```
1 using System;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4 using UnityEngine.Tilemaps;
 5 using Random = UnityEngine.Random;
 7 public class ProceduralGeneration : MonoBehaviour
 8 {
 9
       public Master M;
       public Tilemap Map;
10
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
          filled blocks
       private const int BaseEmptyChance = 45;
18
19
       // Tiles are more likely to be the same as the tiles next to them
20
       private const int NeighbourWeight = 22;
21
       private readonly int BaseFlatChance = 20;
22
       // This is to determine the fastest possible time to run a chunk
23
       private float Speed = 0f;
24
       private float TimeToRun;
25
       private float PreviousTime;
26
       private bool PlatformDeleted = false;
       // This is a grid showing where a tile has to be placed / has to not be >
27
         placed for a jump to work
28
       // 0 means it will be empty, 1 means it will be filled, 2 means that it 
ightharpoons
          can be either
       private List<int[]> Jumps = new List<int[]>();
29
30
       public void Start()
31
32
            // Creates starting values
33
            float width = Camera.main.orthographicSize * Camera.main.aspect * 2;
34
35
            ChunkLength = (int)Math.Ceiling(width);
           Height = (int)Math.Ceiling(Camera.main.orthographicSize * 2 - 2);
36
37
            Speed = M.GetSpeed();
            TimeToRun = ChunkLength * 1.5f / Speed;
38
39
           PreviousTime = Time.time - TimeToRun / 1.5f;
40
           CreateStart();
41
       }
42
43
       private void CreateStart()
44
45
            // Removes the previous map's tiles
46
           Stats.TileColumns.Clear();
47
48
            // Creates the start of the map
49
           var wall = CreateWall();
50
            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++)</pre>
56
57
                 chunk[counter + 1] = platform[counter].ToArray();
58
             }
59
             // Stores a 2 in the next 8 columns to allow other parts to
60
               overwrite the jump
             for (int j = 0; j < 8; j++)
61
62
             {
63
                 AddEmptyJump();
64
             }
65
66
             // Draws out the initial chunk
             DrawChunk(chunk);
67
             Chunks.Add(chunk);
68
 69
             // Creates 2 more chunks
70
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++)</pre>
79
                 wall.Add(1);
80
81
             }
             return wall;
82
        }
83
84
        // Creates a flat surface
85
        private List<List<int>> CreatePlatform()
86
87
        {
             // Creates a random height at which the starting platform should sit
88
89
             int platformHeight = Random.Range(2, Height - 6);
             var platform = new List<List<int>>();
90
91
             // Creates a template for a column as 1 line of the platform
92
             List<int> column = new List<int>();
93
             for (int counter = 0; counter < Height; counter++)</pre>
94
95
             {
96
                 if (counter < platformHeight)</pre>
97
                 {
98
                     column.Add(1);
99
                 }
100
                 else
101
102
                     column.Add(0);
```

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
```

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

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
```

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

else columnList.Add(0);

200

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
201
202
203
                 }
204
            }
205
206
            // Removes the 0th index of jumps and adds a new empty section to it
            Jumps.Remove(Jumps[0]);
207
            AddEmptyJump();
208
209
            return columnList.ToArray();
210
211
        }
212
213
        private void FixedTiles(int[] fixedTiles, int[] previousColumn, float
214
        {
215
            // 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
216
            for (int counter = 0; counter < Height - 4; counter++)</pre>
217
218
219
                 // Checks if the tile to the left was valid position
                 if (previousColumn[counter] == 1 && previousColumn[counter + 1] >
220
                   == 0 && previousColumn[counter + 2] == 0)
221
                 {
222
                     // Checks that there is not already a fixed position for the >
                     if (fixedTiles[counter] == 2)
223
224
225
                         // If the two tiles above the tile are fixed, this will >
                        cause a clash which has to be resolved
226
                         if (fixedTiles[counter + 1] == 2 && fixedTiles[counter + >
```

```
2] == 2)
227
                         {
228
                              // There is a random chance of the tile being
                        above / below the previous one
229
                             int percent = Random.Range(1, 101);
230
231
                             int dropChance = BaseFlatChance;
232
233
                              // Increases the drop chance as it gets higher up
                        the column
234
                             if ((Height - counter) < 8)</pre>
235
                             {
                                  dropChance += 2 * (8 - Height + counter);
236
237
                             }
238
239
                              // Checks the tiles above the previous tile to see
                        if they allow the user to jump
                              if (previousColumn[counter + 3] == 1 ||
240
                        previousColumn[counter + 4] == 1)
241
242
                                  if (percent > dropChance)
243
                                  {
```

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
```

```
6
```

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

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
293
                         {
294
                              FixUp0(fixedTiles, counter);
295
                         }
296
                         else if (fixedTiles[counter] == 0)
297
298
                              FixGap(fixedTiles, counter);
299
                         }
                     }
300
301
                 }
302
             }
303
             // Ensures that the map won't go above the limit of the screen
304
305
             for (int 1 = Height - 4; 1 < Height - 2; 1++)</pre>
306
307
                 if (fixedTiles[1] == 2)
308
                      if (previousColumn[l] == 1 && previousColumn[l + 1] == 0 && →
309
                       previousColumn[1 + 2] == 0)
310
311
                         fixedTiles[1] = 0;
                         fixedTiles[l + 1] = 0;
312
                         fixedTiles[1 + 2] = 0;
313
314
                     }
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;
             fixedTiles[counter + 1] = 0;
329
330
             fixedTiles[counter + 2] = 0;
```

private void FixUp1(int[] fixedTiles, int counter)

private void FixUp2(int[] fixedTiles, int counter)

fixedTiles[counter + 1] = 1;

fixedTiles[counter + 2] = 0; fixedTiles[counter + 3] = 0;

fixedTiles[counter + 2] = 1; fixedTiles[counter + 3] = 0;

fixedTiles[counter + 4] = 0;

331

332333

334

335336

337338

339

340341342

343344

}

{

}

```
...CubeRunner\Assets\Scripts\Level\ProceduralGeneration.cs
```

```
R
```

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

```
398
399
         private void DrawChunk(int[][] chunk)
400
             // Checks every index of the chunk and adds a tile if it contains a 🤝
401
402
             for (int counter = 0; counter < chunk.Length; counter++)</pre>
403
                 for (int j = 0; j < chunk[counter].Length; j++)</pre>
404
405
406
                      if (chunk[counter][j] == 1)
407
                      {
                          DrawTile(new Vector3Int(counter, j, 0));
408
409
                      }
                 }
410
411
412
                 // Adds the column to Tiles
                 Stats.TileColumns.Add(chunk[counter]);
413
414
             // Moves the next tile placement along by the length of the chunk
415
             xIndex += chunk.Length;
416
         }
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++)</pre>
436
                      for (int y = 0; y < Height; y++)
437
438
439
                          DeleteTile(new Vector3Int(x, y, 0), ChunkLength * 4);
440
                      }
441
                 }
442
             }
443
             else
444
             {
                 for (int x = 0; x < PlatformLength + 1; x++)
445
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
             {
                 pos.x += xIndex - deleteIndex;
460
461
                 Map.SetTile(pos, null);
462
463
             catch { }
464
         }
465 }
```