Breakout! Python Clone
Developer Tip Sheet

Contents

Introduction

This developer tip sheet provides concise guidance for working with the UMGC Group 3 project – Breakout Game in Pycharm. It is designed to assist current and future developers in understanding key components of the application, promoting consistency in coding practices, and avoiding common pitfalls.

GitHub Repository: https://github.com/JMicallef97/CMSC-495---Breakout

Core Architecture Overview

GameState:

Role: Base class for all game states (menus, screens, gameplay).
Use: Inherit from GameState to create new scenes or menus.
Tip: Keep game state classes focused on UI and flow control. Logic should stay in the GameManager.

Related Classes:

- GameView (wraps and renders current GameState)
- All state classes (MainMenuState, AdjustDifficultyState, GameOverState, etc.)

Caution / Common Pitfalls:

- Do not update gameplay logic (like score or ball position) from within a state class.
- Avoid tightly coupling state transitions inside each state — instead, use GameView.show_view() for modularity.

GameView:

Role: The central Arcade View wrapper that hosts different GameState screens.
Use: Only one GameView is needed; it swaps current_state as screens change.
Tip: Extend on_show_view() to add fade-in transitions or music cues between screens.

Related Classes:

- GameState and its children (passed into GameView)
- GameStateManager (may orchestrate transitions)

Caution / Common Pitfalls:

- Avoid duplicating game state logic in GameView. It should delegate everything to the current state.
- Forgetting to call self.current_state.update() and draw() in GameView can break the render loop.

GameManager:

Role: Core gameplay controller. Manages paddle/ball logic, collision, scoring, pause, and quit behavior.
Use: Modify this class when adding mechanics like power-ups, ball physics, or gameplay timers.
Tip: Keep logic in update() and visuals in draw() for clean separation.

Related Classes:

- Paddle, Ball, Brick (core objects it manages)
- InputManager (used to read real-time input)
- GameData (used to apply difficulty settings)

Caution / Common Pitfalls:

- Overloading update() with too many responsibilities — consider breaking out helper methods.
- Accidentally allowing state changes (e.g., pause or quit) to skip frame-based checks can cause instability or double-exits.

InputManager:

Role: Centralized tracker for all key and mouse inputs.
Use: Allows multiple parts of the game to react to input without conflicts.
Tip: Extend it to support complex input combinations, input cooldowns, or rebinding keys.

Related Classes:

- GameManager (reads inputs each frame)
- GameView and any state that handles custom inputs

Caution / Common Pitfalls:

- Forgetting to reset keys like 'P' or 'Q' after toggles can cause repeated actions or stuck states.
- Not differentiating between key down and key held can cause unintentional rapid toggles.

GameData:

Role: Stores persistent settings like difficulty values.
Use: Modify to include any values that need to be accessed globally across views or states.
Tip: Extend it for storing settings like sound preferences or game mode options.

Related Classes:

- Paddle, Ball, GameManager (read settings from here)
- Any state menu that allows the player to set difficulty

Caution / Common Pitfalls:

- Avoid storing runtime data (e.g., score, lives) here — keep it focused on static or config-like data.
- Ensure the GameData instance is stateless or reset appropriately if reused in future sessions or replays.

Adding New Game States

To create a new screen (e.g., Settings, Credits, Story):

1. Inherit from GameState.
2. Define your layout and logic using on_draw(), on_key_press(), and similar methods.
3. Use GameView(GameStateInstance) to transition views.

Example: See MainMenuState.py or AdjustDifficultyState.py for reference structure.

Expanding Gameplay

Suggested Features:

- Power-ups: Add falling power-up sprites after brick collisions.
- Multiball Mode: Launch multiple balls after a condition is met.
- Challenge Mode: Add a timer with arcade.get_time() to create a time-attack mode.

Tip: Handle new game rules inside GameManager.update() or via flags passed in through GameData.

Graphics & Visual Assets

Assets are stored in the /Graphics folder.

To add a new theme:

1. Drop new textures into the folder (e.g., paddle, ball, brick).
2. Adjust arcade.load_texture() calls in the relevant class files.

Tip: Standardize filenames to avoid breaking paths (e.g., Ball_Hard.png, Paddle_Classic.png).

Text Rendering Best Practices

Use arcade.draw_text() with anchor_x="center" for centered messages.
For multi-line messages, use two separate draw_text() calls with adjusted y-coordinates.

Example:

```
arcade.draw_text("PAUSED", x, y + 16, color, 20, anchor_x="center")
arcade.draw_text("Press P to resume", x, y - 12, color, 16, anchor_x="center")
```

Tip: To position text cleanly in the center of the lower screen area, compute the midpoint between paddle and brick rows.

High Score System

- Scores are saved to highscores.txt at game over.
- Format includes score and timestamp.

You can extend this to support:
- Player initials
- Difficulty level used
- Sort or filter logic on load

Unit Testing

Tests are stored in /Unit_Tests/.

Includes coverage for:
- Ball movement and reflection
- Paddle behavior
- Brick destruction
- Game manager logic

Tip: Use pytest to run all tests. Add new tests when you introduce new gameplay logic or bug fixes.

Maintenance Best Practices

- Use consistent docstring formatting across all classes and methods.
- Store shared constants in GameData or a new Constants.py.
- Commit often, with descriptive messages that document both fixes and feature additions.
- Keep game logic modular — UI states in GameState, gameplay in GameManager.