Note: If the project throws a dotnet mgcb error, please follow the following steps :

1. **Enter the .config folder (folder may be hidden)**
2. **Right click on the dotnet-tools-json**
3. **Check the box at the bottom which says “Unblock”**

Solution taken from:
https://stackoverflow.com/questions/73099265/monogame-error-msb3073 the-command-dotnet-tool-restore-exited-with-code-1

CONTENTS

GAME OVERVIEW

INTRODUCTION

Lumberjack is a captivating 2D game built using the MonoGame platform, where the players take up the role of a skilled lumberjack with the task of chopping down trees. This game tests the players precision, timing and predictability.

The player has to keep cutting a part of the tree and this will increase the score. If you take a break, then you risk the timer running out. Every time you cut the tree, the timer gets replenished by a certain amount. This way the game keeps going until you either hit a bark and knock yourself or you take breaks and the timer runs out.

There is also a friendly bird that drops eggs on you to give you energy to cut the tree. But mind the branches when you try to pick up these eggs.

The controls of the game are very easy to pick up and the game itself is easy to understand. Currently, in the prototype the game has one level and this level keeps looping until the player meets the gameover conditions.

GAME CONTROLS

Left Arrow	Move Left
Right Arrow	Move Right

IMPLEMENTED REQUIREMENTS

ASSETS

- Most of the assets used in the game were created by me using Photoshop
- All of these textures are loaded using the Content Manager in MonoGame

```
// Load the score font
font = Content.Load<SpriteFont>("Graphics\\gameFont");

// Load the timer textures
timer = Content.Load<Texture2D>("Graphics\\LoadingSprite");
timerBg = Content.Load<Texture2D>("Graphics\\LoadingBgSprite");

// Load the background resources
background = Content.Load<Texture2D>("Graphics/background");
foreground = Content.Load<Texture2D>("Graphics/foreground");
homeScreenBackground = Content.Load<Texture2D>("Graphics/HomeScreenBackground");
controlsScreenBackground = Content.Load<Texture2D>("Graphics/ControlsScreen");
gameOverScreenBackground = Content.Load<Texture2D>("Graphics/GameOverScreen");
```

KEY BINDS

- The player can be controlled using the arrow keys on the keyboard to move left and right. An event manager has been used to separate the key events from the code. The Command Manager class is used for this and we have to pass the key bind and the Game Action it has to implement when it is pressed

```
1 reference
private void InitializeBindings()
{
    commandManager.AddKeyboardBinding(Keys.Escape, ExitGame);
    commandManager.AddKeyboardBinding(Keys.Left, MoveLeft);
    commandManager.AddKeyboardBinding(Keys.Right, MoveRight);
    commandManager.AddKeyboardBinding(Keys.R, RestartGame);
}
```

COLLISIONS

- Collisions are taken care of by the Collision Manager.

Any object you would like to have collision with must inherit from the Collidable class. With this, you set a bounding sphere for this specific object which will then be detected by the collision manager in case of any collisions.

- The Egg and the Player classes have inherited from the Collidable class and every time there is a collision, the OnCollision method will be called which then will increase the score.
- The tree and the player are checked for collisions using enums. The tree has an enum that specifies if the bark has no branches or a branch on the left or the right. The player also has an enum specifying his position either left or right. If the tree and the player are on the same side, then a collision is identified and an appropriate method is called.

```
2 references
private void CheckCollisionsWithTree(ePlayerSide playerSide)
{

    if ((tree.barksList.Peek().BarkSide == eBarkSide.Left && playerSide == ePlayerSide.LEFT) ||
        (tree.barksList.Peek().BarkSide == eBarkSide.Right && playerSide == ePlayerSide.RIGHT))
    {
        EventManager.PlayerDeath.Execute();
    }
}
```

- It can also be noted from the code that there is a clear separation between collision detection and collision response. Event listeners have been used to make the code loosely coupled.

ANIMATIONS

- The animations are loaded and rendered with the game time. This makes the animation frame-rate independent.
- All of the animations are loaded using the Animation class which takes into account the sprite sizes, count and the time.

```
1 reference
public void Update(GameTime gameTime)
{
    CurrentAnimation.Position = Position;

    CurrentAnimation.Update(gameTime);

    // Update cutting animation
    if (isCutting && !CurrentAnimation.Active)
    {
        // Cutting Animation is completed
        isCutting = false;

        if (playerSide == ePlayerSide.LEFT)
            CurrentAnimation = PlayerAnimIdleLeft;
        else
            CurrentAnimation = PlayerAnimIdleRight;
    }
}
```

LEVEL

- A Data-driven approach is followed to design the level. The tree basically shows the level and the branches are read from a text file.

```
2 references
public Level(Stream fileStream)
{
    lines = new List<string>();

    loader = new Loader(fileStream);

    LoadTreeData(fileStream);
}

1 reference
private void LoadTreeData(Stream fileStream)
{
    // Load the level
    lines = loader.ReadLinesFromTextFile();
    levelWidth = (lines.Count > 0) ? lines[0].Length : 0;
}
```

- In the prototype, we have three different types of barks for the tree - No branch, left branch and a right branch.

.	No Branch
[	Left Branch
]	Right Branch

```
1 reference
public eBarkSide LoadNextBark()
{
    if(barkIndex >= levelWidth)
        barkIndex = 0;

    eBarkSide barkSide = eBarkSide.Center;

    if (barkIndex < levelWidth)
    {
        switch (lines[levelIndex][barkIndex])
        {
            case '.':
                barkSide = eBarkSide.Center; break;
            case '[':
                barkSide = eBarkSide.Left; break;
            case ']':
                barkSide = eBarkSide.Right; break;
        }
    }
    barkIndex++;
    return barkSide;
}
```

- When the end of the level is reached, the index is reset again so that it becomes an endless loop turning the game into something similar to an endless runner.
- When the end of the level is reached, the index is reset again so that it becomes an endless loop turning the game into something similar to an endless runner.

SCORING

- There is a score manager in the game that makes use of Event Listeners to update the score and also ensure maintainability in the game.

```
1 reference
public ScoreManager()
{
    _score = 0;
    EventManager.RestartGame.AddListener(ResetScore);
}

1 reference
public void ResetScore()
{
    HighScoreManager.Instance.UpdateHighScore(_score);
    _score = 0;
}
1 reference
public void CutTree() => AddScore(cutTreeScore);
1 reference
public void CatchEgg() => AddScore(catchEggScore);
2 references
private void AddScore(int score)
{
    _score += score;
}
```

- There is also a High Score Manager that makes use of Serialization to demonstrate the game state/load save mechanism.

```
1 reference
HighScoreManager()
{
    _highScore = XMLParse.DeserializeFromXml<int>(filePath);
}

1 reference
public void UpdateHighScore(int highScore)
{
    if (highScore > HighScore)
    {
        XMLParse.SerializeToXml(filePath, highScore);
        _highScore = highScore;
    }
}
```

- There are two ways to score in the game. One is to cut the bark of the tree and the second way is to collect the egg which the bird drops. Both of these scoring systems utilise event listeners.

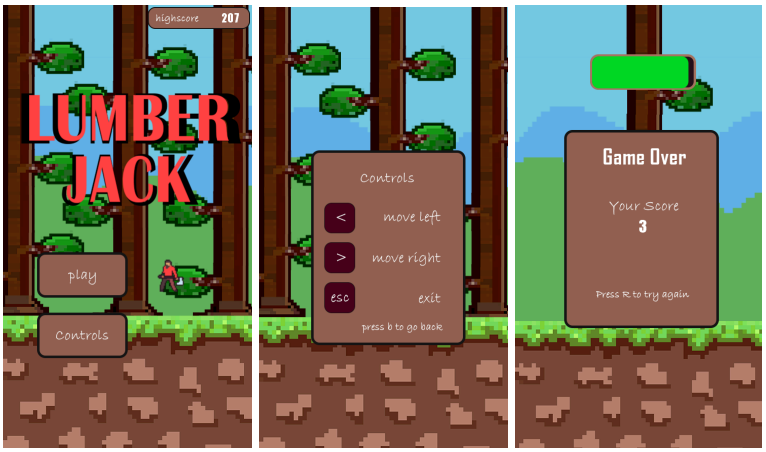
```
1 reference
private void SubscribeToEvents()
{
    // Subscribe to Player Death Event
    EventManager.PlayerDeath.AddListener(player.DeactivatePlayer);
    EventManager.PlayerDeath.AddListener(GameOver);

    // Subscribe to Collect Egg Event
    EventManager.CollectEgg.AddListener(Egg.Instance.DeactivateEgg);
    EventManager.CollectEgg.AddListener(scoreManager.CatchEgg);

    // Subscribe to Cut Tree Event
    EventManager.CutBark.AddListener(scoreManager.CutTree);
}
```

GAME SCREENS

- The Start Screen contains the Play and Controls options. State pattern is used to display these screens. You can use the arrow keys to navigate through the menu and Enter key to select it. The Controls screen basically shows the controls used in the game.



- The Game Over Screen contains your score and an option to restart the game.

NPC

- The bird in the game acts as an NPC. It utilises the power of Finite State Machines to switch between different states. In this case, there are two different states for the bird - Search state, Flee state.

```
1 reference
public void Initialize(Animation anim, Vector2 position)
{
    fsm = new FSM(this);

    // Create the states
    BirdSearchState search = new BirdSearchState();
    BirdFleeState flee = new BirdFleeState();

    // Add the created states to the FSM
    fsm.AddState(search);
    fsm.AddState(flee);

    // Create the transitions between the states
    search.AddTransition(new Transition(flee, () => IsEggDropped));

    // Set the starting state of the FSM
    fsm.Initialise("Search");

    BirdFlyingAnim = anim;
    Position = position;
    Active = false;

    EventManager.RestartGame.AddListener(Reset);
}
```

- The Search state is where the bird searches for the player and the Flee state is when the bird flies away after dropping the egg.

ADDITIONAL ELEMENTS

- The game contains a timer which will end the game if it

reaches zero. There is also a way to keep the game going without the time running out. You get some bonus time every time you cut a part of the tree. So if you continue cutting the tree as fast as possible, you can keep the timer full without risking it running out. This keeps the player engaged and also adds competitiveness to beat their own highscore.

DISCUSSION

An analysis of improvements to the speed of your algorithms that have been, or could be, achieved using your knowledge of hardware architecture

During the development of Lumberjack, the code was written with a focus on maintaining loose coupling to improve scalability and maintainability. However, when addressing enhancements to the speed of the algorithms, numerous strategies can be utilised.

One way is to use parallel processing. With the current hardware capabilities, CPUs have multiple cores to make this possible. In Lumberjack, we could distribute the computational tasks like collision detection and pathfinding for the NPC across multiple CPU cores using multithreading. This could show significant improvements in speed.

Targeted optimizations can also be done. Gaining knowledge on specific hardware architectures and building and optimising the game according to that hardware can improve the performance of the game significantly.

Batch Serialization can be introduced to reduce the number of serialisation calls and improve efficiency. Currently in Lumberjack, only one highscore is being serialised. But if there are more variables, it would be better to batch serialise all gameobjects of the same type together to minimise overhead associated with serialisation calls.

The serialised data can be reduced in size using data compression

techniques to improve serialisation/deserialization speed. So basically we compress the serialised files before saving them to disk and decompress them during deserialization to optimise storage efficiency.

Asset bundles can be created that can consist of all the assets like textures and sounds to minimise the number of content loading calls. Currently all the assets are loaded individually. Instead of this, we could load an entire asset bundle at once to improve loading efficiency.

Asset caching can be used to store frequently used assets in memory for reuse. Commonly used textures, sounds can be cached to avoid loading everytime we launch the game and also minimise disk usage.

By implementing these optimization techniques, Lumberjack can achieve faster asset loading times, reduced memory overhead and improved overall performance, resulting in a smoother gaming experience for players.