Estructura del proyecto

El proyecto "Pixel Wall-E" tiene como objetivo la creación de una aplicación capaz de interpretar un lenguaje de programación personalizado, diseñado específicamente para la creación de arte de píxeles sobre un lienzo digital. La aplicación simula el control de un robot, Wall-E, que ejecuta comandos para pintar píxeles. Este informe actualizado detalla la estructura fundamental del intérprete, incorporando la información de los archivos de código fuente C# proporcionados previamente, junto con los nuevos archivos relacionados con la interfaz de usuario, la gestión de niveles y el control del lienzo: CanvasController.cs, FileManager.cs, LegendController.cs, UIManager.cs, LevelThemeData.cs, JugarManager.cs, LevelManager.cs, y TextureToTileMap.cs.

La funcionalidad central del proyecto implica tomar código fuente (ya sea desde un editor de texto o archivos .pw), procesarlo a través de las etapas de análisis léxico, sintáctico, semántico y, finalmente, interpretar los comandos para modificar una textura Texture2D que actúa como lienzo. Un aspecto crucial del proyecto es la implementación de un robusto sistema de reporte de errores para problemas sintácticos, semánticos y de tiempo de ejecución, así como una interfaz de usuario completa para la interacción.

El lenguaje personalizado ha sido diseñado para controlar a Wall-E en la creación de pixel art. Sus características clave incluyen:

- **Instrucciones (Comandos):** Acciones directas que afectan el estado de Wall-E (posición, color y tamaño del pincel) y el lienzo (dibujar formas, rellenar áreas). Todo código válido debe comenzar con el comando Spawn.
- **Asignaciones:** Permite asignar valores a variables.
- **Funciones:** Rutinas invocables que devuelven valores.
- **Etiquetas y Saltos Condicionales:** Mecanismos de control de flujo para navegar a través del código.
- **Ejecución Secuencial:** Las declaraciones se ejecutan una tras otra, separadas por saltos de línea.

Estructura y Componentes del Proyecto

La estructura del proyecto sigue las etapas tradicionales de un compilador/intérprete, complementadas ahora con una capa significativa de gestión de interfaz de usuario y niveles. Se observan las siguientes grandes áreas:

1. **Núcleo del Intérprete:** Análisis Léxico, Sintáctico, Semántico e Interpretación.
2. **Gestión de UI y Archivos:** Interacción del usuario, carga/guardado de código e imágenes.
3. **Gestión del Lienzo:** Inicialización y manipulación de la textura del lienzo.
4. **Gestión de Niveles y Temas:** Creación, validación y carga de niveles, y manejo de datos temáticos.

Núcleo del Intérprete

Esta sección se mantiene consistente con el análisis anterior, siendo el cerebro del procesamiento del lenguaje.

- **Análisis Léxico (Lexer.cs, Tokens.cs, FunctionRegistry.cs):**
    - **Lexer.cs**: Transforma el código fuente en tokens, con manejo de errores léxicos.
    - **Tokens.cs**: Define los tipos de tokens y la estructura de un token (TokenType, Token struct).
    - **FunctionDefinition.cs y FunctionRegistry.cs**: Definen y registran los comandos y funciones del lenguaje, incluyendo sus nombres, tipos de token, categorías, tipos de retorno y argumentos, lo que facilita la extensibilidad.
- **Análisis Sintáctico (Parsing) (Parser.cs):**
    - **Parser.cs**: Toma el flujo de tokens y construye un Árbol de Sintaxis Abstracta (AST), verificando la gramática y gestionando errores sintácticos con recuperación (Synchronize()). Se asegura que el código comience con Spawn.
- **Análisis Semántico (SemanticAnalyzer.cs, SymbolTable.cs):**
    - **SemanticAnalyzer.cs**: Recorre el AST para verificar la coherencia y el significado del programa. Realiza comprobaciones de tipos, verifica declaraciones de variables y etiquetas, y gestiona errores semánticos.
    - **SymbolTable.cs**: Gestiona las variables y etiquetas declaradas en el programa, rastreando sus tipos y detectando duplicados o referencias no definidas.
- **Interpretación (Interpreter.cs, InterpreterFunctions.cs):**

- **Interpreter.cs**: El motor de ejecución que recorre el AST. Mantiene el estado de Wall-E (posición, pincel) y el lienzo. Gestiona el flujo de control, incluyendo saltos GoTo a etiquetas. Incorpora manejo de errores en tiempo de ejecución, asegurando que los cambios previos al error permanezcan en el lienzo.
- **InterpreterFunctions.cs**: Extensión de Interpreter.cs, contiene la lógica específica para la ejecución de cada comando (ej., ExecuteSpawn, ExecuteDrawLine, ExecuteFill) y función (ej., ExecuteGetActualX, ExecuteIsBrushColor), interactuando directamente con el lienzo (Texture2D).

Gestión de UI y Archivos

Esta es una nueva capa crucial que permite la interacción del usuario con el intérprete.

- **UIManager.cs:**
  - **Rol Principal:** Es el centro de control de la interfaz de usuario principal de la aplicación. Hereda de MenuManager (no proporcionado, pero implícito para la gestión de escenas/menús).
  - **Elementos UI:** Gestiona referencias a RawImage para el lienzo (canvasDisplayImage), TMP_InputField para el editor de código (codeEditorInput), entrada de tamaño (sizeInput) y mensajes de estado/error (statusTextEditor).
  - **Botones y Funcionalidad:** Asigna listeners a botones como resizeButton, loadButton, saveButton, saveImageButton, executeButton, menuButton, levelSection, legendButton, y cleanButton.
    - OnResizeButtonPressed(): Redimensiona el lienzo según la entrada del usuario, con límites definidos (minCanvasSize, maxCanvasSize).
    - OnLoadButtonPressed(): Carga código desde un archivo .pw utilizando FileManager.
    - OnSaveButtonPressed(): Guarda el código del editor en un archivo .pw utilizando FileManager.
    - OnSaveImageButtonPressed(): Guarda el Texture2D del lienzo como una imagen JPG utilizando FileManager.

        - OnExecuteButtonPressed(): Reinicia el lienzo, obtiene el código del editor y llama al método ExecuteCode para iniciar el proceso de interpretación completo (léxico, sintáctico, semántico, ejecución).
        - LegendPanel(): Activa el panel de la leyenda.
        - Clean(): Limpia el área de mensajes de estado/error.
    - **Reporte de Estado y Error:** Los métodos ShowStatus() y ShowError() controlan la visualización de mensajes informativos y de error en statusTextEditor, utilizando colores (blanco para estado, rojo para errores) para diferenciarlos.
    - **Orquestación de la Interpretación:** El método ExecuteCode() orquesta todo el pipeline del intérprete, pasando el código fuente a Lexer, Parser, SemanticAnalyzer e Interpreter, y mostrando los errores de cada fase.
- **FileManager.cs:**
    - **Rol Principal:** Gestiona la interacción con el sistema de archivos del sistema operativo para cargar y guardar scripts y exportar imágenes.
    - **Dependencias:** Utiliza SFB (StandaloneFileBrowser) para abrir y guardar paneles de archivos, lo que es común en aplicaciones Unity de escritorio.
    - **Funcionalidades:**
        - LoadScriptFromFile(): Abre un panel para seleccionar un archivo .pw y lee su contenido.
        - SaveScriptToFile(): Abre un panel para guardar el texto del editor en un archivo .pw.
        - SaveTextureAsJPG(Texture2D texture): Abre un panel para guardar la Texture2D del lienzo como un archivo JPG.
    - **Reporte de Errores (FileManager):** Los métodos incluyen bloques try-catch para manejar IOException y otros errores de sistema de archivos, reportándolos al compiler (otro TMP_InputField en la UI, posiblemente el mismo statusTextEditor en UIManager).
- **CanvasController.cs:**
    - **Rol Principal:** Encargado de la inicialización y gestión de la Texture2D que sirve como lienzo de Wall-E.

- **Funcionalidad:**
  - InitializeCanvas(int size): Crea una nueva Texture2D del tamaño especificado, la configura para pixel art (FilterMode.Point, TextureWrapMode.Clamp) y la inicializa con píxeles blancos. Destruye la textura anterior si existe.
  - GetCanvasTexture(): Devuelve la Texture2D actual del lienzo.
  - GetCurrentSize(): Devuelve el tamaño actual del lienzo.
- **Integración:** Es utilizado por UIManager para la creación y reseteo del lienzo antes de cada ejecución o redimensionamiento.