C++ Programming Course Project Report
Dungeon Crawler, team 4

Joel Sågfors
Johan Rabb
Jon Galàn
Antti Matikainen

Foreword

Our group was assigned the topic "Dungeon Crawler" for our project. In its plainest form, we were to create a simple dungeon crawler game using c++ and the SFML graphics and audio library. The examples given to us for use as inspiration were the original Diablo, Gauntlet, Hammerwatch as well as Rogue and the many games it inspired. For our project, we took inspiration from Gauntlet as well as several games from the Roguelite genre.

At the most basic level, our game was to include:

- Working gameplay as well as a combat system.
- Simple 2d graphics, including a graphical user interface.
- Multiple types of weapons and monster.
- Randomly generated maps for the player to explore.
- A system for storing as well as using assorted items. (Essentially, an inventory system.)

In addition to this, some other features were suggested, such as:

- Non-player characters such as NPCs.
- Randomly generated weapons and monsters.
- Crafting and modifying items.
- Quests or events.
- Advanced user interface, such as an inventory browser or map.
- Sound effects.

As soon as the groups were announced, we began work on our plan. After viewing the videos given for inspiration and discussing our options, we chose to make a real time dungeon crawler in the vein of Gauntlet. We listed a number of features we were intending to implement and how we would accomplish them. These served as the start of the skeleton of our game. We also made the decision to use Slack as our communication method, and Trello for keeping track of schedules and assigning tasks.

- Realtime gameplay.
- Simple top-down graphics with sprites.
- Attacking objects with close range melee attacks or ranged projectiles.
- Collision detection and randomly generated maps.
- A graphical player inventory, as well as weapon switching.
- Keyboard and mouse support.

In practice, we ended up accomplishing these things and more. Not only did we manage random map, monster and item generation, we also included graphics we ourselves made. Sound effects were also implemented, although for them we used samples released under the Creative Commons license.

Architecture overview

Our project uses a fairly straight forward object oriented structure, with the exception of textures and sound buffers which are stored under global variables. The solutions we came to use were ones that were developed via testing and group discussion about how to achieve our goals. Further information about the classes can be found in the doxygen documentation. In our project, we have the following classes:

The Map class which is used as a general "base" class. The Map class is fairly simple, containing a vector of structures representing all the rooms the game map contains. The Map class then generates the rooms based on these structures, either from file or through random generation, depending on the room type. It also contains a simple variable which stores the index of the room the player is currently in, as well as the Room object corresponding to that room. It also holds a pointer to the player character. The Map class only has a few functions. A function for returning the current room, another for moving between rooms (which generates a new room and discards the previous one), and a constructor that creates the game map.

Another class that is fundamental to the functionality of the game is the Room class. It contains all the information for a singular room, as well as a fair deal of functionality. Much like the map, it stores a pointer to the player character object. It also stores information such as its width and height, information on the individual tiles in the room stored in a two-dimensional vector pair. The Room class also stores information such as every monster, NPC, projectile or item in that room. It also stores all the sprites that are in the room as well as if they are in use.

In terms of functions, the Room class contains the functions to draw the contents of the room, including all the monsters, items, npcs and projectiles. It also has functions to assign the character pointer and add objects of the aforementioned types. There are also functions to return information about specific tiles in the room as well as geometrical convenience functions such as accessing all the neighbour tiles of a specific tile or querying whether a given pair of coordinates is within bounds or not. This information is used for collision detection among other things.

The Tile class is the last class used in implementing the map. It is a rather simple class, containing simply the position in the room, the material, if it is penetrable, the sprite and the index of the texture it uses on the global vector. It also has a function that draws the tile (using said sprite and texture index information) as well as another that tells if the player, a monster or a projectile can pass through the tile.

While the Map, Room and Tile classes together make up the play area, the Character class contains all the necessary information and functionality for the player character, barring weapons and items. It is used to store a plethora of information. For use with the inventory system, it holds a vector of pointers to the items in the player's inventory, as well as pointers to the player's equipped ranged and melee weapons. The reason we chose to employ a vector of pointers was to accommodate possible future polymorphism in the item class. It also stores integers that handle the player's health, level and experience.

To accommodate graphics and sound, the player class stores an integer that stores the current orientation of the character. It also stores the character's sprite, texture and the texture for the shadow under the character. To use in sounds, it holds two Sound objects that sound buffers can be connected to as required. A pointer to the Room class is held as it is required for ranged attack functionality.

The Character class also has a wide variety of functions associated with it. There are functions for the basics such as modifying or retrieving variables. There are functions for adding or removing items from the inventory. Ones for drawing the character and ones for moving their position and setting their rotation.

Items and their functionality is handled by the Item class. It is a fairly simple class without too much stored information. It stores the name of the item, what type of item it is (gold, a weapon or a consumable), the sprite for the item, its position, if it is active on the ground, the level of the item, if it is sellable, its price, and in the case of items representing weapons, a pointer to the Weapon it represents.

The Item functions are primarily ones required for their functionality. Aside from the function that draws the item on the ground and handles picking it up when the player walks over it, there are functions for accessing the variables. There are also several functions that are called when the item is used. These handle things like calling the appropriate Character function. There are also multiple constructors. One of these is for a pre-defined item, another for a pre-defined item representing a weapon, and the last is one for a randomly generated item.

The many weapons the player can use are handled by the Weapon class. The weapon class is split into two sub-classes, one for melee and the other for ranged weapons. While they share many variables, there are a few variables only used by one or the other. The weapon's variables store their name, an integer that denotes the type of the weapon, the damage of the weapon, as well as the radius it deals damage in. It also stores an integer that denotes the index of the weapon texture (and in the case of ranged weapons, another for the projectile texture) in the texture vector. Another pair of floating point numbers stores the weapons attack cooldown as well as the speed of the potential projectiles created by ranged weapons. Last but not least, it also holds an integer that saves the level of the weapon. The majority of the weapon class functions are ones that return variables, with the exception of its two constructors. One constructor creates a weapon with set attributes, the other creates a random weapon.

In addition to melee attacks, the character can use projectiles to take on the hordes of monsters. Projectile functionality is taken care of by the titular class. To achieve the functionality required for projectiles. The projectile class stores the Room the projectile is in, if the projectile is fired by a player or a monster. It also has a boolean value for if the projectile is active or has hit a wall. A pair of vectors are used for the position and direction of the projectile. A floating point number is used for storings its speed, while integers hold the damage it deals and the radius of its hitcircle. The class also holds the index of its texture, the texture object and its own sprite.

Aside from the constructor and functions that return or set its various variables, there is a function to draw and update the projectile's position. Another function toggles the boolean value that handles if the projectile is active, to be called when the projectile hits another object. To avoid constantly creating new projectiles, a function can be used to reset them with a new set of attributes at a new location. This is done for performance reason to minimize the amount of projectiles that have to be created. Using this system, projectiles are looped as the projectile vector is used as a buffer that is only expanded when all its projectiles are active.

The many foes the character will face in the dungeon are handled by the Monster class. The base Monster class is abstract, and is just used to define basic variables and functions that the Ranged Monster and Melee Monster subclasses share. The monster classes shared variables store its name, alongside various attributes such as health, experience, movement speed, the position and other assorted variables. The monsters also store its sprite, its sound instance and whether the monster is active and to be drawn or not.

Each Monster subclass has two constructors. The first is for monsters with manually set attributes. The second constructor creates a randomized instance. The randomized monster constructor is used for making many of the monsters found within the game. Aside from functions that return or change the various variables a monster has, there is a fairly length function that handles both the artificial intelligence as well as the drawing of the monster. Other functions handle things such as the monster's attacks or utility functionality such as checking the distance from the player to the monster. Many of the functions are used in support of the monster AI functionality.

There are also friendly denizens of the dungeon. They are represented by the NPC class. NPCs are one of the last things added to the game, and it is not in active use in the current version of the game. Its logic is there to allow for future expansion, and Shopkeepers (which is a subclass of the NPC class) can be encountered and interacted with even in the current version, but there is no logic yet to vary their supply or to balance the economy. The class stores an inventory vector representing items a shopkeeper might be carrying in their shop, or potentially other NPCs might be using for arbitrary purposes. They store a position, texture, sprite and a boolean type that is updated when the player is in range. Its functions are primarily for drawing it, accessing its variables and for shopkeeper functionality.

We also have two namespaces we have defined for use in various functions. First, is the cv namespace, defined in convenience.hpp and ~.cpp. It primarily contains utility functions that would normally be found in a graphic library, but which were added manually since they are not included in SFML. These are basic geometrical/linear algebra functions for e.g. normalizing vectors, finding the distance between two objects or calculating the dot product of two vectors. It also contains a function for checking if an open path exists from one point to another, by employing a simplified pathfinding algorithm using breadth-first graph traversal. These functions see a fair deal of use, as they perform basic tasks that would otherwise have had to repeat the same code over and over.

The other is the s namespace, defined in settings.hpp and ~.cpp. Its purpose is as a centralized place for storing game constants, such as the sizes of different GUI objects or the width of a tile. It also stores a vector containing the sound buffers of the various sound effects in the game, as well as a vector containing all the textures in use. The decision to use global variables was made after examining the possibility of passing the various textures or buffers around, or setting them as variables in other classes. Many textures and sounds are used repeatedly in different parts of the program, so this solution avoided this redundancy. After discussion, it was decided that using global variables accessible through the namespace would be both simpler, as well as better for functionality.

Software logic and the primary game loop

The entirety of the game runs within a loop held in main.cpp. The main file handles everything from necessary initialization and event handling to game logic. While many of the functions used are tied to their own classes and can often string together, everything starts from the main loop. The main function can be broken up into several sections.

At the start, the necessary fonts, sound effects and textures are loaded from files by calling the relevant functions. Then, the window is set up with the SFML RenderWindow and View functions. After this, game variables are initialized. The player character entity is created. The camera view is centered on the player. The map is created and the character is moved to the starting room.

What follows is setup for various UI elements, such as the health bar, inventory and the FPS indicator. They are set up using the various shape objects within the SFML library, moved into place and then filled with the relevant background color. Various other gameplay functionality related variables are also initialized before the main loop itself begins.

After the pre-loop setup, the main loop of the game begins. The loop is broken up into several sections, each handling a separate section of functionality. The start of the gameplay loop primarily handles quitting the game, resizing the window and pausing the game if the window loses focus. The resizing logic adapts the application to any resolution and aspect ration on the fly.

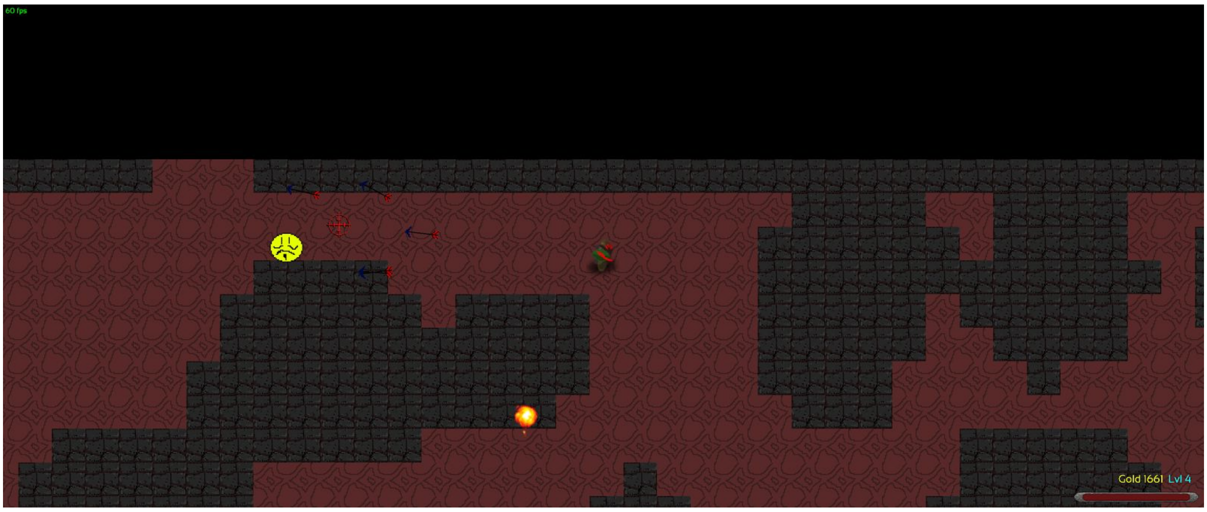
After this comes the code for handling the FPS counter. Essentially, it counts the time elapsed between frames in an effort to calculate how many frames are rendered per second. This amount is then printed at the top left corner of the screen. It is primarily included for testing purposes, as some issues presented themselves at lower framerates due to how time elapsed is used in the code. The FPS indicator can also give the user a view of how well their computer is handling the game.

The next set of code is event handling for movement. The direction is determined via a set of if statements that check if the movement keys corresponding to a direction are pressed. These if statements modify a sf::Vector2f entity used for storing the direction of movement where each component is evaluated separately to allow for sliding collisions (e.g., if the user is moving in the positive (x, y) direction but the x direction is blocked by a wall, they will still be moving in the y direction as per the projection of the velocity vector on the wall). Before the character is moved, their hypothetical position is determined to check if the player would move out of the room. If they are, the room the player is in is updated. Otherwise, the character's move function is called to update their position.

After character movement is the event handling for turning. The code checks the position of the mouse using functions provided by the SFML library. The difference in distance from the character's position to the mouse is calculated, and this is used to calculate the angle that the player sprite is set to.

What follows is the event handling code for attacks. Both types of attack check if the relevant mouse button is pressed, left for melee attacks and right for ranged. A second if-statement is used to check if enough time has passed since the character's previous attack. After this, if enough time has passed (determined by the player's weapon), a sound buffer based on the attack type is loaded into a Sound object and played. Last but not least, melee attacks call the room's performAttack function. In the case of ranged attacks a new projectile is created with stats based on the player's equipped weapon.

The last section of the loop contains all the code for rendering the various objects. The Room's functions are called to render items, monsters, npcs and projectiles and the GUI is drawn on top of everything. The last section of the loop is code related to the pause logic and player inventory.



Instructions for compiling and playing the game

Compiling the code is fairly simple. The included makefile should handle just about everything. The Make command compiles everything, Make Run runs the game itself, and Make Clean cleans up the executable and all the objects. They function well enough for most purposes.

On the off chance one wants to manually compile the game, several libraries are required because of the use of SFML. Each library used is a part of the SFML whole, and are fairly clear about what they cover.

Libraries: -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio

The gameplay itself is fairly simple. The player can move around using WASD, allowing for 8-directional movement. The player can choose a direction to face in using their mouse, attacking with a left or right click. Left clicks attack enemies in a range determined by their attack. Right clicks attack by creating projectiles that strike the foe.

When an enemy is slain, it will drop an item. The player can walk over these items to pick them up. Once the player has picked up a consumable item or another weapon, they can use the tab key to open up their inventory and pause the game (the latter can also be done using the 'P' key). From the inventory menu the player can use consumable items such as health potions or equip new weapons by left clicking on the item in question. The player also has a starting sword and bow they can switch back to if they so desire.

Past the starting screen and one set level, the player can walk from one randomly generated room to another. They can fight monsters, gather gold and find new items. In the current version of the game, the world is fairly simple but allows for easy future expansion with addition of new tiletypes, NPC's, quests, custom dungeons and so on. But in the current version, the map consists of a welcome room, which leads to the main room which in turn functions as a friendly place for the player to regain his breath after hectic combat sessions. Moving north from this room takes the player to a random-generated dungeon with four doors, the southern of which takes the player back to the friendly room while the other three take him to a new random-generated dungeon, thus creating an infinite loop of sorts. When the player dies, he loses some of his progress and finds himself back in this friendly room, from which he can sally forth anew. The purpose of the game is to see how long one can play and which level one can reach without dying, while not being entirely unforgiving if the player dies but would wish to continue.

How the code was tested

The code development and testing progress can be somewhat hard to quantify. The development was an iterative progress. After the very basic skeleton of the project was finished and in a functioning form, we proceeded by adding functionality from our plan one form at a time. Essentially, we used a form of iterative development.

We did not use separate unit tests as they were found unnecessary. In the situations we had them, we simply added them to the start of the main function. The same was done with creating test instances of classes to test their functionality and constructors. In situations where the result was not completely graphical, if something did not work, the code would not compile or it threw our custom exceptions. However, since this is a highly graphical application, any automated form of testing was redundant for most of the features and functions, since their effect was completely visual and could be accepted or discarded at once upon running the application. Likewise, the size of the program allowed for debugging a majority of issues through normal program execution.

When we ran into trouble, we would consult each other for advice to work out the source of the problem. After the simplest forms of issues were sorted out and the code compiled with no segfaults in sight, we moved onto phase two. At this point, issues were often situation specific.

We'd play the game in short bursts in an effort to find them. Once an issue was identified, we attempted to replicate them. With the circumstances of the problem identified, it was simple tracking down the possible causes with valgrind and printing information or implementing custom exceptions in the classes involved. We also used each other and external people for user experience feedback, which gave us valuable information about what to change or add. Additionally, we often reviewed each other's code and asked critical questions in an attempt to locate and solve issues and to maintain a constant understanding of other people's code.

Through this iterative process of developing features and slowly testing and implementing them, we managed to implement everything we planned without any significant issues.

Worklog

When we started work on the project, one of the first things we decided to do was to use Trello to manage who was working on what. We also agreed to have regular meetings where we would work together to code at Maari, in addition to people working on their own time. Instead of a strict split of labour, everyone decided what they wanted to work on by assigning themselves to the task in Trello.

We also chose not to have strict deadlines for features or functionality, instead opting for a more organic process. Things were added when they were done and tested, instead of rushing to have them finished by a specific date.

Features were broken down to cards, each covering a certain feature or an area of functionality. When someone decided to work on one, it was moved into the "In Progress" column and defined several milestones to track progress. Once said feature was finished, it would be moved into the last column, "Done", and the person would move onto another topic.

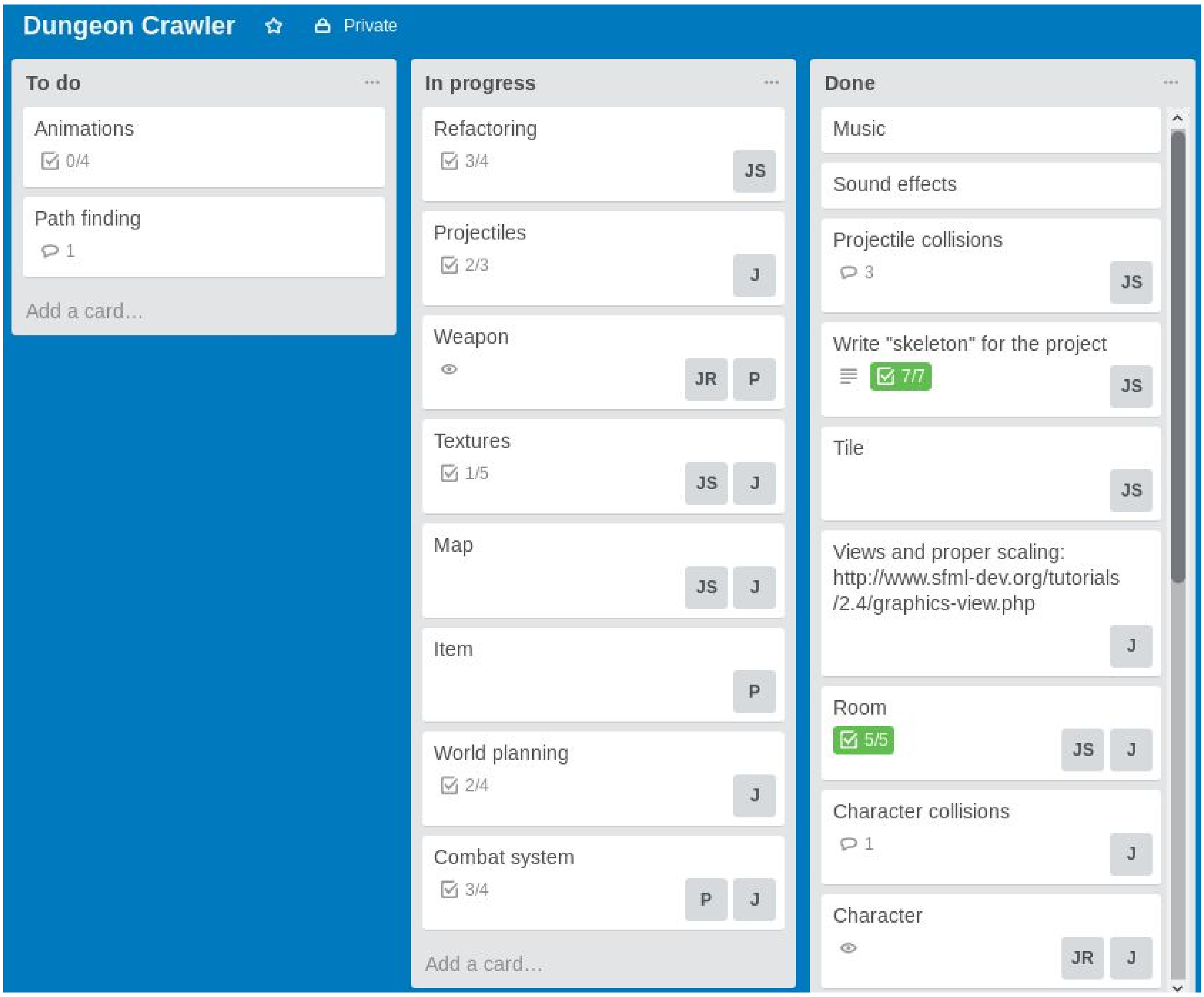
Through the use of Trello and Slack communication, we managed to track what was being worked on. The downside is, however, that we kept no record of what was finished when. Seeing the features we did manage to implement, however, it appears our process worked. We also failed to keep track of who spent how much time and on what. We did, however, have several 4-6 hour meetings at Maari. We agree within the group, upon comparing estimates, that over 250 man hours were spent working on our game.

As far as division of labor goes. It is available on our Trello linked here:
https://trello.com/b/2LEljbKc/dungeon-crawler
In general, the following features were worked on by the following people:

Project plan	Antti, Joel, Johan
Project skeleton & Makefile	Joel
Refactoring	Joel, Jon
Projectiles	Jon
Weapon	Johan, Antti
Textures	Joel, Jon
Map	Jon
Item	Antti
World planning	Jon
Combat system	Antti, Jon
Music/sound effects	Antti, (Jon)
Projectile collisions	Joel
Tile	Joel
Scaling and views	Jon
Room	Joel, Jon
Character collisions	Jon
Character	Johan, Jon
Room random generation	Jon
Other random generation	Antti, Joel
Monsters	Antti

Main loop, drawing	Jon
Animations	Jon
GUI & inventory	Jon
NPCs and shops	Johan
Testing & bugfixing	Everyone
Inline documentation	Everyone, mostly Jon
Doxygen doc	Johan
Project documentation	Antti, (Jon)

It should be kept in mind that several times someone passed a feature or class to someone else. For example, Weapon and Item has had several people working on it. Often, the class functionality would be created by one person with the graphics being worked on by someone else.



Dungeon Crawler ☆ Private

To do
- Animations ☑ 0/4
- Path finding 💬 1
- Add a card…

In progress
- Refactoring ☑ 3/4 JS
- Projectiles ☑ 2/3 J
- Weapon JR P
- Textures ☑ 1/5 JS J
- Map JS J
- Item P
- World planning ☑ 2/4 J
- Combat system ☑ 3/4 P J
- Add a card…

Done
- Music
- Sound effects
- Projectile collisions 💬 3 JS
- Write "skeleton" for the project ≡ ☑ 7/7 JS
- Tile JS
- Views and proper scaling: http://www.sfml-dev.org/tutorials/2.4/graphics-view.php J
- Room ☑ 5/5 JS J
- Character collisions 💬 1 J
- Character JR J

Screenshot of our Trello page

Creative Commons attributions

Music: Artist: Tri-Tachyon, Track: Hundred Years in Helheim, Album: The Spellbreaker EP
Link:
http://freemusicarchive.org/music/Tri-Tachyon/The_Spellbreaker_EP/Hundred_Years_in_Helheim
Licensed under Creative Commons non-commercial attribution license.

Ranged bow attack sound effect: Cartoon Arrow 08.wav, creator: debsound
Link: https://www.freesound.org/people/debsound/sounds/319550/
Licensed under Creative Commons noncommercial attribution license.

Ranged fireball attack sound effect: Space Swoosh - brighter, creator: GameAudio
Link: https://www.freesound.org/people/GameAudio/sounds/220191/
Licensed under Creative Commons 0 license.

Melee attack sound effect: sword-01.wav, creator: audione
Link: https://www.freesound.org/people/audione/sounds/52458/
Licensed under Creative Commons 0 license.

Death sound effect: trumpet3.wav, creator: bennstir
Link: https://www.freesound.org/people/bennstir/sounds/81822/
Licensed under Creative Commons attribution license.

Getting hit sound effect: thud.wav, creator: OtisJames
Link: https://www.freesound.org/people/OtisJames/sounds/215162/
Licensed under Creative Commons 0 license.

Level up sound effect: Win Spacey, creator: GameAudio
Link: https://www.freesound.org/people/GameAudio/sounds/220184/
Licensed under Creative Commons 0 license.

Level up sound effect 2: Mike Koenig
Link: http://soundbible.com/1003-Ta-Da.html
Licensed under: Creative Commons Attribution 3.0

Monster sound effect 1 & 2: Doberman
Link: http://soundbible.com/1963-Giant-Monster.html
Licensed under: Creative Commons Noncommercial 3.0