```csharp
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;
using Random = UnityEngine.Random;

public class ProceduralGeneration : MonoBehaviour
{
    public Master M;
    public Tilemap Map;
    public TileBase Rule;
    private int xIndex = 0;
    private List<int[][]> Chunks = new List<int[][]>();
    private int Height = 0;
    public int PlatformLength = 10;
    private int ChunkLength = 0;
    // This is the base chance for a tile to be empty. It slightly favours filled blocks
    private const int BaseEmptyChance = 45;
    // Tiles are more likely to be the same as the tiles next to them
    private const int NeighbourWeight = 22;
    private readonly int BaseFlatChance = 20;
    // This is to determine the fastest possible time to run a chunk
    private float Speed = 0f;
    private float TimeToRun;
    private float PreviousTime;
    private bool PlatformDeleted = false;
    // This is a grid showing where a tile has to be placed / has to not be placed for a jump to work
    // 0 means it will be empty, 1 means it will be filled, 2 means that it can be either
    private List<int[]> Jumps = new List<int[]>();

    public void Start()
    {
        // Creates starting values
        float width = Camera.main.orthographicSize * Camera.main.aspect * 2;
        ChunkLength = (int)Math.Ceiling(width);
        Height = (int)Math.Ceiling(Camera.main.orthographicSize * 2 - 2);
        Speed = M.GetSpeed();
        TimeToRun = ChunkLength * 1.5f / Speed;
        PreviousTime = Time.time - TimeToRun / 1.5f;
        CreateStart();
    }

    private void CreateStart()
    {
        // Removes the previous map's tiles
        Stats.TileColumns.Clear();

        // Creates the start of the map
        var wall = CreateWall();
        var platform = CreatePlatform();

        // Combines both the wall and the platform into a single chunk
        int[][] chunk = new int[1 + platform.Count][];
        chunk[0] = wall.ToArray();
        for (int counter = 0; counter < platform.Count; counter++)
        {
            chunk[counter + 1] = platform[counter].ToArray();
        }

        // Stores a 2 in the next 8 columns to allow other parts to overwrite the jump
        for (int j = 0; j < 8; j++)
        {
            AddEmptyJump();
        }

        // Draws out the initial chunk
        DrawChunk(chunk);
        Chunks.Add(chunk);

        // Creates 2 more chunks
        CreateNewTiles();
    }

    // Initialises a list of 1's to create a barrier to the left of the map
    private List<int> CreateWall()
    {
        var wall = new List<int>();
        for (int counter = 0; counter < Height; counter++)
        {
            wall.Add(1);
        }
        return wall;
    }

    // Creates a flat surface
    private List<List<int>> CreatePlatform()
    {
        // Creates a random height at which the starting platform should sit
        int platformHeight = Random.Range(2, Height - 6);
        var platform = new List<List<int>>();

        // Creates a template for a column as 1 line of the platform
        List<int> column = new List<int>();
        for (int counter = 0; counter < Height; counter++)
        {
            if (counter < platformHeight)
            {
                column.Add(1);
            }
            else
            {
                column.Add(0);
            }
        }

        // Adds the columns to the platform
        for (int counter = 0; counter < PlatformLength; counter++)
        {
            platform.Add(column);
        }
        return platform;
    }

    // Creates a new chunk and draws it
    public void CreateNewTiles()
    {
        var chunk = CreateChunk();
        DrawChunk(chunk);
        Chunks.Add(chunk);
        if (Chunks.Count > 2)
        {
            DeleteChunk();
        }
    }

    // Creates a chunk
    private int[][] CreateChunk()
    {
        // Calculates the weight
        float time = Time.time - PreviousTime;
        PreviousTime = Time.time;
        float weight = time * 1.5f / TimeToRun;

        var chunkList = new List<int[]>();
        for (int counter = 0; counter < ChunkLength; counter++)
        {
            // A parameter of CreateColumn is the previous column which is difficult for the first column
            // but simple for subsequent columns, hence the selection
            if (counter > 0)
            {
                chunkList.Add(CreateColumn(chunkList[counter - 1], weight));
            }
            else
            {
                // Gets the last column from the previous chunk
                chunkList.Add(CreateColumn(Chunks[Chunks.Count - 1][Chunks[Chunks.Count - 1].Length - 1], weight));
            }
        }
        return chunkList.ToArray();
    }

    // Creates a column
    private int[] CreateColumn(int[] previousColumn, float weight)
```

```
        {
            var columnList = new List<int>();
            // Takes the first set of jumps as the fixed tiles
            var fixedTiles = Jumps[0];

            FixedTiles(fixedTiles, previousColumn, weight);

            // Decides what the tile will be
            for (int counter = 0; counter < Height; counter++)
            {
                // If the tile is fixed, it just uses the number
                if (fixedTiles[counter] == 0 || fixedTiles[counter] == 1)
                {
                    columnList.Add(fixedTiles[counter]);
                }
                else
                {
                    int emptyChance = BaseEmptyChance;
                    if (Height - counter > 4)
                    {
                        // If the tile is not at the top, it checks the tiles to the left below to see what they are and uses
                        // This to influence the chance for the tile to be empty
                        if (previousColumn[counter] == 0) emptyChance += NeighbourWeight;
                        else emptyChance -= NeighbourWeight;

                        if (counter != 0)
                        {
                            if (columnList[counter - 1] == 0) emptyChance += NeighbourWeight;
                            else emptyChance -= NeighbourWeight;
                        }
                    }
                    else
                    {
                        // If the tile is near the top, it drastically increases the chance of it being empty
                        emptyChance = 95;
                    }

                    // This randomly picks the tile
                    if (weight >= 1)
                    {
                        if (Random.Range(1, 101) > emptyChance - 2 * weight) columnList.Add(1);
                        else columnList.Add(0);
                    }
                    else
                    {
                        if (Random.Range(1, 101) > emptyChance + 2 / weight) columnList.Add(1);
                        else columnList.Add(0);
                    }

                }
            }

            // Removes the 0th index of jumps and adds a new empty section to it
            Jumps.Remove(Jumps[0]);
            AddEmptyJump();

            return columnList.ToArray();
        }

        private void FixedTiles(int[] fixedTiles, int[] previousColumn, float weight)
        {
            // This loop looks at the previous column and if there is a valid space for the player to be,
            // it will ensure that they are able to progress
            for (int counter = 0; counter < Height - 4; counter++)
            {
                // Checks if the tile to the left was valid position
                if (previousColumn[counter] == 1 && previousColumn[counter + 1] == 0 && previousColumn[counter + 2] == 0)
                {
                    // Checks that there is not already a fixed position for the tile
                    if (fixedTiles[counter] == 2)
                    {
                        // If the two tiles above the tile are fixed, this will cause a clash which has to be resolved
                        if (fixedTiles[counter + 1] == 2 && fixedTiles[counter + 2] == 2)
                        {
                            // There is a random chance of the tile being above / below the previous one
                            int percent = Random.Range(1, 101);

                            int dropChance = BaseFlatChance;

                            // Increases the drop chance as it gets higher up the column
                            if ((Height - counter) < 8)
                            {
                                dropChance += 2 * (8 - Height + counter);
                            }

                            // Checks the tiles above the previous tile to see if they allow the user to jump
                            if (previousColumn[counter + 3] == 1 || previousColumn[counter + 4] == 1)
                            {
                                if (percent > dropChance)
                                {
                                    FixUp0(fixedTiles, counter);
                                }
                                else
                                {
                                    Jump(fixedTiles, counter, weight);
                                }
                            }
                            else if (percent > 50 / weight)
                            {
                                FixUp0(fixedTiles, counter);
                            }
                            else if (percent > (25 + dropChance / 2) / weight)
                            {
                                FixUp1(fixedTiles, counter);
                            }
                            else if (percent > dropChance / weight)
                            {
                                FixUp2(fixedTiles, counter);
                            }
                            else
                            {
                                Jump(fixedTiles, counter, weight);
                            }
                        }
                        else
                        {
                            // If both are a 0 then the terrain can just carry on
                            if (fixedTiles[counter + 1] == 0 || fixedTiles[counter + 2] == 0)
                            {
                                FixUp0(fixedTiles, counter);
                            }
                            else
                            {
                                // FixUp1 has a land and a gap above which works for the fixed tiles
                                if (fixedTiles[counter + 1] == 1)
                                {
                                    FixUp1(fixedTiles, counter);
                                }
                                // FixUp2 needs a land 2 above and doesn't affect the tile 1 above
                                else
                                {
                                    FixUp2(fixedTiles, counter);
                                }
                            }
                        }
                    }
                    else
                    {
                        if (fixedTiles[counter] == 1)
                        {
                            FixUp0(fixedTiles, counter);
                        }
                        else if (fixedTiles[counter] == 0)
                        {
                            FixGap(fixedTiles, counter);
                        }
                    }
                }
            }

            // Ensures that the map won't go above the limit of the screen
            for (int l = Height - 4; l < Height - 2; l++)
            {
                if (fixedTiles[l] == 2)
                {
                    if (previousColumn[l] == 1 && previousColumn[l + 1] == 0 && previousColumn[l + 2] == 0)
                    {
                        fixedTiles[l] = 0;
                        fixedTiles[l + 1] = 0;
                        fixedTiles[l + 2] = 0;
                    }
                }
            }
        }

        // Commonly used fixed tile groups
        private void FixGap(int[] fixedTiles, int counter)
        {
            fixedTiles[counter + 1] = 0;
            fixedTiles[counter + 2] = 0;
        }

        private void FixUp0(int[] fixedTiles, int counter)
        {
            fixedTiles[counter] = 1;
            fixedTiles[counter + 1] = 0;
            fixedTiles[counter + 2] = 0;
        }

        private void FixUp1(int[] fixedTiles, int counter)
        {
            fixedTiles[counter + 1] = 1;
            fixedTiles[counter + 2] = 0;
            fixedTiles[counter + 3] = 0;
        }

        private void FixUp2(int[] fixedTiles, int counter)
        {
            fixedTiles[counter + 2] = 1;
            fixedTiles[counter + 3] = 0;
            fixedTiles[counter + 4] = 0;
        }

        private void AddEmptyJump()
        {
            // Creates a new index for jumps and fills it all with 2's (empty)
            var emptyJump = new int[Height];
            for (int counter = 0; counter < Height; counter++)
            {
                emptyJump[counter] = 2;
            }
            Jumps.Add(emptyJump);
        }

        private void Jump(int[] fixedTiles, int yIndex, float weight)
        {
            fixedTiles[yIndex] = 0;
            fixedTiles[yIndex + 1] = 0;
            fixedTiles[yIndex + 2] = 0;
            fixedTiles[yIndex + 3] = 0;
            fixedTiles[yIndex + 4] = 0;

            // Prevents accessing invalid indexes of the array
            if (Height - yIndex > 3)
            {
                int length = JumpLength(weight);
                for (int counter = 1; counter < length + 1; counter++)
                {
                    Jumps[counter][yIndex + 1] = 0;
                    Jumps[counter][yIndex + 2] = 0;
                    Jumps[counter][yIndex + 3] = 0;
                    Jumps[counter][yIndex + 4] = 0;
                }
                Jumps[length][yIndex] = 1;
            }
        }

        // Randomly decides the jump length
        private int JumpLength(float weight)
        {
            int percent = Random.Range(1, 101);
            if (percent < 30 * weight)
            {
                return 3;
            }
            else if (percent < 60 * weight)
            {
                return 4;
            }
            else
            {
                return 5;
            }
        }

        private void DrawChunk(int[][] chunk)
        {
            // Checks every index of the chunk and adds a tile if it contains a 1
            for (int counter = 0; counter < chunk.Length; counter++)
            {
                for (int j = 0; j < chunk[counter].Length; j++)
                {
                    if (chunk[counter][j] == 1)
                    {
                        DrawTile(new Vector3Int(counter, j, 0));
                    }
                }

                // Adds the column to Tiles
                Stats.TileColumns.Add(chunk[counter]);
            }
            // Moves the next tile placement along by the length of the chunk
            xIndex += chunk.Length;
        }

        private void DrawTile(Vector3Int pos)
        {
            try
            {
                // Aligns the tile to be set
                pos.x += xIndex;
                Map.SetTile(pos, Rule);
            }
            catch { }
        }

        // Deletes an old chunk
        private void DeleteChunk()
        {
            if (PlatformDeleted)
            {
                for (int x = 0; x < ChunkLength; x++)
                {
                    for (int y = 0; y < Height; y++)
                    {
                        DeleteTile(new Vector3Int(x, y, 0), ChunkLength * 4);
                    }
                }
            }
            else
            {
                for (int x = 0; x < PlatformLength + 1; x++)
                {
                    for (int y = 0; y < Height; y++)
                    {
                        DeleteTile(new Vector3Int(x, y, 0), ChunkLength * 3 +
                       PlatformLength + 1);
                    }
                }
                PlatformDeleted = true;
            }
        }

        private void DeleteTile(Vector3Int pos, int deleteIndex)
        {
            try
            {
                pos.x += xIndex - deleteIndex;
                Map.SetTile(pos, null);
            }
            catch { }
        }
    }
```