**1. Introducción**

**Undertale** es un videojuego de rol lanzado en 2015 que rápidamente ganó popularidad debido a su innovadora narrativa, su sistema de combate único y sus personajes carismáticos. Uno de los aspectos más destacados de Undertale es su capacidad para ofrecer múltiples caminos y finales, dependiendo de las decisiones del jugador a lo largo del juego. La libertad de elección del jugador es un pilar fundamental en su diseño, lo que lo convierte en un caso interesante para análisis y mejora en términos de optimización.

El código fuente de Undertale, disponible en plataformas como GitHub, nos ha permitido profundizar en la estructura interna del juego y observar áreas de oportunidad para optimizar su lógica, especialmente en la forma en que gestiona los eventos de diálogo. Undertale, a pesar de su éxito, cuenta con una estructura de código que puede beneficiarse significativamente de mejoras. Al analizar su código, nos dimos cuenta de que uno de los aspectos que requería mayor optimización era la lógica de eventos, particularmente en la forma en que se gestionan las secuencias de diálogo.

La decisión de analizar Undertale se fundamenta en la relevancia de su código abierto para la comunidad de desarrolladores y en la necesidad de optimización. Como desarrolladores, optimizar su estructura de condicionales, que resultan ser excesivamente numerosos y poco manejables a largo plazo, es clave para mejorar tanto la experiencia de desarrollo como la experiencia del jugador.

**2. Identificación de Problemas de Usabilidad**

Uno de los problemas más notorios en el código fuente de Undertale es su enfoque excesivo en el uso de condicionales if para manejar los eventos de diálogo. Este método se vuelve particularmente problemático cuando se considera la cantidad de eventos y situaciones que el juego debe gestionar.

**Principales Problemas Identificados:**

1. **Estructura de Condicionales Extensa**: El código abusa del uso de if-else para manejar una gran cantidad de eventos. Cada evento, decisión o respuesta del jugador se gestiona con nuevas ramas de condicionales, lo que resulta en una estructura extremadamente extensa y difícil de mantener.
2. **Propenso a Errores**: Dado que cada evento está vinculado a un conjunto específico de condiciones, cualquier cambio en la lógica del juego puede desencadenar errores no previstos. Esto es evidente cuando surgen bugs en el juego, ya que los errores en las condiciones pueden permitir diálogos fuera de lugar o romper la secuencia narrativa.
3. **Difícil de Mantener**: La falta de modularidad en el código hace que la tarea de modificar, mejorar o corregir errores sea tediosa para los desarrolladores. Encontrar y modificar el código relacionado con un diálogo específico puede convertirse en un proceso complejo debido a la gran cantidad de condicionales que interactúan entre sí.
4. **Impacto en la Experiencia del Jugador**: Desde el punto de vista del jugador, los errores o inconsistencias en los diálogos pueden romper la inmersión en el juego. Bugs relacionados con eventos mal gestionados pueden desencadenar diálogos incorrectos o eventos que no deberían ocurrir, lo que genera confusión y frustración.

Estos problemas pueden observarse en varias secciones del juego, y resultan particularmente notorios en escenas donde las decisiones del jugador desencadenan diferentes respuestas por parte de los NPCs. Esto genera situaciones en las que el juego puede comportarse de manera errática si no se cumplen correctamente las condiciones de los diálogos.

**3. Propuesta de Mejora**

El principal problema de la lógica de programación en Undertale es su dependencia de condicionales if para manejar los eventos y diálogos, lo que genera un código extenso, difícil de mantener y propenso a errores. Esto ocurre porque no se emplean técnicas más avanzadas, como funciones que permitan manejar eventos repetitivos de forma eficiente. Nuestra propuesta para mejorar esta situación incluye el uso de **funciones callback** y un **bucle despachador de eventos**, inspirados en las buenas prácticas de programación vistas en la materia PGE.

**Funciones Callback**

Las funciones callback son funciones que se pasan como argumento a otra función, permitiendo que una acción específica sea ejecutada cuando ciertos eventos ocurren. En este caso, las utilizaremos para gestionar los diálogos, asegurando que, cuando el jugador interactúe con un objeto o personaje, el sistema de diálogo se encargue de disparar los eventos correspondientes de forma organizada.

En lugar de usar múltiples if-else para cada posible diálogo, una función callback se encargará de ejecutar el código correcto cuando se cumplan ciertas condiciones. Esto hace que la lógica sea más manejable, evitando la acumulación de condicionales. La callback asociada a los diálogos también nos permite modularizar el comportamiento, facilitando la adición de nuevas interacciones sin generar un desorden en el código.

**Bucle Despachador de Eventos**

Un bucle despachador de eventos es una estructura que permite manejar de forma continua las interacciones y eventos que ocurren durante el juego, procesando cada entrada del jugador de manera eficiente. En el caso de Undertale, este bucle permitirá gestionar la interacción con los diálogos, de manera que cuando el jugador avance por el juego y se active un diálogo, el bucle será quien despache el evento correspondiente.

El despachador de eventos evitará que se tengan que programar manualmente todos los posibles diálogos y respuestas del juego. En su lugar, se delega la responsabilidad de manejar estos eventos al despachador, lo que facilita el proceso de actualización del código en el futuro.

**Mejora en la Lógica de Diálogos**

Aplicando estas soluciones, mejoramos drásticamente la lógica de manejo de diálogos en Undertale. Ahora, en lugar de depender de condicionales if para verificar quién está hablando o cuál diálogo mostrar, las funciones callback y el bucle despachador se encargan de estas tareas de manera mucho más eficiente. El resultado es un código más claro, legible, fácil de mantener y menos propenso a errores.

**4. Implementación del Prototipo**

Para implementar la propuesta de mejora, hemos trabajado sobre un prototipo en **GameMaker**, un entorno de desarrollo de videojuegos conocido por su capacidad para manejar lógica de juegos en lenguajes como GML (GameMaker Language) y C++. Aunque GameMaker no utiliza C++ directamente, su estructura de eventos y lógica nos permitió aplicar conceptos clave de C++ en la implementación del prototipo.

**Explicación de GameMaker**

**GameMaker** es un entorno diseñado para el desarrollo de videojuegos en 2D y 3D, con un lenguaje de scripting propio (GML). Su estructura de programación permite una manipulación más accesible de objetos, eventos y gráficos, lo que lo convierte en una opción adecuada para prototipar videojuegos de complejidad media como Undertale. Aunque GameMaker no usa C++ directamente, sus conceptos fundamentales, como el manejo de callbacks, eventos y bucles, se alinean con los principios de programación que queremos optimizar.

**Desarrollo de la Lógica de Diálogos**

Repositorio open source que contiene el código original: [Rovoska/undertale: Undertale src code (github.com)](https://github.com/Rovoska/undertale)

Link del fragmento de código que buscamos optimizar: [undertale/scripts/SCR\_TEXTTYPE.gml at master · Rovoska/undertale (github.com)](https://github.com/Rovoska/undertale/blob/master/scripts/SCR_TEXTTYPE.gml)

Hemos implementado la lógica optimizada para manejar los diálogos y eventos de interacción a lo largo del juego. En lugar de usar condicionales if repetitivas, empleamos callbacks y un sistema de bucles para gestionar las interacciones de manera eficiente. A continuación, se presentan fragmentos del código utilizado en GameMaker:

**Ejemplo de Fragmento de Código:**

// oTextBox

// Callback para el setup

function setup\_textbox() {

oPlayer.can\_move = false;

draw\_set\_font(fText);

draw\_set\_valign(fa\_top);

draw\_set\_halign(fa\_left);

page\_number = array\_length(text);

for (var p = 0; p < page\_number; p++) {

text\_length[p] = string\_length(text[p]);

text\_x\_offset[p] = 80; // Valor por defecto de desplazamiento del texto

portrait\_x\_offset[p] = 50;

line\_width = textbox\_width - border\*2 - text\_x\_offset[p];

// Determinar si el hablante es el NPC o el Player

if (speaker[p] == 0) { // NPC

if (speaker\_sprite[p] == noone) {

text\_x\_offset[p] = 15; // No hay sprite del NPC, ajustar texto a la izquierda

line\_width = textbox\_width - border\*2;

}

} else { // Player

if (player\_sprite[p] == sInvisible) { // Si el sprite es invisible, no dejar espacio

text\_x\_offset[p] = 15; // Ajustar texto sin espacio para sprite

line\_width = textbox\_width - border\*2;

} else if (player\_sprite[p] != noone) { // Si hay un sprite visible del player

text\_x\_offset[p] = 80; // Dejar espacio para el sprite

line\_width = textbox\_width - border\*2 - text\_x\_offset[p];

}

}

}

}

Este código es parte de la lógica de optimización de los diálogos en nuestro prototipo de Undertale en GameMaker. A través del uso de callbacks para gestionar la configuración de los diálogos, facilitamos la interacción entre el jugador y los NPCs, reduciendo la cantidad de condicionales if necesarias.

**Ventajas del Sistema Optimizado:**

* **Modularidad**: Las funciones callback hacen que cada parte del sistema sea más independiente y fácil de modificar.
* **Eficiencia**: El bucle despachador de eventos maneja las interacciones del jugador sin la necesidad de codificar cada posible respuesta en un if.
* **Mantenimiento**: Se facilita la tarea de corregir errores y actualizar el código sin afectar la estabilidad del juego.

***5. Resultados y Conclusiones***

**Discusión sobre la efectividad de las soluciones implementadas**  
Nuestra propuesta de optimización aplicada en el prototipo presenta mejoras claras frente a la lógica original del código de Undertale. En lugar de depender de largos y redundantes bloques de condicionales *if*, implementamos un enfoque basado en *callbacks* y un bucle despachador de eventos. Esto permite:

1. **Modularidad y Flexibilidad**: Al dividir las responsabilidades en pequeñas funciones manejables (por ejemplo, para la actualización del texto, el manejo de páginas, y la reproducción de sonidos), es mucho más fácil introducir cambios o agregar nuevas funcionalidades sin alterar otras partes del código. El uso de callbacks evita tener que escribir código repetitivo.
2. **Mantenibilidad**: Reducimos el riesgo de errores derivados de condicionales mal gestionadas. Al centralizar la lógica en funciones específicas para cada parte de la interacción (dibujar diálogos, avanzar páginas, gestionar sprites), resulta más sencillo identificar y corregir errores.
3. **Escalabilidad**: Nuestra implementación facilita la expansión del sistema de diálogos. El código actual puede soportar fácilmente más diálogos, personajes, e interacciones sin comprometer el rendimiento o la legibilidad del código.
4. **Eficiencia en el desarrollo**: Utilizar un sistema de eventos como el que planteamos minimiza la posibilidad de errores lógicos al manejar los diálogos. El uso de un bucle despachador simplifica la adición de nuevos casos o condiciones, haciéndolo más eficaz a largo plazo para mantener o expandir el juego.

**Reflexión sobre las limitaciones del prototipo y posibles mejoras futuras**  
Aunque nuestro prototipo optimiza varios aspectos de la lógica de diálogos, existen algunas limitaciones:

1. **Plataforma**: Si bien utilizamos GameMaker, que es un entorno eficiente para la creación de videojuegos, no es un lenguaje puramente de C++, lo cual podría limitar algunas optimizaciones de bajo nivel. Sin embargo, GameMaker es un entorno que emplea C++ en su núcleo, por lo que la eficiencia y manejo de memoria son garantizados de manera indirecta.
2. **Adaptación a grandes volúmenes de diálogos**: Aunque se ha optimizado la lógica, la implementación actual puede necesitar ajustes adicionales si el juego llegara a tener diálogos extremadamente extensos o múltiples lenguajes, donde se requeriría la integración de un sistema de internacionalización.
3. **Mejoras en la interfaz de usuario**: Si bien el sistema de diálogos funciona bien, en el futuro se podría añadir una mayor personalización en cuanto a la visualización de diálogos, como la animación de sprites en tiempo real o reacciones del entorno en función de los diálogos.
4. **Pruebas a largo plazo**: Aunque hemos asegurado la estabilidad del sistema en pruebas internas, una mayor cantidad de casos extremos (jugadores interactuando en momentos inesperados o bajo condiciones no previstas) podría revelar situaciones no cubiertas aún por el prototipo.

**Conclusiones generales sobre el análisis y el proceso de mejora de la interfaz**  
En resumen, el prototipo desarrollado ha demostrado ser significativamente más eficiente, flexible y mantenible que la implementación original de Undertale, particularmente en la gestión de diálogos. La estructura modular y el uso de callbacks y un bucle de eventos proporcionan una base sólida para futuras expansiones del juego. Si bien se utilizaron tecnologías modernas como GameMaker para el desarrollo, el enfoque sigue siendo completamente compatible con los principios de programación en C++, garantizando un rendimiento optimizado.

En términos de usabilidad, la implementación reduce los errores, mejora la experiencia del usuario al evitar problemas derivados de bugs lógicos, y permite una personalización más ágil del sistema de diálogo. Con futuras mejoras, este prototipo puede seguir evolucionando hacia un sistema de interacción más dinámico, eficiente y amigable para los desarrolladores y los jugadores.

LINK al Repositorio GitHub: [PGE\_2024/Tarea\_Parcial01 at master · GastMolina267/PGE\_2024 (github.com)](https://github.com/GastMolina267/PGE_2024/tree/master/Tarea_Parcial01)