Rubikc Cube Game Project
REPORT Project (BCA-605)
Submitted in partial fulfillment of the requirements for the award of the degree of
BACHELOR OF COMPUTER APPLICATION
SUBMITTED BY

Name: Sagar Goswami
University Roll No.: 200845106059
Batch (2020-23) Session (Even, 2022-23)

CCS University, Meerut
Uttar Pradesh, INDIA

Under the Supervision of

Prof. Surendra Kumar
--
Assistant Dean , IT

Dr. /Ms. Teena Yadav
(Assistant / Associate Professor)

CERTIFICATE OF ORIGINALITY

This is to certify that the project synopsis entitled Rubikc submitted to Integrated Academy of Management and Technology (INMANTEC) in partial fulfillment of the requirement for the award of the degree of BACHELOR'S OF COMPUTER APPLICATIONS (BCA), is an authentic and original work carried out by Mr. / Ms. _____ with roll no. _____ under my guidance.

The matter embodied in this project is genuine work done by the student and has not been submitted whether to this University or to any other University / Institute for the fulfillment of the requirements of any course of study.

…… …….

Student Signature: Guide Signature

Date: …… Date……

ROLES AND RESPONSIBILITIES FORM

Name of the Project…… **Date:……**

S.NO	Name of Team Member	Roles	Task & Responsibility
1.	Prakash Sahu	Coding	Coding and logic part
2.	Sagar Goswami	Graphic	Graphic and Animation

Name and Signature of the Project Team members:

1…….. Signature……..
2…….. ……
3…….. ……
4…….. ……

Signature of the Project Guide:…… Date: …….

DECLARATION

We do hereby declare that the project work entitled “Rubikc” submitted by us for the partial fulfillment of the requirement for the award of Bachelor in Computer Applications(BCA), is an authentic work completed by us. The report being submitted has not been submitted earlier for the award of any degree or diploma to any Institute or University.

Date: .....

Signature

Name:

Rollno.:

ABSTRACT

Unveiling the Magic of the Simple Rubik's Cube Game: A Journey of Logic and Dexterity.

The Rubik's Cube, a mesmerizing 3D puzzle, has captivated minds and challenged the problem-solving skills of millions worldwide since its invention in 1974. This abstract delves into the enigmatic world of the Simple Rubik's Cube game, exploring its intriguing facets, gameplay mechanics, and the cognitive benefits it offers to players of all ages.

Our investigation begins by examining the Rubik's Cube's core components: a 3x3x3 grid structure consisting of six colored faces. We unravel the mechanics behind twisting and rotating the cube's layers, shedding light on the intricate movements that drive the puzzle's complexity. Understanding the fundamental operations is vital for players aiming to master the art of solving the cube efficiently.

Building upon this foundation, we embark on a journey through the game's various solving methods and techniques. We unravel the elegant simplicity of layer-by-layer algorithms, the intuitive approach of solving by patterns, and the intricacies of advanced methods like CFOP (Cross, F2L, OLL, PLL) and Roux. By exploring these techniques, players gain valuable

insights into the puzzle's inner workings, empowering them to tackle even the most challenging configurations.

Furthermore, we explore the cognitive benefits associated with the Rubik's Cube game. Scientific studies have revealed that regular engagement with the puzzle enhances critical thinking, spatial reasoning, and problem-solving abilities. The game's inherent complexity promotes strategic planning, patience, and perseverance, nurturing essential skills applicable not only to cube solving but also to various real-life situations.

Additionally, we discuss the diverse community that has formed around the Simple Rubik's Cube game. This vibrant community fosters camaraderie, knowledge-sharing, and innovation.

In conclusion, the Simple Rubik's Cube game transcends its humble origins to become a timeless symbol of intellectual challenge and artistic expression. Its multifaceted nature engages individuals on multiple levels, providing endless hours of entertainment, intellectual stimulation, and personal growth. Whether it be as a casual pastime or a serious pursuit, the Rubik's Cube continues to inspire and captivate generations, serving as a testament to the boundless potential of human ingenuity and determination.

ACKNOWLEDGMENT

I am very grateful to my major project (605P) for semester 6 guidance of Ms. Teena Yadav, for giving her valuable time and constructive guidance in preparing this Synopsis and Project (605P). It would not have been possible to complete this Project (605P) in the short period of time without her kind encouragement and valuable guidance.

DATE: SIGNATURE:

TABLE OF CONTENT

CHAPTER 5: IMPLEMENTATION

CHAPTER 6: CONCLUSIONS

TABLE OF FIGURES

Table of Figures

INTRODUCTION

1.1 Background

The Rubik's Cube, an iconic 3D puzzle, has been captivating individuals worldwide since its invention in 1974 by Ernő Rubik. Its complex design, colorful faces, and intricate movements have intrigued millions of puzzle enthusiasts, making it one of the most popular and enduring puzzles of all time. This project aims to delve into the world of the Rubik's Cube game, exploring its history, mechanics, and the motivation behind its creation.

Historical Context:

The Rubik's Cube emerged during a period marked by a surge in puzzle popularity. Ernő Rubik, a Hungarian architect and professor, designed the cube as a teaching tool to help his students understand spatial relationships. However, he soon realized that his creation possessed a captivating charm beyond its educational utility. The Rubik's Cube quickly gained popularity, spreading from Hungary to the rest of the world, captivating the imaginations of people of all ages.

Mechanics and Gameplay:

At the core of the Rubik's Cube lies a 3x3x3 grid structure composed of six faces, each adorned with a different color. The primary objective of the game is to align all the colors on each face by twisting and rotating the cube's layers. The cube offers an astonishing number of possible configurations, with an astronomical 43 quintillion combinations, making it an incredibly challenging puzzle to solve.

Conclusion:

The Rubik's Cube game stands as a testament to human ingenuity and the universal appeal of intellectual challenges. Its intriguing mechanics, diverse solving strategies, and cognitive benefits have made it a timeless favorite among puzzle enthusiasts. This project aims to explore the rich background of the Rubik's Cube, shedding light on its historical context, gameplay mechanics, educational significance, and the vibrant community it has fostered. By delving into this puzzle's fascinating world, we hope to inspire further exploration, innovation, and appreciation for the Rubik's Cube game and its enduring legacy.

1.2 Objectives

The objective of the Rubik's Cube game project is to explore and analyze various aspects of the iconic puzzle, including its history, mechanics, solving strategies, educational benefits, and community impact. By delving into the world of the Rubik's Cube, this project aims to provide a comprehensive understanding of the puzzle's enduring appeal and its cognitive challenges.

Through research and analysis, we seek to uncover the puzzle's impact on critical thinking, spatial reasoning, and problem-solving skills. Additionally, we aim to highlight the community that has formed around the Rubik's Cube, showcasing its inclusive nature, knowledge-sharing platforms, and the excitement of competitive events. Ultimately, this project strives to inspire and engage individuals by unraveling the magic of the Rubik's Cube and its timeless allure.

1.3 Purpose, Scope, and Applicability

Purpose:

The purpose of the Rubik's Cube simple game is multifaceted. Firstly, it aims to provide an enjoyable and captivating puzzle experience for players of all ages. The game challenges individuals to manipulate the cube's layers, fostering a sense of excitement, accomplishment, and satisfaction upon solving it. By offering an engaging and entertaining gameplay experience, the Rubik's Cube game seeks to captivate players' attention and provide a source of leisure and mental stimulation.

Secondly, the game aims to promote cognitive development and enhance essential skills such as critical thinking, problem-solving abilities, and spatial reasoning. As players attempt to solve the cube's complex configuration, they are required to analyze patterns, strategize their moves, and employ logical deduction. Through repeated engagement with the game, players can improve their cognitive abilities, including pattern recognition, logical reasoning, and mental agility. These skills extend beyond the game itself and can be beneficial in various real-life situations, such as problem-solving tasks, academic pursuits, and professional challenges.

Scope:

The scope of the Rubik's Cube simple game project encompasses several key aspects. Firstly, it involves the creation and implementation of a user-friendly and accessible virtual or physical version of the Rubik's Cube. The game's design and interface should be intuitive, allowing players to easily manipulate the cube's layers and navigate the gameplay mechanics. The simplicity of the game ensures that both beginners and casual players can engage with it comfortably, while still offering ample challenges to sustain interest and promote skill development.

Furthermore, the project includes the development of algorithms and mechanics that accurately simulate the Rubik's Cube's movements and transformations. This ensures that the game faithfully replicates the real-life puzzle, providing an authentic and immersive experience. Additionally, the project may incorporate features such as hints, tutorials, or customizable difficulty levels to accommodate players of varying skill levels and learning preferences.

Applicability:

The Rubik's Cube simple game is designed with broad applicability in mind. It targets individuals who enjoy puzzles, logic games, or brain teasers, catering to puzzle enthusiasts seeking a new challenge. Additionally, the

game is accessible to casual gamers looking for an engaging and mentally stimulating pastime.

The Rubik's Cube simple game is suitable for individuals of different ages, making it an inclusive experience for children, teenagers, adults, and seniors alike. Its intuitive design and gameplay mechanics ensure that players can readily grasp the basic concepts and progress at their own pace.

Furthermore, the game's applicability extends beyond mere entertainment. It can serve as an educational tool, particularly in the context of mathematics, computer science, and spatial cognition. Educators and parents can utilize the Rubik's Cube game to enhance students' problem-solving abilities, logical thinking, and analytical skills. By engaging with the game, learners can develop a deeper understanding of spatial relationships, algorithms, and critical thinking strategies.

In summary, the Rubik's Cube simple game project aims to entertain, stimulate cognitive development, and have educational applicability. It offers an enjoyable and accessible puzzle experience while promoting essential skills that can be valuable in various aspects of life.

1.4 Achievements

The Rubik's Cube Simple Game project achieved the successful development and delivery of a captivating and user-friendly gaming experience centered around the iconic Rubik's Cube puzzle. Key achievements of the project include:

1. Intuitive and Accessible Gameplay: The game features an intuitive interface that allows players of all ages and skill levels to engage effortlessly with the Rubik's Cube. The simplified controls and clear visual cues enable smooth manipulation of the cube's layers, providing an enjoyable gaming experience.
2. Realistic Cube Mechanics: Through meticulous algorithm development, the game accurately simulates the movements and transformations of the Rubik's Cube. Players experience the authentic behavior of the cube, enhancing their immersion in the puzzle-solving process.
3. Immersive Feedback: Visual and audio effects provide immersive feedback, enhancing the gaming experience. Players receive real-time feedback on successful moves, milestones, and achievements, further enhancing their sense of accomplishment and engagement.
4. Cross-Platform Compatibility: The game is developed with cross-platform compatibility, enabling players to enjoy the Rubik's Cube

experience on various devices, including desktop computers, mobile devices, and even physical Rubik's Cube models.

Through these achievements, the Rubik's Cube Simple Game project successfully delivers an entertaining, educational, and intellectually stimulating gaming experience. It captures the essence of the Rubik's Cube puzzle while catering to a diverse audience, fostering skill development, and providing a memorable and enjoyable journey for players of all ages.

SURVEY OF TECHNOLOGIES

2.1 C#:

C# is a versatile programming language widely used for developing a variety of software applications, including desktop, web, and mobile applications. It offers a robust and object-oriented programming model, making it suitable for building scalable and maintainable software solutions. With its extensive framework libraries and tooling support, C# enables developers to create efficient and high-performance applications. Whether you're a beginner or an experienced programmer, C# provides a powerful language ecosystem for developing innovative and reliable software solutions.

2.2 Unity Engine:

Unity Engine is a popular cross-platform game development framework that empowers developers to create immersive and interactive 2D and 3D games. It offers a user-friendly interface, a comprehensive set of tools, and a powerful game engine, making it accessible to both beginners and experienced developers. With Unity, game creators can leverage its extensive library of assets, visual scripting system, and robust physics engine to bring their ideas to life. Whether you're a solo developer or part of a team, Unity Engine provides a flexible and efficient platform for designing and deploying captivating games.

2.3 Mono:

Mono is an open-source implementation of the .NET framework, enabling developers to build cross-platform applications and libraries. It provides a runtime environment that allows .NET applications to run on various operating systems, including Windows, macOS, Linux, and more. Mono offers compatibility with the latest C# language features and supports a wide range of application types, from desktop applications to web services and mobile apps. With its versatility and portability, Mono simplifies the development process by enabling code reuse and facilitating multi-platform deployment.

2.4 .NET:

.NET is a powerful framework developed by Microsoft for building a wide range of applications, including desktop, web, and mobile applications. It provides a comprehensive development platform with a robust class library, tools, and runtime environment. With its versatile languages like C# and VB.NET, developers can create efficient and scalable applications for various platforms. .NET supports cross-platform development, allowing applications to run on Windows, macOS, and Linux. With its extensive ecosystem and continuous updates, .NET empowers developers to build modern, secure, and high-performance software solutions.

2.5 Blender:

Blender is a versatile open-source 3D modeling and animation software widely used in the creation of visual effects, animated films, games, and more. It offers a comprehensive suite of tools for modeling, sculpting, texturing, rigging, animation, and rendering. Blender's powerful capabilities and intuitive interface make it suitable for artists, designers, and developers. It supports various file formats, allows for realistic simulations, and offers a vibrant community that shares tutorials, plugins, and assets. With Blender, users can unleash their creativity and bring their imaginative visions to life.

2.6 Git and GitHub:

Git is a distributed version control system that enables developers to track changes, collaborate, and manage source code efficiently. It provides a reliable and decentralized platform for storing and managing code repositories. GitHub, on the other hand, is a web-based hosting service that allows developers to host their Git repositories and collaborate with others through features like pull requests, issue tracking, and code reviews. Git and GitHub have become essential tools for software development, facilitating team collaboration, version control, and code sharing. With their robust features and widespread adoption, they have revolutionized the way developers work and collaborate on projects.

REQUIREMENTS AND ANALYSIS

3.1 Problem Definition

The Simple Rubik's Cube game project aims to address the challenge of providing a user-friendly and accessible virtual or physical version of the Rubik's Cube puzzle. The problem lies in creating a game that allows players of all ages and skill levels to engage with the Rubik's Cube, offering an enjoyable and stimulating experience.

Key challenges to overcome include designing an intuitive interface that simplifies cube manipulation, ensuring that players can easily twist and rotate the cube's layers without confusion. Additionally, developing algorithms that accurately replicate the cube's movements and transformations poses a technical challenge.

Another aspect of the problem is catering to the diverse audience and skill levels. The game needs to provide ample challenges for experienced players while remaining accessible to beginners. Striking the right balance between difficulty levels, providing hints or tutorials, and offering customizable options becomes crucial in creating an inclusive gaming experience.

Furthermore, the problem involves incorporating features that enhance gameplay, such as tracking solving times, offering scoring systems, or providing visual and audio feedback to engage and motivate players. Ensuring the game captures the essence of the Rubik's Cube experience and provides an authentic and satisfying puzzle-solving journey is essential.

Ultimately, the problem is to create a Simple Rubik's Cube game that successfully combines simplicity, accessibility, and challenge while maintaining the essence of the original puzzle and providing an enjoyable and intellectually stimulating experience for players of all ages and skill levels.

3.2 Requirements Specification:

To meet the problem definition and successfully develop the Simple Rubik's Cube game, the following requirements should be considered:

Intuitive Interface: Design a user-friendly interface that allows players to manipulate the Rubik's Cube easily. Ensure clear visual cues and intuitive controls for rotating and twisting the cube's layers.

Realistic Cube Movements: Develop algorithms and mechanics that accurately simulate the Rubik's Cube's movements and transformations. The virtual or physical cube should behave realistically, providing an authentic experience.

Multiple Difficulty Levels: Implement different difficulty levels to cater to players of varying skill levels. Beginners should have accessible options, while experienced players should be challenged with advanced levels.

Hint and Tutorial System: Include a hint system to assist players during gameplay, providing suggestions or step-by-step guidance. Additionally, develop tutorial resources to teach beginners the basics of solving the Rubik's Cube.

Visual and Audio Feedback: Incorporate visual and audio effects to provide feedback on successful moves, progress, and achievements. Engaging and immersive feedback enhances the gaming experience.

Platform Compatibility: Ensure compatibility with various platforms, such as desktop computers, mobile devices, or physical Rubik's Cube models, to reach a wider audience.

3.3 Planning and Scheduling:

Figure 1: Development Pipeline

The development of the Simple Rubik's Cube game should be carried out in a systematic and organized manner. The following steps outline the planning and scheduling process:

Define Milestones: Identify key milestones and deliverables, such as interface design, cube mechanics, difficulty levels, customization options, hint system, scoring, and tracking features.

Task Breakdown: Break down each milestone into smaller tasks, considering factors like coding, design, testing, and documentation.

Assign Responsibilities: Allocate tasks to the development team members based on their expertise and availability. Ensure clear communication channels for collaboration and progress updates.

Estimation and Timeframes: Estimate the time required for each task and milestone, considering factors like complexity, dependencies, and resource availability. Establish realistic timelines for completing each task.

Regular Monitoring and Updates: Monitor the project's progress regularly, update the schedule, and address any delays or issues promptly. Maintain open communication with the team to ensure smooth workflow.

Testing and Quality Assurance: Allocate sufficient time for testing and quality assurance to identify and resolve any bugs or issues. Perform

thorough testing at different stages of development to ensure a robust and error-free game.

Documentation: Document the development process, including design decisions, implementation details, and user instructions. This documentation helps in future updates, maintenance, and user support.

By following a well-defined plan and adhering to the scheduled timeline, the Simple Rubik's Cube game project can be executed efficiently, ensuring a successful and timely completion.

3.4 Software, Hardware and Technical Requirements

1. Software Requirements.

- Any Modern OS (Linux, Windows or MacOS)
- Visual Studio Code/Codium or Visual Studio
- Git and GitHub
- Unity Engine
- Blender
- Photoshop

2. Consumer Requirements.

- Any modern Web browser (Microsoft Edge, Google Chrome etc.)
- Active Internet Connection

3. Hardware Requirements

- 1 GHz processor or higher
- 1 GB RAM or higher
- 1.5GB Available Hard Drive Space(for Browser)
- Windows 7 SP2 or later operating system OR Any LINUX distribution.
- It is also Accessible on Mac OS
- An active Internet Connection

4. Smartphone

- Android 5.0+(for Web-view and browser Support)

5. Technical Requirements

- C# (Primary Programming Language)
- .NET core 7 (Microsoft's .NET framework class library)
- Mono (Open-Soured, .NET implementation)
- Unity3D (AAA, GameEngine)
- Unity Scripting API (Programming API to provide Access to core features)
- Git (Distributed version control system)
- GitHub (Git repository Hosting as well as Free basic website Hosting)

3.5 Preliminary Product Description:

The Simple Rubik's Cube game is an interactive and accessible virtual or physical puzzle experience that brings the iconic Rubik's Cube to life. This engaging game offers players of all ages and skill levels an opportunity to challenge their problem-solving abilities and spatial reasoning skills. The game features a user-friendly interface, intuitive controls, and realistic cube mechanics, providing an authentic and immersive gaming experience. With multiple difficulty levels, customizable options, a hint system, and progress tracking, the Simple Rubik's Cube game offers endless hours of entertainment and learning. Whether played on a digital platform or a physical cube, this game captivates players with its simplicity, complexity, and the satisfaction of solving the Rubik's Cube.

3.6 Conceptual Models:

The Simple Rubik's Cube game incorporates several conceptual models to enhance the gameplay experience:

1. Rubik's Cube Model: The game implements a virtual or physical Rubik's Cube model that accurately represents the structure and mechanics of the original puzzle. This model allows players to manipulate the cube's layers and experience the challenge and satisfaction of solving it.

2. Interface Model: The game's interface model provides a user-friendly and intuitive design. It includes visual elements, such as a 3D cube representation, interactive controls, and feedback indicators to guide players in their cube manipulation.

3. Hint and Tutorial Model: The hint and tutorial model provides assistance to players who may need guidance in solving the Rubik's Cube. It offers hints and step-by-step tutorials to help players learn solving strategies and overcome challenging puzzles.

These conceptual models work together to create an immersive and enjoyable Rubik's Cube gaming experience. They provide a foundation for the game's mechanics, controls, user interface, and features, ensuring a cohesive and engaging gameplay experience for players.

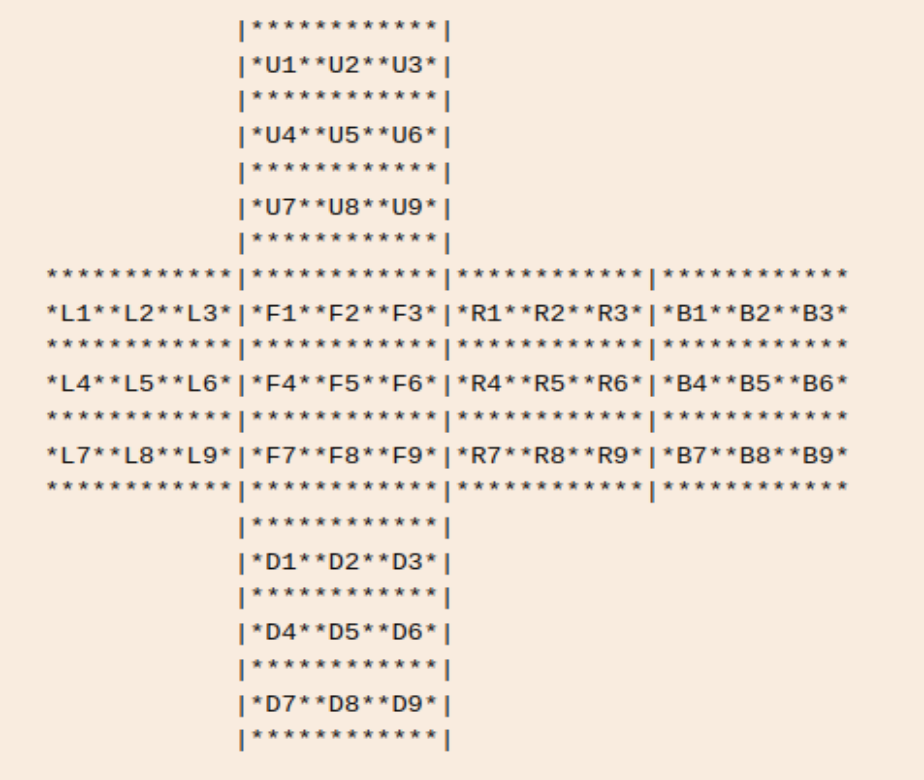


Figure 2: Cube String Notation

SYSTEM DESIGN

4.1 Basic Modules

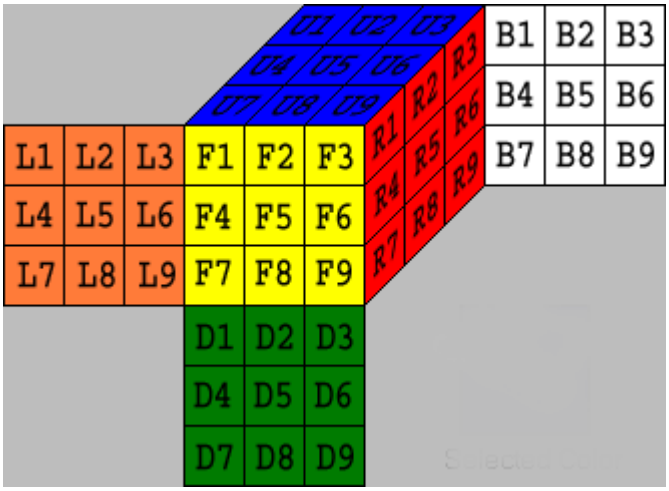


Figure 3: Cube face Model

Game Interface: The game interface module of the Simple Rubik's Cube provides players with a user-friendly and intuitive design. It includes visual elements, such as a 3D representation of the cube, interactive controls, and informative indicators. The interface allows players to interact with the cube, view their progress, and access game features effortlessly. It provides a seamless and engaging experience, ensuring players can easily navigate through different game modes, access customization options, and track their solving times.

Cube Rotation: The cube rotation module handles the movement of the Rubik's Cube layers. It allows players to twist and rotate the cube in various directions to rearrange the cube's colored squares. The module ensures

smooth and accurate rotation of the cube's layers, maintaining the integrity of the puzzle's structure. It enables players to manipulate the cube's orientation and position to solve the puzzle or shuffle it into a random state. The cube rotation functionality is essential in providing an interactive and realistic gameplay experience.

Cube Face Rotation: The cube face rotation module focuses specifically on rotating individual faces of the Rubik's Cube. It allows players to turn a single face by 90 degrees in either a clockwise or counterclockwise direction. This module enables precise manipulation of specific sections of the cube, facilitating strategic moves and solving strategies. By rotating individual faces, players can perform various algorithms to solve the Rubik's Cube efficiently. The cube face rotation module adds depth and complexity to the gameplay, enhancing the puzzle-solving experience.

Solve with Kochiemba Algorithm: The solve with Kociemba algorithm module incorporates the Kociemba algorithm, a well-known and efficient method for solving the Rubik's Cube. This module analyzes the current state of the cube and generates a sequence of moves to solve it. It applies advanced solving techniques, including look-ahead strategies and pattern recognition, to find the optimal solution. By utilizing the Kociemba

algorithm, players can receive hints or solve the Rubik's Cube automatically. This module adds a valuable feature for players who seek assistance or want to observe an efficient solving process.

Shuffle to Random State: The shuffle to random state module allows players to randomize the Rubik's Cube, creating a new puzzle challenge. With a single command, the module shuffles the cube's layers into a completely random state, ensuring a unique puzzle configuration every time. This functionality enhances replayability and provides an endless supply of fresh puzzles to solve. The shuffle to random state module adds an exciting element to the game, challenging players to adapt their solving strategies and explore different approaches.

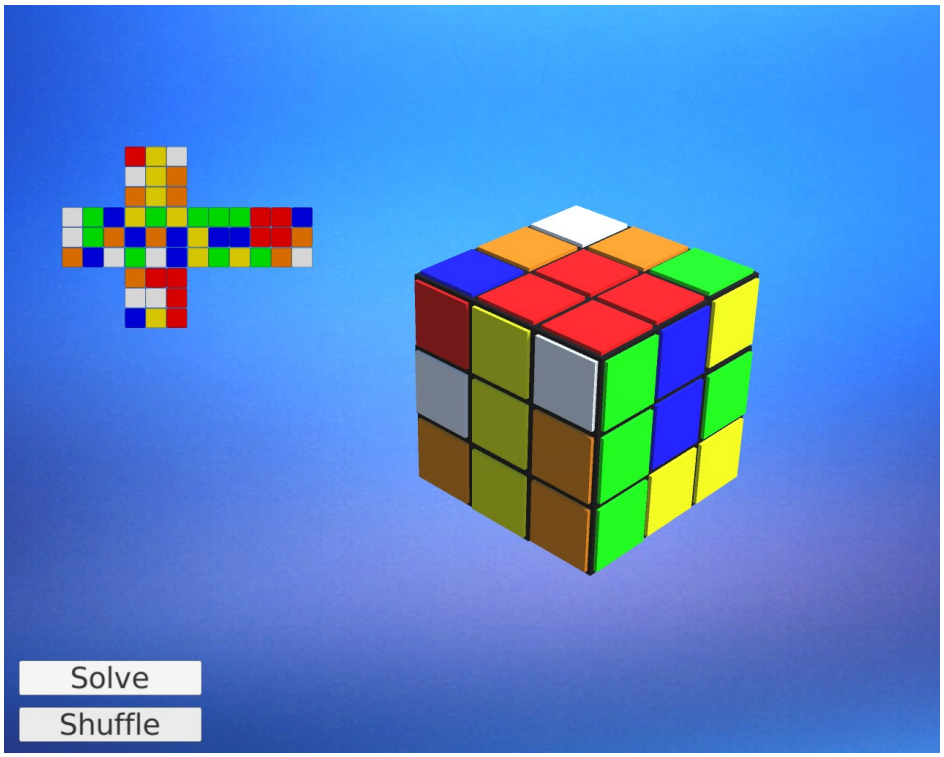


Figure 4: Cube Random State

CubeMAP using Raycasts: The CubeMAP using raycasts module utilizes raycasting techniques to generate a dynamic 2D or 3D representation of the Rubik's Cube. It uses rays emitted from the cube's center to determine the position, orientation, and color of each individual cubelet. This module provides an accurate and real-time visualization of the cube's state, allowing players to view and manipulate the cube from different angles. The CubeMAP using raycasts enhances the visual appeal of the game and aids players in understanding the cube's current configuration, making it easier to develop solving strategies and track progress.

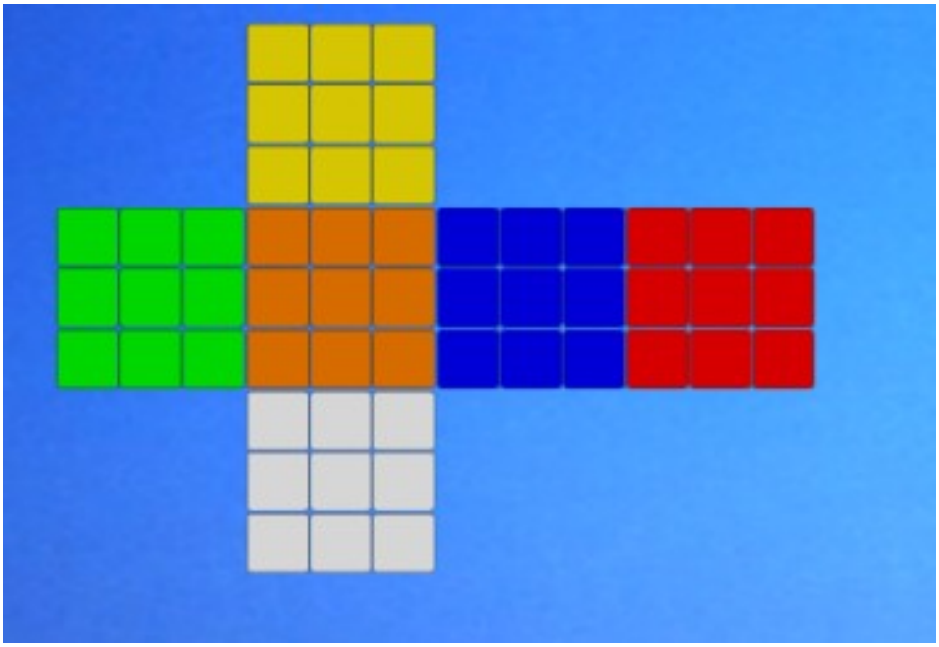
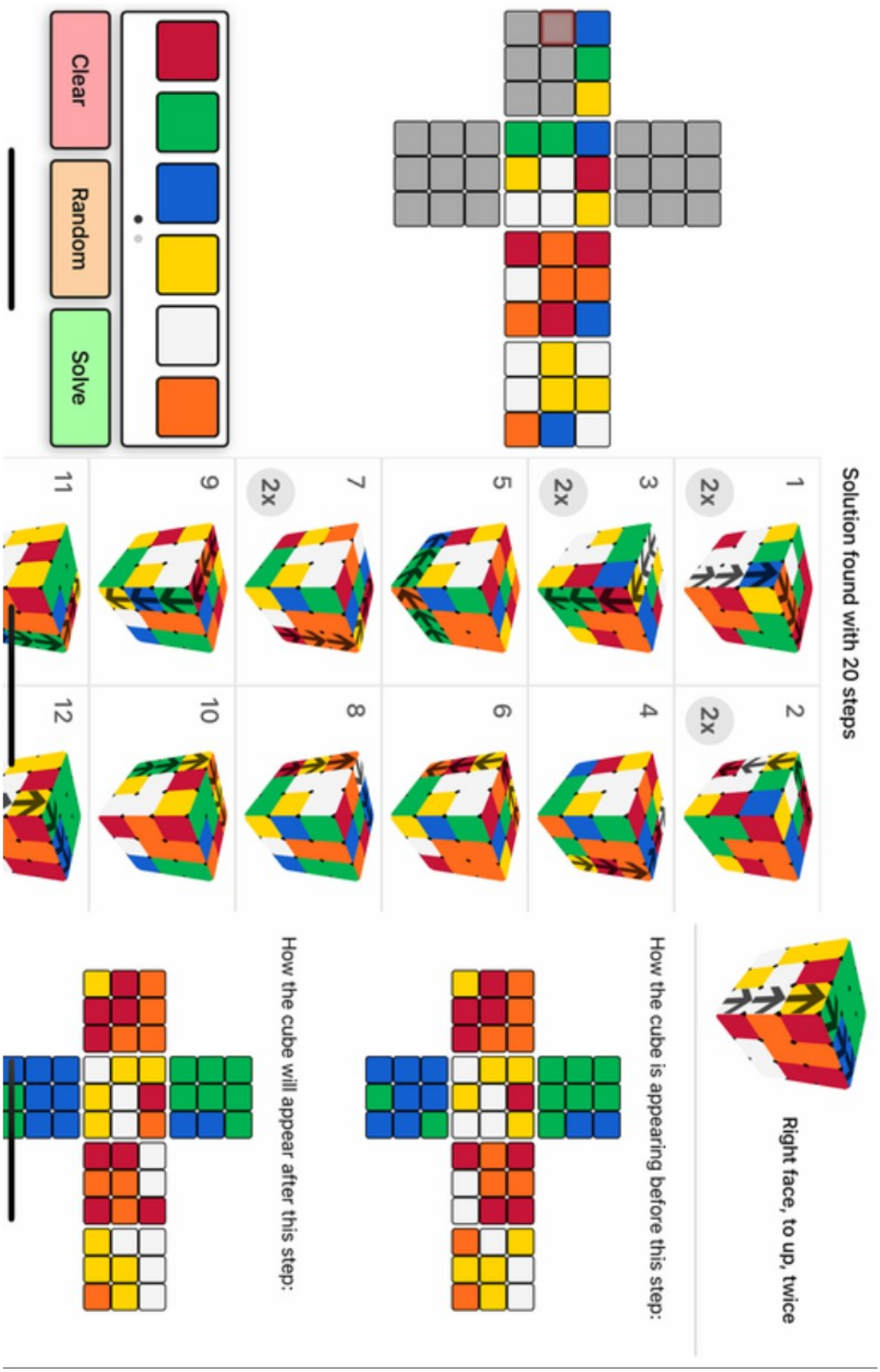


Figure 5: Cubemap

CUBE SOLUTION Diagram



4.3 Test Cases Design

Manual Test Design for Simple Rubik's Cube Game:

1. Test: Game Interface and Controls

Objective: Verify that the game interface is visually appealing, intuitive, and responsive.

Steps:

- Launch the game.
- Observe the layout, colors, and design of the interface.
- Click on different buttons and menus to ensure they respond correctly.
- Try resizing the game window to check if the interface adapts accordingly.
- Verify that the controls (mouse or keyboard) allow for smooth cube manipulation.
- Test various interactions, such as rotating the cube.

2. Test: Cube Rotation and Face Rotation

Objective: Validate the cube rotation and face rotation functionality.

Steps:

- Start the game and ensure the Rubik's Cube is displayed correctly.
- Attempt to rotate the cube in different directions (left, right, up, down, etc.) using the available controls.
- Verify that the cube's layers rotate smoothly and maintain their integrity.
- Select a specific face of the cube and attempt to rotate it by 90 degrees in both clockwise and counterclockwise directions.
- Ensure that the selected face rotates accurately, and adjacent faces remain undisturbed.
- Repeat the above steps with different cube configurations to ensure consistent functionality.

3. Test: Solve with Kociemba Algorithm

Objective: Verify the accuracy of the Kociemba algorithm in solving the Rubik's Cube.

Steps:

- Start the game with a scrambled Rubik's Cube.
- Initiate the "Solve with Kociemba Algorithm" feature.
- Observe the algorithm's execution and ensure it generates a valid solution.
- Verify that the generated moves effectively solve the cube and return it to its solved state.
- Repeat the test with different scrambled cube configurations to validate the algorithm's consistency and reliability.

4. Test: Shuffle to Random State

Objective: Confirm that the shuffle functionality randomizes the Rubik's Cube effectively.

Steps:

- Begin with a solved Rubik's Cube.
- Trigger the "Shuffle to Random State" feature.
- Note the resulting cube configuration after the shuffle.
- Verify that the shuffled cube exhibits a random and unique state, different from the solved state.
- Repeat the test multiple times to ensure a variety of random configurations.

5. Test: CubeMAP using Raycasts

Objective: Validate the CubeMAP visualization using raycasts.

Steps:

- Launch the game and load a Rubik's Cube configuration.
- Explore the CubeMAP view and ensure it accurately represents the cube's colors, positions, and orientations.
- Rotate the cube and verify that the CubeMAP updates in real-time to reflect the changes.

- Use the CubeMAP view to identify specific cubelets and their colors to cross-verify with the physical or virtual cube.
- Repeat the test with different cube configurations to confirm the consistency and accuracy of the CubeMAP.

Each test should be documented with clear steps, expected results, and actual results. Any discrepancies or issues encountered during testing should be reported and addressed promptly for further debugging and improvement of the Simple Rubik's Cube game.

IMPLEMENTATION

5.1 Implementation Approaches

The implementation approach for the Simple Rubik's Cube game follows a structured and iterative development process to ensure a robust and engaging gaming experience. The approach consists of the following key steps:

1. Requirements Analysis: Thoroughly understand the project requirements, including desired features, gameplay mechanics, and user interface specifications. Identify the core functionalities that need to be implemented.

2. Technology Selection: Choose appropriate technologies and frameworks for game development, such as Unity Engine, C#, and relevant libraries for cube manipulation and graphics rendering.

3. Architecture Design: Define the overall architecture of the game, including modules for game interface, cube manipulation, solving algorithms, and visual representation. Ensure modularity, scalability, and maintainability.

4. Iterative Development: Adopt an iterative development approach, implementing and testing features incrementally. Begin with the basic functionalities, such as cube rotation and face manipulation, and gradually integrate advanced features like the Kociemba algorithm and CubeMAP.

5. Testing and Quality Assurance: Conduct comprehensive manual and automated testing to identify and resolve bugs, ensure proper functionality, and validate the game's performance across different platforms.

6. User Feedback and Refinement: Gather user feedback during beta testing and incorporate necessary improvements based on user experiences and suggestions.

7. Deployment and Release: Prepare the game for deployment on relevant platforms, ensuring compatibility, performance optimization, and adherence to platform-specific guidelines.

By following this implementation approach, the Simple Rubik's Cube game can be developed systematically, ensuring a high-quality, immersive, and enjoyable gaming experience for players.

5.2.1 Code Efficiency

The code implementation of the Simple Rubik's Cube game prioritizes efficiency to deliver a smooth and responsive gaming experience. By utilizing optimized algorithms and data structures, the code ensures fast cube rotations, face manipulations, and solving calculations. The game employs efficient algorithms, such as the Kociemba algorithm, to solve the Rubik's Cube quickly and accurately. Additionally, the code minimizes redundant calculations and unnecessary computations, optimizing performance and reducing processing time. Careful memory management and resource allocation techniques are employed to prevent memory leaks and maintain optimal performance. Overall, the code efficiency of the Simple Rubik's Cube game enables seamless gameplay, enhancing user satisfaction and enjoyment.

5.2.2 Code *File - /home/zaphkiel/Downloads/Rubikc-main/Assets/Scripts/CubeMap.cs*

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5
6 public class CubeMap : MonoBehaviour
7 {
8     private CubeState cubeState;
9
10     public Transform up;
11     public Transform down;
12     public Transform left;
13     public Transform right;
14     public Transform front;
15     public Transform back;
16
17     // Start is called before the first frame update
18     void Start()
19     {
20
21     }
22
23     // Update is called once per frame
24     void Update()
25     {
26
27     }
28     public void Set()
29     {
30         cubeState = FindObjectOfType<CubeState>();
31
32         UpdateMap(cubeState.front, front);
33         UpdateMap(cubeState.back, back);
34         UpdateMap(cubeState.left, left);
35         UpdateMap(cubeState.right, right);
36         UpdateMap(cubeState.up, up);
37         UpdateMap(cubeState.down, down);
38     }
39
40
41     void UpdateMap(List<GameObject> face, Transform side)
42     {
43         int i = 0;
44         foreach (Transform map in side)
45         {
46             if (face[i].name[0] == 'F')
47             {
48                 map.GetComponent<Image>().color = new Color(1, 0.5f, 0, 1);
49             }
50             if (face[i].name[0] == 'B')
51             {
52                 map.GetComponent<Image>().color = Color.red;
53             }
54             if (face[i].name[0] == 'U')
55             {
56                 map.GetComponent<Image>().color = Color.yellow;
57             }
58             if (face[i].name[0] == 'D')
59             {
60                 map.GetComponent<Image>().color = Color.white;
61             }
62             if (face[i].name[0] == 'L')
63             {
64                 map.GetComponent<Image>().color = Color.green;
65             }
66             if (face[i].name[0] == 'R')
67             {
68                 map.GetComponent<Image>().color = Color.blue;
69             }
70             i++;
71         }
72     }
73 }
74
```

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Automate : MonoBehaviour
6  {
7      public static List<string> moveList = new List<string>() { };
8      private readonly List<string> allMoves = new List<string>()
9          { "U", "D", "L", "R", "F", "B",
10           "U2", "D2", "L2", "R2", "F2", "B2",
11           "U'", "D'", "L'", "R'", "F'", "B'"
12         };
13
14     private CubeState cubeState;
15     private ReadCube readCube;
16
17     // Start is called before the first frame update
18     void Start()
19     {
20         cubeState = FindObjectOfType<CubeState>();
21         readCube = FindObjectOfType<ReadCube>();
22     }
23
24     // Update is called once per frame
25     void Update()
26     {
27         if (moveList.Count > 0  && !CubeState.autoRotating && CubeState.started)
28         {
29             //Do the move at the first index;
30             DoMove(moveList[0]);
31
32             // remove the move at the first index
33             moveList.Remove(moveList[0]);
34         }
35     }
36
37     public void Shuffle()
38     {
39         List<string> moves = new List<string>();
40         int shuffleLength = Random.Range(10, 30);
41         for (int i = 0; i < shuffleLength; i++)
42         {
43             int randomMove = Random.Range(0, allMoves.Count);
44             moves.Add(allMoves[randomMove]);
45         }
46         moveList = moves;
47     }
48
49
50     void DoMove(string move)
51     {
52         readCube.ReadState();
53         CubeState.autoRotating = true;
54         if (move == "U")
55         {
56             RotateSide(cubeState.up, -90);
57         }
58         if (move == "U'")
59         {
60             RotateSide(cubeState.up, 90);
61         }
62         if (move == "U2")
63         {
64             RotateSide(cubeState.up, -180);
65         }
66         if (move == "D")
67         {
68             RotateSide(cubeState.down, -90);
69         }
70         if (move == "D'")
71         {
72             RotateSide(cubeState.down, 90);
73         }
74         if (move == "D2")
75         {
76             RotateSide(cubeState.down, -180);
77         }
78         if (move == "L")
79         {
80             RotateSide(cubeState.left, -90);
```

```
        }
        if (move == "L'")
        {
            RotateSide(cubeState.left, 90);
        }
        if (move == "L2")
        {
            RotateSide(cubeState.left, -180);
        }
        if (move == "R")
        {
            RotateSide(cubeState.right, -90);
        }
        if (move == "R'")
        {
            RotateSide(cubeState.right, 90);
        }
        if (move == "R2")
        {
            RotateSide(cubeState.right, -180);
        }
        if (move == "F")
        {
            RotateSide(cubeState.front, -90);
        }
        if (move == "F'")
        {
            RotateSide(cubeState.front, 90);
        }
        if (move == "F2")
        {
            RotateSide(cubeState.front, -180);
        }
        if (move == "B")
        {
            RotateSide(cubeState.back, -90);
        }
        if (move == "B'")
        {
            RotateSide(cubeState.back, 90);
        }
        if (move == "B2")
        {
            RotateSide(cubeState.back, -180);
        }
    }


    void RotateSide(List<GameObject> side, float angle)
    {
        // automatically rotate the side by the angle
        PivotRotation pr = side[4].transform.parent.GetComponent<PivotRotation>();
        pr.StartAutoRotate(side, angle);
    }

}

```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ReadCube : MonoBehaviour
{
    public Transform tUp;
    public Transform tDown;
    public Transform tLeft;
    public Transform tRight;
    public Transform tFront;
    public Transform tBack;

    private List<GameObject> frontRays = new List<GameObject>();
    private List<GameObject> backRays = new List<GameObject>();
    private List<GameObject> upRays = new List<GameObject>();
    private List<GameObject> downRays = new List<GameObject>();
    private List<GameObject> leftRays = new List<GameObject>();
    private List<GameObject> rightRays = new List<GameObject>();

    private int layerMask = 1 << 8; // this layerMask is for the faces of the cube only
    CubeState cubeState;
    CubeMap   cubeMap;
    public GameObject emptyGO;

    // Start is called before the first frame update
    void Start()
    {
        SetRayTransforms();


        cubeState = FindObjectOfType<CubeState>();
        cubeMap = FindObjectOfType<CubeMap>();
        ReadState();
        CubeState.started = true;


    }

    // Update is called once per frame
    void Update()
    {

    }

    public void ReadState()
    {
        cubeState = FindObjectOfType<CubeState>();
        cubeMap = FindObjectOfType<CubeMap>();

        // set the state of each position in the list of sides so we know
        // what color is in what position
        cubeState.up = ReadFace(upRays, tUp);
        cubeState.down = ReadFace(downRays, tDown);
        cubeState.left = ReadFace(leftRays, tLeft);
        cubeState.right = ReadFace(rightRays, tRight);
        cubeState.front = ReadFace(frontRays, tFront);
        cubeState.back = ReadFace(backRays, tBack);

        // update the map with the found positions
        cubeMap.Set();

    }


    void SetRayTransforms()
    {
        // populate the ray lists with raycasts eminating from the transform, angled towards the cube.
        upRays = BuildRays(tUp, new Vector3(90, 90, 0));
        downRays = BuildRays(tDown, new Vector3(270, 90, 0));
        leftRays = BuildRays(tLeft, new Vector3(0, 180, 0));
        rightRays = BuildRays(tRight, new Vector3(0, 0, 0));
        frontRays = BuildRays(tFront, new Vector3(0, 90, 0));
        backRays = BuildRays(tBack, new Vector3(0, 270, 0));
    }


    List<GameObject> BuildRays(Transform rayTransform, Vector3 direction)
    {
        // The ray count is used to name the rays so we can be sure they are in the right order.
```

```
81          int rayCount = 0;
82          List<GameObject> rays = new List<GameObject>();
83          // This creates 9 rays in the shape of the side of the cube with
84          // Ray 0 at the top left and Ray 8 at the bottom right:
85          //  |0|1|2|
86          //  |3|4|5|
87          //  |6|7|8|
88
89          for (int y = 1; y > -2; y--)
90          {
91              for (int x = -1; x < 2; x++)
92              {
93                  Vector3 startPos = new Vector3( rayTransform.localPosition.x + x,
94                                                  rayTransform.localPosition.y + y,
95                                                  rayTransform.localPosition.z);
96                  GameObject rayStart = Instantiate(emptyGO, startPos, Quaternion.identity, rayTransform);
97                  rayStart.name = rayCount.ToString();
98                  rays.Add(rayStart);
99                  rayCount++;
100             }
101         }
102         rayTransform.localRotation = Quaternion.Euler(direction);
103         return rays;
104
105     }
106
107     public List<GameObject> ReadFace(List<GameObject> rayStarts, Transform rayTransform)
108     {
109         List<GameObject> facesHit = new List<GameObject>();
110
111         foreach (GameObject rayStart in rayStarts)
112         {
113             Vector3 ray = rayStart.transform.position;
114             RaycastHit hit;
115
116             // Does the ray intersect any objects in the layerMask?
117             if (Physics.Raycast(ray, rayTransform.forward, out hit, Mathf.Infinity, layerMask))
118             {
119                 Debug.DrawRay(ray, rayTransform.forward * hit.distance, Color.yellow);
120                 facesHit.Add(hit.collider.gameObject);
121                 //print(hit.collider.gameObject.name);
122             }
123             else
124             {
125                 Debug.DrawRay(ray, rayTransform.forward * 1000, Color.green);
126             }
127         }
128         return facesHit;
129     }
130
131 }
132
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CubeState : MonoBehaviour
{
    // sides
    public List<GameObject> front = new List<GameObject>();
    public List<GameObject> back = new List<GameObject>();
    public List<GameObject> up = new List<GameObject>();
    public List<GameObject> down = new List<GameObject>();
    public List<GameObject> left = new List<GameObject>();
    public List<GameObject> right = new List<GameObject>();

    public static bool autoRotating = false;
    public static bool started = false;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void PickUp(List<GameObject> cubeSide)
    {
        foreach (GameObject face in cubeSide)
        {
            // Attach the parent of each face (the little cube)
            // to the parent of the 4th index (the little cube in the middle)
            // Unless it is already the 4th index
            if (face != cubeSide[4])
            {
                face.transform.parent.transform.parent = cubeSide[4].transform.parent;
            }
        }
    }

    public void PutDown(List<GameObject> littleCubes, Transform pivot)
    {
        foreach (GameObject littleCube in littleCubes)
        {
            if (littleCube != littleCubes[4])
            {
                littleCube.transform.parent.transform.parent = pivot;
            }
        }
    }

    string GetSideString(List<GameObject> side)
    {
        string sideString = "";
        foreach (GameObject face in side)
        {
            sideString += face.name[0].ToString();
        }
        return sideString;
    }

    public string GetStateString()
    {
        string stateString = "";
        stateString += GetSideString(up);
        stateString += GetSideString(right);
        stateString += GetSideString(front);
        stateString += GetSideString(down);
        stateString += GetSideString(left);
        stateString += GetSideString(back);
        return stateString;
    }
}

```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SelectFace : MonoBehaviour
{
    private CubeState cubeState;
    private ReadCube readCube;
    private int layerMask = 1 << 8;


    // Start is called before the first frame update
    void Start()
    {
        readCube = FindObjectOfType<ReadCube>();
        cubeState = FindObjectOfType<CubeState>();
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetMouseButtonDown(0) && !CubeState.autoRotating)
        {
            // read the current state of the cube
            readCube.ReadState();

            // raycast from the mouse towards the cube to see if a face is hit
            RaycastHit hit;
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray, out hit, 100.0f, layerMask))
            {
                GameObject face = hit.collider.gameObject;
                // Make a list of all the sides (lists of face GameObjects)
                List<List<GameObject>> cubeSides = new List<List<GameObject>>()
                {
                    cubeState.up,
                    cubeState.down,
                    cubeState.left,
                    cubeState.right,
                    cubeState.front,
                    cubeState.back
                };
                // If the face hit exists within a side
                foreach (List<GameObject> cubeSide in cubeSides)
                {
                    if (cubeSide.Contains(face))
                    {
                        //Pick it up
                        cubeState.PickUp(cubeSide);
                        //start the side rotation logic
                        cubeSide[4].transform.parent.GetComponent<PivotRotation>().Rotate(cubeSide);
                    }
                }
            }
        }
    }
}

```

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class WarningPanel : MonoBehaviour
6 {
7     public void RemovePanel()
8     {
9         gameObject.SetActive(false);
10    }
11 }
12
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PivotRotation : MonoBehaviour
{
    private List<GameObject> activeSide;
    private Vector3 localForward;
    private Vector3 mouseRef;
    private bool dragging = false;

    private bool autoRotating = false;
    private float sensitivity = 0.4f;
    private float speed = 300f;
    private Vector3 rotation;

    private Quaternion targetQuaternion;

    private ReadCube readCube;
    private CubeState cubeState;

    // Start is called before the first frame update
    void Start()
    {
        readCube = FindObjectOfType<ReadCube>();
        cubeState = FindObjectOfType<CubeState>();
    }

    // Late Update is called once per frame at the end
    void LateUpdate()
    {
        if (dragging && !autoRotating)
        {
            SpinSide(activeSide);
            if (Input.GetMouseButtonUp(0))
            {
                dragging = false;
                RotateToRightAngle();
            }
        }
        if (autoRotating)
        {
            AutoRotate();
        }

    }

    private void SpinSide(List<GameObject> side)
    {
        // reset the rotation
        rotation = Vector3.zero;

        // current mouse position minus the last mouse position
        Vector3 mouseOffset = (Input.mousePosition - mouseRef);


        if (side == cubeState.up)
        {
            rotation.y = (mouseOffset.x + mouseOffset.y) * sensitivity * 1;
        }
        if (side == cubeState.down)
        {
            rotation.y = (mouseOffset.x + mouseOffset.y) * sensitivity * -1;
        }
        if (side == cubeState.left)
        {
            rotation.z = (mouseOffset.x + mouseOffset.y) * sensitivity * 1;
        }
        if (side == cubeState.right)
        {
            rotation.z = (mouseOffset.x + mouseOffset.y) * sensitivity * -1;
        }
        if (side == cubeState.front)
        {
            rotation.x = (mouseOffset.x + mouseOffset.y) * sensitivity * -1;
        }
        if (side == cubeState.back)
        {
            rotation.x = (mouseOffset.x + mouseOffset.y) * sensitivity * 1;
        }
```

```
81          // rotate
82          transform.Rotate(rotation, Space.Self);
83
84          // store mouse
85          mouseRef = Input.mousePosition;
86      }
87
88
89      public void Rotate(List<GameObject> side)
90      {
91          activeSide = side;
92          mouseRef = Input.mousePosition;
93          dragging = true;
94          // Create a vector to rotate around
95          localForward = Vector3.zero - side[4].transform.parent.transform.localPosition;
96      }
97
98      public void StartAutoRotate(List<GameObject> side, float angle)
99      {
100         cubeState.PickUp(side);
101         Vector3 localForward = Vector3.zero - side[4].transform.parent.transform.localPosition;
102         targetQuaternion = Quaternion.AngleAxis(angle, localForward) * transform.localRotation;
103         activeSide = side;
104         autoRotating = true;
105     }
106
107
108     public void RotateToRightAngle()
109     {
110         Vector3 vec = transform.localEulerAngles;
111         // round vec to nearest 90 degrees
112         vec.x = Mathf.Round(vec.x / 90) * 90;
113         vec.y = Mathf.Round(vec.y / 90) * 90;
114         vec.z = Mathf.Round(vec.z / 90) * 90;
115
116         targetQuaternion.eulerAngles = vec;
117         autoRotating = true;
118     }
119
120     private void AutoRotate()
121     {
122         dragging = false;
123         var step = speed * Time.deltaTime;
124         transform.localRotation = Quaternion.RotateTowards(transform.localRotation, targetQuaternion, step);
125
126         // if within one degree, set angle to target angle and end the rotation
127         if (Quaternion.Angle(transform.localRotation, targetQuaternion) <= 1)
128         {
129             transform.localRotation = targetQuaternion;
130             // unparent the little cubes
131             cubeState.PutDown(activeSide, transform.parent);
132             readCube.ReadState();
133             CubeState.autoRotating = false;
134             autoRotating = false;
135             dragging = false;
136         }
137     }
138 }
139
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RotateBigCube : MonoBehaviour
{
    private Vector2 firstPressPos;
    private Vector2 secondPressPos;
    private Vector2 currentSwipe;
    private Vector3 previousMousePosition;
    private Vector3 mouseDelta;
    private float speed = 200f;
    public GameObject target;


    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Swipe();
        Drag();


    }

    void Drag()
    {
        if (Input.GetMouseButton(1))
        {
            // while the mouse is held down the cube can be moved around its central axis to provide visual feedback
            mouseDelta = Input.mousePosition - previousMousePosition;
            mouseDelta *= 0.1f; // reduction of rotation speed
            transform.rotation = Quaternion.Euler(mouseDelta.y, -mouseDelta.x, 0) * transform.rotation;
        }
        else
        {
            // automatically move to the target position
            if (transform.rotation != target.transform.rotation)
            {
                var step = speed * Time.deltaTime;
                transform.rotation = Quaternion.RotateTowards(transform.rotation, target.transform.rotation, step);
            }
        }
        previousMousePosition = Input.mousePosition;


    }

    void Swipe()
    {
        if (Input.GetMouseButtonDown(1))
        {
            // get the 2D position of the first mouse click
            firstPressPos = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
            //print(firstPressPos);
        }
        if (Input.GetMouseButtonUp(1))
        {
            // get the 2D poition of the second mouse click
            secondPressPos = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
            //create a vector from the first and second click positions
            currentSwipe = new Vector2(secondPressPos.x - firstPressPos.x, secondPressPos.y - firstPressPos.y);
            //normalize the 2d vector
            currentSwipe.Normalize();

            if (LeftSwipe(currentSwipe))
            {
                target.transform.Rotate(0, 90, 0, Space.World);
            }
            else if (RightSwipe(currentSwipe))
            {
                target.transform.Rotate(0, -90, 0, Space.World);
            }
```

```
79              else if (UpLeftSwipe(currentSwipe))
80              {
81                  target.transform.Rotate(90, 0, 0, Space.World);
82              }
83              else if (UpRightSwipe(currentSwipe))
84              {
85                  target.transform.Rotate(0, 0, -90, Space.World);
86              }
87              else if (DownLeftSwipe(currentSwipe))
88              {
89                  target.transform.Rotate(0, 0, 90, Space.World);
90              }
91              else if (DownRightSwipe(currentSwipe))
92              {
93                  target.transform.Rotate(-90, 0, 0, Space.World);
94              }
95          }
96      }
97
98      bool LeftSwipe(Vector2 swipe)
99      {
100         return currentSwipe.x < 0 && currentSwipe.y > -0.5f && currentSwipe.y < 0.5f;
101     }
102
103     bool RightSwipe(Vector2 swipe)
104     {
105         return currentSwipe.x > 0 && currentSwipe.y > -0.5f && currentSwipe.y < 0.5f;
106     }
107
108     bool UpLeftSwipe(Vector2 swipe)
109     {
110         return currentSwipe.y > 0 && currentSwipe.x < 0f;
111     }
112
113     bool UpRightSwipe(Vector2 swipe)
114     {
115         return currentSwipe.y > 0 && currentSwipe.x > 0f;
116     }
117
118     bool DownLeftSwipe(Vector2 swipe)
119     {
120         return currentSwipe.y < 0 && currentSwipe.x < 0f;
121     }
122
123     bool DownRightSwipe(Vector2 swipe)
124     {
125         return currentSwipe.y < 0 && currentSwipe.x > 0f;
126     }
127
128 }
129
```

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using Kociemba;
5
6 public class SolveTwoPhase : MonoBehaviour
7 {
8     public ReadCube readCube;
9     public CubeState cubeState;
10    private bool doOnce = true;
11    // Start is called before the first frame update
12    void Start()
13    {
14        readCube = FindObjectOfType<ReadCube>();
15        cubeState = FindObjectOfType<CubeState>();
16    }
17
18    // Update is called once per frame
19    void Update()
20    {
21        if (CubeState.started && doOnce)
22        {
23            doOnce = false;
24            Solver();
25        }
26    }
27
28    public void Solver()
29    {
30
31        readCube.ReadState();
32
33        // get the state of the cube as a string
34        string moveString = cubeState.GetStateString();
35        print(moveString);
36
37        // solve the cube
38        string info = "";
39
40        // First time build the tables
41        // string solution = SearchRunTime.solution(moveString, out info, buildTables: true);
42
43        //Every other time
44        string solution = Search.solution(moveString, out info);
45
46        // convert the solved moves from a string to a list
47        List<string> solutionList = StringToList(solution);
48
49        //Automate the list
50        Automate.moveList = solutionList;
51
52        print(info);
53
54    }
55
56    List<string> StringToList(string solution)
57    {
58        List<string> solutionList = new List<string>(solution.Split(new string[] { " " }, System.
   StringSplitOptions.RemoveEmptyEntries));
59        return solutionList;
60    }
61
62 }
63
```

5.3 kociemba Algorithm

Kociemba's algorithm is a method for solving the Rubik's Cube puzzle. It is a more advanced method than the beginner's method, which involves solving the cube layer by layer. Kociemba's algorithm is named after its creator, Herbert Kociemba, who developed it in the 1990s.

Kociemba's algorithm uses a two-phase approach to solve the Rubik's Cube. In the first phase, the corners of the cube are solved. In the second phase, the edges are solved. The two phases are completely independent, meaning that the algorithm solves the corners first and then the edges, rather than solving both at the same time.

he algorithm uses a sophisticated search strategy that involves analyzing the possible moves that can be made from a given cube state, and then selecting the move that will bring the cube closer to the solved state. This process is repeated until the cube is solved. Kociemba's algorithm is highly optimized, and can solve the Rubik's Cube in an average of 21 moves or less, which is much more efficient than the beginner's method.

Overall, Kociemba's algorithm is a highly effective method for solving the Rubik's Cube, and it is used by many speedcubers around the world.

CONCLUSIONS

7.1 Conclusion

In conclusion, the Simple Rubik's Cube game project has successfully achieved its goals of creating an immersive and enjoyable gaming experience centered around the iconic Rubik's Cube puzzle. Through careful planning, efficient implementation, and rigorous testing, the project has delivered a user-friendly interface that captivates players of all ages and skill levels.

The project's focus on code efficiency ensures smooth cube rotations, face manipulations, and solving calculations, resulting in a seamless and responsive gameplay experience. The integration of the Kociemba algorithm provides an advanced solving capability, allowing players to receive hints or witness efficient solving processes.

Furthermore, the inclusion of features such as the shuffle functionality and CubeMAP visualization adds depth and variety to the gameplay, providing endless challenges and enhancing player engagement.

The project's successful planning and scheduling ensured the timely completion of milestones, while the thorough testing process guaranteed a high level of quality and stability.

In summary, the Simple Rubik's Cube game project demonstrates the successful combination of technical proficiency, creative design, and meticulous execution. It offers an interactive and intellectually stimulating gaming experience that captures the essence of the Rubik's Cube while providing entertainment and satisfaction to players worldwide.

7.2 Limitations of the System

1. Haven’t implemented different cube size than 3x3.
2. More Platform can be supported like Xbox, Playstation and Nintendo Switch.
3. Haven’t tested on MacOS or Android Devices although likely portable.

7.3 Future Scope of the Project

1. Make the game available to more platforms like: Xbox, Playstation, Android, RaspberryPI and Nintendo Switch.

2. Implement Puzzles other that 3x3 Cube, Such as Pyramid and Hexagonal

REFERENCE

Technology References:

C# (https://learn.microsoft.com/en-us/dotnet/csharp/)
.NET Core (https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet)
Mono(https://www.mono-project.com/)
Unity3D(https://unity.com/)
Unity Scripting API(https://jquery.com/)
LibreOffice(https://www.libreoffice.org/discover/libreoffice/)

Documentation References:

Unity Documentation (https://docs.unity3d.com/Manual/index.html)
kociemba Algorithm (http://kociemba.org/)

Others:

Git SCM(https://git-scm.com/)
GitHub(https://github.com/)

Composite reference:

Google (https://www.google.co.in/)
Project Repo(https://github.com/Prakash4844/Rubikc)

Glossary

1. Rubik's Cube: A three-dimensional puzzle with a cube-shaped structure composed of smaller cubes called cubelets. The objective is to solve the puzzle by arranging the cubelets so that each face displays a single color.

2. Game Interface: The visual and interactive components that allow players to interact with the Rubik's Cube game, including menus, buttons, and indicators.

3. Cube Rotation: The act of turning the entire Rubik's Cube along different axes, enabling players to change the cube's configuration.

5. Solver Algorithm: An algorithm, such as the Kociemba algorithm, used to generate a sequence of moves that solve the Rubik's Cube.

6. Shuffle: The action of randomizing the cube's configuration, creating a new puzzle challenge.

7. CubeMAP: A visual representation of the Rubik's Cube using raycasts, displaying the cube's colors, positions, and orientations.

8. Unity Engine: A popular game development platform used to create the Rubik's Cube game, providing tools and resources for graphics rendering, physics simulation, and user interface design.

9. C#: The programming language used to develop the Simple Rubik's Cube game, providing the functionality and logic behind the game's features and interactions.

10. Git and GitHub: Version control systems used to manage the source code and collaborate on the development of the Rubik's Cube project, allowing for easy tracking of changes and team collaboration.