# 1. [Design Pattern – Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)

# 2. Observer Pattern

## Tại sao lại sử dụng ObserverPattern :

Trong game ta thường có một số đối tượng cơ bản sau:

– Main Character (MC)

– UI : hiển thị điểm số, máu … của MC

– AchivementManager : theo dõi người chơi, tặng những điểm thưởng như : quà khi thực hiện được một tripple kill, nhặt được những item hiếm …

– ..v.v…

Khi người chơi bị mất máu hoặc kill được quái vật liên tiếp, trên UI sẽ phải hiển thị số máu của MC, cái Achivement cần theo dõi để tính điểm thưởng, mọi người sẽ code thế nào? Cho cái đám UI, Achivement … theo Singleton hết, rồi gọi trực tiếp tới UI, Achivement… để thông báo về việc thay đổi máu, điểm … ? Điều này bất tiện và khiến code n bị coupling quá nhiều, các đối tượng trong game dần dần sẽ trở nên “kết dính” với nhau, code không có khả năng sử dụng lại, lỡ mà object nào bị thiếu thì game crash, null pointer ….. khổ sở lắm.

Thay vào đó, áp dụng ObserverPattern, người ta sẽ làm như sau:

– Định nghĩa các events trong game: OnPlayerHPChanged, OnPlayerKillMonster …

– Tạo EventDispatcher: đối tượng trung gian, tiếp nhận và chuyển tiếp các events.

– Khi bắt đầu một scene (hàm Start() đc gọi), đối tượng UI và Achivement … sẽ Register với EventDispatcer để lắng nghe các sự kiện OnPlayerHPChanged, OnPlayerKillMonster.

– MC : khi bị mất máu hoặc giết một con quái vật, ghi điểm …. MC sẽ gửi một message tới EventDispatcher, EventDispatcher thông báo cho những object có đăng kí lắng nghe các event. (Ở đây là UI object và AchivementManager)

Tổng kết là 4 bước:

* Viết EventDispatcher, nhận và chuyển tiếp thông tin (trong demo của mình có rồi, dùng lại cho lẹ, không cần viết lại đâu =))) )
* Định nghĩa các Event trong game, viết tập trung lại một file (EventID chẳng hạn)
* Đăng kí lắng nghe event (RegisterListener) cho các object, xử lý khi nhận đc event
* Thông báo event (PostEvent).

# 3. Command pattern

<https://viblo.asia/p/ap-dung-command-pattern-trong-unity-jvEla4o4Zkw>

Nhự vậy khi chúng ta nhấn phím Space thì phương thức Jump() sẽ được gọi, và nhân vật sẽ nhảy lên. Tuy nhiên, bài toán tiếp tục đặt ra yêu cầu đó là, người chơi có thể thay đổi nút bấm để thực hiện hành động nhảy thay vì nút Space trên bàn phím, ví dụ như bấm phím J.

Lúc này chúng ta không thể code thuần túy như trên được, chúng ta sẽ cần sử dụng Command pattern để dễ dàng cho việc User muốn thay gì thì thay :v

<https://giniwebseo.vn/command-pattern-va-ung-dung-trong-unity-phan-1/>

# ~~Flyweight Pattern~~

<https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/>

Flyweight Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ chúng hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp.

Flyweight Pattern được sử dụng khi chúng ta cần tạo một số lượng lớn các đối tượng của 1 lớp nào đó. Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp. Flyweight Pattern có thể được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy performance của hệ thống được tối ưu.

<https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/>

Used Total: 438.6 MB Unity: 119.1 MB Mono: 16.2 MB GfxDriver: 18.3 MB Audio: 1.3 MB Video: 0 B Profiler: 283.8 MB

Reserved Total: 0.62 GB Unity: 298.4 MB Mono: 16.7 MB GfxDriver: 18.3 MB Audio: 1.3 MB Video: 0 B Profiler: 298.0 MB

Total System Memory Usage: 1.35 GB

Textures: 1184 / 42.6 MB

Meshes: 102 / 2.2 MB

Materials: 74 / 127.0 KB

AnimationClips: 8 / 43.0 KB

AudioClips: 0 / 0 B

Total: 451.1 MB Unity: 113.2 MB Mono: 15.6 MB GfxDriver: 18.3 MB Audio: 1.3 MB Video: 0 B Profiler: 302.7 MB

Reserved Total: 0.61 GB Unity: 298.4 MB Mono: 18.1 MB GfxDriver: 18.3 MB Audio: 1.3 MB Video: 0 B Profiler: 284.0 MB

Total System Memory Usage: 1.34 GB

Textures: 1183 / 42.3 MB

Meshes: 102 / 2.2 MB

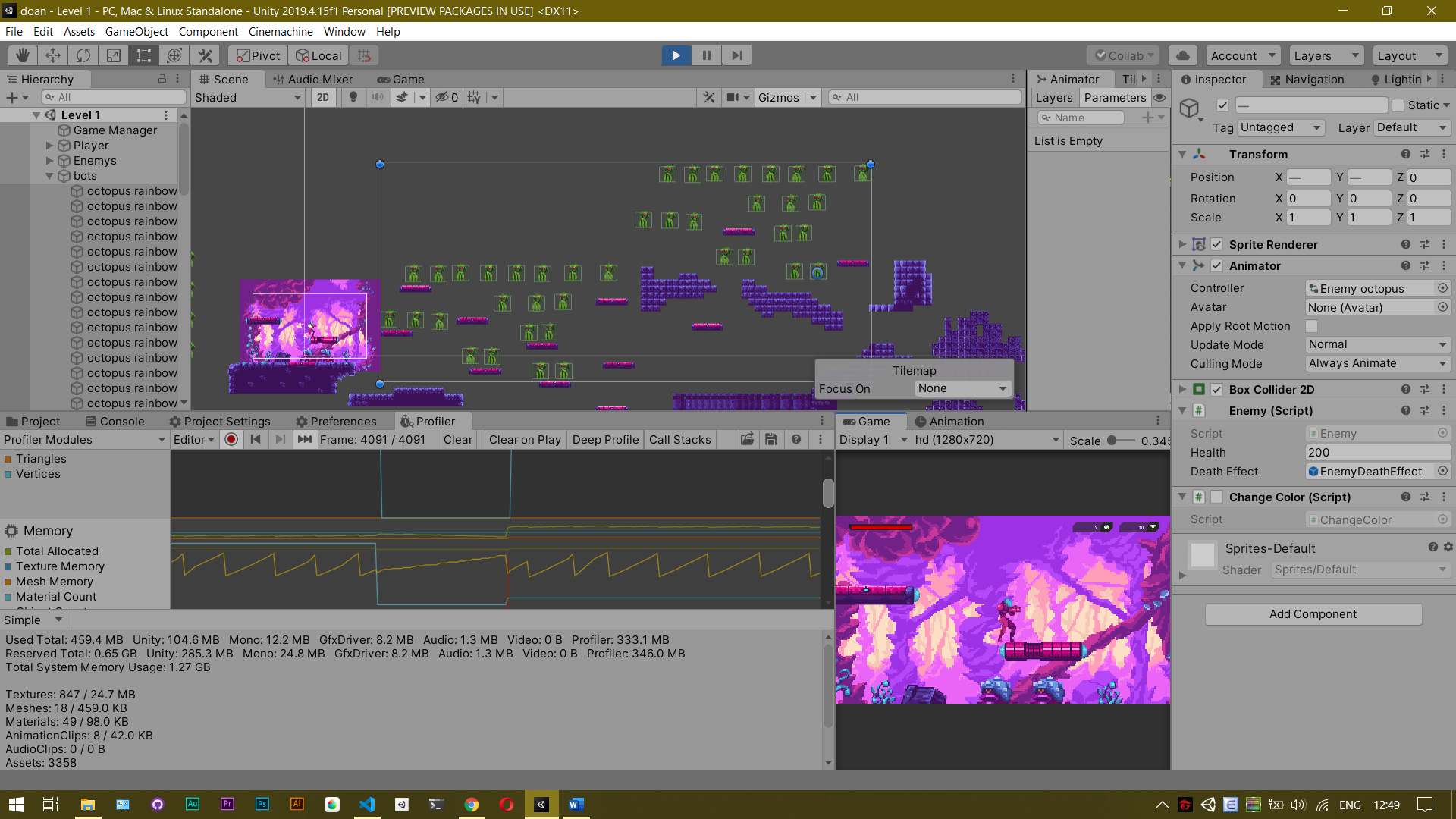
Materials: 114 / 165.0 KB

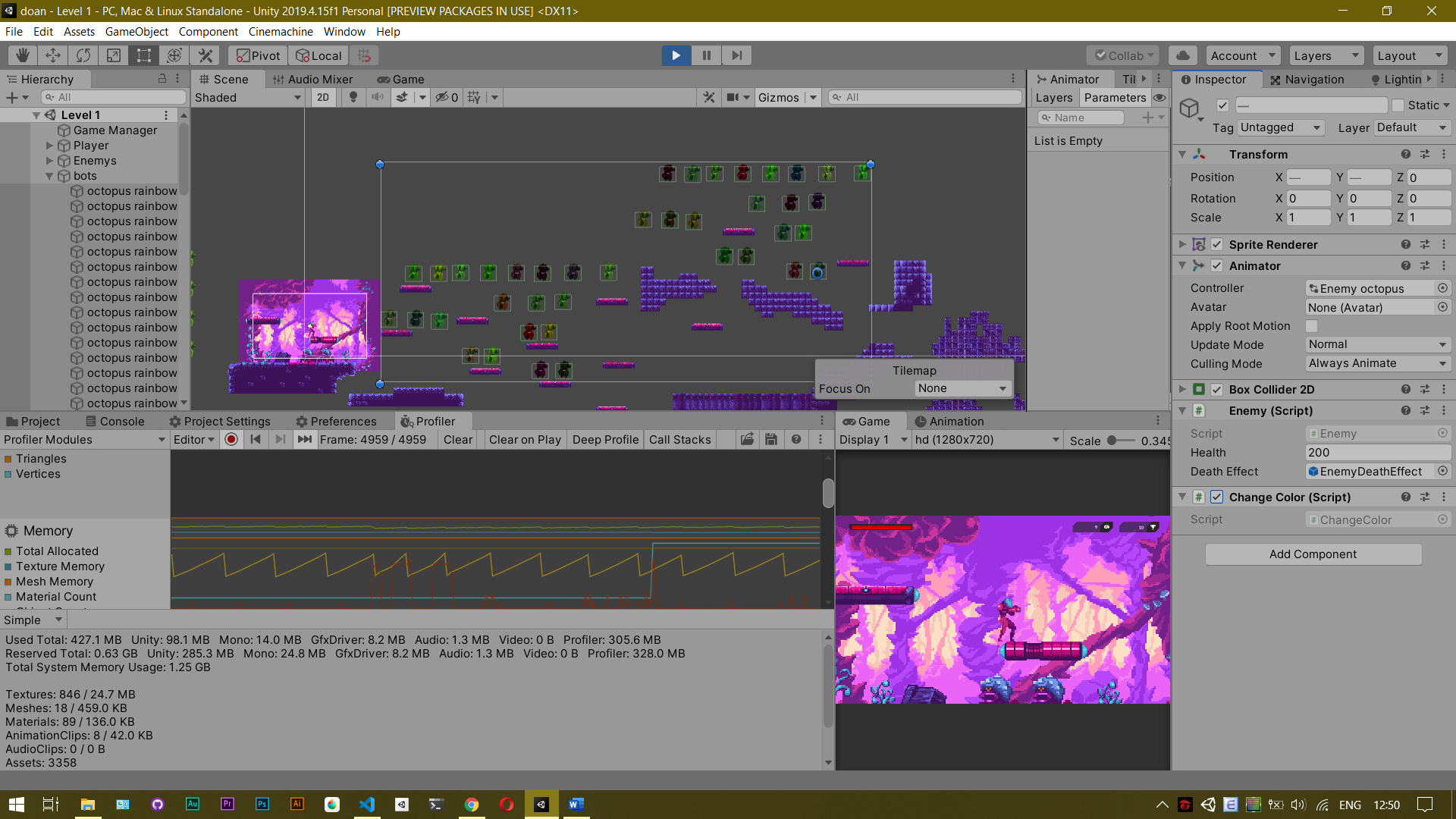
AnimationClips: 8 / 43.0 KB

AudioClips: 0 / 0 B

Assets: 3693

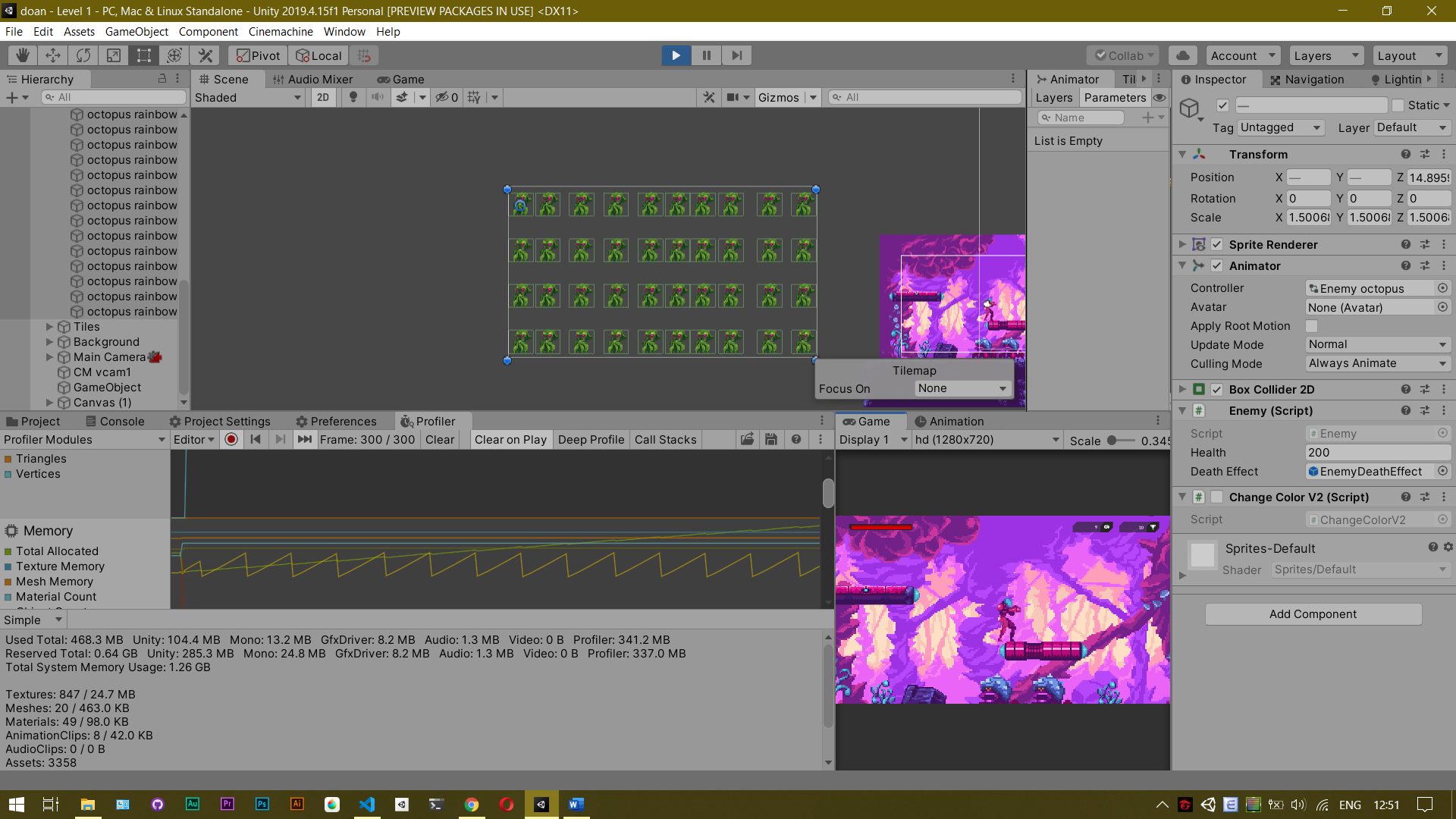
Truoc

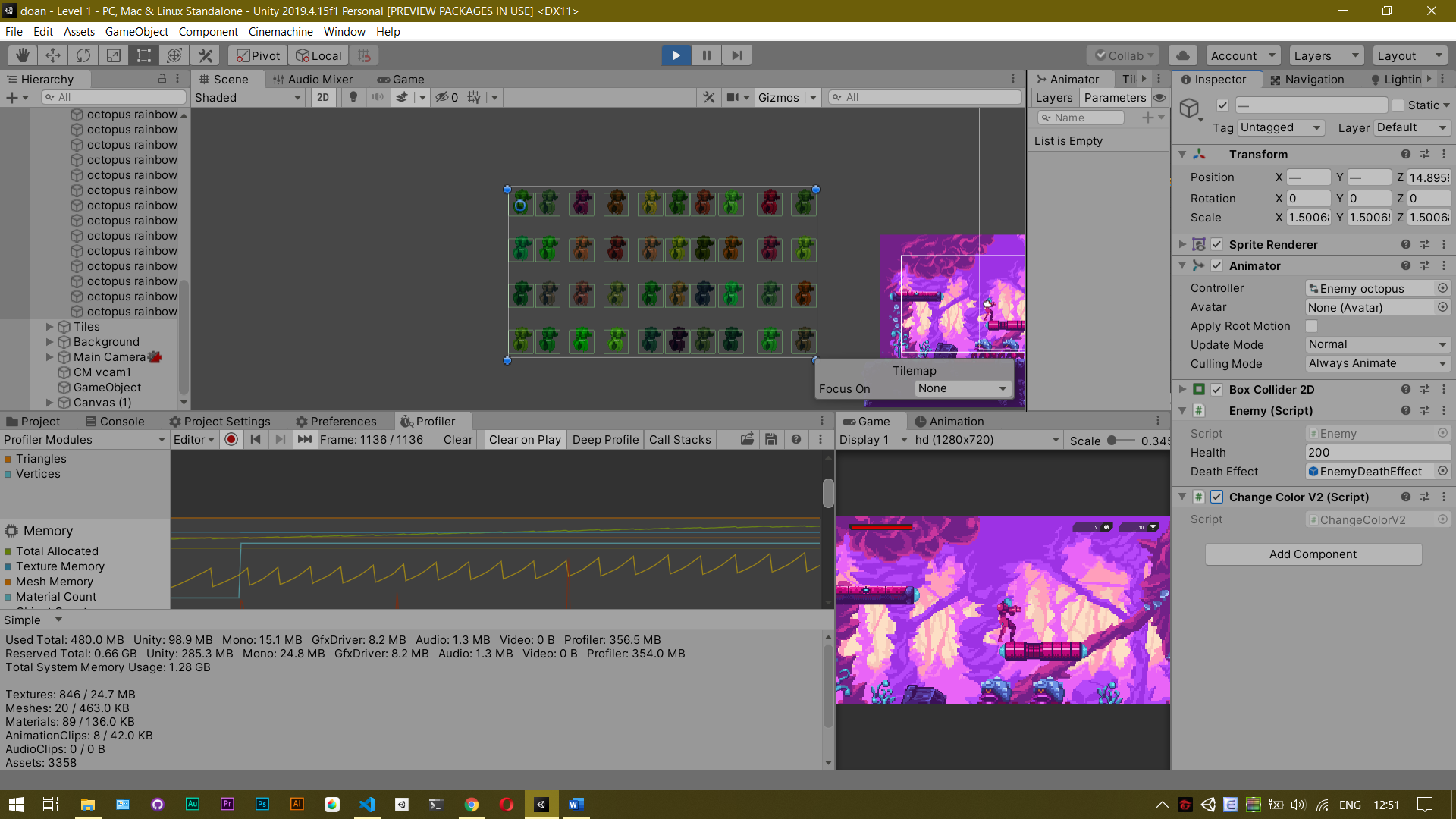




Ap dung

Truoc





<https://hieuntp2.wordpress.com/van-de-tai-su-dung-doi-tuong-va-bo-nho-giam-thieu-chi-phi-cho-qua-trinh-tao-lap-va-huy-doi-tuong/>

# 4. **State Pattern**

<https://gpcoder.com/4785-huong-dan-java-design-pattern-state/>

# 5. Object Pool

<https://gpcoder.com/4456-huong-dan-java-design-pattern-object-pool/>

Object Pool Pattern là một trong những Creational pattern. Nó không nằm trong danh sách các Pattern được giới thiệu bởi GoF. Object Pool Pattern cung cấp một kỹ thuật để tái sử dụng objects thay vì khởi tạo không kiểm soát.

Ý tưởng của Object Pooling là: chúng ta dùng Object Pool Pattern quản lý một tập hợp các objects mà sẽ được tái sử dụng trong chương trình. Khi client cần sử dụng object, thay vì tạo ra một đối tượng mới thì client chỉ cần đơn giản yêu cầu Object pool lấy một đối tượng đã có sẵn trong object pool. Sau khi object được sử dụng nó sẽ không hủy mà sẽ được trả về pool cho client khác sử dụng. Nếu tất cả các object trong pool được sử dụng thì client phải chờ cho tới khi object được trả về pool.

Object pool thông thường hoạt động theo kiểu: tự tạo đối tượng mới nếu chưa có sẵn hoặc khởi tạo trước 1 object pool chứa một số đối tượng hạn chế trong đó.

<https://www.youtube.com/watch?v=tdSmKaJvCoA&ab_channel=Brackeys>

# 6. Factory Pattern

<https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/>

# 7.