BBC Micro:Bit Game – ‘Space Junker’

Practical 1

Student: Nathan Poole (170004680)

Abstract (Word Count: 192)

A video game has been created using *Processing* and *BBC Micro:Bits*. The game is multiplayer (2+ players, but single player is supported) and is controlled completely by Micro:Bits. The game combines a survival mode mechanic with competitive head-to-head; players are ‘Space Junkers’ who compete to collect satellite debris (for score) whilst dodging lethal asteroids (to avoid elimination).

Players use the tilt of their Micro:Bit to control an on-screen avatar fluidly and precisely. Additional inputs from the Micro:Bits support further game mechanics (e.g., player abilities) and quality of life features (e.g., player connection phase and pausing), culminating in an entertaining and polished experience. Several outputs are used meaningfully to enhance the game: speakers support immersive game sounds, individual player statuses are highlighted on their Micro:Bit LEDs, and servos provide passive haptic feedback.

Overall, several game objectives, features, and mechanics influenced from video games in general have been combined with various inputs and outputs, as guided by HCI principles, to provide an arguably original gameplay experience. This satisfies the original practical intention of fully exploring the use of Processing and the capabilities of the Micro:Bit as possible software and hardware options for future projects.

Interaction Overview

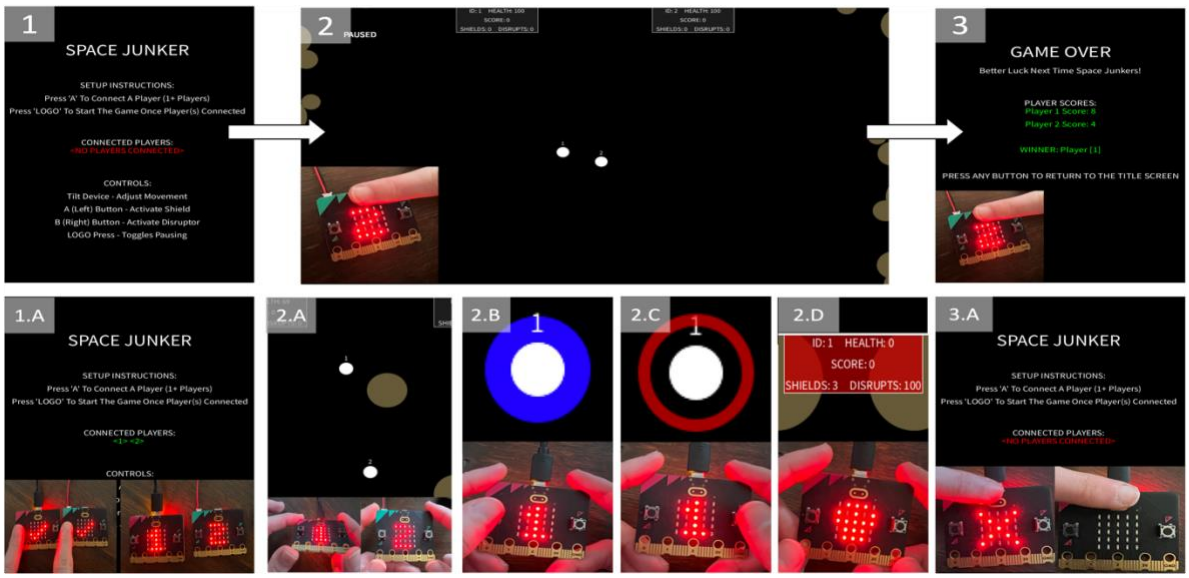

Figure 1: Interaction between hardware and software in the developed game.

Figure 1 outlines game interaction. There is an initial player connection phase (Figure 1, part 1). Players press ‘A’ to connect. The device shows a success icon and presents their player number (part 1.A). Once 1+ player(s) have connected, any player taps the ‘Logo’ to begin the game (part 2). During the game, players tilt their device to move (part 2.A), press ‘A’ to activate shields (part 2.B), and press ‘B’ to activate disrupts (part 2.C). When a player dies, their device shows a skull icon (part 2.D). Once all players are eliminated, the game is over (part 3). Pressing any button at game over returns the players to the title screen and disconnects them, ready for a new game (part 3.A). Interaction with the servo is not shown due to its stability issues (while a player is being hit, the servo acts like a rumble motor).

Parts List

The following details the parts required to make the prototype developed for this practical:

Hardware

- BBC Micro:Bit(s) - *minimum 1 for single-player, 2+ for multiplayer*.
- Micro:Bit Compatible External Battery Pack(s) With Batteries – *1 per Micro:Bit, excluding relay*.
- Micro-USB Cable - *with relevant adapters for computer (i.e., Micro-USB to USB-C for MacBook)*.
- Screen/Display
- Speakers
- Wiring - *red VCC wires, black ground wires, and (orange) signal wires recommended*.
- Micro Servo 9g (SG90) – *1 per Micro:Bit*.
- Misc. Paper/Cardboard/Tape – *to fashion rumble motor housing on servo motors*.

Other

- Processing IDE – *with the ‘minim’ sound library*.

Process

The steps to replicate this project are divided into three distinct phases: (A) *Processing* code, (B) *Micro:Bit* code, and (C) hardware set-up. Each phase is individually addressed but a system overview is given first to understand how these phases connect and combine into the finished product (Figure 2).

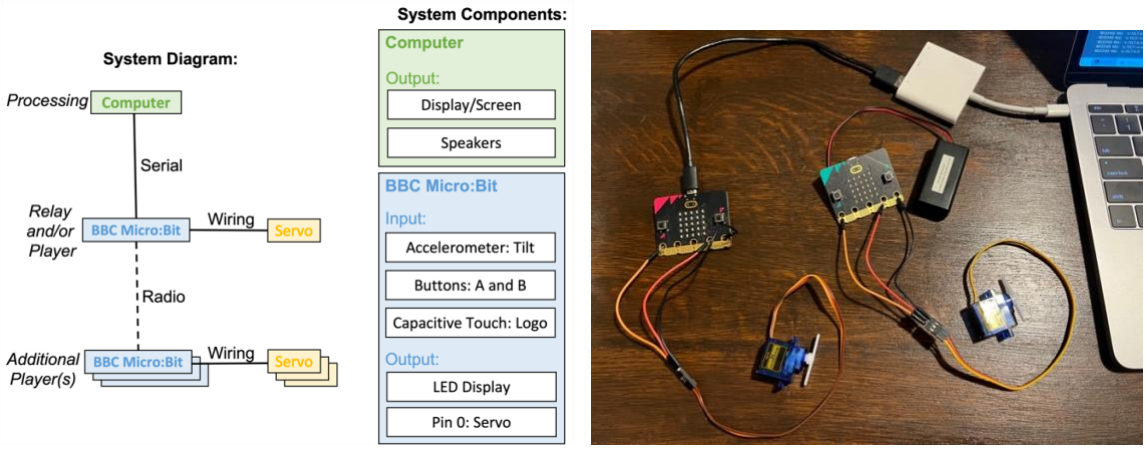

Figure 2: System Overview – A system diagram (left) shows how components connect. A component break-down (middle) highlights the varied inputs and outputs. An example of this system is given (right).

A - Processing Code:

The *Processing* code required for this project is given with this submission but is outlined as follows.

The game files have a hierarchical structure typical of object-oriented programming (Figure 3). '*Game*' is the main file containing the game loop, which triggers game update and render to the display and speakers every frame.

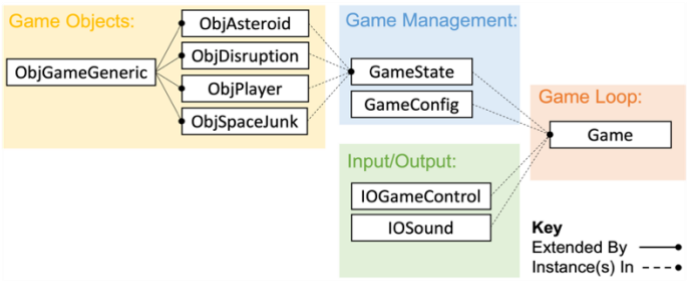

Figure 3: Processing game file hierarchical structure.

'*GameState*' governs the collections of objects comprising the game. Namely, array lists are used for their extensibility, allowing for multiple players and dynamic game elements. Game behaviour decisions are parameterised in '*GameConfig*'. This file controls the game settings, allowing for effective game balancing by adjusting these settings.

'*IOGameControl*' manages player input using event programming and message processing via a serial connection to the Micro:Bit relay (see Appendix C). '*IOSound*' pre-loads sound assets at set-up (game music, alert chimes, etc.) and controls use of them during the game (see Appendix B). The use of sound provides important audio cues, reinforcing awareness of game events and crucially supporting immersion.

All game objects extend a common root class, '*ObjGameGeneric*', which establishes the use of circles to represent game objects and the use of geometry for interactions between them. This allows for convenient and ubiquitous application of collision detection and resolution. Game objects are represented by their own class, so their distinct behaviours are modelled (see Appendix A).

B - Micro:Bit Code:

The *Micro:Bit* code required is given with this submission. The code on all Micro:Bits is identical regardless of whether a Micro:Bit is the relay and/or a player. This code symmetry allows for a concise and manageable implementation of the Micro:Bit logic across all devices, resulting in effective event handling in the face of bidirectional and broadcast communication (see Figure 2). Each Micro:Bit is distinguished by a unique player ID ('1' being the relay with the serial connection).

The relay is responsible for receiving any player inputs via radio messages and forwarding them to Processing via USB, whilst also receiving player output messages from Processing and broadcasting them across the radio group. If the relay is also a player, it must communicate its own input data and process any of its own output messages. For players operating wirelessly, they simply send their input data as required and await messages directed to them via the radio channel. It was necessary to implement a simple but robust protocol for such behaviors to be effectively managed (see Appendix C).

Radio provides convenience by allowing wireless play, but the use of the open medium results in issues. Unexpected or noisy messages may be received, and messages may be lost entirely. Careful implementation is required to only process messages of the expected format. Also, the system should not create an error by the absence or out-of-order arrival of messages.

Finally, use of the Micro:Bit inputs and outputs is outlined and justified as appropriate and meaningful.

- *Input - Accelerometer ('tilt'):* When a device connects, it periodically transmits pitch and roll data until disconnected. The data range is [-180, 180], which conveys the direction and degree of pitch and roll; players input the direction to move by tilting the device in said direction and specify speed by the degree of tilt. This natural mapping of movement is convenient given the small form factor of the Micro:Bits. This movement is fun as it takes skill to learn and master, resulting in the ability to move precisely, which is crucial for success in the developed game.
- *Input - Buttons ('A' and 'B'):* Button presses are quick, singular inputs that are used to perform single-use actions (e.g., connecting to the game and activating an ability).
- *Input - Capacitive Touch ('Logo'):* The capacitive 'Logo' is used for rare events (starting the game and toggling pause) as it is less convenient to use during gameplay. The use of 'Logo' takes advantage of its properties. The game is quick so pausing needs to be quick; capacitive touch results in immediate pausing, instead of waiting for a button to be engaged and disengaged.
- *Output - LED Display:* The LED display is used to reinforce the relationship between the device and the on-screen avatar by constantly showing the player ID and displaying player statuses (connection, death, etc.). This is crucial; it is easy to lose track of an avatar, especially amongst multiple players. The LED display allows the player to quickly re-establish themself. Future work could use colored LEDs as outputs and color-code players as an even better solution.
- *Output - Servomotor:* Servos are used to fashion rumble motors, typically used in video games to provide haptic feedback to players for events such as player hits (i.e., taking damage).

C - System Set-Up:

Final set-up of the hardware is self-explanatory given Figure 2. Most inputs and outputs come from the computer and Micro:Bits. However, the wiring between the Micro:Bits and the servos should be set-up as illustrated in Figure 4. A rudimentary housing can be created around the servo 'blades' such that the 'blades' knock-off this housing when turning, simulating a (weak) rumble motor for haptic output.

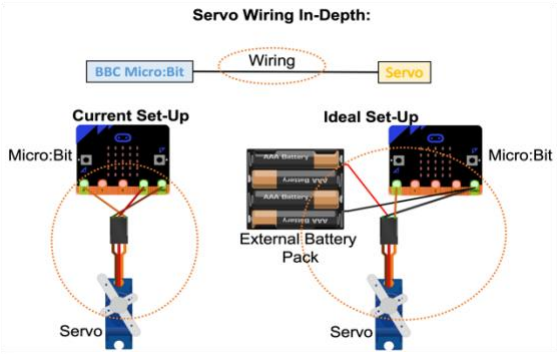

Figure 4: Micro:Bit to servo wiring (made with Fritzing).

Challenges & Key Points (Word Count: 195)

The single-threaded Micro:Bit code resulted in unexpected behavior due to busy code blocks. Radio messages were being missed by the relay due to it waiting for LED animations to finish. This was solved by using instant output actions instead of actions with time intervals (i.e., showing an icon and removing it when conditions are met instead of showing an icon for a few seconds).

The biggest challenge faced relates to the external servomotor used as a rumble motor on each Micro:Bit. These servomotors usually operate in the 4-9V range (the Micro:Bit outputs just 3V), so connected servos rotate inconsistently or not at all. The Micro:Bit gives insufficient power draw. This issue could be easily resolved by powering the servos with external battery packs; however, these could not be acquired for the practical submission. Thus, the servos are kept as a proof-of-concept but require further work.

If I were to do the practical again, I would explore a game where software supplements a physical game via the Micro:Bits. This is the opposite scenario of the current game where real-world data supplements a virtual game. I believe the former is more beneficial for exploring novel human-computer interactions.

References

- Servo Wiring: https://support.microbit.org/support/solutions/articles/19000101864-using-a-servo-with-the-micro-bit
- Micro:Bit Guidance: https://makecode.microbit.org/reference
- Micro:Bit JavaScript Guidance: https://makecode.microbit.org/javascript

Appendix

A - Accompanying Material:

As this report focusses on the design and implementation of the game, specifically regarding interaction between the Micro:Bit(s) and Processing, a supplementary game guide has been included with the submission. This game guide intends to assist players with understanding the game and its mechanics (see *CS5041_P1_GameGuide.pdf*).

B - Game Resource Sources:

The following non-standard libraries are required by the software of this practical.

- *Minim* library (for *Processing)*: http://code.compartmental.net/tools/minim/

The following are references to the sources of any game assets used as part of this practical. It was ensured that the assets could be used in accordance with their respective licenses.

- Game Music: https://www.zapsplat.com/music/game-music-action-game-fast-paced-pumping-high-speed-style-synth-melody-fast-electro-drums/
- Game Over Sound: https://www.zapsplat.com/music/8-bit-game-over-80s-arcade-simple-alert-notification-for-game-2/
- Player Connected Sound: https://www.zapsplat.com/music/game-tone-bright-and-positive-musical-tone-good-for-winning-award-or-point-or-success-version-3/
- Collected Junk Sound - “Mechanical crate pick up”: https://mixkit.co/free-sound-effects/game/?page=2
- Activated Shield Sound - “Arcade retro changing tab”: https://mixkit.co/free-sound-effects/game/
- Activated Disrupt Sound - “Arcade mechanical bling”: https://mixkit.co/free-sound-effects/game/
- Hit Asteroid Sound - “Small hit in a game”: https://mixkit.co/free-sound-effects/game/?page=2
- Player Death Sound: https://www.zapsplat.com/music/explosion-crash-debris-large/

C - Communication Protocol:

The communication protocol is to send strings with a standard format:

<PLAYER_ID>:<MESSAGE_TYPE>:<MESSAGE_DATA>\n

, where <MESSAGE_DATA> may contain multiple strings also delimited by colons.

- <PLAYER_ID> is used to identify the player concerned with the message. Players send their player ID to communicate to the game whom the input message is coming from. The game sends messages with output instructions specifying the player to perform said instruction.
- <MESSAGE_TYPE> is used to convey the purpose of the message. For example, 'PLAYER' messages indicate a player status, such as connected, disconnected, or death. Other messages such as 'TILT' or 'BUTTON' are used to indicate input.
- <MESSAGE_DATA> carries the payload data associated with the message type. For instance, 'TILT' messages have a payload of the form '<ROLL>:<PITCH>', indicating the roll and pitch of the corresponding Micro:Bit device at the time of sending.

Key details must be observed. First, messages have a finishing new-line character denoting the end of the message. Also, messages are always less than or equal to 19 characters. This is because the maximum number of characters that can be sent over radio in a single message is 19 characters (defined in the reference: https://makecode.microbit.org/reference/radio/send-string).

The following are all message forms used in this project:

- <ID>:PLAYER:ACC\n - *Message from the game that player <ID> is connected.*
- <ID>:PLAYER:DEC\n - *Message from the game that player <ID> is disconnected.*
- <ID>:PLAYER:DEATH\n - *Message from the game that player <ID> is dead.*
- <ID>:RUMBLE:AST\n - *Message from the game that player <ID> is hit by an asteroid.*
- <ID>:RUMBLE:DISR\n - *Message from the game that player <ID> is hit by a disrupt.*
- <ID>:BUTTON:A\n - *Message from player <ID> that the 'A' button was pressed.*
- <ID>:BUTTON:B\n - *Message from player <ID> that the 'B' button was pressed.*
- <ID>:BUTTON:LOGO\n - *Message from player <ID> that the 'LOGO' was pressed.*
- <ID>:TILT:[-180, 180]:[-180, 180]\n- *Message from player <ID> giving device roll and pitch.*