Drew Liesching-Schroder

Candidate Number:

OPERATION: CYBER_SECURITY

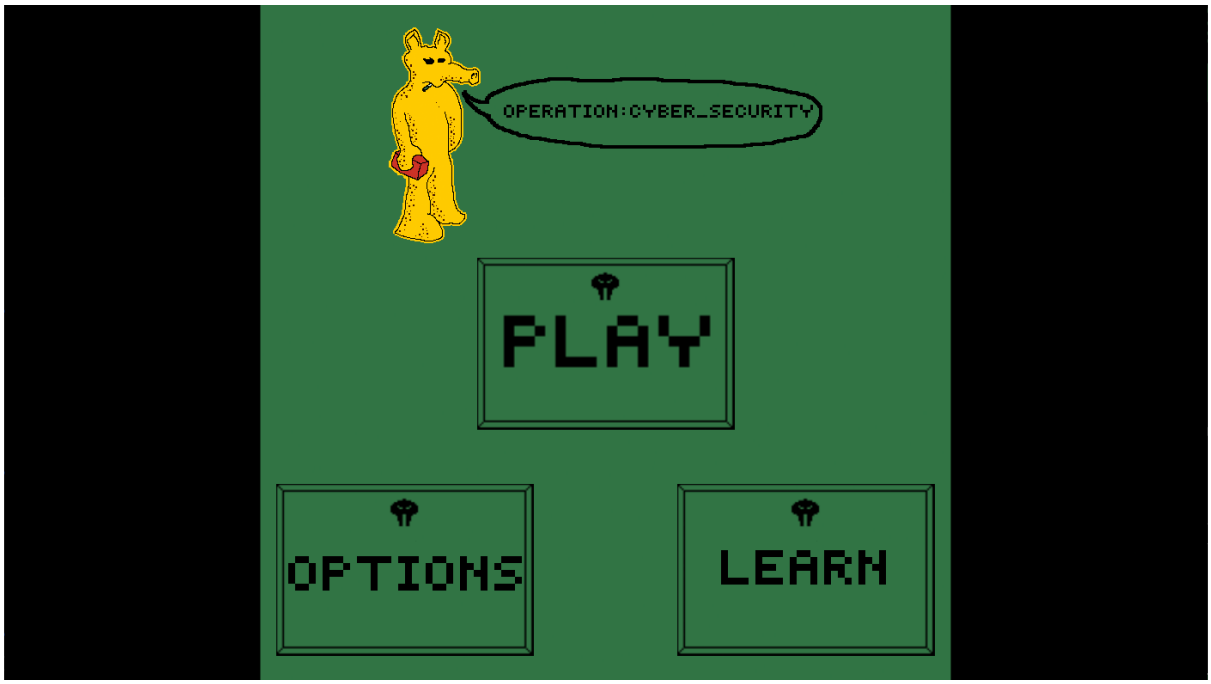
A Level Computer Science

H446/03 – Programming Project

The King Edmund School

Centre Number 16223

May 2022



Project title: Cyber Security / Internet Safety Game

Need to change circus game for another. Change some stuff about stakeholders.

Contents List

1. Description and Analysis of Investigation

1.1 Description of Problem

As the prevalence and necessity of the internet increases during this time of great technological advancements, so has the necessity for users to be educated on computer safety and the threats posed by malware. Tech Radar discusses the dangers of malware in one of their articles, going as far as saying the worst malware is “incredibly dangerous”. This is because the most dangerous malware can do a variety of malicious activity on devices such as compromise the user’s banking details to stealing and corrupting their data. Malware can get infect devices in variety of ways, but the most common is due to users carelessly downloading files, or by being too trusting online.

(ref no.1, Tech Radar, Darren Allan, 2020)

https://www.techradar.com/news/what-is-malware-and-how-dangerous-is-it

According to Tech News World, the most susceptible group cyber attacks such as fraud was those under the age of 25. Sutherland stated that the reasons for this were “tend to be more relaxed in their usage patterns and willingness to share personal data”. This quote likely suggests that young adults are so susceptible due to them not having possibly been educated enough that they are cautious while on the internet. Furthermore, it is a good idea for young adults to be educated about malware and internet safety because of malware threats that businesses deal with. Businesses frequently must deal with the threats of malware, especially ransomware; Cloudwards state 51% of surveyed business were attacked by ransomware in 2020.

(ref no.2, Tech News World, Jack Germain, 2021)

https://www.technewsworld.com/story/87059.html

(ref no.3, Cloudwards, Aleksandar Kochovski, 2021)

https://www.cloudwards.net/ransomware-statistics/

Evidencing the claim of young people being susceptible to malware and cyber crime are statistics stating: 9% of young people having personally been affected by malware and 41% of 11-15 stated they knew the risk of malware getting on their device through illegal streaming services. This supports my claims above that young people have not been well educated on the subject matter. This is because the statistics suggest that 59% did not know the risk of malware on unsafe websites.

(ref no.4, Internet Matters, Internet Matters Team, 2018)

https://www.internetmatters.org/hub/news-blogs/protecting-children-malware-resources-advice-industry-trust/

Therefore, a possible solution to the problem of young people being susceptible to malware and other forms of cyber crime is through educating. Educating would be an effective way of solving this problem, because those who are educated are likely to be more vigilant and take precautionary measures. This opinion is shared by WeArePennine.com, “understanding how different types of malware work can help prevent your computer and network becoming infected” as well as Google, ´You can protect yourself by learning what malware is, how it spreads and how to prevent it”. With all this educating appears to the best solution to this problem.

(ref no.5, WeArePennine, 2017)

https://www.wearepennine.com/it/malware-can-impact-business/

However, educating young adults can be difficult because most young adults will not engage with more traditional methods of educating such as notes. Therefore, I shall be attempting to remedy this problem by the untraditional method of gamification. Gamification is the process of taking an educational topic and teaching it through the medium of a game. However, a study posted on springeropen.com concluded by saying the “technology selected is easy to use and able to attract the students to increase student engagement in the classroom”. Therefore I have to make sure my game is easy to understand, so that engagement is high, otherwise students will not be engaged and not learn.

(ref no.6, springeropen.com, Rafidah Rahman, Sabrina Ahmad, Ummi Hashim , 2018)

https://educationaltechnologyjournal.springeropen.com/articles/10.1186/s41239-018-0123-0#Sec8

As suggested above, gamification engages student’s more than traditional methods of learning. However, engagement alone will not sufficiently educate young people about malware if gamification does not have improved retention rate. Another study concludes stating that gamification improving both retention rate and accuracy of the information gamified. Both retention rate and accuracy are further improved when a social element was introduced.

(ref no.7, Research Gate, Markus Krause and Joseph Williams, 2015)

https://www.researchgate.net/publication/274697615_A_Playful_Game_Changer_Fostering_Student_Retention_in_Online_Education_with_Social_Gamification#:~:text=We%20hypothesize%20that%20gamification%20has,and%2050%25%20for%20retention%20period

From the above research, I have concluded the best way of educating young adults is through the education being taught through a gamified medium. The game must be a simple

game with a social element as this will maximise engagement and retention of the provided education. This will be a superior method of educating over alternative traditional methods such as notes or work sheets. For the project to be successful in educating young adults the game must provide substantial information in a fun and enjoyable way, while still entertaining users. Therefore, it is critical that the game is both educational that it is an effective teaching tool but entertaining so that young adults play the game. This can be achieved by using a simple but fun game loop. This can be achieved successfully through making a genre of game that has a game loop that is simple to understand and quick to learn. For this type of game to stay fun and engaging, the game must be difficult enough that it is not boring, but easy enough to not be frustrating.

A genre that does this is a SHMUP / bullet hell game. This is because the game has a simple concept of surviving and defeating enemies, but the difficulty of the game is from dodging the incoming attacks. This is complex enough to fully engage a user as they must do two objectives at once to succeed. The way of educating in this game will be done through multiple choice questions, which reward the player with a temporary bonus. This bonus (whatever it maybe) will incentivise players to learn as the bonus will help them get higher scores. A social element to help with engagement and retention, will be done with a leader board, as players will want to try and attain the top spot. The easiest way this can be achieved is through learning the topic and using the power ups. Another reason a bullet hell is an effective game for learning is that the games in this genre only have a few levels or follow an infinite run model of gameplay. However playtime of these games are usually high due to engaging and entertaining gameplay and challenges (in game or ones the players set their selves) for the player to complete.

In summary, I shall be solving the problem of young adults being uneducated in internet safety and malware through a SHMUP / bullet hell, that will include multiple choice questions and a social element (leader board).

1.2 Stakeholders

1.2.1 Identification of Stakeholders

This section is for to identifying stakeholders invested in the project. First is important to clarify the meaning of “stakeholder”; for this project it will be any individual or party that has an invested interest of the learning and teaching of cyber security and internet safety (this also includes the prevention of this). The identification is important because they have needs that should be considered. These needs will shape the project into something purposeful to meet the needs of the users. The main stakeholders are young adults, parents and cybersecurity people. Teachers and hackers / scammers are also stakeholders, but lesser ones.

Young Adults (13-18) (Target audience)

The target audience is an obvious stakeholder in the project, as they will be the people playing the game. This stakeholder group are young adults (teenagers can use this as well) aged 13-18 and will have little knowledge on malware, as this project's aim is to inform and educate. No one who is knowledgeable on the subject will have much interest in playing the game. Stakeholders should not be required to be competent gamers to be able to play and learn, due to the use of a simple but fun game loop. The game should be simple to learn and understand, as a complex game will alienate most young adults. The project should be focused on having as large of a target audience as possible, meaning the game will have to be simple and provide basic information introducing concepts and information. The information may potentially start to get harder as the information taught will get more complex and thorough (however this will only be done if I am able to, it may end up just being random). This group of stakeholders will make use of product by playing the game and learning about the dangers of malware. The game will be simple enough for a wide audience to access and will serve the user facts and content regarding malware. The users will be engaged in the project as they will be playing the game, while learning and competing for high scores on a leader board system.

Parents / Guardians

This group of stakeholders are key to this product. Despite not playing the game, this group will have a stake if their children are playing it. The key things parents will want is the game to provide accurate and clear information, while being fun and not being addictive. Parents will not want their children to be misinformed; therefore, it is imperative for there to be enough research. If the game is fun, while being educational this stakeholder will use this to educate their children. An educational game is appropriate for their needs as games are good for educating as they improve retention rate and understanding of the content. The game will also need to have appropriate visuals and language used, as parents would not want their children exposed to visuals, language or themes that is inappropriate or unsafe.

Teachers

This is an unlikely stakeholder in the project; however, a teacher may decide to use the game as a way of educating students. Teachers would only do this however if the content were appropriate, accurate and successfully engaged most of the class in the education provided. If the game were not the things listed above, teachers would not use this game to educate. The game must be engaging, while being educational so that it can be used a useful teaching resource. If the game was to be used as a learning resource, it must have enough information and potential to thoroughly educate a class for a full lesson. If there is enough information, it might not be worth using, so a teacher may find an alternative teaching resource.

Cybersecurity

Many cybersecurity websites write about cybersecurity for children; however, the information is presented as long blocks of text. Due to the dull and boring way the education is presented, it may be hard for young adults to be fully engaged, while reading. Cybersecurity

companies and other young adult educational websites may become interested in the project to teach younger people. This will only happen if the game is playable on a website and can engage and educate young people who click on the website. This game can be the solution to engage the young audience, who click on the website looking to be educated about malware and general internet safety.

Hackers / Scammers

Some hackers and scammers may become a potential stake holder, as they may want to prevent the better education of internet safety. A reason for this is that better informed individuals will make hackers / scammers attempts to steal data or spread malware will be less successful. If my project was embedded on a cyber security website, such as kapersky, then hackers and scammers stake in the game will increase.

Realistically the significance of my project to hackers / scammers is inconsequential, therefore the threat they pose to the game is extremely small. Therefore, their stake in my project is insignificant, however should be noted.

1.2.2 Stakeholder Requirements

This section is dedicated to the identification of stakeholders and evaluate the requirements and how these will shape the project.

Target Audience

Informative: It is important for the game to deliver accurate and a plethora of education around the topic of cybersecurity and internet safety. This is because the purpose of this project is to inform and educate. Therefore, the stakeholders will require the game to deliver ways of learning the topic in depth and revise what they have leant. It is critical for the game to prioritise the educational material over the game itself, as users will be seeking the game to learn about the educational topics covered rather than the gameplay itself. Furthermore, the information provided must be accurate so that the understanding of cybersecurity and internet safety is not hindered by playing this game.

Re-playability: The game must be re-playable so that users play the game many times. This is so they can revise and consolidate their understanding. This will allow for users to be able to remember the safety information long term. However, I shall have to be careful when making the game not to make it addictive. This is so that parents and teachers will still want to suggest the game to potential users, as many of this type of stakeholders may have negative opinions surrounding video games and maybe sceptical of its benefits, even though its purpose is educating users. Therefore, I shall have to assure the game is entertaining enough for users to want to replay, but not so much it is addictive. To summarise the game must be enjoyable that people will want to play, but not to play the game too much.

Social: There should be a social element to the game. Despite the game not offering an online multiplayer mode, it can have a leader board which displays real scores attained by

other real-life users. This gives the game an element of competitiveness, which is desired by many potential users, especially by those who already play video games. This is seen in the real world by the drastic increase of relevance in e-sports. This is evidence of the fact people who play video games enjoy competing with other people.

Satisfying: There should be elements of the game that makes the gameplay satisfying. This means the inclusion of shooting mechanics that are enjoyable to use, consistent movement as well as small details that add to the experience such as: audio and visual ques to indicate collisions. All of this will improve the overall experience and enjoyment for the user.

Parents

Parent would want to make sure the game is appropriate for their children.

Appropriate: parents will want to make sure the game is visually and content appropriate for their children. Images that are too graphic or sexual will be images that parent will not want their children to see. Content wise, the game needs to be appropriate as well, so specific details of the certain cyber threats should be avoided. The game should be visually appealing and fully inform the users without scaring them with unnecessary details or scary images.

Beneficial: This should be a useful tool for parents to use to educate their children as they start to use the internet more as they grow up. Therefore, it is important that the game is quick to provide users with information that will be beneficial to them early on. This is especially true if the parents are looking into this game, as they want to see and understand the benefits that the game will have. If the benefits do not appear quickly and clearly, then the parents may be reluctant to introduce the game to their children.

Simple: Not every parent is proficient in technology such as computers. This means the game should be easy and simple to understand and set up and be as accessible as possible. This will increase the use of this game, especially as parents who are not good with computers will likely not be able to help educate their children and may have to rely on this as a substitute.

Teachers

Teachers be the same as above (especially Parents), but they would also the following:

Playability in the classroom: Teachers be showing this content to children on school computers, which are unlikely to have large amounts of RAM and have a good CPU and GPU therefore the game will need to be able to run on simple hardware that found at a school. The game needs to be reliable and be able to run without constant crashing. If the game cannot run without crashing, then the game will not fit this stakeholder needs. This is because if the game is not reliably stable, then the lesson will be disrupted, and the teacher may not use the game again.

Hardware limitations: Due not all schools having good hardware and peripherals such as keyboard and mouses the game should limit itself to as few keys as possible and allow for the changing of key binds in settings. This will allow for all students in a class to be able to play the game while in the classroom.

Cybersecurity website

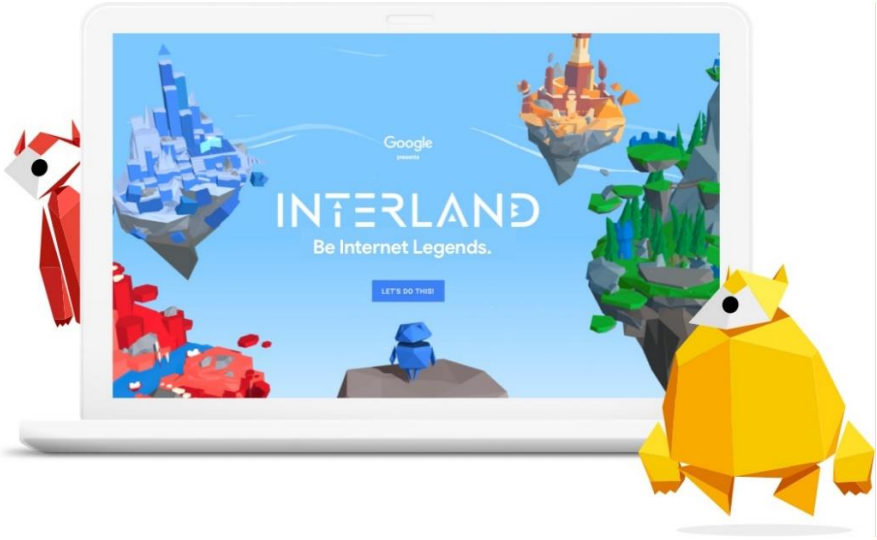
Varied and Relevant Information: If this game was to embed or referenced on a website teaching about cybersecurity and internet safety, the information provided on the game should only include relevant topics. Revision topics should not include information that was is not commonly displayed as users may become frustrated if they answer a question incorrectly (currently questions answer correctly will provide the user with some sort of benefits such as health or increased damage).

Ease of sharing: if the game becomes some use in educating about the dangers of malware, then they may want to use it on their website. The easiest way of doing this is through embedding it into their website. Therefore, I need to consider using code and a game engine that can be embedded into HTML. Alternatively, a link to the game or a download link can be used. If a link to another website was used, the code would still have to be embeddable, however if a download link was used, the game should run when the downloaded file is ran.

1.3 Research

1.3.1 Existing Solutions

Be Internet Legends – Google game
(https://beinternetawesome.withgoogle.com/en_uk/interland/landing)



Overview

This game made by google covers four important parts of internet safety: checking the validity of information and messages; thinking before sharing anything online; keeping personal information such as passwords safe; and internet kindness and how to block cyber bullies.

Each topic is given its own unique game which the user can play, as well as given questions at the end of the game to consolidate the information littered throughout the gameplay on the subject matter.

Despite the games being informative and effective in serving its purpose of being educational, the games lack re-playability. This is due to the same questions being asked each time and the gameplay being too easy. This leads to gameplay being boring and only serving as a medium for children to be educated in a concise way. Consequently, the game is effective at teaching the basics of the topics quickly, however it lacks any replay value and provides lacklustre engagement.

Features

The standout features this game has over others are the graphics and animations. The 3-Dimensional game and smooth animation are the most notable feature as it is more engaging than a static screen.

Another feature is the use of themes. Each topic explored in the minigames are the unique areas the games are played in each having a different colour pallet set to each. This is helpful as it helps separate the information discussed. I believe this will be helpful for younger users as they are able to better distinguish between the ideas of the games they would have just played.

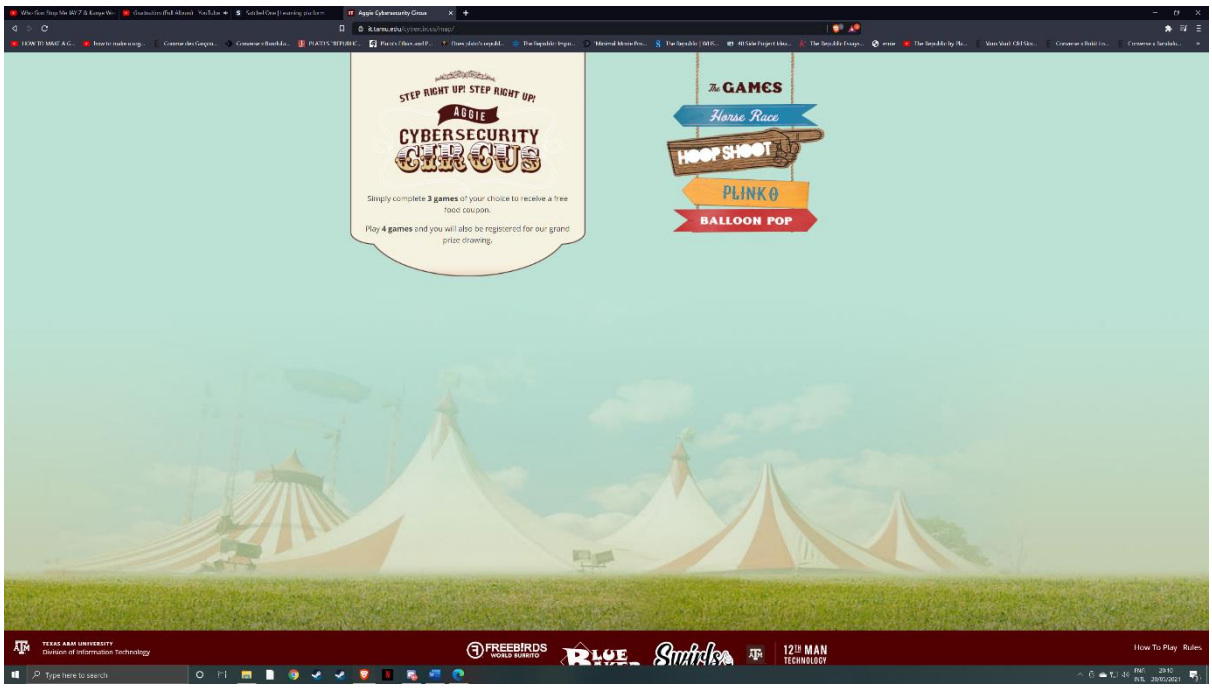
At the end of each game a mini quiz is played. This is an effective inclusion as information was covered are gone over again as quick way to summarise the key points discussed in the game.

I do not have the capability to effectively implement 3-Dimensional graphics into my game, however I shall attempt to have dynamic visuals in my game. This will include an animated background for example, as this will make the game more engaging and aesthetically pleasing.

My game will not be built from several mini games; however, I like the idea of keeping a colour pallet for most of sprites/textures. This will make my game look more coherent and better designed.

I will implement the method of asking revision questions within the gameplay. Internet Legends use questions frequently throughout their game, with one of their minigames being a glorified multiple-choice quiz. I don't believe the inclusion of forcing players to answer questions within the game was beneficial to the gameplay but retracted from it. Therefore, I will attempt to implement a way of asking questions to test the player without interrupting the flow of the game as this can be frustrating. I believe the best way of doing this is through creating a sprite, which when collided with will display a question similar Internet Legends. The game could potentially be played without answering questions, however due to the immense benefit that will provided when a question is answered correctly, players are highly incentivised to answer them correctly. This is especially true, if they attempt to attain a high score, which will be encouraged through a leader board.

Cybersecurity Circus (Aggie Cybersecurity Circus (tamu.edu)) can be deleted if too



Overview

The aim of Cybersecurity Circus is to raise awareness and act as an introduction to the risk of malware and common ways of malware getting on to user's devices. The four minigames are made for university students in Texas, however, is playable and accessible for everyone.

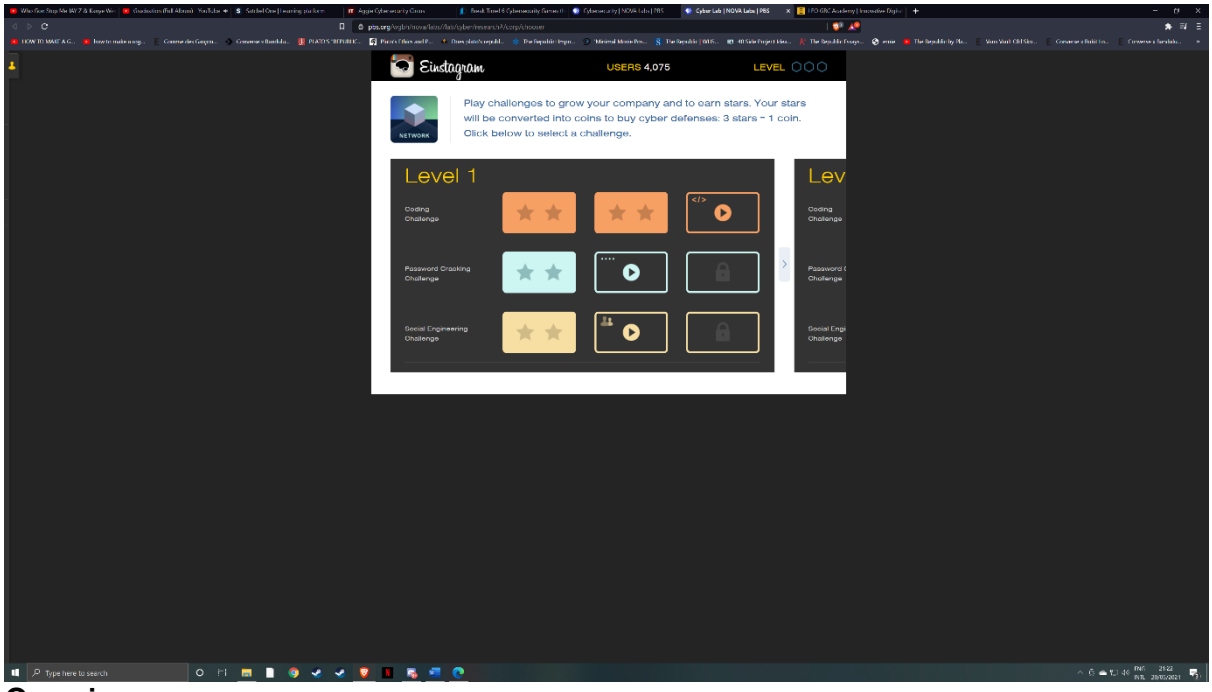
The games are quick and easy, likely designed with the purpose for university students to quickly complete the games. The game consists of three quizzes with unique scoring systems and an anagram game. By the end of the all the minigames, users should have learnt some basic practical information.

The websites aim is to educate the users of how malware can get onto their devices as well as basic safety in a quick way. This game is successful in doing this; however, my aim is for this game to better educate through more information and provide a more engaging experience. This is because Cybersecurity Circus falls short due to the minigames only having 1-3 questions each, and forces users to play through all the games before they can replay a certain minigame. This can be frustrating if the user has a preference of game. The scoring of the games is also lacklustre, due to there being a lack of a leader board. In the research done at the start, it states games with a social element is more engaging and results in users being able to better retain information from the games. Consequently, this game is good in providing quick facts, but fails to be a useful revision resource or effective educational tool.

Features

There are extremely limited features to this game as the three of the minigames are identical except from the visuals and scoring method as they are just multiple-choice questions. The other game is an anagram game, but there are only 2 possible anagrams. The game is simple but lacks any long-term replay-ability. For my project to be a successful way to educate long term it should have useful information presented simply, but a lot of questions are needed, and the games should be more enjoyable to play over longer periods of time.

Cybersecurity lab (https://www.pbs.org/wgbh/nova/labs/lab/cyber/)



Overview

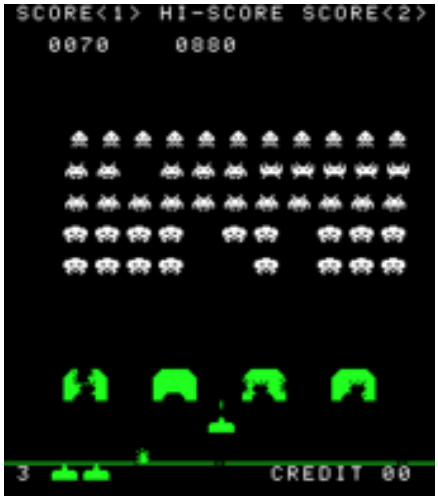
The aim of cybersecurity lab is to educate users of cybersecurity. In this game there is a main story where you own a social media platform, without any cybersecurity. Similarly, to the last game, this game also consists of a variety of different minigames. However, this game's minigames vary from each significantly. This game includes basic coding exercises like scratch, a password game, and a phishing game. These minigames provide an in-game currency to provide defences against the main story of the game. The main game is a defence game, where you buy different protection against hackers. Users can then see how successful they are as better they do the less users they will lose.

This game is successful in educating children about malware and cybersecurity, through use of characters to give the game a narrative, as well as large mix of games. The variety of educational games used to make this game keeps users engaged as they can change games when they start to feel bored of one game. Due to there being a story and minigame levels to complete, users are more engaged as they have a lot of content to complete.

Features

In this game there is a main story with its own game mode and minigames referred to as challenges for users to complete and learn from. At the start of the game, users get to choose a character and social media platform from a small selection. This feature gives players some sort of choice, which can further engage them, as they are more invested in their social media platform. The use of users is a clever way of keeping score and showing how successful the user is doing rather than the score just being referred to as score. It also makes it make sense when score is lost, as poor security would make user stop using the app in real life. Having a punishment for having poor security incentivises players to play smart and think about what they have learnt and properly prepare by playing the minigames. Furthermore, the use of a narrative through fun characters is engaging as it makes the game not feel lifeless and boring. The use of challenges and a narrative with characters are good features, which I am going to use in my game.

Bullet hell / shmup genre (https://en.wikipedia.org/wiki/Shoot_%27em_up)



Overview

SHMUP (“shoot em up”) games are a sub-genre of shooter video games that were popular in arcades. In these games the player must attack the enemies while trying not to be destroyed. Players can be destroyed through a variety of ways including enemy projectile attacks such as lasers, or more simply the enemy getting to close such as in space invaders. However, all games in this niche genre includes players goal being to destroy as many enemies as possible before them their selves are destroyed. Typically, SHMUPs do not abide the laws of physics such as being able to change direction with no inertia and projectiles will move at constant speeds.

Features

Feature of these games have enemies attacking in ways such as shooting, which I will include in my game. My game will more closely resemble a bullet hell game, which will have enemies shooting complex patters of projectiles at the player, meaning they will have to be able to dodge the incoming attacks, while also having to fight back. (This style of game may be frustrating, as these games are typically hard; therefore, an idea maybe a difficulty system where players can choose how hard the game is. Harder the difficulty can have a higher score multiplier. Another possible solution to prevent this game from being to frustrating could include code, which slows enemy attacks for example if the player is consistently getting low scores or increasing the strength of power ups. The latter solution could lead to exploitation though, as good players can get some low scores before trying, which could lead to better runs.) Other features include a: health system, showing how many more hits a player can take before being destroyed; point system, awarding points for time alive an enemies destroyed and round system, a way of the game getting progressively harder as the game continues. A final feature this genre sometimes include are bosses.

My game will include all the features in this genre spoken above. One feature that will be important to include is a boss, as a class can be used. Bosses are generally more difficult than other aspects of a game, therefore can be extremely satisfying for a player to beat. Including features such as a boss, score, and a good difficulty curve are important for maximum engagement and replay-ability.

1.3.2 Essential Features of the Proposed Solution

Educational

The sole purpose of this game is to educate young people long term with this game about cybersecurity and malware. Therefore, an essential feature will be that the game effectively educates users. The knowledge should be reinforced, through questions asked throughout the game as well to make questions they answer incorrectly to come up more frequently. The information should be accurate and relevant so that all stakeholders see some value in the game.

Replay-ability

The game loop should be fast once the user has gotten an understanding of each enemy type and each defence. There should be a tutorial, which will teach the basic gameplay. After that however the game should be very quick, with a key dedicated to restarting the game once it has been won or the user has been defeated. By the game having a quick way to get back to playing after a game reduces the dead time, allowing for more education.

Accessibility

The game needs to be simple to understand and play for it to be accessible to as many people as possible. This is so that many people will be able to understand and enjoy the game, leading to more people learning from it. The information provided should also be simple enough for all the target audience to understand. Furthermore, if the game is overly complicated to play users may not learn from the game as they are more focused on trying to understand what to do. By the game being simple it is accessible to the majority of people, not just those who are adept at video games of the SHMUP genre.

Accessibility – Colour blind

For the game to be as accessible as possible, it is important to account for users who have disabilities. The game should provide colour blind options so that all users are able to clearly visualise all of the game. If users are unable to clearly understand the game, they will not be fully engaged. This will be especially true if users cannot clearly see enemy attacks or enemies and start to lose health without understanding why.

Accessibility – Text

Some users maybe visually impaired and will struggle to see the font on the menus and the health and score. This may take away from the enjoyment of the game if they are unable to clearly see the user interface. Furthermore, they may struggle to answer the questions, which will reduce revision potential. Therefore, there should be an option to increase font size or there should be big font as the default size.

1.3.3 Limitations of the Proposed Solution

Time

A significant problem is the lack of time. Games take a lot of time to fully code, polish and coded to be as efficient as possible. Due to only one person coding the entire game, the game will have negative consequences such as bugs, lack of levels etc. To make the game as high quality as possible the scope of the project should be restricted. It is important to prioritise certain aspects over others such as having no bugs and that the game is playable with no crashes or errors. It is important to be realistic and properly plan and organise my ideas. One way of effectively using my time is using a language such as C++. This is because C++ is an object-oriented language, meaning I can set objects and container classes. This would be time efficient as I would then be able to reuse large amounts of code.

Programming skills

Another limitation my project will face is my inexperience of coding using an object orientated language such as C++. Currently I only have experience using python, therefore I will be learning C++ while I am completing the prototypes of my game. Due to my lack of knowledge of the language, my project may suffer from poor coding practices; this may lead to issues such as memory leaks or logic errors in my code. I can attempt to solve this limitation by reworking my code once I have done most of it, as well as practicing before I start my code. This would involve me learning about concepts found in object orientated languages such encapsulation and inheritance.

Art Skill

The aesthetics of the game are more significant than most people may think. This is due to the appearance of the game will be first thing users will see of the game. If the appearance of the game is not too a good standard, then potential users may have a negative first impression and may decide not to play the game. This is bad because the purpose of this project is to educate as many people as possible, but if the game is not played then the project fails. Therefore, it is important to attempt to remedy this issue by using an art style that is easy draw such as a pixelated style and through practice.

Solving the problem

The purpose of this project is to educate young adults about the potential threats on the internet as they start to use it more. However, I must find a way to make the game entertaining and appealing for the target audience to play, while still educating them in an effective way. This can create issues as they may be reluctant to learn if I cannot integrate the education in an entertaining way. I can attempt to solve this issue through implementing an incentive for the users to answer questions within the gameplay. This will have to be tested heavily by play testers to assure this method is not too unbalanced in the game or becomes too egregious, that it negatively affects the gameplay.

Scope

I need to consider the overall scope of my game. Due to the other limitations discussed above the scope of my game needs to be considered. Therefore, I believe that I need make the game use an infinite run game model rather than levels. This is because levels take a lot longer to code over other ways. Potentially I could code a single level as proof of concept, however I believe that this would not be satisfactory for the stakeholders.

For the infinite run gameplay, I should implement a steady difficulty curve as game is played. To do this I will have more enemy variants spawn in as the users score increases. This will provide more enemies and a more difficult time. The use of bosses would be an effective way to provide difficulty spikes as the game is played; this will prevent the game from becoming tedious and immediate challenges. The additional challenges will encourage the use of players to use power ups and force users to learn to maximise their scores.

Rewards could also be implemented. With this inclusion I could encourage users to play for longer as they will want to unlock rewards for different challenges that I set. This will then lead to the consolidation of knowledge, and the completion of my project’s goal. However,

this feature will be implemented late into the game's prototype and may not be coded at all due to the time restrictions of the project.

Gamification

I will need to consider the quality of education provided to the end user, through using gamification. It is critical to consider that the goal of the project is to educate. Therefore, the emphasis of the game is the educational value it provides. However, by choosing the medium of a game will limit the amount of education provided in comparison to alternate solutions. Furthermore, the inclusion of education may take away from the enjoyment of the game, if not done in an effective way. It is important to try and effectively balance

1.3.4 Hardware and Software Requirements

Storage

Space invader type games range from approximately 10 MB to 260MB on the app store (ALSTEROIDS by RI productions and 1945 – Airplane shooting games by ONESOFT GLOBAL PTE. LTD. respectively). My game should be around 20 MB – 50 MB; this will be due to the basic nature of the game and simple code. My game will be a lot smaller due to there being less enemies, no sound, less levels etc.

CPU

Despite there being a wide range on the complexity of games in this sub-genre of games, they are still playable on mobile devices. Therefore, it can be assumed that the CPU and the necessary hardware can be basic. Hardware such as an i4 single / dual core should be able to run this game successfully. Most modern school computer will include this hardware, if not better; therefore, will be able to be played in school with the purpose of educating students. Touhou 15 can run on a Core 2 Duo processor according to Steam. My game will be a lot simpler, it is therefore reasonable to assume that the game will be supported by most CPU

RAM

Most modern-day computers have at least 6 MB of RAM, if not more. This is enough RAM for most 2D games to be played well. Therefore, I should make sure the game is able to run without issue (such as stuttering or frame rate falling). This should be manageable if I deleted the enemy's and player's bullets and the enemies when they are defeated. By not deleting enemies or bullets when they are killed or left the window, the computer will still consider them to exist. Without use of deletion, thousands of entities will exist at once leading to performance issues. A modern Touhou game can run with 1024MB / 1GB of RAM. All modern computers should have more than this available, so therefore a lack of RAM should not be a concern.

Operating Systems

My game will be optimised to run on the Windows 10 Operating System. Most computers in 2021 now use Windows 10 as their Operating System so this is not a problem. Running this game may be possible for earlier iterations of window, but I don't think this will be possible to

run on other operating systems such as Linux. This is because the game will be coded to support Windows and not made to support other Operating Systems. Windows is the most popular, so I can reach a larger audience by making it for Windows

Browser

As said above, this game should be able to be embedded into a website. Due to the game being simple, most browsers should be able to run this game. The browser can be left up to a user's preference. However, most popular such Google Chrome, Microsoft Edge and Brave will be able to run this game. This is only a concern if users play the game on a browser instead of downloading it from Github or itch.io

1.4 Success Criteria

Success criteria	Measured	Justification
Simplicity	Challenges are simple. Use as few as buttons as possible. Reduce as much visual clutter as possible	Simple games are easier to understand and make it more accessible.
Accurate Education	The gameplay has educational elements like questions that have been properly researched and cross referenced.	The whole purpose of this project is to educate users. Therefore, this game must have a way of educating effectively. Additionally, If the game does not have accurate information for educating young adults, this project fails as it will spread misinformation.
Fun	Survey asking if the game is enjoyable to play.	If the game is not fun to play, users may not be fully engaged in the game. This will lead to reduced retention rate.
Engaging	Survey people Leader boards Quick and fun	More engaging games, as discussed above, improves the player's retention rate and will be the best way for them to be educated using this medium.
Accessible	The number of ways it must have ease of use	The game having accessibility features will help spread the information this game will have. This is because a wider audience will be able to play this game because it will cater for their needs.
Appropriateness	Survey key stakeholders	The game will have to use language and have art assets to be appropriate. Any controversial ideas will also be avoided. If the game is not deemed appropriate for schools or for young adults to play by schools or their parents, the game will struggle to be played as parent and teachers will be preventing the target audience from playing the game.

Informative	Having many questions that are used Having random facts drawn to the menu screen	Due to the nature of this project being educational at its core, it is important that there are several questions and facts constantly being shown to the users.
Goals / Challenges	Challenges to complete Leader board	This is an extension of engaging and can be achieved just through use of leader board. However, by including some challenges or overarching goal / challenge, users may become motivated to play for longer.
Gameplay	Survey from play testers	The quality of gameplay is paramount to the project's success. If the game lacks responsive and quality controls, then users will not use this project to learn from.

2. Design of Proposed Solution

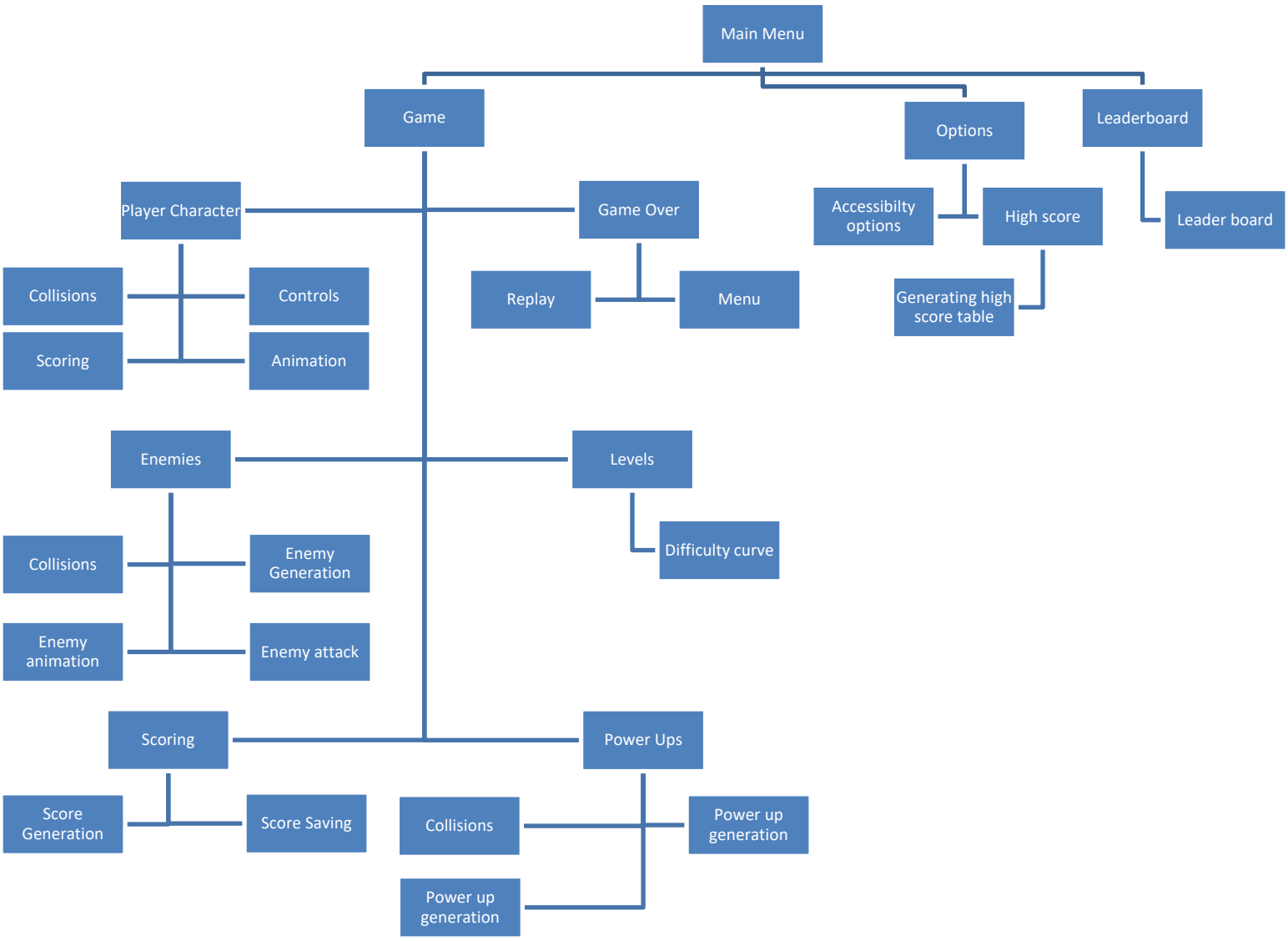
2.1 Abstraction of Problem

In the attempt of making the successful in accomplishing its main aims, specific core elements need to be addressed:

- **Education:** the main way my game shall educate users is with multiple-choice questions. Currently, these multiple-choice questions will appear when the user collide with a power up (only being able to gain the affects if the question is answered correctly). Detailed notes of malware and general computer safety rules may be accessible somewhere in the menus. I am hesitant of this though as I do not want to force notes on to users, as notes are not engaging to read, and therefore will likely not be retained by the average user of my game. An alternate solution is providing a random fact at the end of each run of the game as well as in the different menus. Littering the information around the game will allow for users to gradually learn in a unique way without overloading them with information.
- **Gameplay:** for basic gameplay to reach the fundamentals of the SHMUP genre, these features will need to be implemented:
    - **Playable character:** it is important for the player to be able to move their sprite smoothly and can easily navigate through the barrage of enemy bullets. Movement is a key part of all SHMUPs because dodging enemy attacks are just as important as shooting back at the enemies.
    - **Enemies:** enemies will need to spawn off the screen and gradually move down into the window. Enemies should have different movement patterns and speeds so that they do not become too predictable and boring to play against. To clearly represent this each enemy should be given a unique texture that are easily distinguishable from each other.
    - **Enemy Attacks:** this is a quintessential part of the bullet hell / SHMUP genre. Each different enemy type should have their own unique bullet pattern that they fire. The attack patterns should vary in speed, damage, and rate of fire so that every enemy feels unique. This feature should be heavily play tested to make sure users are able to avoid all attacks, while still providing a challenge.
    - **Power-ups:** power ups will be the main source of education in this project. Power ups should spawn randomly on the screen in the player's half of the screen. This is so that players can safely collide with the power up, rather than losing a good run due to unfair positioning. When the player sprite collides with the power up a multiple-choice question should appear on the screen for the player to answer. To assure that the flow of game play is not drastically disrupted, the question will be timed. If answered correctly, the player will receive a bonus. I have not decided what this will be, but I think it should reward some bonus health or maybe delete all enemies and their attacks off the screen. It its important the benefit outweigh the risk, so that players seek out the questions. If the question is answered incorrectly, nothing will happen; it is important not punish the player for incorrect answers as this may dissuade them from attempting to answer the questions.

    - **Collisions:** when the player collides with an enemy attack, collides with a powerup or when an enemy collides with a player attack the respective event needs to be triggered. Without collisions working properly or fairly, the game will not play as intended as nothing will work, for example score will not increase based off enemies destroyed; player will not lose health and therefore not die; the game will go on forever until manually quit by the user. Due to all of this it is essential for collisions to work.
- **Goal:** including a way for users to be able to track their own high score as well as other users will improve engagement and replay-ability. This is because players will likely want to replay the game multiple times to do as good as possible. Due to power-ups requiring correct answers from multiple choice questions it will further encourage them to learn the correct knowledge to have the best chance of maximising their score. This creates an overall goal for users to try and reach. Some form of challenges may also be introduced, however this would only be added at the end if I have not ran out of time.

2.2 Structure of Solution



2.3 Algorithms of Solution

I think I did most of it

Main Menu

Set Menu:

When keyPress == “1”

    mainGame()

When keyPress == “2”

    Options()

When keyPress == “3”

    leaderBoard()

When keyPress = “ESC”

    quit()

When the code initialises and the game starts, the first screen seen by the player will be the main menu. The main menu is the menu where the player will be able to traverse with ease to find where they would like to do. Currently, I plan for the player to be able to navigate the menu with the use of number keys, however I may consider changing or adding alternate methods. These methods may include the use of a mouse, arrow keys or even WASD keys. If I decide to implement the use of a mouse as way of navigating, I will have to use buttons.

Game loop

```
//likely in menu file

Game gameLoop

gameLoop.run()

//in game.cpp file

void Game::run()

    while window isOpen()

        updateEvents()

        if player health > 0

            update()
```

render

the game loop will start when the player inputs the correct keys to start the game. In the game loop the update and render functions will continue every tick. This will continue until player health is = to 0, then the game will stop updating. I will then implement the pseudo code at the end (which allow for the game to quickly restart again, score to be taken etc.)

Player

InitialiseVariables()

Speed = float value

Attack cooldown max = float value

Attack cooldown = Attack cooldown max

Hp max = float value

Hp = Hp max

InitialiseTextures()

playerTexture.loadFromFile(“Texture/playerTexture”)

InitialiseSprite()

playerSprite.setTexture(playerTexture)

Player Constructor()

playerSprite.setPostition(x,y)

InitialiseVariables()

InitialiseTexture()

InitialiseSprite()

PlayerMove()

Float x = 0

Float y = 0

x = square root($x^2 + y^2$)

y = square root($x^2 + y^2$)

if keyPress == W

y -= speed

if keyPress == S

```
            y += speed
        if keypress == A
            x-= speed
        if keypress ==D
            x+= speed
        move(x,y)
SetHp(const int hp)
        hp = hp
loseHp(cosnt int value)
        hp -= value
Boolean canAttack()
        If attack cooldown >= attack cooldown Max
            Attack cooldown = 0
            Return true
        Return false
updateAttack()
        if attack cooldown < attack cooldown max
            attack cooldown +=  float value
render()
        draw(playerSprite)
```

In this section of code, I have set up everything I currently think I will need for the player. I first will initialise all the variables that the player will need, including health, movement speed and attack cool down. This will allow for player movement, for the player to take damage and for the player to shoot a projectile attack at a rate of fire that the game will be able to handle. The cooldown is especially important, because without it a bullet will be fired at every tick, which maybe too much for the average computer to handle. If it was too much to handle the game may crash or suffer from frame rate issues.

Next, I set a texture for the player from a texture file. This texture is then set to a sprite for the player. This means the player character will be an image rather than a shape.

The variables, sprite and texture are all then placed into a constructor.

Then I have code that will allow for player movement. When WASD are pressed, the player sprite will proceed to move in the respective direction. I use Pythagoras theorem within the movement, so that the player will not move drastically faster when moving diagonally. Pressing the key will increase and decrease the x and y direction by the value set for speed in the initialise variable's function.

I will then create variables for the player's health. Currently I am using functions for setting a new value for health and a function reducing health. There are no plans for the player to be able to regain health points, however this may change with stakeholder feedback if they believe the game is too difficult.

Attack cooldowns are then created. With these attack functions, if the cooldown is less than the max amount for the cooldown, cooldown will increase in value. This continues with each tick, until the cooldown variable supersedes the cooldown max. When this happens canAttack() function will be true and the player will then be allowed to fire.

Finally, the player character is drawn to the window.

Player Attack

```
PlayerAttack Constructor(parameters with data types to be used below set in here)
        AttackSprite.setTexture(Texture/attackspriteTex)
        AttackSprite.setOrigin(x,y)
        AttackSprite.setPosition(x, y)
        AttackDirection.x = x
        AttackDirection.y = y
        AttackSpeed = float value
updateMove()
        AttackSprite.move(AttackSpeed * AttackDirection)
updatePlayerAttackInput()
        if keypress == UP and player canAttack()
                attack.push_back
                        setPosition.x as player x position
                        setPosition.y as player y position
                        x direction, y direction, attackSpeed

updateAttackDeletion()
```

```
        if attack bounds < y 0.f
                delete attack
                attack.erase()
render()
        attack render
```

In player attack I set values in the constructors, using parameter. This is so they can be called in the attack input function. This function may not be used in the same file as the constructor. This is because errors may occur if I code it all in the player attack files, so I may set it all up in one file but create it the attacks and call the texture in the game files, with the use of pointers / references.

The attack move function will work by multiplying the speed by the direction. This can be read as attack speed multiplied by x direction, attack speed multiplied by y direction.

Attack input function works by checking if the cooldown is set to true and if the respective key is pressed. For every tick they are both set to true, a bullet is fired from the position of the player.

Once the bullet's bounds are smaller than y = 0 then the bullet will be deleted as it has reached the top of the screen. This is critical, as without deletion then the player attacks will continue to be created and will still be rendered off screen. This is bad as it can lead to memory leaks, framerate issues and will eventually lead to the game crashing as it is too much for the game to handle.

Enemy entity

```
Class enemyEntity
Protected:
        Sprite enemySprite
        Texture enemyTexture
        Unsigned int scoreCount
        Float enemySpeed
        Int enemyDamage
        Int enemyScore
        Int enemyHp
        Int enemyHpMax
        Boolean go
```

In this class I define everything I currently believe I will need for all my enemies for all the prototypes. I created an enemyEntity class with the purpose of defining the variables I shall be using in the enemy classes. Below I shall show off one example of an enemy class, however I plan to do many and use this class as a parent class that the other enemy classes will use to inherit from. This should reduce the overall size of the game as well as improve code efficiency.

Enemy

```
Class enemyA : protected enemyEntity

    InitialiseEnemyAVariables()

        scoreCount = int value

        enemyScore = enemyScoreCount

        enemySpeed = float value

        enemyDamage = int value

        enemyHpMax = int value

        enemyHp = enemyHpMax

    InitialiseTexture()

        enemyTexture.loadFromFile("Texture/enemyATexture")

    InitialiseSprite()

        enemySprite.setTexture(enemyATexture)

        enemySprite.scale(float value, float value)

    EnemyA Constructor() : enemyEntity()

        InitialiseEnemyAVariables()

        InitialiseTexture()

        InitialiseSprite()

    Const int getScore()

        Return enemyScore

    Const int getDamage()

        Return enemyDamage

    Const int getHp

        Return Hp
```

```
        Const int getHpMax()
                Return enemyHpMax
        Void setHp(const int Hp)
                enemyHp = Hp
        void loseHp(const int hpLoss)
                enemyHp -= hpLoss
        void updateMovement()
                if enemySprite y position <  float value
                        go = true
                else
                        go = false
                if (go)
                        enemySprite.move(0.f, enemySpeed * value over 1 to increase speed)
                else
                        enemySprite.move(0.f, enemySpeed / value over 1 to decrease
speed)
EnemySpawn()
        If enemyASpawn < enemyASpawnMax
                enemyASpawn += float value
                spawn = true
                enemyASpawn = 0
        else
                spawn = false
render()
        draw.enemyA(window)
```

First I have the enemies inherit the definitions from the enmeyEntity class, as all enemy classes will use these. The values of these are set within each of these classes. Polymorphism will be used in conjunction with inheritance to further reduce size of the total game and to increase code efficiency.

The texture will then be called from the texture file and set to a sprite. This will make enemies have the set image of the sprite.

The variables, texture and sprite functions are all placed into the constructors.

I will then need to code functions on for the enemies health, damage to the player and score. These functions will be used to allow the enemy to take multiple hits before being deleted (this is intended to increase difficulty), damage the player if the player's bounds collides with the enemy's bound and score which will increase the score when the enemy is killed.

I then have a movement function, which will use the Boolean go (that was set in the entity class). The Boolean go is used to change to the enemy movement. The example pseudocoded above shows the enemy move quickly down until a certain set point on the window, which will then change the movement to a slower speed.

I will then use similar code to the player's attack cooldown mechanic. If the spawn timer is less than the value given to spawn timer max, spawn timer will increase in value. This will continue until spawn timer is equal or greater than spawn timer max. When this is I will have it so that a new enemy is drawn to screen, and spawn timer value set back to 0 so the process can repeat.

EnemyShoot

Class enemyAttack

Public:

    Texture enemyAttackTexture

    Sprite enemyAttackSprite

    Float EnemyAttackSpeed

    Float EnemyAttackDirection

    Float EnemyAttackDamage

    Float EnemyCanAttack

    Float EnemyCanAttackMax

Class EAAttack : public enemyAttack

Public:

    InitialiseVariables(parameters)

        EnemyAttackTexture = EATexture*

        EnemyAttackSpeed = EASpeed (float value)

        EnemyAttackDirection.x = EADirX (float value)

```
        EnemyAttackDirection.y = EADirY (float value)

        EnemyAttackDamage = EADamage (int value)

    Initialise Move()

        EASprite.move(EASpeed * EADirection)

    updateEAAttackOutput()

        if enemy EACanAttack()

            attack.push_back

                setPosition.x as enemyA x position

                setPosition.y as enemyA y position

                sin(0.5)*function that increments every tick(x direction), cos(0.5)*function that increments every tick (y direction),
EASpeed

    update EAattackDeletion()

        if bounds.left <0

            EAAttack.delete

        If bounds.top < 0

            EAAttack.delete

        If bounds.left > windows width

            EAAttack.delete

        If bounds.top > windows height

            EAAttack.delete
```

I will likely start making an enemy attack class, which will act as a parent class. This is so that all enemy attacks can inherit from this. This is important as the attacks for each enemies will be different. For example for the direction in the enemyAttack output will differ for most of the enemies. This is so that different bullet patterns are created, for example pseudo code for the above direction should create a spiral due to the use of incrementing values and trigonometry values.

In the EnemyAAttack class I start by setting variables as parameters. This is very similar with the player shooting, with the exception of the position where the attack being drawn will be at the enemy's position and that no input from the player will be required for the enemies to fire.

Enemy bullets must be deleted if they reach any point outside of the displayed screen. This means bounds must be set along all sides. This is like player bounds, but instead of having the sprite's position set to not be able to get outside the screen, the sprite is deleted. This is to prevent memory leaks and the program from crashing.

Game Over

```
name = str(input("Type your name now"))

len = name.length()

While len > 10 and if ",", "!", "=" in name:

        Print("name may not contain symbols and must be 10 letters or less")

        name = str(input("Type your name now"))


File = file.open("highscore.txt.", a)

File.write(name)

File.write(",")

File.write(finalScore)

File.write(",")

File.write("\n")
```

This subroutine allows users to type their name into an input, which will be copied into a .txt file. There is a validation check which checks that no more than 10 characters have been used and special symbols have not been entered. This is because some special symbols can be used to manipulate the txt file allowing for scores to be manually entered by players. This will allow them to make the high score table show that they have a higher score than they got.

```
menuCheck = input("Press 1 to play again, Press 2 to return to main menu")

while menuCheck != "1" or "2":

        print("Enter a valid input")

        menuCheck = input("Press 1 to play again, Press 2 to return to main menu")

IF menuCheck = 1:

        mainGame()

else:

        mainMenu()
```

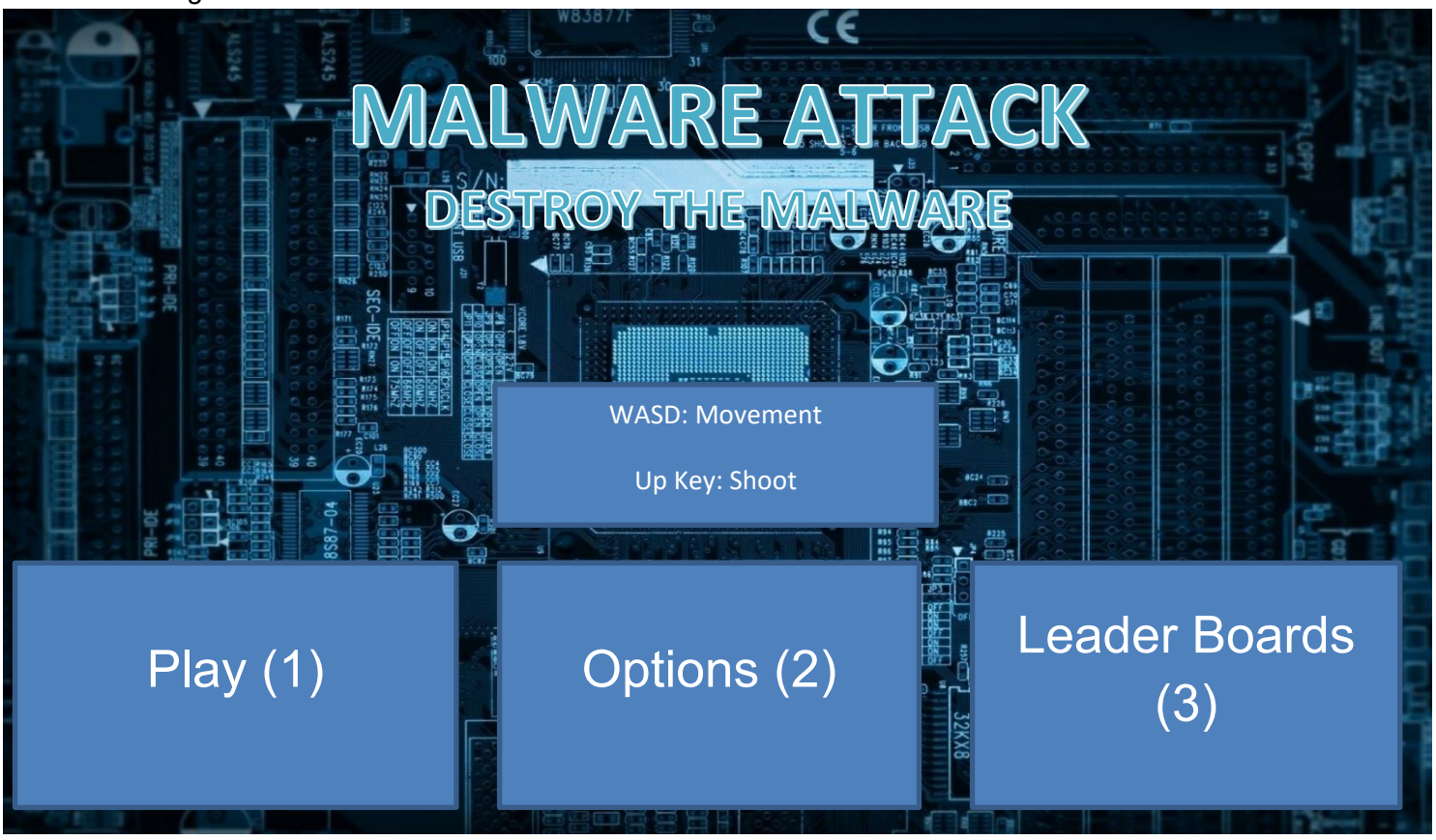
this will make sure the player presses a valid input, so they are taken to the menu or they replay the game.

In summary, the game should start with a main menu, which can navigate throughout all the buttons using just arrow keys, enter key and possibly number keys. The game will have enemies that spawn in when certain conditions are met and shoot projectiles of set patterns created using trig values (sin, cos, tan). The Player will be able to move and shoot and will receive points on defeating enemies. There will be power ups that provide a bonus when a revision question is answered correctly. The game will continue until player health is equal to 0. I believe that this proposed pseudocode will be an effective solution for educating about cyber security and internet safety. Other features maybe added in the future, but this is a good starting point for my project.

2.4 Usability Features

Usability features is something that should be considered during the development of this game. This is so that the game is more accessible to more people, as more people will be able to use it. Key areas to consider include are the GUI, layout, and accessibility features.

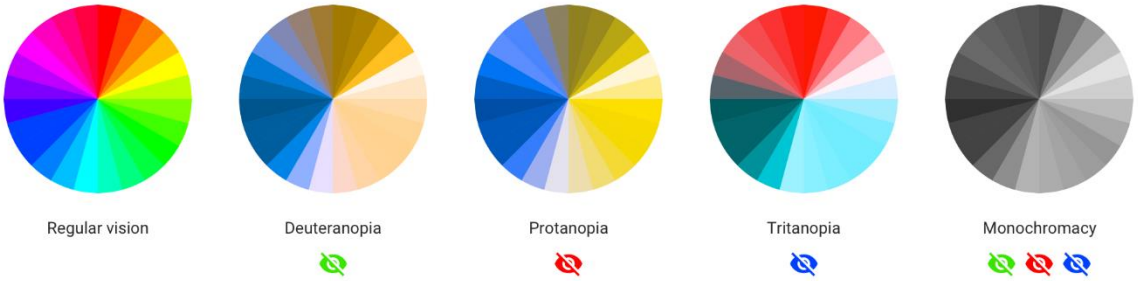
Game Layout: the game layout of the main menu is important as it will be first part of the game users will be able to interact with. If the menu is not laid out in a logical and clear way, users will not be able to easily navigate and traverse through the menu. A poor layout may be enough for some users turn off the game. Therefore, the game menu should have simple layout and look like effort was put into it, as this will be user's first impression of the game. Due to there being only a few controls of the game, it should also be displayed on the menu. This will allow players to immediately know how to play the game. Mock-up of a possible design:



A main menu like this provides a clear layout showing: the name of the game; objective of the game; controls; buttons that will open the game or open another menu.

Buttons: the buttons on the main menu are large and clear, to help those who are visually impaired to see the buttons clearly. The buttons have numbers on them, informing the user that they can be selected with the use of the 1, 2, 3 key respectively.

Accessibility: it is important for a form of colour-blind mode to be incorporated into the game. This is to further increase the total amount of people who can play this game. According to uxdesign.cc (https://uxdesign.cc/color-blindness-in-user-interfaces-66c27331b858) the following image is comparison of: regular vision, Deuteranopia, Protanopia, Tritanopia and Monochromacy.



It is clear to see that most people can see blues and differentiate between them. Therefore, it is a good idea to use blues, especially in the menus, so that users are able to get to the colour-blind mode options and change it to their preference. It is important for accessibility that reds and greens are avoided in the menus.

Another accessibility feature that should be considered is readability of text. Some users will likely be visually impaired and will require larger text. So that all users can clearly read the user interface, I should consider adding a feature in the menu that enlarges text. Having readable text is very important because users are required to read the question and answer it, as there will be no options for the question to be read. However, text to speech could be considered if requested by stakeholders.

An alternative to bigger text is a text to speech feature. This will allow for users to listen to anything that they are hovering over in the menu, as well as anything that is updated (changes in health, occasional change in score). However, a problem with this is that music and sound effects are also planned so this may be hard to hear over other sounds. Another issue this feature is the scope of the project and time limit; realistically, I don't believe I will time to include this feature due to its difficulty and the other features being more important. Therefore this feature will likely not be added, however should be considered, and especially if stakeholders believe this could be useful.

2.5 Identify Key Elements

2.5.1 Variables

Name	Data	Validation	Justification

Background	Background	Sprite / Texture	This will have to be an image such as a jpeg. Otherwise there will not be an image
Player Character	Image, X, Y, Health	NA	Attributes are passed through a constructor class, so user has no input. No validation is required
Malware	Image, X, Y, Health, Attack	NA	Attributes are passed through a constructor class, so user has no input. No validation is required
BossMalware	Image, X, Y, Health, superAttack	NA	Attributes are passed through a constructor class, so user has no input. No validation is required
ButtonClick	Boolean Yes / No	Boolean Data Type ButtonClick = True or False	When the button is click, the variable is set to true. When it is set to true, the respective event will happen, such as the game starting or the a menu being drawn to the screen
ButtonNumber	Boolean Yes / No	Boolean Data Type ButtonClick = True or False	This is similar to the button click variable however, this is referring to how the button's variable can be set by using the number key instead of a mouse click
Score	Integer	This variable will only accept integers passed from time. Score will increase on malware destroys as well.	Only the integer data type is present, so no more validation is needed
Time / Counter	Integer	This variable is created using a timing loop. As the counter variable is increased using +1 the loop waits a second before	As the only thing being passed is 1 no other validation is needed as only integers are used.

		increasing by 1 again. This creates a basic timing loop that will keep score based on time survived. This loop will only run while running is true	
menuCheck	Boolean	Boolean Data Type ButtonClick = True or False	When menuCheck is set to true by button click or using the respective key. When menuCheck is set to true then user is brought back to the main menu. Boolean works here because a button can either be clicked (to set to true) or not clicked (remain false).
colourBlind	Boolean	Boolean Data Type ButtonClick = True or False	When the respective button is clicked then colurBlind will be set to true. This means game assets colours are adjusted to the necessary colours of the players colour blindness
Name	String	String only, no symbols	This field will only accept characters to create a string, not allowing symbols. This is because high scores will be held in a text document and the use of symbols such as a comma can potentially be used to manipulate scores. Entering “Drew, 99999” would mean when a high score table is formed the numbers in the name will become the high score. By ensuring only text can be entered will

			eliminate cheating of this type.
keyDownW	Boolean	keyDownW Boolean KeyDownW = True or False	If the user presses the W button, this will trigger this variable to set to true. This will move the character sprite up the screen. When the key is released the character sprite will stop moving up the screen.
keyDownA	Boolean	keyDownA Boolean KeyDownA = True or False	If the user presses the A button, this will trigger this variable to set to true. This will move the character sprite left across the screen. When the key is released the character sprite will stop moving left across the screen.
keyDownS	Boolean	keyDownS Boolean KeyDownS = True or False	If the user presses the S button, this will trigger this variable to set to true. This will move the character sprite down the screen. When the key is released the character sprite will stop moving down the screen.
keyDownD	Boolean	keyDownD Boolean KeyDownD = True or False	If the user presses the D button, this will trigger this variable to set to true. This will move the character sprite right across the screen. When the key is released the character sprite will stop moving up the screen.
keyDownUp	Boolean	keyDownUp Boolean KeyDownUp = True or False	If the user presses the Up button, this will trigger this

			variable to set to true.
Running	Boolean	Running = True or False	This will be the main loop of the game. When the game starts, this variable is set to true. If the player's coordinates are equal to those of an enemy's projectile, health = health -1. When health = 0 running is set to false. When running is set to false, the game over screen will be called.
Power Up	Boolean	Powered Up = True or false	When collision is made with power up and question is answered correctly power up is set to true. After 10 seconds power up is set to false

2.5.2 Data Structures

High score List

The high score list will made to save 2 inputs from each complete run. The 1st input will be required from the player, which is the string of their name. The 2nd input is automatically put in as it is the score earned in the run. This is not manual, as players could potentially input any score they wanted. This will then be used to make a high score table. The list logic will look like:

Name [1], Score [2],

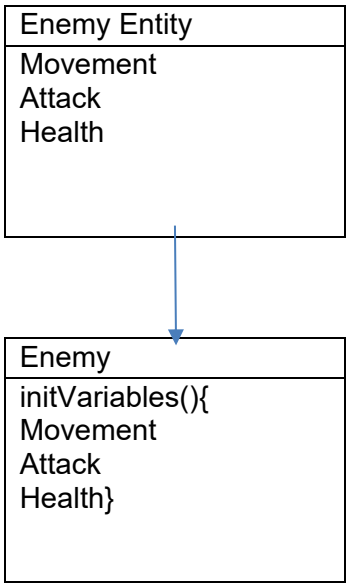
The list will be static in length and will only be able to accept the 0 and 1 positions. This will create a 2D array as the example below shows:

Name [0]	Score Board [1]
Tom [0][0]	100 [1][0]
Drew [0][1]	101 [1][1]
Callum [0][2]	102 [1][2]
George [0][3]	103 [1][3]

2.5.3 Classes

The main class will be used as the basis of all the malware. The create malware class will take an image x, y, movement, and attack and create the malware which will spawn certain

interval (have not decided on this yet as the game will need to be tested to see how often malware should spawn in. This is because too many malwares will lead to the player to get overwhelmed and too little will be boring and not engaging). As the score increases the movement of the enemy malware will increase as well as their attack speed. When the score reaches a certain point, a boss will spawn in for the extra challenge.



2.6 Testing Procedures and Data

Name of Test	Data	Expectation	Justification
Key W	W key	The player's character moves up the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key A	A key	The player's character moves left across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key S	S key	The player's character moves down the screen	The character must be able to move to collide with power ups and avoid attacks. If the player

			is unable to move the game is unplayable and the project is a failure.
Key D	D key	The player's character moves right across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key Up	Up key	The player's character shoots its projectile attack in a linear line (may change with powerups, effects have not been decided yet).	The key feature of a bullet hell SHMUP game is dodging bullets, but also shooting back at enemies. If the player can only dodge the game will become boring and overwhelming as enemies will spawn without any being destroyed.
Play Button	Click button / 1 key	Setting this variable to true takes player from main menu or end of a level into a new game.	If the player is unable to use the play button, they will not be able to access the game. Meaning the project has failed.
Options Button	Click button / 2 key	Setting this variable to true should take the player into the options.	Options button is not as important as the play button, however, is still important. This is because the options menu will have accessibility features such as colour-blind mode.
Leader board Button	Click button / 3 key	Setting this variable to true should take the player to the leader board screen. The high score.txt file should appear in a aesthetically pleasing way.	Arguably the least important of the buttons. This button still needs to work though, because this will show high scores providing a social element. Social elements (as said at the start)

			helps improve engagement and overall retention rate.
Sound	MP3 file	Music should be playing during the main menu, when a button is clicked and when collisions happen	Including sound is important because it increases immersion to the game. If the game didn't have any sound at all, it would be boring, and the player wills struggle to remain engaged for long periods of time
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	If the player id destroyed and there is no way to restart or open a menu, they will be forced to close and reload the game. This would be frustrating and reduces engagement.
Collision	Player coordinates = MalwareAttack coordinates and Malware coordinates = PlayerAttack coordinates and Player coordinates = PowerUp coordinates	If player collides with an attack, they lose a health point; when player health equals 0 game ends. If malware collides with the player's attack, they lose a health point; when malware health equal 0 they are destroyed. If player collides with power up a multiple choice appears on screen and game is paused. If question is answered correctly the power up is granted for a short time.	If collisions do work or work poorly the game may feel unfair as the enemy attacks will lack consistency. This would make the game frustrating to play and reduces replay-ability as players would not want to play a game where their runs are being cut short due to poorly coded collisions. Alternatively, if collisions don't work at all the game will continue indefinitely making the game pointless. The information would not be accessed either as currently the only way user is educated is through the use of multiple-

			choice questions asked when collision is made with power ups. If done poorly or not at all, means the project would have failed
High Score	Time, score	High score is generated based on time survived and malware destroyed.	Including a high score feature improves replay-ability as players will want to beat theirs and others high scores
Colour Blind mode	Options menu data	When enabled game colour changes	To make the game more accessible to a wider audience, the game should include a way for people to see what is happening.
Read text from file. Multiple choice question	.txt document with CSV. Malware question	When player collides with a power up, a question at random will be selected from the txt document. Only if the question is answered correctly will the power up	Having a large range of multiple-choice questions will educate users on a wide range of malware and computer safety knowledge
Save High Score	.txt document withs CSV on scores and names	At the end of a game, the user will get the chance to save their name by typing their name into an input. This will then be saved to a txt document and can be called upon to show a leader board.	High scores need to be saved. This is so they can be displayed, informing players on the scores to beat.
Shooting	Up key	Bullets spawn from the top of the player's sprite	To give the player an objective other than just dodging
Enemies		Multiple enemies are created out of the window before moving into it. Give score when destroyed.	Give a target for the player to shoot at. Score is given for destroying enemies so there is a reason to do so.

Enemy shoot		Enemies should also fire projectiles for the players to dodge. These will be indestructible, so players will be forced to dodged	To increase engagement by increasing difficulty.
Scrolling background		Background is scrolling downwards on loop	Make game feel more dynamic, therefore increasing immersion and engagement.

2.7 Further Testing

Accessibility: the game should be laid out in a user-friendly way, allowing all users to seamlessly traverse through the menus with ease and find what they are looking for efficiently. The best way to test this is by gaining feedback from beta-testers. Assets can then be repositioned to ensure that the GUI is as accessible to as many people as possible.

Education: currently the education will be done through the power ups. When the player collides with the power up, they must answer a multiple-choice question. Players will be incentivised to get the power ups as they will provide a significant buff to the player. However, a player can intentionally try to avoid the power ups and therefore will not get any questions. This can be remedied by maybe having a random fact appear on the main menu and the end level screen. This way players can read a concise note about malware and computer safety each time they play a game or load up the game.

Replay-ability: the game should be as replay-able as possible. If the game repeats too much the game will become predictable and easy, which reduces replay-ability because the game will gradually become less engaging. However, bullet hell games must not have purely random attack patterns as it will too unpredictable and frustrating to play. Enemies must have a variety of set attack patterns which they can randomly use. This can all be tested by use of surveys and looking at how many games are played by each user and their scores. Low scores would suggest the game was too hard; high scores would suggest the game was too easy and few completed games would suggest the game has low replay-ability.

Engagement: the game should be able to engage all users. This is measure to see how well users will be fully engaged and retain the information. If the game has high engagement, users will be able to play the game without being distracted and be able to retain information that is taught. This can be tested by having stakeholders play the game and give me their input. To make the game engaging, the game will need to be of a high quality. This can be achieved with a good difficulty curve. This means as the game start it should be easy, but gradually gets more difficult. If the difficulty gets too hard to quick, the game is too difficult

and is no longer engaging, whereas a game that takes too long to get difficult becomes boring. Therefore, I should aim to make the game with a difficult curve that is appropriate for most young adults.

3. Development of the Solution

3.1 Development

Library

For my project I needed to choose a library that is intended to make games as well as compatible with C++. Therefore, I chose SFML due to its ability to make 2D games; being simple to use and executes code fast: “it is super-fast because it works directly with the software subsystems of our computer without any layers to slow it down”. Due to my game being 2D, SFML is meets all the requirement of my project.

Prototype 1

Game loop

```
11  //main loop - current entry point for the code
12  int main()
13  {
14      Game gameLoop;
15
16      //game loop
17      gameLoop.run();
18
19      return 0;
20  }
```

In main cpp I have set up the game loop. This code will look into the Game class and look at the run function defined in the Game.hpp file and made in Game.cpp.

```
//Game Loop - Used in main
void Game::run()
{
    while (window->isOpen())
    {
        updatePollEvents();

        update();

        render();
    }
}
```

This code creates a while loop which will run continuously until it is broken when the window is closed. While the window is open the updatePollEvents, update and render functions will continuously be running.

Validation is checked here as run() will constantly be checking the update function. This is important because run function is used when window is open. Validation is required to make

the code actually work otherwise events will not work, functions will not update and textures and sprites would not render every tick.

```
}

void Game::updatePollEvents()
{
    //Process events
    sf::Event ev;
    while (window->pollEvent(ev))
    {
        //The X button in the top right will close the window
        if (ev.Event::type == sf::Event::Closed)
            window->close();
        //The escape key will also exit the window
        if (ev.Event::KeyPressed && ev.Event::key.code == sf::Keyboard::Escape)
            window->close();
    }
}

//User inputs - Move, Player shoot
```

In this function the sfml class reference is used. sf::Event holds all the information about a system event that just happened. This is used to acknowledge inputs such as those used in the code of Closed (requests for the window to be closed through clicking the exit button) and the using the escape key to cause the window to close. This allows users to exit the window when they want, without this the code becomes similar to malware.

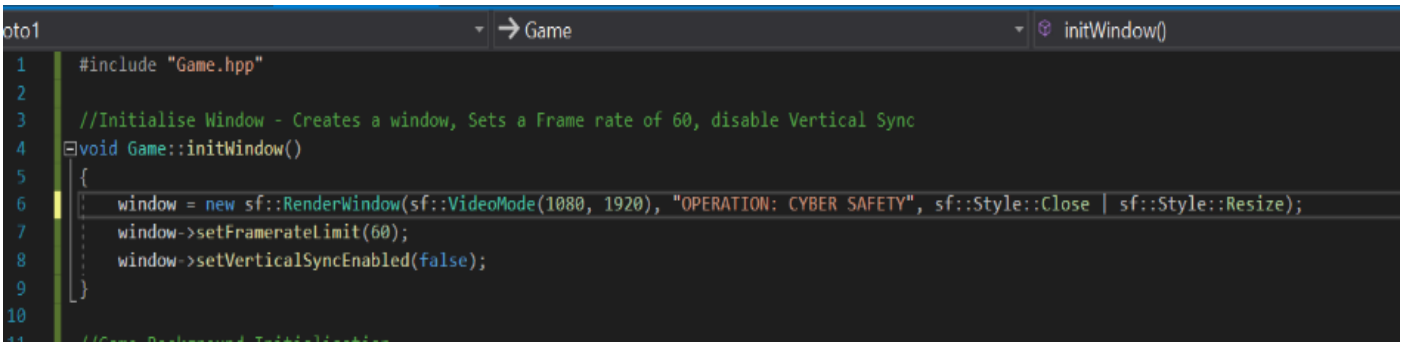
```
//Update
void Game::update()
{
    updateInput();

    updateCollision();

    updateGameBackground();
}
```

This function will constantly update all the functions in this function. This will mean currently player movement will constantly be updated and allows for inputs to be updated; collisions to be constantly updated to prevent players to leaving the bounds of the screen; and for the background to be constantly updated, allowing for it to scroll.

Window

```
private:
    //Window
    sf::RenderWindow* window;
```

sf::RenderWindow is SFML's main class of the Graphics module, which defines a window that can be drawn on. The asterisk is a dereference operator, meaning it is being pointed to by "window" within the Game.cpp file.



In the initWindow() function, window creates a new RenderWindow and using sf::VideoMode a width and height are set. The title of the game is then put, which will appear at the top of the window. Next on this line the style is determined. Through this I allowed the window to have a close button and for the window to be resizable and full screen option. The full screen mode will give the game a stretched resolution, however some users may want the bigger window size.

I then set the windows frame rate to 60. This will force the 60 frames a second maximum. This has the intention of making all computers to run the game similarly as this game is made for users at with expensive computers, as well as users with less-than-optimal computers. This allows for all users to experience a similar experience. Without this some computers will have the player character and enemies move much faster than other users. This will impact the game's difficulty.

Background

```
//Background
sf::Texture backgroundTex;
sf::Sprite backgroundSprite;
sf::Clock clk;
int backgroundSpeed;
float fps = 1000.f / 60.f;
```

First texture, sprite, clock, backgroundSpeed and fps are defined. A texture is an image with the specific role of being mapped to a 2-dimensional entity, and a sprite is a textured entity.

SFML clock and float fps will be used for the background movement and backgroundSpeed is used for determining the speed the background scrolls.

```
//Game Background Initialisation
void Game::initGameBackground()
{
    //Sets background texture from file
    backgroundTex.loadFromFile("Texture/gameBackground2.png.");
    backgroundSprite.setTexture(backgroundTex);

    //Background image will completely and smoothly cycle rather than the last row of the image's pixel being repeated
    backgroundTex.setRepeated(true);
    backgroundTex.setSmooth(true);

    //Sets the speed of the background's movement
    backgroundSpeed = 2;
    //Sets background position so image fills the whole screen
    backgroundSprite.setPosition(0, 0);
```

Within this function the texture of the background is set and then that texture is set to the sprite. This background texture will then be set to be repeated when scrolling as well as having the texture look smooth and not pixelated. Finally in this function the speed of the background scroll is set and position. This will make the sprite cover the whole window and not have any of it not be on the window. There is validation here as the texture has to be set as an image. Without this the background will be not be drawn. For example if a font or document is set as the texture, nothing is drawn.

```
114 void Game::updateGameBackground()
115 {
116     //if elasped time is greater than the fps set, image moves down and clk restarts
117     if (clk.getElapsedTime() >= sf::milliseconds(fps))
118     {
119         sf::IntRect temp = backgroundSprite.getTextureRect();
120         temp.top += -backgroundSpeed;
121         backgroundSprite.setTextureRect(temp);
122         clk.restart();
123     }
124 }
```

In this if statement, the sfml clock will compare the elapsed time with the fps set in game.hpp. if the elapsed time is larger than the fps then the if statement will run. The if statement states that the background sprite is a rectangle, and the rectangle will scroll along the y coordinate. The

```
//draws the background sprite to the game
void Game::renderGameBackground()
{
    window->draw(backgroundSprite);
}

//Renders textures - draws all the textures to the screen and updates the screen to any changes
void Game::render()
{
    window->clear();

    //Draw game background
    renderGameBackground();

    //Draw player
    player->render(*window);

    window->display();
}
```

This code is not just limited to the background, but to the window as well. This code will draw the backgroundSprite to the window. The window will clear everything every second, render all textures, sprites, inputs etc. and display it.

Player Movement

```
void Game::updateInput()
{
    //x and y are set to 0, this value with increase/decrease with player input
    float x = 0.0f;
    float y = 0.0f;

    //pythagoras - this code is intended to make the player not to have a higher speed when moving diagonally
    x = std::sqrt(std::pow((double)x, 2) + std::pow((double)y, 2));
    y = std::sqrt(std::pow((double)x, 2) + std::pow((double)y, 2));

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))
    {
        y -= player->speed; //player moves up (y value decreases)
    }

    else if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
    {
        y += player->speed; //player moves down (y value increases)
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))
    {
        x -= player->speed; //player moves left (x value decreases)
    }

    else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
    {
        x += player->speed; //player moves right (x value increases)
    }
    player->move(x, y); //actually causes player to move.
}
```

The value of x and y are set to 0 to start (if these values were anything else then the player will move by itself). x and y also go through some Pythagoras math to prevent the diagonal movement from being faster than movement in a singular direction. The user can input WASD to move the player character as the values of x and y will be changed for each frame these keys are held down. Once these changes happen to these values they are read from player->move(x,y). This is the move function that is set up in player class (check below).

Validation is used here to check if the keys are pressed, with the use of if and else if statements. Validation in important because it is important to check if a key is pressed and held down, as this will give responsive movement, that will feel good to play. To be more specific when the w key is pressed the player sprite will move up the screen (y value decrease); when s is pressed the sprite will move down the screen (y value increase); when a is pressed the sprite will move left (x value decrease) and when d is pressed the sprite moves right (x value increase).

```
public:
    float speed;
```

Defines speed which will have its value set in the cpp file.

```
    //Functions
    void move(const float dirX, const float dirY);
```

Creates a function in the hpp file. This will allow for movement.

```
{
    //set the value of speed, which is used in player movement
    speed = 1.5f;
}
```

The value of speed is set as 1.5

```
//Movement - Moves sprite with use of speed
void Player::move(const float dirX, const float dirY)
{
    playerSprite.move(speed * dirX, speed * dirY);
}
```

The player sprite movement function takes two arguments: the x direction and the y direction. This is determined through the speed being multiplied by the respective coordinates.

Player

```
10  private:
11      //Define - Player sprite and texture
12      sf::Sprite playerSprite;
13      sf::Texture playerTex;
14
```

Define the player's texture and sprite.

```
//Initialise Textures
void Player::initTexture()
{
    //Loads texture - Texture is loaded from the texture file;
    if (!playerTex.loadFromFile("Texture/playerCharacterSmall.png"))
    {
        //Error message - if the texture fails to load an error message is outputted
        std::cout << "ERROR::PLAYER::Could not load texture file." << "\n";
    }
}

//Sprite initialization
void Player::initSprite()
{
    //Set the texture to the sprite
    playerSprite.setTexture(playerTex);

    //Resize the sprite
    playerSprite.scale(1.0f, 1.0f);
}
```

The player's texture is set and loaded from the texture file. If the texture is not found, then an error message will be outputted to the terminal.

The sprite then has its texture set as well as has its scale set. The scale is currently the image's default; however, I can change easily here if I decide to later.

```
//Sets the x and y co-ordinate of the player
Player(float x = 510, float y = 1600);

//Accessor
const sf::Vector2f& getPos() const;
const sf::FloatRect getBounds() const;

//Modifiers
void setPosition(const float x, const float y);
```

The player has its x and y coordinates set. I also set a get position and get bounds functions.

```
41
42  //Position - returns the current position of sprite
43  const sf::Vector2f& Player::getPos() const
44  {
45      return playerSprite.getPosition();
46  }
47
48  //Bounds of sprite
49  const sf::FloatRect Player::getBounds() const
50  {
51      return playerSprite.getGlobalBounds();
52  }
53
54  //Position - Sets position - x, y
55  void Player::setPosition(const float x, const float y)
56  {
57      playerSprite.setPosition(x, y);
58  }
59
```

The getPos() function will allow me later on to get the position of the player sprite. This is important as I can use this to set the position of where the player's bullets come out of.

The getBounds() function will allow me so set up collisions on the border of the window, so that the player is prevented from leaving window.

Finally, I set the position using the x and y coordinate I set within the parameters of the player constructor.

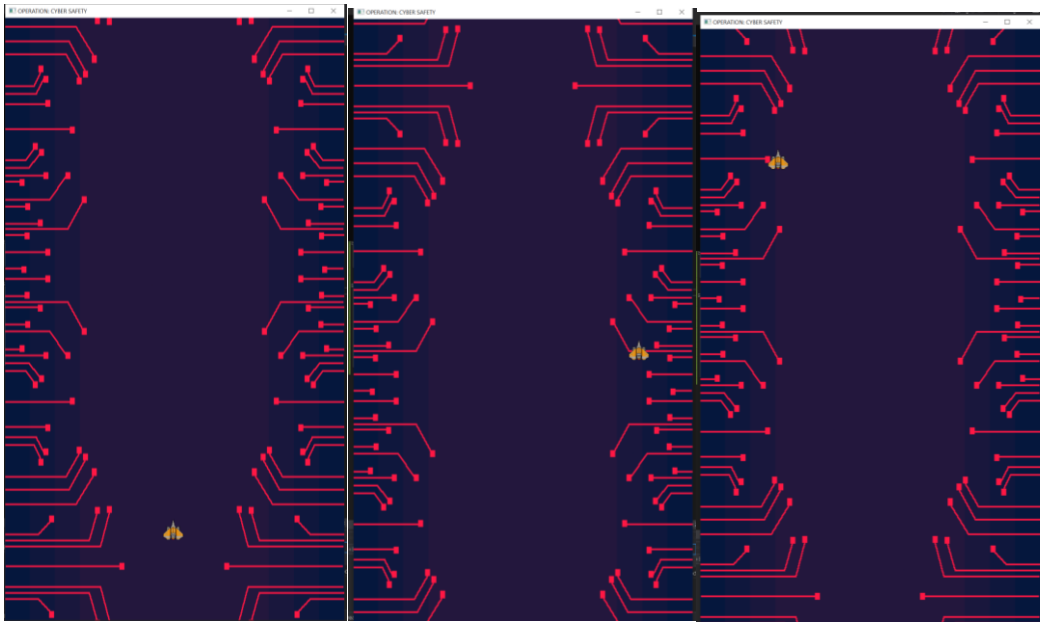
Bounds

```
//Set collisions - Player character cannot get out of bounds
void Game::updateCollision()
{
    //Left collision
    if (player->getBounds().left < 0.f)
        player->setPosition(0.f, player->getBounds().top);
    //Right collison
    else if (player->getBounds().left + player->getBounds().width >= window->getSize().x)
        player->setPosition(window->getSize().x - player->getBounds().width, player->getBounds().top);
    //Top collision
    if (player->getBounds().top < 0.f)
        player->setPosition(player->getBounds().left, 0.f);
    //Bottom collision
    else if (player->getBounds().top + player->getBounds().height >= window->getSize().y)
        player->setPosition(player->getBounds().left, window->getSize().y - player->getBounds().height);
}
```

In this section of code, I set the collisions of the window to prevent the player from leaving the screen. This is done through using the bounds of the window and of the player. For the left and top bound I have it so that if the left player's bound is smaller than 0 player position would be set to 0. What this does is that if the player has its x or y coordinate to 0 it will have its position constantly set to 0 when it tries to get lower than that.

This is slightly different to the right and bottom collision. First the players left/top bounds are added to the width/height of the sprite. This will accurately get the right and bottom bounds of the sprite. When these values try to have their values larger than windows x size and y size respectively. The position of the sprite (when against the right bound) will then get set to the furthest x coordinate – player bounds (so the whole sprite stays in the window) with the y coordinate not changing unless player input dictates it. When the sprite attempts to go over the maximum y coordinate of window, position of the x coordinate would not be set to anything different, but the y coordinate will constantly be set to the window's maximum y size – player's height.

Game Play



These images show that the player sprite is able to move on the window and that the background of window is scrolling downwards and fills the entire screen.

Name of Test	Data	Expectation	Justification
Key W	W key	The player's character moves up the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key A	A key	The player's character moves left across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key S	S key	The player's character moves down the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key D	D key	The player's character moves	The character must be able to move to collide with power

		right across the screen	ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key Up	Up key	The player's character shoots its projectile attack in a linear line (may change with powerups, effects have not been decided yet).	The key feature of a bullet hell SHMUP game is dodging bullets, but also shooting back at enemies. If the player can only dodge the game will become boring and overwhelming as enemies will spawn without any being destroyed.
Play Button	Click button / 1 key	Setting this variable to true takes player from main menu or end of a level into a new game.	If the player is unable to use the play button, they will not be able to access the game. Meaning the project has failed.
Options Button	Click button / 2 key	Setting this variable to true should take the player into the options.	Options button is not as important as the play button, however, is still important. This is because the options menu will have accessibility features such as colour-blind mode.
Leader board Button	Click button / 3 key	Setting this variable to true should take the player to the leader board screen. The high score.txt file should appear in an aesthetically pleasing way.	Arguably the least important of the buttons. This button still needs to work though, because this will show high scores providing a social element. Social elements (as said at the start) helps improve engagement and overall retention rate.
Sound	MP3 file	Music should be playing during the main menu, when a	Including sound is important because it increases immersion

		button is clicked and when collisions happen	to the game. If the game didn't have any sound at all, it would be boring and the player wills struggle to remain engaged for long periods of time
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	If the player id destroyed and there is no way to restart or open a menu, they will be forced to close and reload the game. this would be frustrating and reduces engagement.
Collision	Player coordinates = MalwareAttack coordinates and Malware coordinates = PlayerAttack coordinates and Player coordinates = PowerUp coordinates	If player collides with an attack, they lose a health point; when player health equals 0 game ends. If malware collides with the player's attack, they lose a health point; when malware health equal 0 they are destroyed. If player collides with power up a multiple choice appears on screen and game is paused. If question is answered correctly the power up is granted for a short time.	If collisions do work or work poorly the game may feel unfair as an enemy attacks lack consistency. This would make the game frustrating to play and reduces replay-ability as players would not want to play a game where their runs are being cut short due to poorly coded collisions. Alternatively, if collisions don't work at all the game will continue indefinitely making the game pointless. The information would not be accessed either as currently the only way user are educated is through the use of multiple choice questions asked when collision is made with power ups. If done poorly or not at all, means

			the project would have failed
High Score	Time, score	High score is generated based on time survived and malware destroyed.	Including a high score feature improves replay-ability as players will want to beat theirs and others high scores
Colour Blind mode	Options menu data	When enabled game colour changes	To make the game more accessible to a wider audience, the game should include a way for people to see what is happening.
Read text from file. Multiple choice question	.txt document with CSV. Malware question	When player collides with a power up, a question at random will be selected from the txt document. Only if the question is answered correctly will the power up	Having a large range of multiple-choice questions will educate users on a wide range of malware and computer safety knowledge
Save High Score	.txt document withs CSV on scores and names	At the end of a game, the user will get the chance to save their name by typing their name into an input. This will then be saved to a txt document and can be called upon to show a leader board.	High scores need to be saved. This is so they can be displayed, informing players on the scores to beat.
Shooting	Up key	Bullets spawn from the top of the player's sprite	To give the player an objective other than just dodging
Enemies		Multiple enemies are created out of the window before moving into it. Give score when destroyed.	Give a target for the player to shoot at. Score is given for destroying enemies so there is a reason to do so.
Enemy shoot		Enemies should also fire projectiles for the players to dodge. These will be indestructible, so	To increase engagement by increasing difficulty.

		players will be forced to dodged	

Rate Prototype //Score: 1 out of 5						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	0	3	3	3	2	Improved education
George	0	3	3	4	1	Improved difficulty
Calllum	0	3	1	3	2	Improve gameplay
Drew	0	3	1	4	1	Improve game play to make it more re playable

Sum	0	12	8	14	6
Average	0	3	2	3.5	1.5

Conclusion of prototype 1

Currently from the people that have play tested the game, a shared concern is the lacklustre game play. This is due to that happens is in this prototype is a moveable sprite and scrolling background. Before I can properly address this concern, I shall need to set up for the introduction of enemies (prototype 3). Therefore, I shall focus on setting up the GUI, health and shooting.

Prototype 2 – Everything from this point will also be new

GUI

The intention of a GUI is to show the user necessary information for the player to make good and informed decisions. Things that will be included will player health, score, enemy health and score with the possible inclusion of other displays. To best code this I shall create a GUI class that set up all the GUI related functions.

```
#include "Game.hpp"

class Gui
{
public:
    //Score
    unsigned score;

    //GUI - font
    sf::Font font;

    //GUI - Score text
    sf::Text scoreText;

    //GUI - Game over text
    sf::Text gameOverText;

    //GUI - health bar and text
    sf::Text healthText;
    sf::RectangleShape playerHpBar;
```

Score validation is important as it will give unsigned value. This means that the score will not be negative value, but it can now be a larger value. Since the score increases by integer values from killing enemies, the int data time is currently not needed. Score will currently not

increase with time as planned, however this may change with a later iteration.

In the GUI class I first start by defining variables that will used to show the score, health and game over text. The data type for score is unsigned because it is impossible for the player to achieve a negative value for score and so that the score can reach a high value. The player will be informed of their health using a health bar as well as numerical values. This is because some players may prefer one way over another, and I want to cater to as many potential users as possible.

```
void initGUI()
{
    //Load font
    if (!font.loadFromFile("Font/PressStart2P-Regular.ttf"))
    {
        std::cout << "ERROR::GAME::Failed to load font" << "\n";
    }
```

I

```
    //GUI - Text
    //Score Text
    scoreText.setPosition(800, 1800.f);
    scoreText.setFont(font);
    scoreText.setCharacterSize(20);
    scoreText.setFillColor(sf::Color::White);

    //Game Over Text
    gameOverText.setFont(font);
    gameOverText.setCharacterSize(60);
    gameOverText.setFillColor(sf::Color::Red);
    gameOverText.setString("Game Over");
    gameOverText.setPosition(500, 500);

    //GUI - Health
    //GUI - Health Text
    healthText.setPosition(20.f, 1800.f);
    healthText.setFont(font);
    healthText.setCharacterSize(20);
    healthText.setFillColor(sf::Color::White);

    //GUI - Health Bar
    playerHpBar.setSize(sf::Vector2f(20.f, 10.f));
    playerHpBar.setFillColor(sf::Color::Red);
    playerHpBar.setPosition(sf::Vector2f(20.f, 1850.f));
}
```

then initialise the variables in the initGUI() function. In this function I first set the font from the font file I have in the project. If the font does not load from the font file successfully, I have an error message outputted into the console.

I then proceeded to set up score text, game over text, health text. I set the position, font, font size and colour. For the game over text, I added the string that will be outputted. This is different from health and score as their string will be set in using a stream string which is done in the game.cpp file. This is because the values of score and health will be changing as the game is played, whereas game over will not change, only being printed to the screen.

In addition to setting up the text, I also initialised the player health bar. I set the size of the rectangle, colour and position. Just like the score and health text, I have included the health bar within the string stream of game.cpp.

```
    //Initialise Score - Score starts with the value of 0
    void initScore()
    {
        score = 0;
    }
};
```

The final thing I did in the GUI file was initialise the score by setting it to 0. This means the score will start at 0 when the game is running, but due to the string stream the value will increment over time, as player kill enemies.

```
    //GUI
    Gui* gui;
```

A dereference I made to call the GUI class in the game files.

```
void Game::initGui()
{
    gui = new Gui();
}
```

This is used in the Game.cpp file; new Gui() will assure that enough memory is allocated for the GUI class and will call the code. This will allow for the GUI text and health bar to be called and used in cpp file.

```
69      //Destructor
70      Game::~Game()
71      {
72          delete window;
73          delete player;
74          delete gui;
```

As new is being used, gui as well as everything that uses new must also be deleted. This is because when new is used the memory allocated is not freed unless manually coded too.

```
void Game::updateGUI()
{
    //Stringstream - will update the score and health change
    std::stringstream sss;
    sss << "Score: " << gui->score;
    gui->scoreText.setString(sss.str());

    std::stringstream ssh;
    ssh << "Health:" << player->getHp();
    gui->healthText.setString(ssh.str());

    //Health bar will reduce in size as health decreases
    float hpPercent = float(player->getHp()) / player->getHpMax();
    gui->playerHpBar.setSize(sf::Vector2f(185.f * hpPercent, gui->playerHpBar.getSize().y));
}
```

Here string stream is used. string stream allows for the string to output values that are updated as the game is played. For example, with score the string score is outputted to the screen, << operator will add a string to the string stream object. In this instance it is the score. Health text works very similarly, except it updates the health variable in the player class.

The health bar works differently due to it not being a string but a rectangle. First, I define hpPercent as a float data type and make it equal to player's current health divided by the players maximum health. The size of the bar is set so that as health decreases the x value (width) of the health bar decreases but y (height) will not change as its value is just called

from the GUI class. I multiply hpPercent by 185 because without it, the x size is very small, and the player will struggle to tell how much health they have. So, increase the size to make it more visible.

```
//Update
void Game::update()
{
    player->update();

    updateCollision();

    updateGUI();

    updatePlayerAttackInput();
    updatePlayerAttack();

    updateGameBackground();
}

void Game::renderGUI()
{
    window->draw(gui->scoreText);
    window->draw(gui->healthText);
    window->draw(gui->playerHpBar);

    //Game over screen
    if (player->getHp() <= 0)
        window->draw(gui->gameOverText);
}
```

I include updateGUI function in the game.cpp function so if the player has health, the score and health will continue to show updated results.

Under that I make sure the score text, health text and player health bar are all being drawn to the screen. In an if statement, I put the draw function of gameOverText. This is so that the game over is only drawn to the screen when the game is over.

Player Attacks

An essential part of the game will be implementing a way for the player to attack the enemies (that will be added in the next prototype). I decided to have a regular bullet like projectile that shoots from the middle of the player's sprite. In addition to this regular attack, I shall also be including an attack which will come out of the back of the sprite and go the opposite direction. This will allow for a variety of playstyles, as the behind attack will have a larger sprite and does more damage, but slower movement as well as a slower cool down speed. This means this attack will be used much more scarcely but will give the player an opportunity to finish off enemies that get past them, to attempt to attain a higher score. This idea may be developed to make enemies defeated with behind attack to reward more points, allowing for more skilled players to adopt a more challenging playstyle to get higher scores.

```cpp
#pragma once

#include <iostream>
#include<SFML/Graphics.hpp>

class PlayerAttack
{
protected:
    //Private Definitions
    //Define - Attack sprite
    sf::Sprite attackSprite;

    //Define -  direction and movementSpeed
    sf::Vector2f attackDirection;
    float attackSpeed;
};
```

Due to there being more than one attack (bullet and bomb), I shall be creating a parent class to start, so that the bullet and bomb can both inherit the definitions from this class. In this class I define the attack's Sprite, direction the attacks go in and the speed the attacks travel at. Validation is used for attack speed to be a float variable. This means that the attack speed will move at more specific speeds. This allows me to better test the player attack speed, this will make stakeholders happy as they will get a speed they like.

```cpp
#pragma once

#include "PlayerAttack.hpp"

class PlayerBullet : public PlayerAttack
{
public:
    //Bullet - Texture, x and y position, x and y direction, bullet's movement speed
    PlayerBullet(sf::Texture* texture, float pos_x, float pos_y)
    {
        attackSprite.setTexture(*texture);
        attackSprite.setOrigin(8.0f, 8.0f);
        attackSprite.setPosition(pos_x, pos_y);

        attackDirection.x = 0;
        attackDirection.y = -1;
        attackSpeed = 17;
    }
    //Accessor
    //Bounds - Returns the bounds of the bullet
    const sf::FloatRect getBounds() const
    {
        return attackSprite.getGlobalBounds();
    }

    //Update and Render Bullets
    void update()
    {
        attackSprite.move(attackSpeed * attackDirection);
    }
    //Render - Bullet is drawn to the screen
    void render(sf::RenderTarget* target)
    {
        target->draw(attackSprite);
    }
};
```

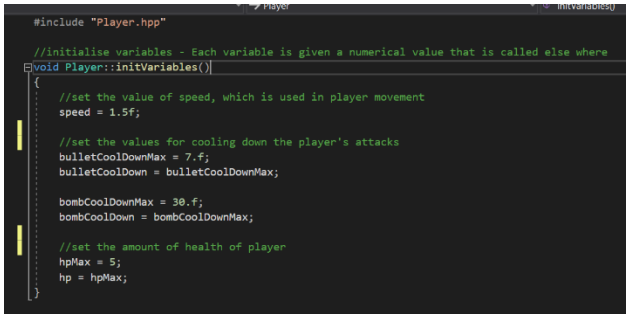
```cpp
#pragma once

#include "playerAttack.hpp"

class PlayerBomb : public PlayerAttack
{
public:
    //Bullet - Texture, x and y position, x and y direction, bullet's movement speed
    PlayerBomb(sf::Texture* texture, float pos_x, float pos_y)
    {
        attackSprite.setTexture(*texture);
        attackSprite.setOrigin(16.0f, 16.0f);
        attackSprite.setScale(1.5f, 1.5f);
        attackSprite.setPosition(pos_x, pos_y);

        attackDirection.x = 0;
        attackDirection.y = 1;
        attackSpeed = 5;
    }

    //Accessor
    //Bounds - Returns the bounds of the bullet
    const sf::FloatRect getBounds() const
    {
        return attackSprite.getGlobalBounds();
    }

    //Update and Render Bullets
    void update()
    {
        attackSprite.move(attackSpeed * attackDirection);
    }
    //Render - Bullet is drawn to the screen
    void render(sf::RenderTarget* target)
    {
        target->draw(attackSprite);
    }
};
```

In both bullet and bomb classes I have included playerAttack (the parent class) so that the bullet and bomb class can inherit the definitions. First in both classes I set a constructor, in which I set the texture, origin, position, direction and speed of the attacks. For the texture and position of the attacks I have used parameters and arguments to set them. This is so that I can set the position of the attacks to the player sprite, and to avoid issues of applying texture to the attacks I have used pointers and dereferences to set it in the game.cpp file.

Next, I create function, which will allow me to get the bounds of the attacks. This will be important as it will be used to delete the player's attack when they leave the screen.

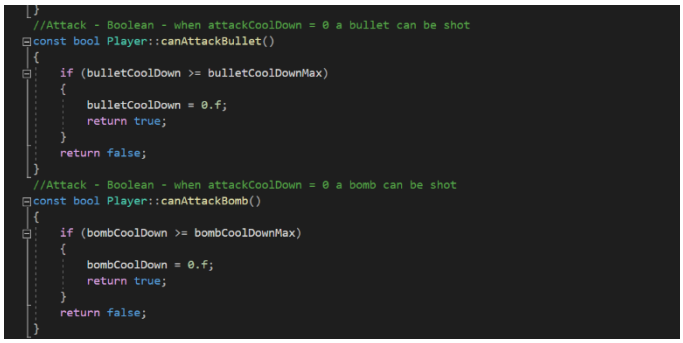
The update() function is used to move the sprite of the player's attack by multiplying the speed by the direction. This will be called in updatePlayerAttack() function in the game.cpp (explained below), which will then be placed in the game update() function.

Finally render functions are used to draw the sprites to the screen.



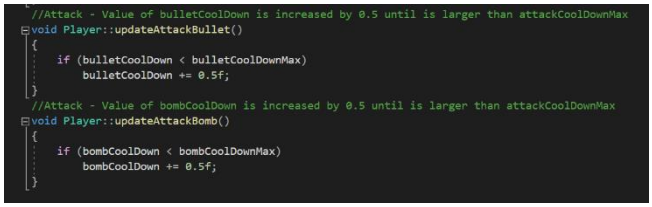
```
#include "Player.hpp"

//initialise variables - Each variable is given a numerical value that is called else where
void Player::initVariables()
{
    //set the value of speed, which is used in player movement
    speed = 1.5f;

    //set the values for cooling down the player's attacks
    bulletCoolDownMax = 7.f;
    bulletCoolDown = bulletCoolDownMax;

    bombCoolDownMax = 30.f;
    bombCoolDown = bombCoolDownMax;

    //set the amount of health of player
    hpMax = 5;
    hp = hpMax;
}
```

So that the bullets and bombs don't fire once per tick, making the game liable to crash and being too overpowered in killing enemies, a cool down must be used. I set the value of the cool down max value as the number I want (smaller numbers having a shorter cool down before next attack is fired), and then set the cool down to equal the max cool down. This is so that when game starts the attacks can be used straight away, otherwise the player will have to wait for the cool down to be completed until they shoot their first attack.



```
//Attack - Boolean - when attackCoolDown = 0 a bullet can be shot
const bool Player::canAttackBullet()
{
    if (bulletCoolDown >= bulletCoolDownMax)
    {
        bulletCoolDown = 0.f;
        return true;
    }
    return false;
}
//Attack - Boolean - when attackCoolDown = 0 a bomb can be shot
const bool Player::canAttackBomb()
{
    if (bombCoolDown >= bombCoolDownMax)
    {
        bombCoolDown = 0.f;
        return true;
    }
    return false;
}
```

The function canAttackBullet() / canAttackBomb both serve a similar purpose. Both check using an if statement if the cooldown of the attack is complete. If the cool down is larger or equal to the max cool down can attack is set to true and the value is set back to 0, else false is returned.



```
//Attack - Value of bulletCoolDown is increased by 0.5 until is larger than attackCoolDownMax
void Player::updateAttackBullet()
{
    if (bulletCoolDown < bulletCoolDownMax)
        bulletCoolDown += 0.5f;
}
//Attack - Value of bombCoolDown is increased by 0.5 until is larger than attackCoolDownMax
void Player::updateAttackBomb()
{
    if (bombCoolDown < bombCoolDownMax)
        bombCoolDown += 0.5f;
}
```

the update attack functions use if statements as well. They both check if cool down is below the max cool down, and if it is 0.5

is added to 0 every tick. This will continue until 7 or 30 is reached respectively. Only then will the can attack function will be true, allowing for a bullet to be drawn.

```
//Player Attacks Texture
std::map<std::string, sf::Texture*> bulletTex;
std::vector<PlayerBullet*> bullets;

std::map<std::string, sf::Texture*> bombTex;
std::vector<PlayerBomb*> bombs;
```

In the game header file, map associative containers (https://www.cplusplus.com/reference/map/map/) are used to map the mapped value of Texture to the key value of bulletTex / bombTex. This will allow for the texture set in game.cpp to linked to the bullet and bomb class.

```
38
39  void Game::initBulletTextures()
40  {
41      bulletTex["BULLET"] = new sf::Texture();
42      bulletTex["BULLET"]->loadFromFile("Texture/playerAttack2.png");
43  }
44
45  void Game::initBombTextures()
46  {
47      bombTex["BOMB"] = new sf::Texture();
48      bombTex["BOMB"]->loadFromFile("Texture/playerAttackBack.png");
49  }
50
```

In the game.cpp file the bullet and the bomb textures are applied to their respective strings. This is will then be used to call the textures of the attacks when the shoot keys are pressed. This code works in conjunction with the map code above.

```
//Destructor
Game::~Game()
{
    delete window;
    delete player;
    delete gui;

    //Delete textures
    for (auto& i : bulletTex, bombTex)
    {
        delete i.second;
    }

    //Delete bullets
    for (auto* i : bullets, bombs)
    {
        delete i;
    }
}
```

Due to the new operator being to use again to allocate memory for the attack sprites, they will have to be deleted again. After the code stops running the memory will be freed up again.

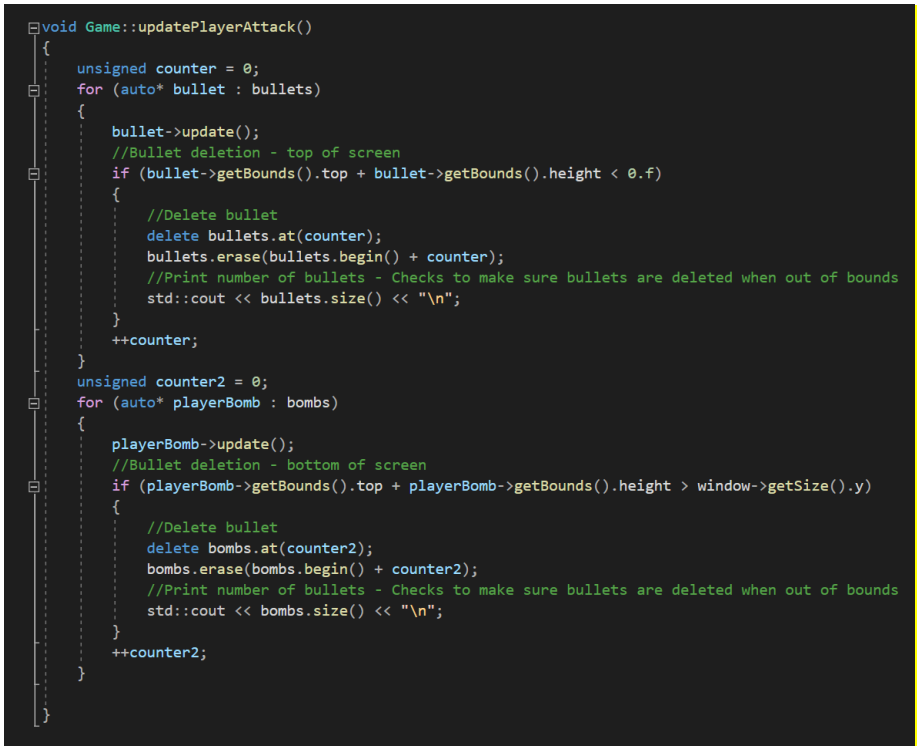
```
void Game::updatePlayerAttackInput()
{
    //Player shoot - Bullets
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && player->canAttackBullet())
    {
        bullets.push_back(
            new PlayerBullet(
                bulletTex["BULLET"],
                //Sets Position - The position where bullet is drawn to the screen
                player->getPos().x + player->getBounds().width / 2.6,
                player->getPos().y));
    }

    //Player shoot - Bombs
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down) && player->canAttackBomb())
    {
        bombs.push_back(
            new PlayerBomb(bombTex["BOMB"],
                //Sets Position - The position where bomb is drawn to the screen
                player->getPos().x + player->getBounds().width / 2,
                player->getPos().y + player->getBounds().width / 1.3));
    }
}
```

These functions have the purpose of allowing for player input to shoot the bullet / bomb. First an if statement is used to check for the correct keyboard input and the can attack functions in the player cpp file. If both are true, then the attack is fired. This is done through use of the push_back method. Essentially, every time an attack is fired the attack texture is applied to the values set within the PlayerBullet / PlayerBomb constructor as well as the position set within this function. The content of bullets / bombs are copied into the new element of the respective classes.

Validation is used here again. When up and down arrows are used and when the cool down is over, validation allows for the player attack sprites to be rendered and move.

```
void Game::updatePlayerAttackInput()
{
    //Player shoot - Bullets
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && player->canAttackBullet())
    {
        bullets.push_back(
            new PlayerBullet(
                bulletTex["BULLET"],
                //Sets Position - The position where bullet is drawn to the screen
                player->getPos().x + player->getBounds().width / 2.6,
                player->getPos().y,
                //x-direction, y-direction, movement speed
                0.f, -1.f, 17.f));
    }

    //Player shoot - Bombs
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down) && player->canAttackBomb())
    {
        bombs.push_back(
            new PlayerBomb(
                bombTex["BOMB"],
                //Sets Position - The position where bullet is drawn to the screen
                player->getPos().x + player->getBounds().width / 2,
                player->getPos().y + player->getBounds().height / 1.2,
                //x-direction, y-direction, movement speed
                0.f, 1.f, 5.f));
    }
}
```

The updatePlayerAttackInput() function is used to check if the player presses the bound keys to attack. If these keys are pressed and the player can attack (checked in the player class). If both are true, then bullets / bombs are fired from the player position. Bullets / bombs are pushed back using the respective sprite. Other variables are set here from the player bullet and player bomb classes. These include the x direction, y direction and movement speed. There is a possibility of adding more to this, but I don't believe that I will make the player's bullets have any special movement, however player bombs maybe changed to have trig functions in their movement. This will allow for more unique gameplay, with a higher skill ceiling. To balance these changes to firing delay will need to be implemented for balancing reasons. However, I am happy with this currently, although I may come back to this idea in a later prototype after I have added the enemies.

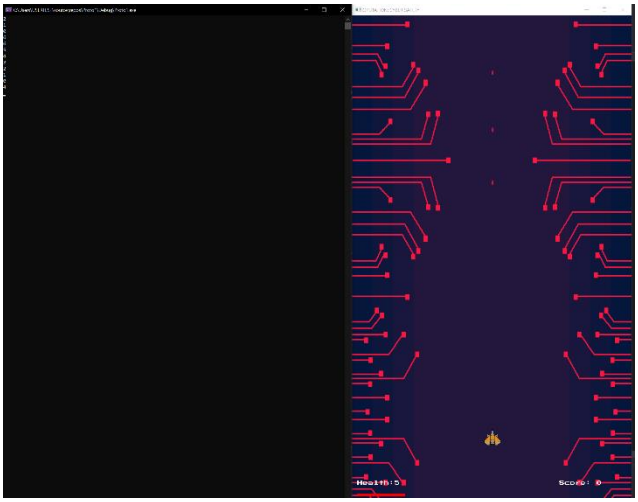


```
void Game::updatePlayerAttack()
{
    unsigned counter = 0;
    for (auto* bullet : bullets)
    {
        bullet->update();
        //Bullet deletion - top of screen
        if (bullet->getBounds().top + bullet->getBounds().height < 0.f)
        {
            //Delete bullet
            delete bullets.at(counter);
            bullets.erase(bullets.begin() + counter);
            //Print number of bullets - Checks to make sure bullets are deleted when out of bounds
            std::cout << bullets.size() << "\n";
        }
        ++counter;
    }
    unsigned counter2 = 0;
    for (auto* playerBomb : bombs)
    {
        playerBomb->update();
        //Bullet deletion - bottom of screen
        if (playerBomb->getBounds().top + playerBomb->getBounds().height > window->getSize().y)
        {
            //Delete bullet
            delete bombs.at(counter2);
            bombs.erase(bombs.begin() + counter2);
            //Print number of bullets - Checks to make sure bullets are deleted when out of bounds
            std::cout << bombs.size() << "\n";
        }
        ++counter2;
    }

}
```

This section of code is used to cull (removing when certain conditions are met) bullets when the bottom of its sprite reaches a position less than 0 on the window. This will therefore delete all bullets when they are no longer visible.

This is similar to the next section of bomb culling. The code is very similar except that the bombs will be deleted at the bottom of window when no longer visible.

This is confirmed with the cout bullet.size() code. This is because whenever a bullet is deleted the remaining number of bullets are printed to the screen. This will always print the number 0 as all bullets will eventually be deleted. There is also cout bomb.size() code which will work exactly like the code just explained, but with the bombs.



(magnified console showing total bullets that currently rendered by the game.)

```
    //Draw bullets
    for (auto* bullet : bullets)
    {
        bullet->render(window);
    }

    //Draw bombs
    for (auto* bomb : bombs)
    {
        bomb->render(window);
    }

    window->display();
}
```

This code will use a for loop and dereference to bullet and bomb to render a bullet or bomb when the shoot key is pressed. This will then render a bullet or bomb sprite to the window each time the key is pressed, and the cool down is over.

Player Health

Player Health points is an important part of most games, as it will provide the additional challenge of survival. This will therefore require a player to avoid incoming enemy attacks, otherwise the game will provide no challenge and lack engagement. This will lead to lack of playtime if the game has infinite health and enemy attacks don't do anything. Another issue that should be considered is the balancing of health; if the player's health is too high then the game is too easy, but if the player's health is too low then the game will be too difficult.

It is important to make the health balanced so that the game is not too hard or too easy. Even with balanced health, the most experienced players will be able to play the game with ease and quickly get bored. Therefore, I will have to find an effective method to remedy this with the enemies in the next prototype.

```
//Define - Health
int hp;
int hpMax;
```

This creates data values to be set in the player's cpp file. hp will be the players hp value that changes as they take damage from enemies, whereas the hpMax will be the maximum health the player has and starts with.

```
const int& getHp() const;
const int& getHpMax() const;
```

This getter will allow for the health values to be called elsewhere in the code.

```
void setHp(const int hp);
void loseHp(const int value);
```

The setHp code will allow for a new health value to be set and loseHp will be used to decrease the value of the player's health.

All the code shown above for player health so far will all work in conjunction with each other to decrease the player's health.

```
//set the amount of health of player
hpMax = 5;
hp = hpMax;
```

In the player.cpp file, initVariables() function I set the value for hpMax as 5. Then I set the hp to be equal to hpMax. This will be overwritten as the game is played by the player and lose health. The value of 5 maybe changed as the game is developed (especially when enemies are added), however I believe that 5 is a good starting point for now.

```
//Hp - function will return the current hp value
const int& Player::getHp() const
{
    return hp;
}
```

Will return the current value of hp when used elsewhere in the code, such as in the game.cpp file.

```
//Hp - function will return the maximum hp value
const int& Player::getHpMax() const
{
    return hpMax;
}
```

Will return the maximum health elsewhere in the code when getHpMax() function is used.

```
//Hp - Sets a new value for health
void Player::setHp(const int hp)
{
    this->hp = hp;
}
```

This will set a new value for health such as when the player collides with an enemy attack. In a later prototype I may change the name for the parameter's name as they are both called hp. This requires the use of this pointer and maybe confusing for other people reading my code.

```
//Hp - Decreases the value of health
void Player::loseHp(const int value)
{
    hp -= value;
}
```

This will decrease the value of health when loseHp() function is used.

```
    //If the player's Hp reaches 0, the Game loop will end. Stops updating.
    if (player->getHp() > 0)
        update();
```

In the run() function I have added code above the update function. The addition of this new code makes it so that the run() function will only call the update() function when the player's current health (using getHp()) is greater than 0. When player's health is less than 0 the game will stop updating.

```
//Game over screen
if (player->getHp() <= 0)
    window->draw(gui->gameOverText);
```

Another use of getHp() function to check if the player's health is grater than 0 is the renderGUI() function. This code will render the game over text when the player has lost all its health.

This screen shot shows what happens when the health is less than 0. I will likely render the Game over string in a more readable by changing its colour and by moving it to the middle of the window. Alternatively, I may consider changing all the textures used in the game currently, however I shall decide based on user feedback.

Furthermore, I will aim to implement a restart option and return to menu screen buttons once the player has lost all their health. But this will be done in a future prototype.

```
std::stringstream ssh;
ssh << "Health:" << player->getHp();
gui->healthText.setString(ssh.str());
```

Here the player health code is used to integrate the gui and health to clearly show the player the health of their character. This will use stringstream to update the player's health through a numbers by rendering the player's current health.

```
//Health bar will reduce in size as health decreases
float hpPercent = float(player->getHp()) / player->getHpMax();
gui->playerHpBar.setSize(sf::Vector2f(185.f * hpPercent, gui->playerHpBar.getSize().y));
```

Another way health is shown to the player is through the use of a health bar. This is made redundant through the method above, however many bullet hell games make use of health bar. Furthermore, I feel as if this inclusion adds more of a dynamic feel to the game as the bar decreases with health. The bar changes size by dividing current health by the maximum health, and as health decreases the x value of bar will also change. Other values for the bar are set in the gui.hpp file such as colour (see above).

Name of Test	Data	Expectation	Justification
Key W	W key	The player's character moves up the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key A	A key	The player's character moves left across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key S	S key	The player's character moves down the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.

Key D	D key	The player's character moves right across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key Up	Up key	The player's character shoots its projectile attack in a linear line (may change with powerups, effects have not been decided yet).	The key feature of a bullet hell SHMUP game is dodging bullets, but also shooting back at enemies. If the player can only dodge the game will become boring and overwhelming as enemies will spawn without any being destroyed.
Play Button	Click button / 1 key	Setting this variable to true takes player from main menu or end of a level into a new game.	If the player is unable to use the play button, they will not be able to access the game. Meaning the project has failed.
Options Button	Click button / 2 key	Setting this variable to true should take the player into the options.	Options button is not as important as the play button, however, is still important. This is because the options menu will have accessibility features such as colour-blind mode.
Leader board Button	Click button / 3 key	Setting this variable to true should take the player to the leader board screen. The high score.txt file should appear in an aesthetically pleasing way.	Arguably the least important of the buttons. This button still needs to work though, because this will show high scores providing a social element. Social elements (as said at the start) helps improve engagement and overall retention rate.

Sound	MP3 file	Music should be playing during the main menu, when a button is clicked and when collisions happen	Including sound is important because it increases immersion to the game. If the game didn't have any sound at all, it would be boring and the player wills struggle to remain engaged for long periods of time
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	If the player id destroyed and there is no way to restart or open a menu, they will be forced to close and reload the game. This would be frustrating and reduces engagement.
Collision	Player coordinates = MalwareAttack coordinates and Malware coordinates = PlayerAttack coordinates and Player coordinates = PowerUp coordinates	If player collides with an attack, they lose a health point; when player health equals 0 game ends. If malware collides with the player's attack, they lose a health point; when malware health equal 0 they are destroyed. If player collides with power up a multiple choice appears on screen and game is paused. If question is answered correctly the power up is granted for a short time.	If collisions do work or work poorly the game may feel unfair as an enemy attacks lack consistency. This would make the game frustrating to play and reduces replay-ability as players would not want to play a game where their runs are being cut short due to poorly coded collisions. Alternatively, if collisions don't work at all the game will continue indefinitely making the game pointless. The information would not be accessed either as currently the only way user are educated is through the use of multiple choice questions asked when collision is made with power

			ups. If done poorly or not at all, means the project would have failed
High Score	Time, score	High score is generated based on time survived and malware destroyed.	Including a high score feature improves replay-ability as players will want to beat theirs and others high scores
Colour Blind mode	Options menu data	When enabled game colour changes	To make the game more accessible to a wider audience, the game should include a way for people to see what is happening.
Read text from file. Multiple choice question	.txt document with CSV. Malware question	When player collides with a power up, a question at random will be selected from the txt document. Only if the question is answered correctly will the power up	Having a large range of multiple-choice questions will educate users on a wide range of malware and computer safety knowledge
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will get the chance to save their name by typing their name into an input. This will then be saved to a txt document and can be called upon to show a leader board.	High scores need to be saved. This is so they can be displayed, informing players on the scores to beat.
Enemies		Multiple enemies are created out of the window before moving into it. Give score when destroyed.	Give a target for the player to shoot at. Score is given for destroying enemies so there is a reason to do so.
Enemy shoot		Enemies should also fire projectiles for the players to dodge. These will be indestructible, so players will be forced to dodged	To increase engagement by increasing difficulty.

Rate Prototype //Score: 1 out of 5						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	0	4	3	3	2	Colours are too similar. Hard to track sprite and see player attacks on the background.
George	0	4	3	4	3	Add enemies
Callum	0	3	1	3	3	Poor quality when full screen (stretched resolution)
Drew	0	3	1	3	2	Lack of enemies negatively impacts engagement.

Sum	0	14	8	14	10
Average	0	3.5	2	3.5	2.5

Conclusion of Prototype 2

Concluding the second development cycle I have successfully implemented a GUI, health and shooting mechanic for the player. This sets me up for the next prototype, which will be the inclusion of enemies. The inclusion of enemies has been heavily requested by my stakeholders, throughout the development cycles, as they believe this will increase engagement of the game, which I agree with.

A new problem discussed was the screen size and colour choices. Due to the game being intended for computers and not phones, I should consider changing the screens orientation to be landscape. Another aesthetic choice that was questioned was the choice of colours used. Due to the colours used: tracking the player, player's attacks and eventually the enemies and their attacks were considered a potential issue. Therefore I should also consider a new colour pallet.

Prototype 3 - Enemies

For this prototype I shall be coding the enemies of the game. This inclusion will be the most important for player's engagement as the main game loop will be dodging and shooting the

enemies. Therefore, it is critical that the enemies are coded to a high standard. This will include having a variety of different enemies (including bosses); this will mean unique movement, shooting patterns, sprites and health. So that enemies have interesting movement and shooting I will potentially consider the use of using trig values (sin, cos and tan) so that patterns such as spirals can coded.

I will be coding the enemy class, enemy spawn rates, enemy movement, enemy health, enemy GUI and enemy shooting.

To start with I shall only code one enemy. This is to make sure that everything works. After I have one working enemy, I shall proceed to add a few more enemies.

Enemy Class

```
#pragma once

#include <iostream>
#include<SFML/Graphics.hpp>

class enemyEntity
{
protected:
    //enemy
    sf::Sprite enemySprite;
    sf::Texture enemyTex;
    float enemySpeed;
    int enemyDamage;
    unsigned int enemyScoreCount;
    int enemyScore;
    int enemyHp;
    int enemyHpMax;
    float eMove;
    float eMoveMax;
    bool go;
};
```

To start, I created an enemyEntity.hpp class. This will be used as a basis for all the other enemy classes. This is so that I can use inheritance and polymorphism, so that the code is a short as possible to save on its size and reduce compilation time. I created several variables which will then have their values set when initialised in their respective enemy classes.

```
#pragma once
#include "enemyEntity.hpp"

class EnemyA : protected enemyEntity
{
private:
    void initEnemyAVariables()
    {
        enemyScoreCount = 1000;
        enemySpeed = 0.7;
        enemyDamage = 5;
        enemyScore = enemyScoreCount;
        enemyHpMax = 1;
        enemyHp = enemyHpMax;
        eMoveMax = 50;
        eMove = eMoveMax;
    }
```

I then proceeded to make it so EnemyA class inherits from enemyEntity so that I can start with initialising variables. Each variable set here, will eventually be set in another enemy class with different values. The use of polymorphism saves me from repeating large sections

of code. This is because I plan to make the enemies similar with only small changes such as to their health and movement patterns. By coding the enemies in such a way will improve the code as there is improved efficiency, due to less code needing to be executed. This should hopefully reduce the space and time complexity of the code.

```
void initEnemyATex()
{
    //Load a texture from file
    if (!enemyTex.loadFromFile("Texture/MF_BOSS_INV2.png"))
    {
        std::cout << "EnemyA - Texture error - Could not load texture file." << "\n";
    }
}
```

After initialising the variables for EnemyA, I then initialised a texture for it. If this texture fails to load a custom error message will be outputted to the console so I can easily remedy the problem.

```
void initEnemyASprite()
{
    //Set the texture to the sprite
    enemySprite.setTexture(enemyTex);

    //Resize the sprite
    enemySprite.scale(1.0f, 1.0f);

    //set origin of the enmey to the middle rather than top left corner
    enemySprite.setOrigin(getBoundsE().width / 2, getBoundsE().top / 2);
}
```

Next, I set the now initialised texture to the EnemyA sprite and set the scale and origin. The texture's scale was good, so it is unchanged. However, I set the origin of the enemy sprite to its centre. This is so when I set its position it will be easier.

```
ublic:
    EnemyA(float pos_x, float pos_y) : enemyEntity()
    {
        initEnemyAVariables();
        initEnemyATex();
        initEnemyASprite();

        enemySprite.setPosition(pos_x, pos_y);
    }
```

I then created a constructor for the initialised variables, sprite and texture. The position is set through having parameter in the constructor, however the arguments in the game.cpp within the enemy spawn code.

```
const sf::Vector2f& getPos() const
{
    return enemySprite.getPosition();
}
```

This will return the position of the enemyA.

```
const sf::FloatRect getBoundsE() const
{
    return enemySprite.getGlobalBounds();
}
```

Will return the bounds of the enemy sprite.

```
const int& getScoreE() const
{
    return enemyScore;
}
```

Will return the value of score set to killing the respective enemy.

```
const int& getDamageE() const
{
    return enemyDamage;
}
```

Will return the damage value set to the enemy. This case refers to the damage the player will receive when colliding with the enemy's sprite.

```
const int& getHp() const
{
    return enemyHp;
}
```

Will return the value of the enemy's current health. This value decreases with collisions with the player's bullets.

```
//Max Hp
const int& getHpMax() const
{
    return enemyHpMax;
}
```

Will return the enemy's maximum health. This will be used for the enemy's health bar.

```
//Hp - Sets a new value for healt
void setHp(const int enemyHp)
{
    this->enemyHp = enemyHp;
}
```

Will set a new value for the enemyHp. This will overwrite the set value in enemyA initialisation.

```
//Hp - Decreases the value of health
void loseHp(const int value)
{
    enemyHp -= value;
}
```

This will decrease the enemy's health value by the parameter value. The enemy's health will change as the value will differ as the enemy may collide with the player's bullet or player's bomb.

```
void render(sf::RenderTarget* target)
{
    target->draw(enemySprite);
}
```

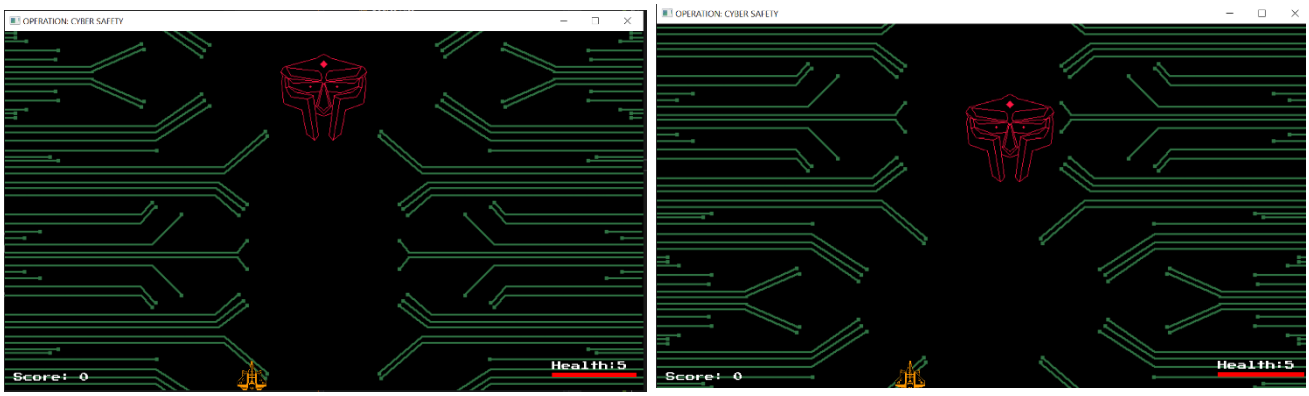
This will be called in the game file to render the enemy to the window.

Enemy Movement

```
void update()
{
    eMove += 0.5f;
    if (eMove >= eMoveMax)
    {
        eMove = -50.f;
    }
    if (enemySprite.getPosition().y <= 100.f)
        go = true;
    else
        go = false;
    if (go)
        enemySprite.move(3 * sinf(1.0f * 3.14159f / 20 + eMove), -1.5 * cosf(1.0f * 3.14159f / 1));
    else
        enemySprite.move(15 * sinf(1.0f * 3.14159f * eMove / -15 * -cosf(1.0f * 3.14159f * eMove)), 0);
}
```

This is the movement of EnemyA. This code will cause the enemy to move down the screen as in high frequency longitudinal wave, until EnemyA's sprite reaches a y position less than 150. Then, by using trig values and setting the y movement to, EnemyA will move left to right very quickly with only slight x movement. This gives the appearance of EnemyA vibrating vigorously. Currently due to trig value and eMove (a float variable which will have its value change each tick) EnemyA drifts towards the right. This is noticeable; however, I may leave this in due to the drift being barely noticeable and will likely prevent there from being any safe spaces when the enemy shooting is implemented.

Design Changes



Currently this is what EnemyA will look like. It is difficult to show the movement clearly. I also decided to make some changes to the aesthetics of the game, which I will quickly justify. In this version: the enemies, background and player have been changed to black and bold colours. This will be easier for players to quickly and easily track enemies, their attacks and the player sprite. Another change was the screen size. This has been changed so that when the screen is maximised the resolution will not be stretched. This will slightly change the game as I will now have more width to work with for enemy moveement. This will also mean I have to change position for the GUI (which have been partially done already) and the player spawn. The final change done is the position of the score and health. In previous iterations of the game the score was on the right, which would lead to problems if the player reaches a high score as the score will be partially rendered off of the screen. Therefore I swapped the health and score so that this issuse will no longer be a problem.

This should fix the concerns raised during the previous stakeholder feedback meeting.

Enemy Spawns

```
//Enemies A
float spawnTimerA;
float spawnTimerAMax;
std::vector<EnemyA*>enemiesA;
```

In the game.hpp I set spawnTimeA and spawnTimerAMax to be float data types. I also set the enemy to a vector and use dereference to enemy class. This is so that game.cpp can use variables from the EnemyA.hpp file.

```
//Initialise Enemies - Set time spawn timers
void Game::initEnemies()
{
    spawnTimerAMax = 1500.f;
    spawnTimerA = spawnTimerAMax;
}
```

I then set the spawn timer for enemy. I plan to do this so that as the game start the enemy is rendered, causing the spawnTimerA to be set to 0. This will then increase by a set value

until it is equal to spawnTimerAMax, causing another enemy to be rendered in. This process will repeat indefinitely.

```
//Constructor
//Calls the functions initialised above
Game::Game()
{
    initWindow();

    initGameBackground();

    initGui();
    gui->initGUI();
    gui->initScore();

    initPlayer();

    initBulletTextures();
    initBombTextures();

    initEnemies();
}
```

I then place the initEnemies (spawn timer) into the constructor.

```
//Delete enemies
for (auto* i : enemiesA)
{
    delete i;
}
```

In the destructor I use the dereference in a for loop to make sure enemies are deleted after they are destroyed or when the game is closed. This prevents any potential memory issues.

```
void Game::updateEnemiesA()
{
    //Spawning
    spawnTimerA += 0.1f;
    if (spawnTimerA >= spawnTimerAMax)
    {
        enemiesA.push_back(new EnemyA(window->getSize().x / 2, -100));
        spawnTimerA = 0.f;
    }
```

In the way I intended to, I create the spawn timer, which will be used to respawn the enemy. The value of spawnTimerA will continuously increment until equal or larger than spawnTimerAMax and cause the enemy to spawn at a set position. The value for spawnTimerA is set back to 0, and the process repeats.

Enemy Bounds

```
//Update
unsigned counter = 0;
for (auto* enemyA : enemiesA)
{
    enemyA->update();
    //Enemy Deletion - Enemy is deleted when reaches the bottom of the screen
    if (enemyA->getBoundsE().top > window->getSize().y)
    {
        //Delete enemy
        delete enemiesA.at(counter);
        enemiesA.erase(enemiesA.begin() + counter);
    }
```

enemyA->update() will call the update function from EnemyA class. This is the movement function; therefore this line will allow for the enemy movement.

Below this is the bounds code for EnemyA. This code will check if the top corner of EnemyA sprite is at the bottom of the window. If this is true, then Enemies A id deleted and erased. This will mean the enemy will no longer be taking up memory spaces and will no longer be rendered on the screen. To delete a vector such as an enemy, iterators must be used. The use of counter will assure the deletion of the appropriate enemy in the enemy array.

```
//Enemy player collision
else if (enemyA->getBoundsE().intersects(player->getBounds()))
{
    player->loseHp(enemiesA.at(counter)->getDamageE());
    delete enemiesA.at(counter);
    enemiesA.erase(enemiesA.begin() + counter);
}
++counter;
```

This code is similar to checking for window bounds, except this is for the player. If the player collides with the enemy sprite, then the player will lose health equal to the enemyDamage value. The enemy will then be deleted and erased.

Enemy Health

```
void Game::updateEncounterA()
{
    for (int i = 0; i < enemiesA.size(); ++i)
    {
        bool enemy_deleted = false;
        for (size_t k = 0; k < bullets.size() && enemy_deleted == false; k++)
        {
            if (enemiesA[i]->getHp() >= 1)
            {
                if (bullets[k]->getBounds().intersects(enemiesA[i]->getBoundsE()))
                {
                    //enemy looses 1 health
                    enemiesA[i]->loseHp(1);

                    //bullet is delted from the game
                    delete bullets[k];
                    bullets.erase(bullets.begin() + k);
                }
            }
```

This section of code will check if the enemy has more than 1 health point. If the enemy is hit by a bullet then the enemy will lose 1 health point and the bullet will deleted.

```
else
{
    //Collision - enemy and bullet collide - score is added, bullet and enemy are deleted
    if (bullets[k]->getBounds().intersects(enemiesA[i]->getBoundsE()))
    {
        //add enemy score (set in Enemy.hpp) to score
        gui->score += enemiesA[i]->getScoreE();
        //enemy is delted from the game
        delete enemiesA[i];
        enemiesA.erase(enemiesA.begin() + i);
        //bullet is delted from the game
        delete bullets[k];
        bullets.erase(bullets.begin() + k);
        enemy_deleted = true;
    }
}
```

If the player shoots the enemy with a bullet when it has less than 1 health though, then the score gui will increase by the amount of score set within the enemy's class. The enemy and bullet are then deleted and erased.

```
for (size_t k = 0; k < bombs.size() && enemy_deleted == false; k++)
{
    if (enemiesA[i]->getHp() >= 1)
    {
        if (bombs[k]->getBounds().intersects(enemiesA[i]->getBoundsE()))
        {
            //enemy looses 1 health
            enemiesA[i]->loseHp(3);
            //bullet is delted from the game
            delete bombs[k];
            bombs.erase(bombs.begin() + k);
        }
    }
```

This code is similar to bit above, except it works with the player's bomb attack. Unlike a bullet attack, the bomb will do 3 health points of damage to an enemy in comparison to the bullet's 1 damage.

```
else
{
    //Collision - enemy and bullet collide - score is added, bullet and enemy are deleted
    if (bombs[k]->getBounds().intersects(enemiesA[i]->getBoundsE()))
    {
        //add enemy score (set in Enemy.hpp) to score
        gui->score += enemiesA[i]->getScoreE()*1.5;
        //enemy is delted from the game
        delete enemiesA[i];
        enemiesA.erase(enemiesA.begin() + i);
        //bullet is delted from the game
        delete bombs[k];
        bombs.erase(bombs.begin() + k);
        enemy_deleted = true;
```

Yet again this code is similar to ones above. However, this differs due to it involving bombs (not the player's bullets) and the fact that killing an enemy with a bomb will give the player an increased score from the enemy. This will encourage a riskier game style, which keep the most experience players engaged, as they consolidate their knowledge.

```
//Update
void Game::update()
{
    player->update();

    updateCollision();

    updateGUI();

    updatePlayerAttackInput();
    updatePlayerAttack();

    updateGameBackground();

    updateEnemiesA();

    updateEncounterA();
}
```

The new enemy functions in the game.cpp class have been included in the game class.

```
//Draw enemy
for (auto* enemyA : enemiesA)
{
    enemyA->render(window);
}
```

This will render the enemies to make sure that the enemy will spawn will be visible to the player.

Enemy GUI

It is important for the enemy to have a graphical user interface for the enemies. By this I am referring to the need for enemies to have a health bar; this is important as it provides the user how much health the enemy has and how quickly they will die, after shooting it for a short amount of time. Unlike players, I will only implement a health bar. This is because I believe having numbers will make the screen to busy and players will focus too much on the number. In this fast-paced genre users shouldn't be fixated on numbers, but just glance to quickly figure if the enemy has much or little health. I believe a health bar would be best for this.

The enemy has had its health set up in last section, therefore I am able to start straight away by coding the health bar.

```
//GUI - health bar and text
sf::Text healthText;
sf::RectangleShape playerHpBar;
sf::RectangleShape eAHpBar;
```

In the section where I set up the player's health text and health bar, I added the rectangle shape for the enemy's health bar. SFML rectangle shape allows me to create a rectangle easily.

```
//GUI - Enemy A
eAHpBar.setSize(sf::Vector2f(50.f, 7.f));
eAHpBar.setFillColor(sf::Color::Red);
```

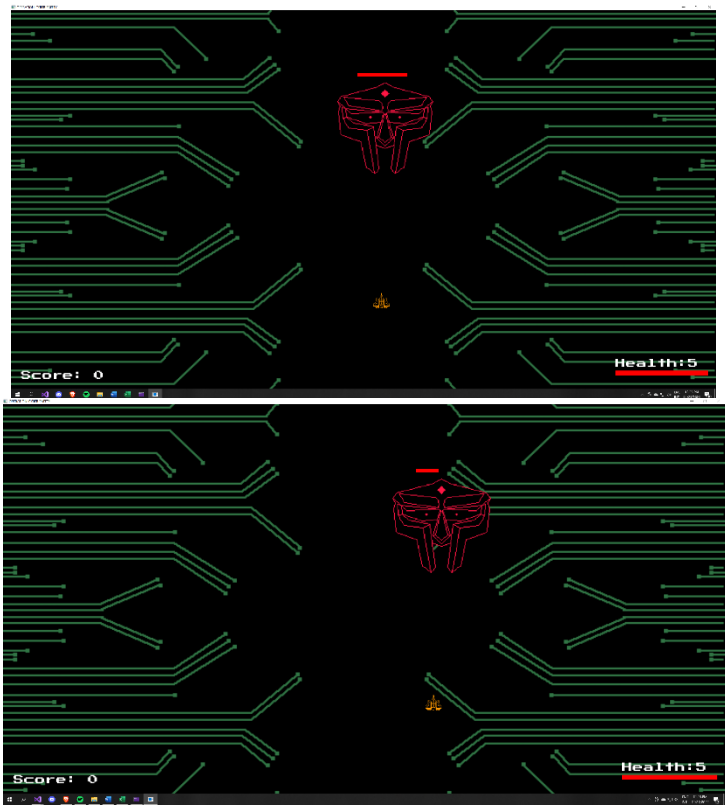
I then set up the enemy the size and colour of the health bars in the initGUI() function. I have made the enemy health to larger size to the player's health bar as the enemy is bigger size.

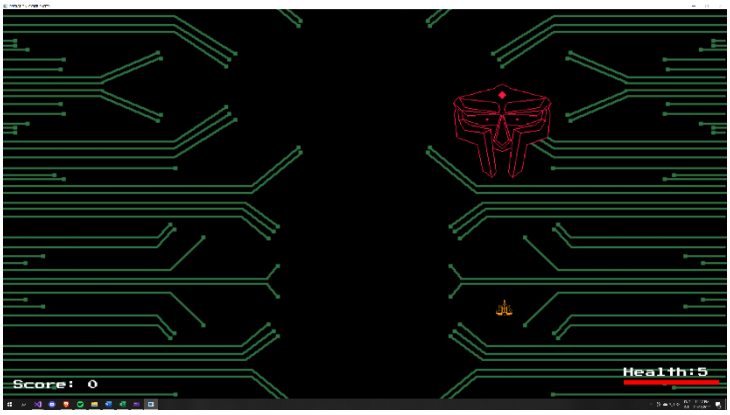
```
//Enemy Health
for (auto* eA : enemiesA)
{
    float eAHpPercent = static_cast<float>(eA->getHp()) / eA->getHpMax();
    gui->eAHpBar.setSize(sf::Vector2f(100 * eAHpPercent, gui->eAHpBar.getSize().y));
    gui->eAHpBar.setPosition(eA->getPos().x - 50, eA->getPos().y -20);
}
```

In the update GUI function in the game.cpp file I have included the enemy health bar changes. The first line of code in the for loop gets will divide the current health by the maximum health. This will give values such as 1 (if the enemy is on full health) and 0.5 (if the enemy is on half health). This will affect the health bar as the size will shrink as the x value is set to the value of eAHpPercent. The Position of the health bar will set to follow the enemy. This is done by setting the coordinate of the enemy minus an x value and y value. This is so that the health bar will stay above the enemy.

```
window->draw(gui->eAHpBar);
```

This health bar is drawn to the screen alongside the rest of the GUI in render GUI function.





Currently the health bar works mostly as it should, however a slight bug is that whenever the enemy has 0 health it will still take one more shot before they are killed. This is not detrimental to the project, however if I have time at the end of this project, I may return to fix the issue. However, this is currently serviceable.

Enemy Shoot

Enemy shooting is very important to the project, as the gameplay is heavily reliant on the enemies having interesting shooting patterns for the player to avoid. It is important to find a way to make the shooting pattern slightly difficult, but not too hard that it is frustrating.

As enemies and the player both shoot, I can use variables set in the Attack.hpp file.

```
#include <iostream>
#include<SFML/Graphics.hpp>

class Attack
{
protected:
    //Private Definitions
    //Define - Attack sprite
    sf::Sprite attackSprite;

    //Define -  direction and movementSpeed
    sf::Vector2f attackDirection;
    float attackSpeed;
};
```

This section of code has not been altered since it was used for the player.

```
class enemyABullet : public Attack
```

First, I need to inherit the variables that I have set. This use of inheritance improves efficiency as I do not need to repeat this section of code. This reduces the memory and the execute time.

```
//Bullet - Texture, x and y position, x and y direction, bullet's movement speed
enemyABullet(sf::Texture* texture, float pos_x, float pos_y, float dir_x, float dir_y)
```

First, I set up a constructor for the enemy attack. In this constructor I use parameters which will set a texture; the x and y coordinate that the attack will be drawn to when drawn; and the direction the attack will move.

```
attackSprite.setTexture(*texture);
attackSprite.setOrigin(8.0f, 8.0f);

attackSprite.setPosition(pos_x, pos_y);

attackDirection.x = dir_x;
attackDirection.y = dir_y;
attackSpeed = 1;
```

In the constructor I set the attack speed as 1 and the origin of the enemy as (8,8). These values can be set here. I shall set the rest in the updateAttack function found in the game.cpp file. This is because for the other values rely on values set elsewhere in the code. An example of this is enemy position. It is easier to call the enemy position in the game.cpp file than to call it in the enemyAttack file.

```
//Bounds - Returns the bounds of the bullet
const sf::FloatRect getBounds() const
{
    return attackSprite.getGlobalBounds();
}
```

This function will return the bounds for the enemy attack sprite.

```
//Update Bullets
void update()
{
    attackSprite.move(attackSpeed * attackDirection);
}
```

This function is responsible for moving the attack in the direction of the of the attack direction multiplied by the speed. Attack direction is constantly changing, meaning the direction of the attack will create a pattern for the user to avoid.

```
//Initialise Enemy Attack Fire Rate / Fire rate
void Game::initAttackPattern()
{
    incrementMaxA = 360.f;
    incrementA = incrementMaxA;
}
```

This function sets the variables incrementA and incrementMaxA. These variables will be used to help create an attack pattern for the enemy I have created.

```
void Game::updateEnemyAttackBounds()
```

Inside of this function, there is for loops which are used to delete all the enemy's attacks, when they are no longer on the screen. This is important as enemies will be shooting a large quantity of attacks in quick succession. Without any deletion, there will be dire performance issues. This will be different to player attack deletion, because the side of the screens will also have to be considered now.

```
unsigned counter3 = 0;
for (auto* eAA : eAAttack)
{
    //Bullet deletion - top | bottom | left | right
    if (eAA->getBounds().top < 0.f |
        eAA->getBounds().top + eAA->getBounds().height >= window->getSize().y |
        eAA->getBounds().left < 0|
        eAA->getBounds().left + eAA->getBounds().width >= window->getSize().x)
    {
        //Delete bullet
        delete eAAttack.at(counter3);
        eAAttack.erase(eAAttack.begin() + counter3);
        //Print number of bullets - Checks to make sure bullets are deleted when out of bounds
        std::cout << eAAttack.size() << "\n";
    }
    ++counter3;
}
```

This for loop will check to see if the enemies attack is no longer on the window. This is done by comparing the bounds of the attack to the bounds of the window. When the bounds of the attack is less than the top or left of the window or more than the bottom or the right of the window, the if statement runs. This if statement will delete the attack and erase it. The console will then output the number of bullets that are then left on the screen. Outputting the number of bullets is useful as it will allow me to check to see the bullet culling is working as it should.

```
void Game::updateEnemyAAttack()
{
    incrementA += 0.1f;
    if (incrementA >= incrementMaxA)
    {
        incrementA = 0.f;
    }

    for (auto* eA : enemiesA)
    {
        if (eA->canAttack())
        {
            eAAttack.push_back(
                new enemyABullet(
                    eAAttacktTex["EAATTACK"],
                    eA->getPos().x,
                    eA->getPos().y + eA->getBoundsE().height / 2,
                    tanf(cosf(2.7 * (sinf(3.14159f / 360 + (incrementA))))), tanf(sinf(2.7 * (cosf( 3.14159f / 360 + ((incrementA)))))));
        }
    }
}
```

This code is a continuation to the code in the enemyAttack.hpp file, as it has the arguments of the parameters. Before the arguments, I have code which will increment incrementA by the value of 0.1 every tick. This will continue until it reaches incrementMaxA (set to 360), then the branching statement if will run. This will set the value back down to 0 for the code to continue, this will be used below for the attack pattern.

An if statement is used pointing to canAttack. When canAttack is set to true, the if statement runs. This code is responsible for firing the attack as well as setting the texture; setting the position to the enemy position; and the y and x direction the attack will go in.

Using trigonometry functions such as tan, sin and cos as well as the incrementing value will allow for an interesting pattern to fir ed.

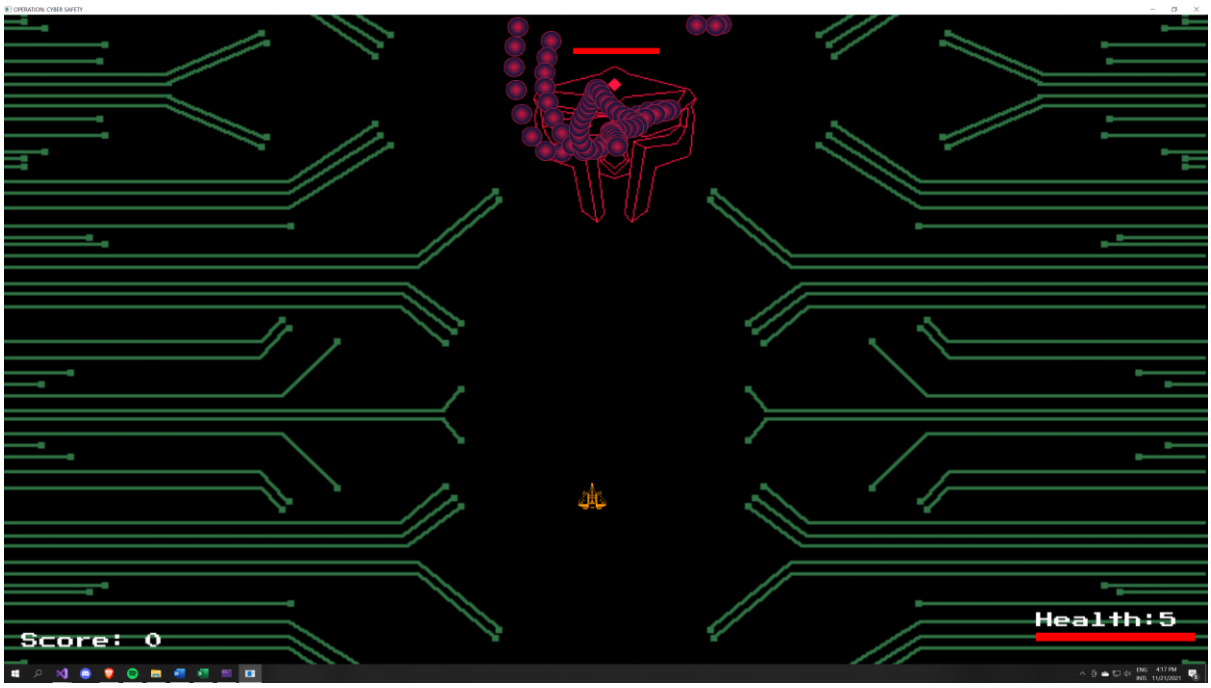
```
for (auto* eAA : eAAttack)
{
    eAA->update();
}
```

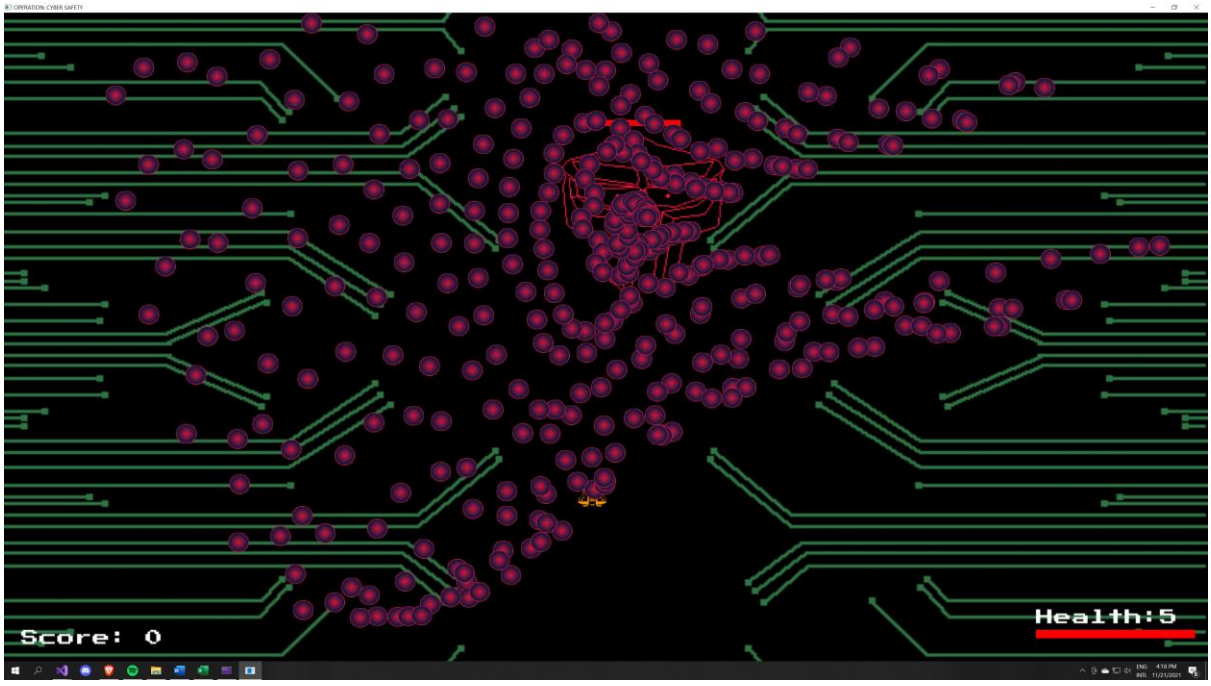
The update function from enemyAttack.hpp is then called in the Game.cpp update function. This will allow the code from that file to be updated every tick.

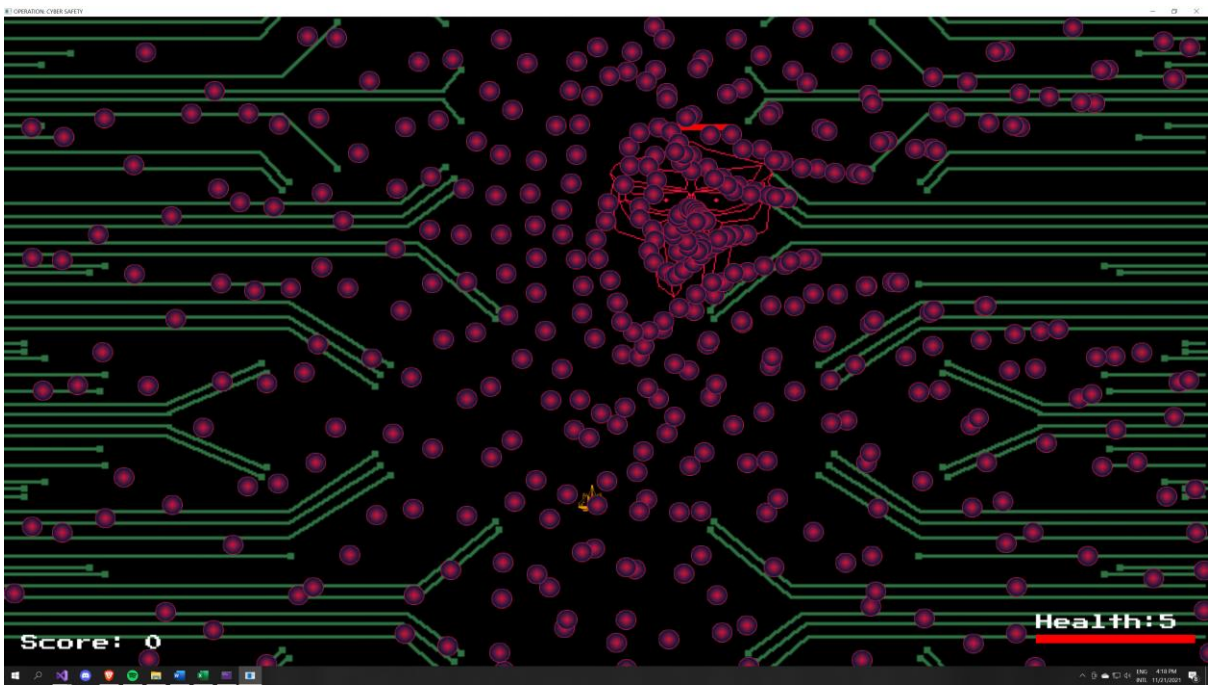
```
//Draw enemy attacks
for (auto* eAA : eAAttack)
{
    eAA->render(window);
}
```

The enemy attack is then drawn to the window each frame.

With all this code the enemy will now be able to attack. However, there are currently no collisions with the player, but the pattern created will look like:


Score: 0
Health:5


Score: 0
Health:5



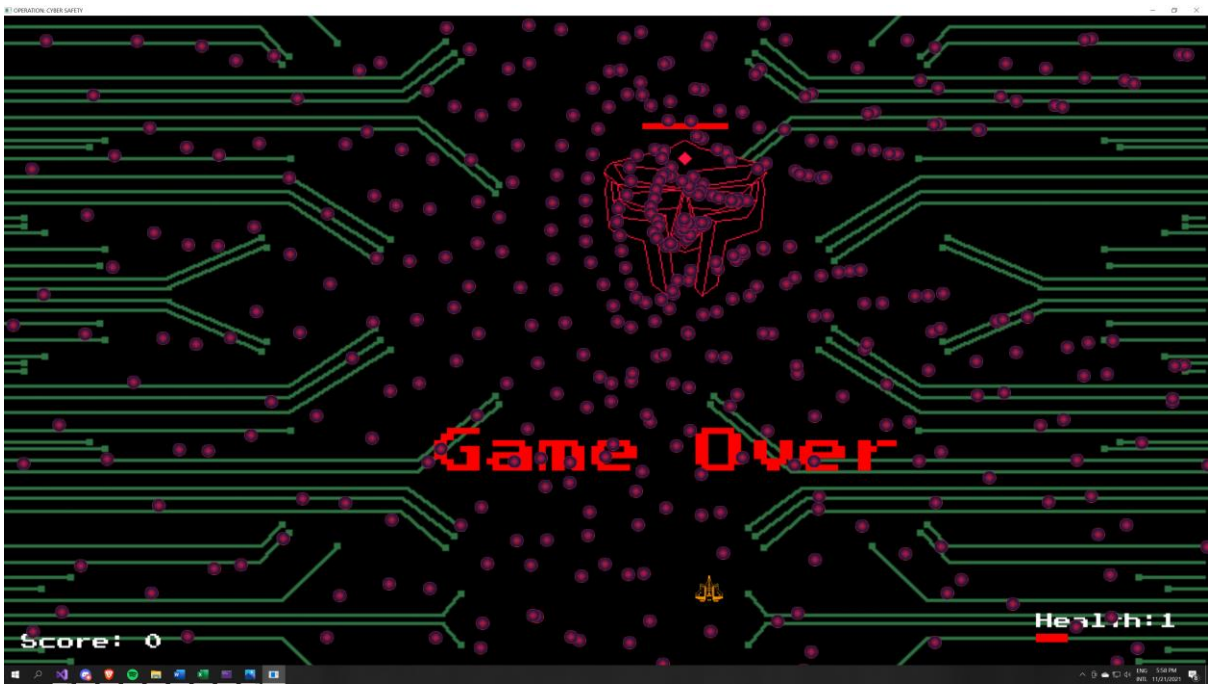
An interesting pattern did form, however there are some problems that are immediately apparent. The first problem is that the amount of attacks maybe too difficult to avoid due to there approximately being 420 attacks on the screen all at once. The other problem is that the player can easily go to the bottom right of the screen and then manoeuvre to the right of the enemy allowing for a safe place as this pattern doesn't seem to fire much in that direction.

However possible remedies for this could be making slight alteration to the trig values used and by reducing the size of the attacks or the player making it easier to dodge. Alternatively, I could experiment with the inclusion of a shield of some kind to make the player immune to attacks for short periods. However, this will have to be heavily tested as this could potentially be to strong if not properly tuned.

Before I make any changes, I shall first consult the stakeholders and include collisions:

```
for (size_t e = 0; e < eAAttack.size(); e++)
{
    if (eAAttack[e]->getBounds().intersects(player->getBounds()))
    {
        //player looses 1 health
        player->loseHp(1);
        //bullet is delted from the game
        delete eAAttack[e];
        eAAttack.erase(eAAttack.begin() + e);
    }
}
```

Adding collisions for player against the enemy attacks is simply this code which I added to updateEncounterA() function. This will check to see if player bounds collide with enemy attacks. When they do collide then player loses 1 health point, and the attack is deleted.



With collisions now working I now know the health works. When the player gets hit 5 times the game is designed so that it currently stops updating. This causes the game to stop updating before the health bar and text changes to 0 (but the actual health values are working fine). This is a slight issue that should be fixed when a replay button is added as I will change it so that the game will still update.

Name of Test		Data	Expectation	Justification
Key W		W key	The player's character moves up the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key A		A key	The player's character moves left across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable

				to move the game is unplayable and the project is a failure.
Key S		S key	The player's character moves down the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key D		D key	The player's character moves right across the screen	The character must be able to move to collide with power ups and avoid attacks. If the player is unable to move the game is unplayable and the project is a failure.
Key Up		Up key	The player's character shoots its projectile attack in a linear line (may change with powerups, effects have not been decided yet).	The key feature of a bullet hell SHMUP game is dodging bullets, but also shooting back at enemies. If the player can only dodge the game will become boring and overwhelming as enemies will spawn without any being destroyed.
Play Button		Click button / 1 key	Setting this variable to true takes player from main menu or end of a level into a new game.	If the player is unable to use the play button, they will not be able to access the game. Meaning the project has failed.

Options Button		Click button / 2 key	Setting this variable to true should take the player into the options.	Options button is not as important as the play button, however, is still important. This is because the options menu will have accessibility features such as colour-blind mode.
Leader board Button		Click button / 3 key	Setting this variable to true should take the player to the leader board screen. The high score.txt file should appear in an aesthetically pleasing way.	Arguably the least important of the buttons. This button still needs to work though, because this will show high scores providing a social element. Social elements (as said at the start) helps improve engagement and overall retention rate.
Sound		MP3 file	Music should be playing during the main menu, when a button is clicked and when collisions happen	Including sound is important because it increases immersion to the game. If the game didn't have any sound at all, it would be boring and the player wills struggle to remain engaged for long periods of time
Level end		High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	If the player id destroyed and there is no way to restart or open a menu, they will be forced to close and reload the game. This would be frustrating and reduces engagement.

Collision		Player coordinates = MalwareAttack coordinates and Malware coordinates = PlayerAttack coordinates and Player coordinates = PowerUp coordinates	If player collides with an attack, they lose a health point; when player health equals 0 game ends. If malware collides with the player's attack, they lose a health point; when malware health equal 0 they are destroyed. If player collides with power up a multiple choice appears on screen and game is paused. If question is answered correctly the power up is granted for a short time.	If collisions do work or work poorly the game may feel unfair as an enemy attacks lack consistency. This would make the game frustrating to play and reduces replay-ability as players would not want to play a game where their runs are being cut short due to poorly coded collisions. Alternatively, if collisions don't work at all the game will continue indefinitely making the game pointless. The information would not be accessed either as currently the only way user are educated is through the use of multiple choice questions asked when collision is made with power ups. If done poorly or not at all, means the project would have failed
High Score		Time, score	High score is generated based on time survived and malware destroyed.	Including a high score feature improves replay-ability as players will want to beat theirs and others high scores
Colour Blind mode		Options menu data	When enabled game colour changes	To make the game more accessible to a

				wider audience, the game should include a way for people to see what is happening.
Read text from file. Multiple choice question		.txt document with CSV. Malware question	When player collides with a power up, a question at random will be selected from the txt document. Only if the question is answered correctly will the power up	Having a large range of multiple-choice questions will educate users on a wide range of malware and computer safety knowledge
Save High Score		.txt document with CSV on scores and names	At the end of a game, the user will get the chance to save their name by typing their name into an input. This will then be saved to a txt document and can be called upon to show a leader board.	High scores need to be saved. This is so they can be displayed, informing players on the scores to beat.
Enemies			Multiple enemies are created out of the window before moving into it. Give score when destroyed.	Give a target for the player to shoot at. Score is given for destroying enemies so there is a reason to do so.
Enemy shoot			Enemies should also fire projectiles for the players to dodge. These will be indestructible, so players will be forced to dodged	To increase engagement by increasing difficulty.

Rate Prototype //Score: 1 out of 5						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment

Tom	0	4	4	4	4	The attack pattern is too difficult
George	0	5	3	4	3	The game could still have more enemies
Callum	0	3	3	4	3	Game does serve its original purpose of being educational
Drew	0	4	3	3	4	Could include more enemies and needs education
Sum	0	16	13	15	14	
Average	0	4	3.25	3.75	3.5	

Conclusion

First thing that was considered the difficulty. The difficulty was high and immediate due to the enemy attacks and player having large hit boxes. This causes collisions to be very frequent. Possible solution discussed was giving the player a higher health pool, a way to regain health, a shield button and changing the hitboxes. Changing the player hitbox so that the centre of the sprite would have to collide with attack for there to collision. I like this idea the most, because this will not take away from the sprites design and improve overall gameplay.

Even with the inclusion of one main enemy, with a large amount of health (bullet sponge), many stakeholders believe only one enemy is still too simple and will not provide enough of a challenge when the hitboxes are changed. Therefore, to remedy this, I shall implement some more enemies. This maybe through using another boss like enemy or many smaller enemies. I believe I may explore both these options as the smaller enemies will break up the monotony of shooting a larger enemy and by having a couple larger enemies provide more entertaining gameplay loop. By having many bosses, my game will become more of boss rush style SHMUP. This will likely provide the most cohesive and engaging gameplay loop.

In the next iteration I shall be adding one (possibly two) low tier enemies and one (possibly two) high tier enemies (like what is there currently). This will be done similarly to how I have done before, but with more types. I shall have to attempt to make sure not to repeat too much code, as this will decrease the efficiency. This can be done with inheritance, encapsulation, and polymorphism. This iterations will also include changes to the player's hitbox, enemy's attacks hitboxes and possible changes to where the code for the enemies is (due to the large size for game.cpp file).

However, an issue one stakeholder had was the game crashing due to this error:

ADD ERROR SCREENSHOT HERE

I believe this is likely caused due to…

I aim to remedy this by

Prototype 4 – Enemies Improved

In this prototype there are key areas my stakeholders have elucidated to me to focus on. These include having a variety of enemies and enemy types, improve gameplay (bullet patterns), and making the game infinite by better utilising the spawn function.

I will also attempt to see if I can make my code more efficient, by attempting to reorganise sections of code to make better use of polymorphism. This will improve the space complexity and make the code more efficient reducing execution time. The code will then have a faster execute time, which will improve player experience as it will go faster.

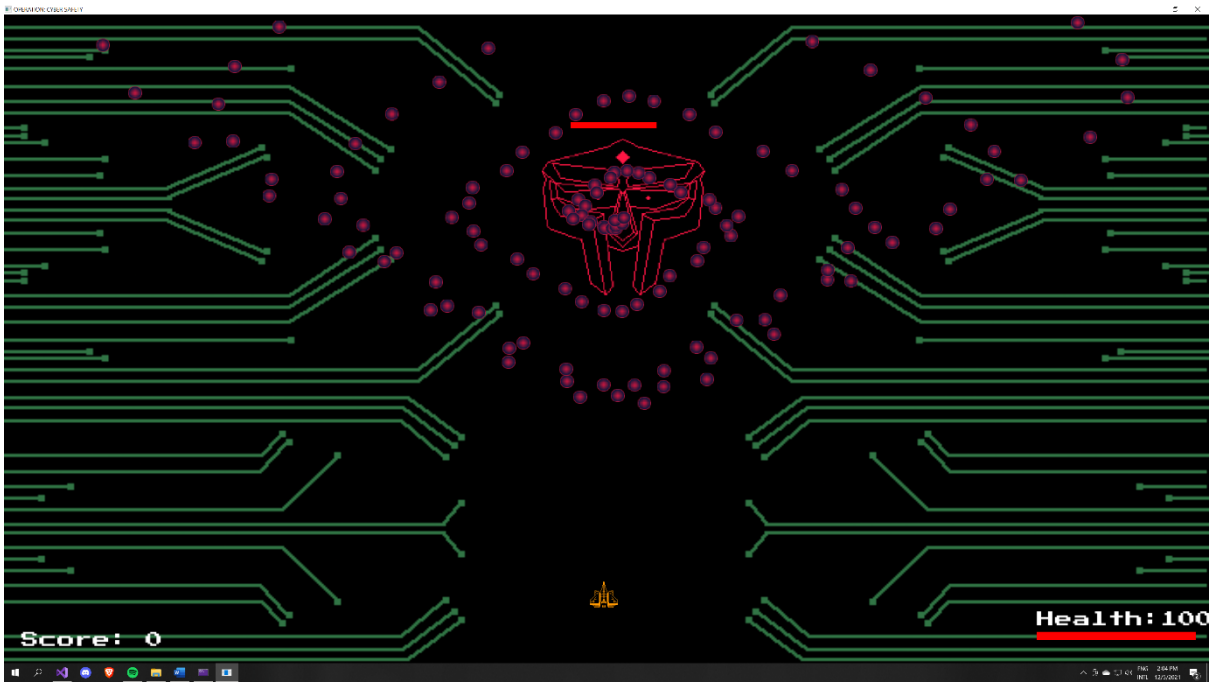
The final thing I investigate in this prototype is the error that is causing my game to crash sometimes in my code.

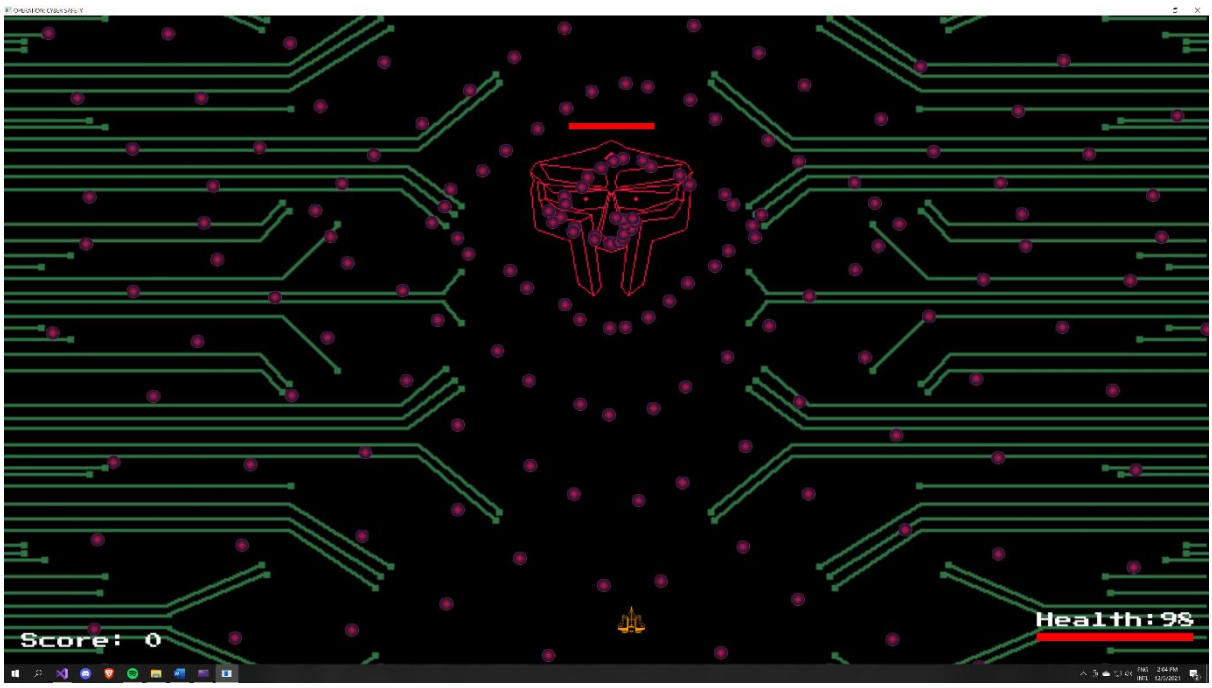
Updated attack pattern

```
3 * tanf(1.0f * 3.14159f / 100 + -sinf(incrementA)), 2.0f * tanf(1.0f * 3.14159f / 100 + -cosf(incrementA))));
//(cosf(2.7 * (sinf(3.14159f / 360 + (incrementA))))), (sinf(2.7 * (cosf( 3.14159f / 360 + ((incrementA))))))));
```

The first change I made was the attack pattern for the enemy from the previous iteration. Under this I have the old math code for the previous pattern.

This would change the attack pattern to appear as it does in the images below.


Score: 0
Health:100


Score: 0
Health:98

Enemy B hpp and efficiency update

Now that I have updated this, I then moved on to adding more enemies to my game. To do this I continued to use encapsulation and inheritance. To ensure the code is as efficient as I can make it with my ability, I am currently testing ways to make the code that overlaps significantly into enemy entity class, alongside the defining of variables.

```
class EnemyB : protected enemyEntity
{
private:
    void initEnemyBVariables()
    {
        enemyScoreCount = 1000;
        enemySpeed = 0.7;
        enemyDamage = 5;
        enemyScore = enemyScoreCount;
        enemyHpMax = 30;
        enemyHp = enemyHpMax;
        eMoveMax = 70;
        eMove = eMoveMax;
        attackCoolDownMax = 1.0;
        attackCooldown = attackCoolDownMax;
    }
```

First, I made good use of encapsulation and inheritance, by using variables defined in enemy entity class. This reduces the amount code that will need to be executed.

```
    void initEnemyBTex()
    {
        //Load a texture from file
        if (!enemyTex.loadFromFile("Texture/Lord_Quas_Para2.png.png"))
        {
            std::cout << "EnemyB - Texture error - Could not load texture file." << "\n";
        }
    }
    void initEnemyBSprite()
    {
        //Set the texture to the sprite
        enemySprite.setTexture(enemyTex);

        //Resize the sprite
        enemySprite.scale(1.0f, 1.0f);

        //set origin of the enmey to the middle rather than top left corner
        enemySprite.setOrigin(getBoundsE().width / 2, getBoundsE().top / 2);
    }
```

Similar to how I initialised the texture and sprite of enemy A, I loaded an image from my texture file as well as set a custom error message to know where the error will be. Then I set the texture to the sprite; added a scale so that I can easily adjust the size of the enemy if it is too big or small; then I set the origin of the sprite to be in its centre.

```
public:
    //returns the enemy postion
    const sf::Vector2f& getPos() const
    {
        return enemySprite.getPosition();
    }
    //return the enemy global bounds
    const sf::FloatRect getBoundsE() const
    {
        return enemySprite.getGlobalBounds();
    }
    //return the score given to enemy
    const int& getScoreE() const
    {
        return enemyScore;
    }
    //return the damge value given to enemy
    const int& getDamageE() const
    {
        return enemyDamage;
    }
    //return hp value
    const int& getHp() const
    {
        return enemyHp;
    }
    //return Max Hp
    const int& getHpMax() const
    {
        return enemyHpMax;
    }
    //Hp - Sets a new value for health
    void setHp(const int eHp)
    {
        enemyHp = eHp;
    }
    //Hp - Decreases the value of health
    void loseHp(const int value)
    {
        enemyHp -= value;
    }
};
```

As stated above, I changed my previous code so that the code is as efficient as I can make it. One way I did this in this prototype was by making better use of inheritance. The code above was originally in the EnemyA.hpp file, meaning that all this code would need to be in all other enemy files. By relocating this code to enemyEntity.hpp, this code will now be inherited and not needed to be duplicated unnecessarily.

```
public:
    //Constructor
    EnemyB(float pos_x, float pos_y) : enemyEntity()
    {
        initEnemyBVariables();
        initEnemyBTex();
        initEnemyBSprite();

        enemySprite.setPosition(pos_x, pos_y);
    }
```

Back to enemy b code, I created a constructor for the enemy which will contains to everything I had initialised the spawn position as well with parameters. This spawn position argument will add later, as I currently am not sure where I want it.

```
//Movement
void updateMove()
{

}
```

I have an empty movement constructor as I currently am not sure how I want this enemy to move. This will likely be decided at the end.

```
//Attack
const bool canAttack()
{
    //attack colldown is larger than attack cooldown max then attack cooldown is set back to 0 and true is returned
    if (attackCooldown >= attackCoolDownMax)
    {
        attackCooldown = 0.f;
        return true;
    }
    //if the if statement is not true then false is returned
    return false;
}
```

This part of the attack cooldown will not differ from the other enemies, so I decided to move this to enemyEntity.hpp file as well.

```
//Attack cooldown
void updateAttack()
{
    //while the attack cooldown is less than cooldown max increase attack cooldown by 0.5 every tick
    if (attackCooldown < attackCoolDownMax)
        attackCooldown += 0.5f;
}
```

Originally, this part of the attack cooldown, was going to be in each enemy file, but after some consideration this can be included in enemy entity as well. This was because I thought that if I want enemies to have different fire rates, I would have had to change this value. However, I can just change the attackCoolDownMax value to do this.

```
void update()
{
    updateMove();
    updateAttack();
}
```

I then place the attack and move code into an update function which will be called in the game update function later.

```
void render(sf::RenderTarget* target)
{
    target->draw(enemySprite);
}
```

Finally, I included enemySprite render function.

Enemy B GUI

```
sf::RectangleShape eBHpBar
```

I first defined the variable of rectangle. Just like with the previous health bars, I shall have rectangles that decrease in size when an enemy takes damage.

```
//GUI - Enemy B
eBHpBar.setSize(sf::Vector2f(50.f, 7.f));
eBHpBar.setFillColor(sf::Color::Red);
```

I then set the rectangle to have the same size as the other enemy health bar and then set the colour to red.

```
std::vector<EnemyB*> enemiesB;
```

So that I can start implementing the GUI code into the game I must create dereference within the game files.

```
for (auto* eB : enemiesB)
{
    float eAHpPercent = static_cast<float>(eB->getHp()) / eB->getHpMax();
    gui->eAHpBar.setSize(sf::Vector2f(100 * eAHpPercent, gui->eAHpBar.getSize().y));
    gui->eAHpBar.setPosition(eB->getPos().x - 50, eB->getPos().y - 20);
}
```

I then used the same code I used for the other enemy health bar. There may be a way to make this more efficient and not have duplicated code, however this beyond my capabilities.

Enemy B Game file

```
//Delete enemies
for (auto* i : enemiesA, enemiesB)
{
    delete i;
}
```

First addition I made was the inclusion of enemies B in the destructor. This means that the enemy B is deleted when it not needed. This reduces the memory used.

```
void Game::initEBAttackTex()
{
    eBAttackTex["EBATTACK"] = new sf::Texture();
    eBAttackTex["EBATTACK"]->loadFromFile("Texture/Lord_QuasS.png");
}
```

I have set the texture to the new enemy attack.

```
spawnTimerBMax = 800.f;
spawnTimerB = spawnTimerBMax;
```

I then set the spawn timer in the same function where I did this for the spawn timer for enemy A

```
unsigned counter7 = 0;
for (auto* eBA : eBAttack)
{
```

Allows me to reference enemy B attack, in the game file.

```
    if (eBA->getBounds().intersects(player->getBounds()))
    {
        //Delete bullet
        //player looses 1 health
        player->loseHp(1);
        //delete eAAttack.at(counter3);
        eBAttack.erase(eBAttack.begin() + counter7);
        //Print number of bullets - Checks to make sure bullets are deleted when out of bounds
        std::cout << eBAttack.size() << "\n";
    }
```

If the enemy B attacks collide with the player then the player loses 1 health and the attack sprite is then deleted and does not

```
//Bullet deletion - top | bottom | left | right
if (eBA->getBounds().top + eBA->getBounds().height >= window->getSize().y or eBA->getBounds().left + eBA->getBounds().width >= window->getSize().x or eBA->getBounds().top < 0.f or eBA->getBounds().left < 0.f)
{
    //Delete bullet
    delete eBAttack.at(counter7);
    eBAttack.erase(eBAttack.begin() + counter7);
    //Print number of bullets - Checks to make sure bullets are deleted when out of bounds
    std::cout << eBAttack.size() << "\n";
}

++counter7;
```

The bullets from the second enemy will be deleted from the screen. This means when the bullet is not on the screen it is deleted. To check that the bullet is being deleted, I have the bullet number printed to console.

```
//Spawning
spawnTimerB += 0.1f and gui->score == 1000;
if (spawnTimerB >= spawnTimerBMax)
{
    enemiesB.push_back(new EnemyB(50, -100));
    spawnTimerB = 0.f;
}
```

The spawning of enemy B works by having a spawn counter and a way to check the score. Currently, the enemy will spawn and no more will spawn until 1000 score is reached. Once this happens enemy will spawn when the clock reaches the value set at the top.

```
//Update
unsigned counter6 = 0;
for (auto* enemyB : enemiesB)
{
    enemyB->update();
    //Enemy Deletion - Enemy is deleted when reaches the bottom of the screen
    if (enemyB->getBoundsE().top > window->getSize().y)
    {
        //Delete enemy
        delete enemiesB.at(counter6);
        enemiesB.erase(enemiesB.begin() + counter6);
    }
```

If enemy reaches bottom of the screen the enemy is deleted. This is so that the enemy doesn't carry on moving without being shown on the screen. This will prevent the performance being decreased.

```
//Enemy player collision
else if (enemyB->getBoundsE().intersects(player->getBounds()))
{
    player->loseHp(enemiesB.at(counter6)->getDamageE());
    delete enemiesB.at(counter6);
    enemiesB.erase(enemiesB.begin() + counter6);
}
++counter6;
```

This section is responsible for enemy collisions with player. This will result in the player losing health = to the value set at getDamgeE function in the enemy class. Enemy B will then be deleted and erased from the window and game.

```
for (int i = 0; i < enemiesB.size(); ++i)
{
    for (size_t e = 0; e < eBAttack.size(); e++)
    {
        if (eBAttack[e]->getBounds().intersects(player->getBounds()))
        {
            //player looses 1 health
            player->loseHp(1);
            //bullet is delted from the game
            delete eBAttack[e];
            eBAttack.erase(eBAttack.begin() + e);

        }
    }
```

This code is the enemy B attacks. If this enemy's attack collides with the player then the player will lose 1 health point and the attack is then deleted. This is important because if the attack is not deleted for every frame the player collides with the attack then the player will lose 1 health and continue to travel. To prevent the game from being too difficult this should be deleted.

```
bool enemy_deleted = false;
for (size_t k = 0; k < bullets.size() && enemy_deleted == false; k++)
{
    if (enemiesB[i]->getHp() >= 1)
    {
        if (bullets[k]->getBounds().intersects(enemiesB[i]->getBoundsE()))
        {
            //enemy looses 1 health
            enemiesB[i]->loseHp(1);

            //bullet is delted from the game
            delete bullets[k];
            bullets.erase(bullets.begin() + k);
        }
    }
```

This code is similar to the one above, however it is reverse. When a player bullet collides with an enemy the enemy will lose 1 health and is then deleted on collision.

```
for (size_t k = 0; k < bombs.size() && enemy_deleted == false; k++)
{
    if (enemiesB[i]->getHp() >= 1)
    {
        if (bombs[k]->getBounds().intersects(enemiesB[i]->getBoundsE()))
        {
            //enemy looses 1 health
            enemiesB[i]->loseHp(3);
            //bullet is delted from the game
            delete bombs[k];
            bombs.erase(bombs.begin() + k);
        }
    }
```

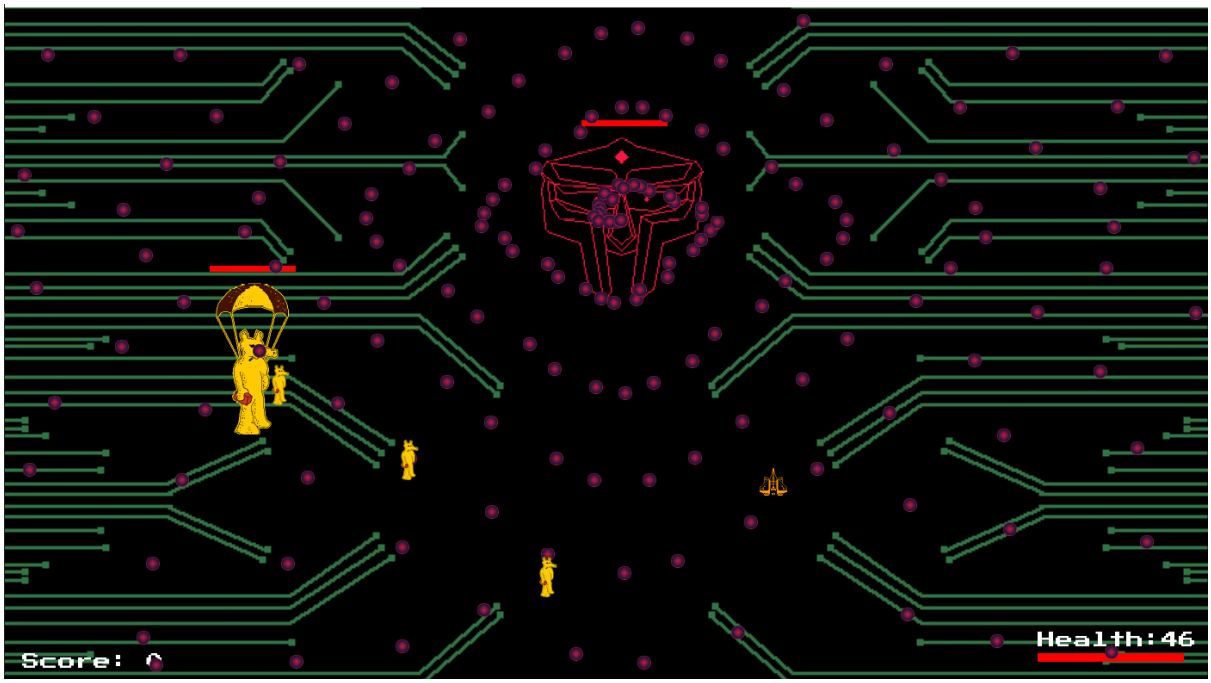
This code is the same above, however is for the player bomb attack. Another difference is the damage the bomb does to enemy B.

```
else
{
    //Collision - enemy and bullet collide - score is added, bullet and enemy are deleted
    if (bombs[k]->getBounds().intersects(enemiesB[i]->getBoundsE()))
    {
        //add enemy score (set in Enemy.hpp) to score
        gui->score += enemiesB[i]->getScoreE() * 1.5;
        //enemy is delted from the game
        delete enemiesB[i];
        enemiesB.erase(enemiesB.begin() + i);
        //bullet is delted from the game
        delete bombs[k];
        bombs.erase(bombs.begin() + k);
        enemy_deleted = true;
    }
```

This code is used for adding score to the score count when using the bomb. Additional points are given when using bombs.

```
for (auto* eB : enemiesB)
{
    if (eB->canAttack())
    {
        eBAttack.push_back(
            new enemyBBullet(
                eBAttackTex["EBATTACK"],
                eB->getPos().x,
                eB->getPos().y + eB->getBoundsE().height / 2,
                player->getPos().x/200,player->getPos().y/200));
    }
}
```

This code is used for the enemy attack. If the canAttack function is true, then the attack is created at the enemy position and these bullets go the player's coordinates. The x and y values are divided by 200 because it was moving towards the player too fast. With this adjustment, users should be able to avoid the attack.



Name of Test		Data	Expectation
Key W		W key	The player's character moves up the screen
Key A		A key	The player's character moves left across the screen
Key S		S key	The player's character moves down the screen
Key D		D key	The player's character moves right across the screen
Key Up		Up key	The player's character shoots its projectile attack in a linear line (may change with powerups, effects have not been decided yet).
Play Button		Click button / 1 key	Setting this variable to true takes player from main menu or end of a level into a new game.
Options Button		Click button / 2 key	Setting this variable to true should take the player into the options.
Leader board Button		Click button / 3 key	Setting this variable to true should take the player to the leader board screen. The high score.txt file should appear in an aesthetically pleasing way.
Sound		MP3 file	Music should be playing during the main menu, when a button is clicked and when collisions happen

Level end		High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again
Collision		Player coordinates = MalwareAttack coordinates and Malware coordinates = PlayerAttack coordinates and Player coordinates = PowerUp coordinates	If player collides with an attack, they lose a health point; when player health equals 0 game ends. If malware collides with the player's attack, they lose a health point; when malware health equal 0 they are destroyed. If player collides with power up a multiple choice appears on screen and game is paused. If question is answered correctly the power up is granted for a short time.
High Score		Time, score	High score is generated based on time survived and malware destroyed.
Colour Blind mode		Options menu data	When enabled game colour changes
Read text from file. Multiple choice question		.txt document with CSV. Malware question	When player collides with a power up, a question at random will be selected from the txt document. Only if the question is answered

			correctly will the power up
Save High Score		.txt document with CSV on scores and names	At the end of a game, the user will get the chance to save their name by typing their name into an input. This will then be saved to a txt document and can be called upon to show a leader board.
Enemies			Multiple enemies are created out of the window before moving into it. Give score when destroyed.
Enemy shoot			Enemies should also fire projectiles for the players to dodge. These will be indestructible, so players will be forced to dodged

Rate Prototype //Score: 1 out of 5						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	0	4	4	4	4	The attack pattern is too difficult
George	0	5	3	4	3	The game could still have more enemies
Callum	0	4	3	4	3	Game does serve its original purpose of being educational
Drew	0	4	3	3	4	Could include more enemies and needs education

Sum	0	16	13	15	14
Average	0	4	3.25	3.75	3.5

Conclusion

This prototype was well received in terms of gameplay, however the project still doesn't resolve the problem. This is will be completed in the next prototype. Another thing I will have to complete is the menu. Due to how I have coded the project so far implementing a menu may be problematic. Therefore, I will have to reconsider restructuring the code so that it is uses a state machine.

Prototype 1

Due to inefficient code, I am going to restart the project coursework using the same development cycle model.

Due to the reduced time frame some alterations to the game shall be made.

- Shooting mechanic will be replaced with heavier focus on dodging many enemies
- Health is removed, after one collision game is over
- Quiz is replaced with facts that can be seen from menu
- Options will have limited options

State Machine

To start I am going to build a state machine. This code will be used to manage different states that I will build later in this project. In concept each part of the game will be a state (such as menu state and game state), which can be placed into a stack. The state at the top of the stack will be the one that runs. States can then be pushed or popped off to change the running state.

```
namespace OCS
```

Throughout the project I will be using the namespace OCS (initials for the game's name). This will keep all the hpp and cpp files linked together.

```
class State
```

The first class I made was the State class.

```
//initialise the state
virtual void Init() = 0;
//handle input of state
virtual void HandleInput() = 0;
//used for updating the screens
virtual void Update(float dt) = 0;
//draw everything to the screen
virtual void Draw(float dt) = 0;
```

I create methods called: Init, HandleInput, Update and draw. To initialise, I set their values to 0. They will be used throughout the project. Update and Draw both take float dt parameters.

```
virtual void Pause() {}
virtual void Resume() {}
```

These pause and Resume methods are used for the state machine and will be elaborated on when used.

```
//unique pointer to state
typedef std::unique_ptr<State> StateRef;
```

In the StateMachine header file there is a unique pointer to the State cpp file.

```
//add state function
//replacing = true will get rid of previous state, setting to false will pause for example
void AddState(StateRef newState, bool isReplacing = true);
```

The AddState function will be used to add the new state to the top of the stack as well as popping the previous top state off the stack. This will make the code more efficient as having multiple states running, when it is not necessary will cause the code to run slower. The parameter takes the argument true, so by default it will pop states. This can be changed to false, however, if I have the time, I will plan to add a quiz in the middle of the game play. Pausing the game can be done by having this value as true, as to not restart the game.

```
//manually remove top state
void RemoveState();
```

This will be used to remove states off the top of the stack by popping. This will not add another state though. Can be used to return the menu after going to the options or the game.

```
//uses the add state and remove state function
void ProcessStateChanges();
```

This function will use the previous functions to actually perform their intended purposes.

```
//return referance to state at top of stack
StateRef& GetActiveState();
```

Will create a reference to the state at the top of stack.

```
//adds state after popping off previous state
void StateMachine::AddState(StateRef newState, bool isReplacing)
{
    this->_isAdding = true;
    this->_isReplacing = isReplacing;

    this->_newState = std::move(newState);
}
```

This method means that a state will be added by replacing the current top state and the new state will become the new top state.

```
//removes top state
//used to get back to previous state
void StateMachine::RemoveState()
{
    this->_isRemoving = true;
}
```

Removes top state

```
void StateMachine::ProcessStateChanges()
{
    //will only run and remove if there are states left
    if (this->_isRemoving && !this->_states.empty())
    {
        this->_states.pop();

        if (!this->_states.empty())
        {
            //resumes the new top state
            this->_states.top()->Resume();
        }
        //will keep removing every state if not set to false
        this->_isRemoving = false;
    }
```

If isRemoving is set to true and there are states left then the top state is popped. This is so that there is always at least one state. If states are not empty then state will continue to update.

```
//if adding set to true
if (this->_isAdding)
{
    //checks to make sure it is not empty
    if (!this->_states.empty())
    {
        //if user want to replace
        if (this->_isReplacing)
        {
            //then state is first popped
            this->_states.pop();
        }
        else
        {
            //if no state is getting replaced
            this->_states.top()->Pause();
        }
    }
```

If is adding is true, then it will check to see if stack is empty. If it is not, it will check if is Replacing is true. If it is then the stack is popped, else top state is used.

```
//push new state on to stack
this->_states.push(std::move(this->_newState));
//initialise state
this->_states.top()->Init();
//prevents multiple states from being added
this->_isAdding = false;
```

After these if statements in the isAdding if statement, these lines are ran. These will add a new state to the top of the stack and initialise it. Then adding is set back to false, so that multiple of the same state is not added to the stack.

```
//gets active state
StateRef& StateMachine::GetActiveState()
{
    return this->_states.top();
}
```

This function will call the current state at the top of the stack.

Asset manager

Now I will add an asset manager. This will be used to handle the majority of textures as to make the code more efficient.

```
//load texture method that takes 2 parameters
void LoadTexture(std::string name, std::string fileName);
//takes name of texture set with previous method and returns a reference
sf::Texture& GetTexture(std::string name);
```

There is first a load texture method that takes two parameters. These will take arguments of an appropriate name given to them as well as the file location. A string will be given to the texture so it can be used with ease. With the file name I will be using definitions. Definitions allow for organisations and quick changes to values and textures.

There is also a get Texture function. This will be used to return the texture.

```
//load font method that takes 2 parameters
void LoadFont(std::string name, std::string fileName);
//takes name of font set with previous method and returns a reference
sf::Font& GetFont(std::string name);
```

The asset manager will also do same, but for fonts. In this prototype there is no font used, but I have added this in now as there will be font required later for strings.

```
private:
    //maps
    std::map<std::string, sf::Texture> _textures;
    std::map<std::string, sf::Font> _fonts;
```

In the private section of asset manger there are maps for both texture and fonts. This will take the given name as a string and can be used throughout the code

```
void AssetManager::LoadTexture(std::string name, std::string fileName)
{
    //local texture variable
    sf::Texture tex;

    //if the texture loads from file
    if (tex.loadFromFile(fileName))
    {
        //it is added to texture map
        this->_textures[name] = tex;
    }
}
```

In this function there is a local texture variable giving texture the string tex. If the texture is loaded from a text file then the texture is added to the map in the header file.

```
sf::Texture& AssetManager::GetTexture(std::string name)
{
    //returns textures based on name previously given to texture
    return this->_textures.at(name);
}
```

Returns a texture when specified by using the applied name.

This section is identical to the font, except from changing texture to font.

Input Manager

This section is used for the mouse. These files will check for the mouse and will allow buttons to be pressed.

```
//boolean statement
//checks if a sprite is left clicked with mouse
bool IsSpriteClicked(sf::Sprite object, sf::Mouse::Button button, sf::RenderWindow& window);
```

In this Boolean function, there are parameters for sprite, left mouse click and render window reference.

```
//get mouse position function
sf::Vector2i GetMousePosition(sf::RenderWindow& window);
```

The other function will be for the mouse. This will be used to get the position of the mouse relative to the window of the game.

```
if (sf::Mouse::isButtonPressed(button))
{
    //creates rectangle covering the given object sprite
    sf::IntRect tempRect(object.getPosition().x, object.getPosition().y, object.getGlobalBounds().width, object.getGlobalBounds().height);
```

If the mouse button is pressed the following code is ran. The first line will create a rectangle covering the position of the sprite. This is done by getting the top left coordinate of the sprite and then getting the height and width of the sprite.

```
//if mouse is inside the rectangle boolean statement returns true
if (tempRect.contains(sf::Mouse::getPosition(window)))
{
    return true;
}
```

When the mouse is then inside of the temporary rectangle and has been clicked the Boolean will be returned as true.

```
//else returns false
return false;
```

Else the Boolean value will be returned as false.

```
sf::Vector2i InputManager::GetMousePosition(sf::RenderWindow& window)
{
    //return the poisiton of mouse relative to window
    return sf::Mouse::getPosition(window);
}
```

This function is responsible for checking the position of the mouse in relation of the game window. This is important as otherwise the code will not know if the mouse if over the relevant sprite.

Game Data

```
struct GameData
{
    StateMachine machine;
    sf::RenderWindow window;
    AssetManager assets;
    InputManager input;
};
//used to access the classes in the GameData structure
typedef std::shared_ptr<GameData> GameDataRef;
```

There is a shared pointer, which will allow access to all instances in the structure. This will be used throughout the code to gain access to all parts of game engine.

```
class Game
{
public:
    //constructor
    //parameters of screen dimensions and title on window
    //used in main.cpp
    Game(int width, int height, std::string title);
```

In this header file there is also the Game class (this will not be where the game code will be). The constructor will take parameters of the screen dimensions and title.

```
//frame rate
//60 fps
const float dt = 1.0f / 60.f;
sf::Clock _clock;

//_data used to access all the game data above
GameDataRef _data = std::make_shared<GameData>();

//called when game start
void Run();
```

In the private section the frame rate is set to be 60fps. SFML clock is also set up to be used in the cpp file. _data is used to be the shared pointer for all parts in the GameData structure. There is also run function.

```
Game::Game(int width, int height, std::string title)
{

    srand(time(NULL));

    //create window
    // dimensions and style
    _data->window.create(sf::VideoMode(width, height), title, sf::Style::Close | sf::Style::Titlebar);
    //chooses starting state
    _data->machine.AddState(StateRef(new SplashState(this->_data)));

    this->Run();
}
```

In the Game constructor the style of the window is made to be closed and so that there is a title bar. The first state is also determined here: the Splash state (explained below). The Run function is also called.

```
void Game::Run()
{
    float newTime, frameTime, interpolation;

    //get time in seconds from when the clock first starts
    float currentTime = this->_clock.getElapsedTime().asSeconds();
    //
    float accumulator = 0.0f;
```

The value for current time is set to be total seconds that have happened since the program has started to run. The value for the accumulator.

```
//while window is open
while (this->_data->window.isOpen())
```

The rest of the code in the Game.cpp file is in a while loop. This while loop will run as long as the window is open; meaning it will loop, until program is closed.

```
//will check to see if states have been changed at all
this->_data->machine.ProcessStateChanges();
```

Will use the ProcessStateChanges function from the state machine file. So as long as the window is open, the program will keep checking to see if any state has changed.

```
this->_data->machine.GetActiveState()->HandleInput();
this->_data->machine.GetActiveState()->Update(dt);
```

While the window is open, inputs are handled and the screen will constantly be updated.

This concludes the game engine part of the prototype.

Definitions File

To keep everything organised I will have a definitions folder. This will hold textures and values which can then be called when needed.

```
#pragma once

//screen dimensions
#define SCREEN_WIDTH 1400
#define SCREEN_HEIGHT 788

//splash state textures
#define SPLASH_STATE_BACKGROUND_FILEPATH "Texture/splashStateLogo.png"
```

This is what it looks like currently, but it will grow as more code is written

Splash State

Before the main menu loads there will be a state screen. Most games will have a logo or name of the game appear before loading the actual game. This will also be done in my game.

```
class SplashState : public State
```

Before I can start coding, I will have to inherit State.hpp because this is a state. All states that are coded will be using inheritance in creating states. This will allow states to access the functions created in the state file. All states will use these functions. If a function is not called into one of these stat functions they will not run.

```
//constructor
SplashState(GameDataRef data);

//functions
void Init();
void HandleInput();
void Update(float dt);
void Draw(float dt);
```

The constructor is first set, taking parameter of data, with reference to GameData.

There are then the functions. These functions are the same as the ones in State.hpp. These functions will be used in every state file.

```
private:
    //
    GameDataRef _data;

    sf::Clock _clock;

    sf::Sprite _background;
};
```

Private section takes reference to game data, sets up the use of a clock and also background sprite.

```
void SplashState::Init()
{
    _data->assets.LoadTexture("Splash State Background", SPLASH_STATE_BACKGROUND_FILEPATH);

    _background.setTexture(this->_data->assets.GetTexture("Splash State Background"));

    _background.setPosition((SCREEN_WIDTH / 2) - (_background.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 2) - (_background.getGlobalBounds().height / 2));
}
```

In the init function I first load the texture Splash State Background by using the definitions file. The file location is set as the definition.

The background sprite is then set as the texture.

The textured sprite is set to be in the middle of the screen. Despite the name, I am using a small image instead of a background, so I had to centre it.

```
void SplashState::HandleInput()
{
    sf::Event event;

    while (_data->window.pollEvent(event))
    {
        if (sf::Event::Closed == event.type)
        {
            _data->window.close();
        }
    }
}
```

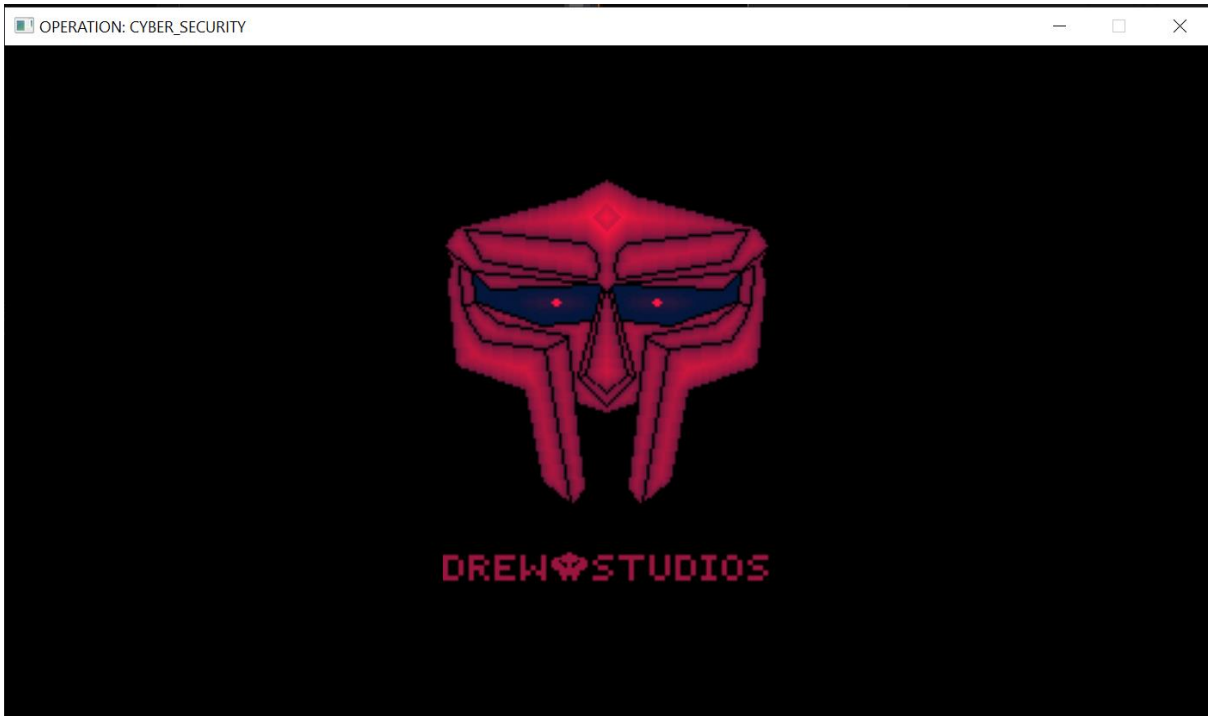
In this function an event is made. Events are important as they are used to make sure the program can be closed easily. Without this the end user may find the program to be problematic, as they would not be able to close it.

```
void SplashState::Update(float dt)
{
    if (_clock.getElapsedTime().asSeconds() > SPLASH_STATE_SHOW_TIME)
    {
        _data->machine.AddState(StateRef(new MainMenuState(_data)), true);
    }
}
```

If the clock that runs as soon as the program starts running is more than the set time in the definitions folder, then a new state will load by adding it to the stack. This means that the splash state is also popped, indicated by the true Boolean. After the splash state, the menu will load.

```
void SplashState::Draw(float dt)
{
    _data->window.clear();
    _data->window.draw(_background);
    _data->window.display();
}
```

The window is then made to be cleared, drawn and displayed every second.



Main Menu State

```
class MainMenuState : public State
```

Main Menu State will first inherit from the State class.

```
MainMenuState(GameDataRef data);

void Init();
void HandleInput();
void Update(float dt);
void Draw(float dt);
```

Just like the splash state, the main menu will have the same functions used.

```
private:
    GameDataRef _data;

    sf::Sprite _background;
    sf::Sprite _title;
    sf::Sprite _playButton;
```

In the private section there is the game data reference. There are also the sprites used to construct the menu. Currently, there is no options button as there may not be enough time to complete the menu. This may be included in a later prototype.

```
void MainMenuState::Init()
{
    _data->assets.LoadTexture("Main Menu Background", MAIN_MENU_BG_FILEPATH);
    _data->assets.LoadTexture("Game Title", GAME_TITLE_FILEPATH);
    _data->assets.LoadTexture("Play Button", PLAY_BUTTON_FILEPATH);
```

The main menu state first loads all the relevant textures. Just like the previous textures, it makes use of the definitions folder.

```
_background.setTexture(this->_data->assets.GetTexture("Main Menu Background"));
_title.setTexture(this->_data->assets.GetTexture("Game Title"));
_playButton.setTexture(this->_data->assets.GetTexture("Play Button"));
```

The sprites from the header file have textures applied to them.

```
_title.setPosition((SCREEN_WIDTH / 2) - (_title.getGlobalBounds().width / 2), _title.getGlobalBounds().height / 10);
_playButton.setPosition((SCREEN_WIDTH / 2) - (_playButton.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 2) - (_playButton.getGlobalBounds().height / 2));
```

The sprites then have their positions set on the main menu. The title is centred in the screen by having its position set to be halfway across the x axis minus the sprite's width. The y position is then set to be near the top of the screen. The play button was the same, except the y position was set so it was low down the screen.

```
void MainMenuState::HandleInput()
{
    sf::Event event;

    while (_data->window.pollEvent(event))
    {
        if (sf::Event::Closed == event.type)
        {
            _data->window.close();
        }
```

The main menu has the necessary event while loop, allowing for the program to be easily closed.

```
    if (_data->input.IsSpriteClicked(_playButton, sf::Mouse::Left, _data->window))
    {
        _data->machine.AddState(StateRef(new GameState(_data)), true);
    }
```

Also, in the while loop there is the sprite click function. If the left mouse button is clicked over the play button sprite, then a new state is added – the game state. This will also pop the main menu, as there is no reason for it to run while the game is played.

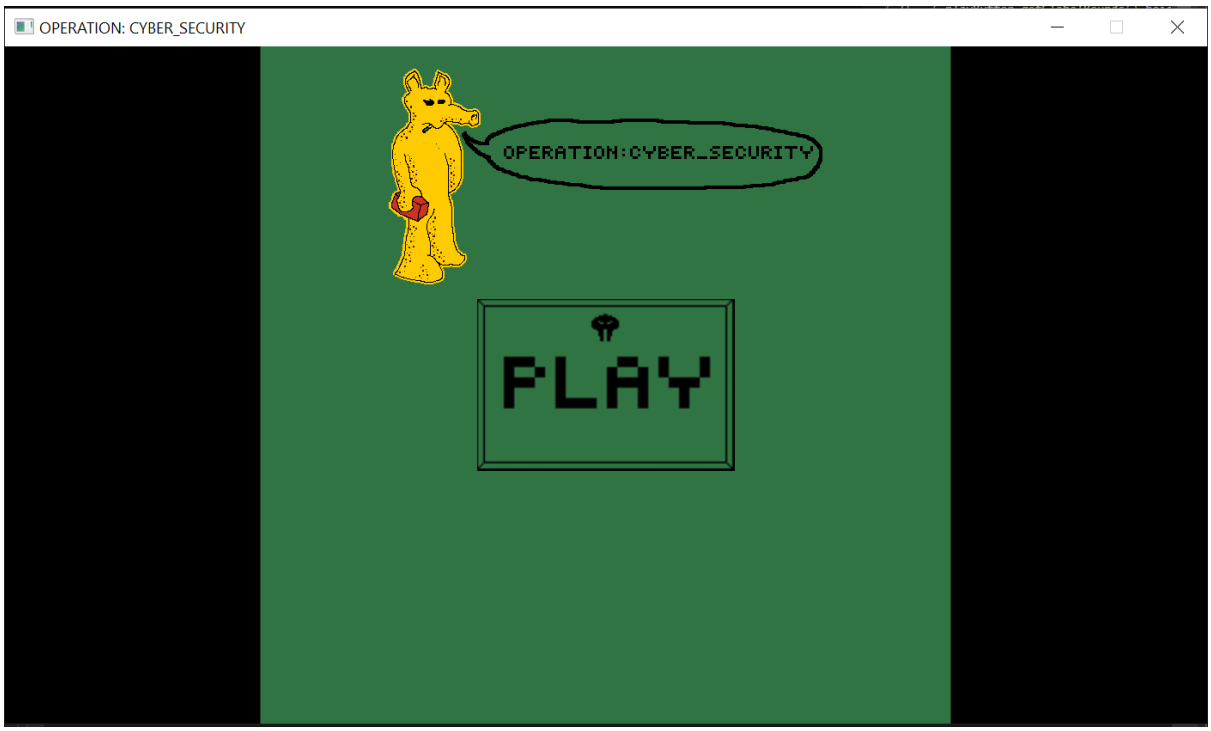
Validation is used here to make sure that the user wants to enter the game state. There is validation here to check if the mouse cursor is in the same position to the sprite in relation to the window. There is another validation check checking for mouse clicks on the sprite.

```
void MainMenuState::Update(float dt)
{

}
```

There is currently no code needed for this function, however, I have it ready if it is needed in the future.

```
void MainMenuState::Draw(float dt)
{
    _data->window.clear();

    _data->window.draw(_background);
    _data->window.draw(_title);
    _data->window.draw(_playButton);

    _data->window.display();
}
```

The final part for the menu so far is the draw function. This has the clear and display functions allowing for the screen to have an updated picture. There are also the draw functions so that the sprites are drawn to the screen.



After the Splash screen is played, the menu screen will be pushed on to the stack, displaying this screen.

Moving Background

The moving background will be the transition between the menu and game.

```
MovingBackground(GameDataRef data);

void MoveBackground(float dt);
void DrawBackground();
```

The background will include a move function and a draw function.

```
private:
    GameDataRef _data;
```

In the private section, there is the usual shared pointer the game data structure.

```
std::vector<sf::Sprite>_movingBackgroundSprites;
```

Use of a vector to make a sprite with the string _movingBackgroundSprites.

```
float fps = 1000.f / 60.f;
```

Sets the frame rate to 60fps, to make the movement of the background smoother.

```
 //set texture to sprite
 _data->assets.LoadTexture("Moving Background", GAME_STATE_MOVING_BACKGROUND_FILEPATH);
 sf::Sprite sprite(_data->assets.GetTexture("Moving Background"));
```

The sprite is set so that it has a texture. This is done like all the others, as it works via definitions.

```
//set position
sprite.setPosition(0, 0);
```

The positions of the backgrounds are set.

```
//moves sprites
_movingBackgroundSprites.push_back(sprite);
```

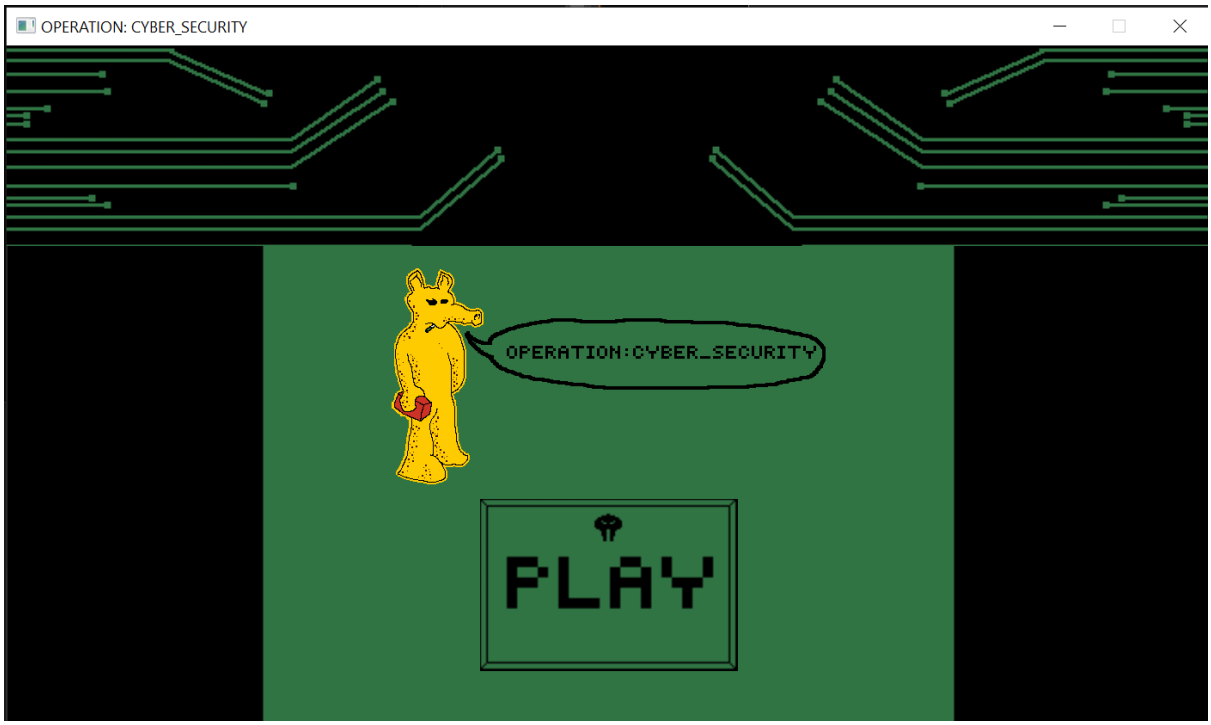
The Push back functions are responsible for moving the sprites.

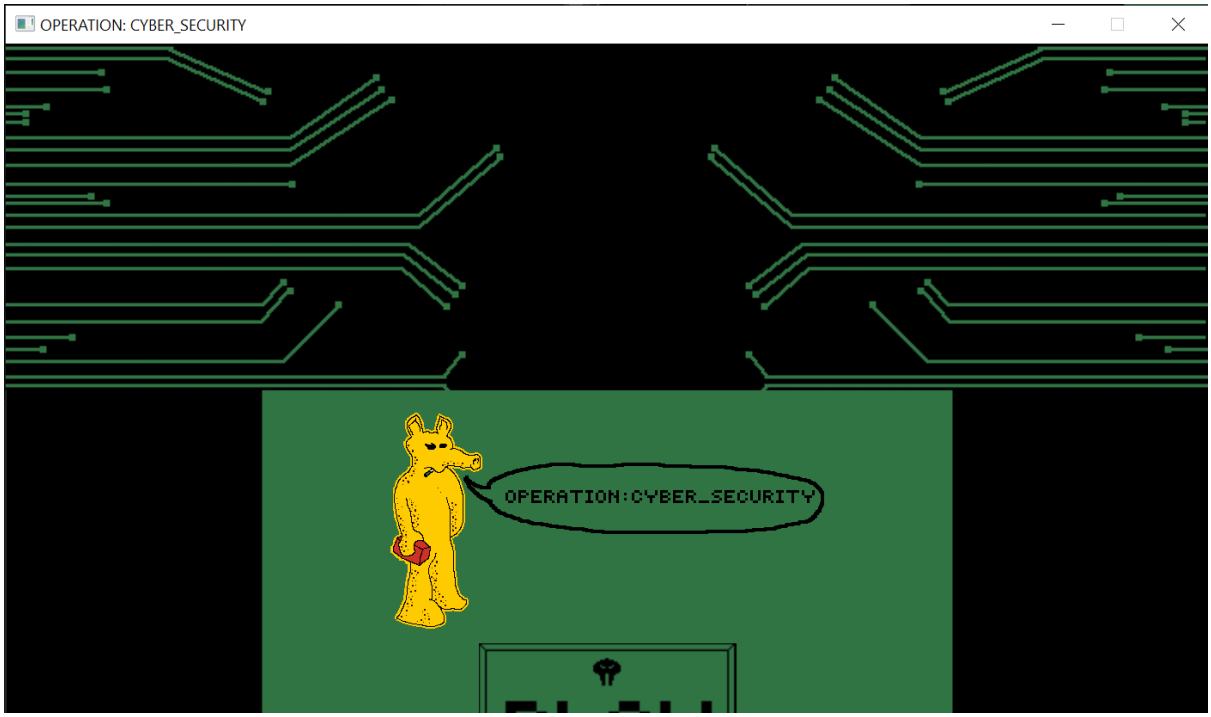
```
void MovingBackground::MoveBackground(float dt)
{
    for (unsigned short int i = 0; i < _movingBackgroundSprites.size(); i++)
    {
        // movement = speed multiplied by delta time
        float movement = BACKGROUND_MOVEMENT_SPEED * dt;

        //direction
        _movingBackgroundSprites.at(i).move(0.0f, movement);
    }
```

In this function, the movement value is set by having the value set in definitions multiplied by dt. This way the speed will be the same on all devices, which is important as this should be playable at schools.

```
//draw
_data->window.draw(_movingBackgroundSprites.at(i));
```

This is the draw function to draw the sprites to the window.





This code works by having the moving background texture being set to a screen shot of the game menu. This means when the play button is pressed the menu state will be popped. The game state will start but the moving background will be in this state and have its y value increase. This gives the illusion of having the main menu screen falling as seamless transition. Despite not adding much value, I think the transition give the game a more professional feel

Game State

The game state will be state where the game is player.

```
class GameState : public State
{
public:
    GameState(GameDataRef data);

    void Init();
    void HandleInput();
    void Update(float dt);
    void Draw(float dt);
```

Similar to other states, Game state inherits from state and have the same functions to use.

```
private:
    GameDataRef _data;

    sf::Sprite _background;

    MovingBackground* movebg;
    Player* player;

    sf::Clock clock;

    int _gameState;

    int _score;
```

In the private section there is the game data reference, background sprite, moving background sprite dereference, player dereference, clock, _gameState variable, and score variable.

```
void GameState::Init()
{
    _data->assets.LoadTexture("Game State Background", GAME_STATE_BACKGROUND_FILEPATH);;
    _data->assets.LoadTexture("Moving Background", GAME_STATE_MOVING_BACKGROUND_FILEPATH);
    _data->assets.LoadTexture("Player Idle", PLAYER_FRAME_IDLE_FILEPATH);
    _data->assets.LoadTexture("Player Move", PLAYER_FRAME_MOVE_FILEPATH);
    _data->assets.LoadTexture("Player Attack", PLAYER_ATTACK_FILEPATH);
```

Textures are loaded into this game state file through using definitions.

```
movebg = new MovingBackground(_data);
```

New function is used for moving background so that it can be used in this file.

```
_background.setTexture(this->_data->assets.GetTexture("Game State Background"));
```

The background sprite is set to the texture tied the respective string.

```
//game states
enum GameStates
{
    eReady,
    ePlaying,
    eGameOver
};
```

In the definition file I have enum GameStates. This will be used to determine when the game is over, is playing ect.

```
_gameState = GameStates::eReady;
```

When the game state is first loaded eReady will be first state.

```
void GameState::HandleInput()
{
    sf::Event event;

    while (_data->window.pollEvent(event))
    {
        if (sf::Event::Closed == event.type)
        {
            _data->window.close();
        }
    }
}
```

Necessary event while loop to close the window

```
void GameState::Update(float dt)
{
    if (GameStates::eGameOver != _gameState)
    {
        player->Animate(dt);

        movebg->MoveBackground(dt);

        _gameState = GameStates::ePlaying;

        player->Input();
    }
```

If the game is not over the player will animate (when I have added it) and the Move background will move off the screen. The game state will also become ePlaying form eReady. This will also hold player input; this will likely hold the player movement.

```
void GameState::Draw(float dt)
{
    _data->window.clear();

    _data->window.draw(_background);
    player->Draw();
    movebg->DrawBackground();



    _data->window.display();
}
```

In the Draw function the screen is cleared and displayed. The background, player and moving background are drawn here. It is important it is in this order as they are drawn in order. Therefore if the background was coded below the others, then it would appear as the only sprite drawn was the background.

Player

```
public:
    Player(GameDataRef data);

    void Animate(float dt);

    void Input();

    void Update(const float dirX, const float dirY);

    const sf::Sprite& GetSprite() const;

    void Draw();
```

The player has an animation function, input function, sprite getter and draw function.

```
GameDataRef _data;

sf::Sprite _playerSprite;
std::vector<sf::Texture> _animationFrames;

unsigned int _animationIterator;

sf::Clock _clock;

sf::Clock _movementClock;
```

A string is set to player sprite.

Textures are set to the animation of the player.

Unsigned integer variable called animation iterator.

Clock and movement clock are both used in this file.

```
Player::Player(GameDataRef data) : _data(data)
{
    _animationFrames.push_back(_data->assets.GetTexture("Player Idle"));
    _animationFrames.push_back(_data->assets.GetTexture("Player Move"));
```

In the constructor the textures are set to the _animationFrames texture. Both will be applied to the same sprite.

```
_playerSprite.setTexture(_animationFrames.at(_animationIterator));
```

The sprite has the textures set to it at animationIterator.

```
_playerSprite.setPosition((_data->window.getSize().x / 2) - (_playerSprite.getGlobalBounds().width / 2),
    (_data->window.getSize().y / 1.3) - (_playerSprite.getGlobalBounds().height / 2));
```

This sets the player position at the start of the game to the bottom middle of the screen. This is a typical place to start in this genre of games.

```
void Player::Animate(float dt)

    if (_clock.getElapsedTime().asSeconds() > PLAYER_ANIMATION / _animationFrames.size())
    {
```

If the elapsed time is larger than the time set in player animation divided by amount of animations there are (2).

```
if (_animationIterator < _animationFrames.size() - 1)
{
    _animationIterator++;
}
```

Animation iterator will increase until it is the same size as the amount of animation frames.

```
else
{
    _animationIterator = 0;
}
```

When this happens animation iterator value is set back to 0.

```
_playerSprite.setTexture(_animationFrames.at(_animationIterator));
_clock.restart();
```

Texture will change approximately every couple seconds, the clock will restart and the animation will repeat.

```
void Player::Input()
{
    //sets x and y values
    float x = 0.0f;
    float y = 0.0f;
```

The value of x and y in this function is set to 0.

```
//Pythagoras - Make diagonal movement same speed as horizontal and vertical movement
x = std::sqrt(std::pow((double)x, 2) + std::pow((double)y, 2));
y = std::sqrt(std::pow((double)x, 2) + std::pow((double)y, 2));
```

Pythagoras theorem is used in this movement function. This is so that the player does not move faster diagonally than linearly. This makes the game fairer and more satisfying to move the sprite.

```
//w decreases y value
if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))
{
    y -= PLAYER_MOVEMENT_SPEED;
}
//s increases y value
else if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
{
    y += PLAYER_MOVEMENT_SPEED;
}
//a decreases x value
if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))
{
    x -= PLAYER_MOVEMENT_SPEED;
}
//d increases x value
else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
{
    x += PLAYER_MOVEMENT_SPEED;
}
```

Pressing the respective keys will cause the y or x to increase or decrease. This will cause the player to move.

Validation is used to make sure that the key is pressed. It is important for there to be validation as it will make sure the ley is pressed to make the player sprite move on the window. The game will need to make sure that when the player wants to move up the screen, the player sprite moves up the screen. This will make sure the game is not frustrating to play as delayed movement, or movement that doesn't work can lead to a loss.

```
Update(x, y);
```

The update function takes the x and y value as arguments here. This is necessary for the next function.

```
void Player::Update(const float dirX, const float dirY)
{
    _playerSprite.move(PLAYER_MOVEMENT_SPEED * dirX, PLAYER_MOVEMENT_SPEED * dirY);
}
```

This function is responsible for player movement as x and y values based on user input. These values are multiplied by a speed value, so that the player doesn't move at a slow and frustrating speed.

```
void Player::Draw()
{
    _data->window.draw(_playerSprite);
}
```

The Draw function will draw the player sprite to the screen.

The animation is only small; however the back of the ship will have fire coming the every few seconds. This makes the gameplay feel more dynamic.

Name of Test	Data	Expectation	Reality
Key W	W key	The player's character moves up the screen Y value decreases	Pass – player sprite moves up the screen.
Key A	A key	The player's character moves left across the screen X value decreases	Pass – player sprite moves across the screen.

Key S	S key	The player's character moves down the screen Y value increases	Pass – player sprite moves down the screen
Key D	D key	The player's character moves right across the screen X value increases	Pass – player sprite moves across the screen
Splash State	State	When code is first loaded, the splash state should be added to stack. This will then display this state to the window. After short delay this state will be switched out to main menu state.	Pass – when code is ran, splash state is displayed. This shows image for a couple seconds before changing state
Main Menu State	State, button	This state will should load after splash state. This state will need to hold buttons that will be used to navigate to all other states	Pass – this state loads after splash state and holds buttons that can be pressed to change
Options state	State, button	This state should have a back button to return to menu and a form of option such as change in key binds or colour blind mode	Fail - unattempted
Education state	State, button	This state should offer an education and way to return to main menu state	Fail - unattempted
Game State	State	This should include the actual game play. Enemies and data classes should be used here to create cohesive game.	Pass – Game state will be loaded after pressing the play button. Main menu is popped and

		The game state should be popped and game over state should be pushed on player collision with enemy.	game state is pushed.
Game over state	State, button, string, txt file	This should load after the game is over. This will give user option to return to the main menu, but also have their score outputted as a string and the high score to be outputted to the screen	Fail - unattempted
Play Button	Click button	Clicking on this sprite will pop main menu state and push game state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Options Button	Click button	Clicking on this sprite will pop main menu state and push option state	Fail - unattempted
Education button	Click button	Clicking on this sprite will pop main menu state and push education state	Fail - unattempted
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	Fail - unattempted
Collision	Player position = Malware position and	If player position intersects with enemy position then the screen needs to flash.	Fail - unattempted

		The game over state needs to pushed to top and previous state should be popped	
Flash	rect	A translucent rectangle will be drawn to the screen on collision	Fail - unattempted
Score	Time, score	High score is generated based on time survived.	Fail - unattempted window. On collision the sprite is erased and score increments
Options	Options button	Additional option such as option to change key bind or add colour blind mode. Should be found in options	Fail - unattempted
Education	Education sprite	A sprite of the education needs to be printed to the screen. This will sprite will have a texture of text with useful facts.	Fail - unattempted
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will have their score saved to text file. This text file will have to be opened to retrieve the high score	Fail - unattempted
Enemies		Multiple enemies are created out of the window before moving into it. These enemies will have unique movement and collisions with player.	Fail - unattempted

		Enemy should be deleted when out of the screen fully.	
Player and enemy shooing	sprite	Enemies shoot and player shoot. This is done with shoot class that creates an object	This has not be done, as the shoot class did not work due to parameter problems with the state machine

My first prototype has succeeded in very few parts of my checklist. However, due to the state machine the process of completing the other parts should be dramatically easier. To summarise I have a working menu and a player that moves and is animated.

Rate Prototype //Score: 0 out of 10						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	0	3	2	7	1	Bounds for player
George	0	4	3	5	2	Enemies
Callum	0	3	2	8	2	Enemies
Drew	0	2	0	7	1	Enemies
Sum	0	12	7	27	6	
Average	0	3	1.75	6.75	1.5	

From discussions with stakeholders, we have determined in the next prototype I need to include the game actual gameplay. This will leave me the final prototype for education and options. This will include enemies and the collisions needed.

Prototype 2

Collisions

To start the second prototype, I first included bounds for the player. This is so they cannot make the player sprite get out of window. This is important because attacks will be deleted when they are out of bounds, so by having the player outside the bounds they will be unable to die.

```
void Player::Bounds()
```

I first created a new function in the player class called Bounds. Here I will construct the relevant code for the sprite so it cannot leave the window.

```
//left collision
if (_playerSprite.getGlobalBounds().left < 0.f)
    _playerSprite.setPosition(0.f, _playerSprite.getGlobalBounds().top);
```

The left collision would check to see where the sprite's left bound. If this was less than 0 then player sprite would have its x position set to 0 position and the y position will be set to the same as it was. This is so that the sprite would not jump around when its x position is less than 0.

```
//right collision
else if (_playerSprite.getGlobalBounds().left + _playerSprite.getGlobalBounds().width >= _data->window.getSize().x)
    _playerSprite.setPosition(_data->window.getSize().x - _playerSprite.getGlobalBounds().width, _playerSprite.getGlobalBounds().top);
```

The left bound of the sprite is added to the sprite's width. This will give the right bound of the sprite. This bound is compared with the windows furthest x position. If the bound is larger or equal to the window bound, then player sprite will have its x position adjusted.

```
//top collision
if (_playerSprite.getGlobalBounds().top < 0.f)
    _playerSprite.setPosition(_playerSprite.getGlobalBounds().left, 0.f);
```

The top collision works by having the top bound of the sprite compared to the 0 y position of the window. When the sprite is smaller than 0 then the x position is not adjusted, but the y position is changed to be 0.

```
//bottom collision
else if (_playerSprite.getGlobalBounds().top + _playerSprite.getGlobalBounds().height > _data->window.getSize().y)
   _playerSprite.setPosition(_playerSprite.getGlobalBounds().left, _data->window.getSize().y - _playerSprite.getGlobalBounds().height);
```

The final collision bound is with the bottom of the window. If the player sprite top bound + its height (giving bottom bound) is larger than the window y size, the y position of the ship is changed to be the y position max value.

The use of the window bounds is use of validation because the player sprite rectangle is compared to the position of the window bounds. Using this validation the player is unable to leave the window, preventing them from hiding off the screen and preventing them from colliding with enemies.

```
//player movement, bounds, and animation
player->Input();
player->Bounds();
player->Animate(dt);
```

Alongside input and animate functions, I have placed the function class. This will make sure the bounds for the player character will constantly be running.

Collisions

Next, I will set collisions that will be used in the game state. This will be used for score reasons and for player collisions with the enemies.

```
//boolean functions
bool CheckSpriteCollision(sf::Sprite sprite1, sf::Sprite sprite2);
bool CheckSpriteCollision(sf::Sprite sprite1, float scale1, sf::Sprite sprite2, float scale2);
```

In the collision class I set up the functions in the hpp class. These functions have the name, but different parameters, meaning this compile time polymorphism done through function overloading (https://www.geeksforgeeks.org/polymorphism-in-c/). By function overloading the code has improved readability, saves memory space and speeds up execution speed.

Validation is used when a Boolean is used. This is because the program will have to check if the value of the function ever becomes true. When it becomes true then I am planning to make the game end. By having the value not change to true, then the game would not be fun as the game will not end.

```
bool Collision::CheckSpriteCollision(sf::Sprite sprite1, sf::Sprite sprite2)
{
    //returns rectangles around the sprites
    sf::Rect<float> rect1 = sprite1.getGlobalBounds();
    sf::Rect<float> rect2 = sprite2.getGlobalBounds();
```

In the first function I created two rectangles around the sprites. I then set the size of these rectangles to the bounds of the sprite I set in the game state.

```
//if sprite 1 intersects sprite 2 true is returned
if (rect1.intersects(rect2))
{
    return true;
}
else
{
    return false;
}
```

I then check to see if the rectangles intersect with each other. If they do then the Boolean will return a true statement, else it will return false. This function will be used for the scoring system, as when the player hits the invisible sprite, it will return true and score will increase.

```
bool Collision::CheckSpriteCollision(sf::Sprite sprite1, float scale1, sf::Sprite sprite2, float scale2)
```

The second function is slightly different as it takes different parameters. The parameters take float scale.

```
//set scale of rectangles around sprite
sprite1.setScale(scale1, scale1);
sprite2.setScale(scale2, scale2);
```

Here the scale will be set. This will change the size of the rectangle created over the respective sprite. This is significant as the enemy sprites will not a perfect square sprite.

Therefore, by doing these collisions will feel more accurate, giving a more enjoyable gaming experience. The rest of this function is the same as the previous function.

GUI / Score

```
//calls font from DEFINITIONS
_data->assets.LoadFont("Font", FONT_FILEPATH);
```

In the Gui constructor I first call the font from the DEFINTIONS. This will allow be set the string for the score using this font.

```
/text style
scoreText.setFont(_data->assets.GetFont("Font"));
```

Here I set font for the score text Text.

```
_scoreText.setString("0");
```

Next, I inputted the initial output for the string. This means when the game is loaded 0 will have already been outputted to the screen.

```
_scoreText.setCharacterSize(65);
```

This sets the size of the font used to font size 65

```
_scoreText.setFillColor(sf::Color::White);
```

This sets the colour of the string to white.

```
_scoreText.setOrigin(_scoreText.getGlobalBounds().width / 2, _scoreText.getGlobalBounds().height / 2);
```

This sets the origin of the text to be in the middle. Usually, the origin is in the top left, which would mean as the score increase in digit size it will no longer be cantered and be closer to the right.

```
_scoreText.setPosition(_data->window.getSize().x / 2, _data->window.getSize().y / 5);
```

This sets the font to appear in the top middle of the screen.

I have opted to set the score in big size in the top middle so the player can easily see the score. The player will likely be closer to the bottom middle of the screen so their view will not be obstructed.

```
//draw function
_data->window.draw(_scoreText);
```

Draw function draws the score to the game screen

```
//update function
_scoreText.setString(std::to_string(score));
```

In the update function the string is set to be updated every time score increases. The to_string function is used to change the integer value of the score to a string so that it is compatible with the above code.

Flash

To add more impact to the collisions I shall create a flash effect on the screen when there is a collision.

```
sf::RectangleShape _shape;

bool _flashOn;
```

In the header file I made a rectangle shape using SFML and a Boolean value for the flash.

```
//flash sprite and size
_shape = sf::RectangleShape(sf::Vector2f(_data->window.getSize()));
_shape.setFillColor(sf::Color(255, 255, 255, 0));
```

In the constructor I have the colour of the rectangle set their maximum values. This does not matter though as the shape is transparent due to the 0. This is still necessary to put in the constructor though so that there is a flash.

I then have the shape set fill colour function. This is necessary, otherwise the game state screen will be white and player won't be able to see.

```
//flash set to true
_flashOn = true;
```

Flash on is set to true in the constructor.

```
if (_flashOn)
{
```

In the flash show function, there is an if statement that says _flashOn. This is set true in the constructor, therefore when this code is called the if statement will run.

```
//flash red
int _flash = (int)_shape.getFillColor().a + (FLASH_SPEED * dt);
```

_flash value will hold the colour of shape set below and will have an int value that increases by FLASH_SPEED multiplied dt. Dt is always increasing but will be reset to 0 on collision. This means after collision _flash value will increase from 0.

```
//set flash to false
if (_flash >= 96.0f)
{
    _flash = 96.0f;
    _flashOn = false;
}
```

If the _flash is more than 96 the flash value is set back to 96 and flash is set to false.

```
//red flash
_shape.setFillColor(sf::Color(255, 0, 0, _flash));
```

The colour the shape in this if statement makes the flash red and only partially transparent.

```
//flash white
int _flash = (int)_shape.getFillColor().a - (FLASH_SPEED * dt);
```

The else statement starts the same as the if statement.

If the flash value is less or equal to 0 then the following code will run:

```
//set flash to false
if (_flash <= 0.0f)
{
    _flash = 0.0f;
    _flashOn = false;
}
```

This makes _flash value set back to 0 then equal false.

```
//white flash
_shape.setFillColor(sf::Color(255, 255, 255, _flash));
```

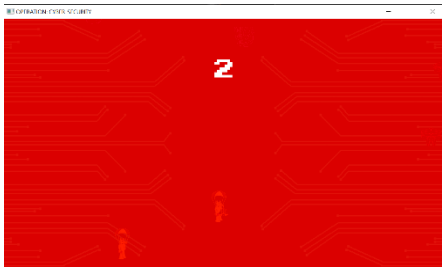
From the colour value set the shape will be white.

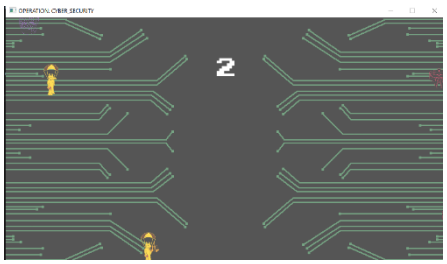
```
_data->window.draw(_shape);
```

The _shape is then drawn to the screen.

This works as when the player collides with an enemy, game state is equal to over. When game state is equal to over, flash show function is called. This will cause the screen to go red until _flash is equal to 96 then the flash is set to false. The else statement works in a similar way but has a white flash this results in the screen flashing red then white after a collision.

Originally the flashing effect would have lasted longer, however, to make the game more accessible to people that suffer from epilepsy, I have the flash very short.





(Images of flash effect after inclusion of enemies)

Enemies

```
//reference to sprites
const std::vector<sf::Sprite>& GetSpriteA() const;
const std::vector<sf::Sprite>& GetSpriteB() const;
const std::vector<sf::Sprite>& GetSpriteC() const;
const std::vector<sf::Sprite>& GetSpriteD() const;
```

In the header file for enemies I have created references for the enemy sprites, which will be called in the game state file. Const has been used as these sprites will remain constant and will not be modified.

```
//reference to score enemy
std::vector<sf::Sprite>& GetScoringSprites();
```

There is also a reference to the scoring sprite. My intention with this is to have an invisible sprite that acts like an enemy, except on collision score increases.

```
std::vector<sf::Sprite> enemySpritesX1;
std::vector<sf::Sprite> enemySpritesX2;
std::vector<sf::Sprite> enemySpritesY1;
std::vector<sf::Sprite> enemySpritesY2;
//sprite for score enemy
std::vector<sf::Sprite> scoringEnemy;
```

Each of these enemies are given their own sprite.

```
//float values, movement
float eMove = eMoveMax;
float eMoveMax = 100;
```

These float variables will be used for the enemy movement.

```
//texture loaded
_data->assets.LoadTexture("Enemy A", ENEMY_A_FILEPATH);
//texture set to sprite
sf::Sprite sprite(_data->assets.GetTexture("Enemy A"));
//set position
sprite.setPosition(_data->window.getSize().x, _data->window.getSize().y - sprite.getGlobalBounds().height - _enemySpawnYOffset);
//enemy push back
enemySpritesA.push_back(sprite);
```

In the this function the texture called from the DEFIINTIONS file, and the texture is then applied to sprite.

The position of enemy A is then set. In this example of code, the enemy will spawn on the right side of the screen and has a random y position.

The sprite then has the push back function used. This works as vectors are dynamic arrays, and the function is used to push sprites into the vector. This means that a new sprite will spawn and have the same attributes the previous ones had, such as the texture and movement.

The other sprites have similar functions except different textures and positions set.

```
//eMove will increase until larger than eMoveMax, before having its value decreased
eMove += 0.5f;
if (eMove >= eMoveMax)
{
    eMove = -0.2f;
}
```

In the move function, this if statement is used. This code will cause the float value eMove to increase until larger or equal than eMoveMax. eMove is reset to a decreased value and the process repeats. An if statement was used over a while loop because a while loop would cause enemy movement to stutter when eMove value was decreased.

Enemy movement for each sprite is done separately as they all have unique movement. Therefore, I shall cover one movement loop in full and then explain how the for loop differs.

```
for (unsigned short int i = 0; i < enemySpritesX1.size(); i++)
```

I is equal to 0, while I less than the number of enemies, I increases. This will allow all of the next code to run.

```
if (enemySpritesA.at(i).getPosition().x < 0 - enemySpritesA.at(i).getGlobalBounds().width)
{
    enemySpritesA.erase(enemySpritesA.begin() + i);
}
```

This code will check the position of enemy A. If enemy position is less than 0 – the sprites width then the enemy will be erased.

The reason the code is 0 – width is so that the enemy is deleted when out of the window. Otherwise, the sprite would be erased when its top left corner is less than 0. This does not as good compared to the sprite being deleted when off the screen.

Enemy sprites need to be deleted. This is so that enemy's size will not continue to grow as they will be considered to exist by the program even though the user cannot see them. After a while the large number of enemies that need to be accounted for could cause the program

to lag. This is bad as this could ruin a user's run and therefore create an unengaging game that is frustrating to play.

```
//frame rate consistenet movement
float movement = ENEMY_MOVEMENT_SPEED * dt;
//sprite movement
enemySpritesA.at(i).move(-movement , 0);
```

If the enemy is not off the screen, then the movement code will run. movement is the result of ENEMY_MOVEMENT_SPEED * dt. This is so that each device will run the game at a consistent speed, hence making the difficulty of the game also consistent.

Enemy A has simple movement, as x value decreases. This makes the enemy move left across the screen.

```
enemySpritesB.at(i).move(sin( 0.5) + eMove / 10, 1 / movement);
```

Enemy B uses the trigonometry value sin and the incrementing value (eMove) created above. This code will cause the enemy to move diagonally right. Due to the trig value, the movement of the sprite is not a straight line. The incremental value will cause the speed to increase, decrease significantly then gradually increasing again.

```
enemySpritesC.at(i).move(20*sin(eMove), movement);
```

Enemy C multiplies the sin value with eMove. This will result the enemy to have a movement pattern like a transverse wave. (Check to right for reference as screen shot cannot clearly represent movement).

```
enemySpritesD.at(i).move(0, movement);
```

Enemy D also has standard movement of it increasing its y value. This means it will move down the window

```
void Enemy::RandomiseEnemyOffset()
{
    //random value in the range 0 to 588
    _enemySpawnYOffset = rand() % (SCREEN_HEIGHT - 200);
    //random value in the range 0 to 1400
    _enemySpawnXOffset = rand() % (SCREEN_WIDTH);
}
```

The random enemy offset is used to give some enemies a random spawn position. This is so that enemies are less predictable and harder to play against.

```
for (unsigned short int i = 0; i < enemySpritesA.size(); i++)
{
    _data->window.draw(enemySpritesA.at(i));
}
```

This is the draw for loop for enemy A, each enemy has their own for loop.

This will keep drawing enemies to the screen when a new one spawns in.

```
const std::vector<sf::Sprite>& Enemy::GetSpriteA() const
{
    return enemySpritesA;
}
```

This is the example of the get sprite function. This is used to return an enemy sprite. This will be used in the game state.

Game State

```
enemy = new Enemy(_data);
flash = new Flash(_data);
gui = new GUI(_data);
```

New operators are used represent the classes of enemy, flash and gui so that they can be used in game state as they return non null pointers.

```
// update score function from gui class is called
gui->UpdateScore(_score);
```

The update score function is called. This will cause the integers of score to be converted into a string and added to the score string at the top of the game.

```
//if gamestate is = playing
if (GameStates::ePlaying == _gameState)
```

This will check to see if game is in play mode. If true the following code will run.

```
//enemy move function
enemy->EnemyMove(dt);
```

The enemy move function is called. This means while the game is playing the enemies will move, however when the game is over they will stop moving.

```
//enemy offset is randomised
enemy->RandomiseEnemyOffset();
```

The enemy offset function is called.

```
//if clock is greater than enemy spawn value then...
if (clock.getElapsedTime().asSeconds() > ENEMY_SPAWN_FREQUENCY_FAST)
```

This if statement will run if the clock is greater than the set value of the fast spawn rate in the DEFINTIONS file.

```
//enemy spawns
enemy->SpawnEnemyA();
enemy->SpawnEnemyB();
enemy->SpawnEnemyC();
enemy->SpawnScoreSprite();

//clock restart so the next enemies can spawn
clock.restart();
```

This if statement will call enemy a, b and c. This means that these enemies will spawn whenever the clock is grater than the set value.

Score sprite will also spawn.

Clock will restart. This is important as without this the enemies spawn in once and then be unable to spawn in again.

```
//if clock is greater than enemy spawn value then...
if (clock2.getElapsedTime().asSeconds() > ENEMY_SPAWN_FREQUENCY_SLOW)
{
    //enemy spawns
    enemy->SpawnEnemyD();
    //clock restart so the next enemies can spawn
    clock2.restart();
}
```

Enemy D has a slower spawn rate to the other sprites, therefore it has its own for loop.

```
//if enemy collides with player sprite then game state = over and clock restart
std::vector<sf::Sprite> enemySprite = enemy->GetSpriteA() && enemy->GetSpriteB() && enemy->GetSpriteC() && enemy->GetSpriteD();
```

Originally, I attempted to get all the enemy sprites and set it to local variable enemySprite. This had the intention of only needing to do this once, however errors appeared when I attempted to do this.

```
//if enemy collides with player sprite then game state = over and clock restart
std::vector<sf::Sprite> enemySprite = enemy->GetSpriteA(), GetSpriteB() ,GetSpriteC(), GetSpriteD();
```

I then tried using commas instead. This showed no errors that prevented compilation. However, when playing the game collisions would only work with the sprite at the start of the list. This would lead me to my next attempt:

```
//if enemy collides with player sprite then game state = over and clock restart
std::vector<sf::Sprite> enemySpriteA = enemy->GetSpriteA();
```

This will return the sprite for enemy A using the string enemySpriteA as a local variable. This attempt makes use of only on get enemy sprite function. I repeated this for loop with embedded if statement. This is not an efficient way of coding, however it makes the code work.

```
for (int i = 0; i < enemySpriteA.size(); i++)
```

For loop will last the entire game.

```
if (collision.CheckSpriteCollision(player->GetSprite(), 0.625f, enemySpriteA.at(i), 0.65f))
{
    _gameState = GameStates::eGameOver;

    clock.restart();
}
```

Uses the collision class to check the player sprite with enemy sprite. The scale of the enemy and player rectangles are smaller than the sprites. This means that the actual sprites will have to overlap properly for a collision to be registered. This is the same for every enemy / player interaction to make the game play feel fairer.

When there is a collision the game state is changed to game over

Clock restart function is also ran.

```
if (collision.CheckSpriteCollision(player->GetSprite(), scoringSprites.at(i)))
{
    //on collision score will increase
    _score++;
```

Collision between invisible scoring sprite and player sprite uses regular collision, without the use of scaling. There is always a guaranteed collision as the scoring sprite takes up the whole screen.

On collision score increments by 1.

```
//the score will be updated on the screen
gui->UpdateScore(_score);
```

The score is then updated using the update score function.

```
//the score enemy will be erased on collision
//as it always collides with player, it is always erased
scoringSprites.erase(scoringSprites.begin() + i);
```

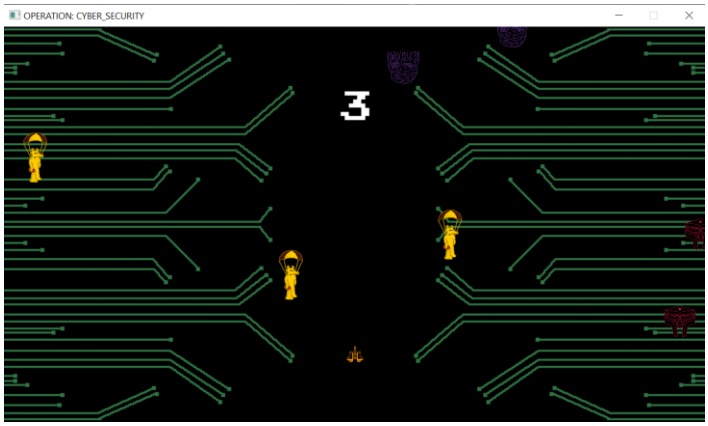
Scoring sprite will be erased after collision with the player. Even the sprite is not drawn to the screen, if too many are spawned without being erased the game can cause a drop in frame rate.

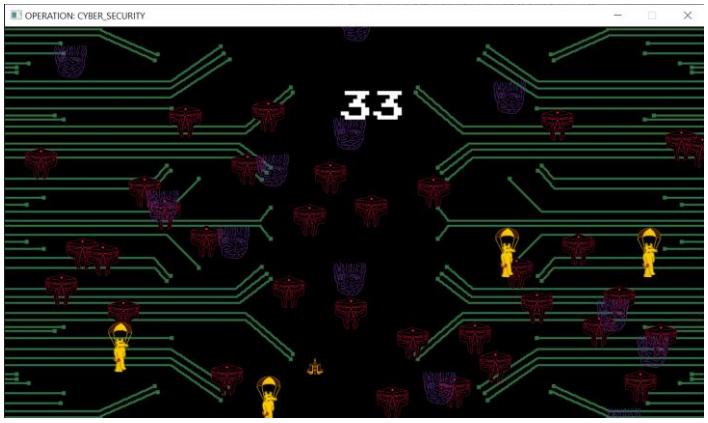
```
//if game state = game over
if (GameStates::eGameOver == _gameState)
{
    //flash effect code runs
    flash->Show(dt);
```

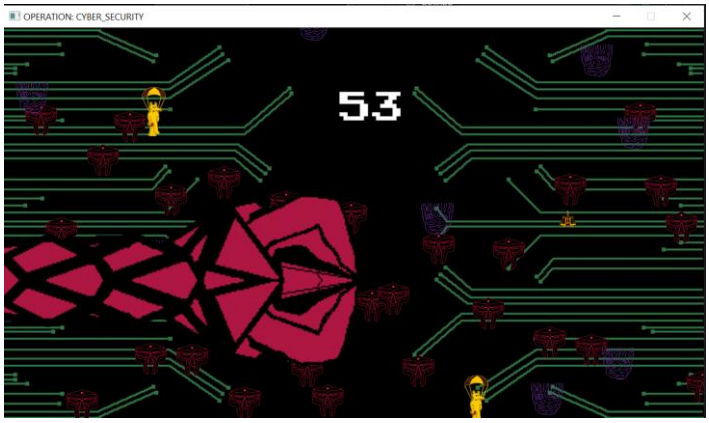
When game state is set to game over, then the flash effect happens.

```
//window clear
_data->window.clear();
//draw sprites to screen
_data->window.draw(_background);
enemy->DrawEnemy();
player->Draw();
movebg->DrawBackground();
flash->Draw();
gui->Draw();
//display function
_data->window.display();
```

All the new sprites added are added to the drawn function to be drawn to the screen.







Name of Test	Data	Expectation	Reality
Key W	W key	The player's character moves up the screen Y value decreases	Pass – player sprite moves up the screen.
Key A	A key	The player's character moves left across the screen X value decreases	Pass – player sprite moves across the screen.
Key S	S key	The player's character moves down the screen Y value increases	Pass – player sprite moves down the screen
Key D	D key	The player's character moves right across the screen X value increases	Pass – player sprite moves across the screen
Splash State	State	When code is first loaded, the splash state should be added	Pass – when code is ran, splash state is displayed. This

		to stack. This will then display this state to the window. After short delay this state will be switched out to main menu state.	shows image for a couple seconds before changing state
Main Menu State	State, button	This state will should load after splash state. This state will need to hold buttons that will be used to navigate to all other states	Pass – this state loads after splash state and holds buttons that can be pressed to change
Options state	State, button	This state should have a back button to return to menu and a form of option such as change in key binds or colour blind mode	Fail - unattempted
Education state	State, button	This state should offer an education and way to return to main menu state	Fail - unattempted
Game State	State	This should include the actual game play. Enemies and data classes should be used here to create cohesive game. The game state should be popped and game over state should be pushed on player collision with enemy.	Pass – Game state will be loaded after pressing the play button. Main menu is popped and game state is pushed.
Game over state	State, button, string, txt file	This should load after the game is over. This will give user option to return to the main menu, but also have their	Fail - unattempted

		score outputted as a string and the high score to be outputted to the screen	
Play Button	Click button	Clicking on this sprite will pop main menu state and push game state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Options Button	Click button	Clicking on this sprite will pop main menu state and push option state	Fail - unattempted
Education button	Click button	Clicking on this sprite will pop main menu state and push education state	Fail - unattempted
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	Fail - unattempted
Collision	Player position = Malware position and	If player position intersects with enemy position then the screen needs to flash. The game over state needs to pushed to top and previous state should be popped	Pass- player and enemies have rects drawn around them. When player rectangle intersects the one of the player them game is equal to over
Flash	rect	A translucent rectangle will be drawn to the screen on collision	Pass: A translucent rectangle will be drawn to the screen on collision
Score	Time, score	High score is generated based on time survived.	Pass: score is generated by having invisible

			sprite spawn and move across the window. On collision the sprite is erased and score increments
Options	Options button	Additional option such as option to change key bind or add colour blind mode. Should be found in options	Fail - unattempted.
Education	Education sprite	A sprite of the education needs to be printed to the screen. This will sprite will have a texture of text with useful facts.	Fail - unattempted
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will have their score saved to text file. This text file will have to be open to retrieve the high score	Fail - unattempted
Enemies		Multiple enemies are created out of the window before moving into it. These enemies will have unique movement and collisions with player. Enemy should be deleted when out of the screen fully.	Pass- 4 different enemies are spawned in with different starting positions and different movement patterns
Player and enemy shooing	sprite	Enemies shoot and player shoot. This is done with shoot class that creates an object	This has not be done, as the shoot class did not work due to parameter problems with the state machine

I have successfully fleshed out the game. Despite it not being my original plan or idea, it serves it purpose as a game. My project is also no longer a bullet hell, but a game that shares some elements from it. Unfortunately, I had to cut the shooting functions as I had problems with the parameters of the function not fitting into the state machine functions. Due to the reduced time restraints, I will cut this feature, with hope of including this gameplay mechanic in a later prototype.

The gameplay of the game is simplistic in nature and design yet manages to maintain an effective difficulty curve capable of keeping a wide audience engaged. Throughout the gameplay there are fast moving purple and yellow enemies that have unpredictable movement that demands the player's attention. As these enemies are being spawned there are several slow-moving red enemies with a high spawn rate. This leads to the screen being cluttered with enemies at around 20 score. This limits the players ability to move without a collision, creating an effective difficulty curve. Once their ability has improved to get to around 50 points a new enemy is introduced – the arm. The arm enemy has a large sprite size and makes left side of the screen extremely dangerous. This in my opinion the safest place to be at the start, as it is difficult to get boxed in by red enemies. Due to these reasons, I believe my game has enough depth to be engaging.

I currently have a form of game that serves as gameplay. This leaves only the other states that need to be included such as education state and the game over state. After these inclusions the game will have included everything, I have set out to complete.

Rate Prototype //Score: 0 out of 10						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	0	8	7	8	6	After game screen
George	0	9	7	8	7	Quick replay option
Callum	0	9	8	8	7	Options
Drew	0	8	8	7	5	Reward system
Sum	0	34	30	31	25	
Average	0	8.5	7.5	7.75	6.25	

The ratings of this prototype have improved significantly. My next focus will be on bettering the menu so that there are options, the option to go into another game after playing, and education.

Prototype 3

Game Over State

To start with I am going to include a game over state, which will allow for quick replays. Currently users must close the program and reload after playing. This in not an enjoyable experience to do therefore this shall be done first

```
//class inherits from public state
class GameOverState : public State
{
public:
    //constructor
    GameOverState(GameDataRef data, int score);
    //state machine functions
    void Init();
    void HandleInput();
    void Update(float dt);
    void Draw(float dt);
```

First, I coded the usual code in the header file for a state. This includes the inheritance from State, constructor and state functions.

```
//struct reference
GameDataRef _data;
```

Reference to structure in game file.

```
//sprite
sf::Sprite _background;
sf::Sprite _gameOverTitle;
sf::Sprite _gameOverContainer;
sf::Sprite _retryButton;
sf::Sprite _medal;
```

Next, I set up the sprites for me to use in the cpp file.

```
//text
sf::Text _scoreText;
sf::Text _highScoreText;
```

I then set up the text

```
//int values
int _score;
int _highScore;
```

And also, the integer values

```
//constructor
GameOverState::GameOverState(GameDataRef data, int score) : _data(data), _score(score)
{
}
```

In the cpp file a constructor is used. This constructor uses the reference to the GameData allowing foe the state to be successfully connected and thus constructed.

```
//will read from respective text file
std::ifstream readFile;
readFile.open("TextFile/Highscores.txt");
```

If stream is used to open the high score text file created.

```
if (readFile.is_open())
{
    //reads whole file and set that value as high score
    while (!readFile.eof())
    {
        readFile >> _highScore;
    }
}
//closes file
```

If the file is opened, the whole file will be read (eof() meaning end of file function). The last line will be interpreted as the high score integer value.

```
//closes file
readFile.close();
```

File is then closed.

```
//write to highscore text file
std::ofstream writeFile("TextFile/Highscores.txt");
```

This creates the local variable writeFile of ofstream for the text file Highscores.txt. This will allow for the file to be written in.

```
if (writeFile.is_open())
{
    //if the score of current run is higher than high score
    if (_score > _highScore)
    {
        //highscore become score value
        _highScore = _score;
    }
    //new score is written into high score file
    writeFile << _highScore;
```

This will compare the values of _highScore from the text file to _score. If _score has a higher value than _highScore, then _highScore is overwritten and the new high score is written to the text file.

```
//closes the file
writeFile.close();
```

File is then closed.

```
//loads textures from DEFINITION file
_data->assets.LoadTexture("Game Over State Background", GAMEOVER_STATE_BACKGROUND_FILEPATH);
_data->assets.LoadTexture("Game Over State Title", GAMEOVER_STATE_TITLE_FILEPATH);
_data->assets.LoadTexture("Game Over State Body", GAMEOVER_STATE_BODY_FILEPATH);
_data->assets.LoadTexture("Bronze Medal", BRONZE_MEDAL_FILEPATH);
_data->assets.LoadTexture("Silver Medal", SILVER_MEDAL_FILEPATH);
_data->assets.LoadTexture("Gold Medal", GOLD_MEDAL_FILEPATH);
_data->assets.LoadTexture("Diamond Medal", DIAMOND_MEDAL_FILEPATH);
```

Textures are loaded from the DEFINITIONS file. This includes textures for the background, title, body image and medals.

```
//set textures to sprites
_background.setTexture(this->_data->assets.GetTexture("Game Over State Background"));
_gameOverTitle.setTexture(this->_data->assets.GetTexture("Game Over State Title"));
_gameOverContainer.setTexture(this->_data->assets.GetTexture("Game Over State Body"))
_retryButton.setTexture(this->_data->assets.GetTexture("Play Button"));
```

The textures of the background, game over title, body image and retry button are all set here.

```
//set positions of sprites
_gameOverContainer.setPosition((_data->window.getSize().x / 2) - (_gameOverContainer.getGlobalBounds().width / 2),
    (_data->window.getSize().y / 1.9) - (_gameOverContainer.getGlobalBounds().height / 2));
```

The body image has its position set to be in the middle of the window (minus the textures width to make it more centre as the left corner would be in the centre of the x axis). The y position is set to be around the middle of the screen.

```
_gameOverTitle.setPosition((_data->window.getSize().x / 2) - (_gameOverTitle.getGlobalBounds().width / 2),
    _gameOverContainer.getPosition().y - (_gameOverTitle.getGlobalBounds().height * 1.08));
```

The position of the title has the same x position of the sprite above. However, the y position is changed so that is above the body image sprite.

```
_retryButton.setPosition((_data->window.getSize().x / 2) - (_retryButton.getGlobalBounds().width / 2),
    _gameOverContainer.getPosition().y + _gameOverContainer.getGlobalBounds().height + (_retryButton.getGlobalBounds().height * 0.08));
```

The retry button has the same x position as the previous 2 sprites. However, it has a y position that is lower than the other two sprites.

The title and retry button both use the _gameOverContainer sprite(body image) to be positioned easier.

```
//set values to the score text text
_scoreText.setFont(_data->assets.GetFont("Font"));
_scoreText.setString(std::to_string(_score));
_scoreText.setCharacterSize(32);
_scoreText.setFillColor(sf::Color::White);
_scoreText.setOrigin(_scoreText.getGlobalBounds().width / 2, _scoreText.getGlobalBounds().height / 2);
_scoreText.setPosition(_data->window.getSize().x / 10 * 6.35, _data->window.getSize().y / 2.0f);
```

This code will set variables of the score text in the game over state.

```
//set values to the high score text text
_highScoreText.setFont(_data->assets.GetFont("Font"));
_highScoreText.setString(std::to_string(_highScore));
_highScoreText.setCharacterSize(32);
_highScoreText.setFillColor(sf::Color::White);
_highScoreText.setOrigin(_highScoreText.getGlobalBounds().width / 2, _highScoreText.getGlobalBounds().height / 2);
_highScoreText.setPosition(_data->window.getSize().x / 10 * 6.35, _data->window.getSize().y / 1.6f);
```

This is done similarly to the score text.

```
//medal if statements
//score will find the highest medal it can be assigned, this becomes the awarded medal
if (_score >= DIAMOND_MEDAL_SCORE)
{
    _medal.setTexture(_data->assets.GetTexture("Diamond Medal"));
}
```

If the score of the game played is higher than the value of the score set to the respective definition, then a medal can be awarded. The if loops determine what texture is applied to the _medal sprite.

```
else if (_score >= GOLD_MEDAL_SCORE)
{
    _medal.setTexture(_data->assets.GetTexture("Gold Medal"));
}
```

The rest of the other possible textures use else if statements.

This works as if the score is too low for the diamond medal, then it will check the gold medal. This process will continue until the appropriate texture is applied. If the score was not high enough, the medal is still drawn, however nothing will be visible as no texture is applied to the sprite.

```
_medal.setPosition(477, 370);
```

Medal position is set. This position will be in the spot inside the main body image.

```
//event loop
sf::Event event;
while (_data->window.pollEvent(event))
{
    //closes window if close is clicked
    if (sf::Event::Closed == event.type)
    {
        _data->window.close();
    }
```

The handle input function contains the event loop and the necessary close window if statement.

```
//if retry button is pressed, user is taken menu state
if (_data->input.IsSpriteClicked(_retryButton, sf::Mouse::Left, _data->window))
{
    _data->machine.AddState(StateRef(new MainMenuState(_data)), true);
}
```

Also in the handle input function is the button to take user to main menu after playing the game. This is done through the add state function created in the state machine file.

If the left mouse button is clicked over the play button sprite, then a new state is added – the main menu state. This will also pop the game over state, as there is no reason for it to be loaded and sit on the stack.

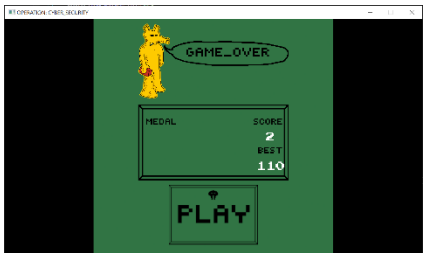
Validation is used here to make sure that the user wants to enter the game state. There is validation here to check if the mouse cursor is in the same position to the sprite in relation to the window. There is another validation check checking for mouse clicks on the sprite.

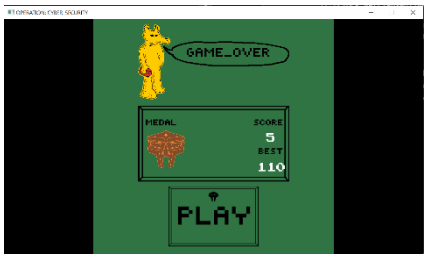
```
//draw sprites to screen
_data->window.clear();
_data->window.draw(_background);
_data->window.draw(_gameOverTitle);
_data->window.draw(_gameOverContainer);
_data->window.draw(_retryButton);
_data->window.draw(_scoreText);
_data->window.draw(_highScoreText);
_data->window.draw(_medal);
//display function
_data->window.display();
```

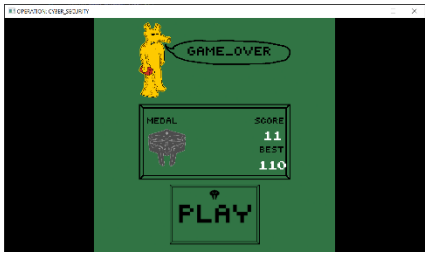
All sprites are drawn and then displayed on the window.

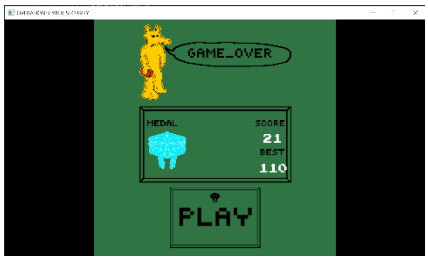
```
//clock reset on collsion so the clock will restart
//this will give a set delay for every run
if (clock.getElapsedTime().asSeconds() > GAMEOVER_DELAY)
{
    //state is changed to game over state
    _data->machine.AddState(StateRef(new GameOverState(_data, _score)), true);
}
```

When game state is equal to the Game over, then this code will run. Due to clocks resetting on collisions with enemies, the game delay value will be consistent on each death. When clock has surpassed the value of game over delay then state will change by adding state of GameOverState.

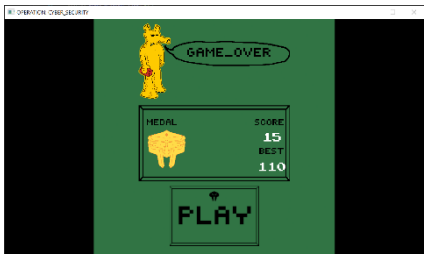








Originally there was going to be social aspects of user's typing in their name, however there is risk they type in inappropriate words. Therefore, I have opted to a medal system to encourage and motivate users to keep playing the game. A that did something similar was Flappy Birds. This was successful in keeping users engaged, so I am hoping this system will be successful for me too.



Education

```
//load textures
_data->assets.LoadTexture("Learn Text", LEARN_TEXT_FILEPATH);
_data->assets.LoadTexture("Back Button", BACK_BUTTON_FILEPATH);
```

Textures are loaded into the init function.

```
//set textures
_background.setTexture(this->_data->assets.GetTexture("Main Menu Background"));
_learnText.setTexture(this->_data->assets.GetTexture("Learn Text"));
_backButton.setTexture(this->_data->assets.GetTexture("Back Button"));
```

Textures are set to sprites in the education state.

```
//set position
_learnText.setPosition((SCREEN_WIDTH / 2) - (_learnText.getGlobalBounds().width / 2), _learnText.getGlobalBounds().height / 10);
_backButton.setPosition((SCREEN_WIDTH / 2) - (_backButton.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 1.2) - (_backButton.getGlobalBounds().height / 2));
```

Set position of the sprites on the screen. The education sprite is at the top of the screen, with the back button at the bottom (below the learn text sprite).

```
sf::Event event;
//event loop
while (_data->window.pollEvent(event))
{
    if (sf::Event::Closed == event.type)
    {
        _data->window.close();
    }
```

Event loop to close window and for events to take place in this state.

```
//state change to menu
if (_data->input.IsSpriteClicked(_backButton, sf::Mouse::Left, _data->window))
{
    _data->machine.AddState(StateRef(new MainMenuState(_data)), true);
}
```

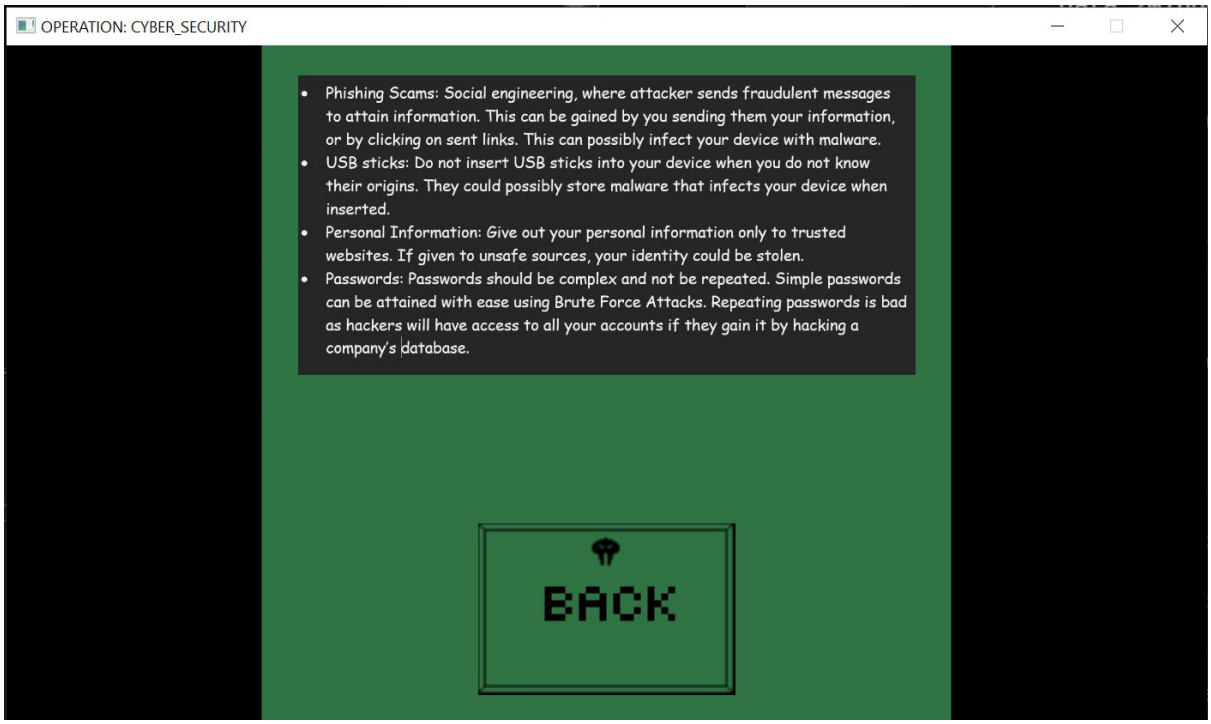
Allows access back to main menu

If the left mouse button is clicked over the back button sprite, then a new state is added – the main menu state. This will also pop the education state, as there is no reason for it to run.

Validation is used here to make sure that the user can gain access to main menu state. There is validation here to check if the mouse cursor is in the same position to the sprite in relation to the window. There is another validation check checking for mouse clicks on the sprite.

```
void LearnState::Draw(float dt)
{
    //clear / draw / display
    _data->window.clear();

    _data->window.draw(_background);
    _data->window.draw(_learnText);
    _data->window.draw(_backButton);

    _data->window.display();
}
```

Draws everything to screen



Options

```
//load textures from definitions
_data->assets.LoadTexture("Option", OPTION_FILEPATH);
_data->assets.LoadTexture("Back Button", BACK_BUTTON_FILEPATH);
```

The texture is loaded from the definitions file. The definition was given a string value so that it can be used. This will give texture for the option button and the back button.

```
//set textures to the sprites
_background.setTexture(this->_data->assets.GetTexture("Main Menu Background"));
_option.setTexture(this->_data->assets.GetTexture("Option"));
_backButton.setTexture(this->_data->assets.GetTexture("Back Button"));
```

The textures that will be used in this state will be set to the sprites.

```
//set positions
_option.setPosition((SCREEN_WIDTH / 2) - (_option.getGlobalBounds().width / 2), _option.getGlobalBounds().height / 10);
_backButton.setPosition((SCREEN_WIDTH / 2) - (_backButton.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 1.2) - (_backButton.getGlobalBounds().height / 2));
```

The position of the option and back sprite are loaded. Background position does not need to be changed as this will take up whole window. The position of the option button and the back button are set so that option is in the top middle of the screen and back button is in the bottom position.

```
while (_data->window.pollEvent(event))
{
    if (sf::Event::Closed == event.type)
    {
        _data->window.close();
    }
```

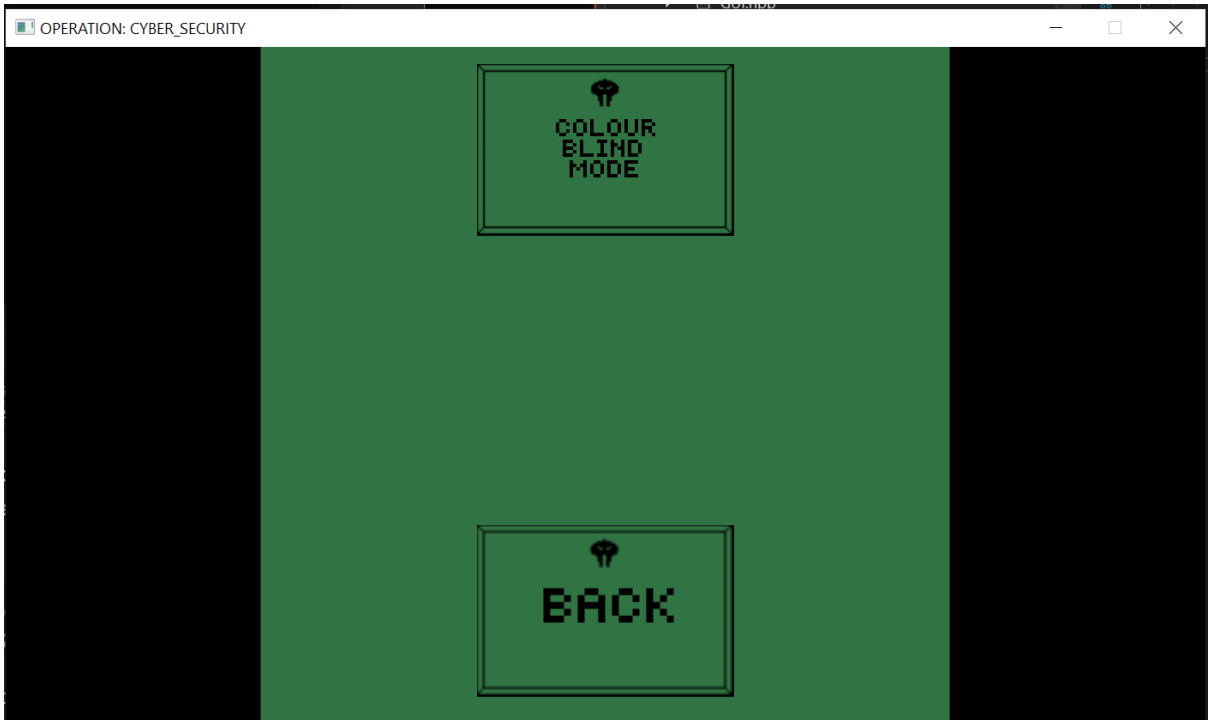
The necessary while event loop that will allow the state to close by clicking the close button.

```
//change state to main menu
if (_data->input.IsSpriteClicked(_backButton, sf::Mouse::Left, _data->window))
{
    _data->machine.AddState(StateRef(new MainMenuState(_data)), true);
}
```

The back button will allow the user to leave this state by adding the state of main menu.

```
void OptionState::Draw(float dt)
{
    //clear
    _data->window.clear();
    //draw
    _data->window.draw(_shape);
    _data->window.draw(_background);
    _data->window.draw(_option);
    _data->window.draw(_backButton);
    _data->window.draw(_shape);
    //display
    _data->window.display();
}
```

This code will allow the sprites to be drawn to the window and displays it.



Name of Test	Data	Expectation	Reality
Key W	W key	The player's character moves up the screen Y value decreases	Pass – player sprite moves up the screen.
Key A	A key	The player's character moves left across the screen X value decreases	Pass – player sprite moves across the screen.
Key S	S key	The player's character moves down the screen Y value increases	Pass – player sprite moves down the screen
Key D	D key	The player's character moves right across the screen X value increases	Pass – player sprite moves across the screen
Splash State	State	When code is first loaded, the splash state should be added to stack. This will then display this state to the window. After	Pass – when code is ran, splash state is displayed. This shows image for a couple seconds before changing state

		short delay this state will be switched out to main menu state.	
Main Menu State	State, button	This state will should load after splash state. This state will need to hold buttons that will be used to navigate to all other states	Pass – this state loads after splash state and holds buttons that can be pressed to change
Options state	State, button	This state should have a back button to return to menu and a form of option such as change in key binds or colour blind mode	Pass- this state will load the option state done by popping menu state and pushing the option state
Education state	State, button	This state should offer an education and way to return to main menu state	Pass- this state will load the education state done by popping menu state and pushing the education state
Game State	State	This should include the actual game play. Enemies and data classes should be used here to create cohesive game. The game state should be popped and game over state should be pushed on player collision with enemy.	Pass – Game state will be loaded after pressing the play button. Main menu is popped, and game state is pushed.
Game over state	State, button, string, txt file	This should load after the game is over. This will give user option to return to the main menu, but also have their score outputted as a string and	Pass- game over state will be pushed after popping the game state on collision with enemy (plus short delay)

		the high score to be outputted to the screen	
Play Button	Click button	Clicking on this sprite will pop main menu state and push game state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Options Button	Click button	Clicking on this sprite will pop main menu state and push option state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Education button	Click button	Clicking on this sprite will pop main menu state and push education state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	Pass – when the player collides with an enemy game state is changed to game over. This cause a flash and then the user is taken to game over state. Does not allow for the player to input their name
Collision	Player position = Malware position and	If player position intersects with enemy position then the screen needs to flash. The game over state needs to	Pass- player and enemies have rects drawn around them. When player rectangle intersects the

		pushed to top and previous state should be popped	one of the player them game is equal to over
Flash	rect	A translucent rectangle will be drawn to the screen on collision	Pass: A translucent rectangle will be drawn to the screen on collision
Score	Time, score	High score is generated based on time survived.	Pass: score is generated by having invisible sprite spawn and move across the window. On collision the sprite is erased and score increments
Options	Options button	Additional option such as option to change key bind or add colour blind mode. Should be found in options	Pass – a pass button created that can be clicked. Fail – I have not put any code in the if statement.
Education	Education sprite	A sprite of the education needs to be printed to the screen. This will sprite will have a texture of text with useful facts.	Pass – a sprite with texture of informative text is printed to the screen
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will have their score saved to text file. This text file will have to be open to retrieve the high score	Pass – Score is written in to the txt file. Previous score is erased
Enemies		Multiple enemies are created out of the window before moving into it. These enemies will have unique movement and	Pass- 4 different enemies are spawned in with different starting positions and different movement patterns

		collisions with player. Enemy should be deleted when out of the screen fully.	
Player and enemy shooing	sprite	Enemies shoot and player shoot. This is done with shoot class that creates an object	This has not be done, as the shoot class did not work due to parameter problems with the state machine

From completing the checklist, I can see that I have completed everything but the options button. Clicking the options button was originally going to change the key binds to the arrow keys or add a filter of colour to make the game more accessible to colour blind people.

Apart from this I have now completed the game that can be used as an educational resource that solves the problem of young people not knowing enough about internet safety / cyber security.

Rate Prototype //Score: 0 out of 10						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	3	8	7	8	6	Lack of accessibility
George	5	9	7	8	7	Education is lacklustre
Callum	3	9	8	8	7	Options
Drew	2	8	8	7	5	Score input doesn't accept
Sum	13	34	30	31	25	
Average	3.25	8.5	7.5	7.75	6.25	

The final reviews from users suggest that the game was very playable that can be replayed multiple times without becoming tedious. The graphics were also considered to be well received; this was likely due to the player sprite having animation and the sliding background when the game first starts.

The gameplay was only thought to be ok. This is likely due to the game having simple mechanics and simple objective to complete. However, the gameplay is serviceable enough; it is simple enough and can be restarted quickly allowing the game to be enjoyable in short cycles. There was also an effective difficulty curve, which makes the game more enjoyable.

Despite this, some stakeholders still believe that the game provided some lacklustre gameplay. This could be resolved by attempting to include the shooting again. This can be done quickly by possibly not doing health and having instant kill on shot. This would require limited ammunition or a slow rate of fire.

Finally, education was done last. Education was not well received, likely as because users can simply ignore this part of the program and just play the game. A way this could have been improved, if this was integrated into the gameplay. An example of this could have been a quiz state in between the game state and game over state. Alternatively, I can attempt to expand the information states by adding more and by making the reading more enjoyable. This could possibly be done by having animation on these screens.

3.2 Post Development Testing

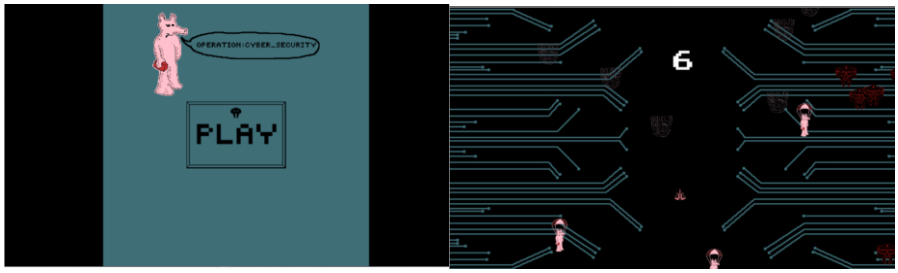
To start Post Development Testing, I shall first add all some features that was not included. This will allow me to have more code to do a variety of tests on. Some of the areas I would be adding to is the options, education, shooting, health and name input. These are important for the game as they

Options Menu / Accessibility

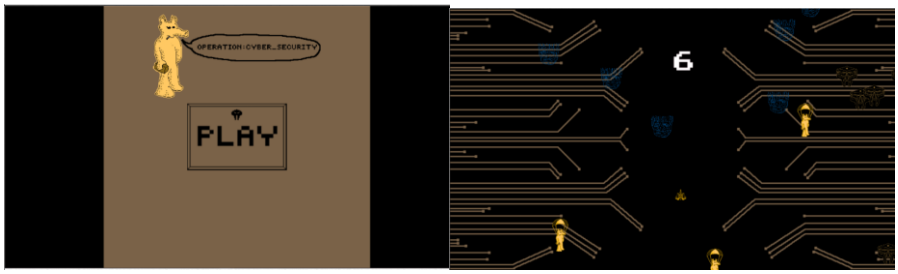
The options menu does not have much purpose as it just contains a button, which contains code that does not work. Therefore, my game so far lacks any accessibility features, which decreases the total audience that can learn from my project. To improve my game, I need to include the options. So far, I have the inclusion of buttons that check for clicks and position of the mouse. Therefore, I must add the code to make the buttons have an effect.

After some testing I have concluded that the colours in the game can still be easily distinguished from each other.

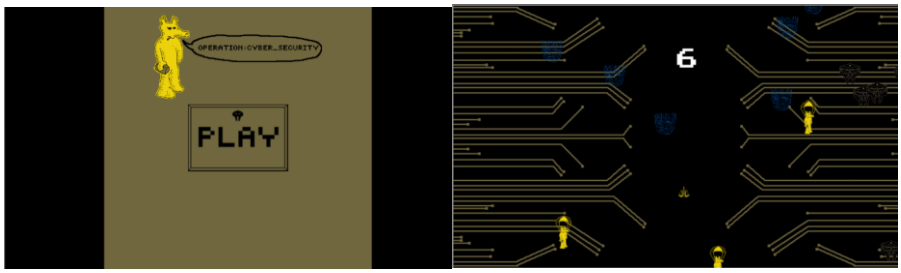
Tritanopia:



Deuteranopia:



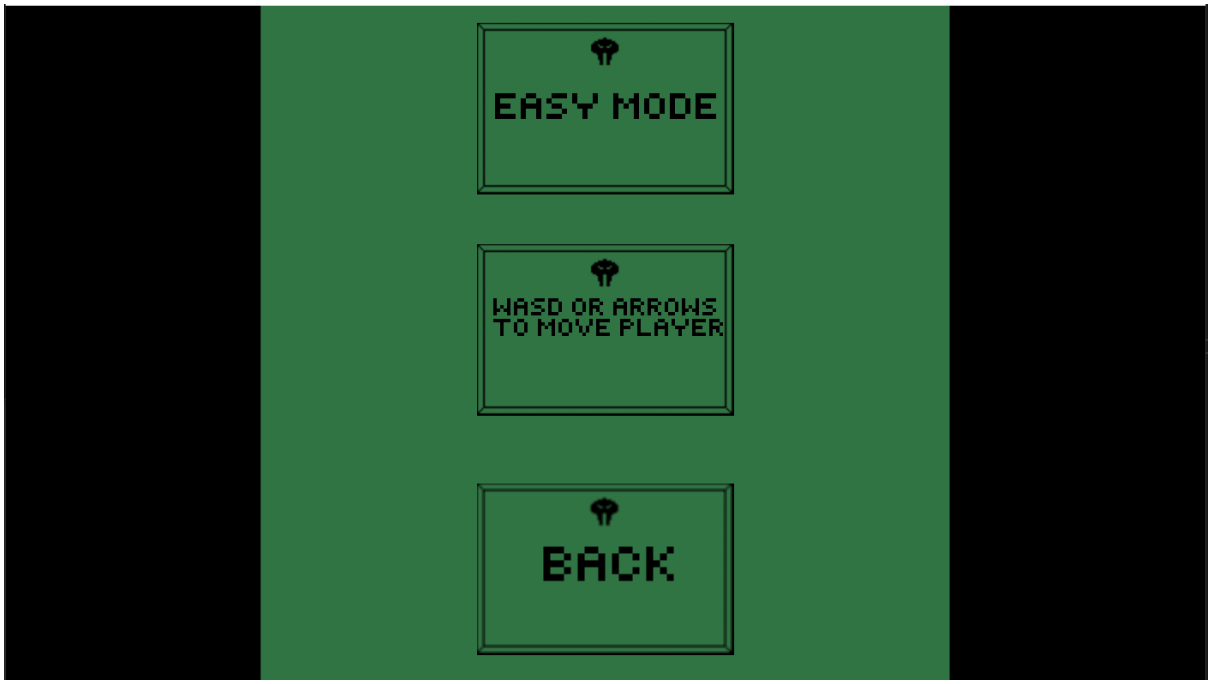
Protanopia:



Hence, I shall include an alternate accessibility feature to my game.

```
//player movement
#define PLAYER_MOVEMENT_SPEED 2.0f
#define PLAYER_MOVEMENT_UP  sf::Keyboard::isKeyPressed(sf::Keyboard::W) xor sf::Keyboard::isKeyPressed(sf::Keyboard::Up)
#define PLAYER_MOVEMENT_DOWN sf::Keyboard::isKeyPressed(sf::Keyboard::S) xor sf::Keyboard::isKeyPressed(sf::Keyboard::Down)
#define PLAYER_MOVEMENT_LEFT sf::Keyboard::isKeyPressed(sf::Keyboard::A) xor sf::Keyboard::isKeyPressed(sf::Keyboard::Left)
#define PLAYER_MOVEMENT_RIGHT sf::Keyboard::isKeyPressed(sf::Keyboard::D) xor sf::Keyboard::isKeyPressed(sf::Keyboard::Right)
```

To make it easier for users who prefer the arrow keys or have physical disabilities, which will prevent use of the wasd keys, I have bound the arrow keys to movement code. I have used xor as to prevent the possibility of the player being able to potentially double their movement speed by using both wasd and arrow keys.



I have also added a new image to the options menu. This image now informs users what keys to use to move the player.

I then gave functionality to the button in the options menu. This is now become easy mode.

```
if (_data->input.IsSpriteClicked(_option, sf::Mouse::Left, _data->window))
{
    if (!ENEMY_MOVEMENT_SPEED == 10.f)
        ENEMY_MOVEMENT_SPEED -20;
    if (!ENEMY_MOVEMENT_SPEED_2 == 10.f)
        ENEMY_MOVEMENT_SPEED_2 - 20;
}
```

This will change the enemy speed of the enemy so that they move at reduced speed. This will make the game more accessible to less skilled players. This will make sure that more players are able to play the game without getting frustrated.

These decreased values seem to have little effect when the button is clicked, but I will continue to use this because I don't think the game will be fun if the speed values was significantly decreased.

Education

Education was another cause for concern for the stakeholders at the end of the third prototype. This was because it was just a wall of text. Another cause was that the sprites had little to do with education. These can be remedied by having each sprite/texture being representative of a type of malware. When the education is loaded it can have the enemies appear and be clickable. Once clicked the player can then read about the malware. This will make the education more focused and relevant for the user. This education was originally meant to become a quiz, but this may not be feasible due to time constraints.

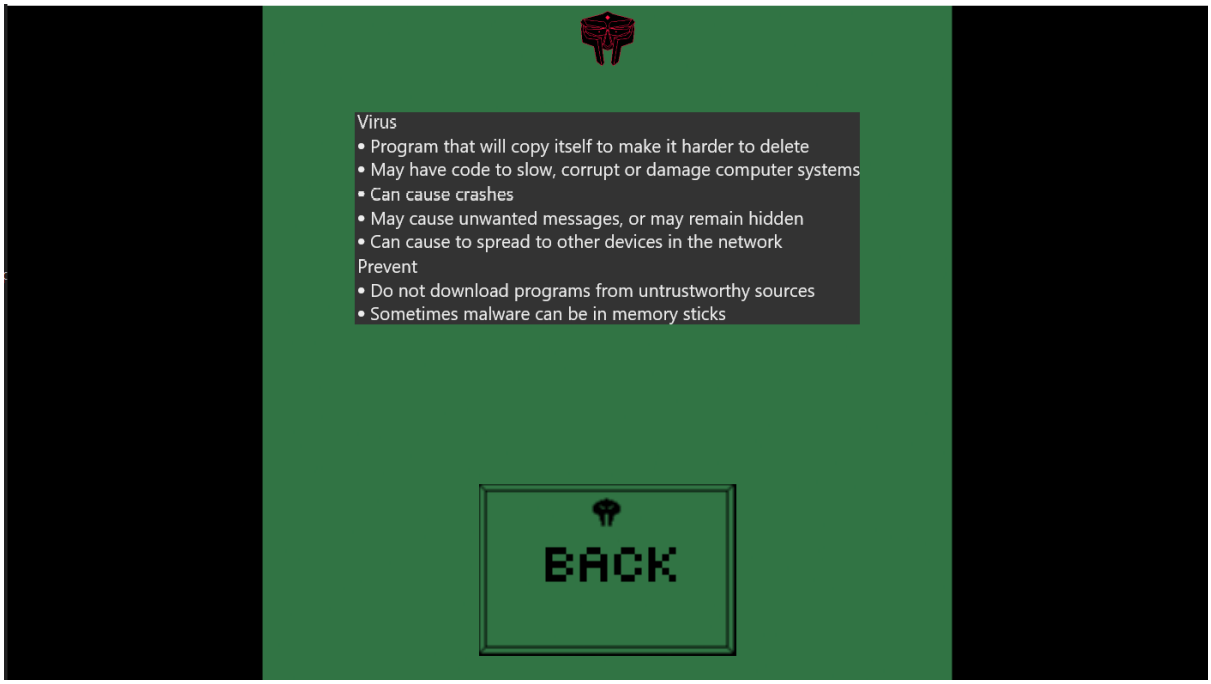
```
//set position
_malwareA.setPosition((SCREEN_WIDTH / 3) - (_malwareA.getGlobalBounds().width / 2), _malwareA.getGlobalBounds().height / 10);
_malwareB.setPosition((SCREEN_WIDTH / 1.5) - (_malwareB.getGlobalBounds().width / 2), _malwareB.getGlobalBounds().height / 10);
_malwareC.setPosition((SCREEN_WIDTH / 3) - (_malwareC.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 2.5) - _malwareC.getGlobalBounds().height / 7);
_malwareD.setPosition((SCREEN_WIDTH / 1.5) - (_malwareD.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 2.5) - _malwareD.getGlobalBounds().height / 7);
```

I have removed the learn text that was originally the source of education and started by having the images that represents the type of malware and threats.

The sprites do not appear to that visually appealling but will work fine.

Next I will have the images be clickable which will lead to another state. In these states there will be a fact file on on each. The information will be concised but very informative.

This was done by having code similar to how it was but for each type. This state no longer gives any information, but now acts as a hub for the user to traverse between the information.



This is the example of how Virus will look, but the other 3 will follow the same style. This state can be reached by clicking the image of the sprite the user wants to learn more about. When back button is pressed user will return to hub.

Music

Another addition made was the inclusion of music. This will make the game more enjoyable to be played.

```
sf::Music music;
if (!music.openFromFile("Music/menuMusic.ogg"))
    std::cout << "error" << "\n";
music.play();
music.setLoop(true);
```

This works like how textures and sprites work. Music will be opened and streamed dynamically from the music file. This different to normal sound effects which will be done statically. This is done because music size can be large which could potentially have a negative effect on compilation speed. Once the music end, the song will then loop to the start. An error message if the music is not successfully found.

Score

The score needs to be changed. This currently works via an unconventional method of having collisions with an invisible sprite being responsible for scoring. This is not great for this type of game and is meant for games where getting to a certain location will get score rather than time based scoring (such as flappy birds). This due to players being closer to the top of the screen will be able to get points slightly faster. Despite this having a negligible effect on the game, for consistency I shall change this.

This has been changed to be a simple time loop. Once the value reaches a certain point the game score will increase and then counter resets. This will be a more consistent counter to time score.

3.2.1 Robustness

Movement / Bounds

Movement and bounds should be tested to make sure that the player cannot escape the bounds. This can be tested by trying each pixel location of the screen. This can be done easily by commenting out the code that changes the state to game over state on collision and taking my time checking each spot multiple times.

```
//_gameState = GameStates::eGameOver;
```

I will also try to collide with the same spot multiple times with a high frequency in attempt to break the bounds.

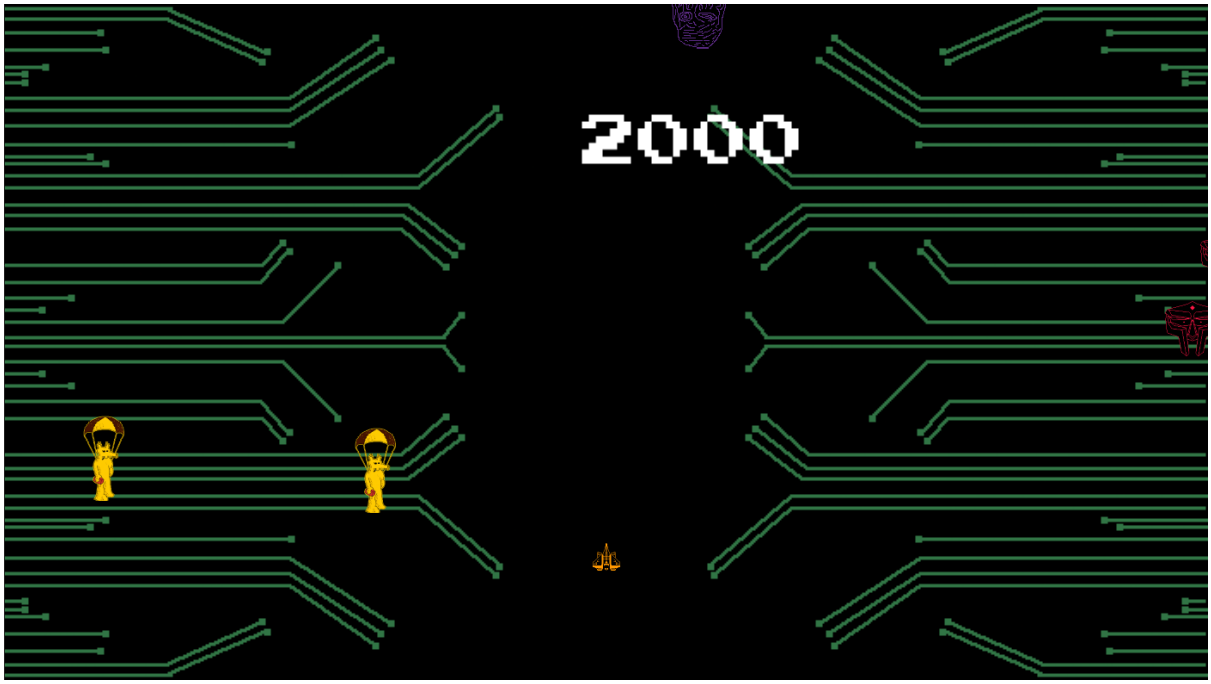
I can confidently conclude that the game's bounds have successfully contained the player to the window without issue. I made several attempts to escape the bounds but was unsuccessful at each attempt.

Score

The score is important to consider. As the player carries on playing the score will continually increment in value, however this may cause problems which I should attempt to investigate.

```
_score = _score + 1000;
```

Instead of having the score increment by 1, I have changed it, so it increases by 1000. This will quickly assist in finding problems.



An immediate problem is that as quadruple digits are reached the score will no longer be centred. This is because the number furthest to the left will be centred and not move as the score increases. However, I do not think this is only noticeable at triple digits. The game gets very difficult after about 30 seconds. Therefore, it is unlikely for users to really notice this.

```
_score = _score + 1000000000000000000;
```

I tested with very large numbers to check the output. This causes the value of score to fluctuate between large positive and large negative values. This is due to me not using unsigned int for score. Unsigned int means that there are no negative values, so larger numbers can be attained.

```
void GUI::UpdateScore(unsigned int score)
```

To fix this I changed to score to accommodate for very large numbers to prevent problems. Realistically this change was unnecessary as the chances of a score so high it causes problems is unlikely. However, this change will make the scoring more robust.

Enemy spawning

I need to make sure that enemies will spawn forever as this game does not use levels, but an infinite run model. Enemies should spawn in as long as the player is alive.

This was tested alongside score as they can be tested simultaneously, as it required me to stay alive for a long period of time to see if they continued spawning.

After waiting 10 minutes with no effect on collisions; I have concluded that enemies will spawn continuously without fault.

Player / Enemy collisions

Player collisions with the enemy should end the game. This should cause a flash on screen and change the state of the game to end game state.

After several runs at the game, most collisions will cause the game to end as stated above. The collisions that did not cause the game to end were due to this code:

```
if (collision.CheckSpriteCollision(player->GetSprite(), 0.625f, enemySpriteA.at(i), 0.65f))
```

This line of code will cause the hit boxes of the sprites to be smaller and closer to the centre of the sprite. Sprites will have a rectangular hitbox that will go further than the sprite. This was therefore likely not a bug, but the intentional feature to prevent unfair deaths.

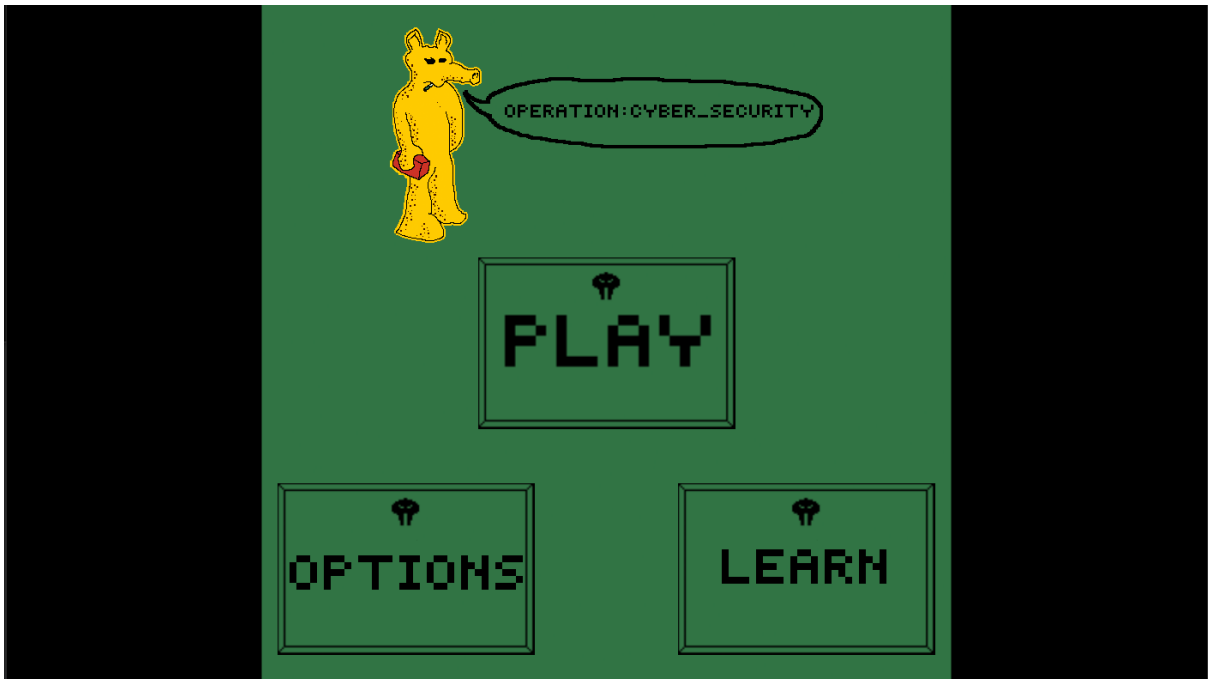
States

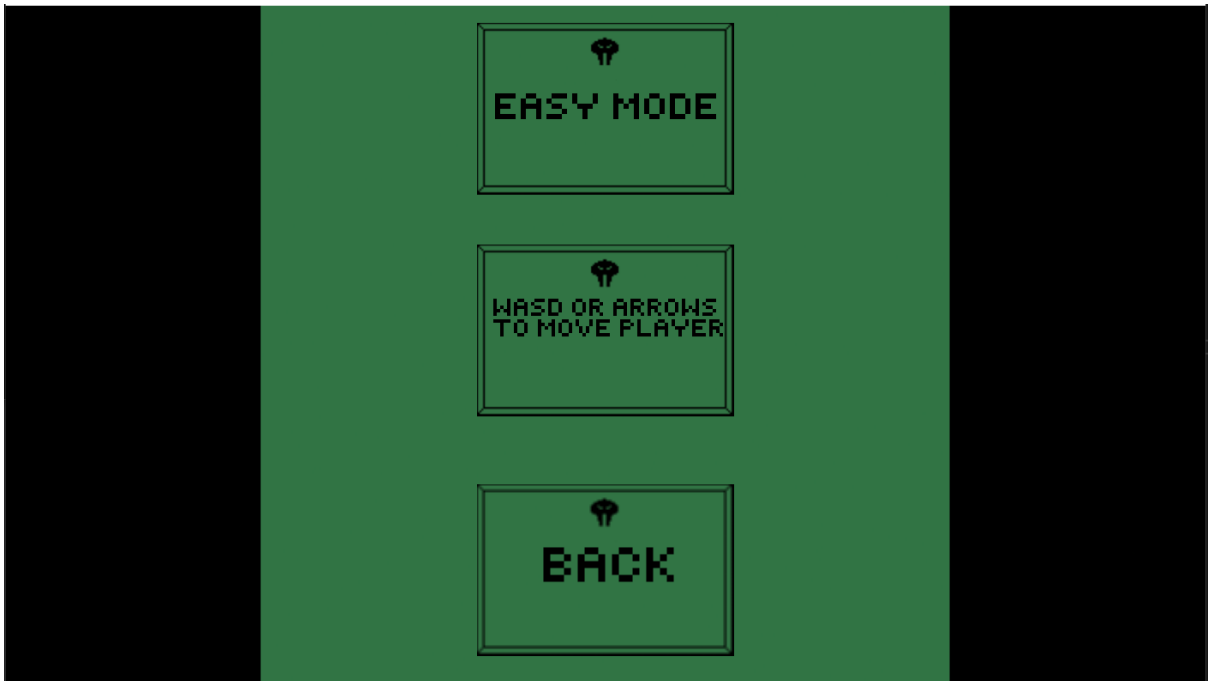
States can be tested by clicking quickly and changing between states in quick succession. This will show if the menu can be broken, by either having the change of states not working anymore or if the game will crash. This will likely be due to how the code works as it relies on use of a stack. If there is an error, I believe it will be due to the stack having all states popped on having multiple of the same scene being added to a stack.

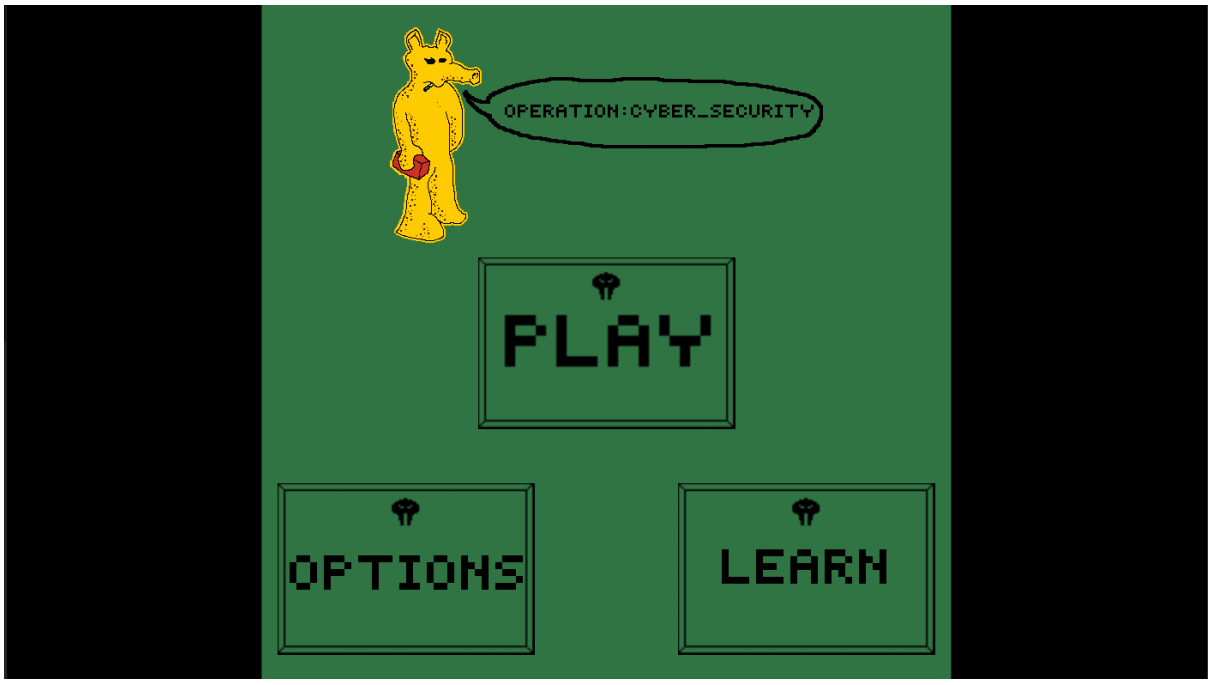
To make this process easier I will make the buttons be placed on the same position on window. This will decrease the time taken for the state to be transitioned.

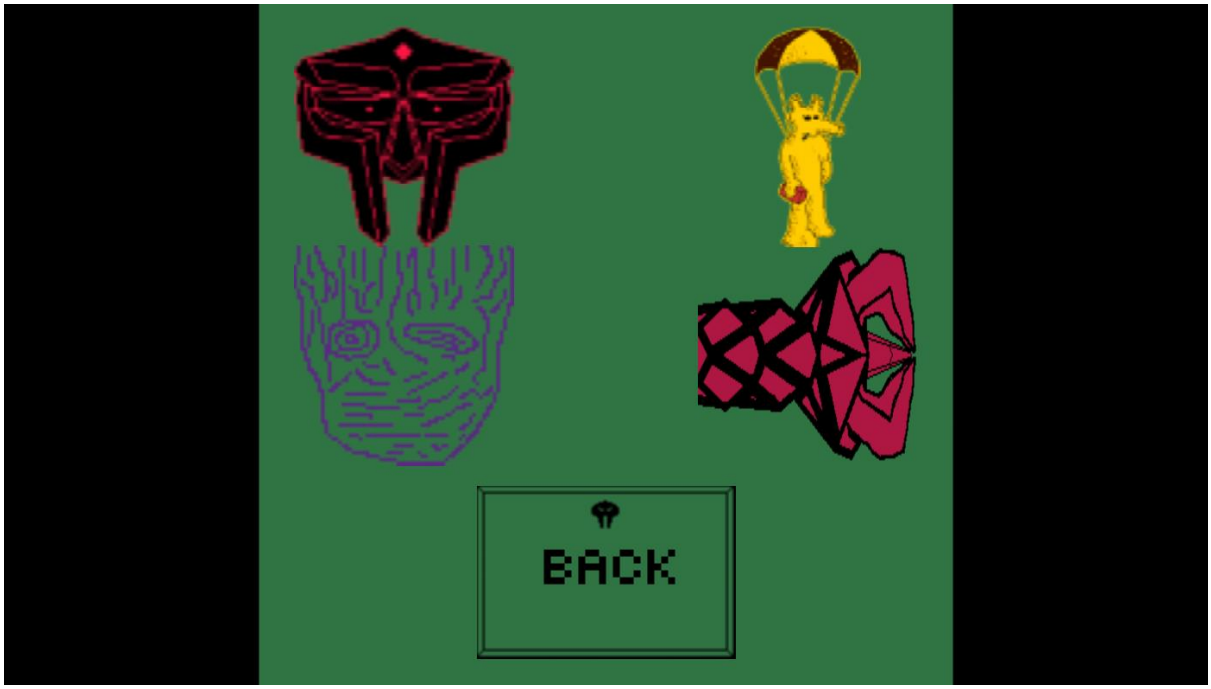
After some testing I have found that sometimes that when clicking the back button on the states that provide the education, the state changes to education then immediately change to the main menu. However, this was only when stress testing the menu. This will likely not happen when users are using the game. If it does, it will not cause the game to break, and is more of a slight nuisance. Therefore, it is not imperative to address this slight problem.

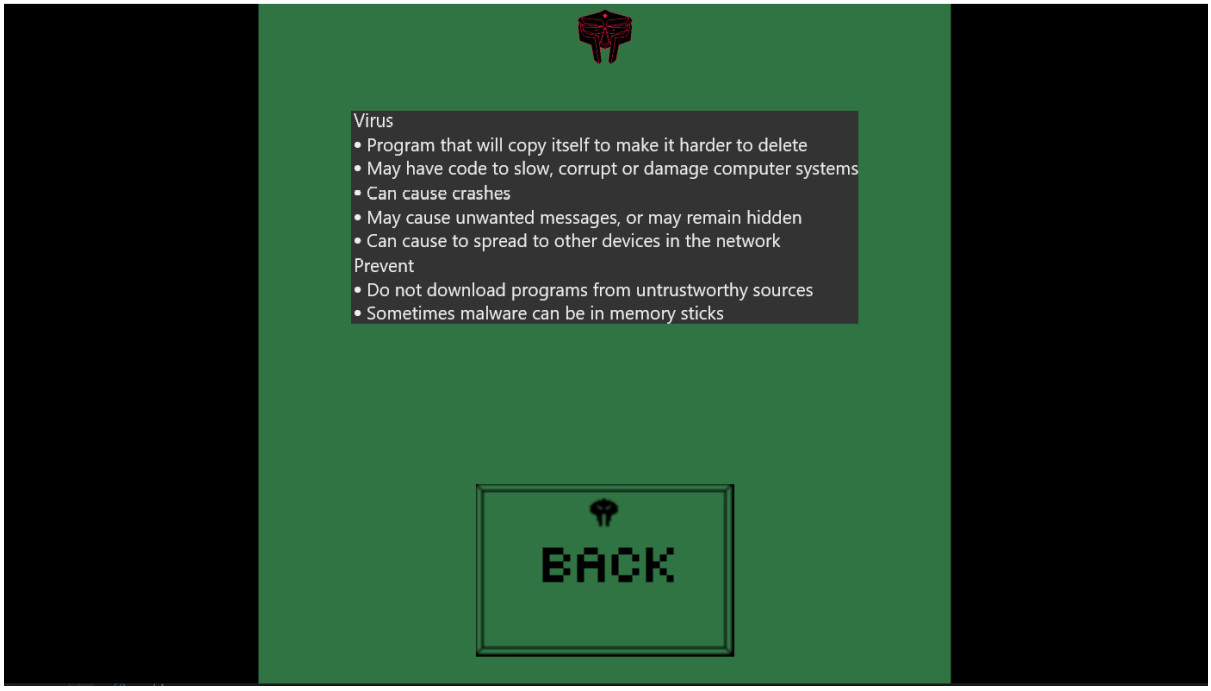
3.2.2 Function

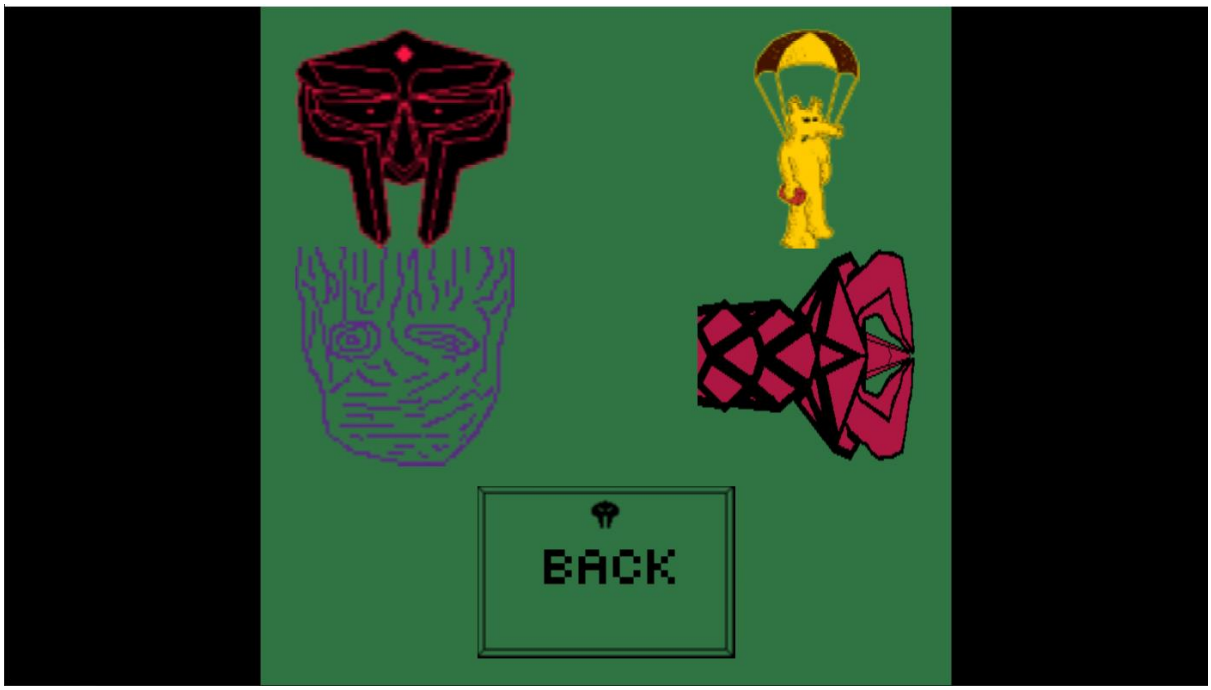


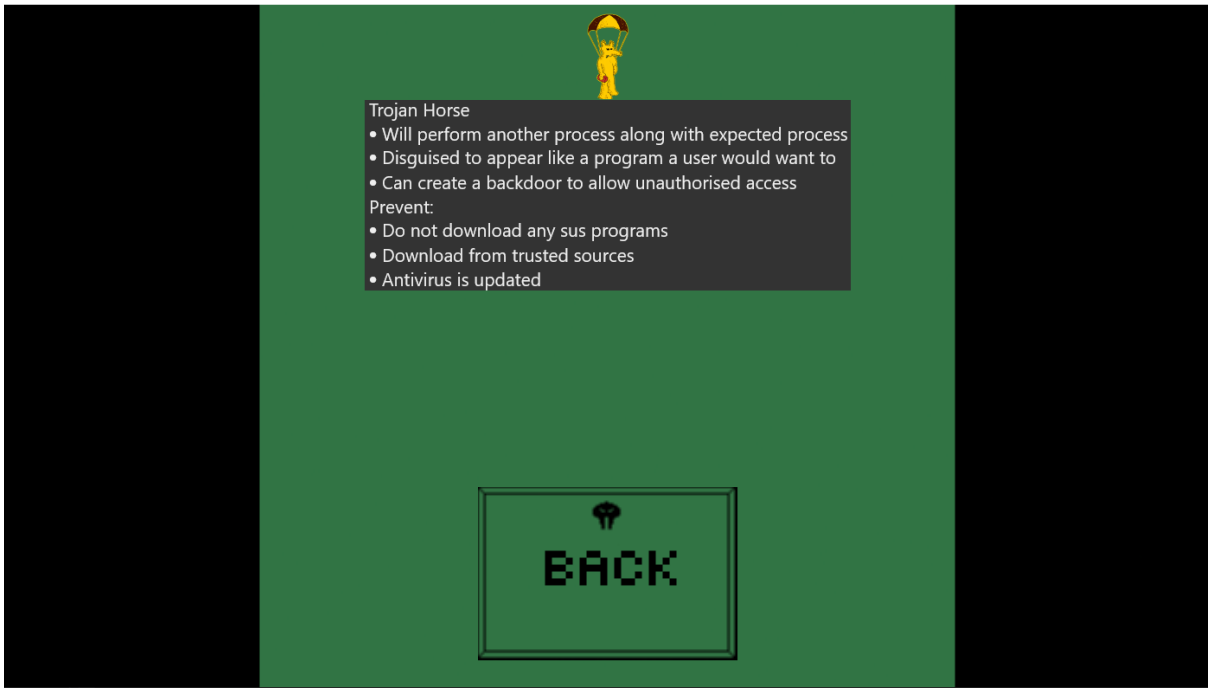

EASY MODE
WASD OR ARROWS TO MOVE PLAYER
BACK

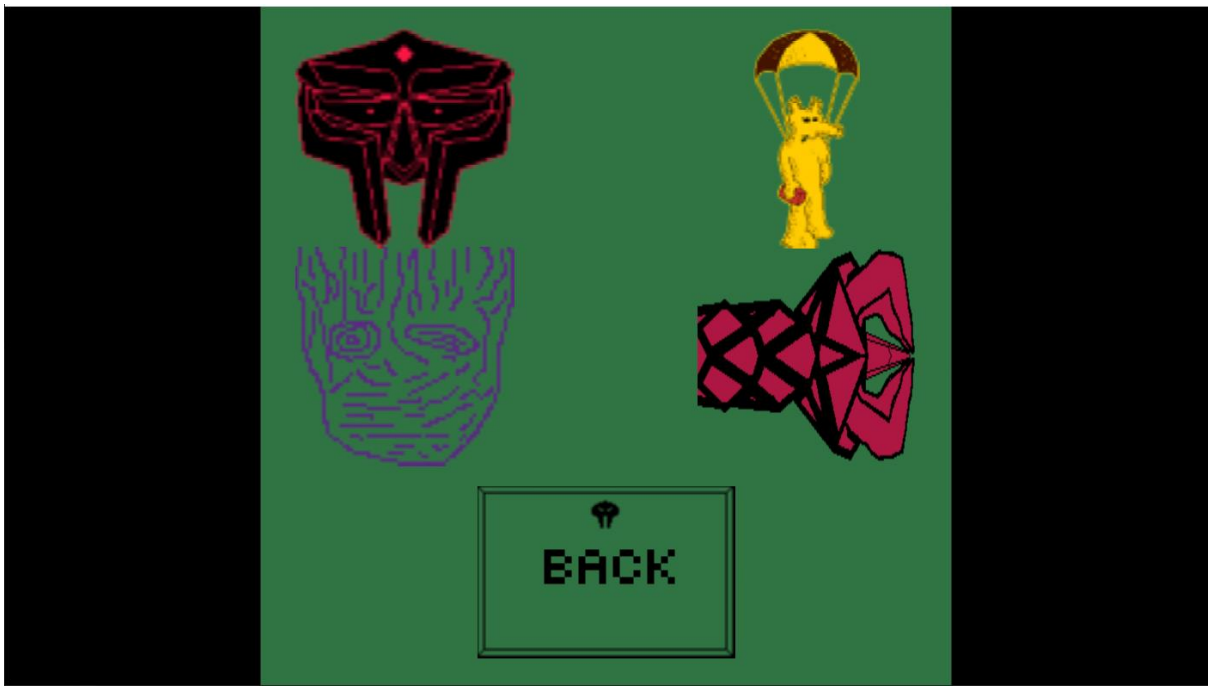

OPERATION:CYBER_SECURITY
PLAY
OPTIONS
LEARN

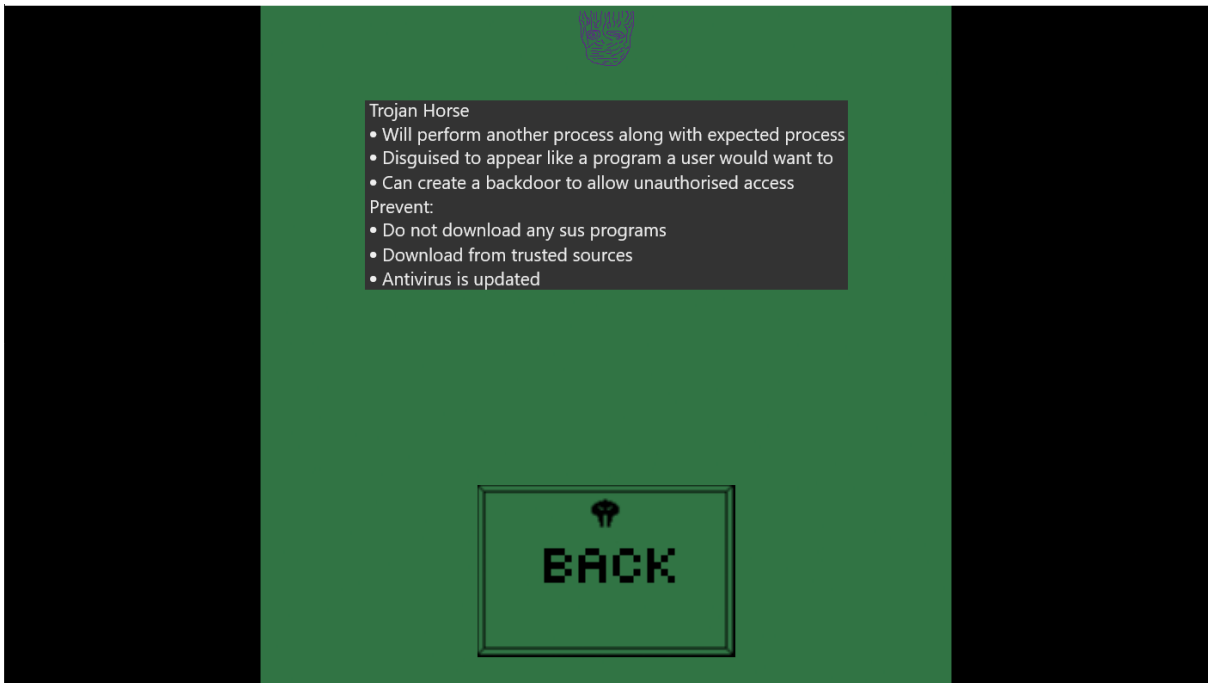

BACK

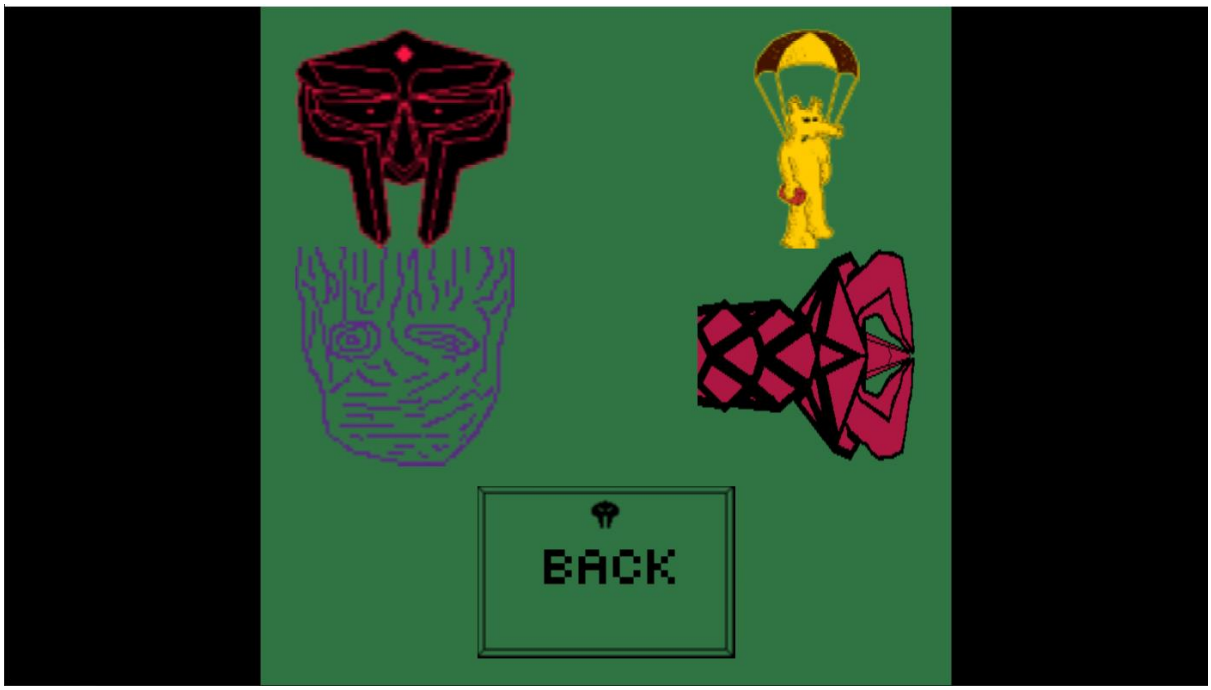

Virus
• Program that will copy itself to make it harder to delete
• May have code to slow, corrupt or damage computer systems
• Can cause crashes
• May cause unwanted messages, or may remain hidden
• Can cause to spread to other devices in the network
Prevent
• Do not download programs from untrustworthy sources
• Sometimes malware can be in memory sticks
BACK

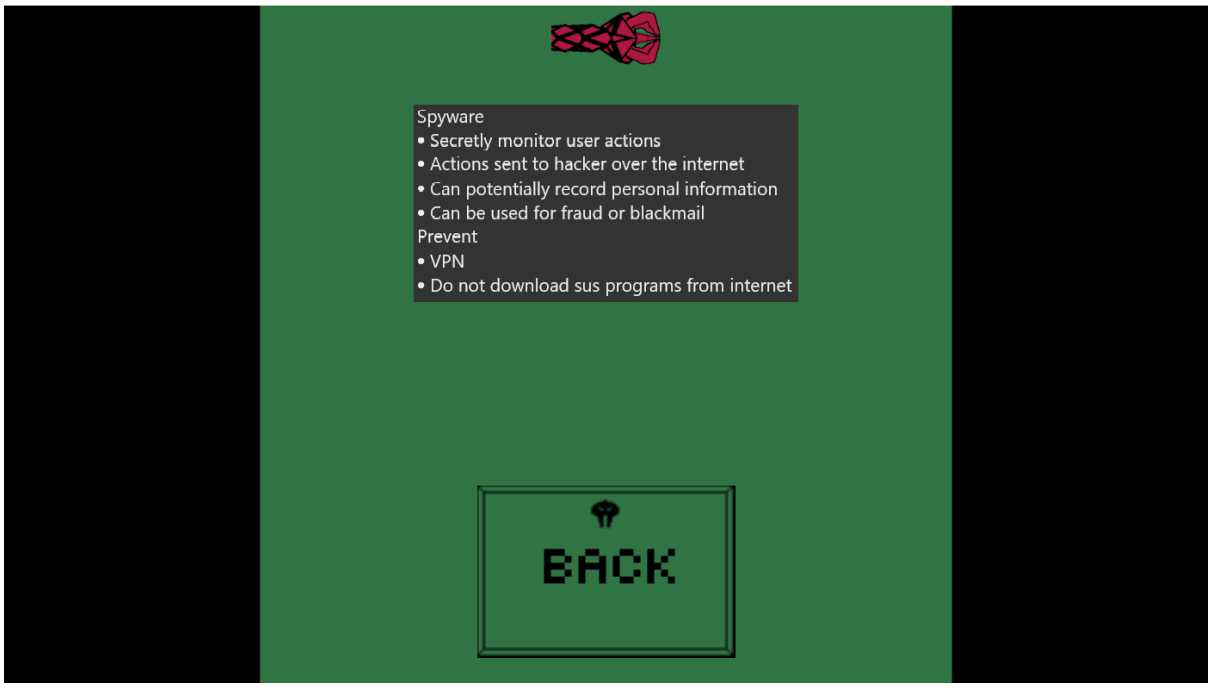

BACK

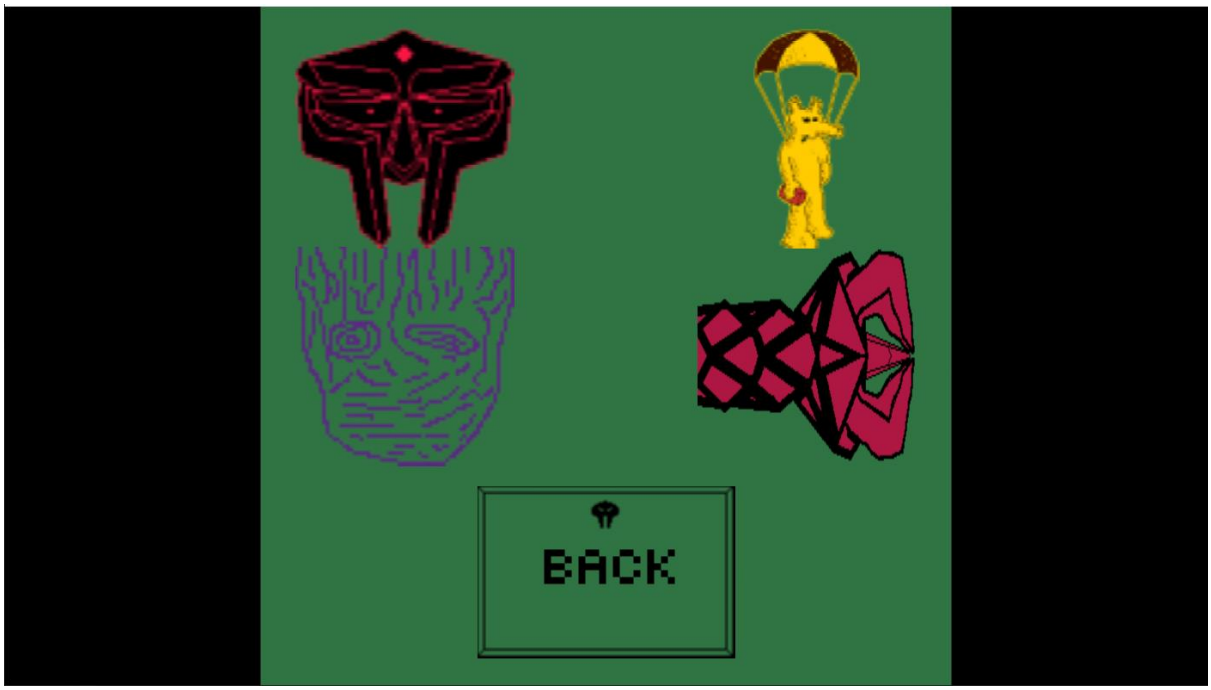

Trojan Horse
• Will perform another process along with expected process
• Disguised to appear like a program a user would want to
• Can create a backdoor to allow unauthorised access
Prevent:
• Do not download any sus programs
• Download from trusted sources
• Antivirus is updated
BACK

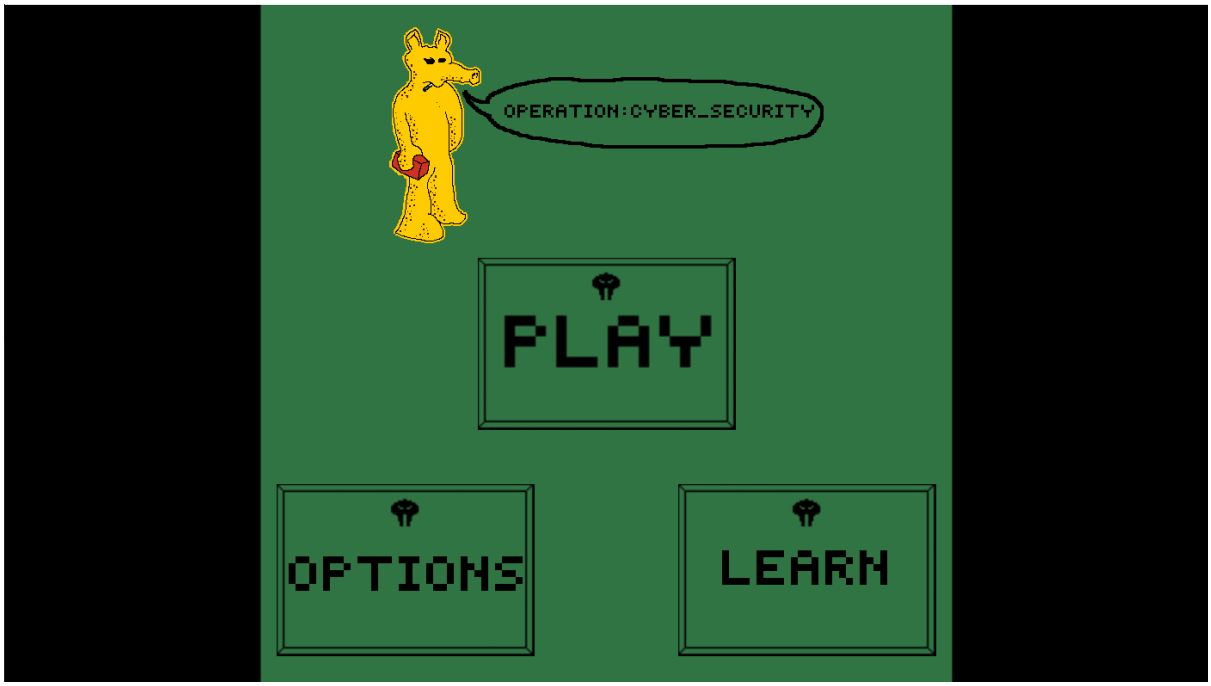

BACK

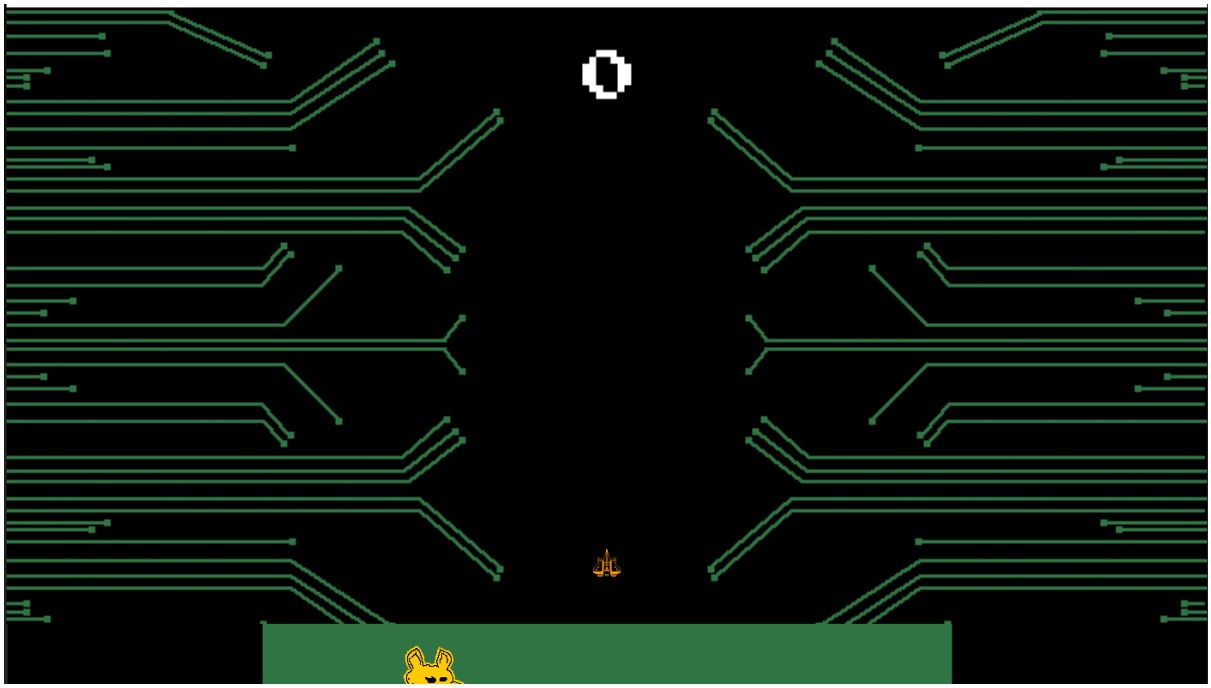

Trojan Horse
• Will perform another process along with expected process
• Disguised to appear like a program a user would want to
• Can create a backdoor to allow unauthorised access
Prevent:
• Do not download any sus programs
• Download from trusted sources
• Antivirus is updated
BACK

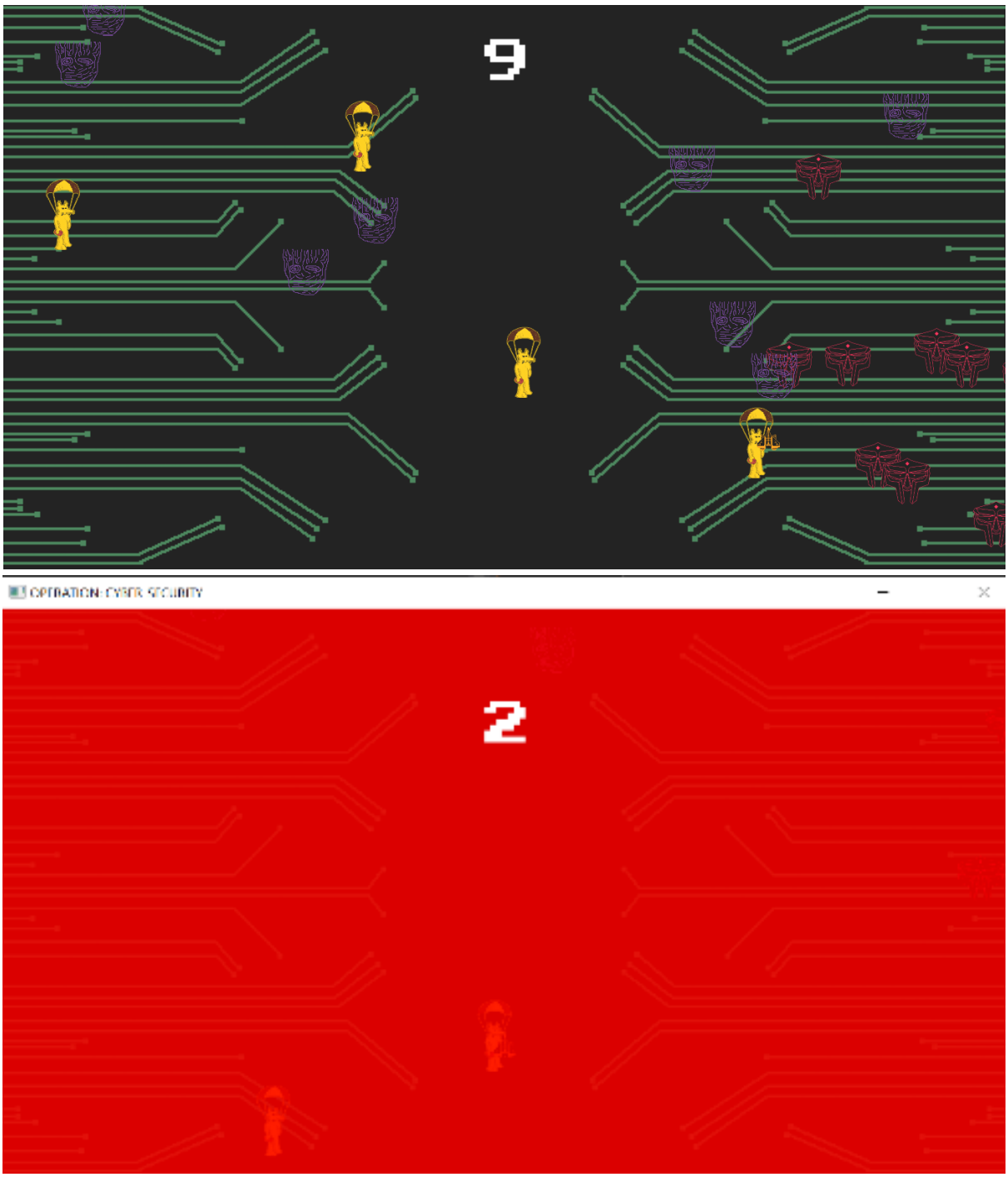

BACK


Spyware
• Secretly monitor user actions
• Actions sent to hacker over the internet
• Can potentially record personal information
• Can be used for fraud or blackmail
Prevent
• VPN
• Do not download sus programs from internet
BACK

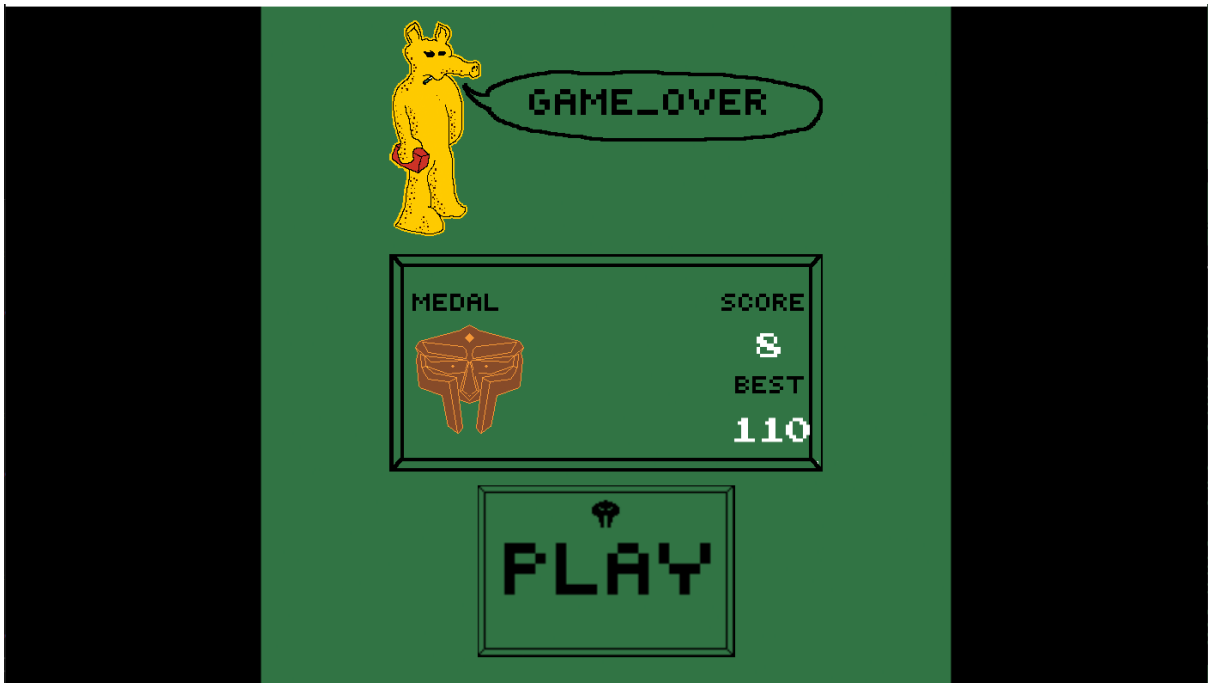

BACK

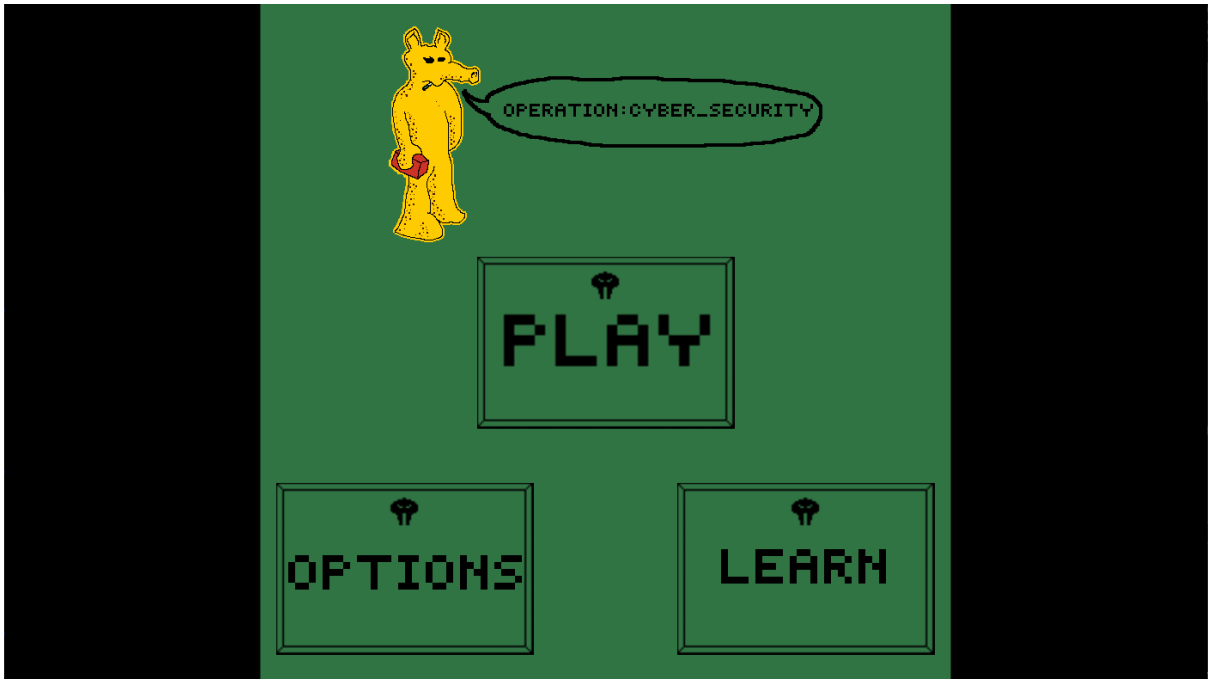

OPERATION:CYBER_SECURITY
PLAY
OPTIONS
LEARN





(red flash screen was taken from another attempt as it is difficult to screen shot)





3.2.3 Usability

The accessibility of this game is done through easy mode button located in the options. This allows accessibility to less skilled players or potentially a younger audience than what was intended.

The other accessibility feature is arrow keys for users who have problems with their left hands, making it difficult to use WASD keys.

To make it clear to user what keys can be used I have placed an image showing keys.

There are images to demonstrate these features effectively.

Final test

Name of Test	Data	Expectation	Reality
Key W	W key	The player's character moves up the screen Y value decreases	Pass – player sprite moves up the screen.
Key A	A key	The player's character moves left across the screen X value decreases	Pass – player sprite moves across the screen.
Key S	S key	The player's character moves down the screen Y value increases	Pass – player sprite moves down the screen
Key D	D key	The player's character moves right across the screen X value increases	Pass – player sprite moves across the screen
Splash State	State	When code is first loaded, the splash state should be added to stack. This will then display this state to the window. After short delay this state will be switched out to main menu state.	Pass – when code is ran, splash state is displayed. This shows image for a couple seconds before changing state
Main Menu State	State, button	This state will should load after splash state. This state will need to hold buttons that will be used to navigate to all other states	Pass – this state loads after splash state and holds buttons that can be pressed to change
Options state	State, button	This state should have a back button to return to menu and a form of option such as change in key binds or	Pass- this state will load the option state done by popping menu state and pushing the option state

		colour blind mode	
Education state	State, button	This state should offer an education and way to return to main menu state	Pass- this state will load the education state done by popping menu state and pushing the education state
Game State	State	This should include the actual game play. Enemies and data classes should be used here to create cohesive game. The game state should be popped and game over state should be pushed on player collision with enemy.	Pass – Game state will be loaded after pressing the play button. Main menu is popped, and game state is pushed.
Game over state	State, button, string, txt file	This should load after the game is over. This will give user option to return to the main menu, but also have their score outputted as a string and the high score to be outputted to the screen	Pass- game over state will be pushed after popping the game state on collision with enemy (plus short delay)
Play Button	Click button	Clicking on this sprite will pop main menu state and push game state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Options Button	Click button	Clicking on this sprite will pop main menu state and push option state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is

			clicked the play button will load the given command
Education button	Click button	Clicking on this sprite will pop main menu state and push education state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	Pass – when the player collides with an enemy game state is changed to game over. This cause a flash and then the user is taken to game over state. Does not allow for the player to input their name
Collision	Player position = Malware position and	If player position intersects with enemy position then the screen needs to flash. The game over state needs to pushed to top and previous state should be popped	Pass- player and enemies have rects drawn around them. When player rectangle intersects the one of the player them game is equal to over
Flash	rect	A translucent rectangle will be drawn to the screen on collision	Pass: A translucent rectangle will be drawn to the screen on collision
Score	Time, score	High score is generated based on time survived.	Pass: score is generated by having invisible sprite spawn and move across the window. On collision the sprite is erased

			and score increments
Options	Options button	Additional option such as option to change key bind or add colour blind mode. Should be found in options	Pass – a pass button created that can be clicked. When clicked the game is now easier
Education	Education sprite	A sprite of the education needs to be printed to the screen. This will sprite will have a texture of text with useful facts.	Pass – a sprite with texture of informative text is printed to the screen
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will have their score saved to text file. This text file will have to be open to retrieve the high score	Pass – Score is written in to the txt file. Previous score is erased
Enemies		Multiple enemies are created out of the window before moving into it. These enemies will have unique movement and collisions with player. Enemy should be deleted when out of the screen fully.	Pass- 4 different enemies are spawned in with different starting positions and different movement patterns
Player and enemy shooing	sprite	Enemies shoot and player shoot. This is done with shoot class that creates an object	This has not be done, as the shoot class did not work due to parameter problems with the state machine

Rate Prototype //Score: 0 out of 10						
User	Educational	Playability	Re-Playability	Graphics	Gameplay	Comment
Tom	6	8	7	8	6	
George	6	9	7	8	7	
Callum	4	9	8	8	7	
Drew	4	8	8	7	5	
Sum	20	34	30	31	25	
Average	5	8.5	7.5	7.75	6.25	

Final feedback from users indicate that all aspects that were being tested by the users were positive. This suggests that the end improvements have made everything at least acceptable for the end users. To improve I could consider some gameplay elements and a greater focus on education.

4. Evaluation

4.1 Evaluation Against Success Criteria

Success criteria	Measured	Justification
Simplicity	Challenges are simple. Use as few as buttons as possible. Reduce as much visual clutter as possible	Simple games are easier to understand and make it more accessible.
Accurate Education	The gameplay has educational elements like questions that have been properly researched and cross referenced.	The whole purpose of this project is to educate users. Therefore, this game must have a way of educating effectively. Additionally, If the game does not have accurate information for educating young adults, this project fails as it will spread misinformation.
Fun	Survey asking if the game is enjoyable to play.	If the game is not fun to play, users may not be fully engaged in the game. This will lead to reduced retention rate.
Engaging	Survey people Leader boards Quick and fun	More engaging games, as discussed above, improves the player's retention rate and will be the best way for them to be educated using this medium.
Accessible	The number of ways it must have ease of use	The game having accessibility features will help spread the information this game will have. This is because a wider audience will be able to play this game because it will cater for their needs.
Appropriateness	Survey key stakeholders	The game will have to use language and have art assets to be appropriate. Any controversial ideas will also be avoided. If the game is not deemed appropriate for schools or for young adults to play by schools or their parents, the game will struggle to be played as parent and teachers will be preventing the target audience from playing the game.

Informative	Having many questions that are used Having random facts drawn to the menu screen	Due to the nature of this project being educational at its core, it is important that there are several questions and facts constantly being shown to the users.
Goals / Challenges	Challenges to complete Leader board	This is an extension of engaging and can be achieved just through use of leader board. However, by including some challenges or overarching goal / challenge, users may become motivated to play for longer.
Gameplay	Survey from play testers	The quality of gameplay is paramount to the project's success. If the game lacks responsive and quality controls, then users will not use this project to learn from.

Simplicity - Met (Page Ref: 135 (Simple user inputs))

I made the game simple by having the player dodge the enemies when they spawn in. this require only a few inputs such wasd or arrow keys. This is an infinite game model meaning no other input ect is needed. From user survey, no user struggled with playing the game or struggled to grasp the concept of the game. By having the game be simple, the total possible amount of players have been increased as difficult / complex games have a smaller audience. There was also an inclusion of easy mode within the game, catering for those who found the difficulty curve of the game to be too punishing.

The simplicity of the game has some limiting factors as well as I have drastically oversimplified the original vision of my game. By making the game more complex such as having shooting would have made the game more appealing and engaging. Further iterations would have allowed me to attain my original vision of having shooting and power ups.

Education – Met (Page Ref: 166 / 178 (Educational text states))

Education in my game has improved within the final iteration and testing of the game. By having an accurate description of each enemy as a type of online threat should hopefully remind the user of facts when they see them. However this is reliant on users seeking the education out of their own accord. Some feedback is that some users will not read the education, and just play the game. Since the gameplay has lacklustre education qualities, the game will struggle to provide a solution to the initial problem. Presenting information in this way is not interesting or engaging. I believe that this criteria was partially met, but if I continued with this game, I would have likely implemented a quiz or some sort of education in the gameplay.

Fun – Met (Page Ref: 146 (Enemies))

Users who played this game all thought that the game was fun. This means that the target audience should also find the game to be fun. The game included enemies that was fun to dodge as they had more accurate collisions. A problem included was that the game had a slower start and can be boring as the game speeds up.

Engaging – Met (Page Ref: 146 (Enemies))

The game is engaging as it provides a high score saver and medals. Users surveyed found this system to be interesting and engaging as they continued to play to try and get the highest medal possible. The game continued to be engaging until this medal was reached, where engagement would dwindle. After some gameplay most users were satisfied and would play again, hence I believe I succeeded in this criteria

Accessibility – Partially Met (Page Ref: 176 (Other key binds for movement))

To make my game more appealing to more people, I included an easy mode and bound arrow keys and wasd to movement system. This will maximise more people who can play the game as some people may struggle with different controls. This could have been fleshed out more and included more options such as colour-blind mode or larger text options.

Appropriate – Met (Page Ref: 182 (Function shows all sprites))

The game did not show any inappropriate images or had any inappropriate information on the screen. Due to online threats and malware not having an actual image I had to attempt to draw a visual representation of them, which I believed was successful. The scoreboard had no way to save names as this game will be played solo and won't have an online leader board. There is no way to have anything inappropriate unless the source code is edited. The game is therefore appropriate enough for parents and teachers to encourage the use of it

Informative – Met (Page Ref: 166 / 178 (Educational text states))

The game provided information about the types of malware and threats in a detailed but brief way. In addition to these preventive measures are also described. This will mean that this is successful in providing informative description of 4 threats. For the scope of this game I think I succeeded in this, however if the scope of the game was larger I would have included more information on more threats.

Goal/Challenges – Partially Met (Page Ref: 163 (Medals))

This was partially met as I have inclusion of the medal system in my game. This provides challenges to the user to try and get the highest medal possible. This would improve engagement and replayability. However, this could have been expanded on as this could've included features such as a challenge list for the user to complete.

Gameplay – Partially Met (Page Ref: 152 (Player dodging enemies, game loop))

I think this was a success. Despite not achieving my original vision, I have included a player which is fun to move and enjoyable enemy movement. There is also fair collision system; this will make the game less frustrating. However, currently it is boring after many runs, this could've been improved by adding more features such as health and shooting.

Evaluation against testing

Name of Test	Data	Expectation	Reality
Key W	W key	The player's character moves up the screen Y value decreases	Pass – player sprite moves up the screen.
Key A	A key	The player's character moves left across the screen X value decreases	Pass – player sprite moves across the screen.
Key S	S key	The player's character moves down the screen Y value increases	Pass – player sprite moves down the screen
Key D	D key	The player's character moves right across the screen X value increases	Pass – player sprite moves across the screen
Splash State	State	When code is first loaded, the splash state should be added to stack. This will then display this state to the window. After short delay this state will be switched out to main menu state.	Pass – when code is ran, splash state is displayed. This shows image for a couple seconds before changing state
Main Menu State	State, button	This state will should load after splash state. This state will need to hold buttons that will be used to navigate to all other states	Pass – this state loads after splash state and holds buttons that can be pressed to change
Options state	State, button	This state should have a back button to return to menu and a form of option such as change in key binds or	Pass- this state will load the option state done by popping menu state and pushing the option state

		colour blind mode	
Education state	State, button	This state should offer an education and way to return to main menu state	Pass- this state will load the education state done by popping menu state and pushing the education state
Game State	State	This should include the actual game play. Enemies and data classes should be used here to create cohesive game. The game state should be popped and game over state should be pushed on player collision with enemy.	Pass – Game state will be loaded after pressing the play button. Main menu is popped, and game state is pushed.
Game over state	State, button, string, txt file	This should load after the game is over. This will give user option to return to the main menu, but also have their score outputted as a string and the high score to be outputted to the screen	Pass- game over state will be pushed after popping the game state on collision with enemy (plus short delay)
Play Button	Click button	Clicking on this sprite will pop main menu state and push game state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Options Button	Click button	Clicking on this sprite will pop main menu state and push option state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is

			clicked the play button will load the given command
Education button	Click button	Clicking on this sprite will pop main menu state and push education state	Pass - A sprite is draw to screen. When the mouse is intersects the sprites position and left click is clicked the play button will load the given command
Level end	High score, new level	Running is set to false. Score is set and allows player to type their name. Gives player option to go to main menu or play again	Pass – when the player collides with an enemy game state is changed to game over. This cause a flash and then the user is taken to game over state. Does not allow for player input of name
Collision	Player position = Malware position and	If player position intersects with enemy position then the screen needs to flash. The game over state needs to pushed to top and previous state should be popped	Pass- player and enemies have rects drawn around them. When player rectangle intersects the one of the player them game is equal to over
Flash	rect	A translucent rectangle will be drawn to the screen on collision	Pass: A translucent rectangle will be drawn to the screen on collision
Score	Time, score	High score is generated based on time survived.	Pass: score is generated by having invisible sprite spawn and move across the window. On collision the sprite is erased and score increments

Options	Options button	Additional option such as option to change key bind or add colour blind mode. Should be found in options	Pass – a pass button created that can be clicked.
Education	Education sprite	A sprite of the education needs to be printed to the screen. This will sprite will have a texture of text with useful facts.	Pass – a sprite with texture of informative text is printed to the screen
Save High Score	.txt document with CSV on scores and names	At the end of a game, the user will have their score saved to text file. This text file will have to be open to retrieve the high score	Pass – Score is written in to the txt file. Previous score is erased
Enemies		Multiple enemies are created out of the window before moving into it. These enemies will have unique movement and collisions with player. Enemy should be deleted when out of the screen fully.	Pass- 4 different enemies are spawned in with different starting positions and different movement patterns
Player and enemy shooing	sprite	Enemies shoot and player shoot. This is done with shoot class that creates an object	This has not be done, as the shoot class did not work due to parameter problems with the state machine

WASD movement – Met (Page Ref:135)

The player movement of wasd effectively moved the player at a constant speed in all possible 8 direction. These keys can be held down to constantly move in a direction. In addition to these keys I have included binds to arrow keys. I made sure to use a xor key a preventative measure to stop both wasd and arrow keys to stop the speed being higher than what it should have been. I could have potentially tested and attempted movement with

joysticks on a controller as SFML accommodates for this. This would have needed more coding and validation, but this would also improve accessibility.

Splash State – Met (Page Ref: 121)

When game is originally loaded a splash screen is loaded. This provides very little functionality to the game and only makes the game appear more professional. This will work consistently when loaded up.

Main Menu State – Met (Page Ref: 124)

The Menu splash state is successfully done as it acts as a hub. This state will allow access to all other states from here. It is organised in a logical and clear way that is intuitive to users to understand use. It will always be pushed to the stack when needed.

Options State– Met (Page Ref: 166)

This option state will load an easy mode button, which change the speed of the enemies as well as informing the users on the key binds. This will clearly help users to quickly understand the game before games are played. To get to the options state, it is clearly found from the menu. The back button will return to the user to the main menu state.

Education State – Met (Page Ref: 164)

Originally there was only going to be one education state, but after feedback this is now 5 states: hub to reach the states and then the four education states. In terms of the state machine this works well. The only issues occur when stress tested and buttons are clicked in quick succession. This will cause the menu to be loaded straight away.

Game State – Met (Page Ref: 130)

This state will be loaded when the play button is clicked. This will successfully load the game state which will load the player and the enemy objects. Additional sprites also load successfully such as the background and the score. Score will continue to increase until collision with enemy. Enemy collision will cause the game to end and will load the next state.

Game end State – Met (Page Ref: 158)

This state is successfully loaded with player/enemy collisions after a short delay. This state successfully contains the users score, users high score (with use of a high score text file that appends when score is larger than high score) and a medal. The user's medal is dependent on their score and consistently draws the correct colour of medal when the game ends. The user has the option to return to the main menu from this state.

Play Button – Met (Page Ref: 125)

A button is successfully drawn to the screen. It will check for the position of the mouse relative to the screen. If mouse position is over button and mouse is left clicked, the if branching statement will cause the code to run. In this case the state machine will push the game state on to the top of the stack.

Options Button – Met (Page Ref: 166)

This works the same as the play button. When this button is clicked, the options state is pushed to the top of the stack. This will have the options state drawn to the screen instead of the menu state. This allows the user to interact with the functions present on this state.

Education Button – Met (Page Ref: 164)

This will push the education state to the top of the stack. Originally this was only going to have a back button, but now it has four more of these buttons to the new states that have the education present on them.

Game End – Partially Met (Page Ref: 151 (collision end the game))

This works by having collisions causing a flash, screen to stop updating and after short delay, state is pushed to the top of the stack. The game will always end when there is a collision. There were no bugs that I could find that caused the game to go on indefinitely.

However this was only partially met as the player was originally going to be able to enter their name at the end of the game.

Collision – Met (Page Ref: 130 (collisions with window border) / 151 (collisions with enemies))

Collisions between players and enemies are more consistent because I have made it so that players and enemies rectangles are closer to the centre of their sprites. This allows for more consistent gameplay as the player will not collide with an enemy when from the player perspective they were not close. This will make the game easier and more enjoyable to play.

Collisions between player and window are also working well. This does not work the same as with enemies, as the player regular rectangle is used for these collisions. This means that the player's sprite edges will not be able to leave the window of the screen.

Flash – Met (Page Ref: 143)

The flash was implemented successfully as it would clearly telegraph to the user that a collision has occurred and that the game has ended. This flash was brief but noticeable to the player. I was careful to make sure that the flash would be suitable so that people who suffer from epilepsy would be able to play the game. This was considered by the test users to be suitable and effective to let the user know that the game was over.

Score – Met (Page Ref: 179)

The score works well as it will continually update. The score will clearly show how well the player is doing. These scores will encourage the player to keep replaying. The score was recently changed to be a more efficient and effective, as using time instead of collisions. Score is an important part of the game as it acts as an indicator for the player's success. Score will be displayed in white text in the middle top of the screen so that it is clearly visible to the player. The score will constantly update every time the clock reaches the necessary value.

Whenever a new high score is reached the player the score will append the previous score held in the text file. This will then be written to the screen when the game ends in the game end state.

Options – Met (Page Ref: 166)

The options in the menu were partially successful as it can be loaded from the main menu. It will also change how the game works as the easy mode will make the game enemies slower. This would be better if there were more options within this menu. This will allow for more accessibility and provide for more usability. This would have included volume options, text size option ect. This will make the game more usable and would make the game more enjoyable.

Education – Met (Page Ref: 164)

This was met as there is a variety of information that can be learnt from this game. This would be educational to the user as they can learn about threats and how to avoid them. There are still some concerns with the education, as there is a lack of implementation in the gameplay. This means there is potential for users not to learn anything from the game as they can easily avoid the education part of the game. It is up to the user to decide on how much they want to be educated and is up to them to learn from this section. The education can act as flashcards from the way it has been set up, which is an effective revision method, but there is nothing indicating this. Therefore, it is up to the user to think of this, which many will not do.

Saving high score – Met (Page Ref: 160)

This was met as whenever the user gets a new high score it will be written into the text document that is saved within the game files. This score will append the text document, deleting the previous score. This will leave the text file containing the highest score attained. There is no player input option within this game, meaning there is no way that their score can be altered within the game. However, the user could potentially go to text document and could enter a fabricated score from there. This would not matter much though as there is no online feature, which will ruin other player's experiences.

Enemies – Met (Page Ref: 146)

Enemies were completed successful as different objects of this class are created in varying positions. There are 4 different types of enemies in this game, 3 of which spawn frequently. The enemies have unique movement and different speeds requiring users to be engaged as they play the game. Having a variety of enemies also make the gameplay more interesting because their designs are unique. The enemies also have a hit box that will end the game with collision with the enemy.

Player and enemy shooting – Not Met in final / Met in earlier prototype (Page Ref: 63)

This was not met in the final iteration of the game. In an early prototype of my game this was achieved and on collision with shoot sprite, health would decrease. But even with this in the early prototype the code was not efficient approximately being 200 lines in the game.cpp file.

This was problematic as it would increase by about 100 lines with each new enemy. Another problem was that they used nested loops, which made the time complexity $O(n^2)$. This dramatically effects the execution time.

The shooting code that originally used was not compatible with the final iteration of the game because the original had parameters that would not be taken by the state machine. Due to my limited knowledge and time scope, I decided I would just try to make the best out of it and made changes to the enemies to compensate for the lack of shooting.

I recognise that this change made it so that the original vision for my game was not met, however I believe the game made was still enjoyable and is more accessible. I also believe the shooting mechanic was worth losing for a good menu system that allows for easy traversal between states.

4.2 Further Development

Education

My original idea for education could not be achieved due to the limited time for the completion of my game. Earlier, before the coding began, I had plans for there to be power-ups that when collided with the player would be given a question. If was correctly answered, the power up was awarded. If my game was further developed, I believe I would attempt to add this into the game.

This could have been added by having new files: powerUp.hpp and powerUp.cpp. These files would have a class which cause objects to spawn on the screen. On collisions the game would pause as a quiz state would be pushed to the stack. This state will then be popped and if the answer was correct the power up would be applied to the player for a limited time. This would make the game easier and more enjoyable for less skilled people. This would lead to more people being able to benefit from my project

Not including this has limited my project as users can currently ignore the educational elements and therefore not gain any of the intended use of my project. With the appropriate changes, this would no longer be an issue

Accessibility

The game only has few accessibility features such as alternate controls, easy mode and colours that can be seen easily by many types of colour blindness. However, there are many other types features that could have been implemented such as bigger text option and mute options to stop the music from playing.

These could have a toggle button in the options state. Due to the options being possibly quite expansive, I could then have experimented with a method that would allowed the user to scroll up and down the screen.

Accessibility and options was an oversight throughout my project. After finishing my project I have a better understanding of the importance of this area. I therefore wished that I spent a

cycle better developing this as my game lacked important features most games have to maximise usability.

By lacking accessibility and usability features as described above, my project has limited amount of people it can be used by. By making necessary changes, there will be more potential users and therefore more users.

Gameplay loop

Gameplay loop was always going to an infinite loop model. However, to make the game more interesting I could have added in more boss like enemies like how the arm enemy only occasionally spawns. These boss enemies would have stayed on the screen and used some sort of path finding algorithm to track the player (possibly a* pathfinding) for a few seconds before leaving the window and being deleted. These enemies would have had their own class but inherited methods from enemy class. This would have been an extra addition I believed would have added a lot of value to the game.

Currently the game follows the loop of:

Spawn -> Dodge enemies -> Collide -> Game Over

If there was more time I would have liked to have made the game loop:

Spawn -> Dodge enemies -> Boss -> Collide -> Quiz (if answered correctly given second life) -> Collide -> Game Over

This game loop would provide more interesting game play as games would last longer and be more difficult after surviving longer. To make this game play more enjoyable I would consider the inclusion of lives similar to an earlier prototype of the game for the player.

How the project is currently there are limitations such as the amount of people of who will replay the game. As it currently is replayability is not much of a concern as limited education can be gained through gameplay, however I am thinking of this with the idea of previously mentioned changes having already be made. Therefore replayability will be more important and the appropriate changes will be made.

Conclusion

I believe that my stakeholder were satisfied with the results of my solution and will would be interested in using my solution.

To conclude, I believe that with additional time spent on my project I could have achieved the vision I had for it. These features would have been developed and result in a more complete and usable solution for the user to resolve their problem of lacking basic computer safety and cyber security knowledge. With inclusion of better integrated education and a larger selection of accessibility options my game would be effective in educating the most people possible.