Boolean Battles

Declaration:

We hereby declare that the work contained in the following document is entirely the work of the undersigned. Those signed below have agreed the following split of marks for the project:

Alex Jones	25%
Charles Crawford	25%
Jack Williams	25%
Liam Wardle	25%

Table of Contents

Abstract

The aim of this project was to build a piece of software which could engage and teach young people about a topic within computer science, based upon the BBC Micro hailed for its robustness and ease of use to teach computer science. From this we have created a working prototype of a game which teaches Boolean logic through appealing on-screen graphics, thought provoking challenge progression and an absence of algebra. This game demonstrates a possible aid for the teaching and learning of the complex topic of Boolean logic symbols and their conjunctive use. Because the game is implemented in an RPG 2D graphics style, with attractive block colours and animation, the game should be engaging and act as a teaching aid for young people. With the symbols used to replace algebraic symbols (i.e. Fire instead of X), the users focus will be moved towards the actual logic symbols and their use, rather than being weighed down by algebraic symbols. The use of the symbols, colour and level design all aggregate to reduce the games likeness to actual classical studying – e.g. following a set of logic questions in a text book. Overall we believe the project was a great success – as a team we have improved our working knowledge of graphical libraries such as SDL, building logic machines and combining a large number of files to make one large working project.

Introduction

Context

During the 1980s BBC introduced a microcomputer named the ‘BBC Micro’ which aimed to introduce children and young people to technology. The micro was introduced to a generation of potential future programmers, and was notable for its robustness, ease of use and quality of software and hardware. This project aimed to build upon this idea and produce a piece of software which would introduce young people to an aspect of computer science in a friendly, approachable and fun manner. The key targets with the software would be making it engaging and educational, providing some aspect of challenge without being so challenging that this draws away from the educational side of it.

Initial Ideas

Initially a list of game genres was consulted which was used to brainstorm ideas which would allow us to build a minimum viable product. Our most highly valued initial ideas were games based upon the bejeweled game mechanic, RPG-based games, driving simulator games and puzzle-solving games. Each idea was infused with an aspect of teaching computer science, the most common themes we had were users entering code, solving Boolean problems and matching code segments together to complete the code. We initially chose three main game ideas to work with and refine until we had decided which would be most suitable. We based our suitability upon feasibility, challenge potential and learning aspects.

Our first initial idea was a Boolean RPG game, where Boolean puzzles were solved to attack an enemy. This provided an easy way of tackling the challenge aspect, by making the puzzles progressively harder, a good basis for the learning aspect of the game as logic is directly taught, and feasibility as RPG graphics are often simple, flat and 2D.

Our second initial idea was a 2D robot platform-style game where a robot is 'missing code' and must be filled in to advance the robots movement and mechanical aspects. The challenge aspect would be tackled by starting with the very basics of coding, i.e. filling out a variable, and ending with complex parts i.e. dealing with nested for loops, this would also deal with the learning aspect as the user is directly learning and writing code. The game itself would be quite simple to design as the robot itself would be easy to animate as it would be a 2D platform style game – which also meant graphics would be simple.

Our final initial idea was a bejeweled-style puzzle game, where the user would match code chunks together to complete the code. The challenge aspect would be tackled by having various tiers of levels, each harder than the last. Learning would be directly through understanding how pieces of code work together – however the user would not ever be writing any code themselves. The graphics would be simple as they are just sliding and connecting simple blocks together.
We rejected several ideas from our initial three, mainly due to difficulty of implementation, lack of engagement, and lack of progression and challenge in the game.

From these three ideas we settled on the final idea of the RPG style Boolean logic game. We rejected the 2D-platform style robot game on the basis that to implement a progression in the game we would have to pre-fabricate a vast quantity of code to be filled in, which would have

taken too much time. We also rejected the bejeweled idea on the same premise, and that the game would ultimately be tedious once past the first few levels, as it lacked what the bejeweled games succeeded with - simplicity.

Chosen Idea

After our initial brainstorming of ideas we decided that our chosen area of computer science to base the project on is that of Boolean logic by implementing our game as an RPG-style logic puzzle solving game. Boolean logic allows simple binary statements to be strung together to form logic circuits which are used as the basis of computer processors. Within this area we focused on the logic symbols and their combined use, which can be daunting and confusing to a person who has just begun studying computer architecture and logic circuits.

The design of the game would be based upon fighting against enemies which have a set health, which an attack will reduce their health by one, and a failed attack would reduce your health by one. At the start of each enemy encounter a question containing the enemies weakness is presented at the top, in the form of a Boolean logic equation, but instead of algebraic values (i.e. A, B), ‘types’ of attack of used. For example a question may show “FIRE V WATER”, indicating that a fire or water attack will successfully reduce this enemy’s health. At the bottom of the screen will be selectable types of attack shown as balls, i.e. a fireball. Selecting a fireball and clicking the attack button will mean a fireball will be shot at the enemy. As the game progresses from level one, through to level ten, the questions get progressively harder. After each level the player will be transported to a map which shows their game progression.

This would be implemented in C using the SDL library to display animated graphics, such as the fireballs animated attacks, and static graphics, such as the background and the enemy/hero characters. Graphics would be designed from the ground up using GIMP2 software and supplemented with non-copyrighted vector images where appropriate. Initially the team would split into groups of two and work on separate parts, such as logic, graphics, map design, and then work together to integrate all of the separate sections.

User Guide

Boolean Battles is designed to have an easy to use interface, with instructions being provided during the first level. We found that a tutorial during this level was the easiest way to simultaneously explain the game and perform the first level. The instructions are provided in the form of enemy dialogues.

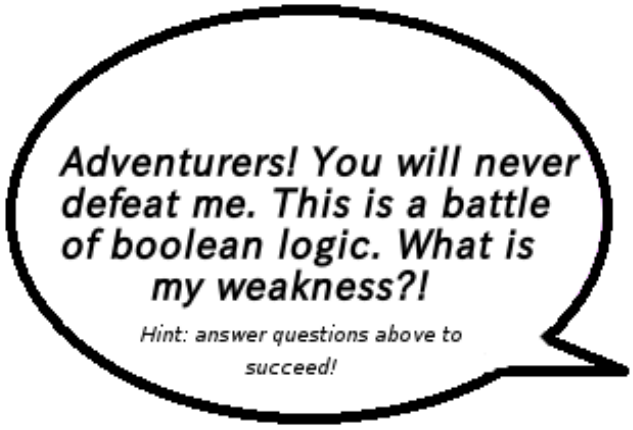

Figure I: Example of a dialogue used as part of the tutorial

Players are tasked with defeating a series of enemies which stand in their way. The player has an initial health of three, and is reduced by one upon input of a wrong answer. The enemy health corresponds to the number of solutions available. All solutions must be given in order to defeat each enemy.

Attacks are in the form of bolts of elements:

Fire 

Ice 

Lightning 

A neutral attack is shown with no colour, which results when none of the options of elements selected is chosen as a solution.

The questions relate to Boolean Logic, where the weakness of the monster is governed by the equations. For example:

Figure II: Example of a question which corresponds to the weakness of an enemy. In this example, the weakness can only be both Fire and Ice.

The answer can be selected by toggling element buttons on or off. When ready to attack, the sword can be pressed which commences battle. If the answer is correct, the hero attacks in the form of animated bolts of the selected elements. However, upon entry of a wrong answer, they are rebounded upon the hero, taking damage in the process. If player health reaches zero, the game is ended.

Upon defeating an enemy, the hero is guided onto the next level where a different enemy resides. With an integrated map version, upon defeating an enemy, the user would be guided back to the map, where the next level becomes unlocked and can be selected. Previous levels can also be repeated with this version.

Coding Standards and House Guidelines

The following list outlines the standards the group adhered to; as to make the code as readable as possible, whilst also allowing for easier and more efficient integration of separate pieces of code from members of the group.

1. Braces in line with initial statements.
2. Minimal use of Global Variables.
3. Consistent indentation (4 spaces).
4. Comments only used where absolutely necessary.
5. Minimal use of Magic Numbers.
6. Meaningful identifiers.
7. Lower-case variable names for initial word's first letter.
8. Non-repetitive code
9. Short functions.
10. All function prototypes declared in the relevant header.

These standards were adhered to as much as possible, however certain features such as comments tended to have differences in breadth between members. The overall consensus was to provide enough detail in comment only to convey understanding to another developer who may need to adapt or change sections of code.

Communication Methods

Various methods were employed to ensure each member was kept up to date with the latest development. Upon initial meeting, we ensured phone numbers were exchanged, and this would provide the base of our communication.

Subsequent applications were utilised:

1. WhatsApp Group chat
2. Facebook Group
3. Google Drive
4. Waffle.io
5. GitHub

WhatsApp and Facebook provided us with the means to schedule weekly team meetings, as to update and inform every one of the current progress and development, and also to agree upon task assignment and fixing of software issues.

Task assignment was digitally assigned using the web application Waffle.io, which enabled us to track individual assignments, features yet to be added, as well as completed tasks. This method of issue/task assignment gave us a simple yet effective form of project management which we used throughout the planning, design and implementation stage.

GitHub was used for creation of the repository, allowing for effective sharing of all work produced by the team, and also ensured we had back-ups of the current implementation of our software.

This was an essential requirement for ensuring the group had the latest versions of Boolean Battles, whilst also providing another means of tracking changes to individual pieces of code.

Google Drive provided a means of sharing material between specific members of the team, for example during production of the presentations.

Project Plan

The group came to the agreement that a single Project Manager was not required, and therefore each member had a significant degree of responsibility for taking on and balancing their workloads. The ease of use of the project management tools we were using meant that the group were able to track latest targets and deadlines. This proved to be a reasonably effective method for a project of this size; however changes may have had to be made with larger-scale projects with greater time-scales.

Weekly team meetings were scheduled to take place every Monday, in which each member updated the team on their developments. We ensured everyone understood their targets and any difficulties were reported. Additional task assignments and scheduling of pair programming also took place regularly, and we made sure the pairs were alternated, as to maximise creativity and group interaction/communication. The team shared the responsibility of writing minutes, as to log the outcomes of meetings.

Upon our initial team meetings, the group agreed on different areas of the project to focus on. With 5 group members, three areas were identified as being necessary starting points. These were as follows:

1. Game Logic (*Liam Wardle*, *Jack Williams)*
2. Graphics (*Charles Crawford*, *Alex Jones*)
3. Game Map (*Yucheng Zhu*)

Each area was designed separately, and extra features and issues were logged for future attention. Our modular design provided us with the ability to concentrate on different areas simultaneously;

modules initially did not depend on each other, which consequently aided modular testing and allowed each member to make progress without having to wait for specific features to be implemented. The design of our modular system is shown in the Appendix.

As to maintain efficiency, the groups and areas of development were kept fixed until each section was developed to a degree of workability. For example, the team responsible for basic game logic had the greatest initial understanding on this section, and therefore were best placed to continue development in this area. However, the team was constantly updated on how the code worked and interacted with other sections of the game.

The initial task assignment is outlined in the following table. These were the primary tasks, which could then be followed by integration of sections, and additional features added as and when possible. Time constraints included additional workloads and other commitments which affected the progress of development. However, achievable targets which took into account these constraints were always set, which allowed for a more flexible approach to deadlines.

Task	Assignment	Date
Module Design	Liam, Jack	28/10/2015
Basic Game Logic	Liam, Jack	28/10/2015
Basic Map Logic	Yucheng	28/10/2015
SDL concepts	Charles, Alex	28/10/2015

Table I: Initial task allocation with dates for commencement, including planning and implementation.

Subsequent assignments included:

- Integration of game logic with SDL

- Integration of map
- Attack animation
- Background design
- Render Sprites, background, buttons
- Creation of dialogues and questions

These were generally self-assigned, depending on the team’s current workloads and preference.

Logic

Early Design

What eventually became known as the “logic branch” of our module tree is where we kept all of the modules responsible for handling the game's mechanics whilst in a battle. These would be the invisible calculations the game would make to determine how the game would progress.

As our game was turn-based, we decided early on that it would be most suitable to have the game controlled by a state machine. Each state would be a different stage in the turn or a set of calculations the game needed to make.

Our initial draft of the state machine had the following states:

- Start – where any initial set-up for the battle could take place. This could include selecting an enemy, choosing a problem for the player to solve and any initial graphical set-up.
- Player Turn – where the player would be in control of the game and could choose what action they wished to take.
- Player Action – where the game would execute the players commands. This would include checking whether or not the attack was successful and displaying the appropriate animations.
- Enemy Turn – where the enemy would take its turn and attack back. We decided that as the enemy turn would be decided by the computer, there was no need to split the choice and action parts into separate states.

- Check Health – here, the game would check the player and enemy health values to determine whether or not the game should continue. If the player had zero health, the game should end and return to the menu. If the enemy had zero health, the player should proceed to the next level. Otherwise, the game should continue to the next turn.
- Win – this would have displayed a win animation and then sent the player to either the next level or map.
- Lose - this would have displayed a lose animation and then sent the player back to the menu, or ended the game.

The final version of the state machine was actually very similar to our initial draft. The main difference between the two is that the final version did not have an Enemy Turn state. Our initial idea was to have the enemy retaliate between player turns, with damage increasing if the player had failed the previous puzzle. However, it seemed unsatisfying for the player to be attacked even when they had correctly worked out the solution to the puzzle. This also did not add any additional puzzles to the game.

Instead, we felt that it would better if every turn was the player's turn. This let the player retain agency and kept their attention on the puzzle, which was the main focus of the game. We changed our design so that the enemy would reflect back the player's attacks if they chose incorrectly. Overall, we felt this gave the game a much better flow and was more focussed on the puzzle side of the game.

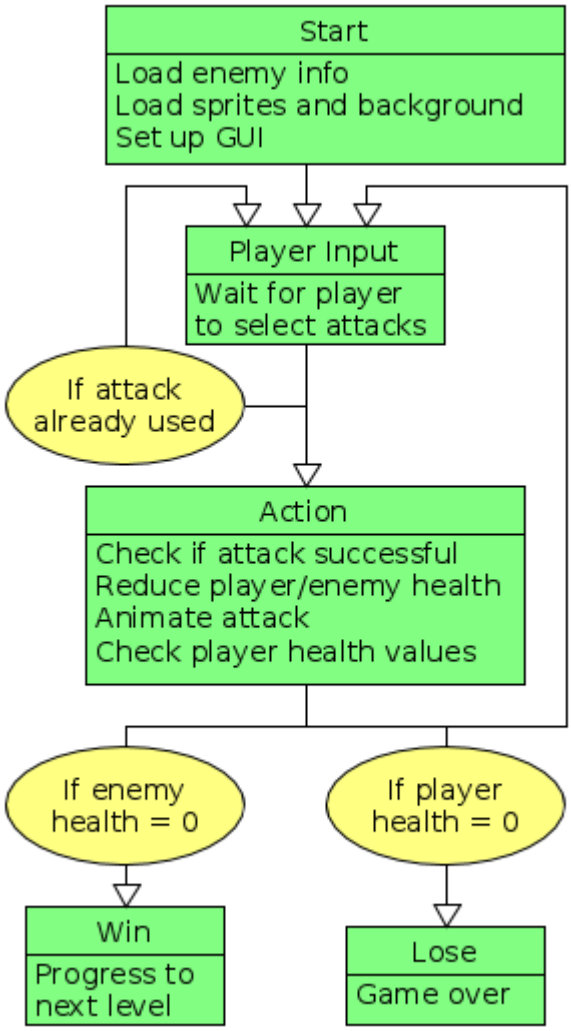

Diagram I: Final State Machine

Checking boolean problems

Having decided on roughly how the game would be structured, the next step was to think about how the game would handle the central mechanic of solving boolean problems. We needed a way for the game to look at the combination of elements the player had chosen and then determine whether or not this satisfied the boolean equation.

It is relatively simple to create a function in C that can check whether or not some inputs satisfy a boolean equation. It becomes considerably more complicated when the boolean equation needs to change from level to level, as it means the equation itself becomes an argument of the function. It may have been possible to create a parsing function that would interpret a string of characters and convert them into an equation, which could then be used to check the player's solution. However, this was beyond the scope of our programming knowledge when we were designing the system.

Instead, our solution was to create an array of “answers” for each puzzle, and then check to see if the player's choice of elements matched any of these. Whilst certainly cruder than the above method, this proved to be powerful and flexible enough to meet then needs of our game. The only major downside was that, as these answers had to be entered manually, there was potential for these answers to be entered incorrectly.

We chose to represent each attack combination as an integer, using the following formula:

$$attack = 100 * L + 10 * I + 1 * F$$

Here, *L*, *I* and *F* will be either 0 or 1, depending on whether or not the player chose to enable their lightning, ice or fire attack. This will generate a unique number for each of the eight possible attack combinations, and meant that we only needed to check one number in our answer array, rather than three. It was also immediately obvious for us to understand what each attack integer meant, as each digit corresponded to an element.

Note that we later realised we could have used binary numbers here instead of decimal and achieved the same result. This also would have meant we could have used bit-wise operations to check which elements were being used, rather than inventing our own formulas. However, by the time we realised this we had already been completed the main framework of the project. Therefore, we decided that it was not worth taking the time to change, nor was it worth running the risk of making the code unstable.

First Prototype

The next task was to build a working prototype of the game. As the display module was being developed independently of the games internal logic, the first prototype had to function without any visual display or input. Instead, it would have to run in the terminal. This kept in line with our goal of being agile developers, as it kept the logic and display branches independent and allowed us to work on different areas simultaneously.

One of our agile development goals was keeping to short development cycles for each leg. Therefore, we wanted to get our first prototype finished as quickly as possible. Whilst we still

wanted to demonstrate the core mechanics of the game, solving boolean logic problems, this meant that some features would be left out. The two main features omitted were multiple enemies and a check to make sure the player had not repeating any attacks. We also chose to leave out the enemy's turn, but as explained earlier, this was eventually cut from the game entirely.

Even though we had a clear idea of how we wanted our game to operate, we struggled at first to create a working module structure. Our original design for modules drew heavily from our state machine structure. We had a module for each state, all governed by a separate state machine module.

When it came to actually writing the code, we discovered that it was difficult to maintain the dependencies we had drawn in our module diagram. With hindsight, it is easy to see that we tried to arrange the modules based on the chronological order we expected them to be used by the programme. We quickly realised this approach would not work and we decided to start again before we wasted too much time on it.

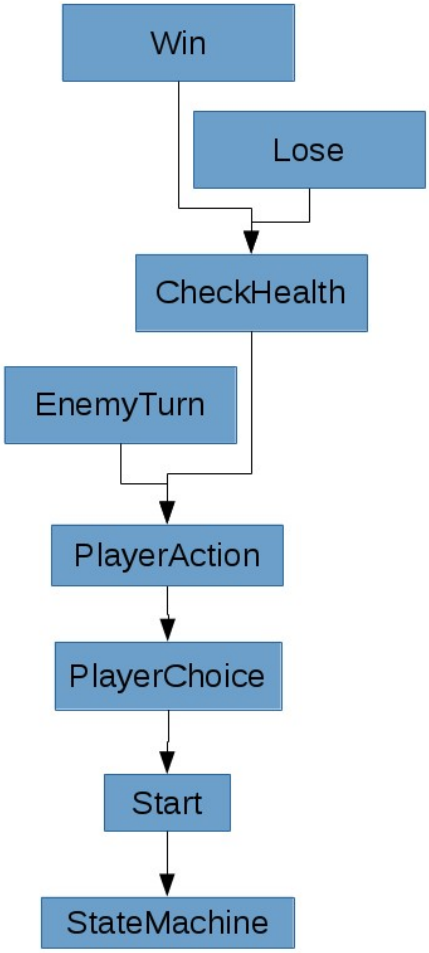

Diagram II: Original Module Diagram (rejected)

Instead, we went back to some of the example projects provided to us in order to look for inspiration. This gave us a much better idea of how to build a project up, starting from the simplest functions and then adding to them. Starting with a base module very similar to one of the examples, we created a new structure which we retained for most of the project.

There are a couple of important features to how we structured our code. The first was to include a test function with every module. These would be able to run when the module was compiled with

the modules below them in the module diagram. This let us quickly test the code one module at a time and check that changes made to the game had not caused any errors. These tests were developed alongside the main body of code.

The second feature of our structure was the separation of interface and implementation of our code. This meant that we wanted to ensure that the header files only contained declarations and definitions that were used elsewhere in the code. Anything else would be kept in the main c file so that it could only be accessed from within the module. This was decided with the goal of keeping the code stable and secure, by making sure that other modules were not accidentally calling functions they did not need to.

Naturally, we chose to start developing at the bottom our module diagram and worked our way up.

Base Module

The only functions contained in the base module are the success and fail modules, which we used to write to the terminal during testing. This was an idea we adapted from one of the example games provided in the course. This allowed us to write test functions more quickly. These functions did not have much of an effect on the game itself.

The header file contained a forward declaration of our battleState enumerated type. This was placed here as most modules needed to be able to access the game's battleState variable at some point.

Enemy Module

This module contained the enemy structure. This was used to keep all of the information about a level's enemy in one place. Originally, this only included basic information, such as the enemy's weakness array and health total. As development progressed, however, we realised that we could use this structure to store all the information required to generate a level. Therefore, we added more properties to the structure, such as the enemy's sprite and background image for a level. This worked as each level was already defined by the enemy encountered within it, so the level and enemy became synonymous.

Check Attack Module

This module was responsible for checking whether or not an attack was successful. This was relatively simple to build once we had decided on the best way to represent an attack (as detailed earlier in this chapter).

The first build of the game did not check to see if a player had already used their selected attack. Once we had decided to add this, we felt that the check attack module was the best place to place it. The checkUsed function was simple and could actually have been placed anywhere, but we felt that this module made the most sense for thematic reasons (as it was still checking attacks). This at least made it easier for us to remember where we had placed it.

Damage Module

The damage module deals damage to either the enemy or player, depending on whether or not the player's attack was successful. It then checks if both the player and enemy are still alive and returns 0 if so. If not, the it ends the battle by returning -1 or 1.

There are also two extra functions, one for winning the battle and one for losing it. Currently, these only print a line to the terminal and then return one of the values above. However, we have included them so that if we wanted to keep working on the game, we could add more functionality here. For example, an animation could play on screen to indicate success or game over.

Input Module

In the final version, the games input is handled by an on-screen GUI. However, we still needed a way to play through and test the game before we had integrated the graphics modules with the logic. Therefore, we developed a simple input module that would use the terminal as input.

In this module, the player would type in the three-digit integer value of their chosen attack. The game would then use this input to calculate damage, exactly as if it had received it from the GUI.

As we knew we would be later removing this from the game, it was kept isolated from the rest of the code in its own module. This meant that once we integrated the graphics, this module could be

easily removed from the game by simply removing it from the makefile.

State Machine Module

The state machine is the backbone of the game, as it is responsible for calling most of the games functions and deciding the flow of the game.

We built this by putting a switch statement inside a while loop. Each time the game would run through the while loop, the switch statement would check what the state variable was currently set to and run the associated functions. During this, one of the functions might change the value of the state variable and let the game move on.

Most of the states would include an SDL function that would play an animation or wait for the player's inputs. These functions would cause the state machine to pause until the function had been resolved. Others would only stay active for a split second.

The game would repeat this until the while loop's condition was broken and the game (or level) would end. This condition was broken by the win or lose state changing the value of the result variable. Currently, that is the only functionality these states have. However, as with the win and lose functions in the damage module, these exist to allow us to smoothly expand in the future.

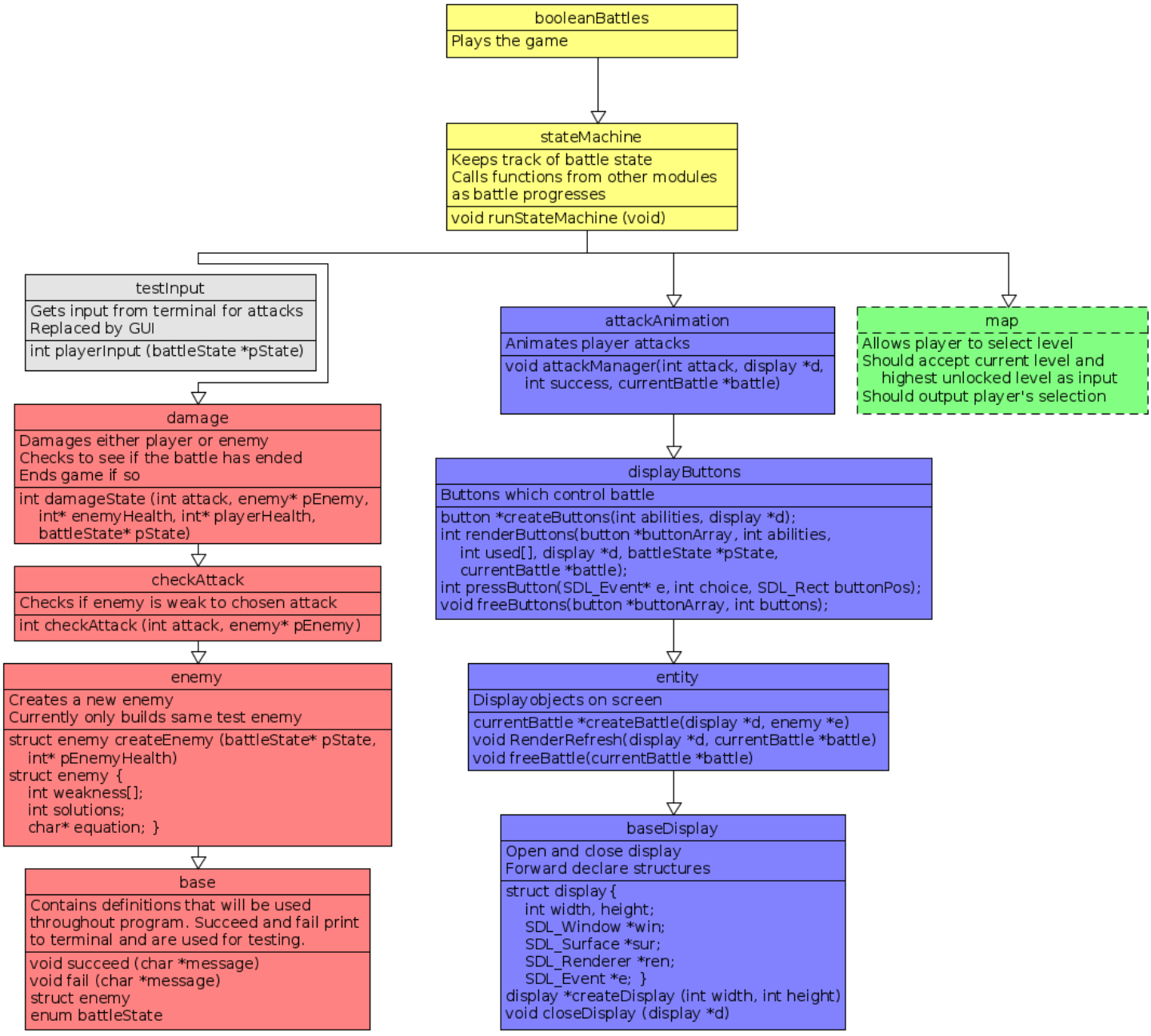

Diagram III: Final Module Diagram

Style and Design

Static Graphics

The general graphical design of the game is based off a fantasy RPG 2D game, with an aim to improve the appeal of it by using fun, block colours in the foreground, and pastel-style graphics in the background to give an edge of appeal.

Theme

The game is not based on a time-frame, however has a running theme of 'fantasy RPG'. This allowed us to hone in on the RPG-side of it, without getting too focussed on realism. Therefore it was appropriate to use any style of hero/enemy we could find that would incorporate a enjoyable and appealing aspect. This allowed us to span most time-eras and locations for a level, such as ninja/dojo final level, and the rainbow/faun early level. We felt there was an importance to using a 'fun' theme, as it helps to keep the user involved and engaged whilst playing the game.

Colour palette

The use of block, 'RGB-Style' colours, are synonymous with old RPG games. These graphics focus less on realistic graphics and more of engagement of the user with content, were vital to producing the game in the way we thought would be most realistic (development wise) and engaging (to the final user). The static background images in each level were designed with intent to be more realistic looking, using smoother edges (anti-aliasing), more detail and use of perspective. This

aimed to draw the focus away from the background, and onto the foreground where the 'learning aspect is, whilst still leaving each level looking attractive and appealing.

Software

The use of the open source software GIMP2 allowed us to design and implement precisely made graphics, that could look as applicable to the game style as possible. The program was excellent for producing perspective backgrounds/foregrounds due to many effects which incorporate distance and allow customization with angles. The ability to manipulate images and build them layer-by-layer was an effective way to produce the graphics for the game, starting in the far background (sky), through the middle ground (active layer – I.e. the volcano in the volcano level) and finally the foreground (trees, fences, etc.).

Sizing

We chose to use a standard 1080 x 720 as we thought this would be compatible with most screens a user would be using for their computer, and it provided enough space for the graphics which were pre-designed to fit into comfortably.

Graphics

Design

To keep our functions simpler we have a display structure which contains information about the window and the renderer. We use a single renderer to render all of our images. In parts of the game involved in rendering we opted to only need to pass our display structure.

We decided that the player control should be simple to understand with as few interactive elements in game as possible. This is why we settled on the choice to have all of the player control in the buttons. These were click-able buttons that toggle between using an ability and not using it, the only other interaction a player can make is skip through the brief enemy introductions and quit out of the game.

Button logic

To simulate a button an area was created using an SDL Rect structure which specified an area on the window. The programme checks if the cursor is within this are. When the user issues a mouse click while the cursor is within this area a value is toggled to capture the choice the user is making, this could be repeated for all of the buttons available to the player. As well as capturing the logic to return to the state machine, we use this value to change between displaying two different images, one to represent an ability not being used and the other to represent the ability will be used. Values for each of the buttons are added together to create a unique value for each possibility of selected buttons. Another special button was also implemented this one was used to commit the player's decision which is returned to the game logic, before this commit button is used the player

is able to make as many alterations to their selection as they please.

To avoid hard coding in each of the different button images we opted to have a file containing the names/paths to the buttons. We were then able to search through this file for the button we want to render.

Attack Animations

The attacks used in the game are designed to not only move statically across the screen but also rotate about an axis. We chose to add this feature because the game felt a bit too static and non-engaging. To accomplish this feature, instead of using a single image we use a sprite clip, this is an image containing the phases of the animation we want to use as shown in the diagram below. We start by looking at the first instance of the spell and rendering it to the screen the area we look at is then shifted along (by 250 pixels in our case) this instance of the spell is used the next time the image is being rendered.

When the spell moves across the screen it travels a set distance, this distance is sufficiently long that the spell can be reflected if the attack is unsuccessful. Upon multiple attacks the programme will always go in the order.

Fire -> Ice -> Lightning

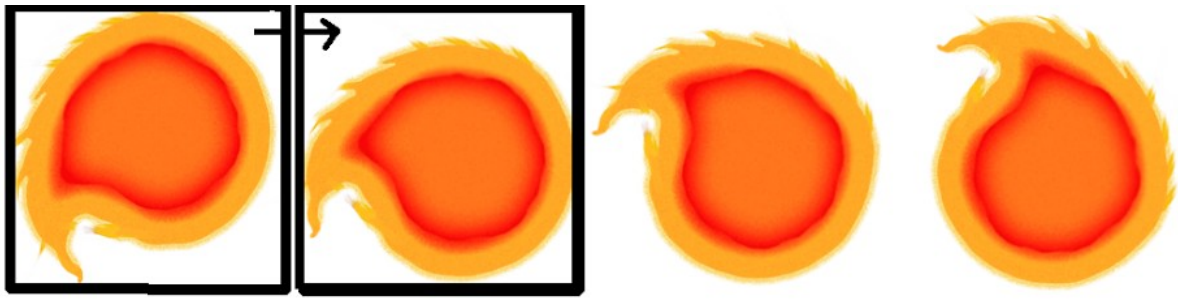

Figure III: Image shows how an area of the image is selected and used this area then moves on to the next part of the sprite clip.

Integration

Agile programming is the technique of solving software development programming problems as they occur. This is achieved by constantly recompiling pieces of code and ensuring they work to some specified rules. Initially we broke into 4 groups focusing on separate aspects of the game. We began by not using an agile approach, developing some parts of the game too far without integration which resulted in a piece not being finished showing the risk of not integrating often. Each sub group would consider their own design, implementation and testing for their element of the game.

After certain pieces of the game were complete an attempt was made to fuse them together. It became immediately clear that integrating aspects of the game together was a very cumbersome problem that took a lot of effort to solve. The main issue arose due to the dependency of modules within our game, we had to ensure that certain libraries were only included once and in an order which made our libraries available to any later modules which required them.

One potential issue when integrating work is that without communicating some standards, members can produce conflicting code and styles. Initially different subgroups were using two different versions of SDL as well using different approaches for displaying images. One method is bliting to surface the other is using a renderer. Fortunately through group meetings the point was raised early and we committed to using renderers and SDL2.

Once parts of the game had been integrated together we moved towards agile development. Work that members of the group were producing were immediately combined and the entire game

complied known as contiguous integration, with the help of github the team would combine our work together often during working periods. It allowed the team to assess our progress more effectively and we were able to make more accurate predictions on how long a new feature would take to implement.

This process is all about moving onto the next stage once the previous is completed. Using the waterfall approach one would need to decide at the start of the project the entire list of features appearing within the game. We began the project with it already in mind that our game would probably evolve with new ideas and as our programming skills improved we would want to improve older code.

Testing

The next section details our approach to testing our code, the problems that arose with testing, and how we overcame them.

We wanted our testing to meet the following criteria:

- Agile – along with the rest of our development, we wanted our testing procedure to allow us to be flexible as our project progressed. Each small part of the project should be thoroughly tested in isolation before being integrated into a larger part.
- Quick – if tests were going to be repeated a number of times, we did not want them to take a long amount of time. Nor did we want to be repeating unnecessary tests. Lastly, we did not want to be doing manual tests where they could instead be automated.
- Comprehensive – obviously, we wanted any testing we did to be consider as many inputs, scenarios and edge cases as possible. This is with the aim of having the most stable code possible.

Taking the above into account, we decided that integrated unit tests would be the best approach. This meant that each module would have its own test function built into it. In the final build of the game, the test functions would never be called (and could be commented out of the code). However, by taking advantage of the module structure we had already developed, we could build the programme up one module at a time, replacing main with the appropriate test function.

Rather than waiting until after we finished development to begin testing, the test functions were made in parallel to the main body of code. Each time we wrote a new module, a test function was

also written before moving on to the next module. This was done in order to capture potential issues as soon as possible. Whilst we were trying to keep our modules independent from each other wherever we could, there were still dependencies. Therefore, we wanted to fix potential bugs before they could have an impact on how we wrote the rest of the code.

Our typical testing cycle went as follows:

1. Make a change to a module.
2. Edit the test function to take into account any changes/new features in the module, paying special care to include any possible edge cases.
3. Compile that module with any below it in the hierarchy and check that it still passes all tests.
4. Add the modules above the current one, one at a time, and check that they still pass their tests.

Testing each module in isolation gave us a greater sense of clarity, as we did not need to worry about other modules obscuring any potential issues. Therefore, we could be confident that each module worked exactly how we intended it to, before integrating it with the rest of the project. Even when the modules were integrated, the isolated testing meant that we could freely edit modules, and then only need to test those modules (and the ones above it in the module hierarchy) again. As such, our testing allowed us to continue working in an agile manner.

Not needing to test the entire project whenever a change was made to one module also allowed our testing to be quicker, which was another one of our aims. To further improve the speed of our testing, we also used automated testing wherever we could. This was not always possible; for

example the display modules required someone to visually check that game looked how we would have expected. We also decided that it would be simpler to manually test any player input, rather than create a programme which could have played the game on our behalf.

However, for a lot of the “logic branch” of the module tree was only responsible for setting certain variables. This made them ideal for automated testing. To do this, we created a success and fail function in our base module, which would either print an error message and terminate the programme when the game failed a test, or print a success message to let us know we had reached the end of the test. We also changed our makefile to automatically run the programme when compiling, which allowed us to quickly compile and test multiple modules in quick succession.

Naturally, how we made our tests comprehensive varied from module to module. Some modules had a wide range of possible inputs and so required a lot of thought in order to make sure we captured every possible edge case. Others were comparatively trivial to test.

Base Module

The only functions contained within the base module were those used during testing to print messages to the screen. Therefore, it was trivial to check this module would work and did not really need testing. We still included a test function for the sake of consistency with other modules.

Enemy Module

The purpose of the enemy module is to create an enemy structure and fill it with the relevant

information, depending on the level. This is simply a matter of copying data from an array, and so there is little room for error.

Our test function checks this by choosing a hard-coded level and checking each property of the structure is set as expected. The only input for the function is the choice of level, which determines which element in the array the function should copy. It is safe to assume that if the function works for one element of the array, it would work for all. There could be an issue if the function was somehow given an input outside the range of the array. This shouldn't happen during the normal play, but the function has a fail-safe built into it to exit the game if this does happen.

When we first created the enemy module, it was used only to store information about the enemy's weaknesses and health. As the project progressed, this became a convenient place to store information about the entire level. Therefore, as we added new properties to the structure, we had to ensure our test function was updated to ensure that these were being correctly copied.

Check Attack Module

The purpose of the checkAttack module is to check if an attack integer exists within the a given enemy's weakness array. It should return 1 if so and 0 otherwise. It also needs to check if an attack has already been used.

To test this, we create an enemy structure and run the checkAttack function with it and every possible attack combination. It may not have been necessary to check it with so many combinations, but it is trivially done with a for-loop and did not create much extra effort to do so.

An astute observer may notice that many of the test functions deal with the same enemy, specifically the 8th entry in the enemy array. This is partly left over from when the code was in its early stages, and enemy 8 was the only enemy in the game. When new enemies were added to the game, most of the test functions were left checking the original enemy they were hard-coded to do so.

As we were adhering to a strictly modular design, the specific enemy input into the functions shouldn't have made a difference. Furthermore, as many of our tests were checking against hard-coded examples, it would have taken considerable time and resources to manually add these extra tests.

Enemy 8 also has a convenient property which makes it ideal for testing. When checking attacks against it in order, from 000 to 111, the first five attacks should fail, whilst the final three should succeed. This lets us quickly wrap these tests inside two for-loop loops (one for failed attacks, one for successes).

Finally, we need to check if the checkUsed function is working. To do so, we create an empty used attack array and fill it with FILLER. We then test it four times with attacks 0, 0, 1 and 1. The first check should pass, as the array only contains filler. The second should fail, as we have already used 0. The third should pass, as we have not yet used 1. Finally, the fourth should fail, as we have already used 1. This should be sufficient to check that it work under all circumstances.

We finish the test by making sure that all of the values in the used array are what we expect.

Damage Module

The damage module is responsible for a few things. Depending on whether or not an attack was successful, it decreases the player or enemy's health by 1 and then checks if both are still alive. It then either sets the current state to WIN, LOSE or PLAYERINPUT, depending on the result of this check. It would also return -1, 0 or 1 based on this check as well.

This gives us two main things to test: does function decrease the right person's health, and does it end the game (by returning 1 or -1) at the correct time. To check this, we would continually call the damageState function repeatedly, under different conditions, until the function returned a non-zero value. We would also count the number of times the function had to be called until this happened.

The test would be passed if a successful attack would return a WIN, or 1, after 3 turns, and an unsuccessful attack would return a LOSE, or -1, after 3 turns.

Originally, the damage module would be given an attack integer and use this to call the checkAttack function. Later on, we changed the format of the code so that checkAttack was called elsewhere, and then the result of this was passed to the damage module. Ultimately, this didn't make much difference to the code, but it does mean that the test functions still check the module eight times (once for each attack module) where really it only needs to be checked twice (once for a successful attack, once for an unsuccessful attack). This does not cause any issues, but is just further evidence of how our code evolved.

Display Modules – Splitting the Module

Whereas the logic modules had their test functions written as the code was developed, the display modules had their test functions written afterwards. This was partly due to the experimental nature of the way the display functions evolved (remember that the team was new to SDL when we first started) and partly due to personal preference of the team members who wrote them.

However, when it came to writing the test functions, the team had a difficult decision to make. We had originally planned for their to only be one display module. This was so we could keep the display structure local to just that module and we would not have to worry about other modules accidentally affecting the display. This would make the code more stable and modular.

However, we had underestimated the number of functions that would need access to the display structure. By the time we came to test the module, it had as many lines of code as the rest of the project combined. This was making the project unwieldy and non-agile.

Ironically, trying to isolate the display structure in attempt to keep the design modular, was directly going against our original goal of having short and easy to manage modules.

This quickly became apparent when we tried to create a single test for all of these functions. We therefore decided to split the display module into several smaller ones. The reasoning was that the increased flexibility this would provide, along with making testing easier, was worth losing a little bit of stability for. Even though this meant that any module could now access the display structure, the small scope of the project meant that this was unlikely to be a problem.

Display Modules – First Integration

With the logic side of the project, most of the testing could be done automatically as we were primarily concerned with making sure that the variables had been set correctly. With the display modules, we also had to introduce more “eyeball testing”, i.e. making sure things appeared as we hoped.

The first, and probably most interesting bug occurred when we first tried to integrate the display with the logic. Until then, our makefile would compile the specified module (and all those below it in our module tree) and then run the test function, at which point we would (hopefully) receive a “module ok” message in the terminal.

When we tried to add the display to our makefile, we discovered that SDL was unable to find any display devices. This bizarre bug confused the entire team (and even Ian) for quite some time. We eventually deduced that the cause of the bug was due to us trying to run the programme directly from the makefile. As Ian explained it, the makefile creates its own virtual environment when compiling. When we asked the makefile to run the test function immediately after compiling, it was trying to run it within this virtual environment. Therefore, it was unable to access some of the computer's hardware, including the screen.

The fix was simple: we removed the line from the makefile which ran the code after compiling. We still kept this line for non-display modules, as they were not affected by this bug.

Display Modules – Other Notes

Testing each of the display modules followed a similar pattern; we would add a new module to the programme and then primarily use eyeball tests to check it worked as expected.

There were some interesting bugs that occurred. For example, we had an issue with the spell animations flickering. This did not appear to be a problem until we started to properly refresh the screen (before then, the spell would leave a “smear” of itself across the screen). We therefore came to the (seemingly logical) conclusion that this flickering was due to the way we refreshed the screen.

However, we later discovered that our animation was trying to include one more frame in its loop than we had in our sprite sheet. This wasn't noticeable before, as the smear effect had hidden the missing frame. Once we realised this, we fixed it by simply reducing the number of frames in our animation.

Another curious bug worth mentioning is one that was not resolved until later on in development. We had noticed before that the close button in the corner of the window did not always properly, and sometimes seemed to take multiple presses until it worked. Once we added multiple levels, we noticed that sometimes this would take us to the next level rather than close.

At this point we realised that the close button had been given the same functionality as the attack button. Therefore, attempting to close the game would instead initiate an attack against the enemy. If the player continuously pressed the button, either they or the enemy would run out of

health and the game would end, which made it appear from the outside that the close button did work (eventually). Of course, once we added more enemies to the game, this approach would sometimes lead the player to the next level.

When the game is waiting the player to select their attack, it also checks to see if the player presses the close button. This all happens within a loop, which pressing the close button would break. However, instead of closing the game, we originally had this return 0 (which is a neutral attack).

Instead, we had this return QUIT, which was defined to be a negative integer that the player could generate as an attack. Returning this value would end the game.

Play-testing

Module testing is useful for testing individual functions. However, it is equally important to check that these functions have been arranged in a way that makes sense. Therefore, we also needed to play test the game as a whole.

There were two main things we were looking for here. First, does the game play how we expect? Second, can the player break the game in any way?

The turn based nature of our game did help make play testing simpler for us, as it restricted the number of things the player can do in the game. Whereas in an action game, the player may be free to move about as they wish and interact with the environment in any number of ways, in our

game the player is restricted to a finite number of moves.

However, this did not mean we could be complacent. As well as playing through the game how we would expect a normal player to do so, we also had to check if there was anything else the player could do to break the game. For example, in some playthroughs we would randomly click the mouse and press buttons on the keyboard to see what effect this would have.

Not every fault in the code is a game-breaking bug; sometimes they would just be things not working how we would expect. This is where play-testing is often more useful than automated module testing. A good example of this would be the enemy which had an incorrect value in its weakness array. This was only detected when playing through the game and would have been impossible to find without human observation.

It did, however, cause said human to temporarily question his own understanding of boolean algebra. In fact, several members of the team played through this level several times without noticing the mistake. This also highlights one of the main weaknesses with eyeball testing: whilst a human can spot things that a computer cannot, they sometimes lack the consistency to do so.

Evaluation

In this section, the project in its entirety is covered; looking at areas that went well or not so well, as well as what the team would do differently if given another chance. Additional features if the game was expanded or developed further are also included.

Making the first version simple but still containing the core features of the finished product proved to be a very successful aspect of our project; giving us a working prototype early on. This allowed us to carry out testing at an early stage, and to resolve issues or to add additional features quickly. Modular design was used from the outset, which made integration and testing significantly easier than otherwise would have been, which can be seen in the Logic section of the report.

One of the most successful aspects of the project during the development stage was our ability to adapt to changes, whilst ensuring the central aspects of our design were not affected. For example, from the outset, our idea of having a state machine controlling each section of the game when running was never changed and was agreed to be a useful way in which to structure the game. Code and features were changed and adapted to work effectively with this state machine.

Another important aspect of the game which we feel tackles the issues of the product containing an educational aspect is the evolution through the levels. The difficulty quickly increases, with the player being introduced to new symbols, equations and requirements to complete levels of the game. For example, the player may not learn that they are unable to repeat answers when playing for the first time; being required to answer every possible solution to beat the level.

Overall, we feel the project progressed well and are happy with the outcome, however one thing we would have changed if we were to repeat the project would be to sort out disagreements earlier, ensuring each team member knows exactly their tasks within the group at any given time. A breakdown in this particular situation resulted in multiple versions of code being produced relating to an individual section of the project. This particular event proved that successful teamwork was a mixed bag, with certain members of the team working well together; being able to co-ordinate work efforts and manage workloads, and also a significant amount of miscommunication which consequently lead to consistently hampered progress.

Additional features

The below list is comprised of certain features that were not able to be implemented in the time-frame provided, as well as features that are currently implemented but have the potential to be expanded upon as to further develop the game in its current state. We feel the potential for expandability is a major positive for this product.

Enemy database: The database was placed in our backlog of work to be carried out, and was only going to be an addition if we felt we had enough time to implement it. It was not deemed to be an essential feature as our working prototype successfully worked with an array of ten enemies. However, we felt for expandability, a database could have provided us an efficient means of adding/changing enemies and their characteristics. We would have used SQL to implement this feature, but our lack of knowledge in the use of SQL would have resulted in extra work.

'Zen version': An additional feature that was considered, but only as a future version of the game. This would involve a game mode with infinite questions, simplified questions and unlimited lives. A version such as this could have the benefit of allowing users to play the game in a non-stress way. An alternative or extra could be a tiered 'Zen mode' to allow for a range of difficulties. The addition of a database could be required for this to work effectively.

Damage animation: Currently, no animation on receiving damage has been implemented, other than the animated attack being rebounded upon the hero when a wrong answer has been chosen.

Add tracked moves, health bars to GUI: To make it more user friendly, health bars to display remaining life could be displayed above the hero and enemy. Currently, health is only shown in the terminal. Adding a list of used combinations to the GUI would also help to keep track of combinations, as to not repeat the answer.

Win/lose animation: An animation for either succeeding or failing could be displayed, which would provide an incentive to beating the game, and also as a way to show the game has ended. Currently, the game closes upon success of failure, without any additional change in display to show this.

Increased number of enemies/levels: An easy feature to implement; by increasing the number of enemies and thereby increasing the numbers of levels available.

Increased difficulties: More variations on Boolean equations could be added, for example an increased number of logical symbols, which in turn will alter the levels of difficulty available.

Map integration: The map was not able to be fully implemented in the time frame provided, with integration proving to be the major difficulty. The map would provide an alternative display to show and choose the level.

Conclusion

Following production of our working design of the product, the group set out to answer the following questions in order to analyse the performance of Boolean Battles against the initial brief. Is the product:

1. Educational?
2. Enjoyable?
3. Repeatable?
4. Easy to understand and use?

We believe that all questions were answered to a degree; however the game's repeatability proves to be the most likely feature that has failed to be answered fully. The linear nature of the game and the natural progression through the levels and change in difficulty means that users are unlikely to replay the game in its current state. Upon successful integration of the map, players would have had the opportunity to choose previously unlocked levels to repeat, however currently there is no incentive to do so. Our additional features section provides answers to this question of repeatability, which we feel would improve this feature.

The group was satisfied with the ability of our product to answer the remaining questions, for example we believe users new to the theory of Boolean logic would find the natural progression and evolution of the game to be both educational and enjoyable; with enemy dialogues providing the means of explaining instructions on how to proceed. The simplicity of the user interface minimises the need for detailed instructions or a separate tutorial. The attack animation provides a

satisfying motive behind understanding the use of a Boolean logic symbol, and answering the Boolean equations correctly in order to attack the enemy and consequently beating the level.

Overall, the group is satisfied with the final prototype being able to answer the most important aspects of the brief; being educational and enjoyable. The game was able to effectively convey computer logic such as NOT, AND and OR gates. Given the lack of programming knowledge prior to September, we are satisfied with the outcome and consider it to be an achievement.

To confirm the success of our product, research would have to be carried out on the target audience, for example holding focus groups and gaining feedback. We were unable to carry out our own market research due to our workloads after being reduced to 4 members, however we do feel this would have been an appropriate method of attaining information regarding our key points and therefore being able to conclude the success of Boolean Battles.

Minutes

8th October

11. Began with a brainstorming session to make a list of possible ideas.
12. During this, a list of game genres was consulted. The list was ordered from easiest to hardest to build a minimum viable product (source: Extra-Creditz). This was to keep our ideas manageable.
13. Ideas were:
    a. Turn based RPG based on solving Boolean logic puzzles to attack.
    b. 2D platformer controlling a robot. The robot has had his code erased and needs the player to periodically re-enter its missing code.
    c. Game based around solving Boolean logic puzzles to manipulate optical illusions.
    d. Driving game where you programme a car with commands to guide it round a track.
    e. Game where a player has to fill in the gaps of incomplete code.
    f. Bejewelled-style game where a player has to match together chunks of code from the same script.

14. Group then discussed advantages and disadvantages of each idea to pick our final three.
15. Final three were:
    a. Boolean RPG
        i. Directly teaches logic
        ii. Easy to make engaging
    b. 2D Robot Platformer

i. Directly teaches code
ii. Platforming is fun
iii. Robots could be made to look like anything (multiple characters possible)
iv. Robots are easy to animate (if on wheels/skis/floating)

c. Bejewelled code matching game
i. Directly teaches code
ii. Causal – could be exported to phones
iii. Bejewelled is already popular with age group
iv. Easy to make visually appealing

16. Rejected ideas:

a. Optical illusions
i. Would be too difficult to create Escher-like graphics
ii. Difficult to introduce coding elements

b. Driving game
i. Worried that it wouldn't be engaging enough when compared to traditional driving games

c. Fill in the gaps
i. Worried that this also wouldn't be engaging by itself
ii. However, decided that the concept could work well in combination with other games
iii. Having settled on a final three ideas (for now) we fleshed out each idea a bit.

d. Bejewelled matching game
i. Considered having the game exactly like Bejewelled, except with the colours replaced by types of variables. So the player would match together ints,

chars, floats etc. There were concerns that this would be too easy and unengaging.

ii. Considered having a grid made up of rows and columns. The player would need to swap tiles on each row in order to make the columns read as completed code. First row would be start of code, second row would be next part and so on.

iii. Group decided this concept could work.

iv. Increasing difficulty, introducing new concepts between levels. Could increase number of rows as the player carries on too.

v. Could have each level actually run code in order to show player what they've made, which provide a satisfying feedback loop for the player.

vi. Should be easy to create graphically

vii. Could be put on phones

e. Robot platformer

i. Initial concept was a robot who has had his code deleted. The player would need to re-enter missing parts of the code in order to progress.

ii. An early challenge could be for the player to declare a variable for the robots movements speed in order for him to be able to move again.

iii. Other challenges might include operating a lift, which runs a for-loop in order to ascend.

iv. Gameplay would involve standard 2D platforming in order to keep player engaged.

v. Game could open with robot being fully operational in order to get player immediately engaged with more "gamey" aspect before throwing them into

coding.

vi. Robot characters have advantage of being easy to animate (providing they have wheels rather than legs) and can look like anything. Could have multiple characters for player to choose from (some cute, some cool).

f. Boolean RPG

i. Player would progress over a map, engaging in battles with monsters.

ii. Each monster would have a Boolean logic statement, which the player must solve in order to beat it. There were two ways we discussed of doing this:

1. Either the player is given a statement (e.g. X & Y) and told whether each variable is true or false (e.g. X = 1, Y = 0). The player would then need to work out if the statement is true for these variables.
2. Or the player is given a statement, and then has to set the variables to either true or false so that the statement is true.

iii. We went with the latter, as it gave the player more options, making it more engaging and challenging.

iv. Decided that instead of algebraic expressions, the challenge could be framed in terms of elemental attacks. So instead of (X & Y), we have “this creature is weak to water and fire attacks.” The player would then need to use fire and water to beat the creature.

v. Creature health based on number of possible solutions, so (X & Y) would have 1 health, whereas (X or Y) would have 3 health.

vi. Some concerns expressed that the fantasy elements might be too boy-y and nerdy, and might only interest kids who would already be interested in computer science. However, the gameplay mechanics could be applied to

other themes as well.

17. Ended meeting there. Group will meet again on 12/10/2015 to discuss any other possible ideas and improvements on existing ones. We will also split the presentation amongst the group.

12th October

Discussion of other in-game unique points/game-play ideas for each potential game:

Discussion of power point contents:

- Intros

- Core ideas

- Each idea:

    - Overview → visual mock-up

    - Unique selling point

    - SWOT analysis

    - Big pretty pictures

    - 5-6 slides per game

Split into groups of 2, each taking a power point (Charles & Jack – RPG game, Josh & Liam – 2D Platform Robot game, Alex & Yucheng – Bejewelled-type slider game)

Agreed to work on them on Tuesday 13th as pairs then assimilate the slides into one powerpoint on the Wednesday 14th/Thursday 15th.

26th October

Set up group GitHub

Modules – prototype discussion – other game discussion

Potential battle question: for loop, iterate over 5 times, guess the variable number

Potential battle question: random choice, assignments

Incorrect answer to a question will choose a random element.

After a few rounds; comparing the types used in the battle so far

Speed pickup item to fit somewhere in the game

Simple text based version of battle

Attack type stored within an array – I.e Fire (1) Water (10) Grass (100)

Tutorial – bring in each logical symbol one at a time. I.e “Not Grass” to introduce th symbol… then “Not Not Grass” to indicate the symbol chaining

Card idea (?) – someone add to this …

win – big reward, lose – small reward

Prototype plan:

Battle layout – static shapes, no animation, printouts of text rather than animations.

State machine:

Enumerated type listing all states the game can be in;

start, then a seperate start script which initializes game, then tells state machine start is finished

and move onto next stage.

If youre in this state, do this, if your in this state, do this

state machine will call only call one script, within this script it will call everything else required.

enum types {

…

…

}

switch

case START: call start scripts

case BATTLE_BEGINS: call battle scripts

Alex / Charles – begin working with SDL – get a background set up, maybe a map?

- Background – Images for characters – try get a variable from code to display a health

Josh/Yucheng/Jack/Liam – State machine work.

START

- Fetch player/boss health

- Generate/fetch a problem

PLAYER CHOICE

- (input buttons – fire/etc)

- take input

PLAYER ACTION

- check solution

- animate

- does an attack – displays damage

- changes boss health

CHECK HEALTH

- if health zero go to win

- if health not zero do something else…

- back to PLAYER CHOICE (during battle)

WIN

- … exit

LOSE

- … exit

Personal Evaluations

Liam Wardle

A large part of my personal contribution came from developing the overall structure and framework, which the rest of the game was built around. Early on, I suggested the idea that we use a state machine to manage the flow of game-play. This was something which the rest of the group were keen to implement, so with their agreement I drew a first draft of the design. This became core to the final design, and I believe that developing this early on expedited the development of the entire project.

Having a background in mathematics, I volunteered to work on the early development of the logic branch. Amongst other things, I developed the module structure and created the system by which we efficiently checked if the player had solved the boolean equation (by converting their answer to an easily checked integer). I think it would be fair to say that I was integral to developing a lot of the actual game-play calculations.

One of the things I have learnt from this unit is how to structure a modular program. My first attempts at this were, frankly, unworkable. However, after doing some research we were able create something which was both flexible and secure. Since then, I have been able to use a modular style in my other units with confidence and ease.

Unfortunately, two of the six members of our group left partway through the project, leaving the remaining four to take on their workload. I feel that one of these member in particular had a

disruptive effect on the group, derailing group meetings with arguments and purposely going against design decisions which the group had already agreed on. The group spent a lot of time attempting to work around his negative attitude. These two members were responsible for looking into the map module, something which was not started until very late into the project and never finished. Had we known from the start that these two members would leave, it may have been possible for the four of us to develop this ourselves. Sadly, this was not the case.

However, the remaining four members were able to work together with excellent chemistry. We used a combination of pair-programming and regular group meetings to ensure that everyone was able to stay informed and express their ideas and concerns. Regularly swapping pairs ensured that everyone understood the project in its entirety.

As I was initially focussed on the logic side of the project, I was not involved with the graphics until it came to our first integration. This was something I wanted to be involved with, as it gave me an opportunity to learn about SDL and work with more members of the group. This resulted in me gaining an understanding of SDL, which I was then able to apply to in other units.

Having came up with a lot of the overall structure of the project, I had a clear idea of how each module should fit together. This meant I was able to give a lot of guidance when it came to integrating modules. This is something I tried to do whenever possible. As such, I tried to involve myself in most of the integration stages.

Alex Jones

Initially Charles and I were tasked to understand SDL and begin work on the graphical side of the project. Charles focused on animation and aesthetics while I focused the player interaction with the game. The player actions were controlled using 4 buttons at the bottom of the screen, three for attack selection and one to confirm the player's choice. To create the appearance of a button I created a rectangular area on the screen which detected a mouse click within it. An image was position over the same area, upon the mouse being clicked the button image would toggle and a value would be given to the logic of the game to process the player choice.

After this task was the first instance where separate aspects of the game needed to be integrated together. Liam and I worked on integrating the game logic with the player interaction. This involved working out the dependencies of the involved modules and altering current work and our makefile.

The rest of the work I did was odd jobs such as finding ways for the game to close at any time the user desired and helping Jack make the game cleaner and easier to play.

What I learned –

This project was my first group project of this size other than my final year project during my undergraduate which was done in pairs. One thing I learnt was that some apps and software are extremely useful when working in groups, github made sharing our work and ensuring we weren't doing the same piece of work as someone else very easy. This was evident when the members not uploading to github ran into this issue. I also like the use of whatsapp to discuss issues and to

organise proper meetings. I also learned that pair programming with someone avoids a remarkable number of errors, the person watching doesn't usually let silly mistakes slip past both people, also it takes a lot less time to find solutions to problems and keeps both people motivated on the work. I would definitely recommend this technique to others considering a group software development project.

Charles Crawford

My personal contributions to the project were mainly based on the graphical side of it. I learnt SDL first with another team member and we began to understand how to make windows appear, etc. I personally produced the attack animations, and was aided by another team member to integrate it into the program. As well as this I did the majority of the graphic design; which included the level backgrounds, and the fireball/iceball/lightening balls – which were multi-image sprite sheets. I also contributed during the initial ideas phase – however I believe everybody contributed equally for that.

I felt that whilst the project as a whole was very successful, there were some major parts which went not so well. The general efficiency of getting a piece of work done was very poor at times, as two specific individuals slowed us down with their non-cooperation. In particular I feel if communication had been better and some people have cooperated more we could have have a working map and health bars to go with the game.

I am happy with the project as I learnt how to code in C better, how to use SDL, and having never used GIMP before, I can now confidently say I am fairly fluent in its use. This was my first experience working on a large scale group programming project, and was very happy with the outcome. While my programming skills are not excellent, I felt my coding inputs were met with support and knowledge from other team members, and anything I did code would be improved (via pair programming) and used in the end.

Jack Williams

Each member of the group contributed significantly to the overall product, and therefore I am happy to agree with the equal weightings. However, our group was reduced to 5 members following disagreements with one particular member in November 2015, therefore individual workloads were required to be increased to counter this problem; something that was achieved successfully. In addition, the group has not heard from one further member since December, and we believe he has left the course for personal reasons.

My contributions to the project included initial brainstorming of concepts and ideas, design and development of the logic side of the game, organisation of team meetings, creation of graphical sprites and general debugging. Almost all of the contributions I made to the group were achieved during pair-programming sessions with various members of the team; initially with Liam during design of the State Machine and Module design and writing the game logic, and more recently with Alex during rendering of sprites and backgrounds. Individual work included the creation of the graphical sprites used as the hero and enemies, which were all open-sourced and edited by myself.

Although being used to working in teams from my undergraduate studies, this was the first opportunity to design and collaborate on a working software product over a significant time-frame; an opportunity I was very excited to be involved with. Although I found it difficult at first due to my lack of experience in programming, and specifically programming with C and SDL, the project became more enjoyable and satisfying upon successful completion of sections of the software. A steep learning curve for myself was de-bugging following integration of multiple pieces of code, for example integrating the graphics with the logic parts of the game. This proved to be the most

intensive part of the programming, but was achieved successfully. This was minimised by our use of test functions for each module, which greatly increased the efficiency during this process.

Communication has been a major problem within the group, with consistent disagreements with one member of the group who left in November 2015. From the offset, our progress was hampered, and in general felt the efficiency of our team meetings in terms of content was greatly reduced due to these disagreements. We felt that writing a formal complaint against this member was appropriate in order for the rest of us to continue with the project. However, the project progressed well regardless, due to the communication between the other members, with WhatsApp, FaceBook and face-to-face meetings keeping everyone up to date with developments.

The experience gained from such a group can only be positive in the long run, for example how to effectively deal with disagreements in a diplomatic fashion, and working to multiple deadlines under increased pressure from reduced team members and other work commitments. Therefore this can be seen as a great opportunity to experience these pressures before working in a career in Computer Science. The way our project has been created and written means that I have gained a greater understanding of working in a modular format, keeping code straightforward to read and understand for others who might adapt or change it.