WORK EXPLANATION

This is the explanation of the decisions made while developing the test project and what other improvements can be made.

COMMON ESSENTIAL SCRIPTS

1. **Save Functions**

The method that I decided to use for saving any value in the game (High score in this case) is using a binary save file. There are three scripts involved:

- **PlayerStats.cs**

  This script is the entry point for the saving process. At the end when the game is over the current score is sent to this script

```
public void AddScore(int amount)
{
    currentScore = amount;
}
```

```
public void GameSave()
{
    if(currentScore > highScore)
        highScore = currentScore;
    SaveManager.instance.SaveGame(this);
}
```

  which checks if it is greater that highscore. If the condition is met the new highscore is changed and it goes to the next script.

- **SaveManager.cs**

  It takes Playerstats as parameter. For saving the game it searches for the save file. Then the playerstats is passed to the PlayerDatabase class(explained Below).

```
PlayerDataBase saveDataBase = new PlayerDataBase(saveData);
```

  Then the playerdatabase is serialized and saved to the file.
  To load the save values it searches for the save file, deserializes it and returns a playerdatabase.

```
PlayerDataBase saveData = converter.Deserialize(dataStream) as PlayerDataBase;
```

  It is called from the playerstats class which gets the playerdatabase and then we can set up playerstats with the saved values.

```
playerData = SaveManager.instance.LoadGame();
```

- **PlayerDatabase.cs**
  This class is just used to store the values. It has a constructor which takes in playerstats. When the file is being saved, a new playerdatabase is created passing playerstats. Then the values from playerstats is saved in the database of this script and then that values are then saved in file as playerdatabase.

```
public PlayerDataBase(PlayerStats stats)
{
    highScore = stats.GetHighScore;
}
```

Any number of values can be saved through this process either Boolean,int, float etc.

2. **GameEvents.cs**

The project follows Observer pattern. This script is just list of events that other scripts can subscribe to.

```
public class GameEvents
{
    public static Action<GameStates> onGameStateChanged;
}
```

Al the action events can be added in this script. As the actions are static we can just call GameEvents.(event name)

3. **SceneManagerScript.cs**

This script is used for loading the scenes using transitions. The scenes are put in an enum with there sceneIndex numbers.

```
public enum Scenes
{
    MainMenu = 0,

    Game = 1,
}
```

While loading the scene we use a coroutine that triggers the transition and loads the scene using the enum.

```
IEnumerator LoadTransition(int sceneIndex)
{
    transition.SetTrigger("Start");
    yield return new WaitForSeconds(1f);
    SceneManager.LoadScene(sceneIndex);
}
```

Improvements that can be done in this is, not all scenes need transition. So we can make separate functions. One that uses transition and other that just directly loads the scene.

MAIN MENU SCENE

There is a main menu in the game which consists of 2 buttons PLAY & QUIT.

There is a display of HighScore under the title which fetches the highscore from the binary save file and displays it on the menu.

The script that handles all the main menu functions is named MainMenuManager.cs.

It handles the Start of the game which takes us to the next scene, Quit the game, and a function to show the highScore.

GAME SCENE

1. **GameStateManager.cs**
   This script handles the state of the game. Whenever the game state is changed it throws and event which is picked up by the appropriate subscriber script.

```
public void ChangeGameState(GameStates state)
{
    gameState = state;
    GameEvents.onGameStateChanged.Invoke(state);
}
```

   There is an enum for all the states for the game. Initially the state is set to the state levelstarting.

```
public enum GameStates
{
    levelStarting,
    inProgress,
    paused,
    gameOver,
}
```

```
ChangeGameState(GameStates.levelStarting);
```

2. **CountdownTimer.cs**
   When the state is levelstarting, the countdown to begin the game starts. Whenever the state is changed to levelstarting the event thrown is captured by this script which starts the coroutine.

```
StartCoroutine("CountDown");
```

The coroutine runs until the value of the time reaches 0 waiting for a second in between. Then changes the game state to inProgress.

```
GameStateManager.instance.ChangeGameState(GameStates.inProgress);
```

```
while (tempTimer > 0)
{
    timerText.text = ((int)tempTimer).ToString();
    yield return new WaitForSeconds(1f);
    tempTimer--;
}
```

3. GameTimer.cs

This script keeps the timer of 30 seconds as said in the description and after the timer is finished it changes the state to gameOver.It uses text and image fill amount as the UI.

```
tempTimer -= Time.deltaTime;
timerText.text = ((int)tempTimer).ToString();
timerImage.fillAmount = tempTimer/timer;

if (tempTimer <= 0)
    GameStateManager.instance.ChangeGameState(GameStates.gameOver);
```

4. PinManager.cs

This script handles the pins in the game. It has a numberofPins variable which can be specified. Whenever the numberofpinsleft variable reaches 0, the game state is changed to gameOver.

```
if (numberofPinsLeft == 0)
{
    GameStateManager.instance.ChangeGameState(GameStates.gameOver);
}
```

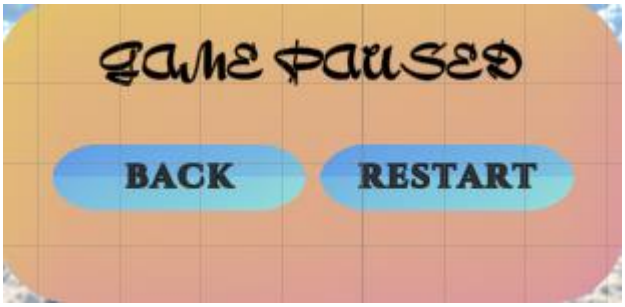
5. Spawner.cs

This script handles all the spawning gameobjects in the game. All the prefabs are specified and when the gamestate is changed to levelstarting, the ball is spawned after which the ground is spawned.
Lastly the pincount is pulled from the pinmanager script and with for loop the pins are spawned in random positions.

```
if(state == GameStates.levelStarting)
{
    GameObject ball = Instantiate(ballPrefab, new Vector3(0, 1, 0), Quaternion.identity);

    cam.GetComponent<CameraFollow>().target = ball;
    ball.gameObject.GetComponent<PlayerControl>().bounds = GroundLength / 2;

    var ground = Instantiate(groundPrefab, transform.position, Quaternion.identity);

    ground.transform.localScale = new Vector3(GroundLength, 1, GroundLength);

    SpawnPins();
}
```

6. **MenuManager.cs**

   This game has a pause menu with 2 buttons to RESUME and to RESTART.



   This script handles the functionality of these buttons and also to change the state of the game accordingly.

7. **LevelDataTracker.cs**

   This script can handle all the data throughout the level. The puspose of this script is as follows.
   The playerstats sricpt will contain the details of the whole game. This is why the level this script is created to handle all the data in the level and when the gamestate is changed to game over everything is saved to playerstats and then the saving process mentioned above takes place.

   This script also handles the UI changes in the game. I could be done in separate script to make it scalable. For simplicity of the project as we only have one variable it is done in same script.

   Another option could be, when the score is changed we might need to play some animation and also change the ui. We can make a new event in the GameEvents.cs script and throw this event when score is changed which will be subscribed by respective classes.

GAME OBJECT SCRIPTS

1. **CameraFollow.cs**

   This script is used to make the camera follow the player. The camera stays in a position with an offset and follows the player using Lerp.

2. **PlayerControl.cs**

   This script handles all the player movements. Whenever the player collides with a pin if the pin is standing then 10 points are added or else if the pin has fallen down, nothing happens.

```
if (collisionInfo.collider.CompareTag("Pin"))
{
    var pin = collisionInfo.gameObject.GetComponent<PinScript>();
    if (pin.IsStanding())
    {
        LevelDataTracker.instance.ChangeScore(10);
        pin.ChangeState();
    }
}
```

3. **Pinscript.cs**

   This script handles the pins. Every pin has this script. When the ball collides with the pin a function isStanding is called. It checks if the transform.up of the pin is < some threshold then the pin has fallen down.

```
public bool IsStanding()
{
    if (transform.up.y > standingThreshold || transform.up.y < -standingThreshold)
        return true;

    return false;
}
```

   This part of the code works perfectly with minor bugs which can be improved.

Overall, the project was quite interesting to develop. There are a lot of ne functions that can be added like a minimap, hit popup, obviously some music and many more.
This was my explanation of the code that I did. I would be very happy to answer any question regarding the project.