



UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

GRADO EN INGENIERÍA INFORMÁTICA

Implementación de un juego de programación visual para la docencia

Autor:

Gonzalo Diez Garrido

Especialidad:

Computación

Director:

Antonio Chica Calaf

Departamento:

Ciencias de la Computación

17 de gener de 2017

Resumen

En estos tiempos, cada vez más gente está aprendiendo a programar, sobretodo niños. Sabiendo que los niños aprenden mejor mientras se divierten, ¿Por qué no hacer un videojuego para que los niños aprendan a programar visualmente mientras juegan?

Actualmente existen numerosas opciones que tratan de cumplir este objetivo. Por desgracia, estas opciones no favorecen a realizar el salto desde una programación visual a escribir código en un lenguaje específico.

ROGUES CODECRAFT nace tratando de cubrir ese salto. Se trata de un videojuego donde el usuario tendrá que programar el movimiento de su robot para que cumpla con el objetivo del nivel. Está programación se hace a base de arrastrar y soltar bloques de código del lenguaje de programación CODECRAFT, el lenguaje simple y expresivo diseñado para este proyecto.

La principal diferencia de este proyecto con sus competidores es que cambia el paradigma de ejecución. En las demás opciones, al ejecutar el programa del usuario, se ejecuta una sola vez y de manera que en cada instante de tiempo, solo se ejecuta una instrucción. En este proyecto, en cambio, en cada instante de tiempo se ejecuta el programa entero del cual se deriva en un movimiento del robot.

Abstract

In these times, more and more people are learning to program, especially children. Knowing that children learn best while having fun, why not make a video game so children learn to program visually while they play?

There are now many options that try to meet this goal. Unfortunately, these options do not favor making the jump from a visual programming to writing code in a specific language.

ROGUES CODECRAFT was born trying to cover that leap. It is a video game where the user will have to code the movement of his robot to meet the objective of the level. This programming is done by dragging and dropping blocks of code from the programming language CODECRAFT, the simple and expressive language designed for this project.

The main difference of this project with its competitors is that it changes the execution paradigm. In the other options, when executing the user program, it is executed only once and so that at each time instant, only one instruction is executed. In this project, however, in each time instant runs the entire program from which it is derived in a movement of the robot.

Resum

En aquests temps, cada vegada més gent està aprenent a programar, sobretot nens. Sabent que els nens aprenen millor mentre es diverteixen, Per què no fer un videojoc perquè els nens aprenguin a programar visualment mentre juguen?

Actualment existeixen nombroses opcions que tracten de complir aquest objectiu. Malauradament, aquestes opcions no afavoreixen a fer el salt des d'una programació visual a escriure codi en un llenguatge específic.

Rogues CODECRAFT neix tractant de cobrir aquest salt. Es tracta d'un videojoc on l'usuari haurà de programar el moviment del seu robot perquè compleixi amb l'objectiu del nivell. Està programació es fa a força d'arrossegar i deixar anar blocs de codi del llenguatge de programació CODECRAFT, el llenguatge simple i expressiu dissenyat per a aquest projecte.

La principal diferència d'aquest projecte amb els seus competidors és que canvia el paradigma d'execució. En les altres opcions, en executar el programa de l'usuari, s'executa un sol cop i de manera que a cada instant de temps, només s'executa una instrucció. En aquest projecte, en canvi, a cada instant de temps s'executa el programa sencer del qual es deriva en un moviment del robot.

Índice

1	Introducción	6
1.1	Contextualización	6
1.2	Actores implicados	7
1.2.1	Desarrollador del Proyecto	7
1.2.2	Director del Proyecto	7
1.2.3	Usuarios principales	7
1.2.4	Otros posibles usuarios	8
1.3	Estado del Arte	9
1.3.1	Lenguajes de programación visual	9
1.3.2	Lenguajes de programación didácticos	10
1.3.3	Videojuegos de programación	10
2	Descripción del proyecto	12
2.1	Objetivos	12
2.2	Descripción del robot y su contexto	12
2.3	Descripción del lenguaje de programación CODECRAFT	12
2.4	Descripción del editor visual de CODECRAFT	13
2.5	Descripción del compilador	14
2.6	Descripción del intérprete	14
2.7	Descripción del editor de niveles	14
2.8	Descripción de los niveles del juego	14
3	Diseño e Implementación	15
3.1	CODECRAFT	15
3.1.1	Gramática	15
3.2	BlockLoader	17
3.2.1	Compilador	17
3.3	Bloques de código	18
3.3.1	BlockCode	18
3.3.2	BlockCodeNumeral	24
3.3.3	BlockCodeWithScope	24
3.3.4	BlockCodeIfElse	27
3.3.5	BlockCodeFunct	27
3.4	Intérprete	28
3.5	Gestión de memoria	30
3.6	Sistema de escenas	31
3.6.1	SceneMenu	32
3.6.2	SceneCode	32
3.6.3	StageUI	33

3.6.4	MenuLevelComplete	33
3.7	Editor visual de CODECRAFT	35
3.7.1	Grupos de bloques del lenguaje	36
3.7.2	Grupos de variables del programa	36
3.8	Sistema de niveles	38
3.8.1	Robot	39
3.8.2	Map	39
3.8.3	Controlador de estados	40
3.9	Editor de niveles	41
3.10	InputManager	42
3.11	MagicView	42
3.12	User Interface	43
3.12.1	BorderSprite	43
3.12.2	Button	43
3.12.3	TextField	44
3.13	Controlador de recursos	45
4	Posibles ampliaciones	46
5	Gestión de proyecto	47
5.1	Objetivos	47
5.2	Alcance y Posibles Obstáculos	48
5.3	CODECRAFT	48
5.4	Editor visual de CODECRAFT	48
5.5	Motor del videojuego	49
5.6	Diseño de niveles	49
5.7	Metodología y Rigor	50
5.8	Descripcion de las tareas	51
5.8.1	Viabilidad del proyecto deseado	51
5.8.2	Planificación del proyecto (Hito inicial)	51
5.8.3	Configuración inicial	51
5.8.4	Desarrollo del proyecto como tal	52
5.8.5	Hito final	54
5.9	Tiempo aproximado para cada tarea	54
5.10	Recursos utilizados	55
5.10.1	Software	55
5.10.2	Hardware	55
5.11	Diagrama de Gant	55
5.12	Plan de Acción	55
5.13	Gestión Económica	56
5.13.1	Hardware	56

5.13.2	Software	56
5.13.3	Recursos Humanos	57
5.13.4	Total	59
5.14	Control de desviación	60
5.15	Sostenibilidad	61
5.15.1	Social	61
5.15.2	Económica	61
5.15.3	Medioambiental	62

1 Introducción

En esta sección describo brevemente en que consiste el proyecto, su contexto y sus implicaciones, y aprovecho para definir algunos conceptos que se utilizan en secciones posteriores, así como para mencionar a los actores implicados en el mismo.

1.1 Contextualización

Un videojuego es un juego electrónico que se juega sobre un dispositivo computarizado, como puede ser en un ordenador personal, una videoconsola o un teléfono móvil, entre otros. Actualmente es uno de los medios de entretenimiento más extendidos, y para muchas personas se ha convertido incluso en su estilo de vida[1]. Es una industria que mueve miles de millones de dólares[2] al año, superando incluso a la del cine y la música.

Los primeros videojuegos nacieron en la década de los 50, y se programaban sobre osciloscopios [7]. La industria fue evolucionando rápidamente, pasando por las primeras Arcades y videoconsolas en la década de los 70, los primeros ordenadores personales en la década de los 80, y las videoconsolas portátiles en los 90 (siendo el máximo exponente la GameBoy[4])[5]. Con el nuevo milenio, llegó la masificación de los smartphones[3], abriendo un mercado ideal para la industria de los videojuegos, por la gran cantidad de usuarios y la facilidad de acceso. A finales de la misma década y gracias a plataformas de distribución digital como Steam, se consolidó el concepto de "indie game developers" [6], que se refiere a equipos de desarrollo muy pequeños, que no están dirigidos por ningún productor, que están menos condicionados y son por tanto más fieles a sus propias ideas, y que a menudo desarrollan para perfiles de usuario muy concretos, en contraposición a las grandes compañías que apuntan a las ventas y al gran público.

En cierto modo, el presente proyecto pertenece a la clasificación Indie. Sin embargo, cabe remarcar que se buscan también otros objetivos más allá del puro entretenimiento. También se pretende introducir al usuario en los conceptos básicos de programación y lógica. Esto se consigue mediante un lenguaje de programación visual, que es la herramienta que permitirá al usuario superar los niveles del juego. Estos niveles consisten en puzzles que hay que resolver, y la solución es justamente un programa expresado usando bloques que tienen un significado computacional. Gracias a una curva de aprendizaje bien elaborada, cada nivel irá enseñando o afianzando algún concepto de programación o lógica.

Por lo mencionado anteriormente, podemos subdividir el proyecto en dos partes principales. En primer lugar, tenemos el diseño del lenguaje de programación visual, que

denominamos CODECRAFT, el diseño e implementación del editor visual para programar cómodamente en CODECRAFT, y el correspondiente compilador e intérprete. En este sentido, cabe aclarar que el lenguaje visual, puede por un lado ser traducido a texto plano y viceversa, y por otro lado puede ser interpretado y ejecutado directamente. En segundo lugar, tenemos el diseño e implementación del juego propiamente dicho, al cual llamamos ROGUES CODECRAFT, un videojuego top-down (visión cenital) 2D por turnos en el cual el robot comandado por un programa CODECRAFT tiene que llevar a cabo con éxito las misiones que se le encomiendan, tales como alcanzar una posición determinada, o actuar sobre el entorno.

1.2 Actores implicados

El desarrollo de este proyecto implica a los siguientes actores:

1.2.1 Desarrollador del Proyecto

El proyecto se desarrollará por una sola persona. Yo mismo llevaré a cabo el diseño de CODECRAFT, el diseño e implementación de su editor, así como de su intérprete y compilador. También diseñaré e implementaré el motor de ROGUES CODECRAFT, y las herramientas para el diseño de niveles. Asimismo, utilizaré dicha herramienta para crear los niveles del juego. También quedan a mi cargo las tareas de escribir la documentación y validación.

1.2.2 Director del Proyecto

El director del proyecto es Antonio Chica Calaf, profesor del departamento de Ciencias de la Computación de la Universitat Politècnica de Catalunya, que está especializado, entre otras cosas, en gráficos y desarrollo de videojuegos. Él será el encargado de guiarme durante el desarrollo del proyecto, y sus consejos se enfocarán mayormente en el diseño y la implementación de ROGUES CODECRAFT, aunque también puede contribuir de forma significativa en el diseño de CODECRAFT.

1.2.3 Usuarios principales

Mi objetivo es hacer un juego que permita aprender los conceptos básicos de programación de una forma divertida y accesible. Los usuarios potenciales son personas que no tienen conocimientos de programación ni nociones de lógica, pero sí muchas ganas

de divertirse, aprender, y afrontar nuevos retos. En líneas generales está enfocado a niños de entre 8 y 16 años, pero también podrían utilizarlo adultos.

1.2.4 Otros posibles usuarios

Además de los usuarios que usen el videojuego para aprender a programar, también hay otros usuarios objetivo. Nos referimos a aquellos que sabiendo o no programar, les interesa la resolución y creación de puzzles. Para estos usuarios me planteo crear un servidor donde puedan subir sus propios niveles, y que los demás usuarios se los puedan descargar.

1.3 Estado del Arte

En esta sección tratamos de clasificar nuestro proyecto dentro de diversos ámbitos, y mencionamos otras herramientas y juegos existentes que guardan relación con el nuestro. Por la tipología del proyecto, mencionaremos brevemente algunos lenguajes de programación visual, y también lenguajes de programación diseñados con fines educativos. Asimismo, comentaremos algunos juegos conocidos de programación, y también juegos de visión cenital y de resolución de puzzles.

1.3.1 Lenguajes de programación visual

Para evitar la intersección con la siguiente sección, mencionaré aquí algunas herramientas de programación visual que no tienen propósitos educativos. De las muchas existentes, quiero destacar algunas de ellas que son bien conocidas en el ámbito de los videojuegos.

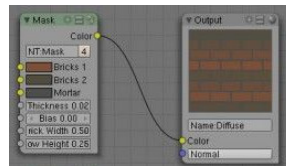


Figura 1: Blender

Blender (f 1) es un editor que permite describir programas de sombreado mediante grafos. Mayormente se utiliza para crear modelos 3D y añadirles texturas. **GameMaker studio** permite la creación de videojuegos de una forma sencilla a base de arrastrar y soltar bloques que realizan tareas concretas predeterminadas. Está pensado para desarrollar muy rápidamente videojuegos con muy pocos conocimientos de programación.

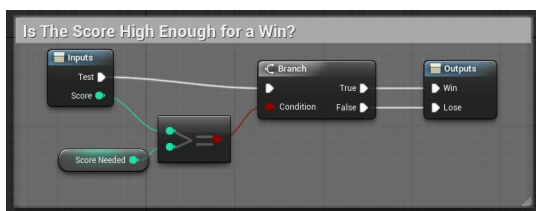


Figura 2: Blueprint de Unreal Engine 4

Unreal Engine 4 (Figura 2) también ofrece un lenguaje de programación visual llamado Blueprints que permite definir comportamientos mediante grafos.

Mi lenguaje **CODECRAFT** difiere de los anteriores en el sentido de que es más bien un lenguaje de programación convencional simplificado y al que se le ha dado un aspecto gráfico con el fin de hacerlo más intuitivo y fácil de usar.

1.3.2 Lenguajes de programación didácticos

Lenguajes de programación diseñados con fines didácticos hay muchos. Dos de los más importantes serían los siguientes: **Scratch** (Figura 3) es el máximo exponente en lenguajes de programación didácticos, además de ser un lenguaje de programación visual. Este lenguaje te permite programar el comportamiento de diferentes objetos y ejecutar código en paralelo. Al igual que **CODECRAFT**, usa un sistema de visualización a base de bloques con significado computacional, que se unen para crear el programa. **Lightbot** (Figura 4), al igual que el anterior lenguaje, usa bloques con significado computacional. En cambio, en este caso, los bloques no se pueden unir, sino que se concatenan para crear un flujo de actividad. **Lightbot** está diseñado para niños, usando imágenes en los bloques en lugar de palabras clave o keywords.

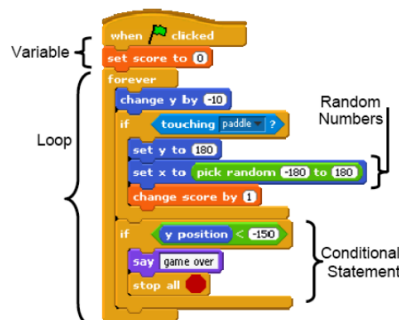


Figura 3: Programa Scratch

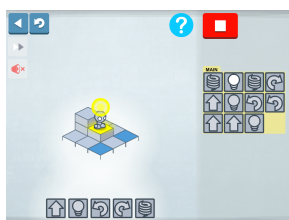


Figura 4: Nivel de Lightbot

CODECRAFT es muy parecido al **Scratch**, cambiando el paradigma de ejecución. El programa **CODECRAFT** se ejecuta completamente cada turno, y de dicha ejecución se procesan las instrucciones que se le dan al robot para que las ejecute y las muestre gráficamente. En el **Scratch**, así como en el **Lightbot** cada turno de la partida se ejecuta un solo bloque. Además, el programa **CODECRAFT** se podrá ejecutar en modo debug, ejecutando bloque por bloque y viendo como se van modificando el valor de las variables, así como por que camino elige en las estructuras de control. De esta manera, no se perderá la parte intuitiva de observar que hace cada instrucción, y a la vez le da más potencia de la que tienen los lenguajes anteriores.

1.3.3 Videojuegos de programación

CodeCombat es un videojuego con el objetivo de enseñar a programar al usuario. Este juego consiste en superar niveles a base de programar el movimiento del usuario, la peculiaridad del cual es que te deja programar en cuatro lenguajes diferentes, ya sea python, javascript, ruby o lua. Los siguientes dos ejemplos, en cambio, han optado por crearse su propio lenguaje de programación. **TIS-100** está enfocado a la programación textual con instrucciones muy simples en una especie de lenguaje ensamblador súper simplificado que se ejecuta en paralelo sobre diferentes nodos.

Human Resource Machine (Figura 5), al igual que el anterior título, elige como lenguaje de programación el ensamblador con un grupo de instrucciones muy básicas, pero esta vez programando de una forma visual, a base de arrastrar y soltar las instrucciones, viendo el resultado de la ejecución de tu programa sobre un avatar que realiza acciones.

ROGUES CODECRAFT en cambio, además de usar la programación como método de controlar tu personaje como los anteriores, ejecuta el programa del usuario completamente cada turno de la partida, en contraposición a los ejemplos anteriores, que cada turno ejecutan una sola instrucción. Esto le da una potencia al usuario que en los demás juegos no la tienen, ya que le permite hacer una inteligencia artificial que reaccione a los diferentes estímulos en cada turno de la partida. Además, **ROGUES CODECRAFT** usa un lenguaje de programación de alto nivel que a la vez simplificado, haciendo que sea fácil de aprender y a la vez con la potencia suficiente para resolver tareas complicadas sin un esfuerzo excesivo.



Figura 5: Nivel de Human Resource Machine

2 Descripción del proyecto

2.1 Objetivos

El objetivo principal de este proyecto es crear un videojuego educativo con el que los usuarios aprendan conceptos de programación. Para poder superar cada nivel del juego, el usuario deberá ofrecer el programa adecuado, de modo que, ese programa, guíe los movimientos de un robot hasta una localización deseada.

De lo anterior se sigue que habrá que definir las posibles instrucciones que pueda recibir el robot y en qué contexto se encuentra, y también cómo son los programas que generarán esas instrucciones. Para facilitar las tareas de programación al usuario, también convendrá desarrollar un editor visual de programas. Y, por supuesto, va a ser necesaria la implementación de un interprete capaz de ejecutar esos programas y comunicar las ordenes al robot, así como la propia implementación del movimiento del robot. Como tarea complementaria, también será conveniente desarrollar un editor de niveles, de modo que su diseño sea fácil y rápido.

En las siguientes secciones describo en detalle cada uno de estos objetivos parciales.

2.2 Descripción del robot y su contexto

Conceptualmente, el robot se mueve en un mundo 2D, que es esencialmente una matriz bidimensional de celdas. Cada una de esas celdas puede ser, o bien un muro (y por tanto de acceso prohibido), o bien un agujero (y por tanto un lugar donde el robot muere), o bien un camino (y por tanto de acceso permitido y sin perjuicios para el robot). Hay una celda especial de tipo camino, que es la celda objetivo para el robot. A cada instante de tiempo, el robot ocupa una de las celdas, y tiene también una orientación, apuntando en una de las cuatro direcciones principales de la matriz (norte, sur, este, oeste). Asimismo, en cada instante de tiempo el robot puede, o bien desplazarse a una celda adyacente en la dirección elegida, o bien modificar su orientación mediante una rotación. Simultáneamente, el robot puede comprobar el tipo de alguna de las celdas adyacentes, con el fin de saber si puede caminar por ellas sin peligro.

2.3 Descripción del lenguaje de programación CODECRAFT

Los programas del lenguaje CODECRAFT son los que deben indicarle al robot qué decisiones tomar a cada instante de tiempo. Por lo tanto, de entre las instrucciones

básicas de CODECRAFT se encuentran las ordenes explícitas de desplazamiento y comprobación del robot. Además de estas, CODECRAFT consta de las típicas instrucciones de control de flujo y asignación, variables y operadores. Además, CODECRAFT tiene tres tipos básicos, que son número (entero), booleano y dirección (que tiene por valores asociados las posibles direcciones absolutas o relativas del robot: norte, sur, este, oeste, adelante, atrás, derecha, izquierda).

Un programa CODECRAFT se ejecuta íntegramente a cada instante de tiempo del robot. Por tanto, la ejecución del programa dá lugar a como mucho un desplazamiento o cambio de orientación del robot, y un posible cambio de estado de las variables del programa, que condicionará el resultado de las ejecuciones íntegras posteriores. Ésta es la principal diferencia con respecto a otros lenguajes de programación visual educativa como SCRATCH, en el que se realiza una única ejecución del programa completo, y se ejecuta una sola instrucción del programa por cada instante de tiempo.

CODECRAFT deberá ser lo suficientemente simple para que sea fácil de asimilar, pero lo suficientemente expresivo para solucionar puzzles complicados a los que se pueda enfrentar el robot.

2.4 Descripción del editor visual de CODECRAFT

Un programa CODECRAFT se puede ver como un árbol cuyos nodos están etiquetados con nombres de instrucción, operadores, y expresiones básicas. Llamaremos bloques a dichos nodos del árbol, por su parecido a un bloque de LEGO. Esto es especialmente relevante para el editor del lenguaje, cuyo objetivo es facilitar la programación, es decir, la construcción de uno de esos árboles.

El editor visual de CODECRAFT permite al usuario programar a base de arrastrar y soltar bloques convenientemente, de modo que se van enganchando para formar árboles. De este modo, se elimina la posibilidad de cometer errores léxicos, sintácticos. Pero, de hecho, también elimina la posibilidad de cometer errores semánticos, ya que, por ejemplo, impide la construcción de un árbol que implique una incompatibilidad de tipos. Por lo tanto, el resultado de un árbol completo construido con el editor va a ser necesariamente un programa correcto. Otro tema es que pueda no resolver correctamente un puzzle.

El editor debe facilitar la selección de los distintos tipos de bloques, y ofrecer funcionalidades adicionales tales como copiar, pegar y borrar subárboles. Nuestro diseño del editor está ligeramente inspirado en el de SCRATCH.

2.5 Descripción del compilador

El compilador tiene por objetivo traducir programas CODECRAFT entre su representación visual como árboles, y su representación textual, básicamente, con el fin de almacenar aquellos programas parcialmente construidos que se le ofrecerán al usuario en cada nivel como punto de partida a completar.

2.6 Descripción del intérprete

El intérprete ejecuta el árbol de un programa CODECRAFT, y mientras lo hace, comunica al robot cuales son las instrucciones que ese programa le ordena. Con el fin de permitir una ejecución controlada, instrucción por instrucción, se plantea el diseño del intérprete mediante el uso de una pila de instrucciones que mantenga cuales son las instrucciones que se encuentran a medio terminar. Esencialmente, el contenido de la pila será la secuencia de los bloques que llevan desde la raíz hasta el bloque actual en ejecución.

2.7 Descripción del editor de niveles

El editor de niveles debe permitir definir una matriz de celdas, de tamaño arbitrario, de los distintos tipos que se permiten en el juego, así como indicar la celda inicial y la celda objetivo del robot. También debe permitir definir el programa (incompleto) que se le ofrece al usuario después para completar.

2.8 Descripción de los niveles del juego

Los niveles del juego deberán implicar una curva de aprendizaje que no frustre al jugador con una gran dificultad en poco tiempo, pero que tampoco lo desmotive por ser resultar repetitivo. Cada nivel deberá tener como objetivo enseñar o afianzar algún concepto. Seguiremos una metodología iterativa, rediseñando los niveles y su progresión hasta que estemos contentos con el resultado.

3 Diseño e Implementación

3.1 CODECRAFT

El lenguaje CODECRAFT es el lenguaje de programación visual diseñado para este proyecto. Este lenguaje se ha diseñado teniendo siempre en mente la idea de que tiene que ser simple y a la vez expresivo. La Simplicidad es necesaria teniendo en cuenta que nuestro público objetivo son personas de corta edad que quieren aprender a programar jugando. La expresividad es necesaria para poder representar soluciones de puzzles complicados.

Esencialmente, el usuario programa a base de construir un árbol, cuyos nodos están etiquetados con instrucciones, operadores, y valores constantes, tales como identificadores, números, booleanos, y direcciones (del movimiento o rotación del robot). Durante el proceso de construcción del árbol, vamos a contar con un programa incompleto, pues habrá nodos y subárboles pendientes de definir. De hecho, al usuario se le ofrecerá un programa incompleto como punto de partida para terminar de construir una solución. Así pues, en la siguiente sección describiremos tanto los programas completos como los incompletos.

3.1.1 Gramática

Describimos a continuación la gramática de los programas CODECRAFT correctos. Simultáneamente, aprovechamos también para definir la gramática de los programas CODECRAFT incompletos, pues su gramática es muy parecida a la de los correctos. En nuestra descripción, hay que eliminar los corchetes “[” y “]” para obtener la gramática de los programas correctos, y en cambio, hay que eliminar los “[” y reemplazar los “]” por “?” para obtener la gramática de los programas incompletos. Como se puede ver, la diferencia reside en que, en el segundo caso, se admite que algunos subárboles de los AST construidos sean vacíos.


```

    program  → var_defines main
      main   → instruction*
    var_defines → instruction*
    instruction → “if” expression “{” instruction* “}” (“else” “{” instruction* “}”)? |
                ”while” expression “{” instruction* “}” |
                “(” [IDENTIFIER] “)” “=” expression “;” |
                “Move” expression “;” | “Rotate” expression “;”
    expression → “(” [inner_expression] “)”
    inner_expression → expression binary_operator expression |
                    unary_operator expression |
                    IDENTIFIER | NUMBER | “True” | “False” |
                    “Forward” | “Backwards” | “Left” | “Right” |
                    “North” | “South” | “East” | “West”
    binary_operator → “+” | “-” | “==” | “<=” | “and” | “or”
    unary_operator  → “not” | “Walkable”

```

En CODECRAFT hay tres tipos de datos, todos ellos básicos, que son número entero, booleano y dirección. El rango de los números queda limitado por su representación con 32 bits. En el caso de los booleanos y las direcciones, tan solo se pueden representar ciertos valores predefinidos de antemano, que son **True** y **False** para los booleanos, y **Forward**, **Backward**, **Left**, **Right**, **Up**, **Down**, **East** y **West** para las direcciones. Los tipos de datos de cada variable se infieren según el valor que les es asignado. Cabe remarcar que, en el entorno visual, sí hay que indicar cual es el tipo de las variables mediante el recuadro correspondiente.

Las secuencias de palabras generadas por la gramática anterior corresponden a los programas correctos (quizás incompletos) representados en modo texto. Los correspondientes AST (Abstract Syntax Trees) de esos programas, corresponden a la representación visual de los mismos que realiza el editor. Esa representación en forma de árbol es también la que se utiliza para llevar a cabo la ejecución del programa. El compilador que hemos implementado en este proyecto es quien se encarga de realizar la traducción entre una y otra representación.

3.2 BlockLoader

Esta parte del programa es la encargada de instanciar todos los bloques de código que componen el lenguaje CODECRAFT. Además, se encarga de inicializar los `BlockGroupDisplayers` que estarán dentro del `BlocksManager`.

Por otro lado, también contiene el código del compilador de CODECRAFT.

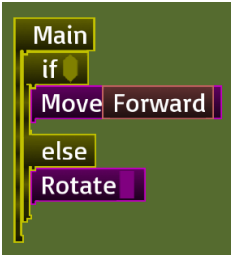
3.2.1 Compilador

Un compilador traduce código escrito en un lenguaje a otro lenguaje, sin cambiar el significado del programa. En nuestro caso, traduce texto en formato CODECRAFT a código interactivo hecho a base de `BlockCodes`.

Este compilador se diseñó para que pudiera ir ampliándose automáticamente a medida que iba evolucionando el lenguaje CODECRAFT, ya que no sabíamos como queríamos que fuera el lenguaje en su versión final.

Es por esto que se creó un script en `python` que parseaba la parte donde se instanciaban todos los bloques de código y generaba el código en `C++` que compila instrucciones y expresiones.

Una característica inusual en nuestro compilador es que permite compilar programas que estén incompletos. Gracias a esto, podemos guardar y cargar código incompleto en el archivo de guardado de un nivel, ayudando a que la curva de aprendizaje sea mucho menos pronunciada, al poder ofrecerle al usuario programas parcialmente incompletos en un inicio para introducir nuevos conceptos.

	<pre>Main { if () { Move (Forward); } else { Rotate (); } }</pre>
---	---

3.3 Bloques de código

Los bloques de código es la representación lógica y visual de cada palabra generada por la gramática del lenguaje. Como ya hemos descrito, no todas las construcciones de la gramática siguen la misma estructura. Es por esto que necesitamos definir multiples clases de bloques de código.

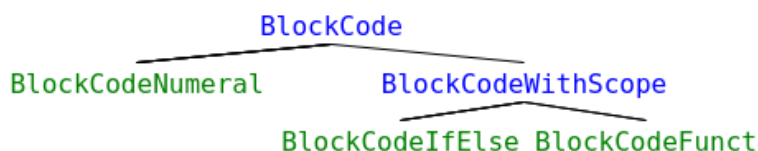


Figura 6: Jerarquía de clases

Tal y como se puede ver en la figura 6, siguen la siguiente jerarquía.

3.3.1 BlockCode

El BlockCode es la entidad más compleja del programa, conteniendo aproximadamente el 10 % del código global (contabilizado en número de líneas). Esta clase se encarga de representar lógica y gráficamente lo que sería un átomo de CODECRAFT.

El diseño de esta estructura de datos fue realizado teniendo siempre en mente que se buscaba el poder crearlo y modificarlo en tiempo de ejecución, por el usuario. Además se buscó que pudiera ser lo más general posible, tratando de tener que usar la herencia lo menos posible.

El objetivo del blockCode es que, además de representar gráficamente un átomo del lenguaje, se pueda ejecutar cierto código por cada bloque. Este código que se ejecuta dentro del bloque es código C++, ejecutado en el mismo programa.

Se podría haber traducido el CODECRAFT a otro lenguaje, ejecutarlo y luego recoger la salida dentro del programa para mover al robot.

Se decidió que se quería hacer la ejecución en el mismo programa para tener la posibilidad de poder ejecutar paso a paso el programa en CODECRAFT y de esta manera poder debugarlo.

Al tener que crear muchos tipos de bloques diferentes (uno por cada átomo del lenguaje) y querer que contuviera C++ nativo, teníamos dos opciones: O hacer una clase por cada bloque que heredara de BlockCode, implementando unicamente una funcion, o usar funciones anónimas para definir el código que simularía el bloque.

Finalmente nos decidimos por la segunda opción, ya que, aunque las funciones anónimas sean más complejas entender y la legibilidad del código empeore, nos permiten definir cantidad de bloques diferentes sin tener que hacer clases nuevas, haciendo que el código no crezca sin necesidad.

Estructura de datos

Un conjunto de `BlockCodes` relacionados entre ellos se representa como un árbol doblemente enlazado donde la raíz es el bloque que gráficamente se representa más arriba a la izquierda. Se podría decir que es un AST, con la diferencia que en lugar de tener nodos que sean `statement sequencics`, cada instrucción es madre de, además de sus argumentos, de la instrucción que le sigue en el caso de que exista una.

Para representar lógicamente esta estructura de datos, tenemos en cada nodo los siguientes punteros:

- **parent**: En el caso de que el bloque actual sea el argumento de otro bloque, apuntará a este otro bloque.
- **beforeSts**: Apunta a la instrucción anterior a esta en caso de que haya una.
- **nextSts**: Apunta a la instrucción siguiente.
- **blockArguments**: Por cada posible argumento que tenga este bloque, guardará un puntero.

Como se habrá podido intuir, si un bloque es el argumento de otro, es decir, **parent** es un puntero a un bloque válido, este bloque nunca podrá tener instrucciones antes o después, ya que no será una instrucción. Análogamente, un bloque que sea una instrucción nunca podrá tener **parent**.

Por tanto, los bloques que son de tipo instrucción(7) nunca tendrán **parent**, pero sí podrán tener **beforeSts** y **nextSts**. En cambio, los bloques que sean de tipo número, booleano (8) o dirección únicamente podrán tener **parent**.

Destrucción

Como ya hemos dicho antes, esta estructura de datos representa un árbol con raíz. Es por esto que definimos a un nodo como dueño de los nodos que cuelgan de él, es decir, los nodos que son apuntados por **nextSts** y **blockArguments**. De esta manera, en el momento de destruirse un bloque, destruirá todos los que dependan de este.

Como el árbol está doblemente enlazado, cuando se borra un nodo, si este está enlazado a uno superior, quedará un puntero a una posición de memoria de un objeto destruido. Es por esto que antes de borrar un bloque, este se emancipa. Esto quiere

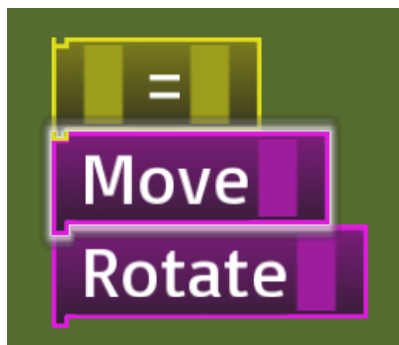


Figura 7: El bloque de tipo instrucción 'Move' tiene a otros dos bloques de tipo instrucción como `nextSts` y `beforeSts`



Figura 8: El bloque de tipo booleano 'and' tiene como parent el bloque 'not'

decir que se separa del bloque que pueda ser su `parent` o `beforeSts`, haciendo que el puntero que le apuntaba a él mismo, ahora contenga un `nullptr`.

Esta emancipación se hace para no dejar punteros inconsistentes, pero en el caso de que el `parent` o `beforeSts` ya se estuviera destruyendo, no será necesario esa consistencia, ya que esa memoria se va a liberar y esos punteros ya no se volverán a usar. Es por esto que se define el atributo `dead`, usado para decidir si es necesario emanciparse en el momento de destruirse.

Estructura de datos interna

La estructura interna ha sido diseñada con el objetivo de que sean dinámicos. Es decir, que en tiempo de ejecución se puedan modificar sus partes, ya sea el número de argumentos o el texto, incluso cambiando de color y texturas (Vease ampliación 4). En este proyecto se incluye únicamente una temática de bloques, pero en el caso de haber tenido algún artista disponible, se hubieran creado más, ya que el código lo permite.

Con este objetivo en mente, un bloque se describe con un vector de `blockParts` llamado `blockDescription`, un vector de strings (`blockStrings`) y un vector de punteros a argumentos (`blockArguments`). Estos `blockParts` representan cada parte de los cuales se compone un bloque. Pueden representar:

- `leftOutside`: Lado izquierdo del bloque.
- `rightOutside`: Lado derecho del bloque.
- `insideFull`: Parte dedicada a poner texto.
- `insideBool`: Argumento de tipo Booleano.
- `insideNumber`: Argumento de tipo Número.
- `insideStatement`: Argumento de tipo Instrucción. (Sin uso por ahora, ya que las funciones no son ciudadanos de primer orden en este lenguaje).
- `insideDirection`: Argumento de tipo Dirección.
- `insideEmpty`: Argumento que puede ser de cualquier tipo y es usado únicamente en bloques que representa operaciones binarias. En el momento que uno de los dos argumentos del bloque gana un tipo, el otro se convierte en ese mismo tipo. Podemos observarlo en el ejemplo de la figura 9.

Por cada texto, guardaremos el string en `blockStrings`, al igual que por cada argumento, guardaremos un `nullptr` en `blockArguments`



Figura 9: (i) tipo genérico (ii) tipo numeral (iii) tipo booleano

En la figura 10 podemos apreciar cuatro bloques de ejemplos, cada uno de un tipo diferente. Para que visualmente sea fácil de identificar los tipos, se representan con formas laterales diferentes.



Figura 10: (s) Statement (d) Direction (n) Number (b) Boolean

Una vez que tenemos construido el vector que describe la estructura del bloque, creamos la estructura donde se almacena toda la información necesaria para poder dibujar el bloque correctamente. Esta estructura se compone por diversos vectores:

- `parts`: Este es el vector de sprites. Tiene el mismo tamaño que el vector de `blockParts` tratado anterior, y es usado para dibujar cada parte del bloque.

- **partsOutlining**: Al igual que el anterior, tiene un elemento por cada parte del bloque y sirve para dibujar el contorno del bloque en el momento de selección.
- **partsActionOutlining**: Este vector tendrá un elemento por cada argumento que tenga el bloque. No sirve para poder delimitar el contorno del slot que tiene un argumento. En la figura 11 podemos ver un ejemplo.
- **blockTexts**: Representa los textos gráficamente usando los strings de `blockStrings`.



Figura 11: Dibujando el contorno del argumento

Esta creación de la estructura que almacena la información se hace en tres fases bien diferenciadas.

La primera fase, llamada `updateBlockStructure`, en la cual se crean todos los vectores anteriores desde cero, rellenandolos a partir de los vectores descriptores. Esta función es llamada cada vez que cambia algo de la estructura principal, como pudiera ser que se añadan slots de argumentos o partes con texto. También se llama cuando se cambia el color o la temática del bloque.

La segunda fase, llamada `updateInsideShapes`, se encarga de darle los tamaños a las partes del bloque que dependen de otras cosas para determinar su tamaño, como pueda ser la parte del bloque donde se va a dibujar el texto, que depende del tamaño del string, o el tamaño del slot de un argumento, que depende del tamaño del argumento en si. Esta función es llamada cada vez que se cambia un argumento o un texto, así como cuando un argumento hace algún cambio interno, ya que si un argumento cambia de tamaño, el bloque deberá adaptarse a este.

La tercera fase y última fase, llamada `updateShape`, es la encargada de posicionar todas las partes en su sitio, incluidos los bloques que puedan ser argumentos de este. Esta fase es llamada justo después de la fase anterior.

En la figura 12 podemos observar como un cambio en cierto `BlockCode` hará que se llame al `updateInsideShape` recursivamente hasta llegar a un bloque que sea una instrucción o a un bloque que no tenga `parent`.

Estados booleanos

Puede tener diferentes estados, dependiendo de, por ejemplo, las diferentes acciones

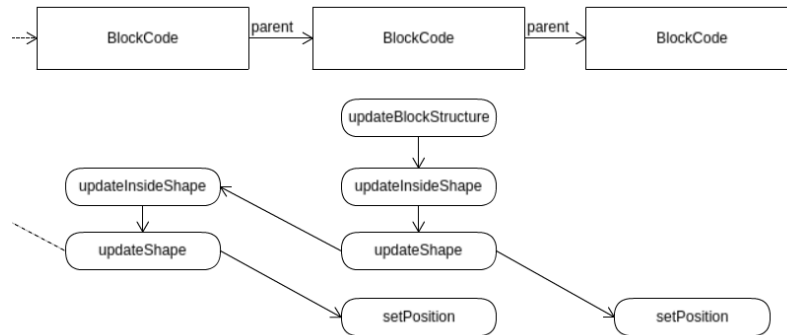


Figura 12: Diagrama de flujo de las tres fases

que pueda estar haciendo el usuario, y que no tienen porque ser exclusivos entre ellos. Estos estados son:

- **init**: Indica si hay que actualizar el contenido visual del bloque al añadir o quitar una parte de este. En el caso de que no queramos ver como se le van introduciendo partes al bloque, este booleano será falso, ahorrandonos actualizaciones innecesarias de la estructura de datos interna.
- **dead**: Como ya explicamos en la sección de **Destrucción**, determina si un bloque está destruyéndose o no.
- **outlining**: Indica si el ratón está encima de ese elemento, haciendo que en el momento del dibujado aparezca un delineado sobre el contorno del bloque. Esto nos sirve para que el usuario sepa que estaría seleccionando en el caso de que clicara.
- **hardOutlining**: Tiene el mismo efecto que el atributo **outlining**, pero este es independiente de la posición del ratón. Es usado, por ejemplo, por el interprete para designar que instrucción se está ejecutando en cada momento. (ver Ampliaciones (4))
- **warning**: Tal y como dice el nombre, advierte al jugador haciendo que el delineado del bloque se vuelva rojo. Es usado en el momento de comprobar si un bloque puede ser ejecutado.
- **selected**: En el caso de estar activo, en el momento de dibujar el bloque, se le aplicará un clareado para producir el efecto de que brilla. De esta manera el usuario sabrá que el bloque está seleccionado.

3.3.2 BlockCodeNumeral

En nuestro lenguaje de programación existen constantes numéricas, y queremos que el usuario pueda usarlas lo más fácilmente posible. Ya que, normalmente cuando se usa una constante numérica, se va a querer cambiar su valor sin necesidad de cambiar su posición en el programa, se pensó que había que darle esta facilidad al usuario.

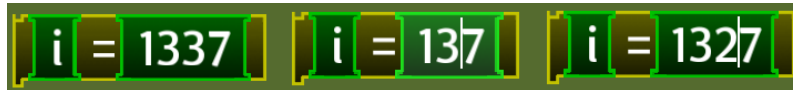


Figura 13: Edición de una constante numérica sin necesidad de moverlo

Es por esto que `BlockCodeNumeral` hereda de `BlockCode`, siendo un bloque de tipo numérico que representa una constante, y el cual permite cambiarle su valor al clicarlo encima, sin necesidad de moverlo o de crear uno nuevo.

Para conseguir el efecto deseado, insertamos un editor de texto (`TextFieldNumeric`) que usa el texto mostrado en el bloque para representarse gráficamente, reutilizando el editor de texto que ya teníamos implementado con anterioridad.

3.3.3 BlockCodeWithScope

Uno de los objetivos principales del lenguaje de programación es que fuera expresivo. Por tanto necesitábamos incluir estructuras de control. Las estructuras de control finalmente elegidas fueron `if`, `if else` y `while`.

Una estructura de control es similar respecto a una instrucción cualquiera en el contexto del flujo del programa: tiene una instrucción que viene antes y cuando acaba de ejecutarse le pasará el testigo a la instrucción que tenga debajo en el caso de que esta exista. Por otro lado, la peculiaridad es que puede haber código que se ejecute o no.

Por estas similitudes y diferencias, se creó una clase nueva que hereda de `BlockCode`, agregando todo lo necesario para soportar la nueva estructura.

Estructura de datos

Al heredar de `BlockCode`, también hereda su estructura de árbol. Además, por la necesidad de tener un subárbol que se puede ejecutar o no, se agrega otra posible relación a las relaciones que ya tenía el nodo:

- `nextScopeStatement`: Apunta a la primera instrucción del nuevo subárbol, tal y como se muestra en la figura 14.

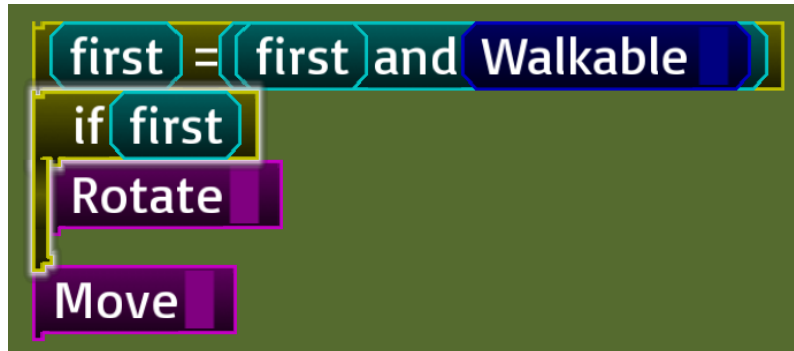


Figura 14: El “`nextScopeStatement`” de la instrucción “if” apunta a la instrucción “Rotate”

Destrucción

El proceso de destrucción de este tipo de bloque es el mismo que el de su predecesor, con la diferencia de que además, en el caso de tener una instrucción en `nextScopeStatement`, también la destruirá.

Estructura de datos interna

Además de la estructura de datos interna heredada, se añaden los sprites que representan la parte vertical del bloque que recoge el nuevo subárbol. Esta parte vertical se extiende para conseguir el tamaño del subárbol y así tratar de representar de una forma gráfica e intuitiva que el nodo actual es el padre del nuevo subárbol.

Por último, cabe añadir que, en contra posición a la creación de la estructura que almacena la información del `BlockCode`, donde solo se actualizaban a los punteros que colgaban de `parent` y de los argumentos, en el caso del `BlockCodeWithScope` también se actualiza la posición del bloque que cuelga de `nextSts`, ya que si el nuevo subárbol crece, el bloque crecerá verticalmente y esto modificará la posición de la siguiente instrucción.

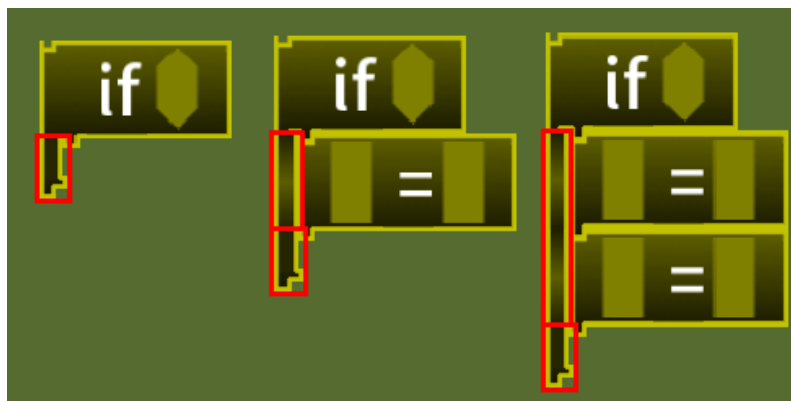


Figura 15: Cada sprite es representado por un rectángulo rojo

Estados booleanos

Con el objetivo de ser lo más usable posible, en los `BlockCodeWithScope` se les añadieron una característica que tienen la mayoría de editores de código modernos, que es la de poder replegar ámbitos para que ocupen menos espacio.

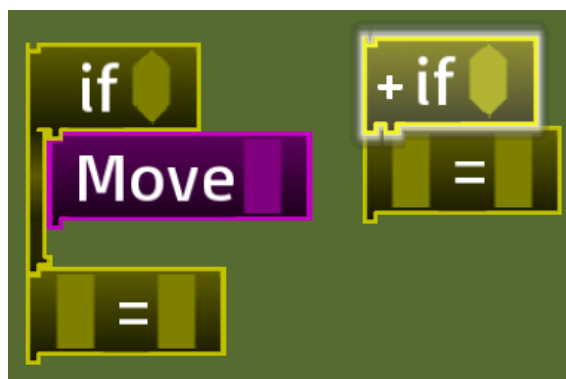


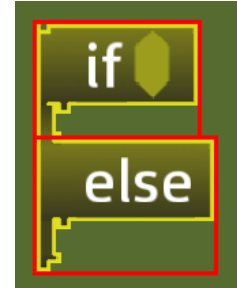
Figura 16: Se puede apreciar como el bloque se modifica para esconder el nuevo ámbito

El nuevo estado, llamado `shrink`, se activa si el usuario clicla encima del bloque a la vez que aprieta la tecla `Ctrl`, replegando el ámbito y a la vez mostrando un símbolo “+” 16, dando a entender que si se clicla ese símbolo, se volverá a mostrar el ámbito escondido.

3.3.4 BlockCodeIfElse

Tal y como hemos dicho en la sección anterior, las estructuras de control que posee CODECRAFT son tres. Por desgracia, con un `BlockCodeWithScope` solo podemos representar dos de ellas (`if` y `while`). Por tanto, para representar gráficamente un `if else` es creada esta clase, que hereda de `BlockCodeWithScope`.

Los objetos de esta clase son dos bloques con ámbito que van juntos 3.3.4, no permitiendo que ninguna otra instrucción sea insertada entre ellas, moviéndose conjuntamente, compartiendo color, temática y estados booleanos del `BlockCode`.



3.3.5 BlockCodeFunct

Acabando con los tipos de clase de bloque de código que existen en el programa, el último es el que representa una función. Ya que en este momento no se le permite al usuario crear sus propias funciones, CODECRAFT solo dispone de dos: `Main` y `VarDefines` 17. Las peculiaridades de esta clase de bloque son que no siguen ni son seguidas por una instrucción, además de que no es necesario pasarle una función anónima a la hora de crearlo, tal y como pasa con los `BlockCode` y los `BlockCodeWithScope`.



Figura 17: Funciones de CODECRAFT

3.4 Intérprete

Un intérprete es un programa que ejecuta instrucciones escritas en un lenguaje de programación sin necesidad de antes compilarlas a lenguaje máquina.

Nuestro intérprete se podría decir que es un intérprete de pila, con la peculiaridad que en lugar de guardar valores en la pila, guarda `BlockCodes` que son del tipo instrucción. De esta manera, el intérprete irá ejecutando instrucciones hasta que la pila se quede vacía o se hayan superado el número de instrucciones que se haya elegido para esa ejecución.

En el momento de empezar a ejecutar un programa `CODECRAFT`, lo primero que hacemos es asegurarnos de que la pila de instrucciones está vacía. Seguido a esto se inserta en la pila la primera instrucción y ya está listo para ejecutarse.

El proceso de un paso de ejecución del interprete es el siguiente:

1. Mira si se ha pasado en el número de instrucciones vacías. Si es así, vacía la pila de instrucciones que quedan por ser ejecutadas.
2. Mira si la pila está vacía. En ese caso, retorna `false` .
3. Se guarda el bloque de arriba de la pila y lo saca de ella.
4. Ejecuta el bloque guardado y retorna `true` .

El booleano de retorno sirve para saber si el programa del usuario ya se ha acabado de ejecutar. De esta manera, si retorna `false` quiere decir que ya ha acabado la ejecución y que no son necesarios más pasos de ejecución.

Por lo explicado hasta ahora, únicamente haríamos un paso de ejecución, ya que en ningún momento se han insertado más bloques en la pila. En cambio, esta inserción se realiza en el momento de ejecutar el bloque guardado.

Los bloques al ejecutarse, si son instrucciones, insertan en la pila del interprete la siguiente instrucción. Es decir, insertan el bloque al que apuntan con el puntero `nextSts`. Una vez insertado, ejecutan el código de la función anónima recibido en la creadora de ese bloque.

El código que se ve a continuación sería la función anónima que ejecutaría el bloque que representa el operador binario “+”.

```
[&](BlockCode* object) {  
    return BlockCodeReturn(  
        Interpret::eval(object->getArgument(0))._number +  
        Interpret::eval(object->getArgument(1))._number  
    );  
}
```

Como se puede observar, en este código se le pide al intérprete que evalúe los dos argumentos que tiene este operador binario y luego se suman los dos valores numéricos, devolviendo este valor como un `BlockCodeReturn`, estructura de datos que usa el intérprete para representar los posibles datos del lenguaje.

La manera de evaluar del intérprete de un bloque que es expresión es la siguiente: Primero ejecuta el bloque y recupera el valor de la ejecución. Después comprueba si el resultado de la ejecución era un identificador o un valor. Si es un identificador, le pide su valor al `MemoryManagement` y, al igual que si fuera un valor, lo retorna.

Este intérprete tiene las siguientes características remarcables:

- Permite ejecutar en modo `Debug`, pudiendo parar el programa `CODECRAFT` en cualquier momento y ejecutándolo paso a paso, remarcando que instrucción es la que se ha ejecutado la última. Por desgracia, no está integrado con la versión final del juego.
- Permite ejecutar una instrucción cada cierto periodo de tiempo. Véase ampliación 4.
- Cuenta el número de instrucciones ejecutadas, así como permite acabar la ejecución del programa si se ejecutan más de cierto número de instrucciones. Como ya sabemos, el problema de la parada es un problema indecidible, pero pudiendo controlar el número de instrucciones ejecutadas, nos permitirá que el juego no se quede colgado cuando un usuario programa un programa “que no acaba”. Tendremos que llegar a un compromiso entre cuantas instrucciones se le deja al usuario antes de cortar la ejecución del programa.

3.5 Gestión de memoria

El `MemoryManagement` es la clase estática encargada de almacenar la memoria de los programas de CODECRAFT ejecutados por el interprete.

Inicialmente se diseñó para poder soportar lenguajes de programación con un número ilimitado de contextos. Finalmente, CODECRAFT solo usa dos contextos diferentes:

- El definido una sola vez por nivel, que es el que define la función `VarDefines`.
- El definido por la función `Main`, contexto que se crea al principio del turno y se elimina al finalizar el turno.

De esta manera, si queremos conservar el valor de una variable de un turno a otro, la tendremos que definir en la función `VarDefines`.

En esta estructura, un ámbito o `Scope` se representa como un `std::unordered_map<std::string, BlockCodeReturn>`, donde se accede por el nombre de la variable a su valor almacenado en memoria. Además, se define como `SymbolTable` un `std::vector<Scope>`.

Con estas dos terminos ya definidos, el `MemoryManagement` se compone por un `std::unordered_map<std::string, SymbolTable>`, donde el string representa el nombre del programa del cual se están guardado los valores de las variables.

Además, se compone por el nombre del programa que está siendo ejecutado. De esta manera podemos fijar el nombre del programa antes de empezar a ejecutarlo y no tenemos que pedirle cada vez de que tabla queremos que nos devuelva los valores. Esto nos permite que el bloque que al ejecutarse le pida asignar cierto valor a cierta variable, no necesite saber en que programa se está ejecutando, y así tener más generalidad en la definición de los bloques.

En el momento de asignar un valor a una variable, se busca si esa variable ya había sido definida en alguno de los ámbitos existentes del programa. Si es así, se le asigna el valor nuevo. En caso contrario, se define la nueva variable en el ámbito más externo y se le asigna su valor.

En el caso de solicitar el valor de una variable, buscará si está definida en alguno de los ámbitos, empezando en el ámbito más externo. Si no se tiene noción de la variable solicitada, devolverá el valor por defecto del tipo solicitado.

3.6 Sistema de escenas

A lo largo del juego se usan diferentes escenas. Cada una tiene sus propios atributos y necesidades, pero todas siguen la misma estructura. Como esta clase se creó en las primeras etapas del desarrollo, necesitábamos que fuera lo más general posible.

Necesitábamos que todas ellas se inicializaran, se ejecutaran, procesaran las entradas, se actualizaran, dibujaran en pantalla, se congelaran en el momento de perder el foco del ratón, que no se deformara la pantalla en el momento de redimensionarla, que pudieran cambiar a otra escena y que se borraran cuando quisiéramos. Además, necesitábamos poder saltar de una a otra independientemente de sus particularidades propias.

Para satisfacer estas necesidades, se creó una clase genérica de escena, de la cual heredarían las demás escenas concretas. De esta manera, la clase abstracta implementa diferentes funciones para satisfacer los requisitos antes expuestos. Algunas de estas funciones serían `init`, llamada en el momento antes de empezar a ejecutar la escena, y `run`, que se queda con el control del programa e implementa el bucle principal del videojuego.

Para almacenar y darles el poder a las escenas, se crea la clase `Game`. Esta clase es la dueña tanto de la ventana donde se dibuja el juego, como de todas las escenas incluidas en el juego. Además, se encarga de llamar a las funciones de las clases estáticas para cargar los datos guardados en la persistencia, así como cargar los recursos que usa el juego, inicializar todos los bloques de código, definir las entradas de teclado y, por último, definir y cargar las escenas. Las escenas son guardadas en un `std::map<std::string, Scene*>`, de esta manera tenemos cada escena identificada por un string, que se usará para poder cambiar de una escena a otra llamando a la función `changeScene` usando su nombre como argumento. De esta manera, el `Game` sabrá que tiene que inicializar la nueva escena y darle el hilo de ejecución.

La parte más importante de la escena genérica es la función `run`. Tal y como hemos dicho antes se queda con el control del programa al implementar el bucle principal del juego. Esto quiere decir que mientras no queramos cambiar de escena, se estará ejecutando el bucle infinito o bucle principal del juego. En este, primeramente se procesa la entrada, después se ejecuta el `update` las veces que hagan falta para mantener las sesenta actualizaciones por segundo, y por último se dibuja en pantalla la escena. Cabe remarcar que las funciones de procesamiento de entradas, el `update` y el dibujado, son funciones que las subclases tienen/deben implementar para que cada escena tenga su comportamiento específico.

A continuación se explican los tipos de escenas que existen en el proyecto.

3.6.1 SceneMenu

La primera escena que se crea y se carga en el juego es el menú principal. En este se muestra una especie de grafo con todos los niveles que dispone el juego. Cada nivel es representado con un `MenuNode`, pequeña clase auxiliar que nos sirve para que en el momento de clicarlo cambiemos de escena cargando el mapa deseado. Además, los nodos que son conexos indican que, de izquierda a derecha, el orden en que deberán ser superados los niveles.

Además, contiene un botón para salir del juego.

3.6.2 SceneCode

Esta es la escena principal. Contiene el editor de código `CODECRAFT` (`BlocksManager`), así como el editor de niveles (`StageEditor`), la pantalla del juego (`Stage`), la interfaz de usuario para poder interactuar con el interprete (`StageUI`) y, por último, el menú que se muestra al superar el nivel o presionar el botón de pausa (`MenuLevelComplete`). Cada clase será explicada más en profundidad en sus propios apartados.

Por tanto, al contener todas partes del programa, se encarga de gestionar la interacción entre ellos, así como de pasarles los inputs necesarios, actualizarlos y dibujarlos. Para conseguir esto dispone de cinco modos diferentes:

- **build**: Es el momento en el que el usuario está construyendo el programa de su robot.
- **play**: Cuando el usuario está visualizando la ejecución de los programas de los robots. En este modo, el jugador no puede modificar su programa.
- **stageEdit**: El modo de edición de niveles. No se puede ni modificar ni ejecutar a los robots directamente.
- **ending**: Momento en el que el robot del usuario ha llegado al objetivo o ha muerto. En este momento el usuario puede reiniciar el mismo nivel, ir al menú, o pasar al siguiente nivel en el caso de haber superado el anterior.
- **pause**: Mostrando el mismo menú que en el caso anterior, en este caso el usuario puede despausar y continuar con lo que estaba haciendo antes de pausar sin perder ningun progreso.

Inicialmente se pensó en crear una escena de este tipo para cada nivel del juego. Finalmente se decidió que únicamente con una `SceneCode` era suficiente, y lo que se haría era recargar el mapa elegido en el momento de su inicialización.

3.6.3 StageUI

Esta pequeña clase nos hace de interfaz entre el usuario y la pantalla.

Nos permite cambiar el tamaño con el que vemos el programa que estamos programando.

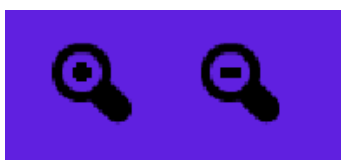


Figura 18: Botones para hacer zoom del programa

Para ejecutar, pausar o reiniciar la ejecución de nuestro robot, lo podemos hacer a través de los botones mostrados en la figura 19. En ella, vemos los tres estados posibles en los que puede estar la ejecución del robot. También se puede observar como dependiendo del estado cambian los iconos de los botones.



Figura 19: (i) Antes de empezar (ii) Ejecutando (iii) Pausado

3.6.4 MenuLevelComplete

Menú que se muestra al superar el nivel o presionar el botón de pausa, mostrándonos estadísticas de la ejecución del programa y permitiéndonos navegar por los niveles.

Posee tres botones con las siguientes funcionalidades:

- Reiniciar el nivel actual.
- Volver al menú principal.
- Avanzar al siguiente nivel. Esto solo es posible si se ha superado el nivel actual.

Las estadísticas mostradas al usuario son tres:

- Número de instrucciones ejecutadas.
- Número de bloques usados.
- Número de turnos necesarios para superar el nivel.

El objetivo de mostrar estas estadísticas es que el usuario pueda ver “como de bien” ha superado el nivel. En el caso de que sus valores obtenidos sean menores que cierto umbral, las estadísticas se mostrarán en color. En caso negativo, se mostrarán en escala de grises. En el caso de no superar el nivel, también se mostrarán las estadísticas en escala de grises.

De esta manera, se incita al jugador a tratar de mejorar su solución del nivel para tratar de optimizar en cualquiera de las tres métricas anteriores.

3.7 Editor visual de CODECRAFT

El editor visual es una de las partes más importantes del juego. Nos permite interaccionar con el programa que estamos construyendo a la hora de jugar. Se compone por tres grupos bien diferenciados: (Figura 20)

1. Conjunto de bloques que forman el programa.
2. Conjunto de bloques que forman el lenguaje (BlockGroupDisplayers).
3. Conjunto de variables definidas por el usuario (VariableGroupDisplayers).



Figura 20: Editor del programa

El editor tiene múltiples características:

Una de ellas sería que se le permite al usuario tener múltiples bloques que no deberán estar conectados a las dos funciones principales que tiene un programa de CODECRAFT (*Main* y *VarDefines*), cada uno representando un árbol 21. Los únicos bloques que se ejecutarán son los que estén conectados a los bloques principales.

Otra característica sería que el usuario puede arrastrar los bloques que están en el grupo 2 para introducirlos en el programa que está creando. También puede arrastrar y soltar los bloques que tiene en su programa para formar el programa. Puede coger expresiones e introducirlos en argumentos de otras expresiones o instrucciones. Puede coger instrucciones y soltarlas entre un par de instrucciones, introduciéndose entre

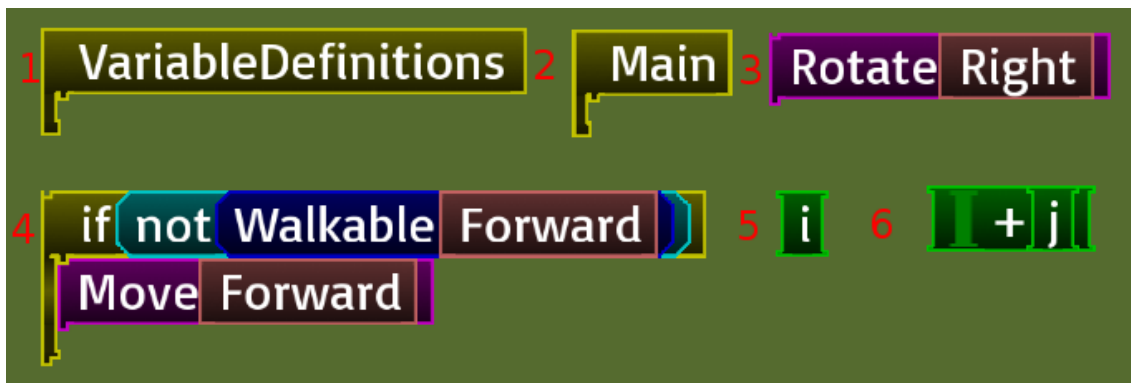


Figura 21: Seis árboles diferentes en el mismo editor del programa

estas. Además, puede copiar, pegar y eliminar un bloque o un grupo de bloques simultáneamente con ayuda del teclado.

Por último, el usuario puede hacer zoom sobre su programa con la ruedita del ratón, además de hacer desplazar hacia arriba o hacia abajo los grupos de variables y bloques del lenguaje.

3.7.1 Grupos de bloques del lenguaje

Tal y como hemos dicho antes, el usuario puede arrastrar bloques desde una especie de expositores para introducirlos en su programa. Estos expositores contienen los bloques del lenguaje ordenados por las categorías: **instrucciones**, **expresiones**, **estructuras de control**, **acciones de robot**, **sensores de robot** y **direcciones de robot**.

Cada expositor es un elemento que permite ponerle un nombre y seguidamente insertarle los bloques que se quieran. Los bloques los muestra verticalmente en la parte inferior en el mismo orden que fueron insertados en el expositor. Además, permite que la parte inferior sea replegada de manera que el usuario no esté forzado a tener los expositores que no le interesan ocupándole espacio de la pantalla.

3.7.2 Grupos de variables del programa

Además de poder arrastrar bloques desde los expositores de bloques del lenguaje, también puede hacerlo desde los expositores de variables del programa. Estos expositores de variables, muestran los bloques igual que los expositores normales, con la diferencia que permiten crear variables de un tipo concreto.

También permite borrar y cambiarles el nombre ((i) de la figura 22), modificando el nombre de todas las apariciones de esa variable en el programa del jugador.

Por último, en el momento de ejecutar el programa del robot, muestra el valor de los variables. ((ii) de la figura 22).

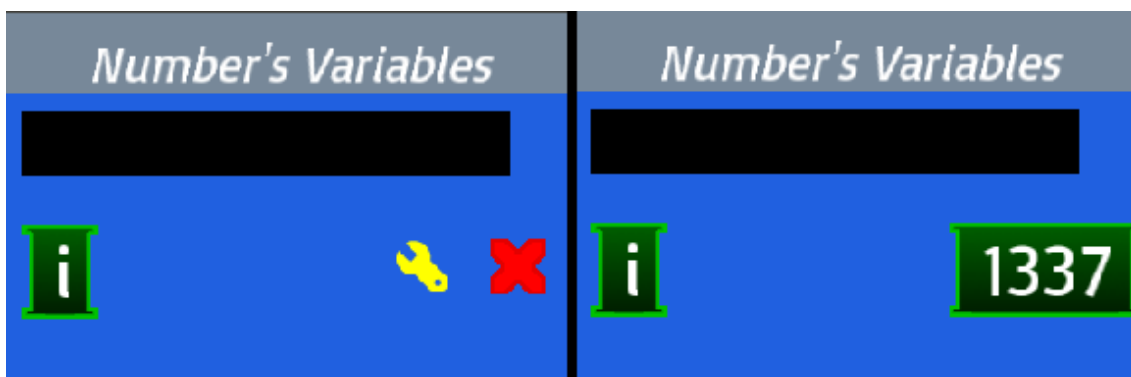


Figura 22: (i) En momento de edición del programa (ii) en momento de ejecución

3.8 Sistema de niveles

Un **Stage** o nivel se caracteriza por tener un mapa, uno o más **Robots**, de los cuales uno será el robot del jugador y un controlador de estados (**StateManager**).

Los niveles son guardados y cargados en formato JSON, ya que esto nos permite insertar nuevos elementos en los niveles sin necesidad de modificar los niveles antiguos.

Inicialmente se carga el nivel desde un fichero en formato JSON, donde se describe lo siguiente:

- El mapa, siendo una matriz de enteros.
- Los **Robots**, cada uno definido como una dirección, una posición en el mapa y su programa

Una vez cargado el nivel, con el mapa y el vector de robots se crea el administrador de estados, encargado de guardar el estado de la partida, así como de evolucionar desde el estado actual al siguiente en el momento que lo necesitemos.

La forma de funcionar de un nivel, es que en cada iteración del juego, se comprueba si el estado actual ya ha sido representado gráficamente. En ese caso, se mira si el estado anterior ya era finalizador. Si lo era, se mostrará el menú de finalización de nivel. Si no lo era, el controlador de estados crea un estado nuevo y se les notifica a todos los robots de su nuevo estado.

Una característica bastante interesante de este sistema de niveles es que permite mostrarle al jugador una presimulación de su programa, mostrándole simultáneamente un número determinados de turnos. Esto hace que el proceso de programar sea mucho mas fluido, ya que cada vez que quieres probar algo no es necesario ejecutar la partida del robot. Podemos ver como sería su funcionamiento en la figura 23.

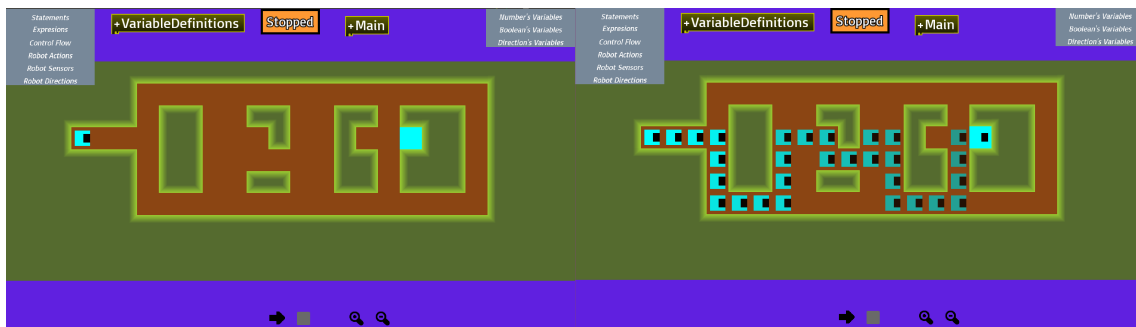


Figura 23: (i) En momento de edición del programa (ii) en momento de ejecución

3.8.1 Robot

Un Robot se compone por dos partes bien diferenciadas:

La primera sería la encargada de guardar el programa destinado a mover ese robot. El robot es el dueño de ese resource, y como tal será el encargado de borrarlo al destruirse. También será el encargado de preparar el Interpret y el MemoryManagement para poder hacer una correcta ejecución de su programa.

La segunda sería la que se encarga de dibujar al robot en la pantalla. Esta se encargaría de recibir la posición y la rotación del robot, tanto la del turno actual como la del turno siguiente, y computar la interpolación en función del tiempo, para después poder dibujarlo.

Esta interpolación no es trivial, ya que depende del estado en el que se encuentre el robot. Por ejemplo, en el caso de que el robot se esté cayendo por un agujero, rotará sobre si mismo a la vez que se va haciendo mas pequeño. En cambio, en el caso de que se choque contra una pared, avanzará hasta la pared y volverá a su posición original ejecutando una pequeña vibración para simular el choque.



Figura 24: Robot cayendose a un agujero

3.8.2 Map

El mapa de un nivel se representa como una matriz de `tiles`. En nuestro juego, un “tile” representa un cuadrado texturizado de siempre el mismo tamaño. Solo tenemos cuatro tipos de `tiles`:

- `wall`: Representa que hay un muro y los robots chocarán contra él si tratan de avanzar en esa dirección.
- `air`: Un agujero donde el robot, si trata de avanzar hacia él, caerá (24).
- `path`: Camino por donde el robot podrá moverse libremente.
- `goal`: Objetivo del robot.

Para que gráficamente no fueran solo texturas diferentes, se crearon programáticamente diferentes tipos de texturas para las `tiles` de tipo `air` y tipo `wall`, de manera que en el momento de conectarlas pareciera que había cierta continuidad (25).

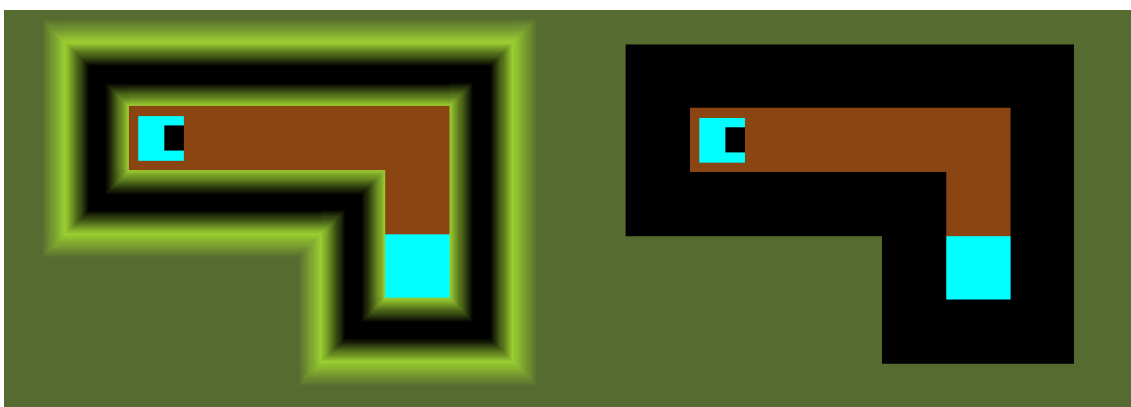


Figura 25: Diferencia entre usar multiples o solo una textura para el mismo tile

3.8.3 Controlador de estados

Un estado representa un turno de la partida. En él se describe la dirección, la acción y la posición de todos los robots.

Este se inicializa con una configuración inicial de robots, y a partir de este estado inicial, se evoluciona para conseguir los estados consecutivos.

Un paso de evolución consiste en, para todos los robots que están vivos, si la cola de acción a hacer por ese robot está vacía, se ejecuta su programa y se encolan todas las acciones pedidas por el programa del robot. Después de esto, se elige la primera acción de la pila de cada robot y se procede a crear el nuevo estado.

La creación de este nuevo estado consiste en ejecutar la acción pedida por cada robot y resolver los conflictos que hayan podido aparecer, como que un robot haya querido caminar hacia un agujero o hacia una pared.

3.9 Editor de niveles

Para facilitarnos lo máximo posible el trabajo de creación de niveles, se ha creado un editor de niveles. En él se te permite cargar un nivel ya creado con anterioridad, modificarlo cambiando el mapa, añadiendo, moviendo o rotando robots. Además, se permite cambiar el programa por defecto que tiene los robots.

Una vez acabada la edición, se puede sobrescribir sobre el nivel cargado o guardarlo con un nombre nuevo.

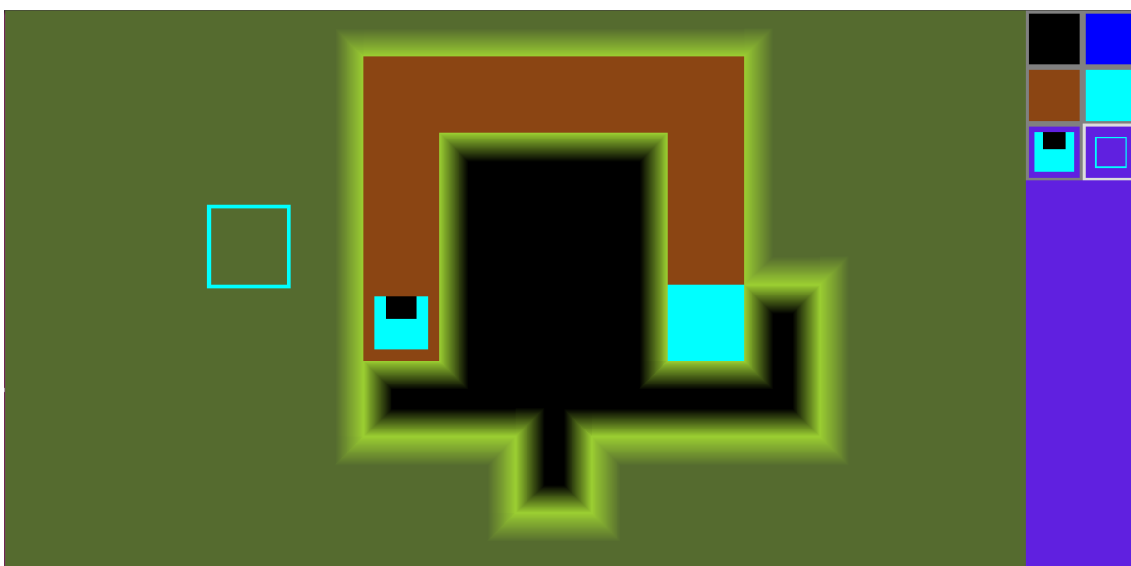


Figura 26: Editor de niveles

Como se puede ver en la figura 26, el editor de niveles está separado en dos partes bien diferenciadas.

La primera parte y la más grande es la que muestra el mapa que estamos editando. En esa parte se puede hacer zoom para alejarnos o acercarnos al mapa, además de que si tratamos de modificar una posición del mapa que está fuera de él, nos ampliará el mapa para poder acogerlo.

La segunda parte es donde se muestran los seis recuadros de la derecha. Esa parte nos sirve para poder elegir el pincel que usaremos a la hora de modificar el nivel. Los primeros cuatro recuadros representan los tipos de `tiles` diferentes. El pincel con un robot sirve para colocar los nuevos robots. Por último, el pincel con el perímetro de un cuadrado turquesa nos sirve para seleccionar los robots que ya hay en pantalla, pudiendo modificarles el programa interno, rotarlos o moverlos de posición, todo esto con diferentes acciones de ratón y teclado.

3.10 InputManager

Clase auxiliar que nos permite relacionar las acciones que se pueden realizar dentro del juego a controles modificables en tiempo de ejecución, además de tener múltiples controles diferentes para la misma acción. El tipo de entrada soportados son el teclado, el ratón y los mandos de videoconsolas.

También nos permite preguntar si esa acción es la primera vez que se está realizando o no, independientemente que se esté realizando durante múltiples iteraciones. Esta diferenciación es útil a la hora de recibir ciertas acciones que está haciendo el usuario, como sería la de abrir un menú.

Para usar esta clase, primero de todo tendríamos que definir todas las acciones que queremos detectar. Después, por cada iteración del bucle principal del juego, comprobaríamos si alguna de las acciones ha sido detectada y actualizaríamos la estructura interna. Internamente se representa como un conjunto de `std::map`, uno para cada tipo de entrada que tenemos. Los elementos del map estarían indexados por la acción que queremos detectar. Por último, podremos pedir el estado de esa acción en cualquier parte del proyecto.

3.11 MagicView

La clase MagicView nace por la necesidad de tener una vista que siempre garantizase mostrar el mismo espacio de la escena, independientemente del tamaño o aspect ratio de la pantalla.

Esta clase permite cinco modos de configuración:

- **Crop:** Mostrará el tamaño de la escena que le pidas. Si es aspect ratio demandado no es el mismo que el de la pantalla, se modificará el viewport, dejando unas bandas negras a los lados o arriba y abajo.
- **Expanded:** Este modo creará un vista que mostrará exactamente lo mismo que en el modo anterior, pero en lugar de modificar el viewport dejando bandas negras, modificará el tamaño de la vista.
- **Viewport:** Este modo es igual que el expanded, pero pudiendo elegir el viewport.
- **Viewport Hardcoded Width:** A esta vista se le dice el viewport y la resolución horizontal.

Además, la clase implementa un par de funcionalidades muy útiles:

- `sf::Vector2f getMouseCoord()`: te da la posición del ratón en coordenadas del world-space
- `void zoomToMouse(float factor)`: te permite hacer zoom manteniendo el punto del ratón en la misma coordenada en world-space.

3.12 User Interface

La interfaz de usuario es una parte muy importante de cualquier juego. Con ella, el jugador interactúa y navega por el juego. Por ello, se han implementado diferentes elementos que, aunque no tienen por qué estar directamente relacionados este juego en concreto, son muy importantes para el proyecto.

3.12.1 BorderSprite

El objetivo de esta clase sería la de tener un Sprite que se pueda usar de fondo de un menú independientemente del tamaño de este, y que al reescalarlo los bordes no se deformen.

Esto se consigue haciendo que en lugar de tener un solo quad en donde se pinta la textura, tenemos nueve, cada uno definido de forma independiente. De esta manera, las esquinas tendrán un tamaño definido fijo, independiente del tamaño global. En cambio, los laterales solo se escalarán a lo largo. Y como es obvio, el centro se escalará para conseguir el tamaño deseado.

3.12.2 Button

Esta clase abstracta sirve de simple interfaz para crear un botón que se pueda clicar.

De esta heredan dos tipos de botones diferentes:

- `CircleButton`: Botón circular con una textura en su interior.
- `TextButton`: Botón rectangular con texto centrado en su interior. Este rectángulo será un `BorderSprite` para que pueda ser del tamaño que queramos. Este tamaño lo podremos elegir de dos maneras: definiendo el tamaño del botón en ancho y alto, o definiendo el tamaño de los caracteres del texto.

3.12.3 TextField

Todo lenguaje de programación que tenga variables, debe tener forma de definir las. En nuestro caso, al poder nombrarlas, necesitamos una forma de introducir texto de forma interactiva al programa. Para esto es creada esta simple clase, con las siguientes características:

- **Cursor Interactivo:** se puede mover hacia los lados con el teclado o con el ratón.
- **Tamaño Máximo:** al introducir más caracteres del tamaño máximo, el TextField ya no aceptará más y se ignorarán.
- **Comprobación de Errores:** se permite heredar de esta clase y decidir que tipo de texto introducido es correcto. Al introducir un texto incorrecto, cambiará el color de este.
- **Texto Propio o Adquirido:** es posible usar un texto ya instanciado de otra clase o tener un texto propio. Cuando el texto es propio, además dibujará un fondo debajo del texto.
- **Activación:** Al clicarse, el TextField se activa y el cursor empieza a parpadear. Al desactivarse, el cursor desaparece y no permite introducir texto.

De esta clase heredan otras dos, en función de si se va a usar para declarar variables o introducir números:

- **TextFieldVariable:** el texto introducido únicamente puede empezar por letra o barra baja, seguido de letra, número o barra baja.
- **TextFieldNumeric:** únicamente es correcto introducir dígitos.

3.13 Controlador de recursos

El juego utiliza diferentes recursos multimedia, y para poder trabajar de una forma sencilla y rápida con ellos, se ha creado pequeño sistema que se encarga de cargarlos y almacenarlos.

En el momento de arranque de la aplicación, este sistema carga todas las texturas, fuentes y shaders. Además, se encarga de generar las texturas relacionadas con contorneado de los bloques y las texturas de las celdas del mapa.

En el momento de su diseño se eligió que se cargaría todo al principio, en lugar de cargarlo en el momento de necesitarlo y liberarlo cuando ya no se necesitara. Se decidió esto porque, al ser una aplicación sin un alto desempeño gráfico, con un número no muy elevado de pequeñas texturas, no generaba un sobre coste de tiempo inaceptable. Además, implementarlo se simplificaba mucho respecto a la segunda opción. En nuestro caso, únicamente había que definir una clase donde en su archivo de cabeceras se definían todos los recursos de forma estática. De esta manera, para poderlo usar en cualquier parte del programa únicamente había que incluir esa cabecera y usar el recurso con la sintaxis `Resources::recurso`.

4 Posibles ampliaciones

- Poder elegir la velocidad de ejecución del interprete.

De esta manera, en lugar de que cada segundo se ejecute todo el programa independientemente de su tamaño, se fijara cierta tiempo por cada instrucción ejecutada. El interprete ya es capaz de hacer esto. En cambio, en el momento de integrarlo en **Stage** salen tres opciones diferentes.

La primera sería que se calculara el número de instrucciones que se ejecutarían en ese turno, y decidir que el tiempo de ese turno sería `tiempo_de_ejecutar_una_instrucción * numero_de_instrucciones`. De esta manera el robot se movería a diferente velocidad dependiendo del tamaño del programa.

La segunda sería que se calculara el número de instrucciones y luego hacer que el tiempo de ejecución de cada instrucción fuera `tiempo_de_turno/numero_de_instrucciones`. De esta manera el robot siempre se movería a la misma velocidad, pero el tiempo que un bloque estaría delimitado sería inconsistente y, seguramente, demasiado pequeño.

La tercera opción sería la de primero ejecutar el programa a velocidad constante y una vez que se haya acabado de ejecutar, el robot se mueva a velocidad constante. De esta manera siempre podríamos ver a la velocidad deseada la ejecución del programa y la velocidad del robot, a costa de no poder ver a robot moverse continuamente.

- Poder subir y descargar niveles de otros usuarios a/desde un servidor.
- Guardar las métricas que salen cuando te pasas un nivel para hacer una clasificación online entre los usuarios, pudiendose mostrar cuando se supera un nivel.
- El usuario pueda definir sus propios bloques, eligiendo que partes quiere que incluya y definiendo el comportamiento con otros bloques. Estos nuevos bloques serían como llamadas a función.

5 Gestión de proyecto

5.1 Objetivos

El objetivo principal de este proyecto es la creación de un videojuego con la finalidad de que se juegue para aprender conceptos básicos de programación y lógica. Tal y como se ha dicho en la sección anterior, este videojuego tratará de rellenar un vacío que hay en el campo de los videojuegos de programación y lenguajes de programación visuales y didácticos, que sería un videojuego en el cual, cada turno de la partida, se ejecuta el programa completo del jugador y de esa ejecución se deduce el comportamiento que el avatar del jugador deberá ejecutar. Para conseguir este objetivo, se deberán hacer las siguientes tareas:

- Diseño del lenguaje de programación visual CODECRAFT, lenguaje que deberá ser lo suficientemente simple para que sea fácil de asimilar, pero lo suficiente potente para solucionar puzzles complicados.
- Editor visual de CODECRAFT, con la suficiente usabilidad para que al usuario le sea intuitivo el uso del lenguaje y a la vez le sea simple la asimilación de los conceptos de programación.
- Diseño e implementación del compilador e interprete de CODECRAFT.
- Diseño e implementación del juego, de tal manera que se deban utilizar programas CODECRAFT para dirigir a los robots dentro de cada nivel.
- Diseño e implementación del editor de niveles, el cual será usando tanto por el desarrollador como por los usuarios para crear nuevos niveles. Deberá ser, al igual que el editor visual de CODECRAFT, fácil e intuitivo de usar. Al ser una herramienta enfocada para el desarrollador, no se tratará de maximizar con tanto esfuerzo su usabilidad.
- Diseño de los niveles del videojuego, así como la curva de aprendizaje. Estos niveles deberán empezar siendo muy simples, con un número de instrucciones disponibles muy pequeño para no abrumar al usuario, y a medida que este vaya superando niveles y ganando experiencia se irán haciendo cada vez más complicados, haciendo disponible una gran cantidad de instrucciones.

5.2 Alcance y Posibles Obstáculos

Describimos a continuación el alcance del proyecto y los posibles obstáculos con los que nos podemos encontrar en cada parte del proyecto.

5.3 CODECRAFT

El lenguaje de programación CODECRAFT no es de propósito general, pues está diseñado justamente para mover el robot del juego ROGUES CODECRAFT. Así pues, su conjunto de instrucciones es limitado, y básicamente se pueden declarar variables de tipo booleano y entero, utilizar estructuras de control habituales, y también llamadas a función con parametrización por valor. Existe un tipo de dato especial que sirve para especificar las direcciones de movimiento del robot.

Deberemos asegurar que el lenguaje permite describir de manera cómoda la solución a los niveles con los que se va a encontrar el robot. En este sentido debemos afrontar una gran dificultad, ya que simplicidad y expresividad son conceptos que a menudo resultan contrapuestos. Con el fin de superar este obstáculo, será conveniente contar con algunos ejemplos de niveles (o cuanto menos una idea aproximada de ellos), y ver como se representa el programa CODECRAFT solución para cada uno de ellos.

5.4 Editor visual de CODECRAFT

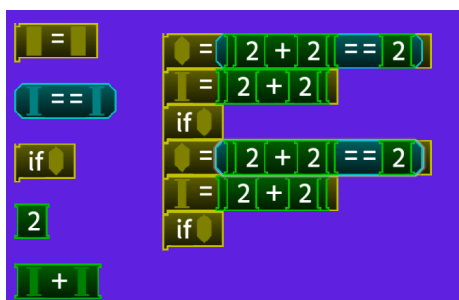


Figura 27: Prototipo inicial

El editor visual representará y agrupará las instrucciones mediante bloques, y permitirá al usuario organizarlos a conveniencia. Permitirá insertar, copiar y arrastrar bloques con el ratón. Es crucial que el editor permita organizar el código de forma cómoda e intuitiva, porque como todo programador sabe, este es un elemento clave en la productividad del proceso de desarrollo. En nuestro caso concreto esto es todavía más relevante porque nuestro enfoque tiene fines educativos. Así pues, de algún modo el editor debería facilitar la comprensión de los programas

a partir de la representación gráfica de los mismos.

Será conveniente contar con ejemplos concretos de programas CODECRAFT y determinar cual es la representación visual más adecuada. Es importante realizar este estudio previo para evitar tener que hacer modificaciones costosas a posteriori. En

este sentido contamos con la ayuda de Pere Pau Vazquez (mencionado en la introducción) como experto en interacción de usuario.

Con el objetivo de mostrar un prototipo inicial se pueden referenciar las figuras 27 y 28

5.5 Motor del videojuego

El motor del videojuego se diseñará e implementará desde cero, usando una librería multimedia llamada SFML (Simple and Fast Multimedia Library). Dentro del motor estará implementado los sistemas para que el videojuego pueda funcionar, tales como el interprete de CODECRAFT, las pantallas o escenarios, el gestor de estados de la partida actual y el editor de niveles.

Al ser la parte del proyecto en la cual más se tendrá que programar, es más probable que introduzca bugs a la hora de implementar nuevas funcionalidades. Para mitigar este problema, tendré que probar exhaustivamente el programa en el momento de cada nueva inserción de código, con el objetivo de encontrar los bugs en el momento que se introducen y de esta manera resolverlos cuando todavía se puede intuir que los produce, ahorrándome una gran cantidad de tiempo en esta tarea.

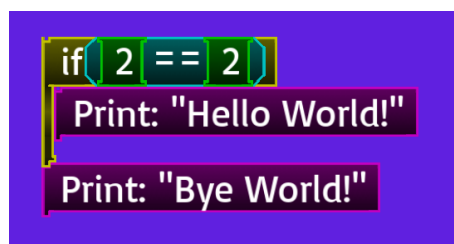


Figura 28: Prototipo inicial

5.6 Diseño de niveles

Sobre el motor del videojuego diseñaremos los niveles para que sean jugados por el usuario. Cada nivel deberá tener como objetivo enseñar o afianzar algún concepto.

Estos niveles deberán representar una curva de aprendizaje que no frustre al jugador con una gran dificultad en poco tiempo, pero que tampoco desmotive al jugador por ser repetitivo. Para tratar de solventar este problema seguiremos una metodología iterativa, rediseñando los niveles y su progresión hasta que estemos contentos con el resultado.

5.7 Metodología y Rigor

Ya que este proyecto es de un tamaño significativo para una sola persona, es importante saber siempre en que punto de la planificación del proyecto se está. Para conseguir esto utilizaré una herramienta llamada Trello que organiza el proyecto en un tablero. En estos tableros se pueden poner listas que categorizan las tareas que están asociadas a ellas. En mi caso tendré cinco listas: Tareas en proceso de realización, prioridad alta, prioridad media, prioridad baja y tareas realizadas. Seguiré la estrategia siguiente: cada lunes planificaré que voy a hacer durante la semana, eligiendo las tareas en orden de prioridad. En el caso de no llegar a realizar las tareas de la semana anterior, analizaré el porqué para tratar de resolverlo lo antes posible.



Con el fin de tener siempre el código accesible y guardar un historial de este mismo, utilizaré un sistema de control de versiones llamado Git, y mas en específico, usaré GitHub, portal en línea que te deja crear y guardar repositorios en sus servidores, teniendo siempre acceso desde un navegador web o terminal. Siguiendo la filosofía de hacer una versión por cada nueva tarea que se implementa, podré mirar el historial para ver la línea temporal de tareas que he ido realizando, así como los cambios que se han hecho en el código.

Además de las dos anteriores, usaré Wakatime, herramienta que te permite hacer un seguimiento del número de horas que programas en el proyecto, fichero o incluso versión al sincronizarlo con el repositorio de GitHub. Gracias a esta herramienta sabré automáticamente el número de horas que le he dedicado a cada tarea.



Por último, las reuniones con el director serán una parte importante, ya que me guiará en el proceso de seguimiento del proyecto, diciéndome si voy atrasado o si una tarea me va a costar más tiempo del que yo mismo esperarí.

Con el fin del validar cada tarea, y como ya se comentó en la sección a la hora de resolver los posibles obstáculos, yo mismo probaré el programa a medida que vaya introduciendo código nuevo. Además, trataré de conseguir el mayor feedback de mis compañeros de la asociación de videojuegos de VGAFIB en la parte del videojuego, así como de los hijos de algunos amigos en la parte del lenguaje de programación.

5.8 Descripción de las tareas

5.8.1 Viabilidad del proyecto deseado

La primera tarea que tuve que hacer fue encontrar un profesor que quisiera ser el director de la idea inicial que era el proyecto y a ser posible, que tuviera experiencia en el desarrollo de videojuegos.

Una vez encontrado este, hicimos una reunión inicial para hablar del alcance de la idea inicial, si tenía sentido esta dentro del estado del arte existente, y de tal manera que diera tiempo de hacer un videojuego lo suficiente divertido y que, además, tuviera las características que yo deseaba, como pueden ser que el control del personaje se hiciera a través de un lenguaje de programación visual, que el editor gráfico de CODECRAFT mostrara en todo momento el valor de las variables definidas o que cada turno de la partida se ejecutara el programa completo del robot.

Decidido que era una buena idea como proyecto, elegimos que características daría tiempo a implementar y cuales no. Esta tarea inicial tomo poco tiempo comparado con las siguientes tareas. También cabe destacar que además de reunirnos inicialmente, ha habido reuniones posteriores para pulir diferentes ideas.

5.8.2 Planificación del proyecto (Hito inicial)

Esta tarea es la que nos ocupa la mayor parte del tiempo en este momento. Consta de las siguientes partes o fases:

- Definición del alcance del proyecto y contextualización.
- Planificación temporal.
- Gestión económica y sostenibilidad

5.8.3 Configuración inicial

Antes de empezar con el desarrollo, necesitaremos hacer la configuración inicial en las máquinas en las que pienso trabajar, en este caso mi ordenador portátil y mi ordenador de sobremesa.

Las plataformas objetivo son Windows y Linux. De tal manera, tendremos que instalar los dos sistemas operativos.

Linux: Esta plataforma será en la que desarrollaré mayormente. Al usar una librería de gráficos de código abierto (SFML), haré un script que me descargue el código,

me lo compile y me lo instale. Además de esto, tendré que instalarme el editor de texto (**Atom**) con el que trabajaré. Al ser también de código abierto, también me lo descargaré y lo compilaré. Dentro de **Atom** instalaré los plugins necesarios para poder trabajar cómodamente en C++, como sería un Linter de C++. También tendremos que instalar **git** como sistema de control de versiones.

Windows: Esta plataforma se usará únicamente para hacer la build de Windows. Para tal, será necesario instalar el **Visual Studio** y **SFML**.

En el momento de tener todo instalado y funcionando, podremos empezar con el desarrollo del proyecto.

5.8.4 Desarrollo del proyecto como tal

Las tareas a realizar en el proyecto quedan agrupadas en las siguientes categorías:

- (A) Diseño del lenguaje de programación **CODECRAFT**.
- (B) Diseño e implementación del editor visual de **CODECRAFT**.
- (C) Diseño e implementación del intérprete de **CODECRAFT**.
- (D) Diseño e implementación del compilador entre **CODECRAFT** visual y texto.
- (E) Diseño e implementación del motor del juego **ROGUES CODECRAFT**.
- (F) Diseño e implementación del editor de niveles de **ROGUES CODECRAFT**.
- (G) Diseño de los niveles de **ROGUES CODECRAFT**.

Sin embargo, cabe remarcar que el desarrollo del proyecto va a ser iterativo y progresivo, pasando por todas esas categorías repetidamente, y de forma casi simultánea. El motivo es que están muy íntimamente ligadas, y este enfoque progresivo va a permitir validar a cada paso que todas las piezas encajan correctamente. De hecho, las diversas ampliaciones que vaya realizando sobre el lenguaje de programación concidionarán el desarrollo en el resto de categorías. Por ese motivo, describo a continuación las sucesivas ampliaciones del lenguaje de programación que me planteo realizar como fases intermedias hasta la definición completa del lenguaje **CODECRAFT** (entre paréntesis indico el nombre con el que me referiré a cada fase más adelante):

1. (**Escritura**) Instrucción de escritura de string constante. Esta instrucción no tiene ninguna utilidad para **ROGUES CODECRAFT**, pero va a resultarme útil como medio de validación. Además, es una instrucción que no necesita de la existencia de variables ni de ningún tipo de dato, de modo que tiene sentido empezar con ella.

2. (**Tipos**) Tipos de datos booleano y número, declaración de variables, e instrucción de asignación.
3. (**Operadores**) Operadores básicos `+`, `-`, `<=`, `==`.
4. (**If**) Instrucción condicional (`if`).
5. (**While**) Instrucción iterativa (`while`).
6. (**Dirección**) Definición del tipo de dato `direccion` del robot.
7. (**Move**) Instrucción `MOVE` para desplazar el robot en una dirección.
8. (**Rotate**) Instrucción `ROTATE` para reorientar al robot.
9. (**IfElse**) Instrucción condicional avanzada (`if {} else {}`).
10. (**Walkable**) Sensor `WALKABLE` para preguntar si se puede caminar por una posición del mapa.
11. (**Otros**) Añadir otras instrucciones y sensores para el robot.
12. (**Funciones**) Definición de funciones sin parámetros.

La mayoría de tareas del proyecto corresponden a la incidencia de una de estas ampliaciones del lenguaje en cada una de las categorías. Por ejemplo, la ampliación `Move` da lugar a las tareas de (1) ampliar el editor visual para permitir utilizar la instrucción `Move`, (2) ampliar el intérprete para reconocer y ejecutarla, (3) ampliar el compilador para traducirla y (4) ampliar el motor del juego para poder reaccionar a las nuevas peticiones del intérprete.

Aparte del tipo de tareas que hemos mencionado, también será necesario llevar a cabo otras que no son una consecuencia directa de una ampliación del lenguaje de programación:

- (**EditorVisual**) Construir la base del editor visual.
- (**Funcionalidades**) Añadir funcionalidades de cortar, copiar y pegar en el editor visual.
- (**Desplegables**) Añadir agrupaciones de bloques desplegados.
- (**Motor**) Construir la base del motor del videojuego: el sistema de escenas, la gestión de los eventos, el sistema de sonido, el sistema de almacenamiento de datos sobre partidas anteriores, la definición y gestión de niveles, y la gestión del estado de la partida.
- (**EditorNiveles**) Construir la base del editor de niveles.

- (Servidor) Construir un servidor donde almacenar niveles y que los usuarios puedan descargarlos.
- (EditorFunc) Añadir funcionalidades al editor de niveles.

5.8.5 Hito final

En esta etapa comprobaremos que el videojuego funciona como se espera de él, se harán las versiones de lanzamiento, y se comprobará que tanto como la presentación final como la documentación están correctamente preparadas.

5.9 Tiempo aproximado para cada tarea

Al llevar más de 6 meses desarrollando el proyecto a tiempo parcial y con la idea de dedicarme a tiempo completo al proyecto durante los cuatro meses que quedan, las horas de la siguiente tabla son una completa invención y no reflejan la realidad, ya que ahora mismo no puedo ni intuir que tareas voy a incluir en el proyecto cuando acabe las que tengo programadas:

Tarea	A	B	C	D	E	F	G	Total
EditorVisual	0	10	0	0	0	0	0	10
Escritura	1	2	1	1	0	0	0	5
Desplegables	0	5	0	0	0	0	0	5
Tipos	5	20	5	3	0	0	0	33
Operadores	1	3	1	2	0	0	0	7
Funcionalidades	0	5	0	0	0	0	0	5
If	1	10	5	2	0	0	0	18
While	1	2	5	1	0	0	0	9
Motor	0	0	0	0	40	0	0	40
EditorNiveles	0	0	0	0	0	10	0	10
Dirección	3	5	5	1	10	0	0	24
Move	1	2	5	1	10	0	1	20
Rotate	1	1	2	1	5	0	10	20
IfElse	1	10	5	3	0	0	5	24
Walkable	1	1	1	1	10	0	10	24
Otros	5	5	5	5	50	0	50	120
Funciones	5	20	10	5	0	0	10	50
Servidor	0	0	0	0	20	0	10	30
EditorFunc	0	0	0	0	0	30	0	30
Total	26	101	50	26	145	40	96	484

5.10 Recursos utilizados

En esta sección se explicarán que recursos serán necesarios para la realización de este proyecto.

5.10.1 Software

- Ubuntu 14.04 LTS
- Atom
- git
- SFML
- Windows 10
- Visual Studio
- Trello

5.10.2 Hardware

- PC de sobremesa (Intel Core i5-2500K CPU a 3.3 GHz x 4, 16 GB RAM, Radeon HD 7850)
- PC portatil

5.11 Diagrama de Gant

5.12 Plan de Acción

Ya que el proyecto se desarrolla por fases y no está completamente claro que funcionalidades tendrá el juego final, el plan anterior no tiene mucha vigencia. En cualquier caso, durante el transcurso del desarrollo puede haber desviación sobre el plan anterior. En el caso de que las tareas se hagan demasiado rápido, se valorará entre dedicarle mas horas la tarea **Otros**. Por otro lado, si el problema es que se tarda demasiado y nos quedamos sin tiempo, ya que el tiempo que tenemos para realizar el proyecto entero es muy limitado, trataremos de tener las características core desarrolladas. Para esto, se recortarán horas en la misma tarea (**Otros**).

la creación de este proyecto y se usa para otras tareas. Además, al ser estudiante de la FIB, tengo acceso a software de Microsoft de forma gratuita.

Producto	Precio	Vida Util	Amortización	Comentario
Ubuntu 14.04 LTS	0.00 €		0.00 €	
Windows 10	0.00 €		0.00 €	Gratis para estudiantes
Atom	0.00 €		0.00 €	
Git	0.00 €		0.00 €	
Github	0,00 €		0.00 €	Gratis para estudiantes
SFML	0.00 €		0.00 €	
Visual Studio	0.00 €		0.00 €	Gratis para estudiantes
Trello	0.00 €		0.00 €	
Total	0.00 €		0.00 €	

5.13.3 Recursos Humanos

Ya que voy a ser el único desarrollador y no estoy graduado, los precios por hora de los recursos humanos van a ser, ya que los de convenio de la UPC no sirven, lo mínimo que cobraría si tubiera que hacer este mismo proyecto para otra persona.

Rol	Developer	Game D	Level D	Horas	Coste
EditorVisual	0	10	0	10	200.00 €
Escritura	1	4	0	5	100.00 €
Desplegables	0	5	0	5	100.00 €
Tipos	5	28	0	33	660.00 €
Operadores	1	6	0	7	140.00 €
Funcionalidades	0	5	0	5	100.00 €
If	1	17	0	18	360.00 €
While	1	8	0	9	180.00 €
Motor	20	20	0	40	800.00 €
EditorNiveles	5	5	0	10	200.00 €
Dirección	8	16	0	24	480.00 €
Move	6	13	1	20	400.00 €
Rotate	3	7	10	20	400.00 €
IfElse	1	18	5	24	480.00 €
Walkable	6	8	10	24	480.00 €
Otros	30	40	50	120	2400.00 €
Funciones	5	35	10	50	1000.00 €
Servidor	0	20	10	30	600.00 €
EditorFunc	0	30	0	30	600.00 €
Total	1860.00€	5900.00 €	1920.00 €		9680.00 €
Contingencia (15%)					1452.00 €
Total				484	11132.00 €

Ya que no existe un apartado para el diseño del videojuego como tal, se han repartido las horas del motor del videojuego entre el Game Designer y el Game Developer, ya que el game design de este videojuego se está tratando de hacer tal y como lo hizo Jonathan Blow con Braid: "The process of designing the gameplay for this game was more like discovering things that already exist, than it was like creating something new and arbitrary. And another way to say that is that there's an extent to which this game designed itself".

El Level Designer, como es obvio, solamente trabajará en el diseño de niveles. El Game Designer trabajará, además de lo especificado anteriormente, en el diseño de CODECRAFT. El Game Developer hará todo lo demás.

Por otro lado, si en lugar de hacer el videojuego como si estuviera contratado, lo cobrara como un indie game developer, el coste de los recursos humanos en este momento del desarrollo sería cero. Si contáramos el coste de oportunidad de habernos dedicado a otra cosa, el coste del proyecto ya es indefinido. Podría costarnos desde mucho mucho dinero y nos hubiéramos dedicado a ser corredores de bolsa y nos hubiera salido muy bien, a haber evitado deudas y daños económicos y humanos,

en el caso de que nos hubieramos dedicado a, por ejemplo, hacer software para una central nuclear que falla y mata a cientos de miles de personas.

En el cálculo del coste total del proyecto se contará la primera opción (convenio con la UPC).

5.13.4 Total

Usando los datos de las tres subsecciones anteriores, se hace la tabla siguiente.

Concepto	Coste
Hardware	150.00 €
Software	0.00 €
Recursos Humanos	11132.00 €
Total	11282.00 €

5.14 Control de desviación

Como ya ha quedado claro, al ser un videojuego indie y haber un solo desarrollador, con un 99% de probabilidad la planificación que se ha hecho es una ficción. De esta manera en el momento de hacer menos tareas porque se haya hecho una planificación demasiado optimista (muy posible, ya que al ser un estudiante que nunca ha trabajado en un proyecto de esta envergadura, pues no tengo una idea muy fiable de lo que va a tardar una tarea en completarse), o de hacer más tareas de las especificadas (ya que puedo estar más motivado o se me pueden ocurrir más cosas que quiero introducir en el proyecto final). En cualquiera de los dos casos, habrá que modificar el número de horas totales a realizar para la completitud del proyecto. Ya que el precio de los desarrolladores es bastante bajo, ya que son becarios, no habrá demasiado problema.

El software, en caso de necesitar más de este, se pueden conseguir programas gratuitos o que sean gratuitos para los estudiantes.

En el caso del hardware, si ocurriera alguna desgracia y se estropeará cualquiera de los dos pcs, se podría seguir desarrollando con uno solo de ellos. Si la desgracia fuera mayor y se estropearan los dos, todavía podría ir a desarrollar al despacho de VGAFIB sin tener que invertir más dinero en el proyecto.

5.15 Sostenibilidad

En esta sección se hablará sobre la sostenibilidad del proyecto en tres aspectos: Social, económico y medioambiental. Además, se le dará una nota arbitrariamente.

5.15.1 Social

Este proyecto, como tantos otros, busca impulsar a los jóvenes a aprender a programar y, en nuestro caso, hacerlo de una forma divertida a la vez que potente.

Puede parecer que las empresas que se dedican a desarrollar un software con el mismo objetivo que este proyecto pueden verse afectadas, pero en realidad está lo suficientemente diferenciado de ellas para no ser competencia directa de su producto, sino una alternativa diferente.

Como cosa buena, este proyecto no va a traducir el lenguaje de código visual a javascript, como hacen las principales alternativas. De esta manera salvamos a los jóvenes de la influencia negativa de aprender ese lenguaje.

A este proyecto le voy a conceder un 7 en sostenibilidad social, ya que va a ayudar a la sociedad a aprender a programar, y programar es el futuro, pero al ser realizado por una sola persona, sin experiencia en la parte técnica ni en la pedagógica, el resultado seguramente sea menos importante que si, por ejemplo, lo hiciera un grupo de investigación del MIT.

5.15.2 Económica

Según los recursos especificados en la sección de Gestión económica, el precio del proyecto, ya de por sí, es bajo. Se podría bajar realizando diferentes cambios en los recursos humanos:

- **Contratando becarios más baratos de otras universidades:** De esta manera el precio sería más bajo, aunque la calidad podría bajar también.
- **Hacer únicamente la especificación y contratando a desarrolladores en países en proceso de desarrollo:** Haciendo únicamente la especificación nos ahorraríamos las horas del Game Developer, dinero que se podría usar para contratar a un equipo de indios para que nos implementaran la especificación deseada.

Al ser todo el software gratis, no podríamos mejorar nada en ese aspecto.

En el caso del hardware, se podría realizar el proyecto con un solo PC, o incluso sin ninguno, tal y como se ha visto en la sección anterior.

Teniendo en cuenta los puntos anteriores, le voy a otorgar un 7 en sostenibilidad económica, ya que se podría realizar el proyecto con menos dinero, pero sacrificando calidad por el camino.

5.15.3 Medioambiental

Al ser un proyecto de software y realizarlo en mi propia casa, no va a tener realmente una huella ecológica, ya que la única contaminación directamente relacionada sería la electricidad gastada por el pc en las horas de uso.

Por otro lado, al ser un videojuego, el gasto energético de usarlo va a ser notable, ya que el procesador y la gráfica van a estar usándose extensamente. En cambio, los gráficos del juego no van a ser demasiado exigentes, así como tiempo de procesado entre frame y frame va a ser menor del total. De esta manera los usuario ahorrarán electricidad, paliando los efectos negativos que tiene la creación de esa electricidad.

De esta manera, en este apartado el proyecto va a tener un 8, ya que lo único que no es agradable con la naturaleza es la generación de la electricidad usada (en el caso que se genere a base de quemar combustibles fósiles o energía nuclear).

Referencias

- [1] Gaming lifestyle, what it means to be a gamer - giant bomb. <http://www.giantbomb.com/forums/general-discussion-30/gaming-lifestyle-what-it-means-to-be-a-gamer-423556/>. Accessed: 25-09-2016.
- [2] Video game industry - statistics & facts - statista. <https://www.statista.com/topics/868/video-games/>. Accessed: 25-09-2016.
- [3] CARLOS MALTEZ A. "Historia y evolución del smartphone". Accessed: 25-09-2016.
- [4] Travis Fahs. "IGN Presents the History of Game Boy". Accessed: 25-09-2016.
- [5] Steven L. Kent. *The Ultimate History of Video Games*. Crown Archetype, 2002.
- [6] Lisanne Pajot and James Swirsky. "Indie Game: The Movie - A Video Game Documentary".
- [7] Marlene Simmons. Bertie the brain programmer heads science council. Ottawa Citizen. p. 17, 1975.