Birds and Blocks: A 2-Player Pygame Challenge

Aadeshveer Singh
24B0926

April 29, 2025

Abstract

This report details the design, implementation, and features of "Birds and Blocks," a two-player physics-based game developed using Python and Pygame-CE. Inspired by Angry Birds, it adapts the core concept into a competitive, turn-based format. The players aim to destroy each other's fortresses using projectiles with distinct abilities. The game features custom-implemented projectile physics, destructible blocks (wood, stone, glass), a strategic bird upgrade system, persistent ELO-based player ratings with a leaderboard, and a complete set of custom pixel art assets and sound effects. The result is a polished, functional game showcasing concepts in game development, physics simulation, and software design.

Contents

1 Introduction

The mobile game Angry Birds captivated players with its simple but addictive physics-based puzzle game play. This project "Birds and Blocks," takes inspiration from this core mechanic but transforms it into a two-player turn-based competitive experience.

The primary objective was to develop a 1v1 fortress destruction game using Python and the Pygame Community Edition (`Pygame-CE`) library. Key requirements included implementing realistic projectile physics without relying on external physics engines, creating distinct projectile types with unique interactions against different block materials (wood, stone, glass), managing game states, handling player turns, and defining win/loss conditions.

Beyond the basic requirements, this project incorporates several advanced features: a card-based bird upgrade system unlocking special abilities, a persistent ELO rating system with a leaderboard, a dynamic camera, extensive custom pixel art and sound effects, and a contextual tutorial system. This report provides a comprehensive overview.

2 Modules Used

The development of "Birds and Blocks" relied on several standard Python libraries and the Pygame-CE library.

- **Pygame-CE:** The core game development library. Used for window creation, input handling, graphics rendering, audio playback, font management, and game loop timing. Essential for all interactive and multimedia aspects.
- **random:** Utilized for various non-deterministic elements including initial fortress block layout generation, particle effect variations, and background cloud placement.
- **math:** Essential for physics calculations (distance, velocity components), aiming trajectory prediction, and launch velocity scaling.
- **os:** Used for interacting with the file system, primarily for loading assets (`os.listdir`) and ensuring the existence of user data files (`os.mknod`).
- **sys:** Used specifically for properly terminating the application via `sys.exit()` when the quit event is detected, ensuring a clean exit, especially when running as a compiled executable. Details can be found in Appedix C.

Adhering to project guidelines, no external physics libraries (e.g., Pymunk) were utilized; all physics simulations were implemented using standard Python and the `math` library.

3 Directory Structure

The project follows a well-organized directory structure, separating code, assets, documentation, and persistent data:

```
.
+-- Birds_and_blocks(windows).exe  # Compiled Windows executable
+-- assets/                # Contains all non-code game resources
|   +-- audio/             # Sound effects (.wav files)
|   +-- fonts/             # Custom font file (custom_font.ttf)
|   +-- images/            # All visual assets (.png)
|       +-- UI/            # UI elements (buttons, screens, icons)
|       +-- background/    # Background scene images
|       |   ... (etc.)
|       +-- projectiles/   # Bird sprites (all types, states, upgrades)
+-- scripts/               # Core Python source code modules
|   +-- __pycache__/       # Python bytecode cache (auto-generated)
|   +-- birds.py           # Bird class (physics, abilities, collision)
|   |   ... (other .py files) ...
|   +-- utils.py           # Utility functions (Animation class)
+-- user_data/             # Stores persistent data between sessions
|   +-- players.txt        # Plain text list of registered player names
|   +-- ratings.txt        # Plain text list of corresponding ELO ratings
+-- report/                # Contains documentation files
|   +-- images/            # Screenshots used specifically in the report
|   |   +-- main_menu.png
|   |   |   ... (other report images) ...
|   |   +-- upgrade.png
|   +-- report.pdf         # Compiled PDF version of this report
|   +-- report.tex         # LaTeX source file for this report
|   +-- references.bib     # BibTeX bibliography database file
+-- main.py                # Main executable script (entry point)
+-- rename.bash            # Utility script (optional)
```

Listing 1: Final Project Directory Structure

This structure maintains a clear separation of concerns: operational game assets in `assets/`, Python logic in `scripts/`, persistent player data in `user_data/`, and all report-related files neatly contained within `report/`. The main script `main.py` serves as the entry point for running from source, while the `.exe` provides a standalone distribution. Note that the executable expects the `assets/` and `user_data/` directories to be present in its runtime directory.

4 Running Instructions

There are two primary ways to run the game: from the Python source code or using the pre-compiled Windows executable.

4.1 Prerequisites (Running from Source)

- Python 3.x installed.
- Pygame Community Edition (`pygame-ce`) installed. Use pip:

```
pip install Pygame-CE
```

4.2 Execution

4.2.1 Option 1: Running from Source

1. Open a terminal or command prompt.
2. Navigate (`cd`) to the project's root directory (where `main.py` is located).
3. Execute the main script:

```
python main.py
```

4.2.2 Option 2: Running the Executable (Windows)

1. Ensure the `assets/` and `user_data/` folders are present in the same directory as the executable file (`Birds_and_blocks(windows).exe`).
2. Double-click the `Birds and blocks(windows).exe` file to launch the game directly.[1]

The executable was created using PyInstaller, bundling the Python interpreter and necessary libraries.

4.3 Controls

Controls are the same regardless of the execution method:

Menu Navigation: Left-click on-screen buttons.

Name Input: Left-click player box, type name, press `Enter` or `Tab`.

Aiming/Firing: Left-click and drag bird in slingshot. Release to fire.

Special Abilities: Left-click mid-air (if bird upgraded ¿ Level 1).

Sound Toggle: Left-click speaker icon (top-left).

Back: Left-click back arrow icon to return to menu.(Disabled in tutorial)

Exit: Use window close button or detect `pygame.QUIT` event (handled internally via `sys.exit()`) to exit the application.

5 Implemented Features

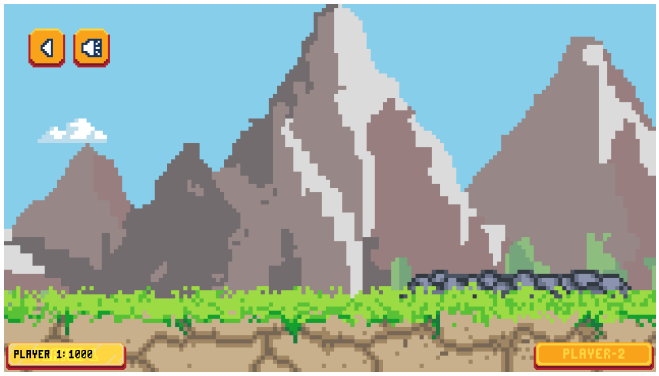
This project implements core mechanics and several advanced features.

5.1 Game Interface and Flow

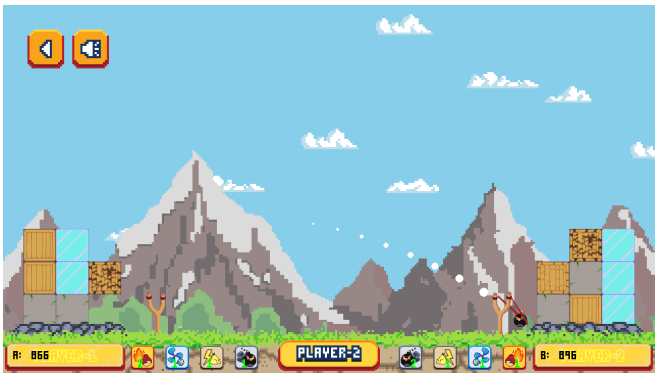
- **Screens:** Includes Main Menu, Player Name Entry, Gameplay, Tutorial overlays, Leaderboard, and Game Over (Figures 1a, 1b, 1c, 1d, 4, 5).
- **State Management:** Game flow controlled by state variables in `main.py`.
- **UI Elements:** Custom buttons, indicators (turn, upgrade levels), text rendering using custom fonts handled by classes in `scripts/modes.py`.
- **Turn Structure:** Alternating turns managed in the main game loop.

[1] **Note:** Some antivirus programs may flag executables created with PyInstaller as potentially suspicious due to the way they unpack components during runtime. This is often a false positive for applications compiled from trusted source code like this project.

(a) Main Menu



(b) Player Name Input



(c) Gameplay Screen

(d) Game Over Screen

Figure 1: Key UI Screens Overview

5.2 Core Gameplay Mechanics

- **Slingshot Physics:** Mouse drag calculates initial velocity based on distance and angle relative to the slingshot origin. Maximum speed is capped asymptotically (`scripts/birds.py`).

$$\vec{v} = v_{max}(1 - e^{-|\vec{r}|}) \cdot \hat{r}$$

  Where, v_{max} is set to 14 and $\vec{r}$ is the cursor position vector relative to the slingshot origin.

- **Trajectory Preview:** Dots predict the flight path based on current launch parameters (`Bird.render`).
- **Projectile Motion:** Custom physics loop in `Bird.calculate_next_pos` updates position using current velocity and applies gravity (`GRAVITY = 1/6` equivalent to 1 unit per second as FPS = 60).
- **Collision Detection:** Bird's hitbox corners are checked against the opponent's block grid. Damage applied on hit (`Bird.collision_check`).

5.3 Projectiles (Birds)

Four types implemented in `scripts/birds.py`: Red (balanced), yellow (versus wood), black (versus stone), and blue (versus glass). Damage calculation in `Bird.damage` includes base damage, speed factor, and type effectiveness multiplier.

$$\text{Damage} = (c_1 + c_2 \cdot |\vec{v}|) \cdot t \cdot u$$

Where c_1 and c_2 are constants for birds, $\vec{v}$ is the velocity vector and t is the type multiplier and u is the upgrade multiplier(`damage_factor`) decided based on bird level and use. $t = 1.5$ if type of bird is same as block it is about to hit, else $t = 0.7$.

[2]Red always induces $t = 0.7$ hence absolute damage of red is higher to compensate.

Projectile(Bird)	c_1	c_2	Type	Special ability
Red	35[2]	2	Basic	Triple damage
Blue	20	2	Glass	Splits into 3 identical projectiles
Chuck	20	2	Wood	Gets a speed boost
Bomb	20	2	Stone	Blasts to do Area damage

Table 1: Bird guide

5.4 Fortress (Blocks)

Managed by `scripts/blockmap.py`:

- **Types:** Wood, Stone, Glass with distinct health (`HEALTH_MAP`).

Block type	Health
Glass	55
Wood	75
Stone	95

- **Destruction:** HP reduced by bird impact damage.
- **Visual Damage:** 5-stage visual degradation based on remaining HP (Fig. 2).
- **Structure:** Grid-based; blocks currently float if support is removed.
- **Generation:** Randomly generated to introduce a new random element in every gameplay.

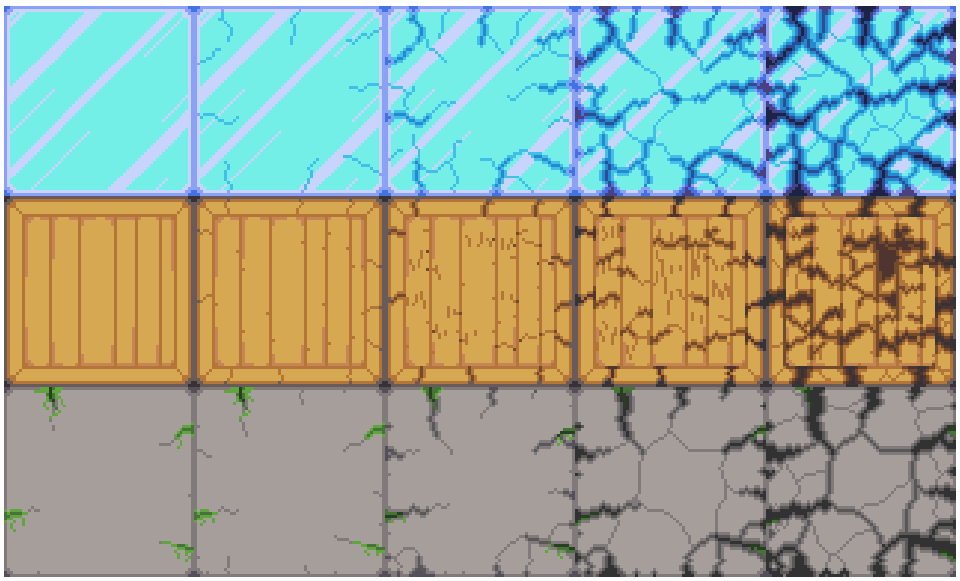


Figure 2: Visual Damage States for Blocks

5.5 Card & Upgrade System

Introduces strategic depth (`scripts/cards.py`, `scripts/player.py`):

- **Flow:** Players choose bird cards, then upgrade cards after exhausting birds.
- **Abilities (Level > 1):** Activated mid-flight via click:

  Red (Damage+)
  Yellow (Speed++)

Blue (Split)

Black (Explosive area damage)

Implemented in `Bird.calculate_next_pos`.

- **UI:** Upgrade levels shown via icons (Fig. 3).

Figure 3: Upgrade Indicators and Card Selection Example

5.6 ELO Rating System

Persistent ranking (`scripts/rating.py`):

- **Algorithm:** Standard ELO calculation (`ELO.update_rating`).

  Expected score of player a is give by:

$$E_a = \frac{1}{1 + 10^{(R_b - R_a)/400}}$$

  Expected score of player b is give by:

$$E_b = \frac{1}{1 + 10^{(R_b - R_a)/400}}$$

  Where R_a is rating of player a and R_b is rating of player b. After a match the winner scores 1 point and loser scores 0 points. This can be used to calculate new rating $R_{new} = R_{old} + 32(\text{Score} - \text{Expected score})$

- **Persistence:** Ratings stored in `user_data/` text files. Two different files are used to store rating as using a single file(like `csv`) might cause problems if the user name has some special character.
- **Integration:** Displayed in name input and game over screens. Players ELO rating can be accessed by simply playing and inputting the name, the name will automatically turn to format `<user_name>:<ELO>`. Top 9 players and their ratings can also be accessed through the leaderboard.

5.7 Visuals and Audio

Custom assets provide a unique identity:

- **Pixel Art:** All visuals created by hand `pixel by pixel` using Aseprite software.
- **Sounds:** Custom `.wav` effects for actions; background music loop.

- **Particles:** Effects for feathers, dust, shards (`scripts/particles.py`). See Appendix A for implementation details.
- **Dynamic Camera:** Zooms/pans to follow projectiles.
- **Animations:** Used for birds, UI, clouds (`Animation` class). Appendix B provides details on its implementation.

5.8 Dynamic Tutorial/Help

Contextual guidance (`scripts/modes.py`):

- **System:** Displays overlay images based on game state (`Tutorial` class).
- **Guidance:** Assists players through different phases (Fig. 4).

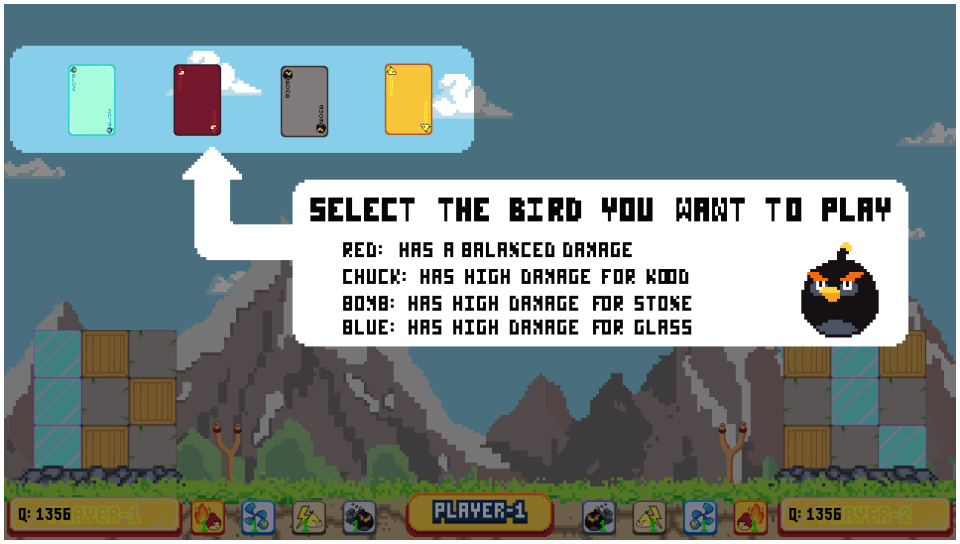


Figure 4: Dynamic Tutorial Overlay

5.9 Leaderboard

Displays player rankings:

- **Access:** Button on main menu.
- **Display:** Shows top players sorted by ELO (Fig. 5).

6 Project Journey

6.1 Learnings

Development involved learning Pygame-CE, game loop/state management, OOP design, custom physics simulation, asset management pipelines (Aseprite to Pygame), UI implementation, animation systems, and data persistence with file I/O for the ELO system.

6.2 Challenges

Major hurdles included tuning the custom physics and damage factors for intuitive gameplay, the extensive time required for custom pixel art creation, adding special visual effects efficiently, balancing the bird abilities and block health, managing complex state transitions smoothly, implementing a non-jarring dynamic camera, and ensuring the ELO system handled file operations robustly.

Figure 5: Leaderboard Screen

6.3 Solutions

These challenges were addressed through iterative testing and refinement cycles, a modular code structure enabled by OOP (using separate scripts), careful state definition and transition logic, extensive code commenting for clarity, using external tools like Aseprite efficiently for asset creation, and prioritizing core gameplay feel in physics implementation. Refactoring, like creating `loader.py`, also improved organization. Proper version management using Git and GitHub was employed throughout development. Finally, PyInstaller was used to package the game into a standalone executable for easier distribution on Windows.

7 Conclusion

"Birds and Blocks" successfully fulfills the project requirements by implementing a two-player physics-based game with custom mechanics using Pygame-CE. The inclusion of advanced features like the upgrade system, ELO ratings, leaderboard, dynamic camera, extensive custom assets, and an interactive tutorial significantly enhances the player experience, resulting in a polished and engaging final product. The project effectively demonstrates competence in various aspects of game development within the given constraints.

A Particle System Implementation Details

The game utilizes a particle system (`scripts/particles.py`) for various visual effects like feathers, dust, and block shards. This system consists of a `Particles` manager class that holds and updates a list of individual `Particle` objects. This appendix details the implementation of the core `Particle` class itself, which manages the state and movement of each visual effect element.

A.1 The `Particle` Class

This class represents a single visual effect particle, managing its state and movement.

A.1.1 Initialization (`__init__`)

A particle is created with specific attributes:

```
# From scripts/particles.py
class Particle:
    def __init__(self, animation, window_size, type, pos, vx, vy=1,
                 effects=None, idx=None):
        self.anim = animation.copy() # Independent animation state
        self.pos = list(pos)         # Initial position
        self.vx, self.vy = vx, vy    # Initial velocity
        self.effects = effects if effects else [] # Behaviour modifiers
        # ... other setup like initial frame ...
```

Listing 2: Particle Class __init__ Snippet

Key initialization aspects:

- Takes an `Animation` object for its visual frames.
- Sets initial position and velocity.
- The `effects` list (e.g., `'gravity'`, `'radial'`, `'sequence'`, `'cloud'`, `'fast'`) dictates behaviour modifications like applying gravity, randomizing velocity/position, controlling lifespan, or altering speed.

A.1.2 Update Logic (`update`)

Called each game frame to update the particle's state:

```
# From scripts/particles.py
def update(self):
    # Update position
    self.pos[0] += self.vx
    self.pos[1] += self.vy # + Potential 'float' effect variation

    # Apply effects like gravity
    if 'gravity' in self.effects:
        self.vy += 0.2 # Adjust gravity value as needed

    # Check lifespan: depends on effects
    if 'sequence' in self.effects:
        # Example: Remove if off-screen left (except clouds)
        finished = self.pos[0] < -self.anim.img().get_width()
        if finished and 'cloud' in self.effects:
            # Reset cloud position for looping effect
            # ... reset logic ...
            return False # Clouds don't "finish" this way
        return finished
    else:
        # Standard particles finish when their animation ends
        return self.anim.update()
```

Listing 3: Particle.update() Core Logic

The method updates position based on velocity, applies effects like gravity, and determines if the particle's lifecycle is complete (based on animation finishing or other conditions like going off-screen for sequence-based effects). It returns `True` when the particle should be removed.

A.1.3 Rendering (`render`)

Simply draws the particle's current animation frame at its position:

```
# From scripts/particles.py
def render(self, surf):
    surf.blit(self.anim.img(), self.pos)
```

Listing 4: Particle.render() Method

This system allows for flexible creation of various visual effects by combining different animations and behavioural flags.

B Animation Class Implementation

To handle multi-frame sprite animations consistently across different game objects (birds, blocks, UI elements, particles), a dedicated `Animation` class is implemented in `scripts/utils.py`. This class encapsulates animation data and logic, separating it from the game objects themselves.

B.1 Purpose and Usage

The primary goal is to manage a sequence of images (frames) and cycle through them over time based on a specified duration per frame. Game objects hold an instance of this class for their visual representation. Example usage when initializing a game object: `self.anim = self.game.assets['projectile']['basic']['idle'].`

B.2 Class Implementation

B.2.1 Initialization (`__init__`)

Sets up the animation instance:

```
# From scripts/utils.py
class Animation:
    def __init__(self, images, img_dur=5, loop=True):
        self.images = images        # List of pygame.Surface frames
        self.img_dur = img_dur      # How many game frames each image lasts
        self.loop = loop            # Whether the animation repeats
        self.frame = 0              # Internal counter tracking progress
        self.done = False           # Flag for non-looping animations
        self.length = len(images)   # Number of images in the sequence
```

Listing 5: Animation Class __init__ Snippet

B.2.2 Updating State (`update`)

This method is called once per game frame to advance the animation:

```
# From scripts/utils.py
def update(self):
    if self.loop:
        # Increment frame counter and loop using modulo
        self.frame = (self.frame + 1) % (self.length * self.img_dur)
    else:
        # Increment frame counter only if not done
        if not self.done:
            self.frame += 1
            # Check if end of animation reached
            if self.frame >= self.length * self.img_dur:
                self.done = True
```

```
    # Return True if a non-looping animation has finished
    return self.done
```

Listing 6: Animation.update() Logic

It increments an internal frame counter. If looping, it wraps around using the modulo operator. If not looping, it stops incrementing and sets the `done` flag when the end is reached. It returns the `done` status, useful for triggering events upon animation completion (like removing finished particles).

B.2.3 Getting Current Image (`img`)

Calculates and returns the correct image surface for the current frame:

```
# From scripts/utils.py
def img(self):
    # Calculate the index into the images list
    image_index = int(self.frame / self.img_dur)
    return self.images[image_index]
```

Listing 7: Animation.img() Method

This method is called by the game object's render method to draw the appropriate frame.

B.2.4 Utility Methods

- `copy()`: Creates a new instance of the `Animation` with the same images and settings but with its own independent `frame` counter and `done` status. This is crucial so that multiple game objects using the same animation sequence (e.g., multiple identical particles) animate independently.
- `set_frame(frame)`: Allows manually setting the internal `frame` counter, useful for synchronizing animations or resetting them to a specific state (e.g., setting block damage appearance).
- `get_frame()`: Returns the current value of the internal `frame` counter.

This class provides a simple yet effective mechanism for managing sprite animations throughout the "Birds and Blocks" project.

C Application Exit Handling

Proper application termination, especially for compiled executables, is handled using the `sys` module. The event loop checks for the `pygame.QUIT` event (typically triggered by closing the window) and calls `sys.exit()`.

```
# Inside the main event loop (in game.run)
import sys # Ensure sys is imported at the top of main.py

# ... inside the 'for event in pygame.event.get():' loop ...
if event.type == pygame.QUIT:
    pygame.quit() # Uninitialize pygame modules
    sys.exit()    # Terminate the Python process cleanly
    # return # Or use return if in an async function context
# ... rest of event handling ...
```

Listing 8: Handling Quit Event in main.py

Using `pygame.quit()` first is recommended to allow Pygame to clean up its resources before the script exits via `sys.exit()`.

Bibliography / References / Resources Used

- Pygame Community Edition Official Website and Documentation: https://pyga.me/
- Python 3 Documentation: https://docs.python.org/3/
- Aseprite (Pixel Art Software): https://www.aseprite.org/ (Used for asset creation)
- Pixabay (Free stock sounds): https://pixabay.com/ (Used to find sound effects)
- ELO Rating System (Conceptual understanding): https://en.wikipedia.org/wiki/Elo_rating_system
- Original Game Inspiration: Angry Birds 2 - Official Gameplay Trailer https://www.youtube.com/watch?v=17GtzVBX31o
- Beginner Pygame tutorial by FreeCodeCamp https://www.youtube.com/watch?v=FfWpgLFMI7w
- Advanced Pygame tutorial by DaFluffyPotato https://www.youtube.com/watch?v=2gABYM5M0ww&t=20017s
- Pygame CE vs Pygame Video: https://www.youtube.com/watch?v=FZ1d3tDEitQ