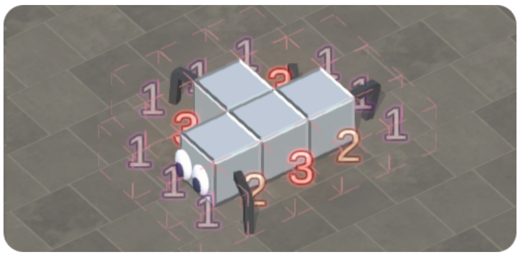
minecrawlers

Rising Star 2022

Kian Bennett

University of Portsmouth



[Concept Outline](#)

[Planning and Pre-production](#)

[Source Control](#)

[Development Timeline](#)

[Project Difficulties](#)

[QA and Optimisations](#)

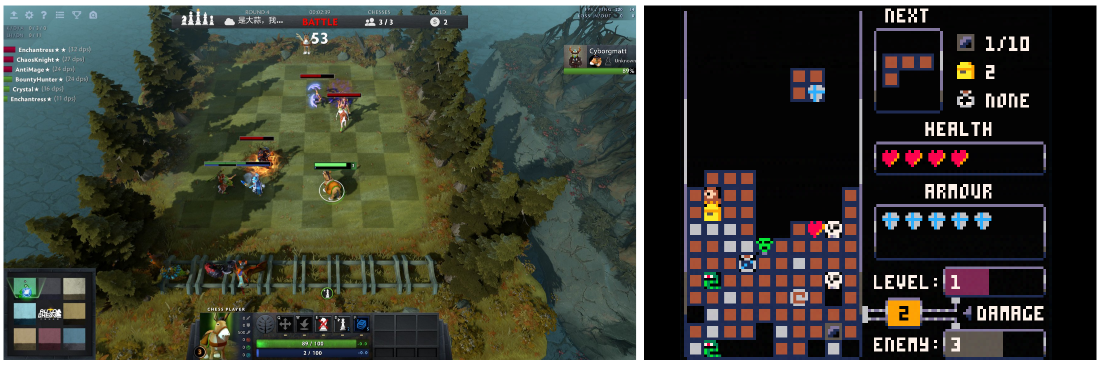
[Extending the Code Base](#)

[Final Reflection](#)

[Third Party/Premade Assets](#)

Concept Outline

I really enjoy playing games which combine elements of multiple genres or classic games. Two examples that immediately come to mind are *Dota Auto Chess*, which combines the MOBA genre with chess, and *RogueRis*, which combines the dungeon crawling of a roguelike with the grid rotation and formation of Tetris. I like that it provides players with familiar mechanics in a completely fresh setting.



The genre-fusion of Auto Chess (left) and RogueRis (right) were big inspirations for minecrawlers

I decided to take the top-down arena action gameplay of a twin-stick shooter and combine it with the mechanics of minesweeper, in which each enemy would be made up of a grid of blocks - the player would need to shoot the blocks without bombs and defuse those with bombs.

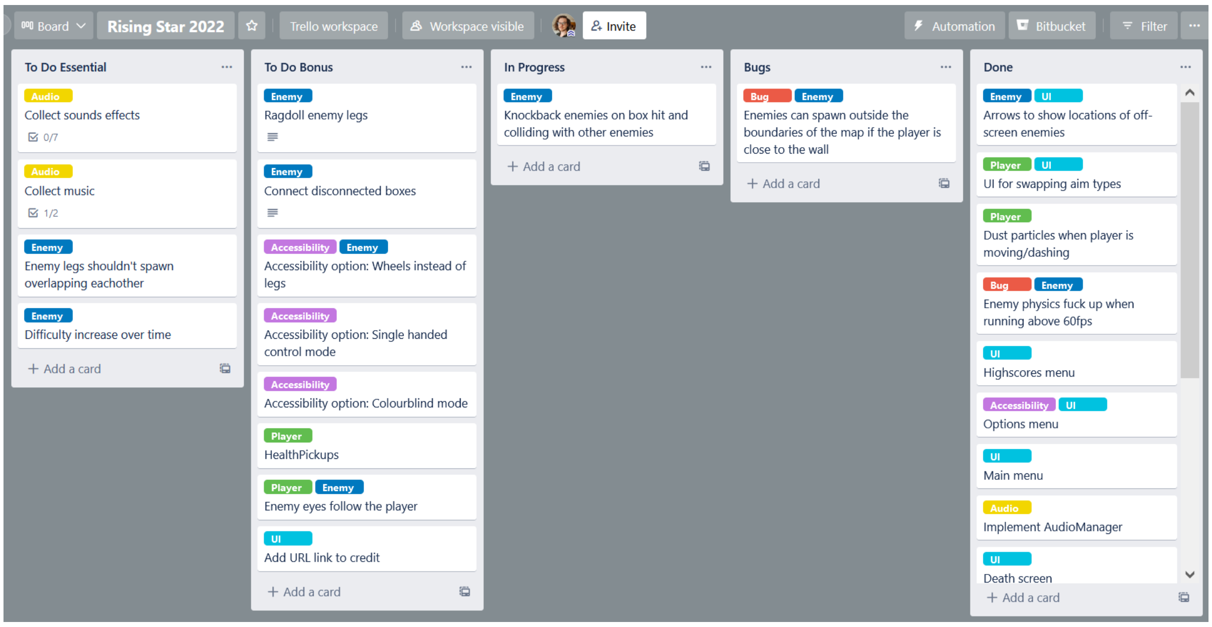
The procedural nature of the enemies would also provide an excellent opportunity to explore the procedural animation of legs using inverse kinematics, something which I have been looking to learn about for a while. I am fascinated by procedurally generated content and the ways it can enhance gameplay and replayability while reducing time spent on hand-authored content.

Planning and Pre-production

I believe that two of my strengths in game development are polished, clean visuals and responsive game-feel, so I planned my feature set with these in mind. My goal was to limit myself to one core feature - shooting the minesweeper-style enemies - and get that as polished and as focused as I could. By limiting my scope I could hopefully avoid feature creep and improve the core experience, at the expense of adding multiple mechanics. This is something I've struggled with in past projects so I was sure to keep this in mind at every stage of development.

I used a Trello board throughout the project to log my tasks and track my overall progress to make sure I didn't overrun or succumb to feature creep. By labelling my tasks I could see at a glance what areas of the project needed the most attention and what I'd be working on that day.

Something different I tried this time was splitting my To Do categories into "Essential" and "Bonus" - tasks that were essential to the core experience and those which would be nice to have only if time allowed, respectively. This helped a lot with knowing which tasks to prioritise and where development time would be best spent.



To Do Essential
- Audio — Collect sounds effects — 0/7
- Audio — Collect music — 1/2
- Enemy — Enemy legs shouldn't spawn overlapping eachother
- Enemy — Difficulty increase over time
- + Add a card

To Do Bonus
- Enemy — Ragdoll enemy legs
- Enemy — Connect disconnected boxes
- Accessibility, Enemy — Accessibility option: Wheels instead of legs
- Accessibility — Accessibility option: Single handed control mode
- Accessibility — Accessibility option: Colourblind mode
- Player — HealthPickups
- Player, Enemy — Enemy eyes follow the player
- UI — Add URL link to credit
- + Add a card

In Progress
- Enemy — Knockback enemies on box hit and colliding with other enemies
- + Add a card

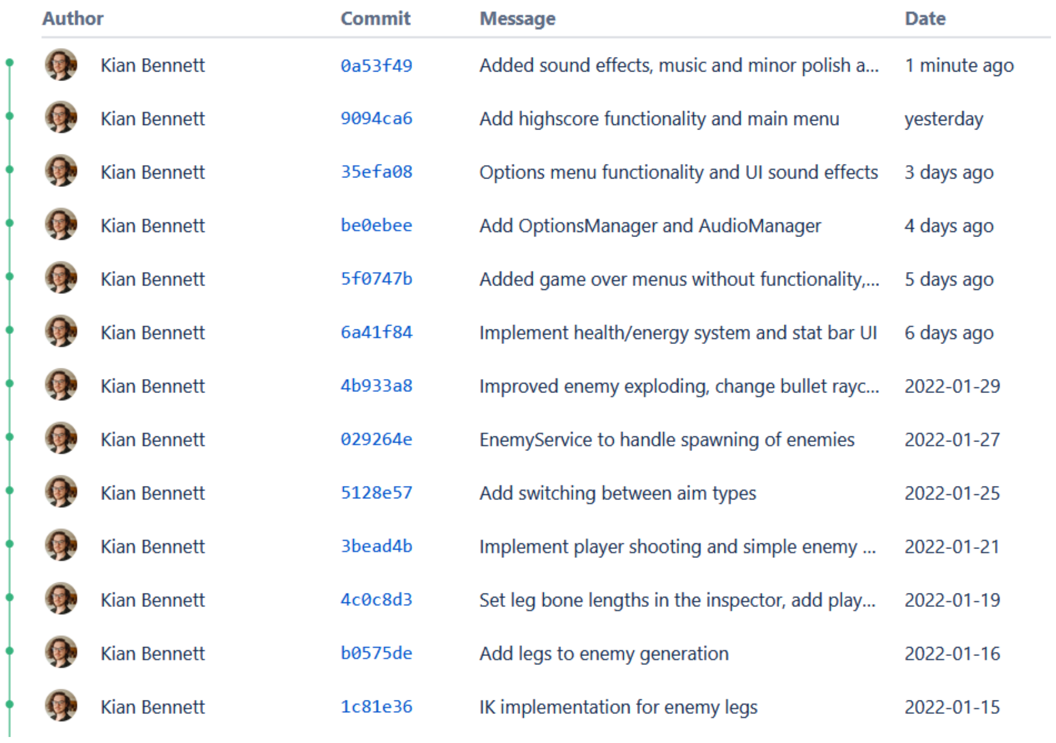
Bugs
- Bug, Enemy — Enemies can spawn outside the boundaries of the map if the player is close to the wall
- + Add a card

Done
- Enemy, UI — Arrows to show locations of off-screen enemies
- Player, UI — UI for swapping aim types
- Player — Dust particles when player is moving/dashing
- Bug, Enemy — Enemy physics fuck up when running above 60fps
- UI — Highscores menu
- Accessibility, UI — Options menu
- UI — Main menu
- Audio — Implement AudioManager
- UI — Death screen
- + Add a card

My Trello board with To Do Essential, To Do Bonus, In Progress, Bugs and Done card categories

Source Control

Early on in the project I got into the habit of committing my changes to Bitbucket every time I finished a significant chunk of work. Fortunately I never needed to roll back my changes but if I did need to restore from backup I would never lose more than a couple of days of work at any time.



Author	Commit	Message	Date
Kian Bennett	0a53f49	Added sound effects, music and minor polish a...	1 minute ago
Kian Bennett	9094ca6	Add highscore functionality and main menu	yesterday
Kian Bennett	35efa08	Options menu functionality and UI sound effects	3 days ago
Kian Bennett	be0ebee	Add OptionsManager and AudioManager	4 days ago
Kian Bennett	5f0747b	Added game over menus without functionality,...	5 days ago
Kian Bennett	6a41f84	Implement health/energy system and stat bar UI	6 days ago
Kian Bennett	4b933a8	Improved enemy exploding, change bullet rayc...	2022-01-29
Kian Bennett	029264e	EnemyService to handle spawning of enemies	2022-01-27
Kian Bennett	5128e57	Add switching between aim types	2022-01-25
Kian Bennett	3bead4b	Implement player shooting and simple enemy ...	2022-01-21
Kian Bennett	4c0c8d3	Set leg bone lengths in the inspector, add play...	2022-01-19
Kian Bennett	b0575de	Add legs to enemy generation	2022-01-16
Kian Bennett	1c81e36	IK implementation for enemy legs	2022-01-15

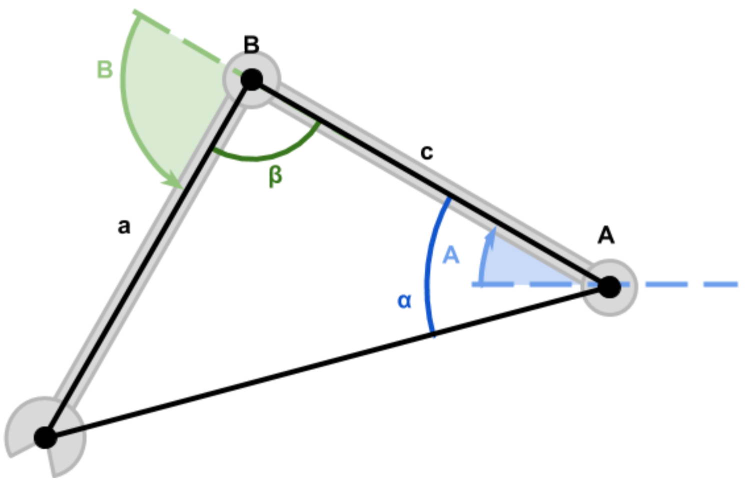
My Bitbucket commit history

Development Timeline

Procedural Leg Animations

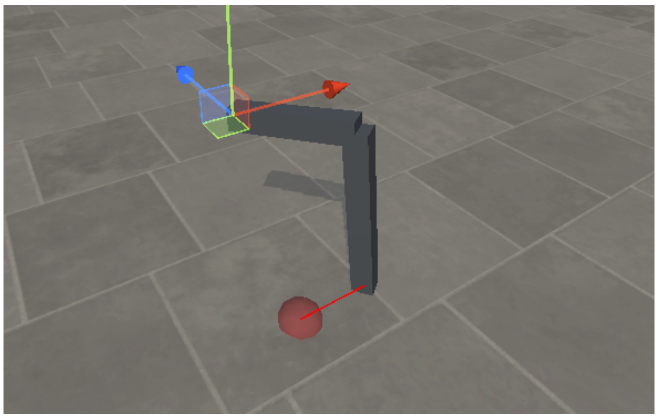
My goal with the leg animations was to have a simple 2-joint leg that would attach to some of the outward faces of the enemy's blocks, and would animate in a crawling, spider-like way. This was both a core part of the game's design and a completely new technique to me, so allocated a significant proportion of development time to it. This is something that could be achieved with simple premade animations, but as I wanted the enemies to be able to move in any direction (e.g. knockback from a bullet), and for my own edification of the technique, I chose a procedural solution.

The first step (pun intended) was to set up the inverse kinematics for the leg. Fortunately, the solution for two joints is much simpler than 3 or more joints, as there are only configurations of joint angles in which the leg can reach the target, instead of infinite. I followed Alan Zucconi's excellent guide Inverse Kinematics in 2D to get the joint angles in the XY plane, then rotate the entire leg so it pointed toward the target. This is perhaps less flexible than a true 3d solution, but it fit my needs and was more approachable as a newcomer to the field.



When only two joints are needed, the problem is resolving the angles of a triangle. As the lengths of the 'bones' a and c are fixed, we can use trigonometry to resolve the angles α and β (image credits: Alan Zucconi, 2018)

Once I had the joints correctly angled so the foot would reach the target position, it was time to set up the step animation. I loosely followed the steps outlined by Codeer in their guide Unity procedural animation tutorial (10 steps) - each leg has a target position that is fixed to the enemy body, and once the distance from the foot to this position exceeds a certain threshold, it moves towards a point ahead of the target position using Vector3.MoveTowards. To make the leg raise off the ground I calculated the percentage distance from the previous foot position to the target position and used this to get a sine wave: `LegPosition.y = Params.StepHeight * Mathf.Sin(LegMovementPercentage * Mathf.PI);`

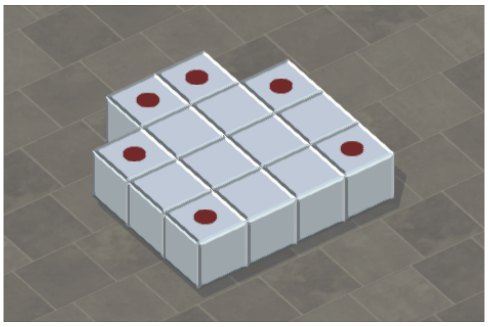

Each leg has a target position (red sphere) - once distance from this to the foot (red line) reaches a certain threshold, the leg takes a step forward

Enemy block generation

Generating the bomb layout was fairly simple - I made an *x* by *y* grid then placed *n* bombs at random points (the same as standard minesweeper. The challenge was then placing non-bomb blocks so that there could be a continuous perimeter of numbers greater than or equal to one around the enemy.

I had a few rules that would be checked for each cell to see what should be placed there:

- If a cell has a bomb, add a box
- If a cell has no bomb, and all its neighbours are nearby at least 1 bomb, then add a box
- If all of a cell's adjacent neighbours are bombs, then don't add a box (to avoid the player being unable to destroy it if it's competely blocked in by bombs)


An example bomb layout (bombs are represented by red dots)

Although this ruleset is simple, I found it generally made for bomb layouts that played well as long as the ratio between grid size and number of bombs isn't too low or high - through experimentation I found that a good number bombs is `Sqrt(GridWidth * GridHeight) * 2`.

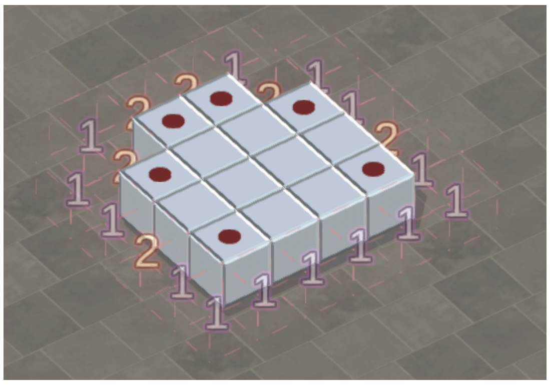
Placing eyes and legs on the enemy board

To get the possible position at which the legs and eyes could be placed, I first had to get a list of outward box faces. For each face of each box, if it didn't have a box adjacent to it in that direction it is considered an outward face. To get the face in which to place the eyes, I sorted this list first by the *z* position to get the frontmost faces, then by absolute *x* position to get the face closest to the centre.

To place the legs on the outward faces, I settled on simply adding a leg to every other outward face. This generally works well but as the faces aren't ordered by position along the perimeter this doesn't always result in an even spread. For now, ordering the outward faces is beyond the scope of the project.

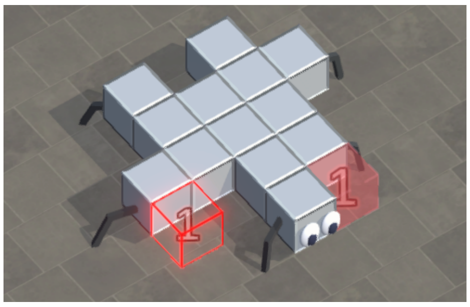
Bomb indicator blocks

After spawning all boxes, if a cell doesn't have a box but is adjacent to at least one bomb, a bomb indicator is placed in that box. When the player aims at a box, the indicators within range are highlighted to make it more obvious in the heat of battle.



Bomb indicators are placed to show how many bombs are nearby, just like traditional minesweeper

I initially struggled with the design of the bomb indicators. Since many were present they could easily present too much visual noise. It took a lot of iteration to get the right balance of readability without obscuring the enemy blocks.

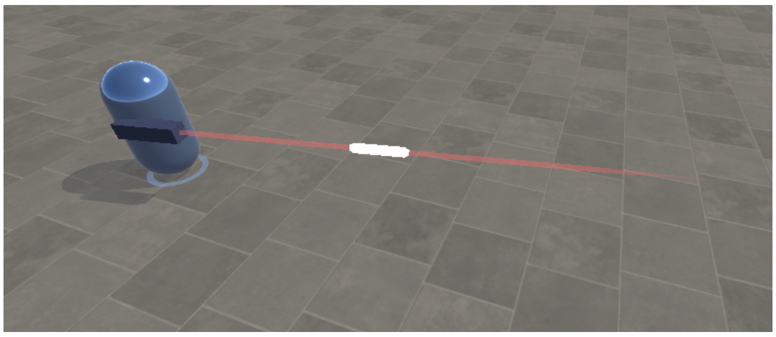


Initial prototype of the bomb indicators, although they looked alright individually, were far too visually obstructive when many were surrounding an entire enemy

Player movement and shooting

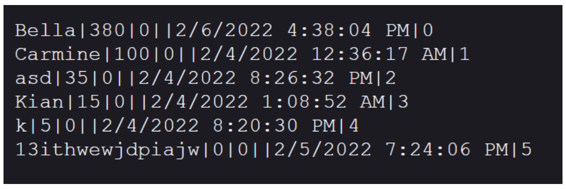
I purposefully kept the player's design as simple as possible, so I could focus on the core mechanics and hone the visuals. I added a few simple controls that could quickly add engagement and game-feel without complicating the design and taking too much development time, such as dashing, delay between switching aim types and an energy bar.

To add some 'juice' to the game and improve the game-feel, I added screen-shake, health bar shake, dust particles when dashing and sound effects when lowering your weapon, running out of energy etc.

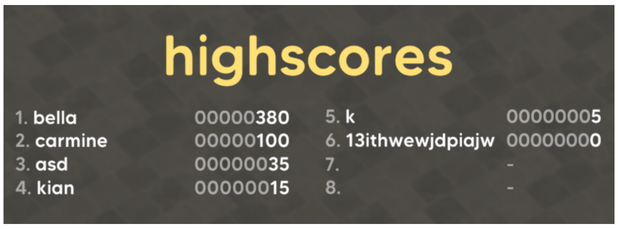

I used Unity's LineRenderers to show the bullet and aim lines - the alpha of the aim line falls of towards the end to produce a tapered effect

Online Leaderboard

I believe that adding online competition always does wonders to increase a game's replayability and longevity. To achieve this I used the free service [Dreamlo]() to upload and retrieve high scores. I used UnityWebRequest to send a request to Dreamlo's REST API, and to upload the score with username after the player dies, and display a response in a table in-game.



```
Bella|380|0||2/6/2022 4:38:04 PM|0
Carmine|100|0||2/4/2022 12:36:17 AM|1
asd|35|0||2/4/2022 8:26:32 PM|2
Kian|15|0||2/4/2022 1:08:52 AM|3
k|5|0||2/4/2022 8:20:30 PM|4
13ithwewjdpiajw|0|0||2/5/2022 7:24:06 PM|5
```



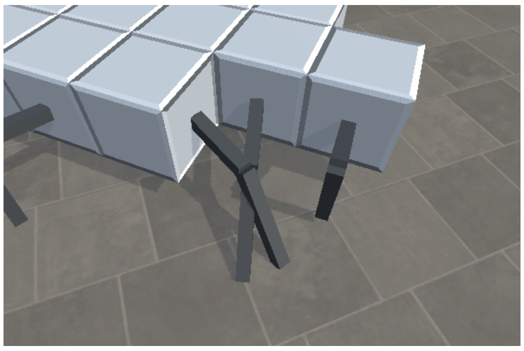
highscores

1. bella	00000380	5. k	00000005
2. carmine	00000100	6. 13ithwewjdpiajw	00000000
3. asd	00000035	7.	-
4. kian	00000015	8.	-

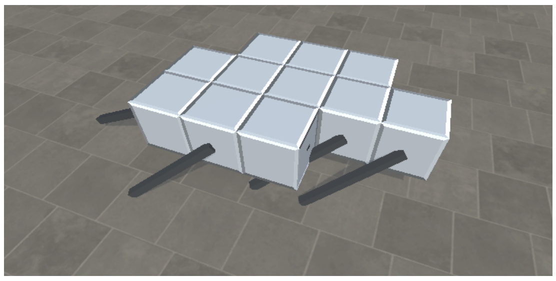
The pipe-delimited scores sent by Dreamlo by HTTP request (left), displayed in a table in-game (right)

Project Difficulties

I struggled with generating interesting enemy shapes that played well and looked natural. One of the biggest difficulties was the legs - if two faces were in a right angle to each and both had legs, they would intersect. A solution to this would be to either do a check on every neighbouring box to see if this particular configuration exists, then remove the leg, or to order the outward faces by their distance along the perimeter of the enemy. The latter would also help with even leg distribution, but is a problem whose complexity is beyond the scope of this project for now.


Legs that spawn at right-angled faces can intersect each other when moving

Another issue with the legs that plagued the entire development of the project is that beyond a certain enemy speed the legs are just unable to catch up. I have some ideas on how to investigate this (I believe it's to do with the MoveTowards call in EnemyLeg.Update), but for now limiting the enemy's movement speed will have to suffice.


Legs will get stuck catching up if the enemy moves beyond a certain speed

QA and Optimisations

The advantage of having a limited feature set and few interacting systems is that most bugs were able to appear naturally during development and could be fixed as and when they appeared. I did also leave the last day of development as a specific QA and gameplay testing period, in which I was able to fix a couple of minor bugs (mostly due to script execution order dependencies) and tune the balance of the enemies.

A handful of known bugs that emerged last minute do remain however, and I didn't have time to fix:

- Enemies spawn off the edge of the map
- Enemies can spawn inside each other

This is because multiple enemies and a fixed map size was added fairly late in development, which could have been avoided with a stronger QA consideration.

Extending the Code Base

- Board
    - Width/Height/Bomb count random instead of fixed
    - 2D array for grid instead of 1D
    - Made the _neighbours array static - as it's the same for all Board objects this saves on memory
    - Replace prefab instantiation as no longer using UI elements, moved this into Build function
    - Move nearby danger text display out of Box and into BombIndicatorBlock
    - Change the hit callback from Button OnClick to bullet collision
- Box
    - Split the bomb number funtionality into a separate class, BombIndicatorBlock
- Game
    - Nothing could be reused from the game class, this is replaced with the GameService singleton that handles scoring, game over etc.

Final Reflection

Overall I am pleased with how I managed my time on this project around working a full time job. Limiting myself to weekends and the occasional weekday evening meant I could work on this project over and extended period of time without burning myself out. Setting clearly defined goals and limiting the mechanics I planned to implement definitely helped with this, and I am pleased to have been able to stick to them without giving in to feature creep.

I am incredibly happy with how the game-feel turned out, I believe it feels tactile and responsive and each action carries a clear visual and audible consequence. However, limiting my feature set does hinder replayability as the gameplay loop can get repetitive after a while - something I tried to avoid with online leaderboards to increase the game's longevity.

If I was to spend more time on this project I would address the issues with enemy generation and add support for more interesting and diverse shapes. I would also add more game mechanics such as health pickups, player abilities, and enemies spawning in waves. I'm not particularly happy about the game having no clear ending (you keep playing until you die), so that would also be something to implement.

Third Party/Premade Assets

- MADE Tommy Soft font by MadeType (https://www.dafont.com/made-tommy-soft.font)
- Polished Concrete, Staggered texture by Architextures (https://architextures.org/textures/163)
- 512 Sound Effects (8-bit style) by SubspaceAudio (https://opengameart.org/content/512-sound-effects-8-bt-style)
- Game Music Loop - Rising by HorrorPen (https://opengameart.org/content/game-music-loop-rising)
- Heroism by Edward J. Blakeley (https://opengameart.org/content/heroism)