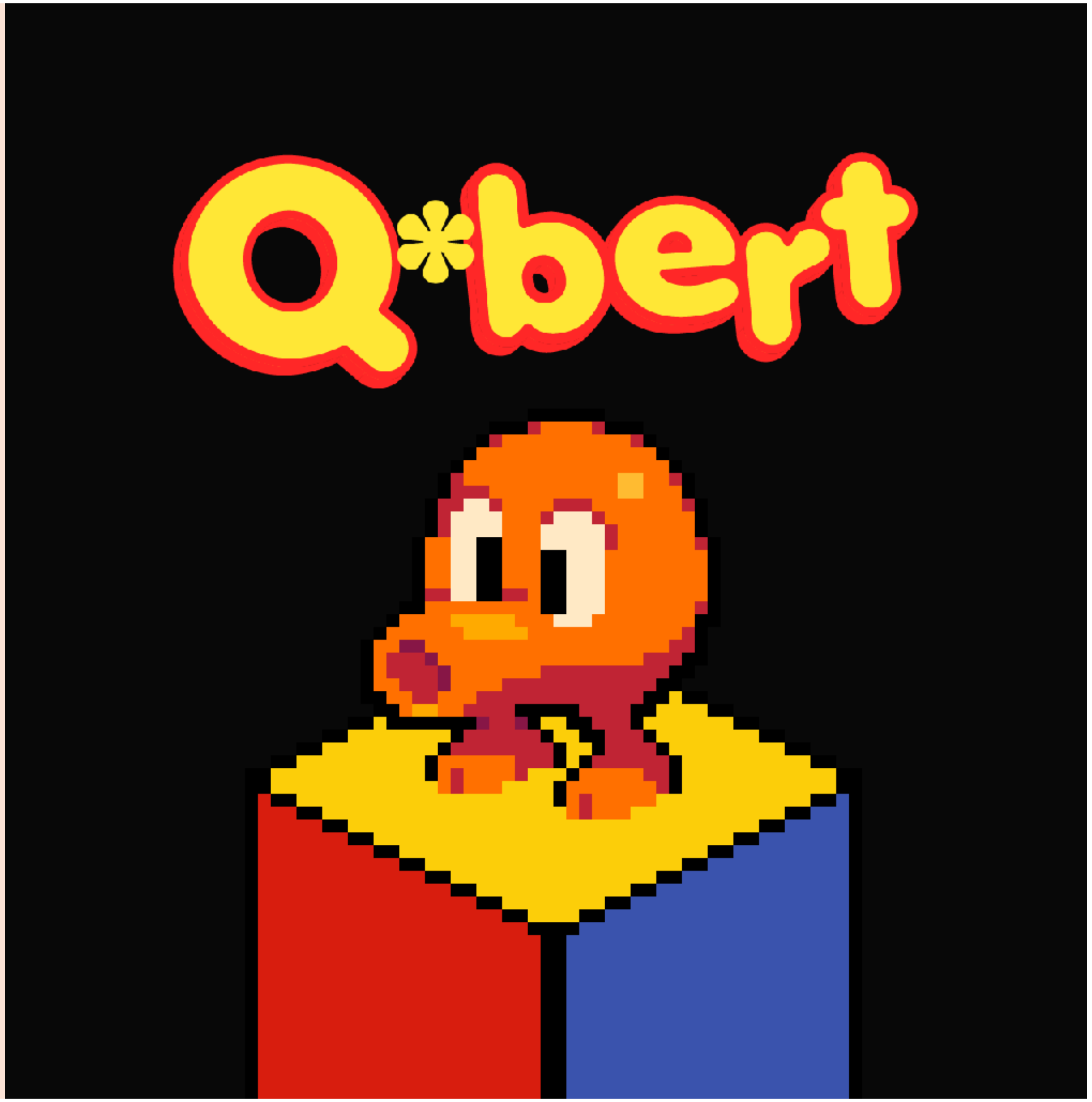
Q*bert in MultiProcess

By: Alies, Gianni, Will



Introduction-What was our project?

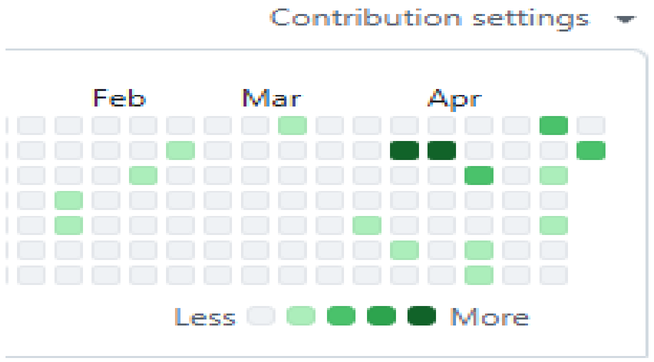
- We rebuilt the original arcade classic, Q*bert, within pythons pygame in a multiprocessed environment.
- The main goal was to initially rebuild the game from scratch, then to multiprocess it to allow smooth running within pygame’s engine

Building from the ground up

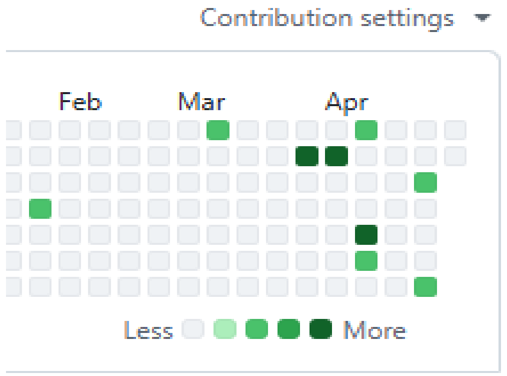
- When starting this project, we built the Q*bert game from the ground up.
- Meaning doing all assets, sounds, programming, and most important multiprocessing
- We wanted to build the game from the ground up so we would fully understand how multiprocessing would work and how to implement it if we needed to in the future

Contributions

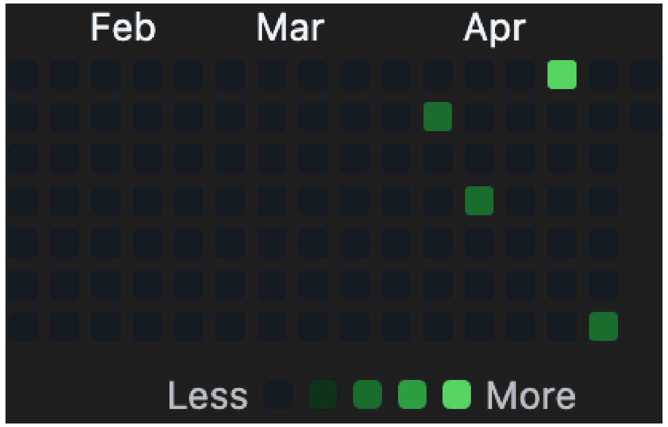
Alies Contributions

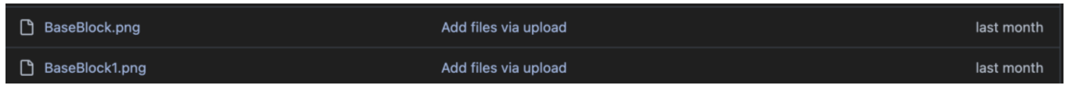


Gianni’s Contributions



Will's Contributions





```
BaseBlock.png     Add files via upload     last month
BaseBlock1.png    Add files via upload     last month
```

(Sprite creation was not showing up on the contributions timeline)

Note: Certain aspects and code was shared over discord so take this with a grain of salt

Contributions Expanded

- We divided the work between the three of us

Alies worked on – Getting assets loaded in a working, animations, mulitprocess with redball and jump sound. And most base logic revolving around the game.

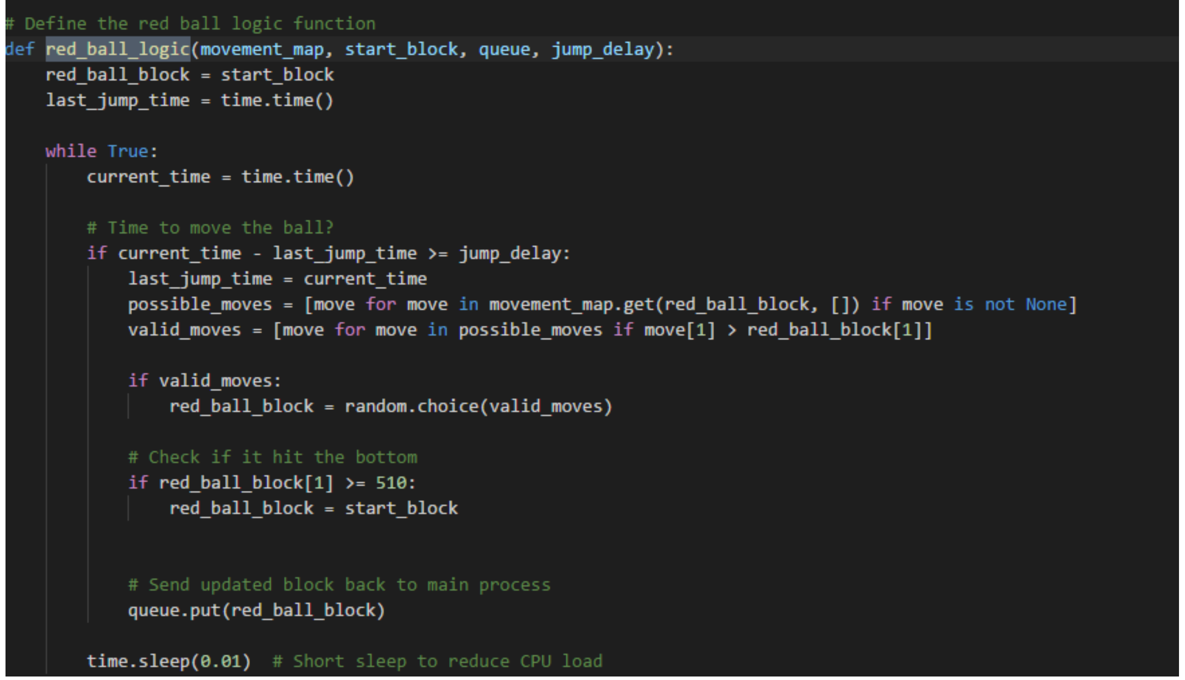
Gianni covered the Title screen, Internal stage mapping, character movement, multiprocessing of the rainbow discs( ) as well as their movement

Will Acquired and refined all necessary sprites, made the life system, integrated a game over screen, working score, and other UI elements, and implemented the ability to capture each square.

What we learned

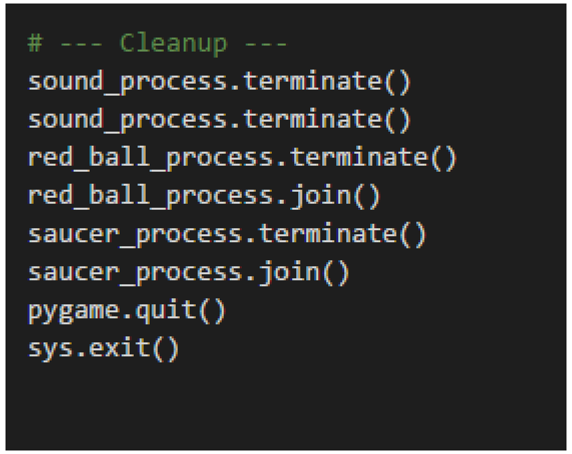
- The pygame library is simple yet effective approach
- Python runs on one core by default, and we wanted to limit parallelism for consistency, simplicity, or resource control
- Pygame basics on game building
- How mulitprocessing works on python

Python MultiProcessing



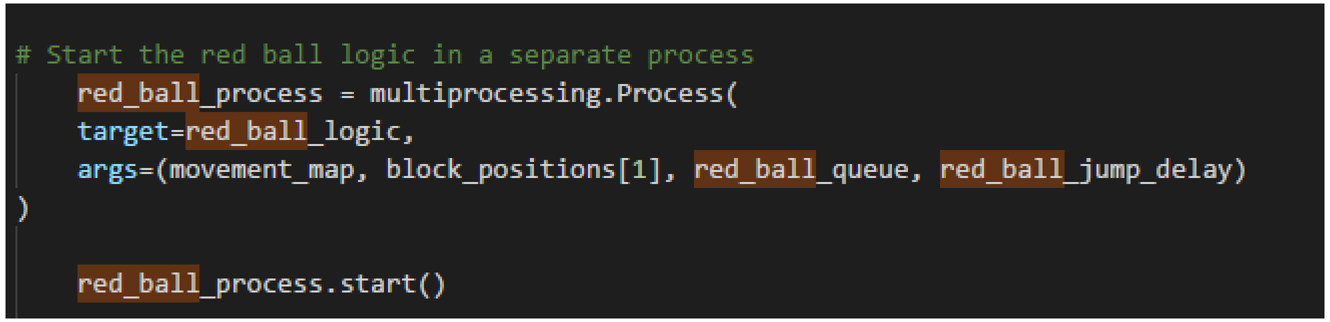
```python
# Define the red ball logic function
def red_ball_logic(movement_map, start_block, queue, jump_delay):
    red_ball_block = start_block
    last_jump_time = time.time()

    while True:
        current_time = time.time()

        # Time to move the ball?
        if current_time - last_jump_time >= jump_delay:
            last_jump_time = current_time
            possible_moves = [move for move in movement_map.get(red_ball_block, []) if move is not None]
            valid_moves = [move for move in possible_moves if move[1] > red_ball_block[1]]

            if valid_moves:
                red_ball_block = random.choice(valid_moves)

            # Check if it hit the bottom
            if red_ball_block[1] >= 510:
                red_ball_block = start_block


            # Send updated block back to main process
            queue.put(red_ball_block)

        time.sleep(0.01)  # Short sleep to reduce CPU load
```

Clear the CPU cache after program is done



```python
# --- Cleanup ---
sound_process.terminate()
sound_process.terminate()
red_ball_process.terminate()
red_ball_process.join()
saucer_process.terminate()
saucer_process.join()
pygame.quit()
sys.exit()
```

Make a method where the code for the red ball logic(target)And start it



```python
# Start the red ball logic in a separate process
    red_ball_process = multiprocessing.Process(
    target=red_ball_logic,
    args=(movement_map, block_positions[1], red_ball_queue, red_ball_jump_delay)
)

    red_ball_process.start()
```

Challenges Faced

- Finding a way for each team members machine to be able to load the sprites properly.
- Implementing features in a way that wouldn't break previously existing ones.
- Debugging through many lines of code
- Adding multiprocessing causing game to not run

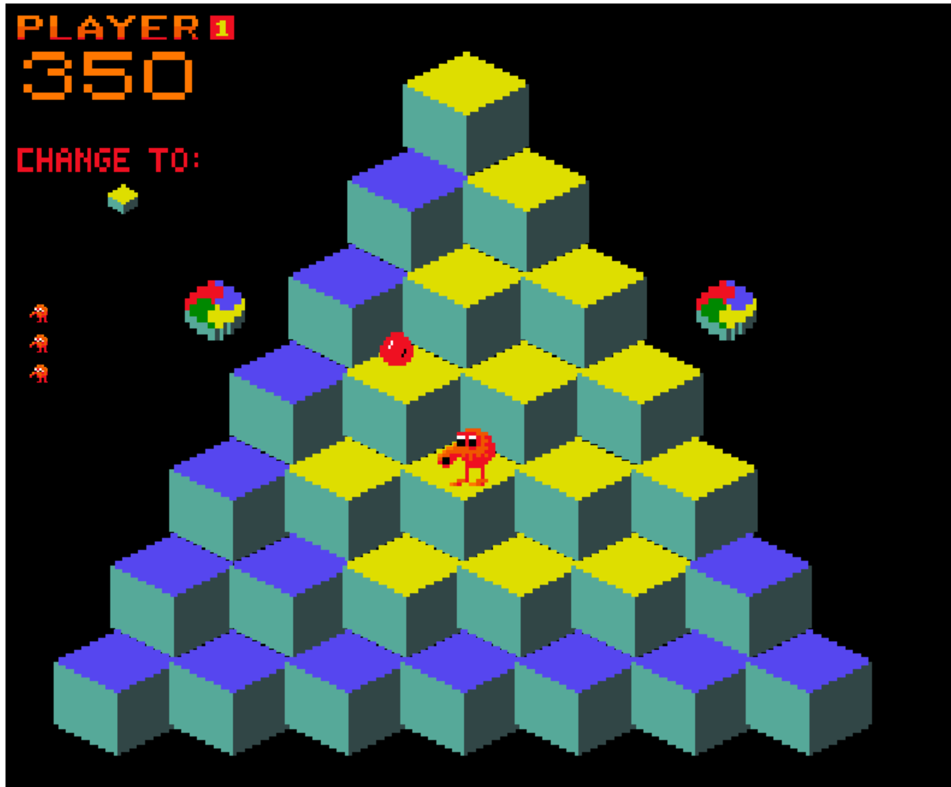
Results

•Achieved smoother gameplay and responsive controls

•Better CPU usage and workload distribution

•Foundation laid for scalable real-time games in Python

{Actual screenshot}



Ethics Reflection

Alies – We acknowledge that *Qbert** is an intellectual property owned by [rights holder, e.g., Sony Interactive Entertainment]. Our game does not seek to infringe upon or misappropriate any copyrighted assets, trademarks, or proprietary gameplay mechanics associated with *Qbert**.

Gianni - We seek to thoroughly understand the innerworkings of this timeless classic, and in doing so, further understand the capability of multi-processing in video games themselves. In our quest to understand the underlying innerworkings, we don’t seek profits nor mass distribution, we seek sole knowledge of the mechanisms behind the scenes of this classic title.

Will - By creating a game to which we know already exists, we seek to recreate it for the sake of understanding how the game works within a computer’s processing and *not* to profit or otherwise distribute the game. This game will be constructed purely for educational and demonstrations purposes.

GITUB - Code

- https://github.com/AliesKrepelka/qber-multiprocess.git

- Around 490 lines in total

Summary

We chose to build Q*bert from scratch with multiprocessing to deepen our understanding of both game architecture and parallel programming. By implementing multiprocessing, We were able to decouple various components of the game such as game logic, rendering, audio, and input handling allowing them to run in parallel. This approach not only improved performance but also provided a modular structure that’s easier to debug and scale. Building everything from the ground up gave us full control over the system design and taught us how to handle synchronization, inter-process communication, and state consistency in a real-time application.