MỤC LỤC

[**1. Tối ưu hóa CPU 2**](#_30j0zll)

[**2. Tối ưu hóa GPU 5**](#_1fob9te)

[**3. Tối ưu hóa bộ nhớ (RAM & VRAM) 6**](#_3znysh7)

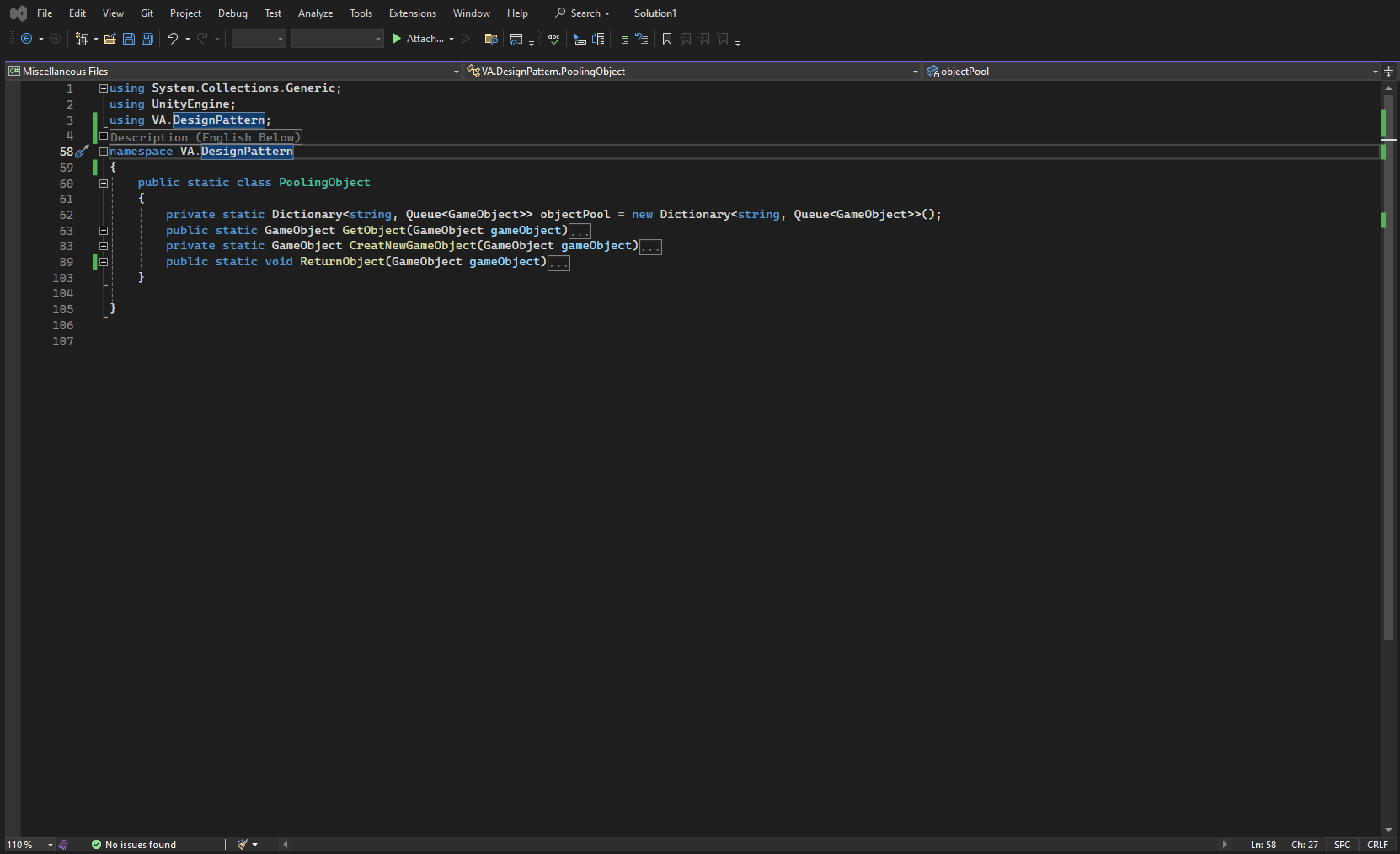
[**4. ARN/Crash 7**](#_tyjcwt)

[**5. Tối ưu bộ nhớ vật lý 9**](#_1t3h5sf)

Tối ưu hiệu năng trong Unity là một chủ đề rộng, bắt đầu bằng cách tập trung vào các yếu tố chính sau:

# Tối ưu hóa CPU

* Tránh sử dụng Update() ở mọi script:
  + Nguyên nhân:
    - * Update được gọi mỗi frame (60 lần/giây nếu máy chạy ở 60 FPS), càng nhiều hàm update thì CPU sẽ phải duyệt qua tất cả các hàm update. Mỗi lần gọi một hàm update, unity sẽ thực hiện context switching giữa nhiều script (gây tiêu tốn tài nguyên của CPU). Các hàm update cũng là chạy song song với nhau, khó để theo dõi hàm nào đang gây tiêu tốn tài nguyên
    - context switching: mỗi lần Update của một script được gọi, CPU phải thực hiện quy trình Chuyển quyền xử lý sang Script đó (nạp bộ nhớ, lấy ttin hàm cần thực hiện, việc này khá tốn tài nguyên của CPU) -> thực thị logic trong update -> quay lại xử lý các script khác trong vòng lặp game.
  + Các khắc phục:
    - Dùng sự kiện (Event, Coroutine) thay vì kiểm tra liên tục.
    - Gộp các Update() vào một class quản lý duy nhất (ví dụ: dùng System.Action để quản lý callback).
* Giảm Draw Calls bằng Batching
  + Nguyên nhân:
    - Draw call là một lệnh CPU gửi đến GPU để yêu cầu render ra 1 đối tượng 3D trên màn hình. Một đối tượng có nhiều material khác nhau, mesh khác nhau thì sẽ tạo ra càng nhiều draw call
  + Cách khắc phục
    - Dùng **Static Batching** cho object tĩnh (Static checkbox).
    - Dùng **Dynamic Batching** cho object nhỏ (< 300 vertices).
    - Dùng **GPU Instancing** cho object có cùng material để giảm draw calls.
  + Hint:
    - **Static Batching** cho môi trường (tường, sàn, cây tĩnh).
    - **Dynamic Batching** nếu có nhiều object nhỏ di chuyển.
    - **GPU Instancing** cho object lặp lại như NPC, viên gạch.
* Object Pooling để tái sử dụng GameObject
  + Nguyên nhân:
    - Khi bạn liên tục tạo và hủy object bằng Instantiate() và Destroy(). **Tốn CPU** để khởi tạo và thiết lập object mớ. **Tạo rác bộ nhớ** → Unity sẽ kích hoạt Garbage Collector **(GC)** để dọn dẹp, GC sẽ gây giật lag khi chạy.
  + Cách khắc phục:
    - Tránh Instantiate() và Destroy() liên tục.
    - Dùng **Object Pooling**, tạo sẵn object và bật/tắt khi cần. (Đã có Pooling Object trong dự án, đọc mô tả trong script để biết thêm chi tiết)



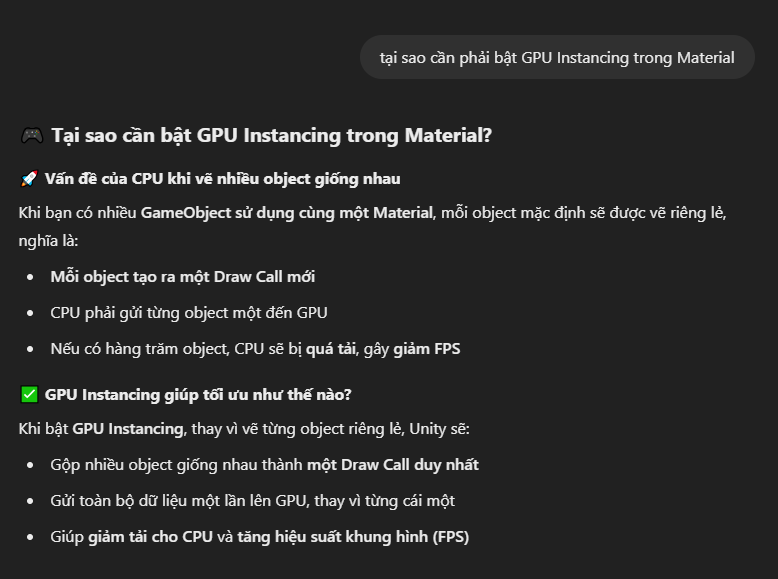
* + Unity có OnDisable() và OnEnable() để quản lý tái sử dụng object.
* Tối ưu hóa vật lý (Physics Optimization)
  + Nguyên nhân:
    - Mỗi khi hai Collider tương tác, hệ thống vật lý phải kiểm tra xem chúng có va chạm không. Nếu không kiểm soát, Unity sẽ phải kiểm tra tất cả các Layer với nhau, gây quá tải CPU.
    - **FixedUpdate()** là nơi Unity xử lý hệ thống vật lý, nhưng nó **không chạy theo FPS**, mà chạy theo **Time.fixedDeltaTime** (mặc định là 0.02s, tức 50 lần/giây).
  + Giải pháp:
    - Điều chỉnh **Collision Matrix** để tránh tính toán va chạm không cần thiết.
    - Giảm tần suất FixedUpdate() nếu không cần cập nhật vật lý quá nhanh.
    - Dùng **Rigidbody Interpolation** thay vì tăng tần suất cập nhật vật lý.
* Giảm GC Alloc (Garbage Collection)
  + Nguyên nhân:
    - Trong Unity, mỗi khi bạn **cấp phát bộ nhớ mới trên heap**, nó sẽ tạo **GC Alloc**. Khi lượng bộ nhớ này vượt quá giới hạn, Unity **tạm dừng game** để dọn rác (**GC collection**), gây ra giật lag.
    - Những nguyên nhân phổ biến gây GC Alloc cao:
      * Tạo đối tượng mới liên tục trong Update()
      * Sử dụng string và string concatenation (+)
      * **Dùng Linq hoặc foreach** (có thể tạo bộ nhớ tạm)
      * Dùng Instantiate / Destroy liên tục thay vì Object Pooling
      * Dùng GetComponent, Find, SendMessage quá nhiều
  + Cách khắc phục:
    - Hạn chế dùng new trong Update(), thay vào đó dùng object tái sử dụng.
    - Dùng List<T>.Clear() thay vì tạo list mới.
    - Dùng StringBuilder thay vì nối chuỗi (+ hoặc string.Format), hoặc tối ưu hơn nữa, có thể dùng TextMeshPro do TMP có tối ưu riêng.
* HINT:
  + Có thể mở **Profiler** (Window > Analysis > Profiler) và xem CPU Usage, tập trung vào Rendering và Script

# Tối ưu hóa GPU

* Giảm Draw Calls bằng Batching:
  + Đã nói ở phần trên
  + Cách kiểm tra draw calls:
    - **Mở Stats** (trên Game View) → Xem “Batches” (số lượng draw call)
    - **Dùng Frame Debugger** (Window → Analysis → Frame Debugger) để xem chi tiết cách Unity vẽ từng object.
* Giảm sử dụng shader quá nặng:
  + Các loại shader khác nhau sẽ tốn tài nguyên khác nhau
  + Tối ưu shader bằng cách:
    - **Dùng Mobile Shader** thay vì Standard Shader
    - Giảm số lượng **texture sample** trong Shader
    - **Dùng Unlit Shader** cho đối tượng không cần ánh sáng.
* Tối ưu hệ thống LOD (Level of Detail) (Khi ở xa, không cần hiển thị chi tiết)
  + Dùng LOD Group (component LOD Group trong unity)
  + Dùng Impostor Texture (chỉ hiển thị ảnh tĩnh khi vật thể ở rất xa).
* Nén ảnh:
  + Các định dạng nén phổ biến: ASTC, ETC

# Tối ưu hóa bộ nhớ (RAM & VRAM)

* Mô tả:
  + Bộ nhớ trong game gồm hai phần chính là RAM(Chứa dữ liệu game như logic, script, animation, physics, AI, v.v) và VRAM (Chứa dữ liệu liên quan đến đồ họa như texture, mesh, shader, v.v.)
* Cách khắc phục:
  + Tránh cấp phát động trong vòng lặp
  + Tái sử dụng các cấu trúc cấp phát động trong vòng lặp
  + dùng struct thay thế class nếu có thể
  + Dùng pooling thay vì instantiate và destroy để tránh garbage collector hoạt động quá múc cần thiết
  + Dùng Animation Compression
  + Dùng sprite atlas
  + Chọn Material → Tick vào **Enable GPU Instancing**

****

# ARN/Crash

* Nguyên nhân:
  + **Thiếu quyền hoặc xung đột giữa các Android API do khi game yêu cầu một quyền nào đó (ví dụ: WRITE\_EXTERNAL\_STORAGE, ACCESS\_FINE\_LOCATION), nhưng quyền chưa được cấp hoặc API bị thay đổi trên các phiên bản Android mới**
    - API là gì và có ảnh hưởng đến game như thế nào
      * Quyền truy cập (Permissions), Khi game cần sử dụng **Camera, Microphone, GPS, Storage**, Android yêu cầu người dùng cấp quyền thông qua API
      * Nếu game gọi một API nhưng không có quyền, sẽ xảy ra lỗi và có thể dẫn đến **ARN Crash**
      * Sự khác biệt giữa các phiên bản Android
      * Plugin hoặc SDK của bên thứ ba lỗi thời
      * Threading sai cách
      * Do API gốc của Android
  + **Lỗi liên quan đến Garbage Collection & Memory Leak**
    - Mô tả: Lỗi **"App Not Responding" (ARN Crash)** có thể xảy ra do **Garbage Collection (GC) và Memory Leak** làm tràn bộ nhớ RAM, khiến Android **kill process của game** để giải phóng tài nguyên.
    - Dấu hiệu:
      * Game chạy **mượt lúc đầu**, nhưng sau một thời gian **bị giật lag và crash**.
      * Khi kiểm tra bằng **Android Logcat**, thấy lỗi kiểu: "Foreground process has exceeded memory limit, killing: com.yourgame"
      * **Heap size tăng liên tục**, nhưng GC không thể giải phóng bộ nhớ kịp thời
    - Nguyên nhân:
      * Tạo quá nhiều object mới thay vì tái sử dụng
      * String Concatenation (Chuỗi gây nhiều Allocation) (Mỗi lần nối chuỗi bằng +, Unity **tạo object mới trên Heap** (Vì string là immutable). Nên dùng stringbuilder hoặc …)
      * Không xóa event listener gây rò rỉ bộ nhớ: Nếu đăng ký event nhưng **không hủy đăng ký khi object bị hủy**, GC sẽ **không thể dọn object**, dẫn đến **memory leak**.
      * Object Pooling
    - Debug ARN Crash do bộ nhớ trên android
      * Dùng Profiler để kiểm tra RAM & GC:
      * 1. **Mở Unity Profiler** (Window > Analysis > Profiler)
      * 2. **Chọn "Memory" & "CPU Usage"** để xem bộ nhớ có tăng liên tục không
      * 3. Chạy game lâu & quan sát GC Alloc
  + **Sử dụng Multi-threading không đúng cách**
    - Việc sử dụng multi threading nên cân nhắc kỹ, nếu game quá đơn giản (casual, hyper-casual, minigames, …) thì việc sử dụng multi threading là không cần thiết, vì nó có thể gây phức tạp hóa bài toán.
    - Sử dụng multi threading giúp tối ưu hiệu suất trong các trường hợp như:
      * Tải dữ liệu trong nền (file, sever, asset bundle,...)
      * Xử lý AI phức tạp
      * Xử lý mô phỏng nặng (các game có sử dụng vật lý nhiều)
      * Xử lý đồ họa không cần real-time (Mesh generation, Procedural Content, v.v.)
  + **Lỗi do xử lý Texture & Shader**
    - **Nguyên nhân:**
      * Dùng Texture có độ phân giải quá lớn
      * Không nén Texture
      * Không dùng Sprite Atlas cho UI
      * Sử dụng Shader quá nặng trên Mobile
    - Cách khắc phục:
      * Giảm kích thước texture về 512x512 hoặc 1024x1024, Texture format thì ASTC hoặc ETC2 sẽ là 2 cái tối ưu nhất trong các format khác cho game của tôi
      * Sử dụng Texture Atlas để gộp nhiều texture nhỏ vào một tấm lớn
      * Bật MipMaps nếu texture được dùng trên vật thể có khoảng cách thay đổi
      * Dùng Unlit Shader nếu không cần lighting
      * Dùng Mobile Shader hoặc Shader Graph với Simple Shader
      * Giảm bớt tính toán Per-Pixel, thay bằng Per-Vertex nếu có thể
      * Bật **GPU Instancing** trong **Material Settings** để giảm draw call với các gameobject có dùng chung 1 material, giảm draw call
      * Dùng **Addressables hoặc AssetBundle** để load texture bất đồng bộ

# Tối ưu bộ nhớ vật lý

* **Tại sao cần tối ưu dung lượng game?**
  + **Giảm thời gian tải game: File nhỏ hơn → tải nhanh hơn**
  + **Tiết kiệm bộ nhớ: Tránh chiếm quá nhiều storage của người chơi**
  + **Giảm thời gian cài đặt: Đặc biệt trên Android (APK size càng lớn, cài càng chậm)**
  + **Tăng tỷ lệ giữ chân người chơi: Nếu game quá nặng, người chơi sẽ xóa để giải phóng dung lượng**