la clase base. Por lo que movimos shoot and move al player controller y en el base dejamos solo los comandos de movimiento (pues no sabíamos si el movimiento de los enemigos iba a ser igual que el del jugador)
- **Interface Segregation**: también en algo que no se llego a implementar, pero para la carga y guardado de datos no todos aquellos que guardan datos cargan y viceversa. Por tanto, decidimos dividir la interfaz en 2 (ILoader y ISaver)
- **Dependency Inversion**: Este principio lo aplicamos cuando hicimos el movimiento. Los comandos de move no podían tener el rigidbody del jugador directamente porque si quisiéramos mover algo que no fuera un rigidbody íbamos a tener problemas. Por ello implementamos un intermediario (aunque al final solo hay un tipo, MovePlayer, porque los jugadores son lo único que se mueve) para dar mas flexibilidad y escalabilidad en este contexto

5. Decisiones de diseño

El principal problema en cuanto a diseño del proyecto es en torno a la arquitectura del jugador y todo lo que lo rodea. Cuando empezamos a crear jugadores nos dimos cuenta de que una factory no nos daba toda la flexibilidad que queríamos, ya que aunque hiciéramos una factory con muchos argumentos no podíamos hacerlo bien para jugadores controlados por personas y por la IA, ya que tienen módulos distintos como veremos mas adelante. Buscando en refactoring guru encontramos el patron builder, que se ajustaba mucho mas a nuestras necesidades (permite añadir los componentes uno a uno para luego montar todo el jugador. Así, en Team (que es donde se crean los jugadores) tenemos lo que hemos llamado "pseudofactories", ya que dándole únicamente la posición (y el camino de Navmesh que debe seguir) podíamos crear jugadores controlados por el jugador o por la IA de forma sencilla. Como no tenemos mucha experiencia con el patron builder no sabemos si la solución a la que hemos llegado es la mejor, si eso que hemos hecho como métodos deberían haber sido clases aisladas o no, pero creemos que es una solución que respeta en gran medida los patrones de diseño. No se han implementado factories en otros lados porque:

1. No hay enemigos humanoides que si hubieran necesitado una factory
2. Solo se usa un tipo de arma por partida y esta no se sabe cual de los 3 prefabs es, por lo que al ser solo un tipo decidimos pasárselo al builder directamente y que este lo añadiera al jugador
3. Otras cosas que a lo mejor podrían haber usado factories las podíamos obtener por Serialize fields o instanciando, siendo estos casos de componentes que no tenian varias clases.

Se ha decidido centralizar la gestión de la pintura en una clase porque esto facilita buscar sobre que color esta el jugador y luego el recuento final. Si la clase PaintManager tiene conocimiento de todas las superficies que se han pintado luego no hay que ir buscandolas una a una Se ha decidido usar state en vez de strategy porque pensamos que nos da mas flexibilidad y porque para el jugador (por ejemplo) la transición entre estados se modela con una finite state machine, lo que hace mas sencillo trabajar con state que con strategy. En cuanto a las decisiones sobre la pintura (shader, partículas...) se ha tratado de hacer algo similar a lo que ya existía, adaptándolo a nuestro caso y al mismo tiempo manteniendo la funcionalidad. Se ha tratado de combinar los mejores aspectos de

cada implementación, pero desconocemos como de eficiente es la implementación. Para la detección de colisiones se ha usado raycast donde se ha podido. Se decidió no hacer una clase SceneManager ya que usamos las funcionalidades que incorpora Unity y se opto por los information container para pasar información porque pensamos que escalaria mejor y que además probablemente se podría construir un SceneManager encima de lo que tenemos. Las funcionalidades que hemos decidido no implementar ha sido o bien porque requería tiempo para algo no crucial para el juego (tienda), porque requería muchísimo tiempo o una cantidad de tiempo indeterminada que podía ser alta (modo de enemigos no humanoides por olas y distintos tipos de balas según el arma, que por las partículas podía llevar mucho tiempo) o porque era el primer paso de algo que venía después (guardado)

Aunque ya lo hemos mencionado, se implemento el patron command y el player controller para que cuando se desarrollara el modulo de IA este pudiera acceder facilmente a las acciones del jugador sin que tuvieramos que modificar nada

Orden de implementación

1. Menú, movimiento y cámara.
2. Personaje y mapa de prueba.
3. Pintura.
4. Importar modelo de armas.
5. Disparos.
6. Transformación en calamar.
7. Temporizador.
8. Builder.
9. Animaciones.
10. Muerte de personajes.
11. Munición.
12. Máquina de estados del jugador.
13. IA.
14. Equipos.
15. Inicio y final de partida.