

```
1 using System;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.Tilemaps;
5 using Random = UnityEngine.Random;
6
7 public class ProceduralGeneration : MonoBehaviour
8 {
9     public Master M;
10    public Tilemap Map;
11    public TileBase Rule;
12    private int xIndex = 0;
13    private List<int[][]> Chunks = new List<int[][]>();
14    private int Height = 0;
15    public int PlatformLength = 10;
16    private int ChunkLength = 0;
17    // This is the base chance for a tile to be empty. It slightly favours ↗
18    // filled blocks
19    private const int BaseEmptyChance = 45;
20    // Tiles are more likely to be the same as the tiles next to them
21    private const int NeighbourWeight = 22;
22    private readonly int BaseFlatChance = 20;
23    // This is to determine the fastest possible time to run a chunk
24    private float Speed = 0f;
25    private float TimeToRun;
26    private float PreviousTime;
27    private bool PlatformDeleted = false;
28    // This is a grid showing where a tile has to be placed / has to not be ↗
29    // placed for a jump to work
30    // 0 means it will be empty, 1 means it will be filled, 2 means that it ↗
31    // can be either
32    private List<int[]> Jumps = new List<int[]>();
33
34    public void Start()
35    {
36        // Creates starting values
37        float width = Camera.main.orthographicSize * Camera.main.aspect * 2;
38        ChunkLength = (int)Math.Ceiling(width);
39        Height = (int)Math.Ceiling(Camera.main.orthographicSize * 2 - 2);
40        Speed = M.GetSpeed();
41        TimeToRun = ChunkLength * 1.5f / Speed;
42        PreviousTime = Time.time - TimeToRun / 1.5f;
43        CreateStart();
44    }
45
46    private void CreateStart()
47    {
48        // Removes the previous map's tiles
49        Stats.TileColumns.Clear();
50
51        // Creates the start of the map
52        var wall = CreateWall();
53        var platform = CreatePlatform();
```

```
51
52     // Combines both the wall and the platform into a single chunk
53     int[][] chunk = new int[1 + platform.Count][];
54     chunk[0] = wall.ToArray();
55     for (int counter = 0; counter < platform.Count; counter++)
56     {
57         chunk[counter + 1] = platform[counter].ToArray();
58     }
59
60     // Stores a 2 in the next 8 columns to allow other parts to
    overwrite the jump
61     for (int j = 0; j < 8; j++)
62     {
63         AddEmptyJump();
64     }
65
66     // Draws out the initial chunk
67     DrawChunk(chunk);
68     Chunks.Add(chunk);
69
70     // Creates 2 more chunks
71     CreateNewTiles();
72 }
73
74 // Initialises a list of 1's to create a barrier to the left of the map
75 private List<int> CreateWall()
76 {
77     var wall = new List<int>();
78     for (int counter = 0; counter < Height; counter++)
79     {
80         wall.Add(1);
81     }
82     return wall;
83 }
84
85 // Creates a flat surface
86 private List<List<int>> CreatePlatform()
87 {
88     // Creates a random height at which the starting platform should sit
89     int platformHeight = Random.Range(2, Height - 6);
90     var platform = new List<List<int>>();
91
92     // Creates a template for a column as 1 line of the platform
93     List<int> column = new List<int>();
94     for (int counter = 0; counter < Height; counter++)
95     {
96         if (counter < platformHeight)
97         {
98             column.Add(1);
99         }
100         else
101         {
102             column.Add(0);
```

```
103     }
104 }
105
106 // Adds the columns to the platform
107 for (int counter = 0; counter < PlatformLength; counter++)
108 {
109     platform.Add(column);
110 }
111 return platform;
112 }
113
114 // Creates a new chunk and draws it
115 public void CreateNewTiles()
116 {
117     var chunk = CreateChunk();
118     DrawChunk(chunk);
119     Chunks.Add(chunk);
120     if (Chunks.Count > 2)
121     {
122         DeleteChunk();
123     }
124 }
125
126 // Creates a chunk
127 private int[][] CreateChunk()
128 {
129     // Calculates the weight
130     float time = Time.time - PreviousTime;
131     PreviousTime = Time.time;
132     float weight = time * 1.5f / TimeToRun;
133
134     var chunkList = new List<int[]>();
135     for (int counter = 0; counter < ChunkLength; counter++)
136     {
137         // A parameter of CreateColumn is the previous column which is
138         // difficult for the first column
139         // but simple for subsequent columns, hence the selection
140         if (counter > 0)
141         {
142             chunkList.Add(CreateColumn(chunkList[counter - 1], weight));
143         }
144         else
145         {
146             // Gets the last column from the previous chunk
147             chunkList.Add(CreateColumn(Chunks[Chunks.Count - 1][Chunks
148                 [Chunks.Count - 1].Length - 1], weight));
149         }
150     }
151     return chunkList.ToArray();
152 }
153
154 // Creates a column
155 private int[] CreateColumn(int[] previousColumn, float weight)
```

```
154     {
155         var columnList = new List<int>();
156         // Takes the first set of jumps as the fixed tiles
157         var fixedTiles = Jumps[0];
158
159         FixedTiles(fixedTiles, previousColumn, weight);
160
161         // Decides what the tile will be
162         for (int counter = 0; counter < Height; counter++)
163         {
164             // If the tile is fixed, it just uses the number
165             if (fixedTiles[counter] == 0 || fixedTiles[counter] == 1)
166             {
167                 columnList.Add(fixedTiles[counter]);
168             }
169             else
170             {
171                 int emptyChance = BaseEmptyChance;
172                 if (Height - counter > 4)
173                 {
174                     // If the tile is not at the top, it checks the tiles to the
175                     // left below to see what they are and uses
176                     // This to influence the chance for the tile to be empty
177                     if (previousColumn[counter] == 0) emptyChance +=
178                         NeighbourWeight;
179                     else emptyChance -= NeighbourWeight;
180
181                     if (counter != 0)
182                     {
183                         if (columnList[counter - 1] == 0) emptyChance +=
184                             NeighbourWeight;
185                         else emptyChance -= NeighbourWeight;
186                     }
187                 }
188                 else
189                 {
190                     // If the tile is near the top, it drastically increases
191                     // the chance of it being empty
192                     emptyChance = 95;
193                 }
194
195                 // This randomly picks the tile
196                 if (weight >= 1)
197                 {
198                     if (Random.Range(1, 101) > emptyChance - 2 * weight)
199                         columnList.Add(1);
200                     else columnList.Add(0);
201                 }
202                 else
203                 {
204                     if (Random.Range(1, 101) > emptyChance + 2 / weight)
205                         columnList.Add(1);
206                     else columnList.Add(0);
207                 }
208             }
209         }
210     }
```

```

201     }
202
203     }
204 }
205
206 // Removes the 0th index of jumps and adds a new empty section to it
207 Jumps.Remove(Jumps[0]);
208 AddEmptyJump();
209
210 return columnList.ToArray();
211 }
212
213 private void FixedTiles(int[] fixedTiles, int[] previousColumn, float weight)
214 {
215     // This loop looks at the previous column and if there is a valid
216     // space for the player to be,
217     // it will ensure that they are able to progress
218     for (int counter = 0; counter < Height - 4; counter++)
219     {
220         // Checks if the tile to the left was valid position
221         if (previousColumn[counter] == 1 && previousColumn[counter + 1]
222             == 0 && previousColumn[counter + 2] == 0)
223         {
224             // Checks that there is not already a fixed position for the
225             // tile
226             if (fixedTiles[counter] == 2)
227             {
228                 // If the two tiles above the tile are fixed, this will
229                 // cause a clash which has to be resolved
230                 if (fixedTiles[counter + 1] == 2 && fixedTiles[counter +
231                     2] == 2)
232                 {
233                     // There is a random chance of the tile being
234                     // above / below the previous one
235                     int percent = Random.Range(1, 101);
236
237                     int dropChance = BaseFlatChance;
238
239                     // Increases the drop chance as it gets higher up
240                     // the column
241                     if ((Height - counter) < 8)
242                     {
243                         dropChance += 2 * (8 - Height + counter);
244                     }
245
246                     // Checks the tiles above the previous tile to see
247                     // if they allow the user to jump
248                     if (previousColumn[counter + 3] == 1 ||
249                         previousColumn[counter + 4] == 1)
250                     {
251                         if (percent > dropChance)
252                         {

```

```
244         FixUp0(fixedTiles, counter);
245     }
246     else
247     {
248         Jump(fixedTiles, counter, weight);
249     }
250 }
251 else if (percent > 50 / weight)
252 {
253     FixUp0(fixedTiles, counter);
254 }
255 else if (percent > (25 + dropChance / 2) / weight)
256 {
257     FixUp1(fixedTiles, counter);
258 }
259 else if (percent > dropChance / weight)
260 {
261     FixUp2(fixedTiles, counter);
262 }
263 else
264 {
265     Jump(fixedTiles, counter, weight);
266 }
267 }
268 else
269 {
270     // If both are a 0 then the terrain can just carry
271     on
272     if (fixedTiles[counter + 1] == 0 || fixedTiles
273     [counter + 2] == 0)
274     {
275         FixUp0(fixedTiles, counter);
276     }
277     else
278     {
279         // FixUp1 has a land and a gap above which works
280         for the fixed tiles
281         if (fixedTiles[counter + 1] == 1)
282         {
283             FixUp1(fixedTiles, counter);
284         }
285         // FixUp2 needs a land 2 above and doesn't
286         affect the tile 1 above
287         else
288         {
289             FixUp2(fixedTiles, counter);
290         }
291     }
292 }
293 }
294 else
295 {
296     if (fixedTiles[counter] == 1)
```

```
293         {
294             FixUp0(fixedTiles, counter);
295         }
296         else if (fixedTiles[counter] == 0)
297         {
298             FixGap(fixedTiles, counter);
299         }
300     }
301 }
302 }
303
304 // Ensures that the map won't go above the limit of the screen
305 for (int l = Height - 4; l < Height - 2; l++)
306 {
307     if (fixedTiles[l] == 2)
308     {
309         if (previousColumn[l] == 1 && previousColumn[l + 1] == 0 &&
310             previousColumn[l + 2] == 0)
311         {
312             fixedTiles[l] = 0;
313             fixedTiles[l + 1] = 0;
314             fixedTiles[l + 2] = 0;
315         }
316     }
317 }
318
319 // Commonly used fixed tile groups
320 private void FixGap(int[] fixedTiles, int counter)
321 {
322     fixedTiles[counter + 1] = 0;
323     fixedTiles[counter + 2] = 0;
324 }
325
326 private void FixUp0(int[] fixedTiles, int counter)
327 {
328     fixedTiles[counter] = 1;
329     fixedTiles[counter + 1] = 0;
330     fixedTiles[counter + 2] = 0;
331 }
332
333 private void FixUp1(int[] fixedTiles, int counter)
334 {
335     fixedTiles[counter + 1] = 1;
336     fixedTiles[counter + 2] = 0;
337     fixedTiles[counter + 3] = 0;
338 }
339
340 private void FixUp2(int[] fixedTiles, int counter)
341 {
342     fixedTiles[counter + 2] = 1;
343     fixedTiles[counter + 3] = 0;
344     fixedTiles[counter + 4] = 0;
```

```
345     }
346
347     private void AddEmptyJump()
348     {
349         // Creates a new index for jumps and fills it all with 2's (empty)
350         var emptyJump = new int[Height];
351         for (int counter = 0; counter < Height; counter++)
352         {
353             emptyJump[counter] = 2;
354         }
355         Jumps.Add(emptyJump);
356     }
357
358     private void Jump(int[] fixedTiles, int yIndex, float weight)
359     {
360         fixedTiles[yIndex] = 0;
361         fixedTiles[yIndex + 1] = 0;
362         fixedTiles[yIndex + 2] = 0;
363         fixedTiles[yIndex + 3] = 0;
364         fixedTiles[yIndex + 4] = 0;
365
366         // Prevents accessing invalid indexes of the array
367         if (Height - yIndex > 3)
368         {
369             int length = JumpLength(weight);
370             for (int counter = 1; counter < length + 1; counter++)
371             {
372                 Jumps[counter][yIndex + 1] = 0;
373                 Jumps[counter][yIndex + 2] = 0;
374                 Jumps[counter][yIndex + 3] = 0;
375                 Jumps[counter][yIndex + 4] = 0;
376             }
377             Jumps[length][yIndex] = 1;
378         }
379     }
380
381     // Randomly decides the jump length
382     private int JumpLength(float weight)
383     {
384         int percent = Random.Range(1, 101);
385         if (percent < 30 * weight)
386         {
387             return 3;
388         }
389         else if (percent < 60 * weight)
390         {
391             return 4;
392         }
393         else
394         {
395             return 5;
396         }
397     }
```



```
398
399     private void DrawChunk(int[][] chunk)
400     {
401         // Checks every index of the chunk and adds a tile if it contains a 1
402         for (int counter = 0; counter < chunk.Length; counter++)
403         {
404             for (int j = 0; j < chunk[counter].Length; j++)
405             {
406                 if (chunk[counter][j] == 1)
407                 {
408                     DrawTile(new Vector3Int(counter, j, 0));
409                 }
410             }
411
412             // Adds the column to Tiles
413             Stats.TileColumns.Add(chunk[counter]);
414         }
415         // Moves the next tile placement along by the length of the chunk
416         xIndex += chunk.Length;
417     }
418
419     private void DrawTile(Vector3Int pos)
420     {
421         try
422         {
423             // Aligns the tile to be set
424             pos.x += xIndex;
425             Map.SetTile(pos, Rule);
426         }
427         catch { }
428     }
429
430     // Deletes an old chunk
431     private void DeleteChunk()
432     {
433         if (PlatformDeleted)
434         {
435             for (int x = 0; x < ChunkLength; x++)
436             {
437                 for (int y = 0; y < Height; y++)
438                 {
439                     DeleteTile(new Vector3Int(x, y, 0), ChunkLength * 4);
440                 }
441             }
442         }
443         else
444         {
445             for (int x = 0; x < PlatformLength + 1; x++)
446             {
447                 for (int y = 0; y < Height; y++)
448                 {
449                     DeleteTile(new Vector3Int(x, y, 0), ChunkLength * 3 +
```

```
PlatformLength + 1);
450         }
451     }
452     PlatformDeleted = true;
453 }
454 }
455
456 private void DeleteTile(Vector3Int pos, int deleteIndex)
457 {
458     try
459     {
460         pos.x += xIndex - deleteIndex;
461         Map.SetTile(pos, null);
462     }
463     catch { }
464 }
465 }
```