Final Project C++

Título del juego:

Arcane Pixels: Battle of Elements

Características	Detalles
Ambiente	El juego se desarrolla en un entorno en 2D al estilo pixel art. Ambientado en un escenario minimalista de bloques blancos sobre un fondo negro. Los bloques funcionan como plataformas y terrenos donde los participantes pueden moverse, saltar, agarrar las armas que van cayendo del cielo y disparar en distintas direcciones.
Sistema de elementos	El jugador y los bots pueden recoger armas mágicas aleatorias durante la partida que van cayendo del cielo en lugares aleatorias del mapa. Estas armas están vinculadas a los elementos de fuego, agua, tierra, aire y electricidad, con ventajas y desventajas únicas para cada elemento, aportando un nivel de estrategia al juego. Cuando un jugador recoge un arma, su outline cambia de color según el elemento de la misma, agregando un toque de color al diseño en blanco y negro.
Implementación de IA	El juego está configurado para un jugador humano y cuatro bots. Los bots utilizan la inteligencia artificial para tomar decisiones autónomas, lo que hace que el juego sea más dinámico y desafiante. Se considerará el uso de técnicas de aprendizaje automático para mejorar la capacidad de los bots para adaptarse y responder a las acciones del jugador.

Características	Detalles
Modo de Juego	El juego adopta un estilo de Battle Royale, donde solo puede haber un ganador. Si el jugador humano es eliminado, el juego termina. Para ganar, el jugador debe ser el último superviviente, lo que implica derrotar a todos los bots.
Controles y Mecánicas de Combate	El juego cuenta con controles simplificados y mecánicas de combate fluidas, lo que permite al jugador y a los bots moverse por el mapa y interactuar con los objetos con facilidad. Los participantes pueden disparar sus armas mágicas o lanzarlas para recoger una nueva.
Diseño de Personajes y Apariencia	Los personajes tienen un diseño minimalista en blanco, con extremidades claramente distinguibles y movimientos legibles. Las armas mágicas son simples, parecidas a varitas, y cambian de color según el elemento que representan.
Progresión	El jugador puede seguir su progreso a través de un historial de partidas ganadas.

▼ Listado de requerimientos

1. **Requerimientos Técnicos y de Diseño**

   a. **Diseño Gráfico**: El juego tendrá un estilo gráfico 2D en pixel art. Los elementos visuales serán minimalistas, principalmente en blanco y negro, con colores adicionales para indicar los elementos de las armas mágicas.

   b. **Interfaz de Usuario**: Se debe diseñar una interfaz de usuario que permita a los jugadores navegar por las opciones del juego, empezar la partida o ver su historial de partidas o salir del juego.

   c. **Sonido y Música**: Se debe incluir música de fondo y efectos de sonido para los movimientos de los personajes, los ataques y otras acciones en el juego.

2. **Requerimientos de Juego y Mecánicas**

   a. **Mecánicas de Juego**: El juego debe ser un Battle Royale donde solo puede haber un ganador. Los participantes deben poder moverse por el

mapa, interactuar con objetos y recoger armas mágicas. Se debe poder eliminar a otros bots y ser eliminado.

b. **Físicas y Colisiones**: Debe haber un sistema de físicas que permita a los personajes moverse, saltar, recorrer y soltar las armas y así mismo disparar en una dirección según la ubicación del mouse. El sistema de colisiones debe detectar cuando los ataques golpean a un jugador, y cuando los personajes chocan con los bloques del ambiente.

    i. **Movimiento y Saltos**: Los personajes deben ser capaces de moverse en cualquier dirección y saltar. El motor de físicas debe gestionar la velocidad de movimiento y la altura del salto, así como la gravedad que afecta a los personajes cuando están en el aire.

    ii. **Interacción con las Armas**: Los personajes deben ser capaces de recoger y soltar armas. Cuando un personaje recoge un arma, el arma debe seguir al personaje, y cuando el personaje suelta el arma, el arma debe quedarse en la ubicación donde se soltó.

    iii. **Disparo de las Armas Mágicas**: Cuando un personaje usa una arma mágica, el motor de físicas debe gestionar la velocidad y la dirección del efecto mágico (como un rayo, una llama, un chorro de agua, etc.). La dirección del efecto mágico debe ser determinada por la ubicación del mouse en relación con la posición del personaje. Cada efecto mágico puede tener sus propias propiedades y comportamientos únicos, por ejemplo, un rayo podría moverse muy rápido y en línea recta, mientras que un chorro de agua podría moverse más lentamente y en una trayectoria curva. Además, el sistema de físicas debe ser capaz de gestionar los efectos de las armas mágicas en otros personajes y en el entorno. Por ejemplo, un rayo podría paralizar a un personaje durante un corto período de tiempo, o una llama podría causar daño continuo durante un cierto período de tiempo. Además, algunos efectos mágicos podrían interactuar con el entorno de maneras únicas, como un chorro de agua que puede apagar las llamas.

    iv. **Colisiones**: Se debe tener un sistema de detección de colisiones que gestione cuando los proyectiles golpean a otros personajes, cuando los personajes chocan con el entorno y cuando los personajes intentan recoger un arma. Cuando un proyectil golpea a un personaje, ese

personaje debe recibir daño. Cuando un personaje choca con el entorno, debe ser impedido de moverse a través del objeto con el que está chocando.

v. **Sistema de Daño**: Debe haber un sistema que gestione el daño que los personajes reciben cuando son golpeados por un proyectil. Esto puede implicar la reducción de la salud del personaje y la eventual eliminación del personaje cuando su salud llega a cero.

c. **Sistema de Elementos**: Debe haber un sistema de elementos que permita a las armas mágicas estar vinculadas a uno de los cinco elementos: fuego, agua, tierra, aire y electricidad. Cada elemento debe tener ventajas y desventajas únicas.

d. **Controles**: Los controles del juego deben ser simples e intuitivos. Los jugadores deben ser capaces de moverse, saltar, disparar y lanzar armas con facilidad.

e. **Diseño de Personajes y Apariencia**: Los personajes deben tener un diseño minimalista en blanco, con extremidades claramente distinguibles. Deben parecer simples píxeles o bloques blancos. Las armas mágicas deben tener un diseño simple y parecer varitas, con un color distintivo para cada elemento que representan: fuego (rojo), agua (azul), tierra (verde), aire (gris) y electricidad (amarillo). Además, cuando un personaje recoge una arma mágica, el color del personaje debe cambiar al color del elemento que representa la arma. Esto significa que el personaje adoptará el color de la varita que esté llevando en ese momento, proporcionando una retroalimentación visual inmediata al jugador sobre qué tipo de arma mágica está utilizando el personaje.

f. **Progresión**: Debe haber un sistema de progresión que permita a los jugadores ver un historial de sus partidas ganadas.

3. **Requerimientos de Inteligencia Artificial (IA)**

a. **Bots**: Debe haber cuatro bots que jueguen junto al jugador humano. Estos bots deben ser capaces de moverse por el mapa, recoger armas, atacar y evitar ataques.

b. **Decisiones Autónomas**: Los bots deben ser capaces de tomar decisiones autónomas para hacer el juego más desafiante. Esto podría implicar decidir cuándo atacar, cuándo esquivar, qué arma recoger y cómo moverse por el mapa.

c. **Machine Learning (Opcional)**: Se podría considerar la implementación de técnicas de aprendizaje automático para mejorar la habilidad de los bots para adaptarse y responder a las acciones del jugador. Este es un requerimiento más avanzado que puede consumir más tiempo, y conocimiento, por lo tanto se deja como opcional.

4. **Funcionalidades Adicionales**

   a. **Música y Efectos de Sonido**: El juego debe incluir música de fondo y efectos de sonido para mejorar la inmersión. Los sonidos podrían incluir los de los movimientos de los personajes, los ataques, y otros sonidos ambientales.

   b. **Requerimientos de Armas Mágicas**

      i. **Generación Aleatoria**: Debe haber un sistema que genere armas mágicas de manera aleatoria en diferentes ubicaciones del mapa durante la partida.

      ii. **Recogida de Armas**: Los personajes (tanto los jugadores como los bots) deben ser capaces de recoger estas armas mágicas para usarlas en el combate.

      iii. **Uso de Armas**: Las armas mágicas deben tener efectos únicos en el combate, basados en el elemento al que están vinculadas.

      iv. **Inteligencia Artificial (IA)**: La IA de los bots debe ser capaz de reconocer la presencia de estas armas mágicas, moverse hacia ellas para recogerlas, y utilizarlas en el combate de manera efectiva.

5. **Pruebas**

   a. **Pruebas de Unidad**: Se deben realizar pruebas de unidad en todos los componentes individuales del código para asegurar que funcionen

correctamente.

b. **Pruebas de Integración**: Una vez que los componentes individuales hayan sido probados, se deben realizar pruebas de integración para asegurar que todos los componentes trabajen juntos correctamente.

6. **Documentación**

a. **Documentación del Usuario**: Debería haber una documentación completa y fácil de entender para los usuarios, que explique cómo jugar el juego, cómo usar la interfaz de usuario, y cualquier otra información importante.

b. **Documentación del Código**: El código del juego debe estar bien documentado para facilitar su mantenimiento y desarrollo futuro.

c. **Manejo de Errores**: El juego debería tener un manejo robusto de errores para evitar que los jugadores experimenten fallas o cierres inesperados del juego.

▼ Objetos con los que se pueden cumplir esos requerimientos

1. **Requerimientos Técnicos y de Diseño**

a. **Diseño Gráfico:** En Qt, se puede usar la clase `QGraphicsScene` para manejar los elementos gráficos del juego. Los personajes y las armas mágicas pueden ser representados como objetos de la clase `QGraphicsPixmapItem`, que es una subclase de `QGraphicsItem` diseñada para dibujar imágenes de píxeles.

b. **Interfaz de Usuario**: Qt proporciona la clase `QMainWindow` que se puede usar como la ventana principal del juego. Se puede usar `QMenu` y `QAction` para crear un menú de opciones. Para mostrar el historial de partidas, se podría usar un `QDialog` con un `QListWidget` o un `QTableWidget`.

c. **Sonido y Música**: La clase `QMediaPlayer` de Qt puede ser usada para reproducir música y efectos de sonido.

2. **Requerimientos de Juego y Mecánicas**

    a. **Mecánicas de Juego**: La lógica del juego puede ser implementada en una clase personalizada, tal vez llamada `GameLogic`. Esta clase podría tener métodos para manejar el inicio de una partida, el fin de una partida, la eliminación de un personaje, entre otras cosas más.

    b. **Físicas y Colisiones**: Se puede implementar la física del juego y la detección de colisiones usando las clases `QGraphicsItem` y `QGraphicsScene`. Cada `QGraphicsItem` puede tener un `shape()` y un `boundingRect()`, que son utilizados por `QGraphicsScene` para la detección de colisiones. Para las físicas, se podría tener una clase `PhysicsEngine` que se encargue de actualizar la posición y velocidad de los personajes y proyectiles en cada frame, basándose en las fuerzas que actúan sobre ellos (como la gravedad y los movimientos del jugador).

    c. **Sistema de Elementos**: Se podría tener una clase abstracta `MagicWeapon` con subclases para cada tipo de arma mágica (`FireWeapon`, `WaterWeapon`, etc.). Cada subclase podría implementar su propio comportamiento, como la forma en que afecta a otros personajes y cómo interactúa con el entorno.

    d. **Controles**: La clase `QMainWindow` (o cualquier otra clase que herede de `QWidget`) puede recibir eventos de teclado y ratón. Se puede redefinir los métodos `keyPressEvent()`, `keyReleaseEvent()`, `mousePressEvent()`, etc., para manejar estos eventos y controlar el movimiento del personaje y las acciones de disparo.

    e. **Diseño de Personajes y Apariencia**: Como mencioné antes, los personajes y las armas pueden ser representados como objetos de la clase `QGraphicsPixmapItem`. Se podría tener una clase `Character` y una clase `MagicWeapon` que hereden de `QGraphicsPixmapItem` y añadan funcionalidad adicional, como cambiar el color del personaje cuando recoge un arma.

    f. **Progresión**: Se podría tener una clase `GameHistory` que mantenga un registro de las partidas ganadas por el usuario. Esta clase podría tener métodos para agregar una nueva partida ganada al historial, obtener el número total de partidas ganadas, y así sucesivamente. Esta información podría ser almacenada en un archivo, y la clase `GameHistory` se encargaría

de leer y escribir esta información. Para mostrar el historial de partidas al usuario, se podría usar un `QDialog` con un `QListWidget` o un `QTableWidget`.

3. **Requerimientos de Inteligencia Artificial (IA)**
    a. `Bot`: Esta sería una subclase de `Character` con funcionalidades adicionales para la toma de decisiones autónomas. Se podría tener métodos para decidir la siguiente acción basándose en el estado actual del juego (como la ubicación de las armas mágicas, la ubicación de los otros personajes, la salud restante, etc.).
    b. Para las decisiones autónomas, se podría implementar una variedad de algoritmos. Por ejemplo, se puede usar un algoritmo de búsqueda para encontrar el camino más corto hacia una arma mágica, o un algoritmo de toma de decisiones como el árbol de decisiones para decidir cuándo atacar o esquivar.
    c. Para el aprendizaje automático, se necesitaría una forma de recolectar datos sobre el rendimiento de los bots y ajustar sus algoritmos de toma de decisiones en función de estos datos. Esto podría implicar la creación de una clase `BotTrainer` o algo así.

4. **Funcionalidades Adicionales**
    a. `AudioManager`: Esta clase podría encargarse de reproducir la música de fondo y los efectos de sonido. Se podría tener métodos para reproducir, pausar, y detener la música, así como para reproducir diferentes efectos de sonido.
    b. `WeaponManager`: Esta clase podría encargarse de la generación aleatoria de armas mágicas, y podría tener métodos para generar una nueva arma en una ubicación aleatoria, verificar si un personaje está en una ubicación donde puede recoger una arma, y asignar una arma a un personaje cuando la recoge.

5. **Pruebas**

a. y b. Se podría tener una clase un archivo Test.h y Test.cpp respectivamente para probar cada clase principal del código, con métodos para probar cada funcionalidad de la clase.

6. **Documentación**

   a. y b. La documentación del usuario y del código serían principalmente textuales.

   c. `ErrorHandler`: Esta clase podría encargarse de manejar cualquier error que se produzca durante la ejecución del juego. Se podría tener métodos para registrar errores y mostrar mensajes de error al usuario.

▼ Jerarquía del proyecto

/ArcanePixelsBoE

- ArcanePixelsBoE.pro

/Headers

- mainwindow.h
- Character.h
- Bot.h
- MagicWeapon.h
- FireWeapon.h
- WaterWeapon.h
- ElectricityWeapon.h
- AirWeapon.h
- AudioManager.h
- GameLogic.h
- WeaponManager.h
- ErrorHandler.h
- PhysicsEngine.h

- GameHistory.h
- Test.h

/Sources

- main.cpp
- mainwindow.cpp
- GameLogic.cpp
- Character.cpp
- Bot.cpp
- MagicWeapon.cpp
- FireWeapon.cpp
- WaterWeapon.cpp
- AudioManager.cpp
- WeaponManager.cpp
- ErrorHandler.cpp
- PhysicsEngine.cpp
- GameHistory.cpp
- Test.cpp

/Forms

- mainwindow.ui

/Resources

/Audio

- Music files
- Sound effect files

/Images

- Character sprites
- Weapon sprites

- Platform sprites
- Weapon effects sprites

▼ Detalles de los objetos (atributos y métodos) con lo que se va a pasar a implementar el proyecto

▼ Clase Character

```
#include <QGraphicsPixmapItem>

class Character : public QGraphicsPixmapItem {
public:
    Character(); // Constructor
    void move(int direction); // Mueve el personaje en una dirección
    void jump(); // Salta el personaje
    void attack(); // Ataca con el arma equipada
    void equipWeapon(); // Equipa un arma mágica
    void receiveDamage(int damage); // Recibe daño

private:
    int health; // Salud del personaje
    int speed; // Velocidad de movimiento del personaje
    MagicWeapon* equippedWeapon; // Arma mágica equipada
};
```

▼ Clase MagicWeapon

```
#include <QGraphicsPixmapItem>

class MagicWeapon : public QGraphicsPixmapItem {
public:
    MagicWeapon(); // Constructor
    void use(); // Usa el arma mágica

private:
    int power; // Poder del arma mágica
};
```

▼ Clase Bot (hereda de Character)

```
#include "Character.h"

class Bot : public Character {
public:
    Bot(); // Constructor
    void makeDecision(); // Toma una decisión autónoma

private:
    int decisionMakingFrequency; // Frecuencia con la que el bot toma decisiones
};
```

▼ Clase AudioManager

```
#include <QMediaPlayer>

class AudioManager {
public:
    AudioManager(); // Constructor
    void playSound(QString soundFile); // Reproduce un sonido
    void playMusic(QString musicFile); // Reproduce música
    void stopMusic(); // Detiene la música

private:
    QMediaPlayer* mediaPlayer; // Reproductor de medios
};
```

▼ Clase WeaponManager

```
#include "MagicWeapon.h"

class WeaponManager {
public:
    WeaponManager(); // Constructor
    void generateWeapon(); // Genera un arma mágica en una ubicación aleatoria

private:
    QVector<MagicWeapon*> weapons; // Lista de armas mágicas
};
```

▼ Clase Platform

```
#include <QGraphicsPixmapItem>

class Platform : public QGraphicsPixmapItem {
public:
    Platform(); // Constructor
};
```

▼ Clase Projectile (para los ataques de las armas)

```
#include <QGraphicsPixmapItem>

class Projectile : public QGraphicsPixmapItem {
public:
    Projectile(); // Constructor
    void move(); // Mueve el proyectil

private:
    int speed; // Velocidad del proyectil
    int power; // Poder del proyectil
};
```

▼ Clase GameLogic

```
#include "Character.h"
#include "Bot.h"
#include "Platform.h"
#include "Projectile.h"

class GameLogic {
public:
    GameLogic(); // Constructor
    void startGame(); // Inicia el juego
    void endGame(); // Termina el juego
    void checkCollisions(); // Verifica las colisiones

private:
    QVector<Character*> characters; // Lista de personajes
    QVector<Bot*> bots; // Lista de bots
    QVector<Platform*> platforms; // Lista de plataformas
    QVector<Projectile*> projectiles; // Lista de proyectiles
};
```

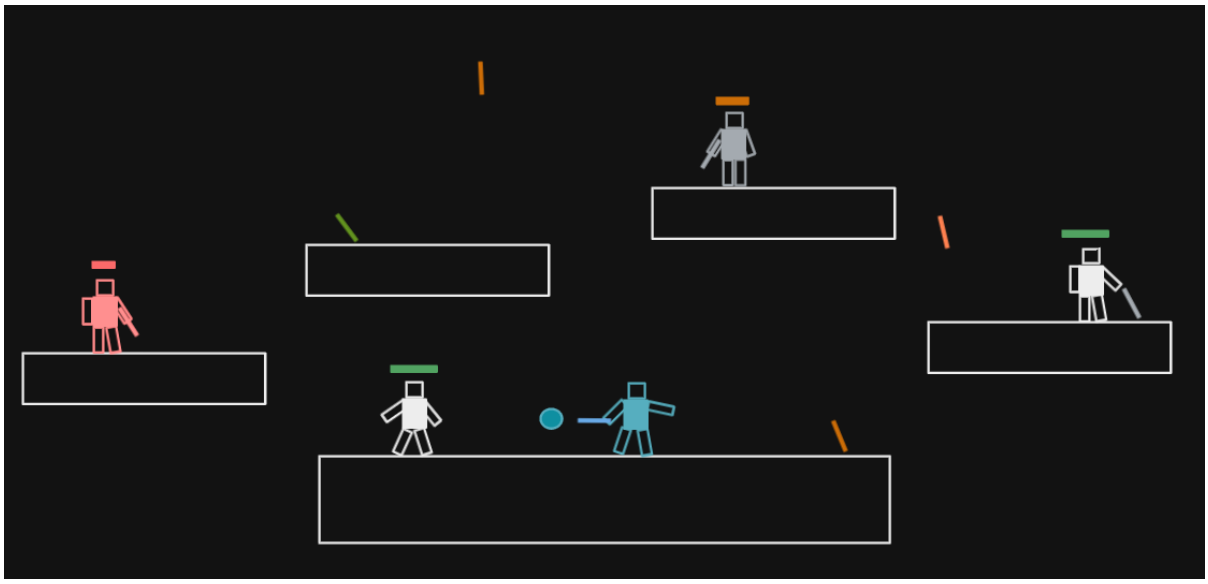
▼ Clase ErrorHandler

```
class ErrorHandler {
public:
    ErrorHandler(); // Constructor
    void logError(QString error); // Registra un error
    void displayError(QString error); // Muestra un error al usuario
};
```

▼ Clase MainWindow

```
#include <QMainWindow>
#include <QKeyEvent>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    // Funciones de evento de teclado
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

    // Funciones de evento de ratón
    void mousePressEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
};
```

▼ Ilustraciones



▼ Recursos

- **Mono Warrior 1bit Character**
- **4-Color Weapon Pack**
- **Various 1-bit pixel art style 16x16px. tiles, patterns and frames.**
- **1-BIT 16x16 2D Platformer Starter Tileset**
- **Free 1 bit dungeon tileset**
- **TiM - FREE Character Animations**
- **An 16x16 tileset and a background of a forest in 1-bit**
- **OpenGameArt.org**
- **EVil Wizard 2**
- **FREE-2D Pixel Art Character Asset Pack**
- **Skeleton Sprite Pack**
- **Explore game assets on itch.io**
- **FREE PIXEL MAGIC SPRITE EFFECTS PACK**
- **FREE WIZARD SPRITE SHEETS PIXEL ART**
- **FREE CHARACTERS & SPRITES**

- **LIBRESPRITE**