Development of a Bomberman - like game in JavaScript and HTML5

144175

May 19, 2016

1 Introduction

Game development volume has increased dramatically in recent years, mainly because of the introduction of different game engines and frameworks, that make the development of full-blown interactive applications much easier and more cost-effective to create and maintain. However, it was chosen for this project to develop an implementation of a classic game without the use of any frameworks or libraries whatsoever(excluding the use of an A* algorithm implementation in JavaScript, and a library to register a cheat code). To do this, a game engine had to be created, that handles mouse clicks, keystrokes, calculates collisions, processes and times animations, plays sound and displays animations in real time, while running smoothly at the same time. The animations sprites themselves were also developed from scratch. The application was developed using pure JavaScript and HTML5.

2 Application description

This section is offers a high level description of the application and the design choices taken from a user experience and user interface point of view.

2.1 Game description

The application developed here is a variation of the hugely popular 1983 Bomberman game developed by Shinichi Nakamoto. As with the original game, the main objective here is to work your way though a maze of obstacles and other players to be the last-man-standing. Player can plant bombs that can destroy obstacles (and other players) but not walls, and collect perks that improve their performance in the game. The available perks are speed, that increases the players movement speed by 50%, bomb, that lets the player plant another additional bomb (the default is one at a time), and explosion, that increases the explosive range of the bomb. Players have the option to either play by themselves against bots or against another player. There are 3 different game modes to choose from

when playing by yourself or with another human player: VS, Battle and Battle Royale. VS allows 2 players to go head-to-head against one another, while the Battle mode adds 2 additional player to the game thus increasing the total number of players to 4. Battle Royale increases the total number of players even further to 24. Another update on the historical game, is the introduction of bomb explosions in 8 directions as opposed to 4 in the original. This introduced many issues since explosions needed to be calculated based on what objects are hit along the way(if a player or objects is caught in an explosion, he/ it blocks the fire from spreading in that direction, but it can still carry on in the other 7 directions). Each player can move in 8 different directions, as opposed to 4 in the original game. To achieve this, a more sophisticated game engine had to be created with an animation for each direction that not only detects collisions, but also where that collision has taken place. There is also the opportunity to enter a cheat code to gain immortality for the player. By entering the Konami code(up, up, down, down, left, right, left, right, b, a), player 1 gains immortality and cannot be killed until the next game. By entering a different code (up, up, down, down, left, right, left, right, a, b) player 2 gain immortality if a multi-player game mode has been chosen.

3 Software Design

This section discusses the design choices made in producing the interactive web application. Firstly, the overall structure of the application and will be show, where it was chosen to use a Model-View-Controller(MVC) design paradigm to structure the application. Secondly, each of the three parts of the MVC implementation will be discussed in greater detail. The Model houses the definitions of the games model and internal behaviour. Here, all logic concerned with the application's interaction with the model is located, as well as mediator classes that facilitate loose coupling between all classes. The Controller handles all user input and sends it down to the Model logic layer for processing. The View layer is concerned with displaying the changing states of the application, as well as handling all animation specific code. While developing the application, it very quickly emerged that without structure, anything more than a simple application is too difficult to manage, and making small changes in one part of the application would have dramatic un unforeseen effects somewhere else. Therefore, it was decided that the application should be as loosely-coupled as possible, to improve upon improvements, maintainability and extensibility. To achieve these goals, it was chosen to use a Model-View-Controller approach[8]. While is possible to create our application without it, the added flexibility is of great importance when developing an application such as a video game, where small changes have to be performed constantly to improve user experience.

3.1 Model

The model is concerned with retaining the state of the application as well as updating it. It is split up into 3 sub-layers:entity, logic and service layer.

3.1.1 Entity Layer

The entity layer contains the data model of the application, where the definitions for the Game, Player, Perk, Animation, Bomb, Fire are held. The data model is manipulated by the logic layer. BY keeping the our model separate from everything else, a separation is achieved that makes change implementation easier.

3.1.2 Logic Layer

This layer allows for access and manipulation of the data model. A mediator pattern is implemented here[9] with one single object that is used to share information between all classes. This allows all entity layer classes to only store information about themselves. When information needs to be exchanged between objects, the mediator object fulfils this purpose by storing the state of all objects within itself and making calls to the model when requested by the service layer. Here collision detection,the artificial intelligence that controls non-human players, object destruction and perk spawning and collection is implemented.

Collision detection is implemented by comparing the desired movement direction with the game model to determine, if such a move is possible. To achieve this, many difficulties had to be overcome. Since the player has the option to move horizontally, vertically, as well as diagonally, it is not enough to simply stop the player, if an obstacle has been met, e.g. if the player is moving northwest and encounters a vertical obstacle, it should still be allowed to move north, as there is no obstacle there. To incorporate this into the design of the application, it was necessary to keep track of where a collision occurred, not just whether it did. By knowing which side a rectangle shaped object, it is possible to simply set the movement of that object in that direction to zero, thus allowing for the object to move in another direction.

The artificial intelligence control system was implemented based on the following assumptions:

1. the main objective of each bot is to kill all other players
2. the bot has to avoid being killed by other players and itself
3. to improve its chances of survival, as well as offensive power, a bot has to collect perks to improve its performance
4. in order to reach its opponents and collect perks, a bot has to destroy boxes

A naive implementation based on these assumption would be for a bot to initially set all of the games players as targets. Once a target is destroyed, it is removed

from the list. From there, the bot simply moves in the direction of its last target and keeps on deploying bombs when it is close enough. At the same time, a bot does not move in the direction of its target, if that would mean getting in the firing range of a bomb. If the bot is already too close to a planted bomb, it moved away from it. When encountering a destroyable object(box), the bot also deploys a bomb to destroy it and flees from the explosion spot. If, after a box has been destroyed, a perk is exposed, the perk is added to the targets list and now the bod tries to reach it as quickly as possible, thus improving its chances of survival. This solution, although very simple and omitting any tactical knowledge available to the artificial intelligence agent, still proves to be challenging to overcome. The initial implementation showed some shortcomings. Firstly, the bot would get stuck in between objects if there is something in the way of it and its target. This led to the development of a more sophisticated implementation or an AI engine. The algorithm works as follows for each bot when initialising a game:

1. Shuffle the list of all players randomly and put them in a list of targets
2. For each step, perform A star search to find the shortest path to the last target.
   (a) Rocks have a weight of 0, meaning that there is no path
   (b) Boxes have a weight of 5
   (c) Plain grass tiles have a weight of 1
3. if a path is found, the player moves along it for 1 tile
4. if the bot encounters a player or box in its next move, a bomb is planted if there is a bomb in the range of the player, A star is performed again to find a safe escape position. The bot moves there and waits for the bomb to explode before returning to step 2

3.1.3 Service Layer

Here different services are exposed to the Controller and View. The service layer can access the logic layer, in order to update the game state with changing inputs, as well as to retrieve the state to the presentation(view) layer.

3.2 View

The view layer handles the presentation of the underlying data - model to the screen. Here, animation rules are defined based on the underlying model, e.g. how users players "move" when a button is pressed, how bomb explosion are animated or how damage is dealt to players and destroyable objects.

3.2.1 Player Animation

Player animation is done using sprite sheets as described by M. Weeks [14]. The advantage of using one image file for each movement direction, with all possible animations on it is that it reduces the performance overhead of having to load/store many different files for each animation. Each player has an animation sprite sheet for each direction of movement. Based on the direction of travel, the appropriate animation is fetched and looped though, thus creating the illusion of movement. The speed of the animation loop is adjusted based on the speed of the player i.e. the larger the speed of a the player is, the quicker the animation loops through. Figure 1 shows the animation of player 1. When playing a multiplayer game, player 2 and all bots have a different animation as shown in figure 2. Initially, the animation sprites were created using solid colours

Figure 1: Player 1 graphics sprite sheet with AA

and black outlines but this gave the impression that the images are blurry and jagged. To tackle this and remove the aliasing artefacts, more colours and shades were introduced to give the images more depth, colour and smoothness. This resulted in a more natural look for the animations.

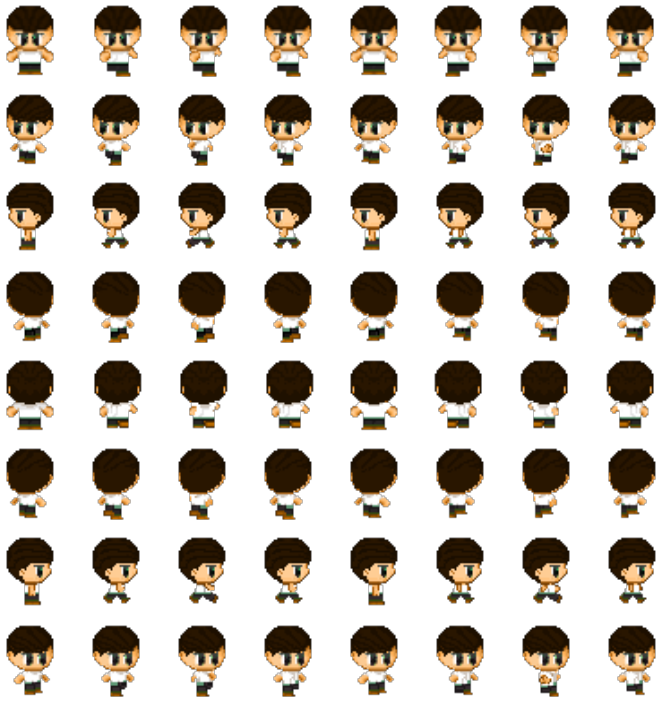

Figure 2: Player 2/bot graphics sprite sheet with AA

3.2.2 Explosion Animation

Explosions are animated in a similar way to players. Once a bomb is planted by a player, a timer is started representing the time that it will take for the bomb to explode. This is shown in the animation as a countdown clock in seconds that starts at 3 and ends with 1. Once the timer reaches 0, a different animation is created, displaying the explosion. The bomb animation sprite sheet can be seen in figure 3.

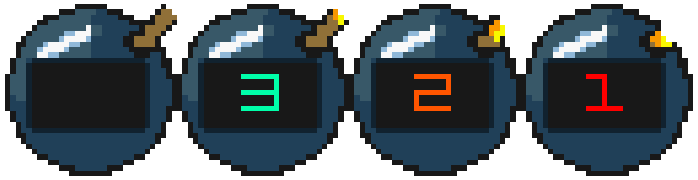

Figure 3: Bomb animation graphics with AA

An explosion is a collection of fire sprite sheets(figure 4). When adding more fire objects to the explosion sequentially, starting from the location of the bomb that set it off outwards, a nice animation of an explosion is obtained.

Figure 4: Explosion animation sprite sheet with AA

3.2.3 Menus

The menu interface is implemented by using simple animations with 5 different views: game view, main views 1 and 2, help menu and game over menu. These are displayed and hidden based on user actions.

3.3 Controller

The controller layer is responsible for tracking changes in user input by using event listeners. The listeners used are "keyup" and "keydown" listener. "Keydown" listeners are used to update the state of the controller and indicate that the player is moving in a particular direction. After each key press, the service layer is called, that is responsible for updating the values of pressed keys. The "keyup" listener is used for planting bombs, just like with the other listener, once this listener is fired, the appropriate bomb is planted, if the corresponding button is pressed.

4 Conclusion and further improvements

This project brings together many aspects of multimedia including mouse and keyboard input, graphics animation and sound, to produce a entertaining game based on the concept of the classic Bomberman game. To achieve this, a custom game engine had to be created from scratch, that implements collision detection, animation, and smooth gameplay, as well as an artificial intelligence implementation for bot players. All animation graphics were created by hand using Piskel[6]. All game music and special effects for perks and explosions were retried SoundCloud. However, there are aspects of the game that could be improved upon. Firstly, the game should offer the option to mute the sound, if necessary, as that would give the player the opportunity to choose. Also, even though the game menus are interactive, the lack any kind of animation of sound effects, that could be added to further improve the appeal of the application. Additionally, the AI engine could be further improved to actively seek out perks and collect them, to improve performance. Given enough computational power, it would be possible to introduce an even more sophisticated AI implementation using a min-max search algorithm.

References

[1] Louis Acresti. Cheet. http://lou.wtf/cheet.js/, 2016. [Online; accessed 06-May-2016].

[2] alex. http://stackoverflow.com/questions/9880279/how-do-i-add-a-simple-onclick-event-handler-to-a-canvas-element, 2012. [Online; accessed 29-April-2016].

[3] Bomb explosion sound. http://soundbible.com/tags-explosion.html, 2014. [Online; accessed 07-April-2016].

[4] CoolAJ86. http://stackoverflow.com/questions/2450954/how-to-randomize-shuffle-a-javascript-array, 2015. [Online; accessed 09-April-2016].

[5] CreativeJS. http://creativejs.com/resources/requestanimationframe/, 2014. [Online; accessed 22-April-2016].

[6] Julian Descotes. piskel. http://www.piskelapp.com/, 2016. [Online; accessed 02-May-2016].

[7] Brian Grinstead. http://www.briangrinstead.com/blog/astar-search-algorithm-in-javascript, 2015. [Online; accessed 29-April-2016].

[8] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[9] Tommi Mikkonen. Formalizing design patterns. In *Proceedings of the 20th international conference on Software engineering*, pages 115–124. IEEE Computer Society, 1998.

[10] Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame, 2016. [Online; accessed 12-April-2016].

[11] Spritesheets. http://gamedevelopment.tutsplus.com/tutorials/an-introduction-to-spritesheet-animation--gamedev-13099, 2016. [Online; accessed 15-April-2016].

[12] Tsugumo. http://gas13.ru/v3/tutorials/sywtbapa_almighty_grass_tile.php, 2016. [Online; accessed 07-April-2016].

[13] W3 schools. http://www.w3schools.com/games/game_controllers.as/, 2016. [Online; accessed 25-April-2016].

[14] Michael Weeks. Creating a web-based, 2-d action game in javascript with html5. In *Proceedings of the 2014 ACM Southeast Regional Conference*, page 7. ACM, 2014.