Breakout Game Documentation

Overview

The Breakout game is a classic arcade game implemented in Unity, featuring a paddle, ball, bricks, and a scoring system. The game includes several components that manage game state, audio, levels, and user interface. The architecture follows common design patterns such as Singleton, Command, and Decorator patterns to ensure modularity and maintainability.

The Breakout game code is structured to ensure clear separation of concerns and extensibility. The Singleton pattern provides global access to managers, the Command pattern encapsulates game actions, and the Decorator pattern enhances the behavior of game objects. The UI components respond to game state changes to keep the player informed and engaged.

1. Game Architecture

1.1 Singleton Pattern

- **Purpose**: Ensures that only one instance of a class exists and provides a global point of access.
- **Implementation**: `GenericSingleton<T>` class

```
public class GenericSingleton<T> : MonoBehaviour where T : Component
{
    private static T instance;
    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                instance = FindObjectOfType<T>();
                if (instance == null)
                {
                    GameObject obj = new GameObject();
                    obj.name = typeof(T).Name;
                    instance = obj.AddComponent<T>();
                }
            }
            return instance;
        }
    }

    public virtual void Awake()
    {
```

```
        if (instance == null)
        {
            instance = this as T;
            DontDestroyOnLoad(this.gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

1.2 Command Pattern

- **Purpose**: Encapsulates a request as an object, thereby allowing parameterization and queuing of requests.
- **Implementation**: `ICommand` interface and various command classes

```
public interface ICommand
{
    void Execute();
}

public class StartGameCommand : ICommand
{
    public void Execute()
    {
        BreakoutGameManager.Instance.StartGame();
    }
}

public class UpdateScoreCommand : ICommand
{
    private int points;
    public UpdateScoreCommand(int points)
    {
        this.points = points;
    }
    public void Execute()
    {
        BreakoutGameManager.Instance.UpdateScore(points);
    }
}
```

1.3 Decorator Pattern

- **Purpose**: Allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.
- **Implementation**: `BrickDecorator` abstract class and `ExplodingBrick` concrete class

```
public abstract class BrickDecorator : Brick
{
    protected Brick decoratedBrick;

    public BrickDecorator(Brick brick)
    {
        decoratedBrick = brick;
    }

    public override void OnHit()
    {
        decoratedBrick.OnHit();
        ApplyDecoration();
    }

    protected abstract void ApplyDecoration();
}

public class ExplodingBrick : BrickDecorator
{
    public ExplodingBrick(Brick brick) : base(brick) { }

    protected override void ApplyDecoration()
    {
        Debug.Log("Explosion Effect Applied!");
    }
}
```

2. Game Components

2.1 BreakoutGameManager

- **Purpose**: Manages the game state including score, lives, and level transitions.
- **Key Methods**:
    - `StartGame()`: Initializes the game state.
    - `UpdateScore(int points)`: Updates the score and checks for win conditions.
    - `LoseLife()`: Decreases lives and handles game over conditions.
    - `SetLevelValues(int levelLives, int levelScoreToWin, int currentLevel)`: Sets values based on the current level.

```
public class BreakoutGameManager : GenericSingleton<BreakoutGameManager>
{
    public UnityEvent OnScoreChanged = new UnityEvent();
    public UnityEvent OnLivesChanged = new UnityEvent();
    public UnityEvent OnLevelChanged = new UnityEvent();
    public UnityEvent OnResetLostBall = new UnityEvent();
    public UnityEvent<bool> OnGameOver = new UnityEvent<bool>();
    public UnityEvent<bool> OnBallResleased = new UnityEvent<bool>();

    internal static int score { get; private set; } = 0;
```

```
    internal static int lives { get; private set; } = 3;
    internal static int level { get; private set; }
    internal static bool won { get; private set; } = false;
    internal static int scoreToWin { get; private set; } = 150;

    private Queue<ICommand> commandQueue = new Queue<ICommand>();

    private void Update()
    {
        if (commandQueue.Count > 0)
        {
            var command = commandQueue.Dequeue();
            command.Execute();
        }
    }

    public void QueueCommand(ICommand command)
    {
        commandQueue.Enqueue(command);
    }

    public void SetLevelValues(int levelLives, int levelScoreToWin, int
currentLevel)
    {
        lives = levelLives;
        scoreToWin = levelScoreToWin;
        level = currentLevel;
    }

    public void StartGame()
    {
        score = 0;
        OnScoreChanged?.Invoke();
        OnLivesChanged?.Invoke();
        OnLevelChanged?.Invoke();
    }
}
```

2.2 SoundManager

- **Purpose**: Handles playback of sound effects based on game events.
- **Key Methods**:
    - `PlayOnBallCollide()`, `PlayOnBrickBreaking()`, etc.: Methods to play specific sound effects.

```
public class SoundManager : MonoBehaviour
{
    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioScriptableObject levelsScriptableObject;

    private void Awake()
    {
        audioSource = audioSource ?? GetComponent<AudioSource>();
```

```
    }

    private void OnEnable()
    {
        AudioManager.Instance.OnBallColliding.AddListener(PlayOnBallCollide);

AudioManager.Instance.OnBrickBreaking.AddListener(PlayOnBrickBreaking);
        AudioManager.Instance.OnBallDroping.AddListener(PlayOnBallDroping);
        AudioManager.Instance.OnLevelFailed.AddListener(PlayOnLevelFailed);
        AudioManager.Instance.OnLevelSuccess.AddListener(PlayOnLevelSuccess);
        AudioManager.Instance.OnBtnClick.AddListener(PlayOnBtnClick);
    }

    private void OnDisable()
    {

AudioManager.Instance.OnBallColliding.RemoveListener(PlayOnBallCollide);

AudioManager.Instance.OnBrickBreaking.RemoveListener(PlayOnBrickBreaking);

AudioManager.Instance.OnBallDroping.RemoveListener(PlayOnBallDroping);

AudioManager.Instance.OnLevelFailed.RemoveListener(PlayOnLevelFailed);

AudioManager.Instance.OnLevelSuccess.RemoveListener(PlayOnLevelSuccess);
        AudioManager.Instance.OnBtnClick.RemoveListener(PlayOnBtnClick);
    }
}
```

2.3 LevelManager

- **Purpose**: Manages the loading and initialization of game levels.
- **Key Methods**:
    - `InitializeLevel(int levelIndex)`: Sets up the level based on index.
    - `LoadNextLevel()`: Advances to the next level.

```
public class LevelManager : MonoBehaviour
{
    [SerializeField] private LevelsScriptableObject levelsScriptableObject;
    [SerializeField] private Transform levelBricksContainerTransform;

    internal static int currentLevel { get; private set; } = 0;

    private void Start()
    {
        InitializeLevel(currentLevel);
    }

    private void InitializeLevel(int levelIndex)
    {
        if (levelIndex < 0 || levelIndex >=
levelsScriptableObject.levels.Length)
        {
```

```
            Debug.LogError("Level index out of bounds.");
            return;
        }

        var level = levelsScriptableObject.levels[levelIndex];
        BreakoutGameManager.Instance.QueueCommand(new
SetLevelValuesCommand(level.lives, level.scoreToWin, levelIndex));
        BreakoutGameManager.Instance.QueueCommand(new StartGameCommand());
        Instantiate(level.levelBrickContainer,
levelBricksContainerTransform);
    }

    public void LoadNextLevel()
    {
        currentLevel++;
        if (currentLevel < levelsScriptableObject.levels.Length)
        {
            InitializeLevel(currentLevel);
        }
        else
        {
            Debug.Log("No more levels to load.");
        }
    }
}
```

2.4 BreakoutUI

- **Purpose**: Manages the user interface elements and updates them based on game events.
- **Key Methods**:
    - `UpdateScore()`, `UpdateLives()`, `UpdateLevel()`: Update UI elements with current game data.
    - `ShowEndScreen(bool victory)`: Displays the end screen with the result of the game.

```
public class BreakoutUI : MonoBehaviour
{
    [SerializeField] private Text instructionText;
    [SerializeField] private Text scoreText;
    [SerializeField] private Text livesText;
    [SerializeField] private Text levelText;
    [SerializeField] private GameObject endScreen;
    [SerializeField] private Text endScreenText;
    [SerializeField] private Button restartBtn;

    private void OnEnable()
    {
        BreakoutGameManager.Instance.OnScoreChanged.AddListener(UpdateScore);
        BreakoutGameManager.Instance.OnLivesChanged.AddListener(UpdateLives);
        BreakoutGameManager.Instance.OnLevelChanged.AddListener(UpdateLevel);
        BreakoutGameManager.Instance.OnGameOver.AddListener(ShowEndScreen);
```

```
BreakoutGameManager.Instance.OnBallResleased.AddListener(OnBallReleased);
        restartBtn.onClick.AddListener(RestartGame);
    }

    private void OnDisable()
    {

BreakoutGameManager.Instance.OnScoreChanged.RemoveListener(UpdateScore);

BreakoutGameManager.Instance.OnLivesChanged.RemoveListener(UpdateLives);

BreakoutGameManager.Instance.OnLevelChanged.RemoveListener(UpdateLevel);

BreakoutGameManager.Instance.OnGameOver.RemoveListener(ShowEndScreen);

BreakoutGameManager.Instance.OnBallResleased.RemoveListener(OnBallReleased);
        restartBtn.onClick.RemoveListener(RestartGame);
    }

    private void OnBallReleased(bool isActive)
    {
        instructionText.gameObject.SetActive(isActive);
    }

    public void RestartGame()
    {
        AudioManager.Instance.OnBtnClick?.Invoke();
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void UpdateScore()
    {
        scoreText.text = "Score: " + BreakoutGameManager.score;
    }

    public void UpdateLives()
    {
        livesText.text = "Lives: " + BreakoutGameManager.lives;
    }

    public void UpdateLevel()
    {
        levelText.text = "Level: " + (BreakoutGameManager.level + 1);
    }

    public void ShowEndScreen(bool victory = false)
    {
        endScreen.SetActive(true);
        endScreenText.text = victory ? "Victory!\nFinal Score: " +
BreakoutGameManager.score : "Game Over\nFinal Score: " +
BreakoutGameManager.score;
    }
}
```

2.5 GameConstants

- **Purpose**: Defines constant values used across the game.
- **Key Constants**:
  - `Ball`: Tag for the ball object.
  - `HorizontalAxis`: Input axis name for paddle movement.

```
public class GameConstants
{
    public const string Ball = "Ball";
    public const string HorizontalAxis = "Horizontal";
}
```

2.6 Brick and Decorators

- **Purpose**: Represents the bricks in the game, with decorators adding additional behaviors.
- **Key Classes**:
  - `Brick`: Base class for bricks with hit points and point value.
  - `BrickDecorator`: Abstract decorator class for adding behavior to bricks.
  - `ExplodingBrick`: Concrete decorator that adds explosion effects to bricks.

```
public abstract class BrickDecorator : Brick
{
    protected Brick decoratedBrick;

    public BrickDecorator(Brick brick)
    {
        decoratedBrick = brick;
    }

    public override void OnHit()
    {
        decoratedBrick.OnHit();
        ApplyDecoration();
    }

    protected abstract void ApplyDecoration();
}

public class ExplodingBrick : BrickDecorator
{
    public ExplodingBrick(Brick brick) : base(brick) { }

    protected override void ApplyDecoration()
    {
        Debug.Log("Explosion Effect Applied!");
    }
}
```

3. User Interface (UI)

3.1 UI Elements

- **Components**:
  - `Text`: Displays score, lives, level, and end screen messages.
  - `Button`: For restarting the game.

3.2 UI Updates

- **Purpose**: Update the UI elements based on game state changes.
- **Key Methods**:
  - `UpdateScore()`: Updates the score display.
  - `UpdateLives()`: Updates the lives display.
  - `UpdateLevel()`: Updates the level display.
  - `ShowEndScreen(bool victory)`: Shows the end screen with the result.

```
public class BreakoutUI : MonoBehaviour
{
    // UI elements
    [SerializeField] private Text instructionText;
    [SerializeField] private Text scoreText;
    [SerializeField] private Text livesText;
    [SerializeField] private Text levelText;
    [SerializeField] private GameObject endScreen;
    [SerializeField] private Text endScreenText;
    [SerializeField] private Button restartBtn;

    // Methods for updating UI elements
    public void UpdateScore() { /* ... */ }
    public void UpdateLives() { /* ... */ }
    public void UpdateLevel() { /* ... */ }
    public void ShowEndScreen(bool victory = false) { /* ... */ }
}
```