Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo de Fin de Grado

Análisis, diseño e implementación de un tutorial para resolver el cubo de Rubik

Autor: Jose Mario Castellano Cuesta
Tutor: César González Ferreras

Resumen

Actualmente, si una persona quisiera encontrar una solución al cubo de Rubik, tiene al alcance muchas páginas web que ofrecen el método básico para que cualquiera que quiera aprenderlo, pueda. Como es una solución genérica se puede aplicar para cualquier cubo. Sin embargo, los usuarios principiantes no están acostumbrados a la terminología empleada y a los movimientos del cubo, por lo que se suelen perder en algún paso y tienen que volver a empezar desde cero.

El programa realizado para este TFG amplía el trabajo de esas páginas web, ofreciendo la posibilidad de poner la configuración del cubo físico que tenga el usuario en un modelo 3D y ofrecer una solución personalizada donde se vea la resolución paso por paso y así mejorar el aprendizaje.

Abstract

Currently, if someone were to find a solution to the Rubik's Cube, they would have available lots of webpages that teach the beginners method to anyone willing to learn it. As it is a generic solution, it can be applied to any cube. However, newcomers are not used to the cube's moves and its terminology, so they may miss some steps and have to start from scratch again.

The program developed for this project extends the labor of these webpages, offering the possibility to transfer the configuration of their physical cube in a 3D model and give a customized solution where its steps can be navigated so that the learning process is improved.

Índice

Índice de figuras

1. Introducción

Prácticamente cualquier persona tiene un cubo de Rubik en su casa. Si esa persona quisiera intentar resolverlo por su cuenta le llevaría meses de estudio, por lo que se suele recurrir a buscar la solución en alguna página web.
Las páginas web que se encuentran en primera plana ofrecen una solución muy intuitiva una vez se ha entendido el funcionamiento del cubo y que además no requiere aprenderse un gran número de algoritmos.
Este programa está diseñado con los principiantes del mundillo del cubo de Rubik en mente, ofreciendo un tutorial con una mezcla de estos sistemas.

1.1. Motivación

Llevo siendo un fan del cubo de Rubik y los puzles mecánicos desde hace muchos años. Para solucionar por primera vez el cubo recurrí a páginas web que tuvieran el método para principiantes, pero como no estaba familiarizado con los movimientos y las interacciones que tienen las piezas entre sí, me costó más de lo que debía ya que me equivocaba en algún movimiento y tenía que volver a empezar.
Este TFG existe para intentar desarrollar un programa que me habría gustado tener en esa época, tanto para resolver el cubo por primera vez, como para seguir practicándolo.

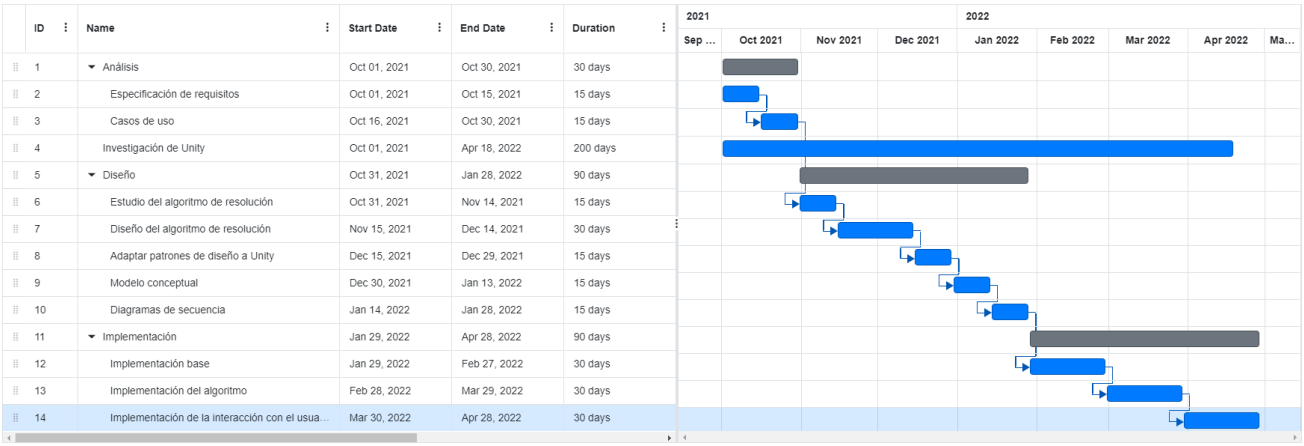
1.2. Objetivos

Como el programa está pensado para la gente que quiere aprender a solucionar el cubo de Rubik y no solamente encontrar una solución y dejarlo, el programa necesitará como los siguientes objetivos principales:

- **Cubo físico:**
  El programa tiene la misma intencionalidad que las páginas web mencionadas. Que el usuario tenga en las manos el cubo de Rubik mientras aprende o practica y utilice el software como guía.
- **Modelo en 3D:**
  Un modelo en el que el usuario pueda ver claramente los movimientos que se están ejecutando y mover el cubo sobre su eje para ver todas sus caras.
- **Explicaciones:**
  Una parte del programa dedicada enteramente a explicar los pasos a seguir en la resolución, dedicando apartados a cada caso que pueda ocurrir.
- **Personalización:**
  Este sería el objetivo sobre el que el programa debería girar. La posibilidad del usuario de colorear el modelo 3D del cubo para que esté igual que el que tiene presente y ofrecer una solución mientras aprende el método básico.
- **Aleatoriedad:**
  Este objetivo está dirigido a los usuarios que ya tengan más práctica con el cubo que no quieran dedicarle tiempo a la personalización y solamente quieran repasar. El programa les podría ofrecer una secuencia aleatoria para desarmar el cubo.

2. Planificación

El modelo de desarrollo del proyecto es el RAD (Rapid Application Model)[6]. Se ha escogido este modelo debido a que en este proyecto no hay un cliente en concreto al que entrevistare, por lo que los requisitos deberían estar bien definidos al principio y a elección del desarrollador. Este modelo viene bien si se quiere un desarrollo en el que el análisis y el diseño están completos y la implementación se hace de manera incremental.
En la Figura 1 se muestra un diagrama de Gantt de la planificación del proyecto



ID	Name	Start Date	End Date	Duration
1	Análisis	Oct 01, 2021	Oct 30, 2021	30 days
2	Especificación de requisitos	Oct 01, 2021	Oct 15, 2021	15 days
3	Casos de uso	Oct 16, 2021	Oct 30, 2021	15 days
4	Investigación de Unity	Oct 01, 2021	Apr 18, 2022	200 days
5	Diseño	Oct 31, 2021	Jan 28, 2022	90 days
6	Estudio del algoritmo de resolución	Oct 31, 2021	Nov 14, 2021	15 days
7	Diseño del algoritmo de resolución	Nov 15, 2021	Dec 14, 2021	30 days
8	Adaptar patrones de diseño a Unity	Dec 15, 2021	Dec 29, 2021	15 days
9	Modelo conceptual	Dec 30, 2021	Jan 13, 2022	15 days
10	Diagramas de secuencia	Jan 14, 2022	Jan 28, 2022	15 days
11	Implementación	Jan 29, 2022	Apr 28, 2022	90 days
12	Implementación base	Jan 29, 2022	Feb 27, 2022	30 days
13	Implementación del algoritmo	Feb 28, 2022	Mar 29, 2022	30 days
14	Implementación de la interacción con el usua...	Mar 30, 2022	Apr 28, 2022	30 days

Figura 1: Diagrama de Gantt del proyecto

3. El cubo de Rubik

El cubo de Rubik es un puzzle mecánico inventado por el profesor Erno Rubik en 1974. Originalmente se creó para enseñar a sus alumnos a adaptarse a los movimientos tridimensionales. En su momento, Rubik no tuvo un método de resolución para su creación hasta pasado un mes.
Un año después se patentó el cubo bajo el nombre de Cubo Mágico, nombre que estuvo vigente hasta que se distribuyó a nivel mundial.

Mientras el cubo estaba solamente distribuido en Hungría, el matemático David Singmaster lo descubrió en una conferencia de matemáticas en Helsinki[4]. Se fascinó por el cubo y elaboró un documento llamado "Notes on Rubik's Magic Cube"[7] que se centraba en un análisis basado en la teoría de grupos. Entre otras cosas, este documento introduce conceptos que se utilizan actualmente para resolver el cubo como darle nombres a las caras según su orientación o la resolución capa por capa, la cual es la que se utiliza en el programa de este proyecto.

El cubo de Rubik oficial tiene en las 6 caras los mismos colores y en la misma disposición y para todas las explicaciones dadas se toma como referencia dicha disposición.

El cubo de Rubik tiene 9 piezas (y 9 pegatinas) en cada cada cara. En total son 26 piezas (y 54 pegatinas) las cuales se pueden dividir en tres tipos: **centros, aristas** y **esquinas**. Para explicarlas mejor, en este trabajo se imagina que la cara de un cubo es una matriz de 3x3.

Los **centros** son piezas de un solo color y que no se pueden mover por lo que son las piezas que van a definir el color de las caras. Sus coordenadas dentro de la cara son [1, 1]. Si por ejemplo decimos que una cara es naranja significa que el centro es naranja, aunque en ese momento no haya ninguna otra pieza del mismo color.

Las **aristas** son piezas con dos colores por lo que se encuentran en dos caras a la vez. Sus coordenadas son [0, 1], [1, 0], [1, 2] y [2, 1]. Cada arista conecta con las coordenadas [0, 1] de sus caras contiguas suponiendo que la cara que tomamos como referencia fuera la cara superior.

Las **esquinas** son piezas con tres colores. Se encuentran en tres caras a la vez y sus coordenadas son [0, 0], [0, 2], [2, 0] y [2, 2].

En la Figura 2 se encuentran ilustrados estos conceptos.

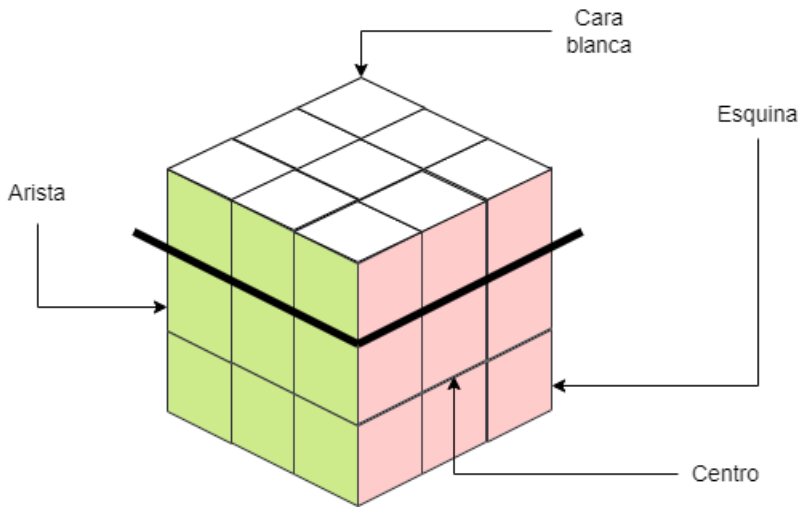


Figura 2: Cubo de Rubik y sus piezas

Las esquinas y las aristas son únicas, no van a haber dos piezas con la misma combinación de colores.
Evidentemente en cada pieza no se puede repetir el mismo color, si se diera el caso significaría que hay más de una cara con ese color.

Los movimientos están definidos por giros horarios o antihorarios alrededor del centro de una cara. Los giros son de 90 o 180 grados. Cada movimiento cambia la orientación de la cara y tres colores de las cuatro caras contiguas.

4. Herramientas utilizadas

Este apartado está dedicado a las herramientas utilizadas para este proyecto. Todas son gratis ya sea porque la Universidad ofrece licencias o porque ya lo son de por sí.

- **Unity:** El motor con el que funciona el programa. Se ha escogido este motor ya que es el más sencillo de utilizar si no se quieren hacer modelos 3D muy complicados. Además es gratis si la empresa o autónomo que lo utilice no ha ganado más de 100.000 $ el año fiscal anterior o si se pretende hacer un proyecto no comercial.
- **C#:** El lenguaje de programación por defecto de Unity. Se puede usar algún otro lenguaje mediante plugins, pero puede que dé errores, además de que el motor está optimizado para C#.
- **Visual Studio:** El IDE. Puede integrarse con Unity para gestionar mejor los scripts y compilar el proyecto. (Esta funcionalidad ya la tiene Unity de por sí, pero de nuevo, si se quiere hacer un scripting mínimamente complejo, con Visual Studio es más manejable).
- **Astah UML:** Es el programa utilizado para crear los diagramas UML.
- **draw.io:** Una página web para hacer dibujos. Es sorprendentemente útil para crear esquemas.
- **onlinegantt.com:** Una página web para crear diagramas de Gantt online para planificar el proyecto.
- **TeXstudio:** El editor de textos utilizado para redactar la memoria del TFG.

5. Unity

Se dedica un apartado a Unity ya que esta herramienta es la que más ha condicionado la forma de abordar este proyecto.
Unity es un motor de videojuegos desarrollado por Unity Technologies lanzado en 2005. Es fácil aprender su funcionamiento y es gratis por lo que es muy popular entre desarrolladores independientes de videojuegos.
La documentación de Unity se encuentra en su página web[8] y también se pueden encontrar en YouTube algún tutorial sencillo para saber por dónde empezar[5].

5.1. Game Objects

Es la clase base para las entidades de Unity. Pueden ser por ejemplo objectos en 3D, 2D o elementos de una interfaz de usuario. Lo más importante de esta clase son las clases que tiene agregadas: los Components.

5.2. Components y Transforms

Los Component son la clase base de todo lo que va a agregado a un GameObject y son los que contienen su información y los que van a determinar su comportamiento.

Los Transform heredan de Component y tienen información de la posición, rotación y escala y métodos para modificarlos, por lo que si se quiere actualizar alguno de estos valores del GameObject, se actualizan desde aquí. Más adelante se verá en profundidad cómo se pueden implementar estos cambios si por ejemplo queremos que un GameObject se mueva en la escena.
Otra utilidad para Transform es poder crear GameObjects padres e hijos. Cuando un GameObject es hijo de otro, todas las modificaciones en el Transform que se hagan en el padre, se trasladarán también a las del hijo.

Hay más clases que heredan de Component como MeshRenderer que dice cómo se tiene que renderizar en pantalla el GameObject o BoxCollider que interactúa con el motor de físicas de Unity. Pero el que más importa son los scripts.

5.3. Scripts

Los scripts heredan de Monobehaviour que hereda de Behaviour que hereda de Component como se ve en la Figura 3. Los Components de por sí tenen la versatilidad suficiente, pero si se quiere un programa con cierta complejidad hay que recurrir a los scripts. Aquí es donde vendría el código escrito en C#.
Los scripts manejan eventos del programa, modifican otros Components e incluso crear nuevos.
Lo que podría considerarse como los métodos más importantes de los scripts agregados a los GameObjects son las Funciones de Evento, que vienen definidas en la clase MonoBehaviour (y que por lo tanto los scripts los heredan).

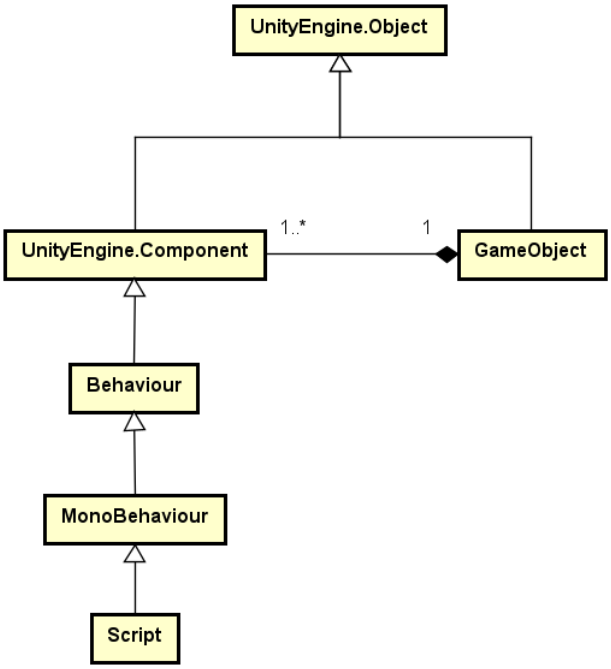


Figura 3: Diagrama de clases de los scripts

Importante aclarar que aunque sean similares, los scripts no son como el resto de clases escritas en C# ya que, de nuevo, estos van agregados a GameObjects además de heredar los métodos de MonoBehaviour.

5.4. MonoBehaviour y Funciones de Evento

MonoBehaviour es la clase de la que todos los scripts heredan (y por lo tanto tiene que estar explícitamente escrito en el código). Esta clase define las Funciones de Evento. Algunas de estas funciones son: Start o Update.

El motor de Unity tiene una lista con los GameObjects existentes y cada vez que se quiera mostrar un fotograma en pantalla, antes de mostrarlo pasa por esa lista y llama a las Funciones de Evento que tienen los scripts asociados a esos GameObjects. En la Figura 4 vienen resumidas las funciones existentes y el orden en el que se llaman

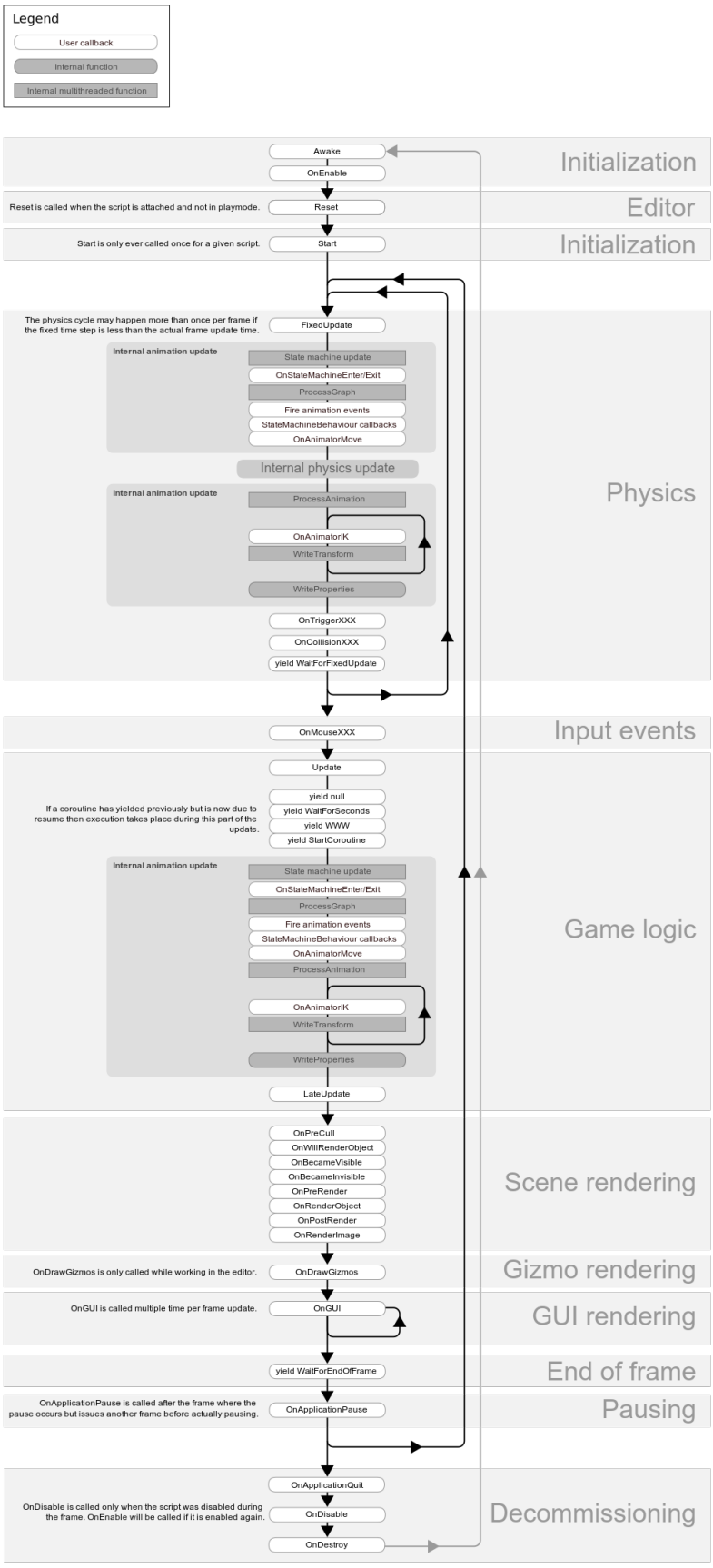


Figura 4: Orden de llamada a las funciones de evento

Un ejemplo para entender con más sencillez estas funciones sería explicar Update.
Dentro del script está definido el método Update que hereda de MonoBehaviour. En este método, por ejemplo, tenemos escrito que el GameObject al que está agregado el script se mueva un pixel a la izquierda. Como este método está escrito dentro de una función de evento, cuando el programa esté en ejecución, el motor de Unity llamará a esta función por cada fotograma que se muestre en pantalla (evidentemente si dicho GameObject está activo en ese momento). Esto significa que si el programa funciona a 60 fotogramas por segundo, el GameObject se moverá 60 píxeles a la izquierda por cada segundo.
Como aclaración a Update. Se puede observar que el hecho de que el funcionamiento de los GameObject vaya asociado a los fotogramas mostrados puede generar problemas, ya que estos no tienen porqué ser estables ni todas las máquinas pueden soportar ese ritmo. Por eso Update se utiliza para animaciones y para interacciones con el usuario. Cuando se quiera manejar las físicas se utiliza FixedUpdate, ya que la periodicidad con la que se llama es fija.
Hay una pregunta muy interesante que se puede formular y es por qué no usar siempre FixedUpdate y en verdad técnicamente se podría, pero hay que tener en cuenta de hay ciertas animaciones que puede que funcionen mal y puede haber ocasiones en las que el programa no detecte algún input por parte del usuario[9].

5.5. Coroutines

Hay un problema con las Funciones de Evento y es que si se quiere realizar algún método de forma muy casual, es decir, que se determine que no sea eficiente comprobar si ese método debe suceder, o si es un método pesado, estas funciones consumen muchos recursos.
Para esto existen las Coroutines. Ejecutan tareas que se llevan a cabo durante varios fotogramas, es decir, no tiene porqué completarse antes de que llegue el siguiente fotograma.
Volviendo al ejemplo anterior de un GameObject moviéndose a la izquierda, usar Update es la elección adecuada si ese GameObject va a estar moviéndose permanentemente. Sin embargo, si queremos que solo se mueva durante un tiempo determinado o cuando el usuario pulse algún botón en concreto, se están gastando recursos en comprobarlo. También puede darse el caso de que el método sea muy complejo no dé tiempo a ejecutarlo entero antes del siguiente fotograma sin que la tasa de FPS decaiga.

Para profundizar más en la diferencia entre escribir una función en Update y en un Coroutine, nos sirve el segundo ejemplo puesto en este subapartado: un GameObject que se mueve durante un tiempo determinado cuando el usuario pulse un botón.
En ambos casos se puede escribir un detector de inputs dentro de Update. En el caso de la implementación basada en el Update, habría que escribir a mayores una máquina de estados que diga si el GameObject ya se está moviendo o no y así no resetear la función y que el GameObject se mueva más tiempo del debido si el usuario vuelve a pulsar el botón antes de que pare. Con Coroutine, se lanza el método y no se vuelve a lanzar hasta que termine su ejecución.

5.6. Prefabs

El sistema de prefabs de Unity permite guardar y manipular un GameObject con sus Componentes definidos. Este sistema sirve para agilizar el proceso de creación de un proyecto, especialmente si se tiene en cuenta que un GameObject puede ser padre de otros GameObjects.

6. Análisis

En este capítulo se presenta el análisis del proyecto. Al no haber un cliente en concreto, los requisitos no se pueden especificar mediante una entrevista sino que están basados en lo que el desarrollador cree que un potencial usuario pueda necesitar.
A continuación se presentan los requisitos funcionales, no funcionales y los casos de uso.

6.1. Requisitos funcionales

- **RF1 - Visualización 3D:** El sistema deberá mostrar un modelo 3D del cubo de Rubik
- **RF2 - Movimiento propio:** El sistema deberá poder girar las caras del cubo 3D
- **RF3 - Conocimiento propio:** El sistema deberá poder reconocer el estado del cubo
- **RF4 - Interactividad con el cubo:** El sistema deberá permitir al usuario rotar el cubo 3D
- **RF5 - Modos de aprendizaje:** El sistema deberá permitir que el usuario seleccione el tipo de aprendizaje que quiera
- **RF6 - Aleatoriedad:** El sistema deberá ser capaz de generar una secuencia aleatoria de movimientos para el cubo
- **RF7 - Colorear:** El sistema deberá permitir al usuario colorear la piezas del cubo
- **RF8 - Resolución:** El sistema deberá ser capaz de resolver el cubo de Rubik
- **RF9 - Interactividad con la resolución:** El sistema deberá permitir al usuario navegar por los movimientos de resolución
- **RF10 - Textos explicativos:** El sistema deberá tener textos que expliquen los procesos de resolución

6.2. Requisitos no funcionales

- **RNF1 - Motor del programa:** El sistema se creará con Unity
- **RNF2 - Lenguaje de programación:** El programa usará C# como lenguaje de programación
- **RNF3 - Tiempo de respuesta:** El sistema deberá tardar menos de 1 segundo en responder
- **RNF4 - Idioma:** La aplicación deberá estar en castellano con posibilidad de traducir a otros idiomas
- **RNF5 - Instalación:** La instalación de la aplicación se hará descomprimiendo el archivo
- **RNF6 - Usabilidad:** La aplicación será fácil de utilizar
- **RNF7 - Rendimiento:** La aplicación tendrá un coste de computación bajo
- **RNF8 - Sistema operativo:** La aplicación será compatible para Windows

6.3. Casos de uso

El diagrama de casos de uso se encuentra en la Figura 5

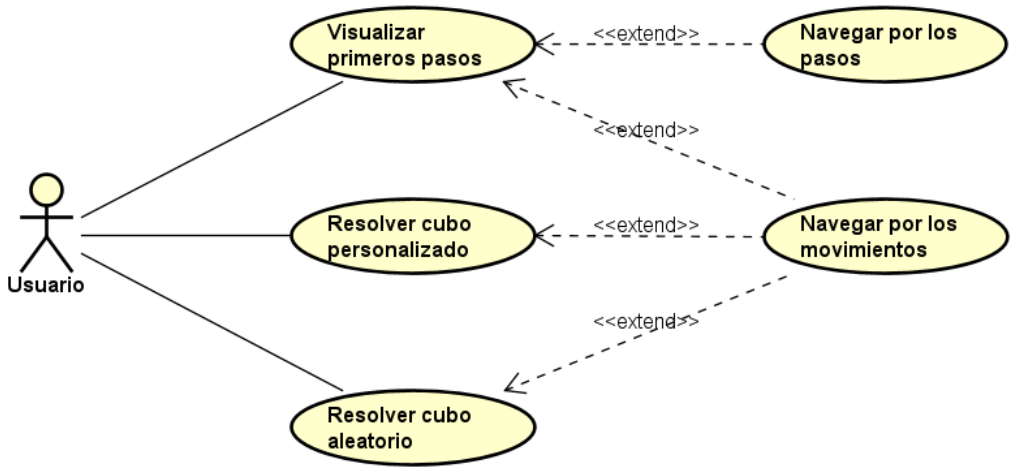


Figura 5: Diagrama de casos de uso

Las siguientes tablas sirven para detallar los casos de uso

Caso de uso	1 - Visualizar primeros pasos	
Descripción	El actor visualiza los primeros pasos para aprender a resolver el cubo	
Actor	Usuario	
Precondición		
Secuencia base	1	El actor selecciona el método de aprendizaje de los primeros pasos
	2	Ref. CU-4
	3	Ref. CU-5
Postcondición		
Secuencia rama	1.1	El actor decide volver al punto inicial del programa
Secuencia excepción		

Caso de uso	2 - Resolver cubo personalizado	
Descripción	El actor personaliza el cubo para resolverlo	
Actor	Usuario	
Precondición		
Secuencia base	1	El actor selecciona la opción de personalizar el cubo
	2	El sistema muestra el menú correspondiente
	3	El usuario selecciona un color y una pieza del cubo
	4	El sistema registra esa modificación
	5	El usuario selecciona la opción de resolver
	6	El sistema encuentra una solución para la configuración actual del cubo
	7	Ref. CU-5
Postcondición	El cubo queda resuelto	
Secuencia rama	1.1	El actor decide volver al punto inicial del programa
	5.1	El actor vuelve a seleccionar colores y piezas
Secuencia excepción	6.1	El cubo está mal configurado y no se puede encontrar una solución
	6.2	El sistema avisa al usuario para volver al paso 3

Caso de uso	3 - Resolver cubo aleatorio	
Descripción	El sistema genera un cubo aleatorio y lo resuelve	
Actor	Usuario	
Precondición	El cubo queda resuelto	
Secuencia base	1	El actor selecciona la opción de cubo aleatorio
	2	El sistema genera una secuencia aleatoria de movimientos y la resuelve
	3	El sistema muestra al usuario dicha secuencia
	4	Ref.CU-5
Postcondición	El cubo queda resuelto	
Secuencia rama		
Secuencia excepción		

Caso de uso	4 - Navegar por los pasos	
Descripción	El actor navega por los pasos del tutorial	
Actor	Usuario	
Precondición	Se ha realizado el CU-1	
Secuencia base	1	El sistema muestra los pasos disponibles
	2	El actor selecciona el paso deseado
	3	El sistema determina la configuración de dicho paso
	4	Ref. CU-5
Postcondición		
Secuencia rama	2.1	El actor vuelve a seleccionar otro paso
Secuencia excepción		

Caso de uso	5 - Navegar por los movimientos	
Descripción	El actor navega por los movimientos del cubo	
Actor	Usuario	
Precondición	Haber realizado CU-1, CU-2 o CU-3	
Secuencia base	1	El actor selecciona ver el siguiente movimiento
	2	El sistema muestra el siguiente movimiento
Postcondición		
Secuencia rama	1.1.1	El actor selecciona ver el anterior movimiento
	1.1.2	El sistema muestra el anterior movimiento
	2.1	El sistema muestra un texto descriptivo y vuelve al paso 2
	2.2	El sistema cambia la cara frontal por defecto del cubo y vuelve al paso 2
Secuencia excepción		

7. Arquitectura

La arquitectura del proyecto es la propuesta por Unity.

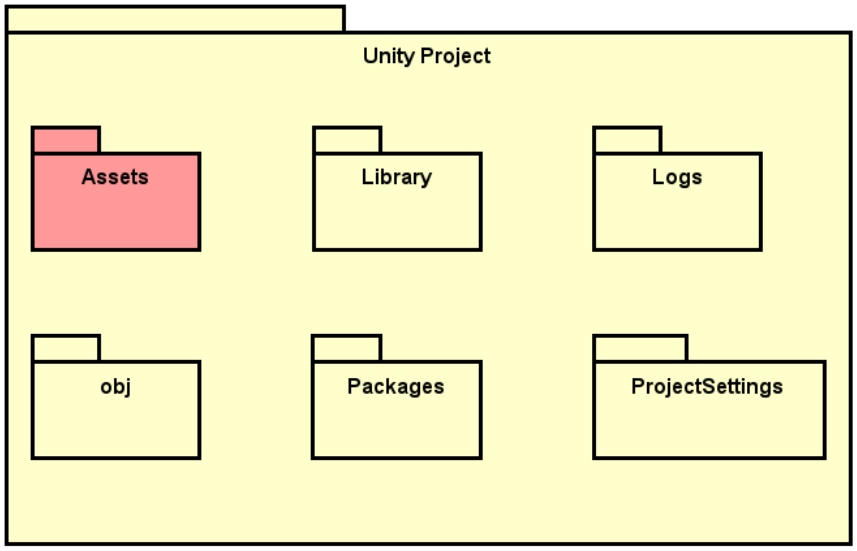


Figura 6: Diagrama de la arquitectura general

En la Figura 6 se muestra la arquitectura general de un proyecto cualquiera creado en Unity. Si no se quiere modificar mucho las configuraciones básicas, al usuario medio solo le importará la carpeta Assets, donde estará guardado todo lo que haya escrito y hecho para el proyecto. En el resto de carpetas se encontrarán bibliotecas, logs y demás.

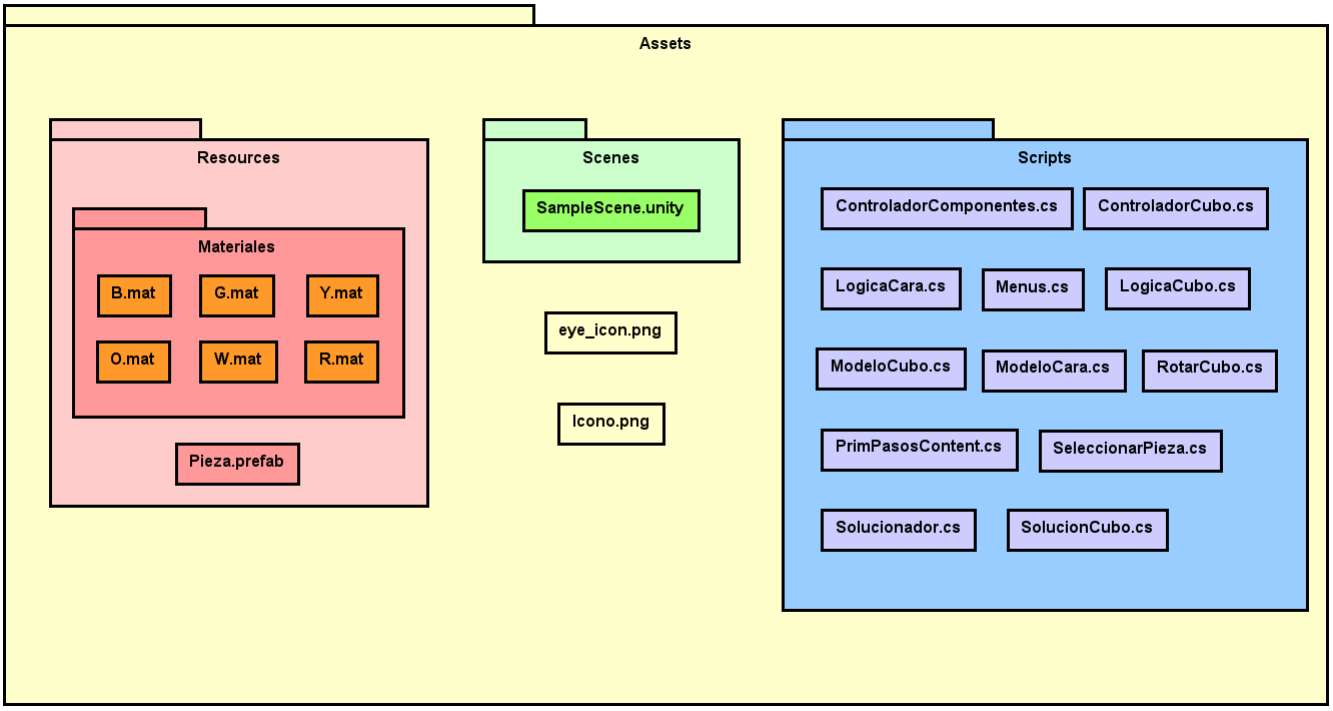


Figura 7: Diagrama de la arquitectura de Assets

En la Figura 7 está detallada más a fondo la arquitectura de Assets. En ella se ven tres componentes principales: Resources, Scenes y Scripts.

En Resources están guardados todos los componentes relacionados con los modelos 3D del proyecto. Los materiales son las texturas de los modelos 3D. Los seis archivos .mat representan los seis colores del cubo.
Un archivo .prefab es un conjunto de polígonos básicos definidos por el diseñador 3D con el fin de reutilizarlo varias veces. Este prefab en concreto es una pieza básica del cubo. Básicamente es un cubo negro con seis poliedros rectangulares por encima actuando como pegatinas. El cubo estará compuesto por 27 prefabs de piezas ocultando las pegatinas según toque.

En Scenes se guardan las escenas. Las escenas sirven para estructurar mejor el proyecto. Hay que recordar que Unity es un motor pensado para hacer videojuegos, así que las escenas están pensadas para que los distintos niveles de un juego se puedan organizar de una forma más sencilla. Como este proyecto solo requiere de una escena, se usará la que genera Unity por defecto.

En Scripts es donde están guardados los archivos C# que son los clave para el funcionamiento del programa. Se hablará más en detalle de estos archivos en el apartado de diseño.

Los dos archivos sueltos en Assets son dos imágenes. Una es para colocarla en la interfaz y otra servirá como icono en el archivo ejecutable

7.1. Despliegue

El despliegue de un proyecto de Unity es tan sencillo que no hace falta un diagrama. Dentro del editor se pueden configurar los detalles para la construcción del programa (como por ejemplo, el sistema operativo en el que va a correr o el propio nombre de la apliacion) y Unity generará un archivo comprimido. En este archivo habrá una carpeta que contendrá las librerías dinámicas necesarias y las escenas creadas en el proyecto (en este caso, una) ya compiladas. Además de esta carpeta se crea un archivo .exe para ejecutar la aplicación.
El archivo comprimido se puede descomprimir en cualquier dispositivo compatible con el desplegable y estaría listo para ejecutar sin ningún otro paso.

8. Diseño

8.1. Diseño del algoritmo de resolución

Para tener un tutorial completo, se tiene que poder encontrar una solución al cubo de Rubik. En este apartado se explica el algoritmo de resolución que se implementará en el programa.
La información sacada para diseñar el algoritmo corresponde a varias páginas web[3][2] y al conocimiento que tenía de antes, que al haber pasado tantos años, la página web que consulté en su día ya no existe y no puedo referenciarla.

8.1.1. Notación

Para empezar a analizar el algoritmo hay que introducir la notación utilizada.
Las seis caras del cubo están referenciadas con una letra. Esa letra dependerá de la posición relativa a la perspectiva del usuario con el cubo, teniendo una cara paralela al suelo y otra frente a la vista. Las caras están nombradas por: **F** (Front), **U** (Up), **L** (Left), **R** (Right), **D** (Down), **B** (Back).
Cuando se quiera formular un algoritmo, se necesita especificar primero cuál es la cara frontal (F) y cuál es la cara superior (U), ya que esas son las dos mínimas con las que se pueden suponer el resto.
Para representar un movimiento hay tres opciones:

- La letra de la cara sola (p. ej. **U**). Significa que esa cara se tiene que girar en sentido horario 90 grados.
- La letra de la cara seguida por un apóstrofo (p. ej. **U'**). La cara se gira en sentido antihorario 90 grados.
- La letra de la cara seguida del número 2 (p. ej. **U2**). La cara se gira 90 grados dos veces en el mismo sentido o lo que es lo mismo, se gira 180 grados.

Hay más notación dependiendo de si se quieren incluir cambios de perspectiva o demás, pero no se necesitan para este trabajo. Sin embargo, para el diseño del algoritmo he expandido la notación.

8.1.2. Expansión de la notación

Ya que se necesitan generalizar los algoritmos, habrá veces en las que no se sepa cuántas veces se tiene que girar una cara y en qué dirección, así he expandido la notación para que se puedan incluir variables. La estructura es la siguiente: Inmediatamente después del nombre de la cara se escribe entre paréntesis una variable (p. ej. **U(X)**). Esta variable está definida fuera de la secuencia y puede tomar los valores de: Girar en sentido horario, girar en sentido antihorario, girar 180 grados o no girar.
La otra regla introducida es la posibilidad de negar una variable. Se introduce el símbolo de negación lógica (¬) inmediatamente antes de una variable (p. ej. **U(¬X)**). Esa cara se girará en sentido contrario al que indica la variable. Si la variable dijese que la cara se gira en sentido horario, su negación implica que se tendría que girar en sentido antihorario y viceversa. En el caso en el que la variable indicase girar 180 grados o no girar, su contrario significaría lo mismo.

Un ejemplo para concluir la explicación de la notación: **U L' U(X) R U(¬X)**

Explicación: (Previamente definidas la cara frontal, superior y el valor de X) Girar cara superior en sentido horario, Girar la cara izquierda en sentido antihorario, Girar la cara superior en sentido X, Girar la cara derecha en sentido horario, Girar la cara superior en sentido negación de X.

8.1.3. Algoritmo general

El algoritmo de resolución está compuesto por siete partes principales. En el siguiente diagrama de flujo en la Figura 8 está explicada la secuencia general.

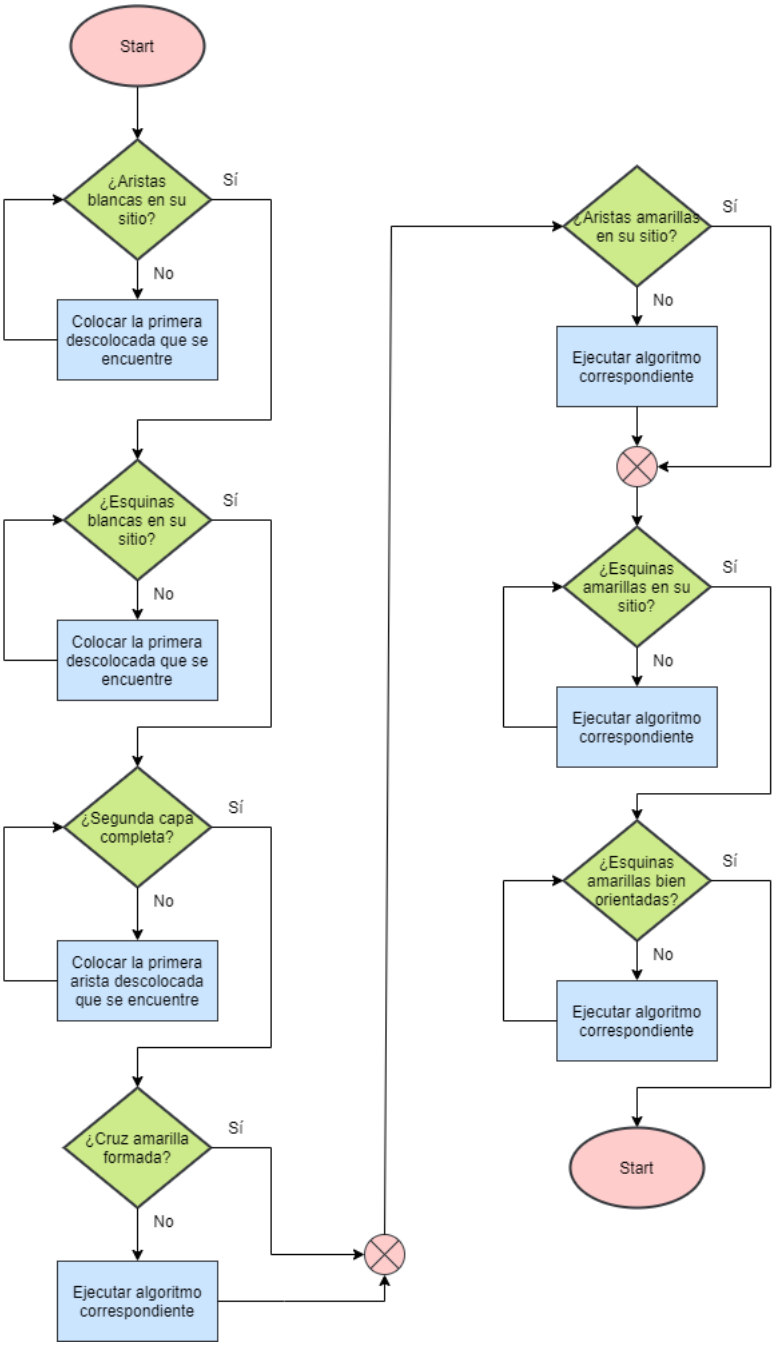


Figura 8: Diagrama de flujo del algoritmo de resolución general

En este gráfico se han ignorado las caras superiores para que sea más claro, pero se puede tener en cuenta que la cara superior en los cuatro primeros pasos es la blanca y en los tres últimos pasos, la amarilla. La elección de esas caras es arbitrario pero si se eligieran otras, tiene que cumplirse que las dos sean opuestas.

Otro punto a anotar es que el algoritmo diseñado no está solamente para solucionar el cubo de Rubik, sino que está hecho para facilitar el aprendizaje, ya que se incluyen movimientos extra para que al usuario le quede más claro el porqué de ciertas secuencias (como se podrá ver más adelante en el segundo paso en el caso de que una esquina blanca esté en la parte inferior de una cara lateral) además de que hay soluciones mucho más óptimas.

Hay un apartado dedicado para cada paso, pero se ve que la secuencia general es sencilla. Se formula

una pregunta y si la respuesta es negativa se ejecuta el algoritmo correspondiente a esa parte hasta que la respuesta sea positiva.
Cada paso itera sobre lo conseguido en el anterior teniendo cuidado de no descolocar las piezas que ya están correctas. Los siete pasos son:

- Cruz blanca
- Esquinas blancas
- Segunda capa
- Cruz amarilla
- Posición de las aristas amarillas
- Posición de las esquinas amarillas
- Orientación de las esquinas amarillas

8.1.4. Paso 1: Cruz blanca

El primer paso consiste en hacer una cruz blanca, es decir, colocar las aristas blancas en la cara blanca. También hay que tener en cuenta el color contiguo a la arista blanca y hacer que coincida con su cara respectiva. Por ejemplo: Con la arista blanca-naranja, el color blanco tiene que estar situado en la cara blanca y el color naranja tiene que estar a la vez situado en la cara naranja.
En este paso se puede ignorar por completo cualquier otra pieza que no sea una arista blanca, como se puede ver en la Figura 9

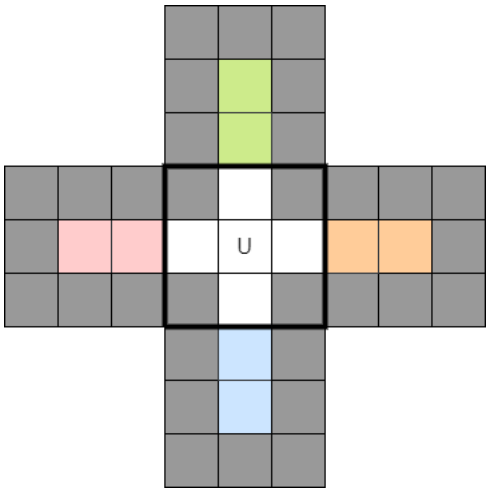


Figura 9: Esquema del cubo con la cruz blanca

El mayor problema con diseñar un algoritmo de este paso es que se suele realizar de forma intuitiva, por lo que no hay un algoritmo directamente aplicable. Lo que he podido hacer es organizar los casos que se pueden dar y a partir de ellos, qué secuencia de movimientos se puede hacer, como se puede ver en la Figura 10. No hay ningún sitio que explique este paso con la notación ya que sería más complejo de entenderlo así que descrito de forma natural. Además tendría que recurrir a la notación expandida que he inventado para este trabajo.

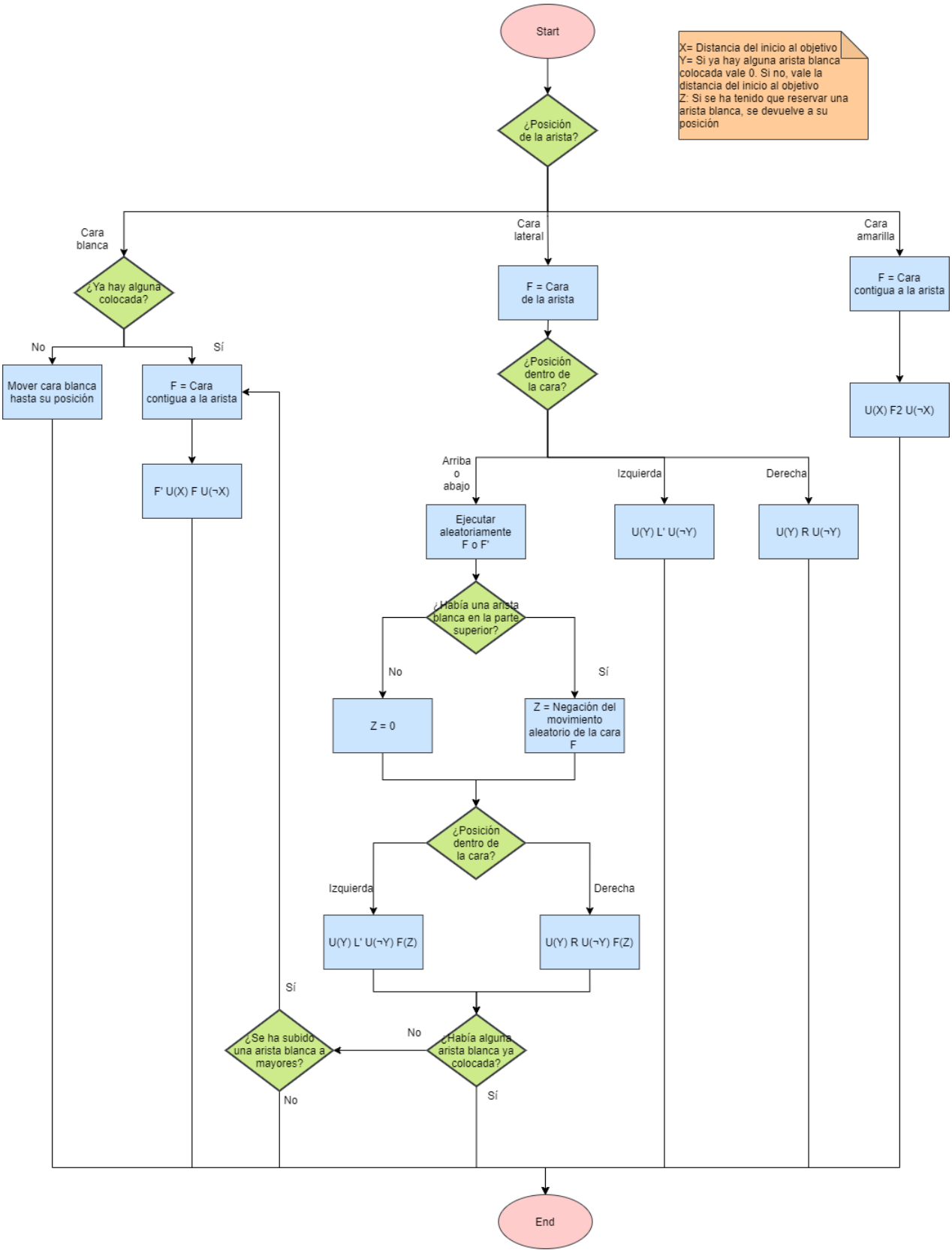


Figura 10: Diagrama de flujo de la cruz blanca

Fuera del algoritmo, se hace la primera pregunta de la Figura 8 '¿Aristas blancas en su sitio?'.

Hasta que no estén las cuatro colocadas, esté algoritmo se repetirá. Una vez entra en el algoritmo busca una arista blanca aleatoria y determina su posición. Hay tres posibilidades: Si el color blanco de la arista está en la cara blanca, si está en alguna cara lateral o si está en la cara amarilla.

Si la arista está en la cara blanca hay dos opciones sencillas. Si es la única arista que está colocada, se gira la cara blanca hasta que llegue a su posición correcta.
En el caso de que sí que la hubiera, se ejecuta lo que se llama reservar una arista. Esto es, se gira la cara en la que se encuentre el color no blanco de la arista para así poder mover la cara blanca sin modificar el resto de las aristas. En el algoritmo se utiliza una variable X. Ese es el número de giros que tiene que hacer la cara blanca. Una vez girada la cara blanca, se quita la arista blanca de la reserva y se vuelve a colocar la cara blanca en su sitio.
Ejemplo: Una arista blanca-naranja está colocada entre la cara roja y la blanca. La arista se reserva girando la cara roja. A continuación, la cara blanca se gira dos veces, ya que la distancia de movimientos entre la cara roja y la naranja es de 180 grados (X). Se quita la arista de la reserva girando de nuevo la cara roja y se vuelve a girar dos veces la cara blanca.

Si la arista está en la cara amarilla, se hace un proceso similar al último explicado pero aquí no hace falta recurrir a las reservas. Se gira la cara blanca la distancia equivalente a las dos caras, se sube la arista desde la cara amarilla a la blanca con dos giros y se devuelve la cara blanca a su posición original.
Ejemplo: Una arista blanca-naranja está colocada entre la cara verde y la amarilla. La cara blanca se gira una vez en sentido antihorario, ya que la distancia de movimientos entre la cara verde y la naranja es de 90 grados (X). Se sube la arista blanca a la cara blanca con un giro de 180 grados y se vuelve a girar la cara blanca una vez en sentido horario para devolverla a su posición original.
Aclaración: El algoritmo utilizado es U(X) F2 U(¬X). También se podría usar D(X) F2 D(¬X) que es similar.

El caso en el que la arista blanca esté en una cara lateral es el caso más complejo. Si la arista dentro de la cara se encuentra a la izquierda o a la derecha, se aplica el algoritmo correspondiente. La variable Y existe por si se da el caso en el que todavía no haya ninguna arista colocada correctamente, ya que así se ahorran movimientos, pero podría considerarse opcional. La distancia de movimientos se calcula de la misma manera que con la variable X. Es más, el algoritmo es el mismo que en el caso de que de la arista esté en la cara blanca con alguna pieza ya colocada. La diferencia es que aquí no hay que reservar la arista porque técnicamente ya se encuentra reservada.
En el caso de que la arista blanca dentro de la cara se encuentre arriba o abajo, la cara frontal se gira en sentido horario o antihorario (aleatoriamente, no importa en qué dirección) para que así se encuentre a la izquierda o a la derecha y se aplique el mismo algoritmo. Sin embargo, puede darse el caso en el que ya había una arista colocada correctamente en la parte superior, así que una vez terminado el algoritmo, hay que volver a girar la cara. Esto está determinado mediante la variable Z. Con la variable Z sucede algo parecido a la variable Y, es una variable que existe para ahorrarse algún movimiento. Si no hubiera una arista ya colocada no hace falta volver a girar la cara frontal.
Finalmente, puede suceder que si no había ninguna arista colocada correctamente, se puede subir de manera accidental una arista blanca a mayores. (Para que se diera este caso, dicha arista tendría que haberse encontrado inicialmente en la parte derecha de la L o en la parte izquierda de la R. Como es un caso muy específico y fácil de resolver ni siquiera se tiene en cuenta en las tomas de decisiones anteriores). Si ha sucedido, se aplica el algoritmo en el que una arista blanca está en la cara blanca y ya hay alguna colocada.
Ejemplo: Una arista blanca-naranja entre la cara roja y la blanca. Eso implica que la arista se encuentra en la parte de arriba de la cara roja. Se gira la cara roja en sentido horario, se gira la cara blanca una vez en sentido horario, ya que es la distancia que hay entre la cara naranja y la verde. Se gira la cara verde en sentido horario y se vuelve a girar la cara blanca en sentido antihorario para devolverla a su posición original.

El resumen es que este algoritmo es menos complejo de lo explicado si se hace de forma intuitiva

interiorizando el funcionamiento del cubo, pero en este caso hay que formalizarlo para que el programa lo pueda resolver.

8.1.5. Paso 2: Esquinas blancas

El segundo paso consiste en colocar las esquinas blancas en su sitio teniendo cuidado de que el resto de los colores coincidan con el de sus caras (y no descolocar las aristas blancas). El problema de formalizar este paso a un algoritmo definido es el mismo que en el primer paso: que se resuelve de manera intuitiva. De nuevo no hay ningún sitio que se pueda referenciar y hace uso de la notación expandida. El diagrama de flujo resultante se encuentra en la figura 11.

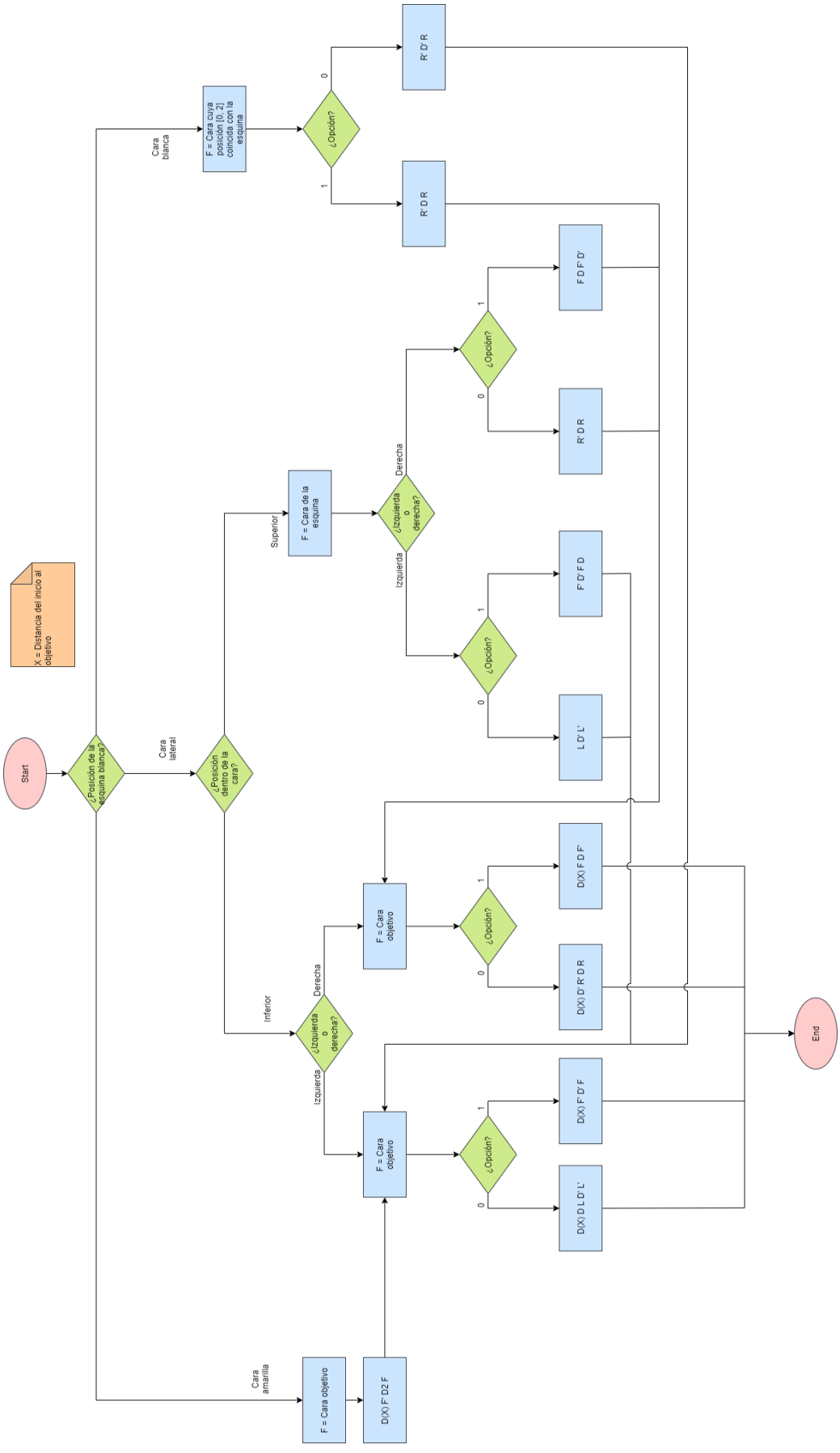


Figura 11: Diagrama de flujo de las esquinas blancas

Este diagrama es el más largo porque se incluyen varias opciones dentro de algunos casos.
El caso principal es cuando una esquina se encuentra en una esquina lateral, ya que los casos en los que se encuentre en la cara amarilla o la cara blanca derivarán en él. Lo primero es comprobar si se encuentra en la parte inferior o superior de la cara y después si está a la izquierda o a la derecha. Si está en la parte inferior izquierda o derecha (coordenadas [2, 0] o [2, 2]), se mueve la cara amarilla X veces hasta que la esquina esté inmediatamente debajo de la posición que le correspondería. A partir de ahí se hace una reserva, pero hay dos maneras de reservarlo. Una es moviendo la cara amarilla y otra es moviendo la cara frontal. En el diagrama se ve que se toma la opción 0 o 1, la cual es totalmente aleatoria. Una vez reservada, es bajar su posición correspondiente a la cara amarilla y sustituir la esquina que se encuentra en ella.
Ejemplo: Una esquina blanca-azul-naranja se encuentra en la parte inferior derecha de la cara verde. Se gira la cara amarilla 180 grados, ya que para que la esquina esté inmediatamente debajo de su posición correspondiente tiene que encontrarse en la cara azul. Hay dos opciones a continuación:

- Reservar la esquina girando la cara amarilla en sentido antihorario y ponerla en su posición bajándola al girar la cara naranja en sentido antihorario, girando la cara amarilla en sentido horario y devolviéndola a su posición girando en sentido horario.
  Para este caso, si se analiza algo más en profundidad, se ve que hay un paso extra. En vez de girar inicialmente la cara amarilla 180 grados y luego -90, se podría haber girado 90 y ya, pero se realiza de este modo para que el usuario internalice el porqué de estos movimientos.

- Reservar la esquina girando la cara frontal (azul) en sentido horario. Ponerla en su posición girando la esquina amarilla en sentido horario y devolverla a su sitio volviendo a girar la cara frontal en sentido antihorario.

Si la esquina se encuentra en una cara lateral en la parte superior (coordenadas [0, 0] o [0, 2]), la intención es bajarla a la parte inferior para luego resolverla como tal. De nuevo, hay dos opciones y se escoge la secuencia que se quiera.
Ejemplo: Una esquina blanca-azul-naranja se encuentra en la parte superior derecha de la cara verde. Para bajarla a la parte inferior derecha hay dos opciones:

- Bajar la esquina girando la cara derecha en sentido antihorario, girar la cara amarilla en sentido horario y volver a subir la cara derecha en sentido horario. Con estos tres movimientos la esquina ha quedado en la parte inferior derecha y se procede a ejecutar su algoritmo correspondiente.

- Bajar la esquina girando la cara frontal (verde) en sentido horario, reservar la esquina girando la cara amarilla en sentido horario y devolver la cara frontal a su posición girándola en sentido antihorario. De nuevo, la esquina ha quedado en la parte inferior derecha.

Si la esquina se encontrase en la cara amarilla, la secuencia inicial consiste en pasarla a la parte inferior izquierda de una cara lateral. Después de eso, se resuelve como tal.

Si la esquina se encontrase en la cara blanca (pero no en su posición correspondiente) se baja la esquina a la parte inferior de una cara lateral. Dependiendo de la opción escogida, estará en la izquierda o en la derecha y se resolverá como tal.

8.1.6. Paso 3: Segunda capa

A partir del paso 3 los algoritmos son más sencillos porque hay menos casos posibles y las secuencias son más estrictas.
Actualmente, con los dos pasos anteriores realizados, la cara blanca está completa y los colores superiores de las caras laterales coinciden con su centro.
El objetivo de este paso es tener colocada la segunda capa. La segunda capa serían las cuatro aristas

que no corresponden a la cara blanca o la cara amarilla (p. ej. arista roja-azul). Así, quedarían colocadas correctamente las coordenadas [1, 0] y [1, 2] de cada cara.
El diagrama de flujo de este paso se encuentra en la Figura 12.

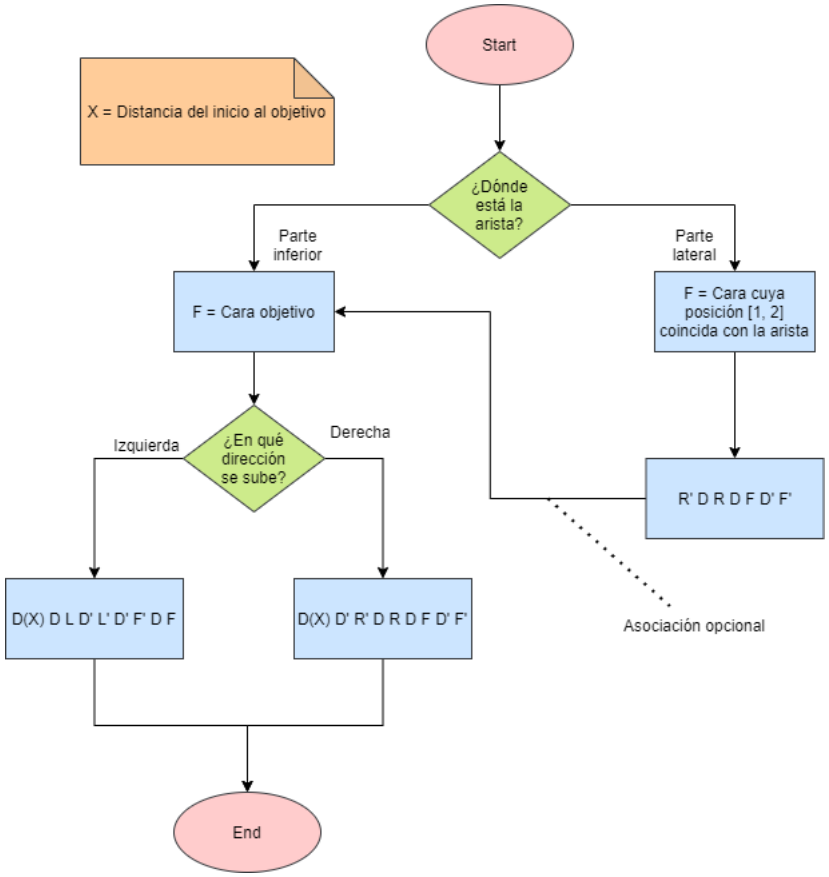


Figura 12: Diagrama de flujo de la segunda capa

Los pasos son sencillos. Se observa la parte inferior de las caras laterales buscando las aristas que corresponderían a la segunda capa. (Como aclaración, las aristas que se buscan son: roja-azul, azul-naranja, naranja-verde y verde-roja). Si se encuentra una, se gira la cara amarilla hasta que el color de la arista coincida con el centro (**D(X)**). Dependiendo de si su posición correcta se encontrara en la parte izquierda o en la parte derecha de la cara, se ejecuta la secuencia de movimientos que toque. El otro caso que se puede dar es que todavía haya aristas de la segunda capa que no estén colocadas pero tampoco se encuentren en la parte inferior de las caras. Eso significa que en verdad se encuentran en la segunda capa pero están descolocadas (Dos ejemplos: una arista verde-roja colocada en la posición correspondiente a la arista azul-roja. O una arista verde-roja colocada donde le corresponde pero con el color rojo en la cara verde y el color verde en la cara roja). La solución es que primero se suba una arista aleatoria a esa posición con una secuencia de movimientos muy similar al primer caso. Gracias a esto, la arista ya se encuentra en la parte inferior de las caras laterales y se puede resolver como tal. El diagrama indica que la asociación al primer caso es opcional, ya que cuando el algoritmo termina, se vuelven a comprobar las aristas descolocadas y se terminaría teniendo exactamente los mismos procesos.
Ejemplo: Una arista azul-naranja en la cara verde con el color naranja de frente. Se gira la cara amarilla en sentido antihorario, así el color naranja de la arista coincide con el naranja del centro de la cara. Aquí se comprueba que el lugar que corresponde a la arista se encontraría a la izquierda. Finalmente ejecuta el algoritmo correspondiente.

8.1.7. Paso 4: Cruz amarilla

A partir del cuarto paso, la única cara en la que hay que centrarse es la amarilla. (La cara amarilla y las laterales se resolverán a la vez en el último paso). Estos pasos son menos intuitivos y más estrictos. Cada conjunto de movimientos que se haga se tiene que asegurar de que la cara blanca y la segunda capa queden como estaban.

El objetivo de este paso es hacer una cruz amarilla. Igual que en el primer paso, en el de la cruz blanca, se deben ignorar los colores que tengan las esquinas pero la gran diferencia es que esta vez no hay que tener en cuenta que el color de las aristas coincida con el de los centros de las caras laterales (Poniendo como ejemplo la Figura 9, en el paso 4 la arista con el color rojo no tendría por qué coincidir con el centro rojo). Esto es debido a que se harán coincidir en paso 5.
El diagrama de flujo de este paso está en la Figura 13.

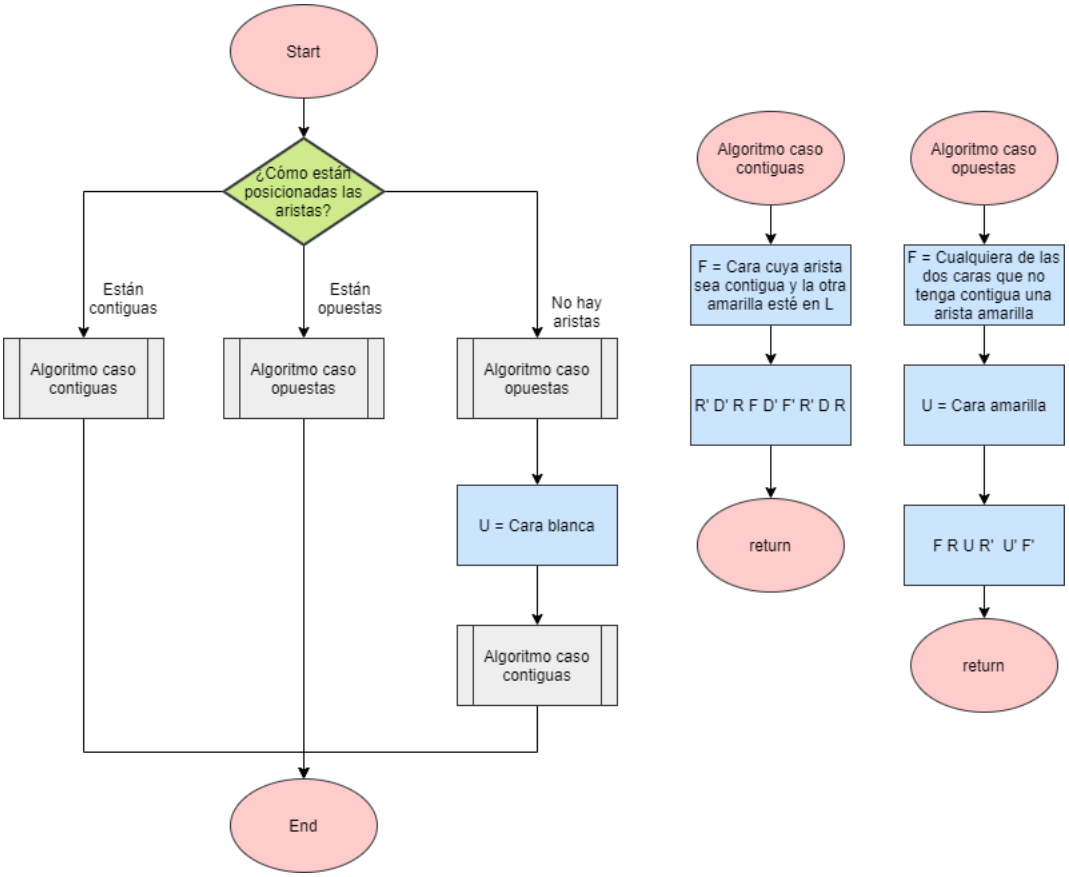


Figura 13: Diagrama de flujo de la cruz amarilla

Los algoritmos utilizados en este paso no se pueden encontrar en las páginas de referencia porque son los que encontré en la página web que ya no existe. He preferido usar estos porque me parecen más intuitivos y cuando el usuario haya internalizado el funcionamiento del cubo queda más claro el porqué de esos algoritmos.

Para hacer la cruz amarilla se observan qué aristas tienen el color amarillo en la cara amarilla. Solamente pueden estar posicionadas cero, dos o cuatro aristas. Si son cuatro, este paso se salta como se ve en la Figura 8.

- Si hay dos aristas, se comprueba la posición relativa entre ellas. El proceso de resolución cambia dependiendo de si las aristas están contiguas (p. ej. una arista en la cara verde y otra en la cara roja) o están en posiciones opuestas (p. ej. una arista en la cara verde y otra en la cara azul).

- Si no hay ninguna arista, se hace una combinación de los dos procesos predefinidos. Al ejecutar la secuencia del segundo caso, se colocan dos aristas amarillas contiguas y se resuelven igual que en el primer caso. (Como apunte, si el primer proceso predefinido que se hiciera fuera el del primer caso, el número de aristas en la cara amarilla seguiría siendo cero).

8.1.8. Paso 5: Posición de las aristas amarillas

Una vez la cruz amarilla está formada, ahora se necesita que las aristas estén posicionadas en su cara correspondiente (algo similar a la Figura 9). A partir de este paso, la cara amarilla será la cara superior. El diagrama de flujo de este paso están en la Figura 14.

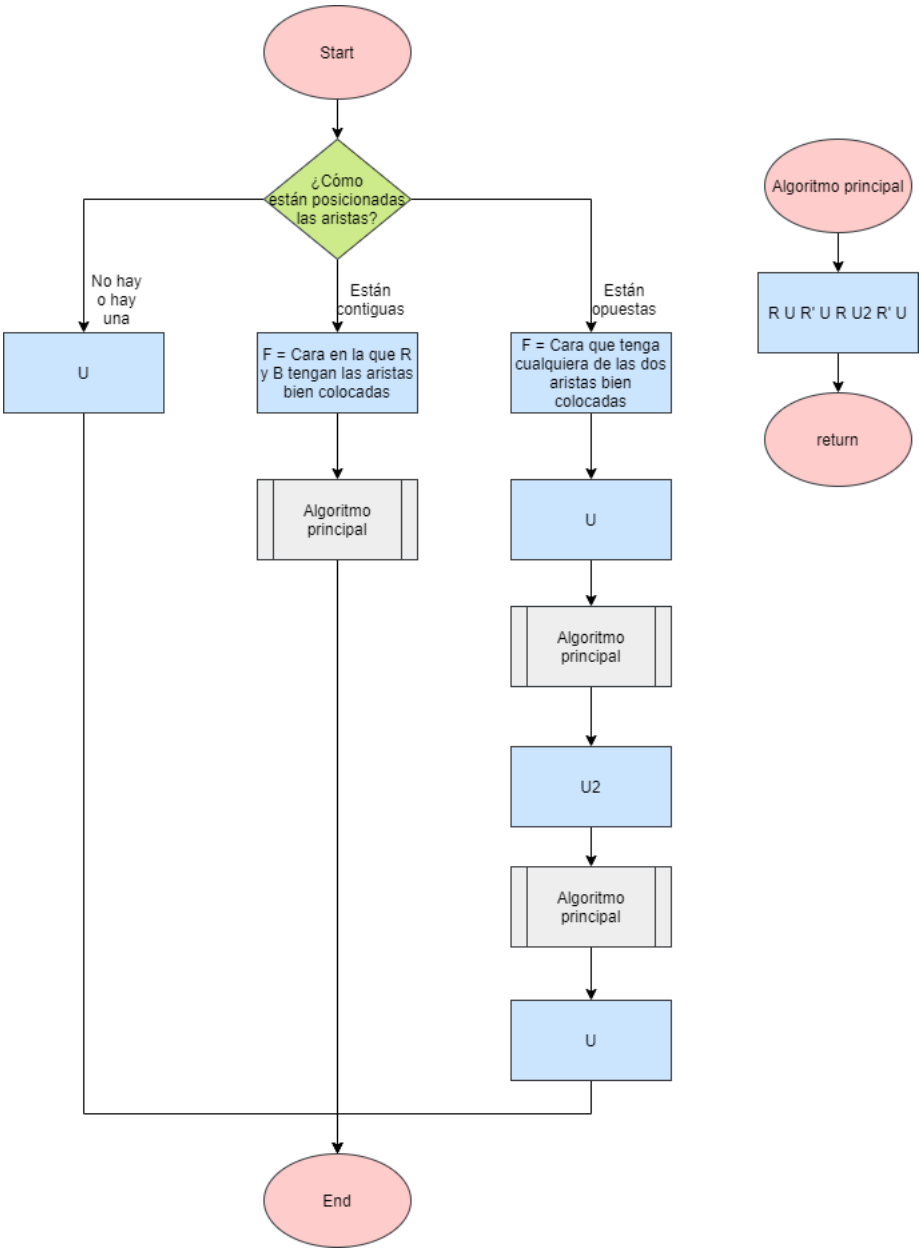


Figura 14: Diagrama de flujo de la posición de las aristas amarillas

Los algoritmos a ejecutar en sí son sencillos, pero elegir la cara frontal es más complicado. Se comprueban cuantas aristas bien posicionadas hay ahora mismo, es decir, que su color coincida con el

centro. Es difícil de entender con palabras la configuración de las aristas que se pide y la cara frontal que hay que escoger, así que hay dos esquemas a mayores que lo explican visualmente. Puede haber tres casos:

- Si ninguna o una arista coincide con su centro se gira la cara amarilla y se vuelve a comprobar.
- Dos aristas coinciden con su centro y son contiguas. La cara frontal que se escoge será la cual sus caras R y B sean las que contienen esas aristas. Un esquema de esa configuración se encuentra en la Figura 15. Luego queda ejecutar la secuencia de movimientos.
- Dos aristas coinciden con su centro y son opuestas. La cara frontal que se escoge es cualquiera de las dos en la que la arista esté bien colocada. Un esquema de esa configuración es encuentra en la Figura 15. Una vez elegida la cara frontal se gira la cara amarilla en sentido horario, se ejecuta el proceso predefinido, se mueve 180 grados la cara amarilla, se vuelve a ejecutar el proceso predefinido y finalmente se gira la cara amarilla en sentido horario.
  Si se mira la página web de referencia se ve que en vez de hacer giros adicionales de la cara amarilla, se mueve el cubo en sí, pero creo que queda más claro al usuario de esta manera.

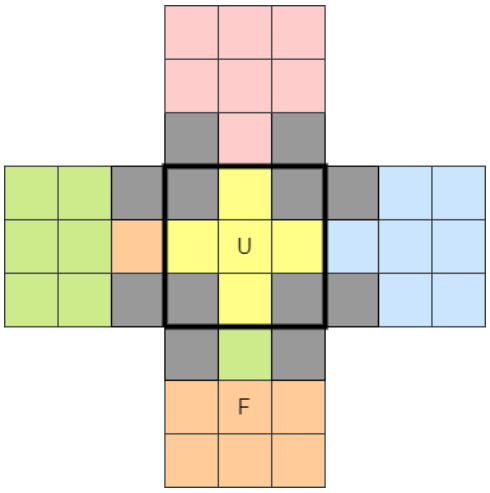


Figura 15: Esquema de aristas amarillas contiguas

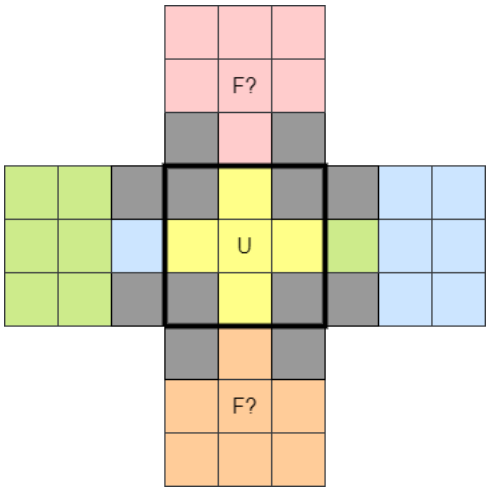


Figura 16: Esquema de aristas amarillas opuestas

8.1.9. Paso 6: Posición de las esquinas amarillas

Ya teniendo las aristas amarillas colocadas correctamente quedan solamente las esquinas amarillas. Se divide en dos pasos, uno para colocarlas en la posición que las corresponde (paso 6) y otro para orientarlas dentro de su posición (paso 7). El algoritmo se encuentra en la Figura 17.

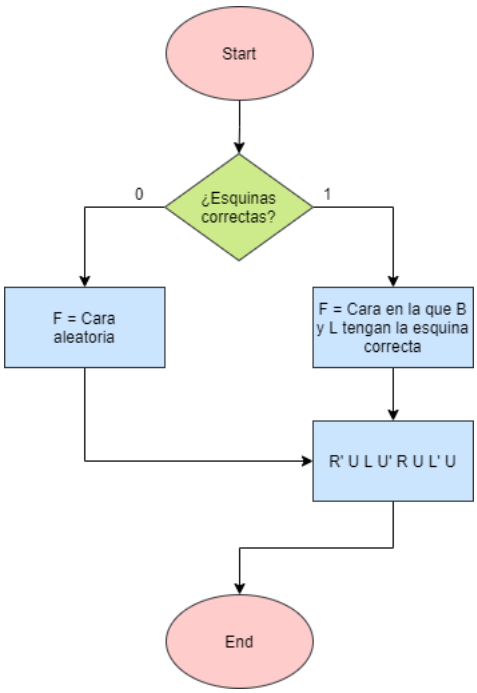


Figura 17: Diagrama de flujo de la posición de las esquinas amarillas

Lo importante a tener en cuenta en este paso es que las esquinas no tienen por qué estar bien orientadas. (Por ejemplo, aunque en esquina amarilla-azul-naranja no coincidan los colores y la parte azul coincida con la cara naranja, la parte amarilla coincida con la cara con la cara azul y la parte azul coincida con la cara amarilla, esta esquina se considera como bien posicionada).
Solo puede haber o 0 o una esquina en su posición (si hubiera 4 se saltaría este paso). La secuencia de movimientos es la misma, pero la decisión que hay que tomar es elegir la cara frontal.
Si no hay ninguna esquina en su sitio, se puede escoger la cara frontal que se quiera. Si hay una, se escoge la cara frontal como se indica en el diagrama y se ejecuta el conjunto de movimientos. La razón por la que se elige esta cara como la frontal es porque las tres esquinas que no están posicionadas cambian de posición entre ellas.
Ejemplo: La esquina amarilla-roja-verde es la única que está bien posicionada. Imaginando la cara amarilla como una matriz, esto equivaldría a las coordenadas [0, 0], así que se escoge como cara frontal la naranja. Ahora, ejecutando el algoritmo lo que se consigue es ir cambiando las posiciones de las tres esquinas restantes en sentido horario. La esquina [0, 2] pasa a ser la [2, 2], la [2, 2] pasa a ser la [2, 0] y la [2, 0] pasa a ser la [0, 2]. Si aún así siguen sin estar en su posición, se repite Casi de seguro este algoritmo se tendrá que repetir varias veces.

8.1.10. Paso 7: Orientación de las esquinas amarillas

El último paso es el más sencillo de todos. Una vez están bien posicionadas las esquinas, es posible que no estén bien orientadas 0, 2, 3 o 4 esquinas. En cualquier caso el algoritmo siempre será el mismo. Se encuentra detallado en la Figura 18.

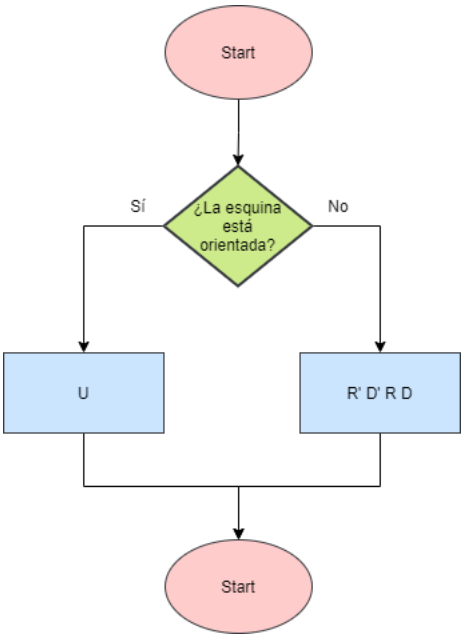


Figura 18: Diagrama de flujo de la orientación de las esquinas amarillas

Se escoge una cara frontal tal que la esquina que está mal posicionada se encuentre en la parte derecha. El número de veces que se repita el algoritmo dependerá de cómo está orientada la esquina. Una vez esté bien orientada, se comprueba si hay alguna otra esquina mal posicionada y se vuelve a girar la cara amarilla para que la esquina mal orientada vuelva a coincidir con la parte derecha de la cara frontal y se repite todo de nuevo. (Esto como mínimo habrá que hacerlo una vez. No puede haber solamente una esquina mal posicionada).

Al terminar este paso ya se tendría el cubo de Rubik resuelto.

8.2. Modelo conceptual

El diagrama del modelo conceptual se encuentra en la Figura 19

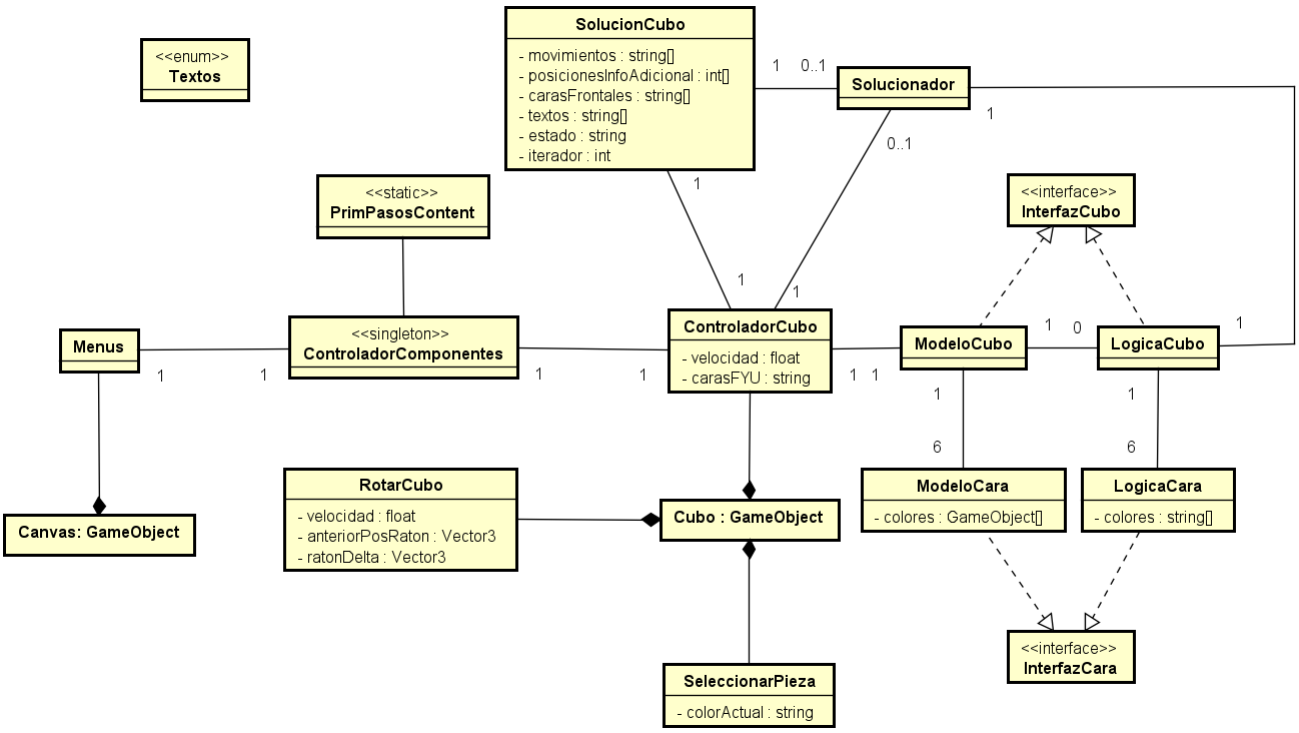


Figura 19: Diagrama de modelo conceptual

Como se ha visto en el apartado dedicado a hablar de Unity, las clases están determinadas por los scripts asociados a los objetos del motor.
Hay cuatro de esos scripts: Menus, ControladorCubo, RotarCubo y Seleccionar Pieza. Como se ve, están relacionados a los objetos mediante una asociación de composición, ya que no pueden existir sin dicho objeto y dejarán de existir si ese desaparece.

A continuación se entrará más en profundidad para hablar de las clases más importantes. Se entrará en más profundidad en algunas clases en apartados siguientes.

- **ControladorCubo**: Es un script asociado al modelo 3D del cubo. Esta clase funciona como controlador en un patrón MVC.
- **ModeloCubo y LogicaCubo**: Estas dos clases son muy similares pero contienen información diferente. En ambas clases están guardadas las relaciones que tienen las piezas entre sí. el funcionamientos de ModeloCubo está basado en las piezas como GameObject y en LogicaCubo, simplemente como datos abstractos. Las diferencias están más detalladas en el apartado de implementación.
- **SolucionCubo y Solucionador**: SolucionCubo es la clase que representa los pasos de resolución, además de los textos que se tendrían que se tendrían que mostrar por cada paso. Solucionador es el encargado de ejecutar el algoritmo de resolución y devolver una SolucionCubo.

8.3. Mapa de la interfaz de usuario

El mapa de la interfaz del usuario se encuentra en la Figura 20

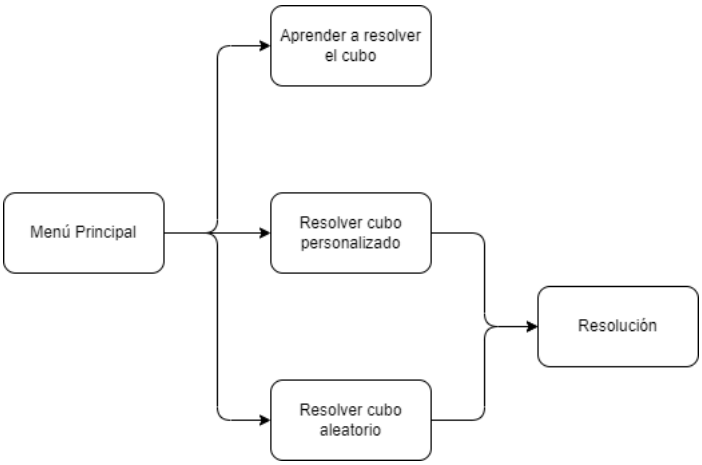


Figura 20: Mapa de la interfaz del usuario

Esta interfaz ha sido fuertemente influenciada por los casos de uso que se pueden dar.
El menú principal muestra las tres opciones: Aprender a resolver el cubo, Resolver cubo personalizado y Resolver cubo aleatorio.
Esas tres interfaces tienen un modelo 3D del cubo situado a la izquierda con botones para navegar por los movimientos además de un botón para volver al menú principal. En la primera opción, hay botones para seleccionar el paso que se quiere aprender y a la derecha se encuentran los textos que lo explican. En la segunda opción, a la derecha hay seis botones con distintos colores para poder colorear el cubo y un botón para resolver. En la tercera opción el cubo directamente se empieza a mover.
La segunda y tercera opción acaban en el menú de resolución una vez el cubo se haya resuelto. El menú es parecido al de aprender a resolver el cubo, solo que el texto va cambiando según los movimientos por los que navegue el usuario.

Para más información sobre las interfaces, se pueden visualizar en el manual de usuario.

8.4. MVC en Unity

El patrón de diseño MVC es muy útil en casos como este en el que el usuario interactúa con la aplicación mediante una interfaz. Sin embargo hay un problema en aplicar este patrón en Unity y es que no se tiene control directo de las clases que muestran información en pantalla.
Aún así, MVC es un patrón demasiado útil como para ignorarlo y para este proyecto se ha hecho una adaptación.

Los GameObjects como Cubo o Canvas funcionan en esencia como la **Vista**. Son los objetos que se encargan de representar información en la pantalla (en este caso el modelo 3D y la interfaz). Estos objetos tienen sus métodos y propiedades propias pero como ya se ha visto, no se puede crear un script que las manipule directamente, solo un controlador.

Los **Controladores** son ControladorCubo o Menus. Se encargan de responder a cualquier interacción que haga el usuario con la interfaz o cuando las clases correspondientes al Modelo quieran actualizar la Vista. Un caso
A mayores existe ControladorComponentes que se encarga de que los dos controladores principales se comuniquen con más facilidad, por ejemplo, si el usuario decide ir a un menú en el que el modelo 3D del cubo cambie, Menus manda un mensaje a ControladorComponentes, que manda un mensaje a ControladorCubo que actualiza el GameObject Cubo.

El resto de clases pertenecen al **Modelo**. Son las que manejan la información con la que funciona el programa. Por ejemplo ModeloCubo, que contiene la información de las piezas que se encuentran en una cara, aunque sea ControladorCubo la clase que se encargue de decir al cubo que mueva una cara. O Solucionador, que es la clase que contiene el algoritmo de resolución del cubo.

8.5. Diagramas de secuencia

Este apartado está dedicado a entrar en profundidad en algunas secuencias interesantes.

8.5.1. Awake y crear ControladorComponentes

En apartados anteriores se ha explicado cómo los scripts interactuan con el motor de Unity. Los dos siguientes diagramas (Figura 21 y Figura 22) demuestran como funciona la función Awake.

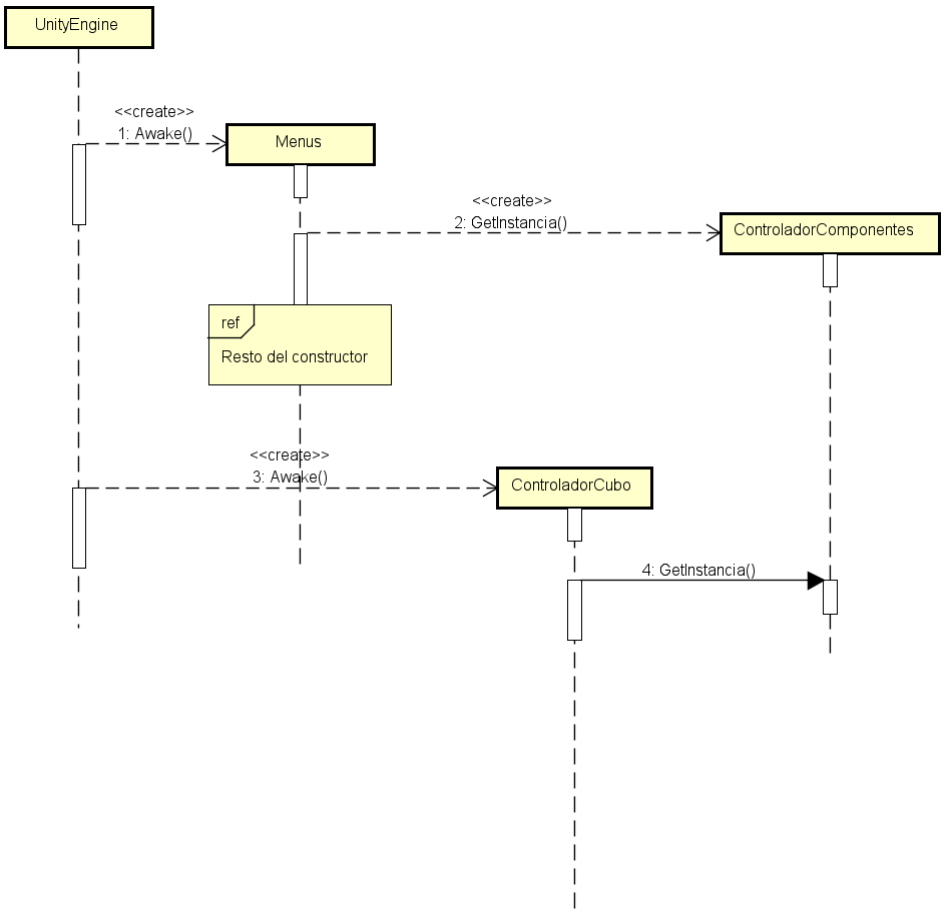


Figura 21: Diagrama de secuencia de la creación de ControladorComponentes

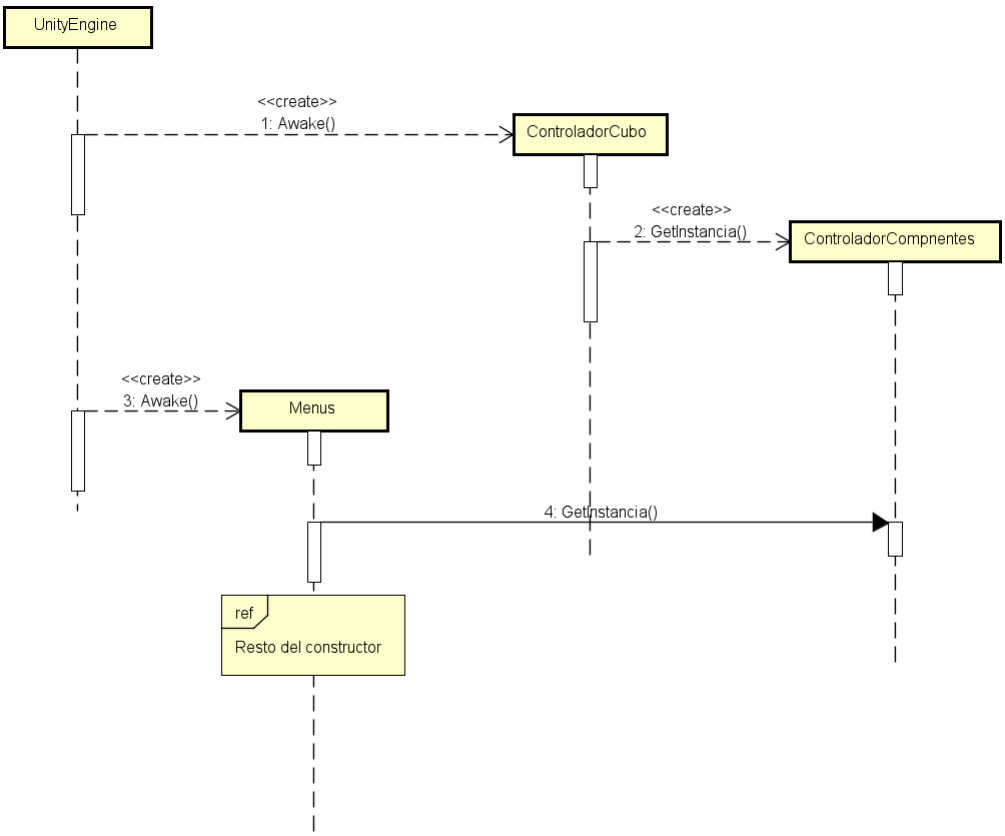


Figura 22: Diagrama de secuencia alternativo de la creación de ControladorComponentes

Awake es una función de evento que puede tener un script (heredado de MonoBehaviour) asociado a un GameObject. Esta función es llamada desde el motor nada más inicializarse el GameObject.
En este programa, Awake es utilizado para inicializar un Singleton. Hay dos diagramas ya que el motor hace dos llamadas a Awake y aunque son prácticamente instántaneas, no se hacen en paralelo. El motor tiene una lista interna de los scripts a los que hay que llamar, por lo que en este caso no se sabe si primero va a llamar a Menus o a ControladorCubo. Hay posibilidad de hacer un gestor propio para tener más control, pero en este caso no es necesario.

En resumen, el motor hace Awake al primer script que decida. Dicho script inicializa ControladorComponentes. El siguiente script que reciba la llamada Awake intenta también inicializar ControladorComponentes pero como es un Singleton simplemente le devuelve la intancia ya creada.

8.5.2. Update

El caso de Update es similar a Awake en cuanto a que es una función llamada desde el motor a un script asociado a un GameObject. La diferencia es que mientras que Awake solo es llamado una vez (al inicializarse), Update es llamado una vez por cada fotograma que vaya a mostrar en pantalla.
Esto significa que el rendimiento de la funcionalidad del programa depende también de hasta qué punto puede la máquina soportarlo gráficamente, ya que a menos fotogramas por segundo que pueda renderizar, menos Update y por lo tanto menos interactividad con el usuario. Existe FixedUpdate que es una función que se llama cada 20 milisegundos, pero se recomienda dejar esa función solo para las físicas del programa (como por ejemplo, si un botón estuviera asociado a rotar las caras del cubo). Además en este caso se ha escogido Update porque la funcionalidad a la que está asociada no es muy exigente.

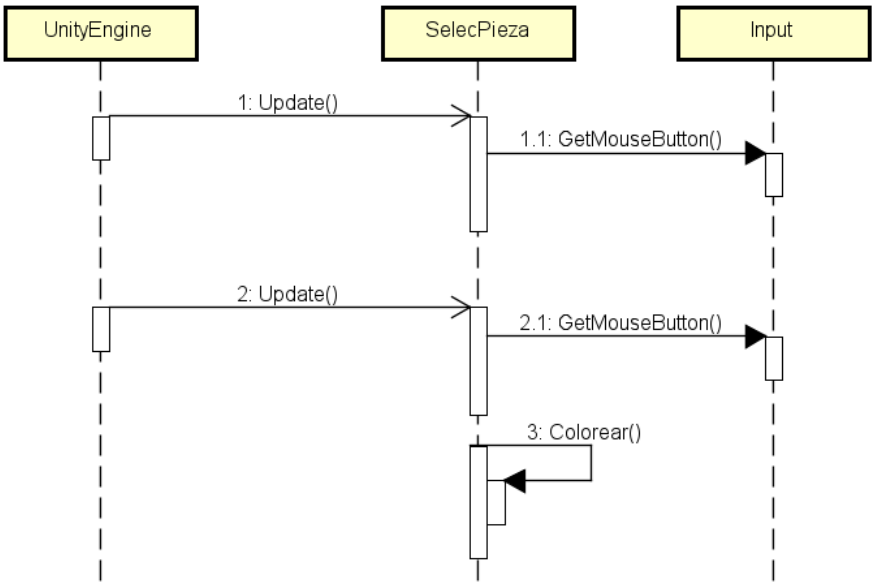


Figura 23: Diagrama de secuencia para colorear una pieza

En la Figura 23 se ve el funcionamiento de Update. El motor llama a SelecPieza y este comprueba si se ha pulsado el click izquierdo del ratón. Si se ha pulsado, se sigue la secuencia y si no, se termina hasta que el motor lo llame de nuevo en el siguiente fotograma.

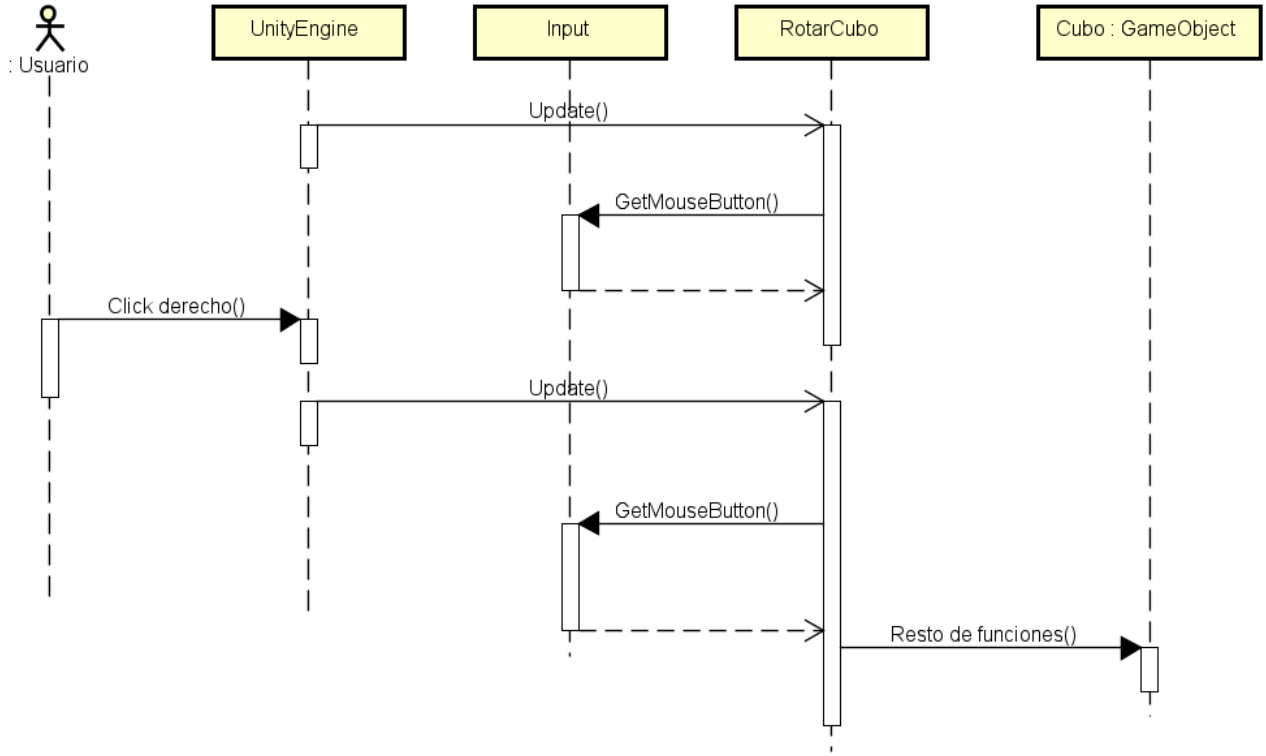


Figura 24: Diagrama de secuencia de la rotación del cubo

En la Figura 24 se muestra que la forma en la que el usuario puede hacer girar el cubo sobre sí mismo funciona de forma similar: Se hace un Update, se comprueba si se está pulsando el click derecho

y si se da el caso, la secuencia sigue. La forma en la que se puede hacer rotar el cubo sobre un eje mediante el ratón es interesante, pero no era relevante para explicar el Update.

8.5.3. Constructor de ControladorCubo

En el constructor de ControladorCubo (Figura 25) se encuentran las claves que darán forma al programa.

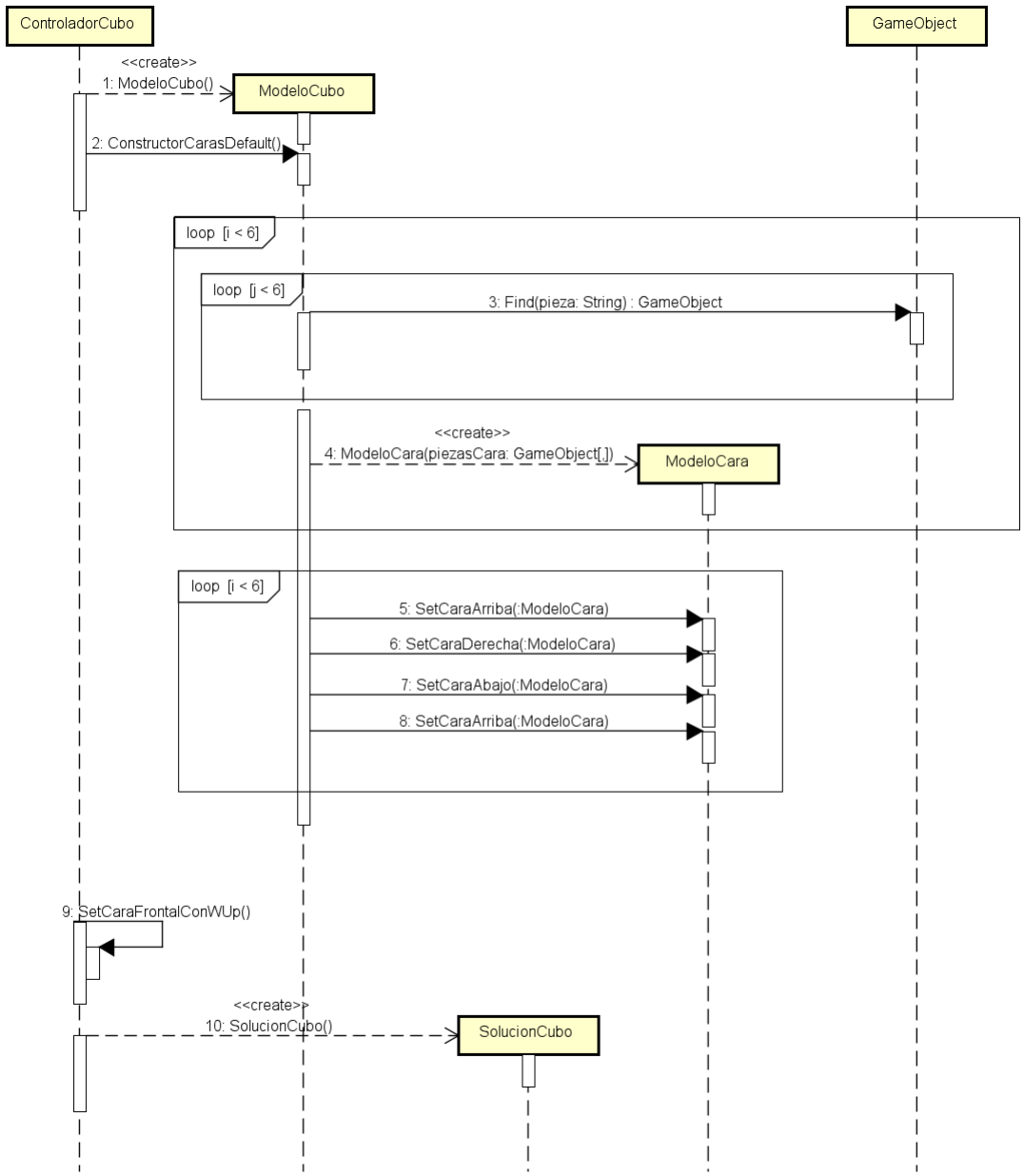


Figura 25: Diagrama de secuencia del constructor de ControladorCubo

En este constructor primero se crea un ModeloCubo.
Hay varias formas de crear un ModeloCubo, pero si se está llamando desde el constructor de ControladorCubo, se supone que el cubo está en su estado por defecto, es decir, resuelto.
En ModeloCubo se asociarán todas las piezas 3D prefabricadas a una variable para poder manejarla y también se crearán seis ModeloCara, las cuales se relacionarán entre sí para decidir las posiciones

relativas entre ellas. La implementación más en profundidad de ModeloCubo estará explicada más adelante.
Por último se crea una SolucionCubo donde se guardarán los movimientos.

8.5.4. Solucionar cubo

Quien va a manejar la solución del cubo es el ControladorCubo. Primero se crea una SolucionCubo donde se guardarán todos los movimientos que se tienen que ejecutar y todos los textos que se tienen que mostrar. Una vez creada, se crea un Solucionador al que se le pasará la Solución donde tiene que guardar todo y un ModeloCubo para que sepa el estado actual del cubo. Con estos datos se puede solucionar.

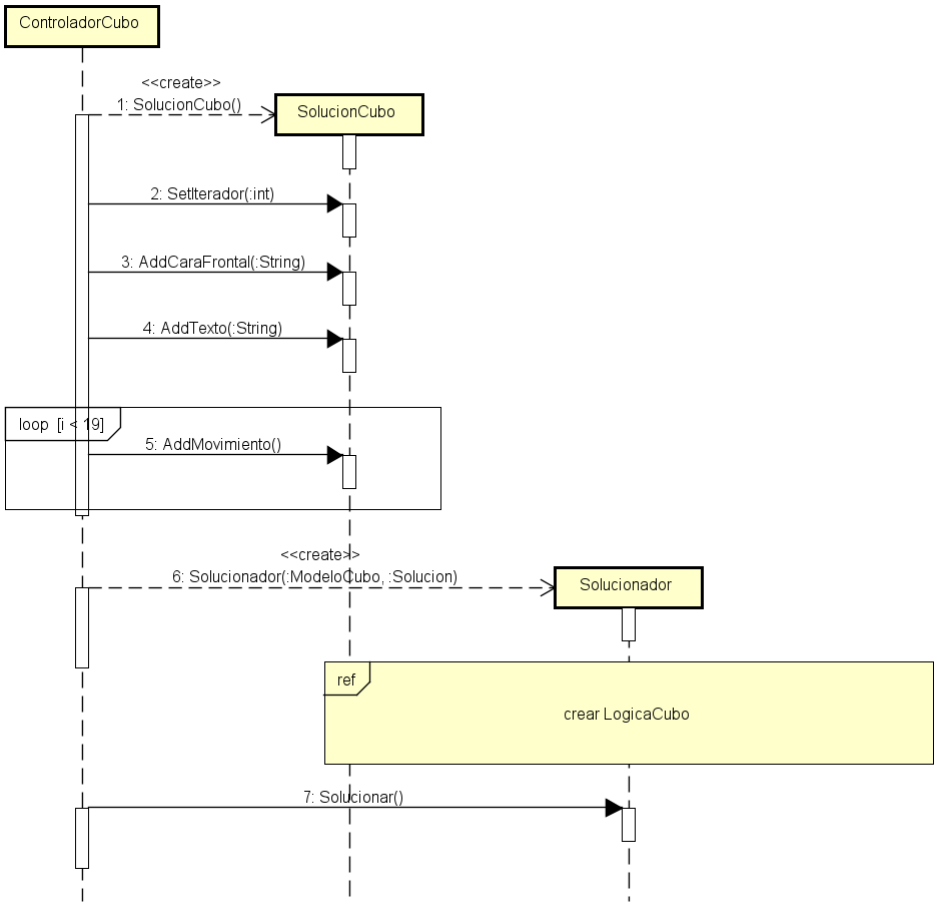


Figura 26: Diagrama de secuencia de crear una solución al cubo

En la Figura 26 se da el caso concreto en el que el estado del cubo se ha generado de forma aleatoria, por lo que se tienen que hacer unas configuraciones adicionales a SolucionCubo para que también queden representados esos movimientos aleatorios. Más adelante se explicará en profundidad su implementación.

8.5.5. Crear LogicaCubo

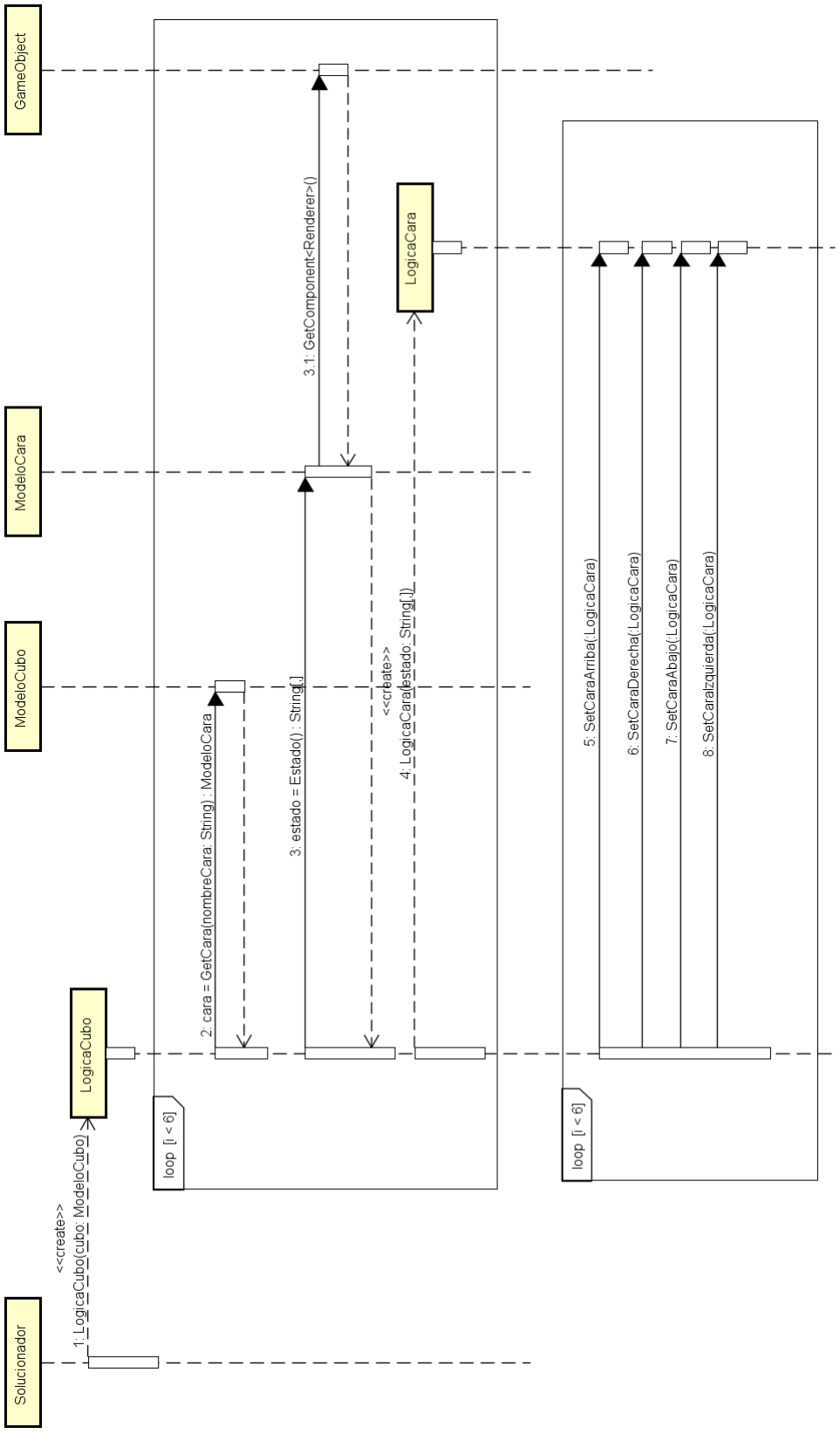


Figura 27: Diagrama de secuencia de la creación de LogicaCubo

LogicaCubo es utilizada para poder generar una solución al cubo (Figura 27).
Se crea desde Solucionador, ya que es la clase que va a utilizar esta información y se le pasa ModeloCubo para saber el estado actual. El constructor de LogicaCubo es similar al de ModeloCubo, pero en vez que recurrir a los prefabs para consultar el estado del cubo, se recurre a ModeloCubo.
Esta clase es muy importante para Solucionador ya que puede manejar la lógica del cubo como quiera gracias a la capa de abstracción que le ofrece al no tener que recurrir directamente a la información de los prefabs.

9. Implementación

Se dedica un apartado a la implementación para explicar más en profundidad las claves del programa aparte de las adaptaciones realizadas para ajustarse a Unity.

9.1. Modelo 3D

El modelo 3D es relativamente sencillo de modelar en este proyecto. Consiste principalmente en cubos. Aún así conviene explicar que se han utilizado herramientas que ofrece Unity para facilitar su desarrollo.
Anteriormente se ha explicado en qué consisten los prefabs: GameObjects agrupados para manejarlos como un conjunto. El prefab base sería una pieza del cubo de Rubik. Está compuesta por un cubo negro rodeado de seis prismas rectangulares actuando de pegatinas. Se instancian 27 prefabs y se oculta la visibilidad de las pegatinas según toque.

9.2. Rotación de caras

La rotación de las caras es interesante. Primero hay que saber qué piezas hay que mover, que eso lo deciden las clases ModeloCubo y ModeloCara. Pero otro obstáculo es ejecutar esta rotación a lo largo de un tiempo determinado y no de repente, es decir, que haya una animación.
Este objetivo se puede conseguir mediante dos formas: escribiendo el código dentro de Update o lanzando una Coroutine. En este proyecto se ha decantado por la segunda opción porque este es un método que no se va a ejecutar permanentemente. Las razones por las que Coroutine viene mejor en este caso están detalladas en el apartado dedicado a explicar Unity.
El método utilizado para realizar esta rotación gradual es calcular la orientación final de la cara e ir mostrando por pantalla orientaciones intermedias entre la orientación inicial y la final.

Un problema muy grave a destacar en estas rotaciones es algo bastante conocido en el entorno de desarrollo de videojuegos: el Gimbal Lock.

9.2.1. Gimbal Lock

Cuando estaba programando las rotación me encontré con este obstáculo que no sabia que existía y lancé una pregunta a StackOverflow: GameObject puede rotar alrededor de Y y Z pero no de X[1]. El usuario derHugo contestó que el problema residía en el Gimbal Lock. Un gimbal (o suspensión cardán) es un mecanismo consistente en tres aros concéntricos que permite que un objecto que se encuentre dentro gire sobre su eje. En la mayoría de software utilizado para animar modelos 3D utilizan este mecanismo para representar rotaciones. El Gimbal Lock sucede cuando dos ejes se alinean y el restante pierde algún grado de libertad de movimiento, como me sucedía a mí en la duda.
Hay dos formas de representar la orientación de un gimbal: mediante ángulos de Euler o mediante Quaternions (cuaterniones).

Los ángulos de Euler son representaciones de las tres coordenadas de los ejes. Sin embargo, este tipo de representación es la que da lugar al Gimbal Lock y por eso se utilizan los Quaternions.
Los Quaternions usan cuatro dimensiones (en Unity, las dimensiones X, Y, Z, W). Este tipo de representación es menos intuitiva que los ángulos de Euler, pero es la que hay que utilizar si se quiere evitar el Gimbal Lock. Por suerte, Unity posee una función que permite escribir los movimientos de los Quaternion como si fueran ángulos de Euler.

En conclusión, como se puede ver en la pregunta de la página web referenciada, para rotar una cara, primero se calcula el Quaternion en el que quedaría con el movimiento finalizado. En un principio se podría pensar que es cogiendo los ángulos de Euler y restarlos 90 grados, pero el método correcto es definir el Quaternion equivalente a ese movimiento (mediante el método Quaternion.Euler(float, float, float)) y multiplicarlo al Quaternion actual del objeto.

9.3. ModeloCubo y ModeloCara

ModeloCubo contiene seis ModeloCara y cómo están relacionadas entre sí.
El término modelo en ModeloCubo no se refiere al modelo 3D, sino al término que se suele usar en diseño de software. Contiene toda la información sobre el estado de las piezas 3D y tiene métodos para poder consultarlo o modificarlo (evidentemente, quien llama a estos métodos no es la propia clase sino otra que actúe como controlador).
La mayor ventaja de estas clases es tener registradas qué piezas se encuentran en una cara, ya que cada vez que una cara se mueva, se están cambiando la posición de las piezas de cinco caras a la vez.
ModeloCara contiene una matriz de 3x3 con referencias a los GameObjects correspondientes. Cuando se quiera girar una cara del cubo, ControladorCubo llama a ModeloCubo indicando la cara deseada y si es en sentido horario o antihorario. Entonces se hacen dos cosas: Se devuelve la referencia del GameObject padre que hay que rotar (el concepto de padre se explica al final del apartado) y a continuación, las referencias de las piezas cambiarán de posición tanto en su misma cara, como irán rotando en los ModeloCara contiguos que tenga.
Una nota muy importante sobre el movimiento del cubo. El encargado de mover el cubo 3D es ControladorCubo que actúa de controlador del GameObject cubo. Esta clase decide qué caras se mueven y en qué sentido pero tiene que usar de manera auxiliar ModeloCubo porque es quien tiene interrelacionadas las piezas y quien cambiará la información de qué piezas hay en cada cara una vez se ejecute el giro.
Una particularidad de Unity: a los modelos 3D se les puede asignar un padre. Esto significa que los hijos harán los mismos movimientos que haga el padre. Hay una función escrita para que ocho piezas de ModeloCara se conviertan en hijas de la pieza del centro y así sea más sencillo controlar los movimientos (de nuevo, esta función se llamará desde ControladorCubo).

9.4. LogicaCubo y LogicaCara

Estas dos clases se diferencian de las dos anteriores en que no dependen de los prefabs.
La utilidad de estas clases es similar. Guardan el estado del cubo pero en vez de guardar prefabs, guardan los colores asignados a las piezas.
De esta manera se pueden hacer estados imaginarios sin desmontar el estado actual del cubo 3D, además de que es menos costoso trabajar con colores que con prefabs enteros.

9.4.1. Algoritmo de resolución

El algoritmo de resolución implementado es la adaptación del diseño explicado en apartados anteriores.
Los datos que va a manejar se encuentran en LogicaCubo y LogicaCara.
LogicaCara tiene funciones que facilitan la implementación del algoritmo. Una de ellas es poder moverse por las caras mediante posiciones relativas, es decir, si se tiene escogida la cara azul se puede llamar a la cara izquierda directamente, en vez de escoger la cara roja.
Las posiciones relativas son inicialmente arbitrarias. Se tiene como referencia el esquema de la Figura 28

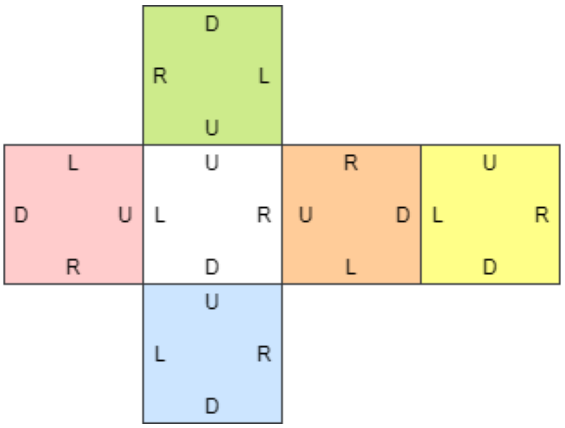


Figura 28: Esquema de las posiciones relativas de las caras

Un concepto inventado para la implementación es la distancia que hay entre aristas.
Hay algunas veces en las que una arista tiene que regresar a su posición original u otras en las que tiene que ir a una cara concreta pero no se sabe su posición relativa. El cálculo de la distancia sirve para saber cuantas veces hay que rotar una cara y en qué dirección. Otra ventaja que tiene es que es fácil de manipular si más adelante se quiere hacer el movimiento inverso.

También se tiene en cuenta que hay varias opciones igual de válidas para resolver un paso en concreto. En estos casos se decide de forma aleatoria.

9.5. Guardar los datos de la solución

Los datos de resolución que se vayan obteniendo en el Solucionador se van guardando en la SolucionCubo que se le ha pasado.
SolucionCubo es una clase que funciona como un iterador, es decir, tiene dos métodos públicos para moverse hacia adelante o hacia atrás de forma secuencial en una estructura de datos.
Internamente tiene cuatro listas. Tres para guardar los movimientos, los ángulos de la cámara y los textos. Se toma la primera como la más importante ya que es la que va a tener más entradas y la cuarta lista sirve para apuntar dónde empieza una secuencia nueva y la siguiente vez que se a llame Next o Previous mande nuevos textos o ángulos de cámara antes de seguir con los movimientos. En la Figura 29 hay un esquema que representa el funcionamiento interno.

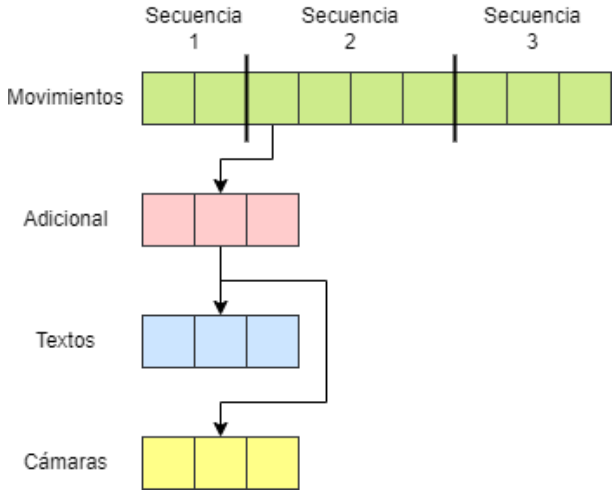


Figura 29: Esquema del funcionamiento interno de SolucionCubo

9.6. Configuraciones imposibles

En la parte de aprender a resolver el cubo de forma personalizada, el usuario puede colorear el cubo de forma incorrecta, así que el programa tiene que hacer una comprobación para saber si esa configuración puede existir. Se comprueba con estos tres requisitos:

- **Número de colores:** Cada color solo se puede repetir nueve veces, es decir, el número de pegatinas que se encuentran en una cara. Es el filtro más básico y solamente con este se pasarían por alto muchos casos.
  Ejemplo: La cara blanca está coloreada entera de color blanco y en la cara naranja hay otra pieza con el color blanco. Con esto suman diez instancias del mismo color.

- **Mismo color:** Una pieza no puede tener dos mismos colores repetidos. Eso significaría que hay dos caras con el mismo color. La mayoría de los casos que se tendrían que hacer en esta comprobación no pasa el primer filtro, sin embargo puede darse el caso en el que aún así haya solo nueve colores de cada.
  Ejemplo: Hay una arista que equivaldría a la blanca-naranja tiene dos colores naranjas. En un principio podría valer el primer filtro porque sobraría un color naranja y faltaría uno blanco. Sin embargo puede darse el caso de que ese color blanco también esté sustituyendo a una pieza que tendría que ser de color naranja, por ejemplo la esquina blanca-naranja-azul pasa a ser blanca-blanca-azul. Sigue habiendo nueve colores de cada, pero no pasa este filtro porque hay piezas en las que coinciden el mismo color.

- **Colores opuestos:** El último filtro es un caso que cumpliría con los requisitos de los dos anteriores pero seguiría habiendo una configuración errónea. Una pieza no puede tener dos colores de caras que sean opuestas.
  Ejemplo: una arista blanca-amarilla no puede existir ya que son dos caras que no tienen relación entre sí.

10. Conclusiones y trabajo futuro

Se han alcanzado todos los objetivos planteados al iniciar el proyecto.

10.1. Objetivos alcanzados

- Se ha expandido la notación de la resolución del cubo de Rubik y se ha racionalizado el algoritmo (especialmente en los primeros dos pasos) para poder trasladarlo a un programa informático.
- Se ha conseguido aprender Unity no solo para este proyecto, sino para cualquier otro futuro.
- Se ha podido adaptar este proyecto al funcionamiento de este motor.

10.2. Trabajo futuro

El proyecto podría portearse sin dificultad a otros sistemas operativos ya que Unity tiene opción de elegir el SO a la hora de construir el proyecto. Sin embargo, he decidido no adaptarlo por el momento porque requeriría testeo a mayores. Un punto débil de este proyecto es que está basado principalmente de cara al usuario pero no ha habido pruebas con usuarios.
Con estas pruebas se podría saber qué problemas hay con la interfaz, por si hay alguna parte que no se entiende bien o simplemente porque no es cómoda de utilizar. Pero especialmente estas pruebas son importantes debido a que el programa está dedicado a explicar un algoritmo complejo hasta cierto punto. Con feedback del usuario se podría saber qué puntos no están del todo bien explicados y cómo se podrían arreglar.

A parte de esto, no se me ocurre qué otras funciones añadir al programa, cosa que seguramente vendría sola haciendo tests con usuarios.

Appendices

Apéndice A Manual del usuario

A.1 Instalación

Para instalar el programa simplemente se descomprime el archivo y ya está listo para ejecutar abriendo el .exe

A.2 Menú Principal

El menú principal es lo primero que ve el usuario al ejecutar el programa.

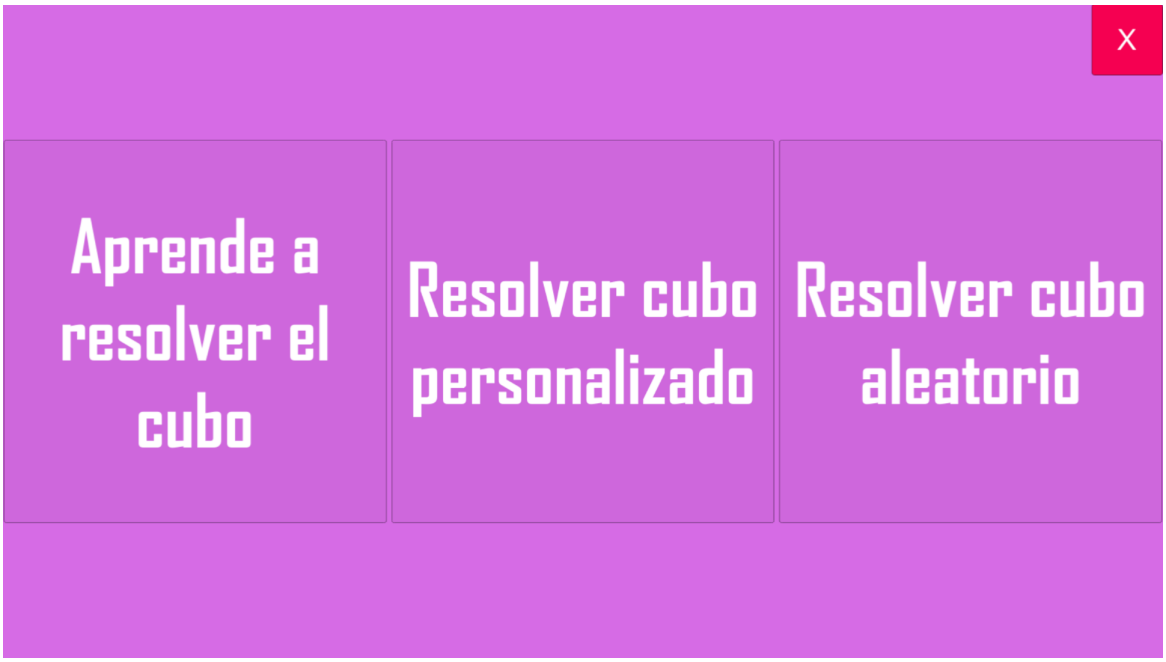


Figura 30: Menú principal

A.3 Aprender a resolver el cubo

Si se elige el primer menú, el de aprender a resolver el cubo se muestra un menú con el modelo 3D del cubo, un texto que explica el programa y varios botones los cuales si se pinchan en ellos, muestran los distintos pasos de resolución. Hay algunos pasos que tienen más pasos anidados, si se pincha en estos, se despliegan más botones para elegir.
Las flechas que señalan a la izquierda o derecha sirven para navegar por los movimientos de ese paso.

Figura 31: Menú de aprender a resolver el cubo

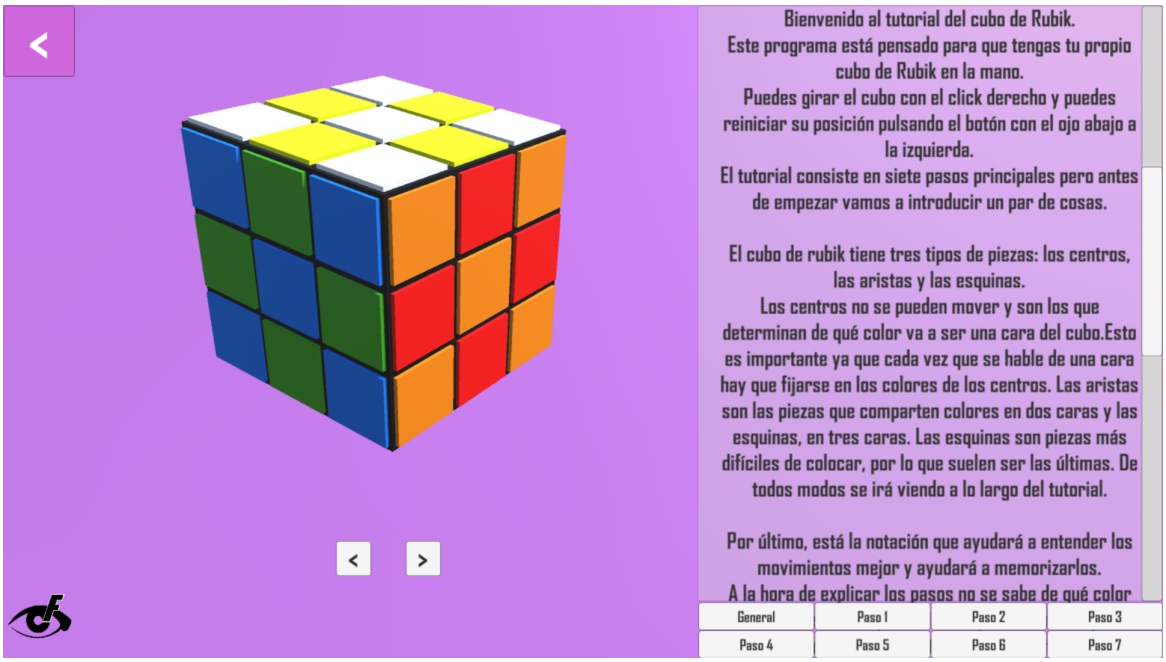


Figura 32: Ejemplo de secuencia de movimientos completada

Figura 33: Menú desplegado de un paso con diferentes casos

En las Figuras 34 y 35 hay piezas del cubo que están en negro. Estas piezas están así porque son irrelevantes para los pasos que se explican en ese momento y así facilitar el aprendizaje.

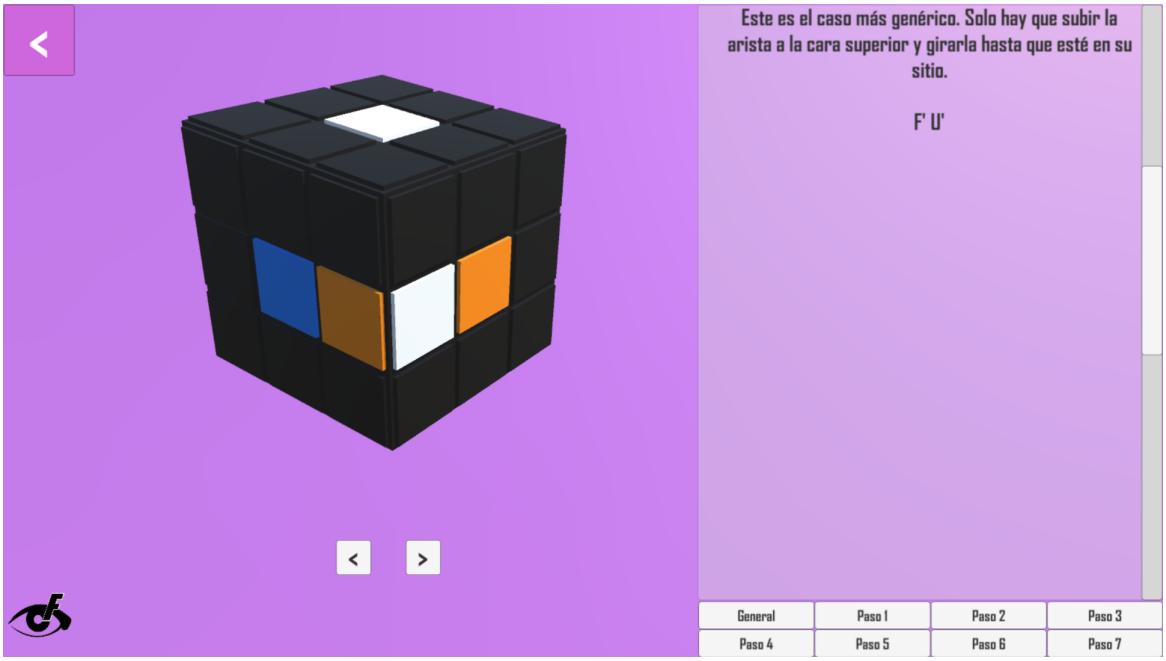


Figura 34: Paso 1 Caso 0

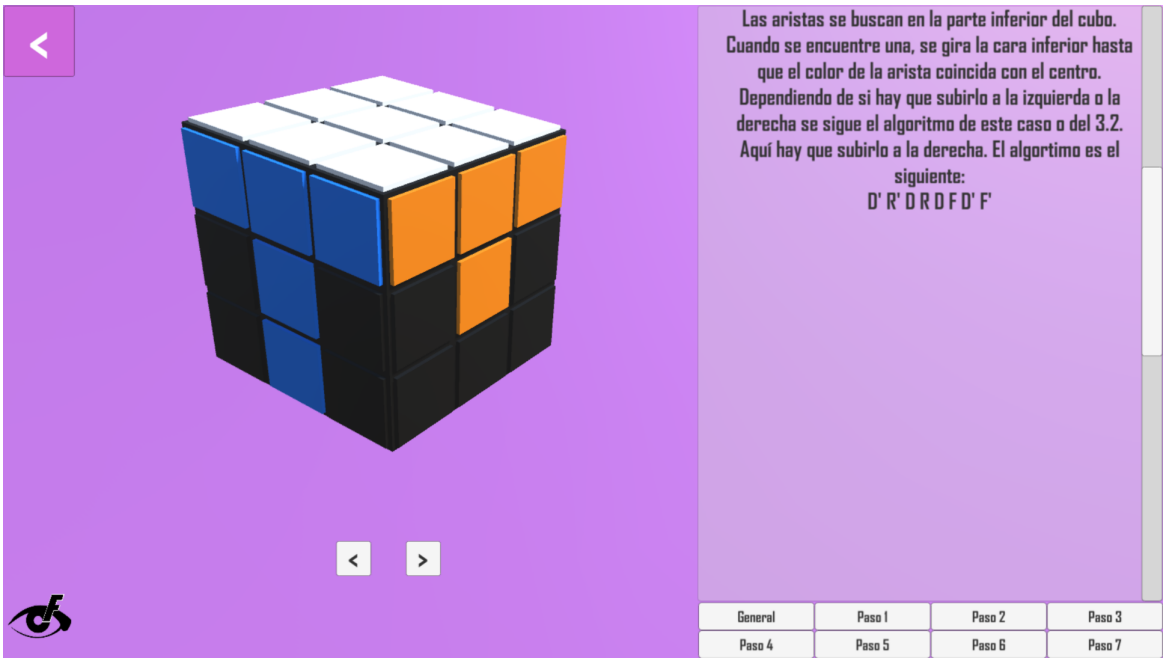


Figura 35: Paso 3 General

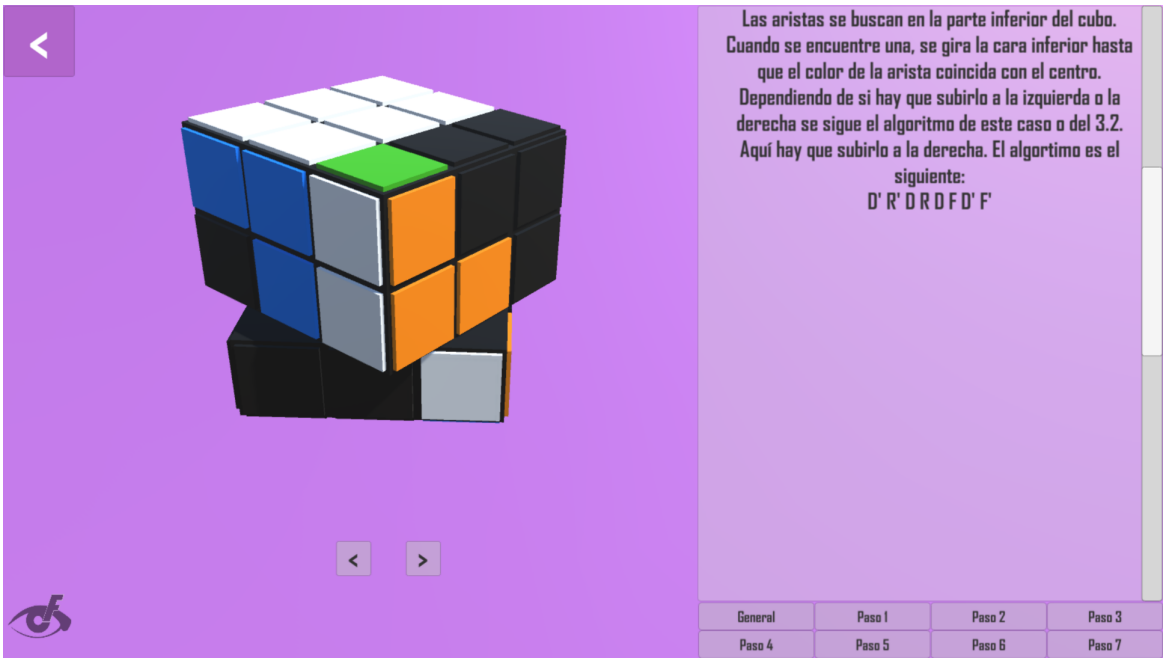


Figura 36: Cara rotando al pinchar en la flecha derecha o izquierda

A.4 Resolver cubo personalizado

Desde el menú principal, la segunda opción a elegir es resolver el cubo personalizado. Se muestra el modelo 3D y seis colores sobre los que el usuario puede pinchar. Si pincha un color, luego puede pinchar en una pieza del cubo y esta se coloreará. Una vez el usuario haya decidido que el cubo está coloreado como debe, puede pinchar en Resolver.

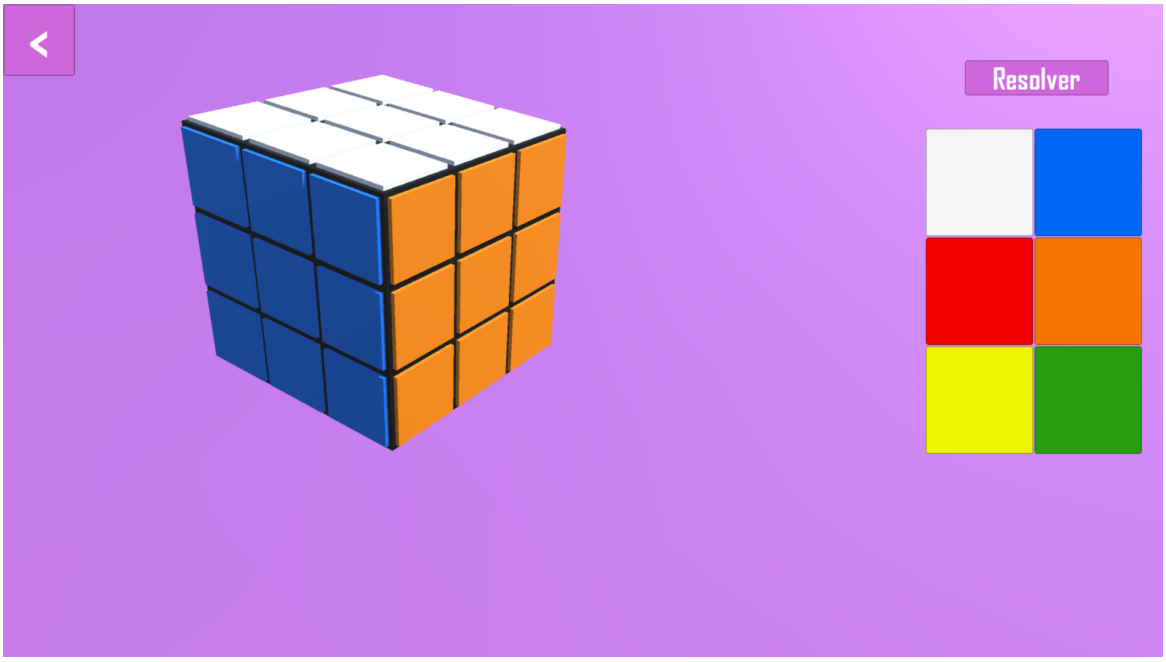


Figura 37: Menú de resolver cubo personalizado

Como curiosidad en relación a la Figura 38, el usuario no puede colorear los centros para evitar confusiones.

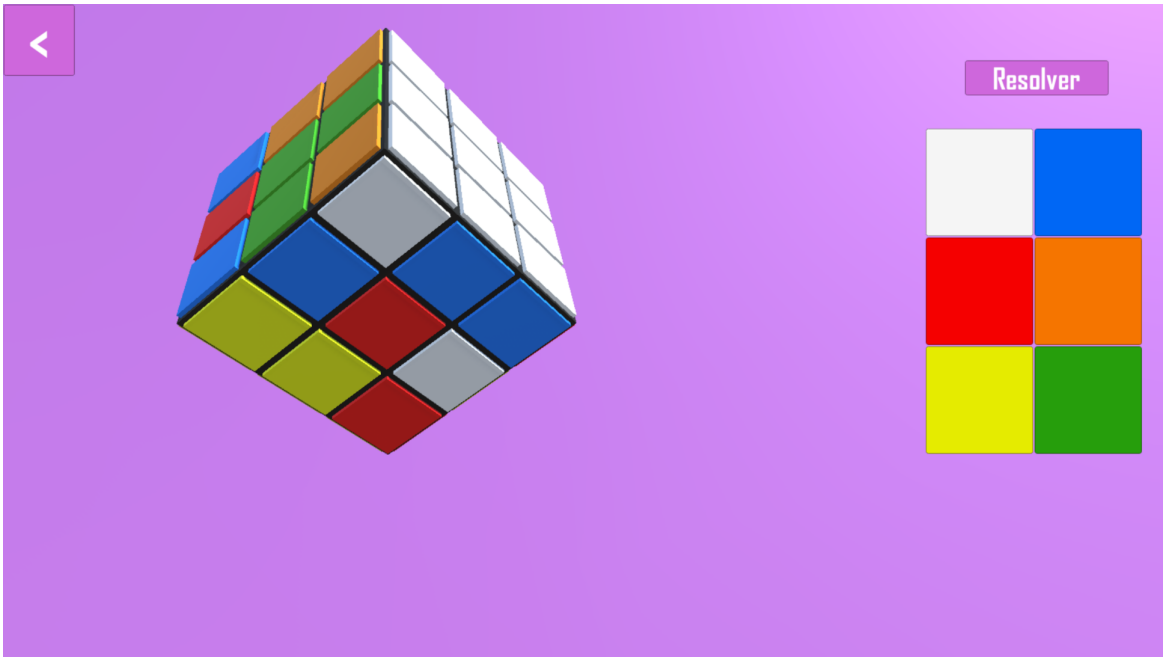


Figura 38: Cubo coloreado por el usuario

Figura 39: El usuario ha pinchado en Resolver con una combinación imposible

A.5 Resolver cubo aleatorio

La tercera y última opción que tiene el usuario en el menú principal es resolver un cubo aleatorio. Si el usuario elige esta opción, el cubo se pondrá automáticamente a desordenarse.

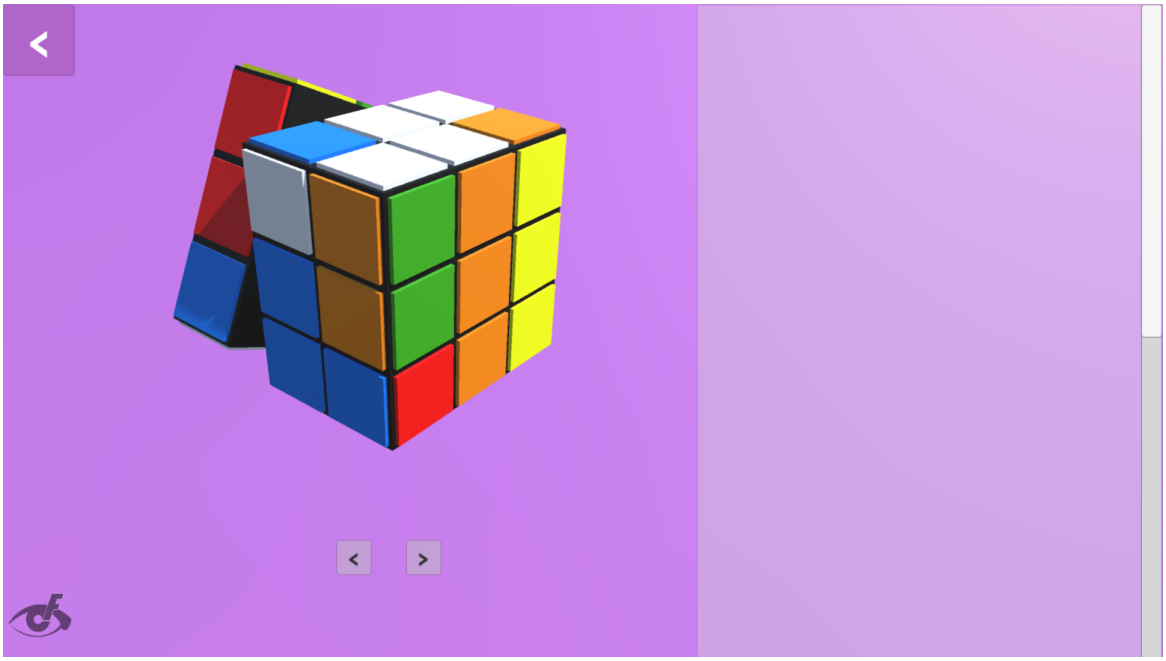


Figura 40: Cubo desarmándose con una secuencia aleatoria

A.6 Navegar por la resolución

Este menú aparece una vez el usuario haya configurado de manera correcta su cubo o el cubo haya realizado una secuencia aleatoria. El usuario puede navegar por los pasos con las flechas que apuntan

a la izquierda o derecha y se muestra un texto que explica dichos pasos.

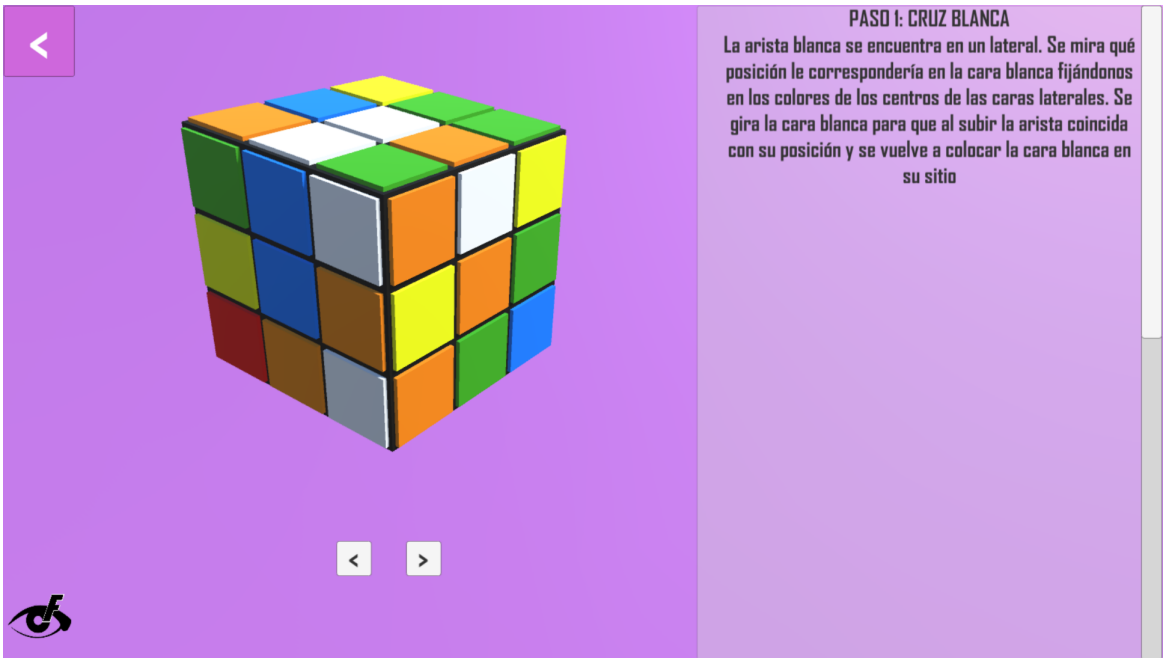


Figura 41: Ejemplo de un paso explicado durante la resolución

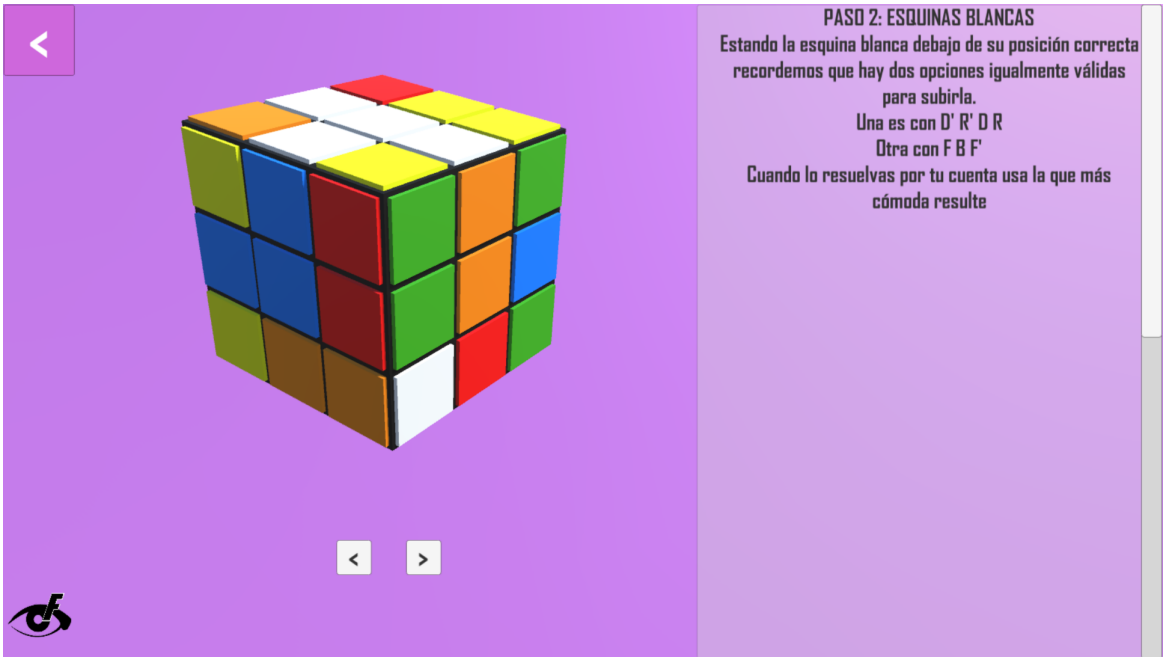


Figura 42: Otro ejemplo de un paso explicado durante la resolución

Apéndice B Entregables

En esta memoria se proporciona un enlace al Google Drive donde están guardados los entregables del proyecto:

https://drive.google.com/drive/folders/14RQZbnzxzkXpwkzGdorOX72llmH9UDLC?usp=sharing

En el Leeme.txt vienen definidas las carpetas.

En Entregable se encuentra la aplicación lista para ejecutar. Se descomprime y se ejecuta el archivo Cubo de rubik.exe. Puede dar problemas con el antivirus debido a las dll.

En el zip Proyecto se encuentra el proyecto Unity y el código fuente. El código fuente se encuentra en la dirección Assets -> Scripts. Si se quisiera visualizar de manera correcta el proyecto entero se necesita un editor de Unity.

No se han podido mandar todos estos archivos a la sede debido a que el proyecto ocupa más de lo permitido.

Referencias

[1] *GameObject can rotate around Y and Z but not X.* URL: https://stackoverflow.com/questions/66798655/gameobject-can-rotate-around-y-and-z-but-not-x.

[2] *How to solve a Rubik's Cube easily.* URL: https://cubesolve.com/.

[3] *How to solve the Rubik's Cube - Beginners Method.* URL: https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/.

[4] Wayne Johnson. *Interview with David Singmaster.* URL: https://www.twistypuzzles.com/articles/spotlight-singmaster/.

[5] Stefan Persson. *LEARN UNITY - The Most BASIC TUTORIAL I'll Ever Make.* URL: https://www.youtube.com/watch?v=pwZpJzpE2lQ.

[6] *Rapid application development model (RAD).* URL: https://www.geeksforgeeks.org/software-engineering-rapid-application-development-model-rad/?ref=lbp.

[7] David Singmaster. *Notes on Rubik's Magic Cube.* 1981. ISBN: 978-0894900433. URL: https://maths-people.anu.edu.au/~burkej/cube/singmaster.pdf.

[8] *Unity User Manual.* URL: https://docs.unity3d.com/Manual/.

[9] *Why do we not always use FixedUpdate?* URL: https://answers.unity.com/questions/1097715/why-dont-we-always-use-fixedupdate.html.