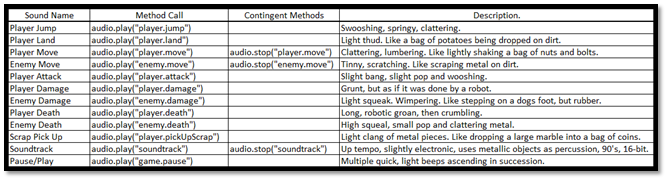
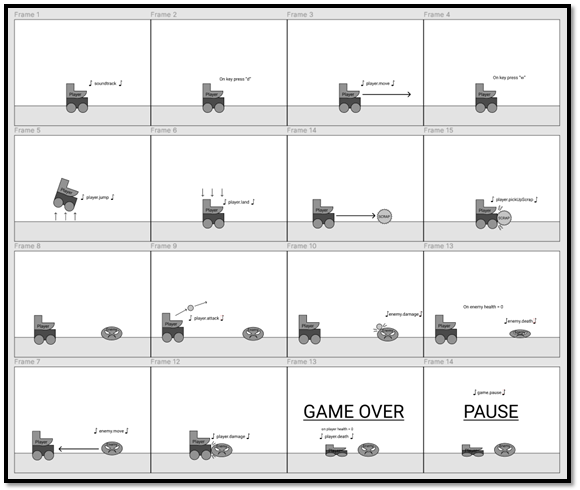
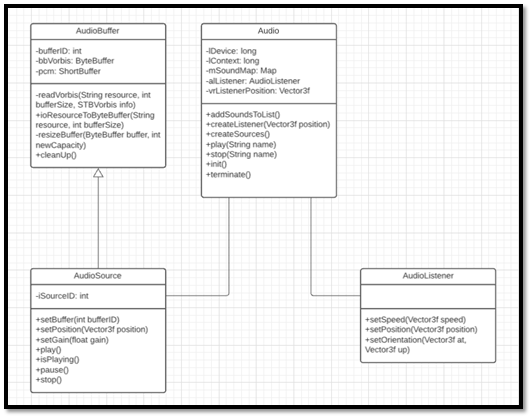
**Assessment 2 – Audio EVF.**



This is a table of all the sounds that will be developed and used in our game, the methods required to call those sounds, their contigent methods mostly relevant to whether the sound is looping or not, and a non-detailed description of how the sounds should turn out when created. The expectation of this table is to provide a clear understanding of the sounds that will be required, and to not waste time creating incorrect sounds that don’t fit the intended action. It also indicates the methods required to call the sounds, with appropriate wording, and alludes to whether they are looping, or play a single time. When beginning the process of creating the sounds in Ableton Lite, the planning process is already complete, and we will only need to fit the sounds as close as possible to the specifications in this table.



This interaction map is an example of how each sound will interact with actions performed in the game. It gives everyone involved in the planning process a graphical method of visualising the actions related to sound generation, when those sounds are generated, what causes them, which characters, and the input or condition that triggers them. This avoids any confusion as to what time and on what action to place which sounds. This diagram also offers clarity on when finalising sounds, or parralel actions are needed, for example: Whenever the player.jump sound is used, player.land must be played in conjunctor, and when the Player dies, the player.death sound needs to be played as well as the “game over” screen displayed, and the player death process ran.



This is a class diagram of the Audio related classes, and how they work together. As you can see, the process starts in the AudioBuffer class where the audio files are decoded and loaded into a buffer. The AudioSource class extends AudioBuffer to maintain accuracy in the bufferID and sourceID parameters. Its purpose is to recieve the buffer data and store them for use later and to handle the position of the sound being played in the 3D game space (though we are not using the 3rd dimension in our project), the gain (volume) of the sounds, and the methods required to play, pause and stop the audio sources. These sources are fed into the Audio class, the main class that is the controller for all audio related interactions, AudioBuffer and AudioSource variables are created and loaded with sound data from external files, then stored in a HashMap to be easily retrieved and played when required in game. The AudioListener class is necessary to control the input or “position of the listener” in the 3D gamespace. It can be placed anywhere in the 3D space to provide a sense of depth and immersion to the sounds playing in game. For example: as the Enemy approaches the Player, the sound of it moving can be made to increase in volume as it gets closer.

Extended Feature 3 Validation Testing:

|  |  |
| --- | --- |
| **Validation** | **Failure** |
| Sound plays when related action occurs. | * Sound plays “out of synch” - at incorrect time. * Incorrect sound plays – incorrect source. * Sound does not play. * Sound loops when it is supposed to play once. * Sound plays once when it is supposed to loop. |
| Sound is clearly audible. | * Sound is crackly, due to incorrect file input. * Sound plays abruptly, due to not maintaining the thread. |
| Sound plays at an appropriate volume. | * Sound is too loud, quiet or inaudible. |
| Sounds are of a high quality, and in line with group expectations. | * Sounds appear low quality and not made with consideration. |
| Sound data loads into correct buffers and sources from file. | * Incorrect sounds play when calling. * Errors occur when compiling program. |