Eryk Pecyna
Jay Chandra
Kodi Obika

BomberCaml: CS51 Final Project

Introduction:

As our alternative final project, we decided to create the Bomberman video game (explained in more detail below) with OCaml. Similarly to how most video game programming endeavors rely on the object-oriented paradigm, our implementation mainly involved the creation of various classes (from individual parts of the map, like walls and boxes, to characters and enhancements, like players/enemies and extra bombs/fire) and modules ("game.ml" , "gameobj.ml" , etc.) to hold each of our classes and functions.

What Is Bomberman?:

Bomberman is a strategy-based video game in which the player, who initially spawns with bombs that self-destruct a few seconds after being dropped, ultimately attempts to eliminate the enemies that initially spawn with the map. One accomplishes this by dropping bombs strategically throughout the map in an attempt to capture the enemy within the blast radius, in which case the enemy is eliminated. Initially, the player can drop only one bomb at a time, which means that each player can have one bomb on the map at a single point in time, and the radius of the bomb explosion is one. The map itself is made of boxes and walls. A wall surrounds the entirety of the map and there are single wall pieces located in every other row and every other column. Boxes are then interspersed between the walls and the two cells around the character are always left empty. Please see Figure 1 for further clarification on this map layout. Bombs can destroy boxes. However, bombs cannot destroy walls. Additionally, there are powerups hidden in some of the boxes meant to aid the player in their mission. These powerups can be accessed by capturing their associated boxes within the blast radius of a dropped bomb and then walking over them. While the original game incorporates a variety of enemies and powerups, ours has one type of enemy, whose only ability is eliminating the player by contact, and two types of powerups - one being the ability to drop extra bombs simultaneously, and the other increasing the blast radius of a player's dropped bombs by one.

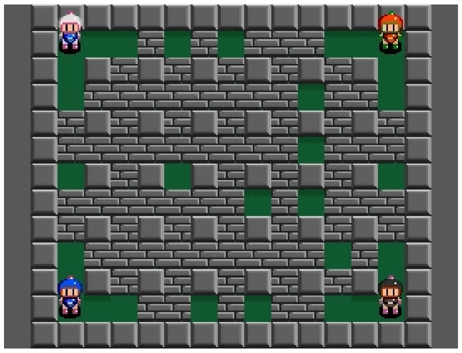

Figure 1. Map layout with boxes represented as brick squares and indestructible walls as solid gray squares.

Our Implementation:
As briefly mentioned before, there are a few ways in which our implementation differs from the original video game, but the core structure of the game was ultimately retained in our implementation. The way in which our implementation generally works is outlined below (see Figure 2 for a visual representation). Our game is an implementation of the "story mode" of Bomberman because there is only one player with multiple computer-controlled enemies.

Player: The player is represented by a green circle that spawns at the bottom left corner of the map. It can be moved by pressing the 'W' (up), 'A' (left), 'S' (down), and 'D' (right) keys. The player holds an infinite amount of bombs but can only drop a finite amount at a time. Additionally, one cannot walk through walls or boxes. Upon contact with an enemy, the game ends and the player is eliminated.

Bombs: A bomb is dropped whenever the player presses the spacebar (while stationary). After a few seconds, the bomb explodes, and any boxes, players, or enemies that are caught in or walk into its blast radius (represented by the yellow squares that temporarily appear on the screen) are eliminated. The initial blast radius of a bomb is one, and this increases by one with each extra radius powerup that the player collects. Also, when the player acquires multiple bombs, they are able to drop one bomb that triggers the explosion of another bomb that may have only been recently dropped.

Enemies: As mentioned previously, in our game, there will be a single type of enemy that can be destroyed with a single explosion from the bomb. The enemy is represented by a red circle that will spawn randomly in cells that are occupied by the player or the walls. The enemies will be able to move over boxes and will be able to "kill" the player if it touches the player.

Map: The map is mostly implemented in the same way as the original Bomberman game. The destructible components of the map are represented by brown squares that represent boxes. The indestructible components are walls that are represented by grey squares. While the positions of the walls are the same between our implementation and the game's implementation, the position of the boxes are created slightly differently. We have decided to make the chance that a box will be placed in a location that is not already taken up by a wall to be 0.6. While this is lower than what the original Bomberman game has, this was a game design choice given the lower complexity of our implementation. This chance can be easily altered in the code.

Powerups: As mentioned previously, the only powerups that are implemented in this game are the extra bomb and extra radius powerup. The extra bomb is shown by a black circle on a yellow square. The extra radius powerup is shown by a red circle on a yellow square. The chance that each powerup will appear when a box is destroyed by a bomb is currently implemented as 1/10 per powerup. However, like the chance of a box appearing in the generation of a map, this is easily altered in the code.

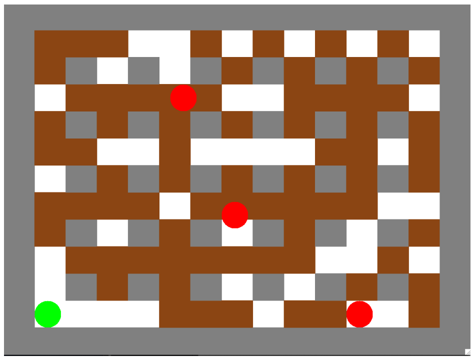

Figure 2: Static visual representation of the game map with three enemies shown as three red circles and a player as a green circle.

Overall Code Structure:

We used the following files to implement the game:

- game.ml: Actually opens the window, draws the graphics, starts the game loop, and handles the game over screen. The game can be run after compiling all of the files and typing ./game.byte into the terminal.
- gameobj.ml: Holds all of the graphical objects described above (boxes, walls, players, enemies, etc.). Every object inherits from the virtual drawable class, players and enemies inherit from a moveable object, and the firepower and extrabomb classes inherit from powerup class.
- gamestate.ml: Holds the implementation of the functions used to create and update the state of the game. Implements the interactions between the different objects in the game.
- util.ml: Holds objects that do not necessarily have a place in "gameobj.ml" but provide auxiliary functionality.

We decided that this structure, wherein our files exist as parts of an abstraction hierarchy (with game.ml holding the actual game, gameobj.ml holding its objects, gamestate.ml modeling the interaction between the objects, and util.ml holding functions that aid in the implementation of the game but aren't necessarily an important part of the game structure) was the most comprehensible way for us to essentially carve our software at its joints.

Design Deliberations:

Throughout the process of implementing the game, there were a number of tradeoffs and overall design decisions that we had to make. For instance, we initially were having issues with the controls of the player object because of OCaml's limited keyboard events that can be found in the Graphics module. The main issue stemmed from the fact that there was no key-up detection. However, we were able to find a fix provided by a flush_kp function found in an online computer science forum (Stack Overflow). Although this does not allow for the functionality of holding down a key to move continuously in non-Mac OS environments, it was the simplest patch to allow us to take advantage of OCaml's limited set of events.

The player object itself is implemented in gameobj.ml. It inherits from the moveable class, because it was necessary to make enemies, which would have similar functionality to the player with only a few differences. The player was special in that it would hold information about the number of bombs that it can drop at once and the radius of the bombs that it can drop.

Moveable objects, namely the player and enemies, have a significantly different structure from all the other game objects. Whereas boxes, walls, bombs, and exploding tiles are all stored in an array of the game map, the player and enemies are stored separately from this array. We knew that parts of the map would only ever stay within their square of the grid that makes up the map, therefore we were able to implement the map as an array of objects that existed within a small square. If we were to assign an enemy to any of these squares, we would lose the object (e.g. a box) that was previously stored there. Although we could have done this without losing information by using a more complicated structure for the components of the array, it would have been harder to animate the enemies and players movement through the grid. So we instead decided to store the moveable objects separate from the map itself, allowing them to move freely of the coordinate system we created for the other map. We still use that coordinate system to implement the collisions between all objects, but the implementation of the moveable objects is completely disjoint from the game Array and could exist without it.

Also, we decided to implement moveable objects in a way that would be less jarring to the viewer. For this reason, we created an animate method that would have enemies and player to the new space in a three step process. This would make it such that the player and the enemies do not "teleport" when moving. The implementation of this was made with the move and animate functions. The move function was called in the gameState file. When the player or enemy was asked to move, it would be sent the new coordinates of the move and the move function would update values like how many pixels one-third of a move would be and the direction of the move. Meanwhile, in the draw function, which is being called approximately every 0.05 seconds, the animate function is called, moving the player object by one-third of the total move. Of course, this is not exactly 0.05 seconds due to the time required to process all of the other function calls in drawstate. This implies that the game is running at less than 20 frames per second. The animation makes it such that the move button cannot be "spammed," meaning that when the move animation is running, the person playing the game cannot tell the player to move by inputting a WASD key press.

After playing the game and testing it, we decided to override the first step of the animation of players in order to make the controls more responsive to human input and make the player's movement seem snappier. Meanwhile, we left the enemy class with all of the steps of animation in order to make its animation smoother. We figured this tradeoff made sense because in most video games, the player moves slightly faster than the enemies. The speed of the enemies can be increased by overriding the animation in a similar way if needed in the future.

Additionally, we decided to implement the storage of enemies in a list rather than a set for the sake of simplicity (dealing with objects as a type within a set proved to be a bit difficult). Even though there are efficiency benefits associated with using a set (mainly in that access to its elements have constant time complexity rather than the linear time complexity that list access has), we figured that since there are only three enemies, the difference wouldn't be too consequential.

For the map, we chose to implement it as a class itself in order to more clearly coordinate the interactions between the different objects in the game. This centralized hub for updating the game map divides the code clearly between the game objects and the updating of the map. If there was no map object, we would have to implement game objects such that they communicate with each other seamlessly. This would require lengthy match statements and extensive methods in each game class that would only be ideal in situations where the number objects is very low. With so many moving pieces we believed that a map class would be most ideal.

For the powerups, we decided to make a larger powerup object that is inherited from all of the subclasses, which are all of the actual powerups. The reason we did that is because all powerups are handled in similar ways in the state object. With more objects, the length of the match statement would increase and there would be significantly more repeated code in the explode method. If the number of powerups does increase to a very large amount, it might be interesting to think about a single power up object with more arguments including the color of the object and the id of the object. The id would distinguish between the different types of objects and the color would be inputted into the draw method. For our implementation, the subclass-superclass seemed like the better choice because there were only two powerups, but for a game with five objects or more, this approach might be better.

The enemy movement was designed to be random except for the fact that we didn't want the enemy to move back to where it had just come from. The purpose of this was to make sure that the enemy would not stay at the top of the map while the player just collected powerups in the other corner. The other deviation from randomness stems from the design of the map. Because, in the map, there are hallways and alternating walls, we have the enemy alternating between moving in the same direction that it had just moved in and taking a random direction that would not move it back to where it had just come. This makes it such that the enemies don't constantly try to move into walls when the enemy has a wall below and above it. Our solution to the problem means that there are moments that the enemy stays on a wall for only a short amount of time before it moves. An alternate implementation of the computer controlled enemy could be checking for walls adjacent to the position of each enemy and having it move to a position that doesn't have a wall. However, as we found in testing, this implementation actually decreases the fully random motion of the enemy because the enemy pausing and moving adds another piece of variance for the player to account for.

Visual Representation:
See the following videos for a visual representation of how the game works:
https://www.youtube.com/watch?v=gaPPV-W52oU& feature=youtu.be
The first minute and ten seconds demonstrates how the game is won, with 0:16 - 0:29 showing what the extra bomb powerup looks like and how it works, and the second half of the video demonstrates one way in which the game is lost (coming into contact with an enemy), with 1:45 - 1:50 showing what the extra fire powerup looks like and how it works.
https://www.youtube.com/watch?v=8r3b3mTwin8& feature=youtu.be
This second video demonstrates the other way in which the game is lost (getting caught in the blast radius of your bomb) and further illustrates how the enhancements work.
https://youtu.be/Zp1zKCZbioI
This video demonstrates the effect different constants have on the game.

"Future PSET" Add-Ons
Given the instructions of treating this project as a "future PSET," here are some additional implementations that students could make to our project. Firstly, the enemy movement can be improved upon in the moveEnemies function from the state class in conjunction with the enemy class. This implementation could potentially include a detection of the walls around the enemy and prevent the enemy from pausing on the border due to trying to move into a wall. Another add-on could be more powerups including the skate powerup that increases the movement speed of the player. Right now the animation takes 2 steps (technically three but one is skipped) for the player, but this can be changed by a future alternate implementation of the animate function in the moveable class. An easier powerup to implement could be the 1-up powerup, which just adds a life to a player.

Post-Meeting Modifications:
The final project meeting with Professor Shieber and the course heads opened our eyes to a few design-related issues and parts of our implementation that would not necessarily work as well as the project scales up.

For instance, there were a number of places in our code where we utilized "magic numbers," or constant numbers with no clear explanation as to where they came from. We fixed this by making a configuration file (called "config.ml") with all of the constants that we used, making clear what these constants are supposed to represent. This way, if we ever decide to change these constants, there is an established place for doing so, and if there are multiple places in which a constant is used, it is much easier to make that change. Through making the config.ml file and the separation of the "magic numbers," we actually were able to easily scale the map without having to specifically calculate the dimensions of the screen every time.

There were also a number of opportunities in our code to get rid of side effects that augmented the impurity of our code. For example, when creating the list of enemies, instead of keeping the for loop that we initially used, we utilized the init function in OCaml's List module. Changes such as these definitely helped make our code not only look much more elegant but also more side-effect free.

Post-Meeting Deliberations:

A major fix that we would want to implement in the future is the abstraction barrier of the movement in pixels. This would entail altering the generateMap function such that it takes the size of the each cell in the map as an input. Every function that deals with the actual pixel movement of the objects will instead make a call to the "convertToPixel" function that will convert the movement input of (1,0) to (80,0) if the length of each cell in the game is 80, and then the object would move by that much. The purpose of this, as expressed by Professor Shieber, would be to not have to deal with the integer division that we had to do in the generateMap function (which could easily lead to rounding errors).

Another concept that we could analyze the tradeoffs of is the implementation of objects in a way such that all of the objects know the locations of other objects due to a mutable value that represents the map array in each object. This would make our state object much simpler but would make the game objects much more complicated. This would also entail the use of event listeners, which would facilitate the interaction between the user and the machine via keyboard events, as well as the interaction between the various objects utilized in our code.

In general, there were a few opportunities in our code for us to utilize the edict of abstraction more, which would especially help if we decided to scale the project up. For instance, a way in which this notion could be implemented is in events that involve ticking. We have two functions that deal with "tick" events (events that rely on a timer) - one for bombs and one for exploding. However, we could instead have a global tick and then keep track of at what point during the ticking that each object has to do what it needs to do, which would definitely help if we decided to have more objects rely on timing (for example, having powerups only be active for a limited amount of time). However, because these are the only two objects that would rely on timing in this game, it made more sense to use this simpler implementation rather than add ticking to all objects. Even if we were to add ticking to all objects we would still, in some way, have to independently track each bomb object to differentiate between separate timings for explosions. While this might be possible to implement in OCaml, we could not figure out how to allow the game objects stored in the gameState file to have access to anything inside of the gameState file. Listeners seem like they would be a fix, but they require events that involve inter-thread communication that we do not know how to implement. Other languages with more robust object-oriented features such as Java allow for more interaction between objects in the form of passing certain methods as arguments, but we could not figure out how to do this in OCaml.

During the presentation, the length of some of our methods was discussed, specifically movePlayer and moveEnemies in gameState. Although the methods are lengthy, over half of each are helper functions. In the code's current state, it would be difficult to move these functions away from the methods that they are used in. However, if we restructured our code and built it along the guidelines suggested by Professor Shieber, we would most likely move a lot of this functionality to the game objects which should result in a cleaner gameState.

Conclusion:

In essence, in creating Bomberman with OCaml, we discovered that all of the edicts and programming principles that we became acquainted with throughout the semester were imperative (no pun intended) to writing not only working code, but also clean, elegant, and efficient code that we could truly be proud of. We also became more aware of certain important concepts when working on the same code including branches in git and the idea that the master branch should always have a working version of the game. The branches helped us separate work such that our time working on the project could be maximized. One person would be working on the powerup objects while another would be working on how the map changes when the powerup is picked up, for example. Overall, we spent roughly 52 person-hours on the implementation of BomberCaml and we are happy with the working game that we have created.