Project Schedule

Name: Beat Breaker

Student Name: Jake Fitzgerald
Student No: C00288105

Phase 1 (November 10th - November 30th)

Phase 2 (December 1st - December 24th)

Phase 3 (January 1st - January 30th)

Phase 4 (February 1st - February 30th)

Phase 5 (March - April) [FINAL PHASE]

Phase 1: Core Systems & Setup (Weeks 1 – 2)

- SFML Project setup with file structure.
- Input handling:
  - Input / Event handling type (Keyboard keys, Joystick controller)
  - Latency and grace period for detecting if the input was on beat.

- Basic scene layout (Menu, Options, Gameplay, Results, etc).
- Visuals
  - Window resolution (static/dynamic)
  - Sprite sizes, consistent numbers (64 by 64 pixels, etc)
  - Use debug SFML shapes for testing (toggle with key)
  - UI Buttons for scene navigation (clickable), Font import and basic text.

Simple character controller (physics, gravity, friction (air/grounded), collision detection)

- States: [Horizontal movement, Jumping, Falling, Block breaking]
- Simple animation
- Basic animations (move left/right,jump, falling, hit)

Character gameplay attributes:

- Health timer (slowly depletes based on BPM count of the current MIDI track).
- Lives (how many times the player can fail during a run).

Gameplay HUD

- Score Counters [Current Score, Personal Best]
- Health Gauge
- Lives Counter
- Track “Deepness” (How far you are into a song).

Pause Menu

- Pause/Retry when game is paused.
- Stop the timings for the MIDI track and be able to resume at the same place.

Main Menu

- Simple clickable buttons that lead to the different game scenes
  [TEMPORARY SCENES TO TEST/DEBUG]
  (Gameplay Test, Character Controller Test, Block Generation Test, MIDI Parsing Test, Leaderboard Test, Results Scene Test, other testing scenes…).

Level generation

- Preset test level using non-randomised block patterns
- Generic block patterns used for balance
  (Plus shaped obstacle block with health in the middle [RISK/REWARD])
- Using MIDI to influence which block patterns are chosen and what blocks are used in the randomised block patterns

- (More snare hits allow for higher amounts of single non-joined blocks, less would have more joined block colours to allow for less hits needed to be used to move downwards through the level.

Timeline scrolling

- Horizonal trigger collider scrolls down through the screen vertically in time with each measure of the music track. When it reaches the end it then spawns back at the top at the start of the measure.

Input timings with MIDI track

- Test scene that prints the timings of the MIDI track for when inputs are determined to be in either the range of ‘Too soon’, ‘Perfect’, ‘Too late’ -> All considered ‘On-beat”.
- ‘Grace period’ implementation to allow for it to be easier to hit the beats by allowing a small range of hitting the beat to be accepted. This also helps if the game slows down at any time causing inputs to be dropped.
- Allow to dynamically adjust latency based on the current framerate, this could also assist in dropped inputs if the game encounters slowdowns.
- Simple HUD element component that will be used in the finished gameplay scene: Beat counting display -> Set of sf::rectangle shapes that are aligned in a row. A rectangle shape’s colour is changed from Blue to Red as the beat is aligned with said rectangle.
  Example: Common time (4/4)
  ]Hit the first beat]
  1, 2, 3, 4
  RED, Blue, Blue, Blue

  [Hit the second beat]
  1, 2, 3, 4
  Blue, **RED**, Blue, Blue

  [Hit the third beat]
  1, 2, 3, 4
  Blue, Blue, **RED**, Blue

  [Hit the final beat]
  1, 2, 3, 4
  Blue, Blue, Blue, **RED**

  This will also make debugging visually clearer as to when the beat is synced with the MIDI’s music. Using the MIDI parser, we can also change the amount of rectangles that represent the current time signature (3/ 4, 6/ 8, etc). This makes it easier for the user to understand when they should be hitting the key press.

Collectibles:

- Basic spawning, collision and visuals (debug hitbox, basic sound cue on pickup).
- Item List:
  - Health (Record)
  - Bomb (Walk into to activate) (Destroys other blocks surrounding it)

- 1 UP (Adds to Lives Counter)
- Slow time (Beat Sweeper)
- Air Bubble (Stops health drain for a period of time)

First playtest
- Determines if the basic controls are intuitive and easy to remember when playing.
- Recording gameplay through screen capture to watch it back if any bugs appear and how to replicate them.
- General feedback for if the game is fun and interesting to play.

Phase 2: Simple prototype showcasing core gameplay (Weeks 3 - 4)

First playable demo
November Presentation

Character controller polish
- Better collision detection for edge cases such as when the player is inbetween the space when two blocks from above are falling, which one damages the player. Also if the player is at the very edge of a block falling, allow the player to automatically jump to the side of the block IF that space is open (no other blocks there).

Options Menu
- Volume adjustment [Music, SFX]
- Difficulty [Adjust “Grace Period” range to make it more lenient or more strict

Block class:
- Sf::vector2f coordinates (location in the global gameplay space)
- Collider: (Player + other blocks)
  - Sf::rectangle shape + sf::colour (debugging colours)
- Hitbox: (Player [DAMAGE])
  - Sf::rectangle shape + sf::colour (debugging colour)
- Block types:
  - 4 different colours: BLUE, PINK, GREEN, RED
  - Heavy - takes multiple hits to break + drain portion of health
- Texture/ Sprite:
  - 4 different colours: BLUE, PINK, GREEN, RED

Level Generation:
- SRand can be inefficient and not always generate a new number everytime it is ran.
- (every iteration we need to generate a new block section)
- We can use Mersenne Twister to not have this issue.
- We can also create an additional function to randomise this random number after it is generated, to ensure it a unique number everytime (i.e. generate another number and add or minus the original random number).

Second playtest:

- More bug finding.

MIDI Parse
- Covert from the older version of Big Endian to work with our current C++ code.
- Parse out string data so that we can get our BPM, Channel Name, Time Signature
- Calculate the distance between the notes. Use BPM and the time data where the note is stored in MIDI [MIN, SEC, MSEC].

Sound Manager
- Reserves a certain amount of sf::sounds in a pool.
- Create a sf::soundbuffer and store that in map (sound name & buffer name)
- Get an sf::sound from the pool, if we can't then we make another one.
  (Added a check in case our pool gets too big, especially with different SFX playing)
- Play the sound based on the SoundType's (MUSIC/SFX) volume.
- Need to add in a randomise pitch function for certain sounds (Block breaking).

Options Menu
- Music toggle button
  - Sets the volume for SoundType::MUSIC to 0.0f
- Music Test button
  - Plays a Music sound.
- SFX toggle button
  - Sets the volume for SoundType::SFX to 0.0f
- SFX Test button
  - Plays aSFX sound.
- FPS Display toggle button
  - Toggles the sf::text for the game's frames per second on screen.
- Save Preferences button
  - Writes the preferences (volumes) to a .TXT file.
- Load Preferences button
  - Reads the preferences (volumes) from a .TXT file.
- Return button
  - Loads the Main Menu scene.

Difficulty Modes
- Easy, Medium, Hard
- Influence the timing range for inputting 'Breaking' input:
  (Easy - increases range, Hard - decreases range).
- Influence certain Block Generation sections:
  (Amount of Health, Life, X Blocks will spawn).
- Influence the amount of points you score overall:
  (Multiply the Depth, Item usage, lives lost score -> (+/- [Easy/Hard)))
- Influence the speed for how fast the Player's health drains.
- Selected in the Main Menu with three different buttons.

HUD (UI)
- Simple HUD object that is created when the game starts.

- Display different UI elements for the Gameplay scene, Player and other utilities like an FPS counter currently.

Presentation:
- Showcase the small portion of work completed so far.
- Slides showing snippets of code to explain how it works in the current version of the project, and how it can change in the next phase of the project.
- How far is the core features completed (percentage along with pie charts).
- Using illustrations created for the existing documentation in the slides to explain certain ideas or how certain sections of code will work.

Phase 3: (Weeks 4 - 7) (Christmas Break)

Tuning :
More polish for character tuning (Movement speed, falling speed) and bug fixing.

Visual Polish:
- Add visual polish such as particles for picking up Collectibles, disappear animation. Hit spark for when breaking a Block (different sprites/audio cues for if it is [invalid, too late, perfect, too soon].
- Icons and different audio cues for when you chain a combo and how many blocks were in said combo chain.
- Background art (sprite) for different scenes (Main Menu, Gameplay, Options, etc)

Joystick Controller Inputs:
- Allow for inputs from a controller (DS4, Xinput, etc).

Third playtest:
- More bug finding.

(After Christmas)
Phase 4: (Weeks 8 - 12) (January - Midterms)
Visual Polish:
Creation of different visualiser for representing either parsed Midi data or Live Midi streamed from Midi Instrument.

Track Visualiser
- Displays the different tracks inside a Midi file by instrumentation.
- Displays an sf::rectangle shape that appears only when that track is played.
- Displays the track's name that is parsed from the midi file as a unique string.

Piano Visualiser
- Each key represented by a unique sf::rectangle shape. The keys are layed out in the same fashion as a real life piano with white keys on the base and blacks keys overlayed on top of them.

- Read the parsed Midi track (singular) and assign each key on the visualiser to the corresponding keys in the Midi track's note vector (may need to ignore certain notes since Piano's typically have 88 keys, our visualiser has 85 for simplicity and the actual note data in a Midi track is always from 0 -127. We can use middle C as the starting point and ignore the extra notes from the highest and lowest pitches, since they are usually never used.

Drum Visualiser

- Displays a visual of a drumkit with each of it's sections isolated allowing us to animate them (through scaling/colour) when the midi reads one of those sections.
- Playback for each of the different drum sounds using the Audio Manager.
- Read the parsed Midi tracks and assign each track based on matching instrument's name using a string (i.e. Kick track notes assigned to Kick (sprite, sound, animation, etc)

HUD Grid added to allow easier placement for the different entities in each visualisation scene (Text boxes, Rectangle shapes, Buttons, etc).

HUD playback buttons

- Allows for the Midi file's to be controlled using the buttons that are placed near the bottom of the screen.
- List of buttons:
    - Play
        - Begins playback based on the current section of the song.
    - Pause
        - Paused the playback and holds this section's location until another button is pressed.
    - Stop
        - Stops playback and resets the song to the beginning.
    - Skip to End
        - Skip to the end of the song.
    - Skip to Start
        - Skip back to the beginning of the song.
    - Mute On/Off
        - Turns the song's audio buffer(s) on or off.

Midi Parser:
Parsed header chunk to have readable data for valid midi file using a string ("MThd"), Midi Version (0, 1, 2), Track count and Ticks per Quarter Note.

Track read as a valid track with the string ("MTrk"), then defined Meta Events (FF xx), Running Status byte on Channel Messages (0x80) and Data Events (higher than 0x80).

VLQ (Variable Length Quality) stores the delta times between each event inside a Midi Track

Meta Event List:

metaEvent = 0xFF,
tempo = 0x51,
timeSignature = 0x58,
keySignature = 0x59

Data Event List:

noteOff = 0x80,
noteOn = 0x90,
afterTouch = 0xA0,
controlChange = 0xB0,
programChange = 0xC0,
channelAftertouch = 0xD0,
pitchBend = 0xE0,
systemExclusive = 0xF0

Then we can store the parsed data into a vector of tracks to be read in by the different visualisers. (see ‘struct MidiTrack’ and ‘struct MidiNote’ in Documentation)

Live Midi
Live input being read from a Midi Instrument (Keyboard) that can be visually represented with the different visualisers (Piano Visualiser).

- Keeping track of different incoming messages in the Midi stream
- Always check if a new note already is being played (Running Status is already figured out in the parser).
- Calculate the correct timings and convert them from Ticks (microseconds) into actual seconds to be used with delta time in SFML.
- Flag and ignore useless data we don’t need like System Exclusive messages and other status data for specific instrument details (extra ports, channel messages, etc)

Phase 5: [FINAL PHASE]