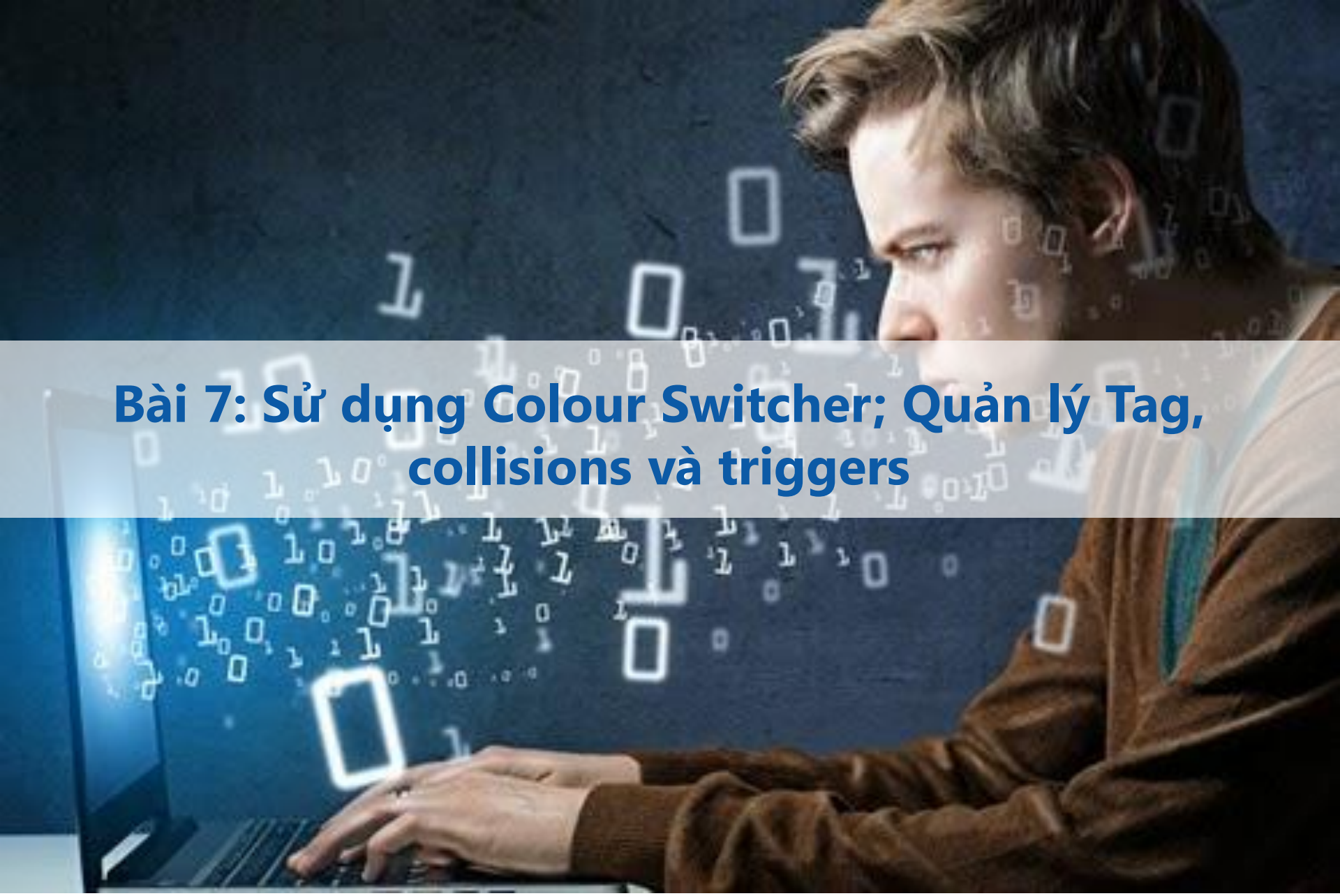


Bài 7: Sử dụng Colour Switcher; Quản lý Tag, collisions và triggers

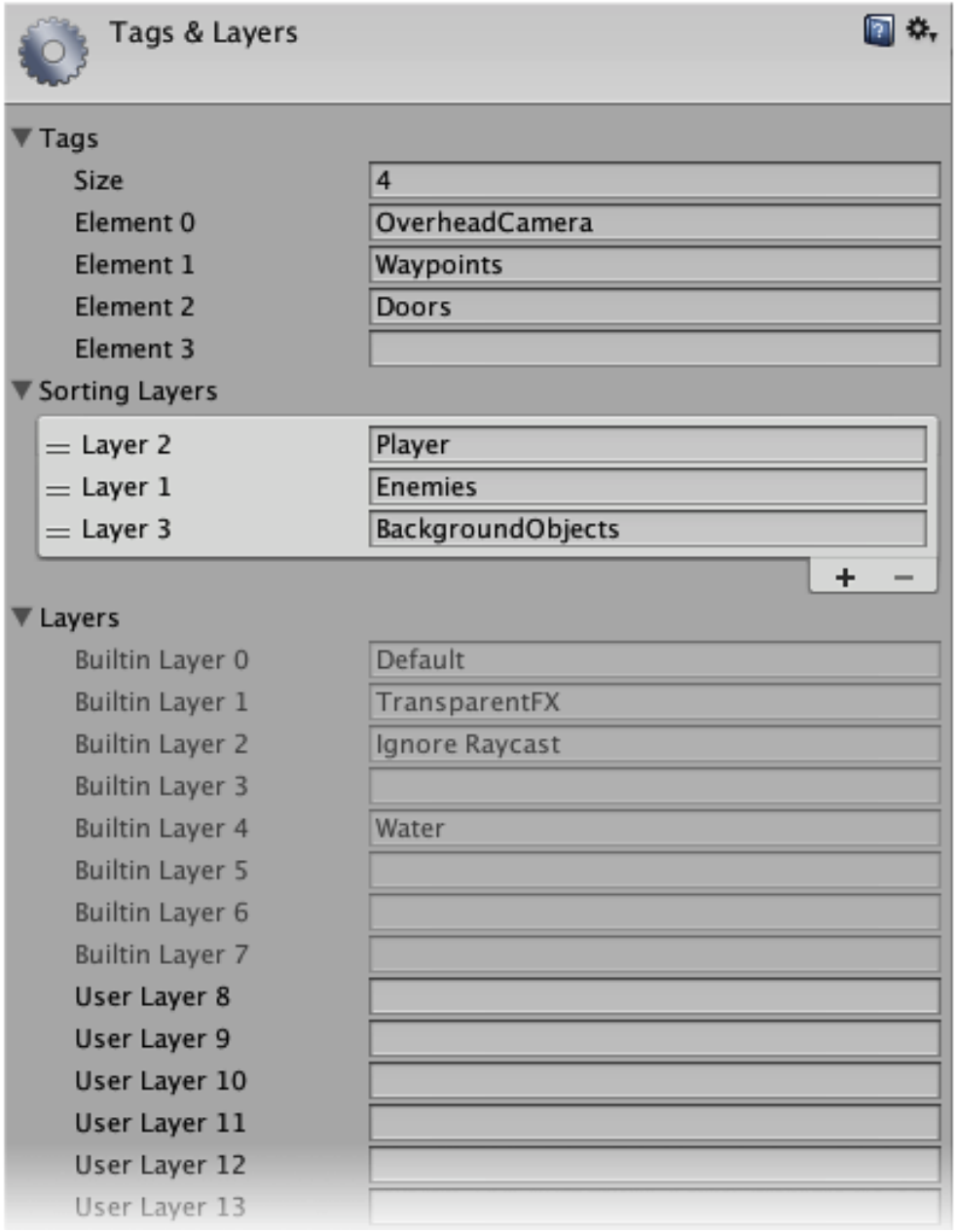
Mục tiêu

- Quản lý Tag
- Collision
- Trigger
- Quản lý bộ nhớ
- Sử dụng serialization

Quản lý Tags và Layers trong Unity3D

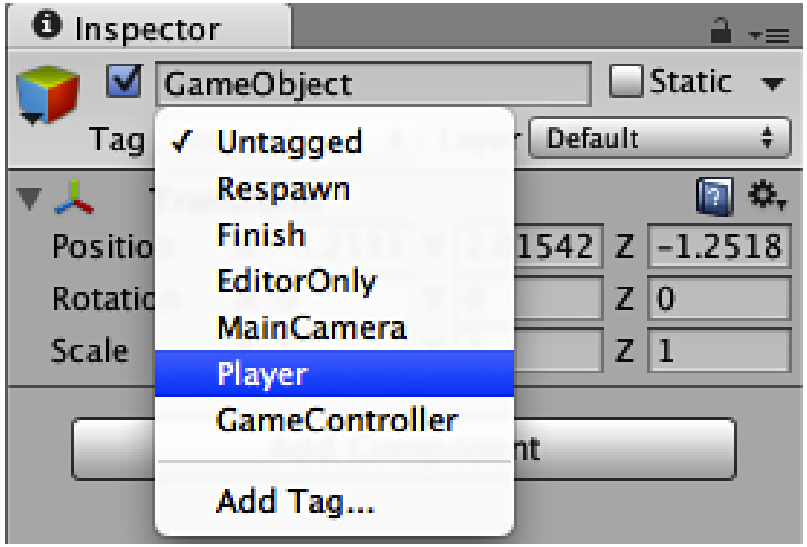
- Để làm một Game việc thiết Lập các tags và layer rất quan trọng. bài này sẽ giúp bạn hiểu được ý nghĩa của chúng có tác dụng như thế nào trong Game của chúng ta.
- Để Quản lý các Tag và Layers cho phép bạn thiết lập Layers, Tags và Sorting Layers. Để xem chúng, chọn **Edit -> Project Settings -> Tags và Layers.**

Quản lý Tags và Layers trong Unity3D



Quản lý Tags và Layers trong Unity3D

- **Tags** được đánh dấu bằng các giá trị, có thể dùng để xác định các đối tượng trong Project của bạn.
- Trong Game của bạn có rất nhiều các đối tượng làm thế nào để xác định chúng? Câu trả lời là Tags có thể giúp bạn giải quyết vấn đề đó.
- Applying Tag cho đối tượng: Chọn đối tượng trong cửa sổ Hierarchy, trong Inspector dropdown mục tag (ở bên trên góc bên trái Inspector) chọn Tag tương ứng(hình bên dưới).



Quản lý Tags và Layers trong Unity3D

- **Ví dụ:**
- Trong Game bắn súng của bạn có hai nhóm đối trượng chính là "Players" và "Enemys", Tags Tương ứng là "Player" và "Enemy"(bạn có thể đặt tên khác). Trong trường hợp va chạm nếu đạn của các đối tượng có cùng Tag thì không sao, ngược lại nếu không cùng Tag thì bên bị bắn sẽ chết (hoặc bị trừ máu) code ví dụ bên dưới:

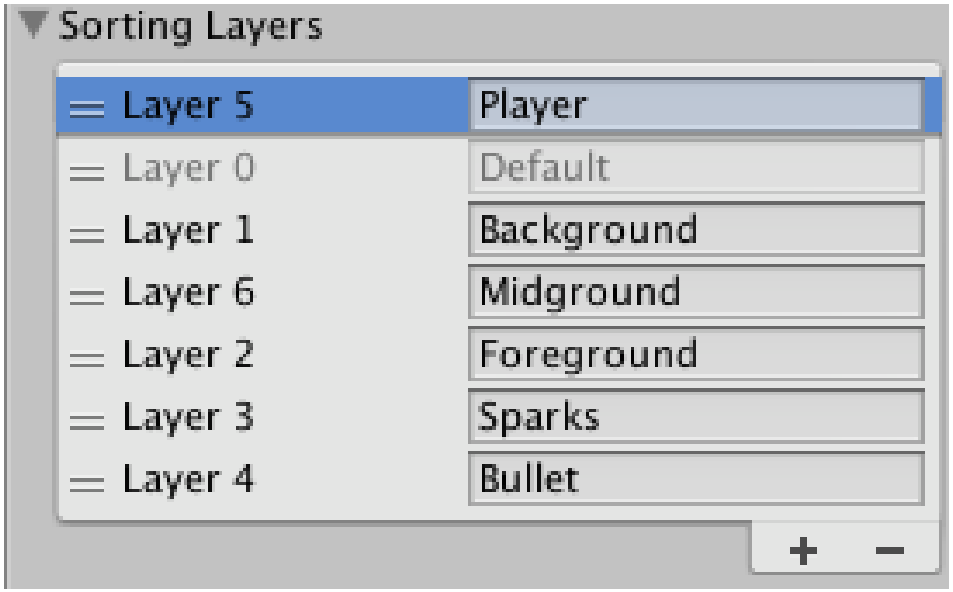
```
void OnTriggerEnter2D (Collider2D other)
{
    //Get item's layer name
    if (other.tag == "Player") {
        Destroy(other.gameObject)
    } else {
        return;
    }

}
```

Quản lý Tags và Layers trong Unity3D

- Thêm Tag: được add bằng cách nhập tên vào Element trống ở cuối phần Tags. Click chuột trái vào các Element nếu bạn muốn sửa và chuột phải nếu bạn muốn xóa chúng.
- **Sorting layer:** được sử dụng kết hợp các Sprite đồ họa trong hệ thống 2D. Các Sorting dùng để phân biệt các lớp phủ khác nhau.
- VD ta có các Layer: background, player,....vv..(*note: các tầng của layer, tầng trên sẽ che tầng dưới*).

Quản lý Tags và Layers trong Unity3D

- Để thêm layer chọn dấu"+" ở dưới tag Sorting layer, nếu muốn xóa chọn layer và click "-" bên cạnh "+".



Quản lý Tags và Layers trong Unity3D

- Để thêm layer chọn dấu"+" ở dưới tag Sorting layer, nếu muốn xóa chọn layer và click "-" bên cạnh "+".
- Layer: Các Layers được sử dụng trong Unity như là một cách để tạo ra các nhóm đối tượng chia sẻ với nhau các đặc điểm cụ thể.
- Layer chủ yếu được sử dụng để hạn chế các hoạt động như raycasting hoặc rendering, chúng chỉ áp dụng cho các nhóm đối tượng có liên quan
- Trong quản lý, tám lớp đầu tiên là mặc định được sử dụng bởi Unity và không thể chỉnh sửa.
- Tuy nhiên, các lớp từ 8 đến 31 có thể được đặt tên tùy chỉnh chỉ bằng cách gõ vào hộp văn bản thích hợp

Collision

- Collision nghĩa là sự va chạm – khái niệm này gắn liền với Component Collider trong Unity. Unity cung cấp cho chúng ta một Class Collider trong đó có 3 Messages Sent (một kiểu event – sự kiện) để kiểm soát trạng thái của C

```
void OnCollisionEnter(Collision info)
{
Debug.Log("Phat hien va cham voi: " + info.gameObject.name);
}

void OnCollisionStay(Collision info)
{
Debug.Log("O lai voi: " + info.gameObject.name);
}

void OnCollisionExit(Collision info)
{
Debug.Log("Da thoat ra khoi: " + info.gameObject.name);
}
```

Collision

- **Lưu ý:** Class này cung cấp 6 Messages Sent về Collision, tuy nhiên chúng ta khảo sát trước về 3 Messages Sent trên, 3 Messages Sent còn lại sẽ được nêu ngay sau đây (ở phần nói về Trigger).
- **Ví dụ:** Cách đơn giản nhất để làm quen với code là khảo sát ví dụ, chúng ta sẽ khảo sát thử **OnCollisionEnter** – một Messages Sent thường dùng nhất:
- Click phải ở cửa sổ Project và chọn Create -> Script->Ngôn ngữ phát triển là C#

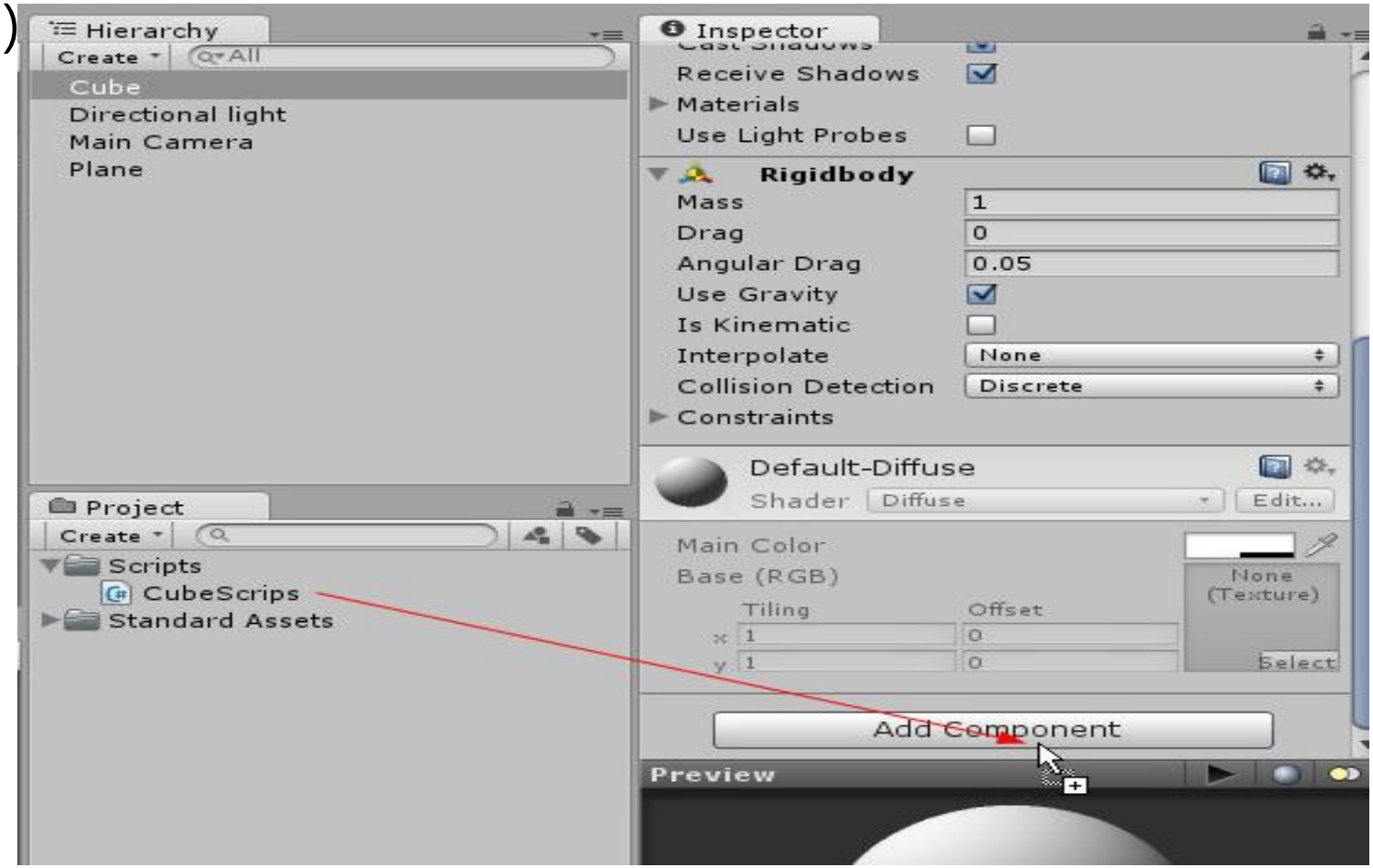
Collision

- Đặt tên tùy ý rồi double click để mở file, viết đoạn Code như hình sau:

```
void OnCollisionEnter(Collision info)
{
    Debug.Log("Phat hien va cham voi: " + info.gameObject.name);
}


void OnCollisionStay(Collision info)
{
    Debug.Log("O lai voi: " + info.gameObject.name);
}

void OnCollisionExit(Collision info)
{
    Debug.Log("Da thoat ra khoi: " + info.gameObject.name);
}
```

Collision

- Quay lại Unity, kéo thả tập tin code vừa tạo vào Cube để attach, nhấn play và theo dõi (Lưu ý: chúng ta cần mở Console lên để xem nhé, nhấn Ctrl+Shift+C thì để hiện thị)

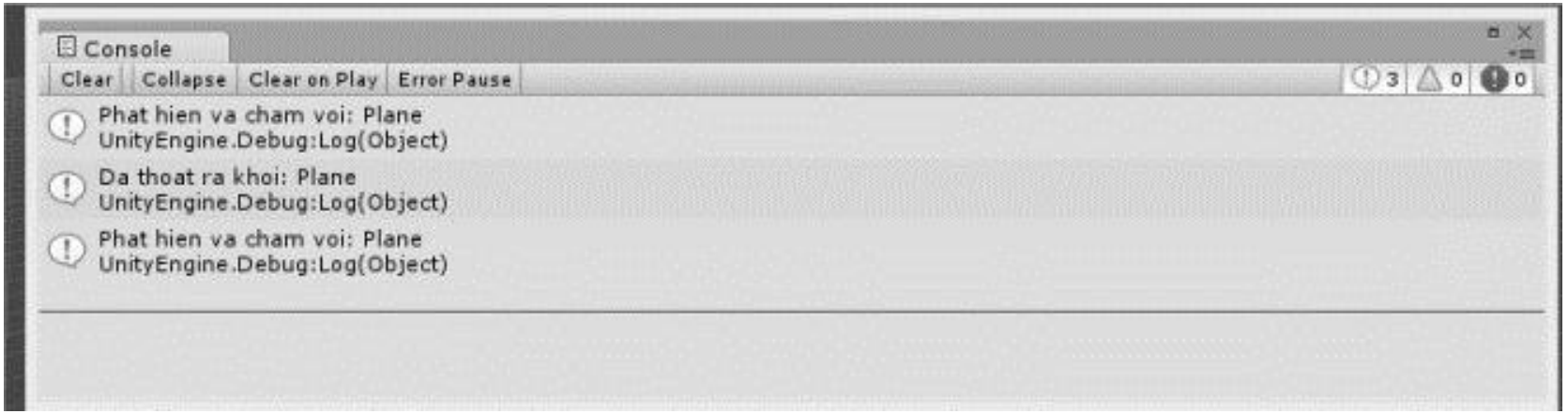


Collision

- Hình trên là kết quả sau khi Cube rơi và va chạm với Plane. Điều này chứng tỏ OnCollisionEnter đã phát hiện sự va chạm giữa Collider của nó (Cube) và Plane và thực thi lệnh in ra Debug. Info với type là Collision là biến nắm giữ thông tin của sự va chạm, bên trên viết info.gameObject.name nghĩa là tên của GameObject có collider va chạm với collide

Collision

- Tương tự với OnCollisionEnter, OnCollisionStay và OnCollisionExit sẽ bắt sự kiện Stay(ở lại) hoặc Exit(ra khỏi, thoát ra) khi có một collider nào thực hiện những tương tác này với collider của GameObject được attach code.(Nếu bạn kích hoạt cái **Material** là **bouncy** thì bạn sẽ thu được kết quả như hình dưới)



Collision

- Cube sẽ vừa va chạm với Plane và khi xảy ra thì hàm Exit sẽ được gọi và trả về là đã thoát ra khỏi Plane
- ***Lưu ý:** Các Message Sent trên sẽ không có tác dụng khi attach vào đối tượng không có collider, Unity sẽ không báo lỗi vấn đề này.

Trigger

- Chúng ta đã khảo sát về code xoay quanh Collision, như vậy chúng ta đã phần nào suy ra được những khả năng ứng dụng của chúng. Bây giờ chúng ta sẽ khảo sát về một ứng dụng khác và rất phổ biến của Collider và Collision đó là Trigger.
- Trigger theo Google Translate nghĩa là "**Kích hoạt**" hay "Cò súng". Trong Unity, Trigger là một loại Collider có thể xuyên qua, nếu Collider thông thường dùng để phát hiện va chạm và ngăn chặn mọi tác động xuyên qua Object của một Collider khác thì Trigger chỉ đóng vai trò phát hiện ra sự va chạm nhưng vẫn cho Collider tạo nên sự va chạm đó đi xuyên qua Object.

Trigger

- Thao tác tạo Trigger cũng giống như tạo Collider, tuy nhiên sau khi tạo ra Collider, chúng ta **check** chọn giá trị "**Is Trigger**" trong **Component Collider** đó.
- Dưới đây là đoạn code được attach vào Object được dùng làm Triger:

```
GameObject OldTree;

void OnTriggerEnter () {
        OldTree.animation.Play("Falling");

}
```

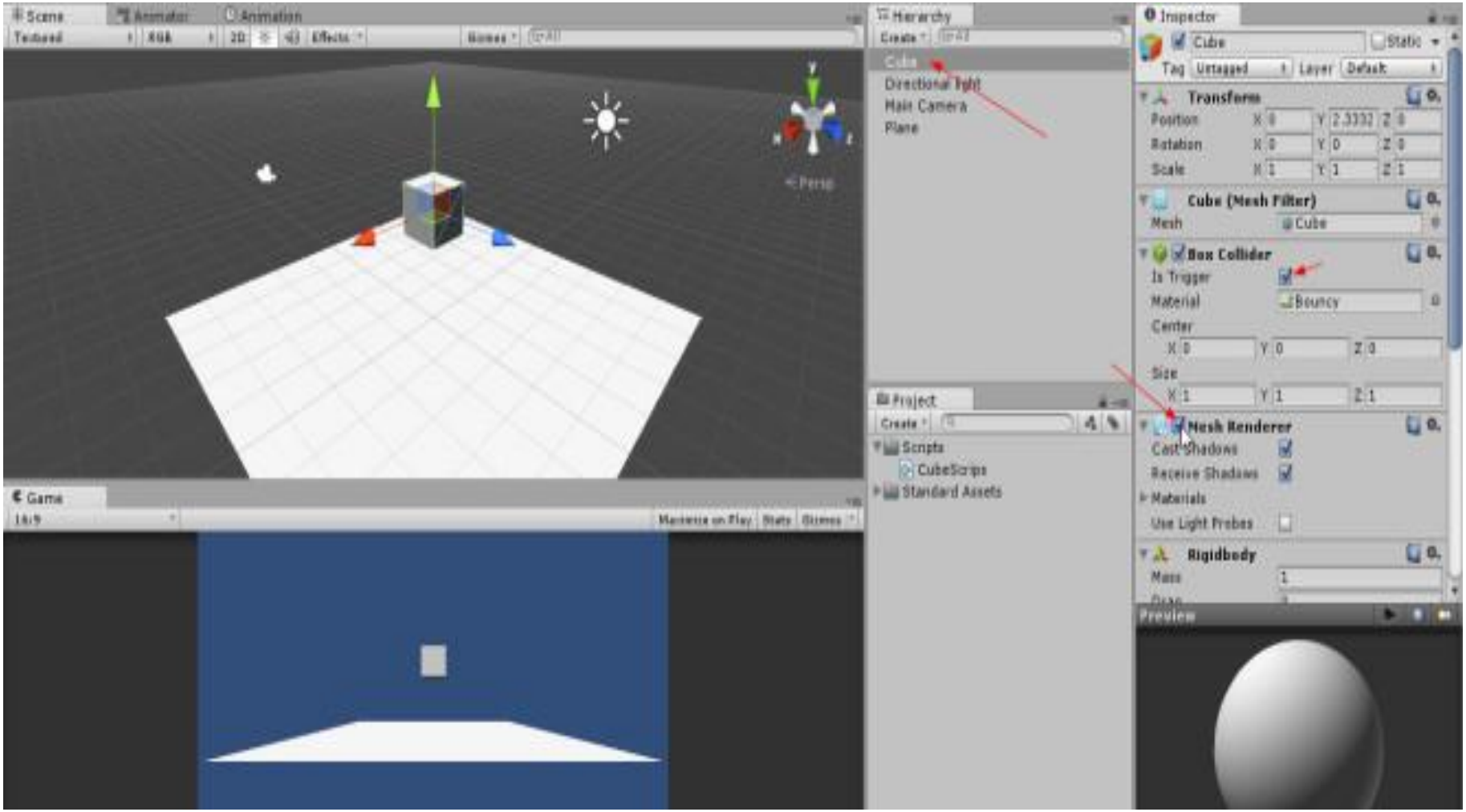
Trigger

- Như vậy chúng ta thấy, Triger **gần như hoàn toàn giống Collider** nhưng có tính **ứng dụng khác ở chỗ Triger có thể xuyên qua**. Dưới đây là 2 Message Sent khác cùa Trigger, 3 Message Sent này hợp cùng 3 Message Sent chúng ta đã khảo sát ở phần 1 tạo nên 6 Message Sent gắn liền với Class Collider, chuyên đảm nhận các vấn đề xoay quanh Collision Của Collider.
- **-OnTriggerExit**
- **-OnTriggerStay**

Trigger

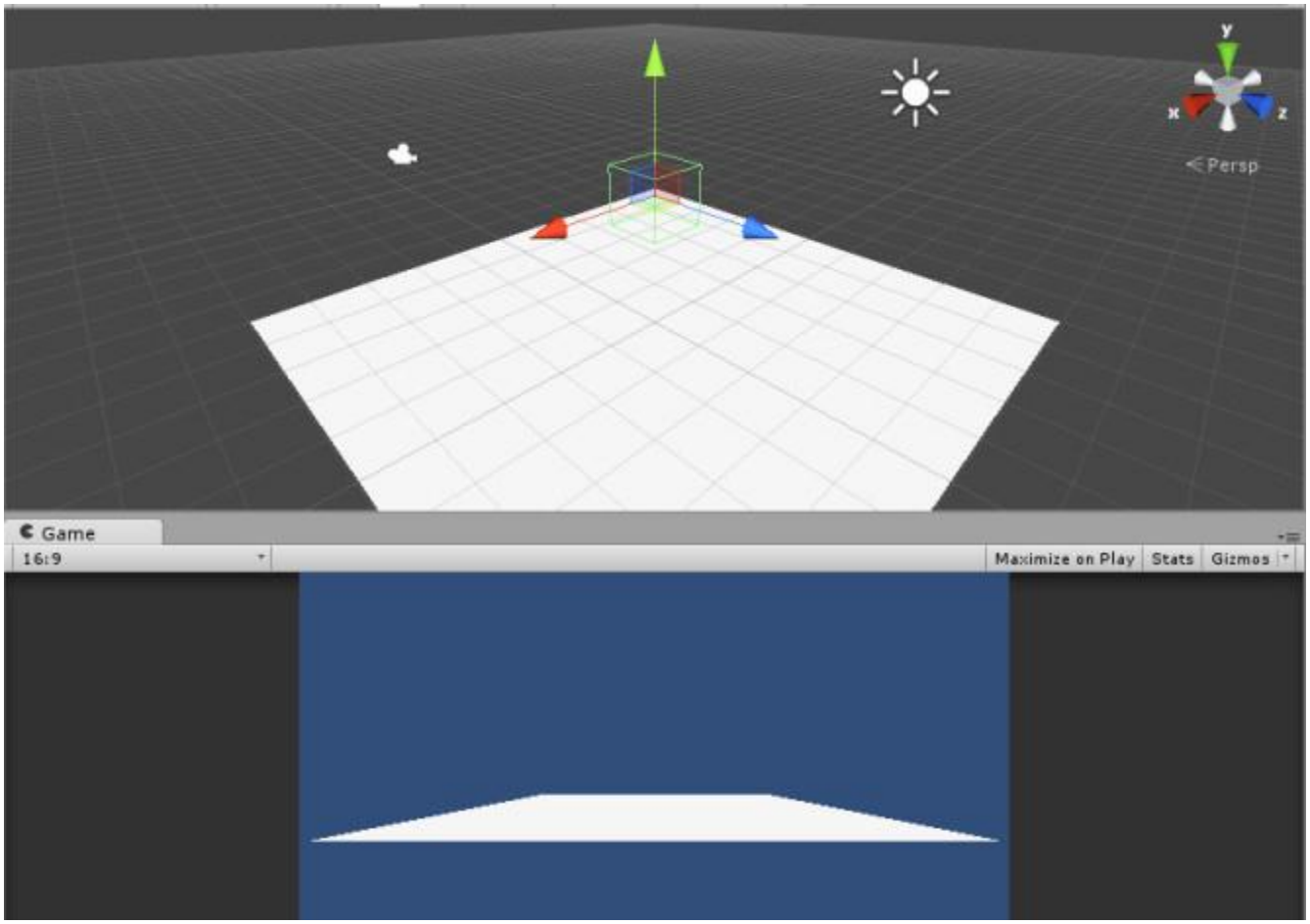
- ***Lưu ý:** Để xây dựng Trigger bắt sự kiện ở một vị trí nào đó như trên, chúng ta có thể tạo đơn giản bằng một **cube**, **check "Is Trigger"** sau đó bỏ check **Mesh Renderer** của Cube.
- Nhắc lại tính ứng dụng của Collider, ngoài ứng dụng xoay quanh va chạm, ta có thể lợi dụng khả năng ngăn cản các đối tượng có collider xuyên qua bằng cách tạo ra các Collider vô hình cho "trấn thủ" ở những nơi mà chúng ta không muốn cho người chơi đi qua.

Trigger


Hierarchy
Cube
Directional light
Main Camera
Plane
Inspector
Cube
Transform
Position
Rotation
Scale
Cube (Mesh Filter)
Mesh
Box Collider
Is Trigger
Material
Bouncy
Center
Size
Mesh Renderer
Cast Shadows
Receive Shadows
Materials
Use Light Probes
Rigidbody
Mass
Project
Create
Scripts
CubeScript
Standard Assets
Game
Persp

Trigger

- Sau khi bỏ check **Mesh Renderer** thì kết quả như dưới, đối tượng Cube đã bị ẩn



Quản lý bộ nhớ trong Unity

- Khi chúng ta tạo một mảng, chuỗi hoặc đối tượng thì sau đó bộ nhớ sẽ giao cho một vùng nhớ được gọi là HEAP.
- Khi những thứ không được sử dụng trong một thời gian dài thì bộ nhớ đó sẽ được sử dụng cho những thứ khác.
- Trước đây, các lập trình viên phải allocate và release các khối bộ nhớ HEAP một cách rõ ràng với các hàm gọi chức năng liên quan.
- Bây giờ việc quản lý bộ nhớ được thực hiện tự động bởi hệ thống thời gian thực giống như là Unity's Mon develop engine.
- Quản lý bộ nhớ tự động cho phép công sức viết code đơn giản rõ ràng hơn và giảm thiểu khả năng rò rỉ bộ nhớ .

Reference và kiểu giá trị trong Unity

- Khi bất kỳ hàm nào được gọi, sau đó các giá trị của các đối số của nó sẽ được sao chép vào một vùng bộ nhớ dành riêng cho tham chiếu riêng biệt.
- Các kiểu dữ liệu, cái mà cấp phát chỉ một vài byte, có thể được sao chép rất nhanh chóng và dễ dàng.
- Tuy nhiên, nó được phổ biến cho các mảng, các đối tượng và các chuỗi và nó sẽ không hiệu quả nếu những loại dữ liệu đó được sao chép một cách thường xuyên.

Reference và kiểu giá trị trong Unity

- May mắn thay, điều này không bắt buộc; bộ nhớ lưu trữ thực tế cho một chương trình được cấp phát từ một 'con trỏ' chỉ chiếm một vùng nhớ rất nhỏ và giá trị heap được sử dụng để ghi nhớ "địa chỉ" bộ nhớ của nó.
- Sau đó con trỏ cần phải được sao chép thông qua đối số.

Reference và kiểu giá trị trong Unity

- Kiểu mà có thể được sao chép và lưu trữ trực tiếp trong quá trình chạy được gọi là các kiểu giá trị.
- Chúng bao gồm char, float, integer, Boolean và các kiểu struct trong Unity của (ví dụ, color và Vector3).
- Các kiểu được lưu trữ trong vùng heap và sau đó truy cập thông qua con trỏ được gọi là kiểu tham chiếu. Strings, object và mảng là những ví dụ của các kiểu dữ liệu này .

Cấp phát bộ nhớ và bộ thu gom rác trong Unity

- Vùng nhớ chưa sử dụng của HEAP được liên tục theo dõi bởi trình quản lý bộ nhớ.
- Trình quản lý bộ nhớ sẽ phân bổ không gian không sử dụng khi một khối mới của bộ nhớ được yêu cầu.
- Thủ tục này sẽ xảy ra cho đến khi các yêu cầu tiếp theo đã được xử lý. Nó không chắc rằng mỗi bộ nhớ được phân bổ từ HEAP vẫn được sử dụng.

Cấp phát bộ nhớ và bộ thu gom rác trong Unity

- Để xác định được khối đống không sử dụng, trình quản lý bộ nhớ tìm kiếm qua tất cả các biến tham chiếu đang hoạt động và đánh dấu chúng là 'live'.
- Vào cuối của quá trình tìm kiếm, quản lý bộ nhớ xem xét bất kỳ không gian giữa các khối bộ nhớ trực tiếp khi trống rỗng, và cũng có thể được sử dụng để cấp phát bộ nhớ tiếp theo.
- Quá trình định vị và giải phóng bộ nhớ không sử dụng được gọi là quá trình thu gom rác (GC).

Tối ưu bộ nhớ

- GC không phải làm bằng tay và không thể được nhìn thấy bởi các lập trình viên nhưng quá trình thu thập thực sự đòi hỏi rất nhiều thời gian của CPU.
- GC là cần thiết cho các nhà phát triển để tránh những sai lầm khi giải phòng bộ nhớ không đúng lúc.

```
functionStringConatinationExample (integerArray: int[])
{
varmyLine=integerArray[0].ToString();

for(var i=1; i<integerArray.Length;i++)
{
myLine +=","+integerArray[i].ToString();
}
return myLine;
}
```

Script Serialization trong Unity

- Serialization là cốt lõi của Unity Editor.
- Nhiều tính năng của nó được xây dựng trên đỉnh của hệ thống lõi serialization và đáng kể nhất là khi bạn đang sử dụng Unity Editor, nó serialize thành phần MonoBehaviour được hỗ trợ bởi các script của bạn

Các built-in feature sử dụng serialization

- Inspector window
- Prefabs
- Instantiation
- Saving
- Loading

Các built-in feature sử dụng serialization

- Inspector window:
  - Built-in feature sử dụng serialization Inspector không giao tiếp với C# API
  - Để biết giá trị của một asset mà nó được kiểm tra: Nó request đến GameObject để tự serialize, và sau đó hiển thị dữ liệu đã được serialize này.

Các built-in feature sử dụng serialization

- Prefabs:
  - Bên trong một Prefab là dòng dữ liệu tuần tự của một (hoặc nhiều) GameObjects và component
  - Khái niệm Prefab chỉ tồn tại trong quá trình chỉnh sửa dự án trong Unity Editor
  - Những sửa đổi lắp ghép được sao lưu vào một dòng serialization bình thường khi Unity biên tập làm một build, và khi GameObjects được khởi tạo trong build, không có tham chiếu đến các GameObjects là Prefabs.

Các built-in feature sử dụng serialization

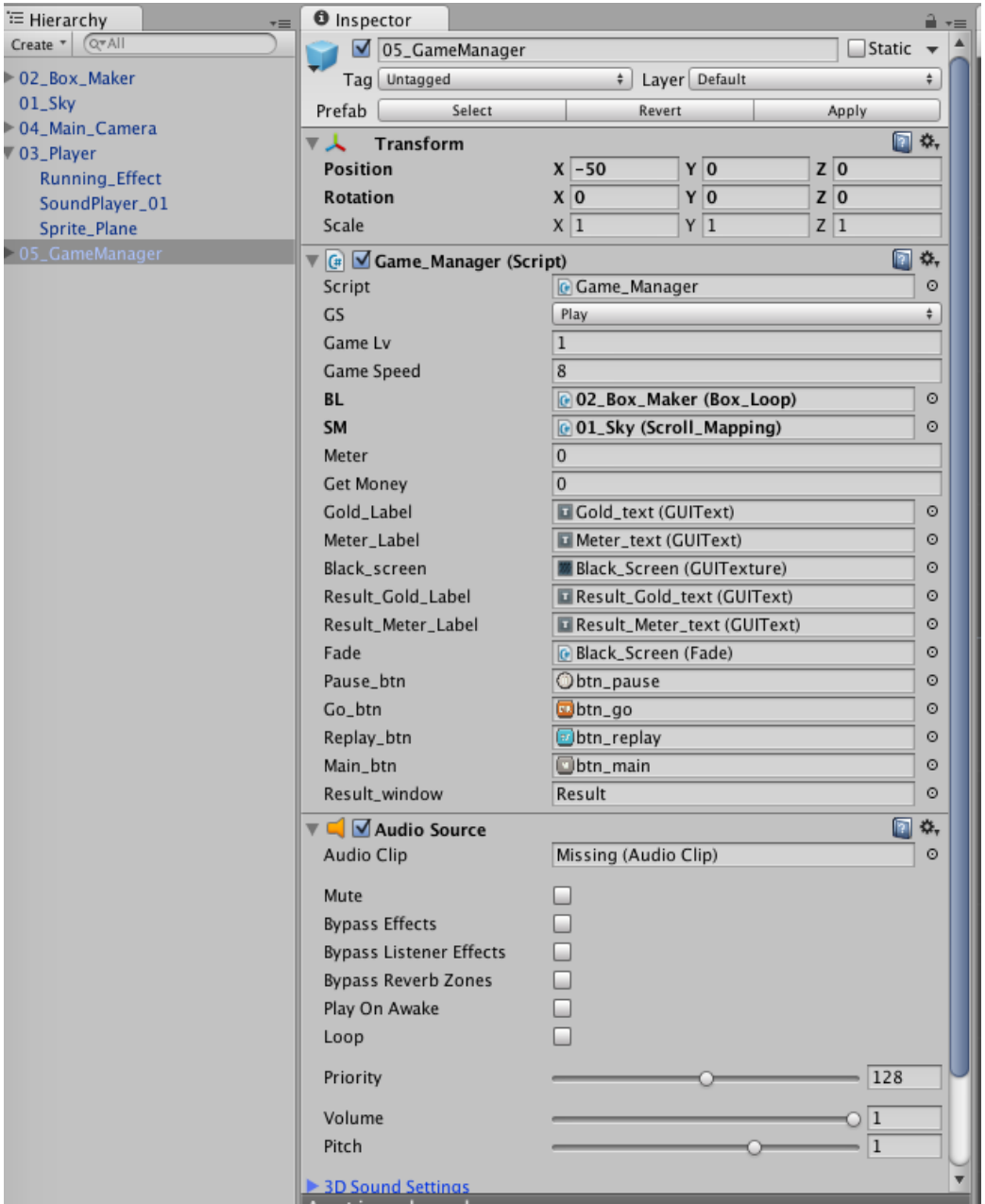
- Instantiation:
  - Khi bạn gọi Instantiate() vào bất cứ thứ gì tồn tại trong một scene (chẳng hạn như một prefab), các Editor sau đó tạo ra một GameObject mới và deserializes (có nghĩa là tải) dữ liệu vào GameObject mới.
- Saving:
  - Nếu bạn mở một file .unity Scene với một text editor, và đã thiết lập các Unity Editor để buộc serialization text, nó sẽ chạy serializer với một phụ trợ YAML.
- Loading:
  - Trong Editor YAML tải (load) cũng như tải thời gian chạy của Scene, Asset, và AssetBundles đều sử dụng hệ thống serilization

DEMO

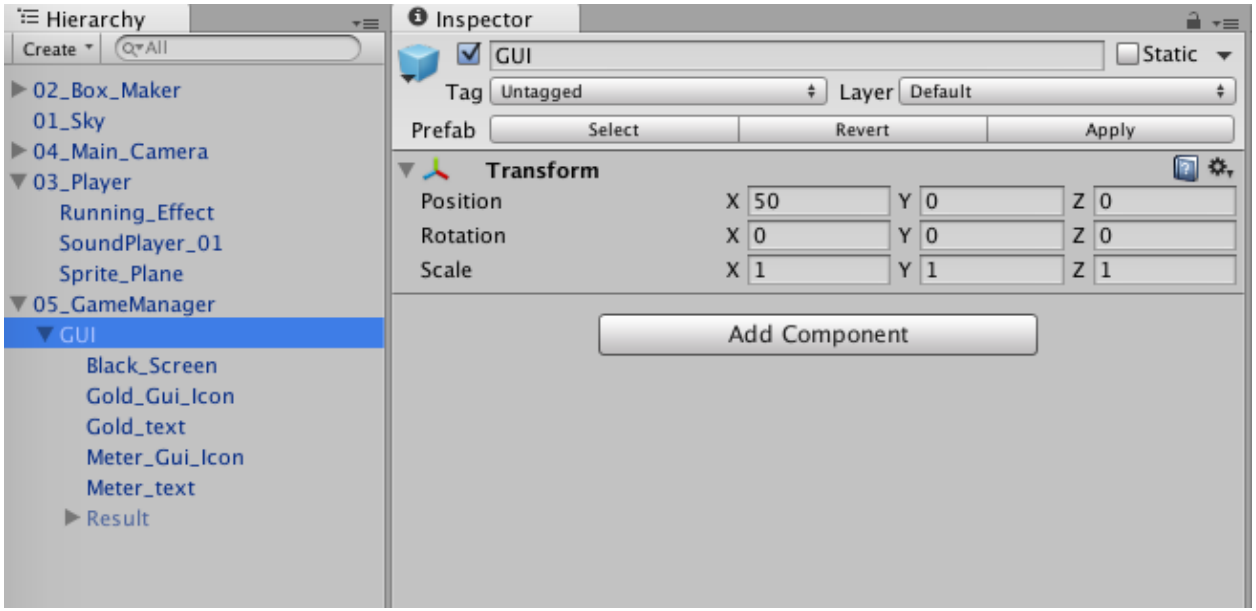
Xây dựng scene Player: (phần 1)

Character chạy trên địa hình và ăn coin

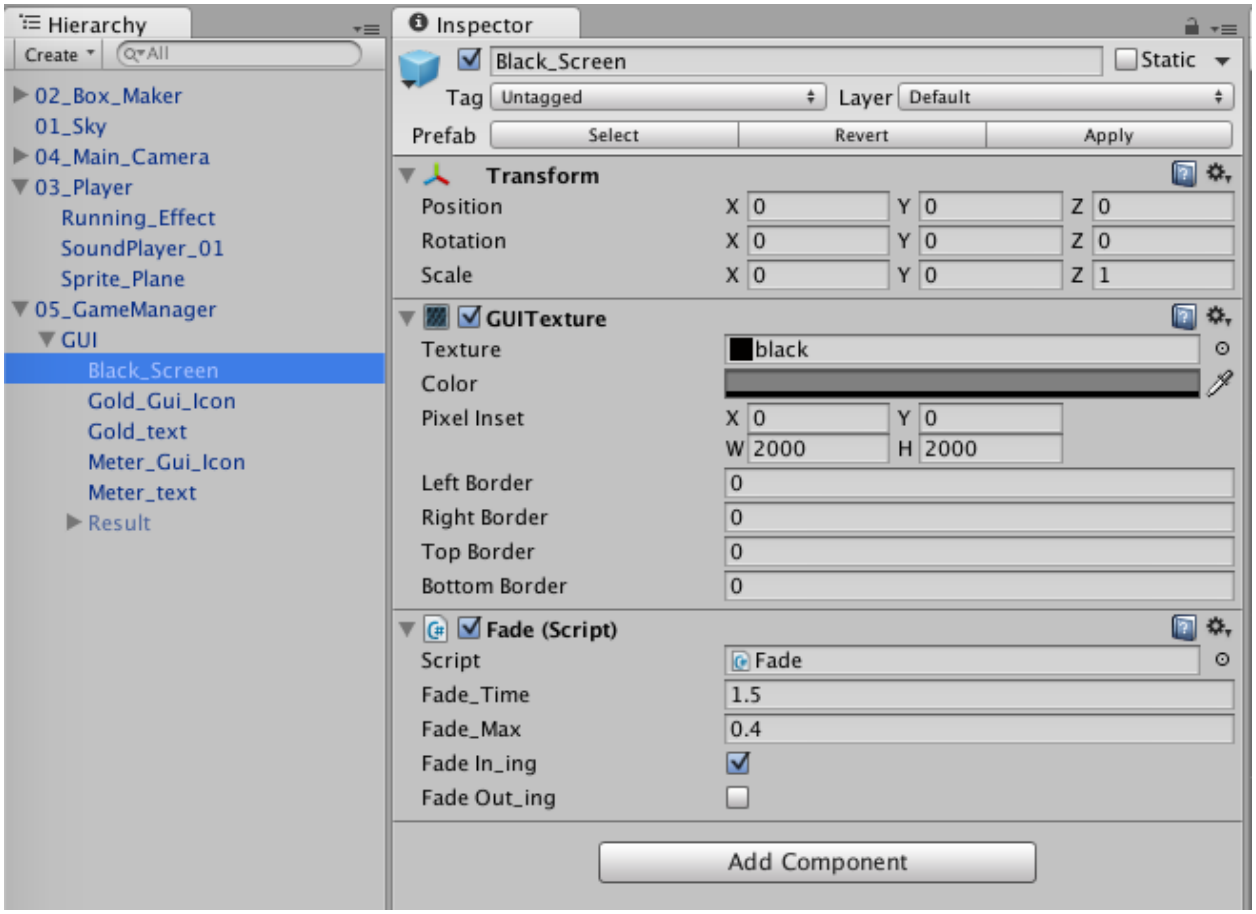
Cấu hình GameManager



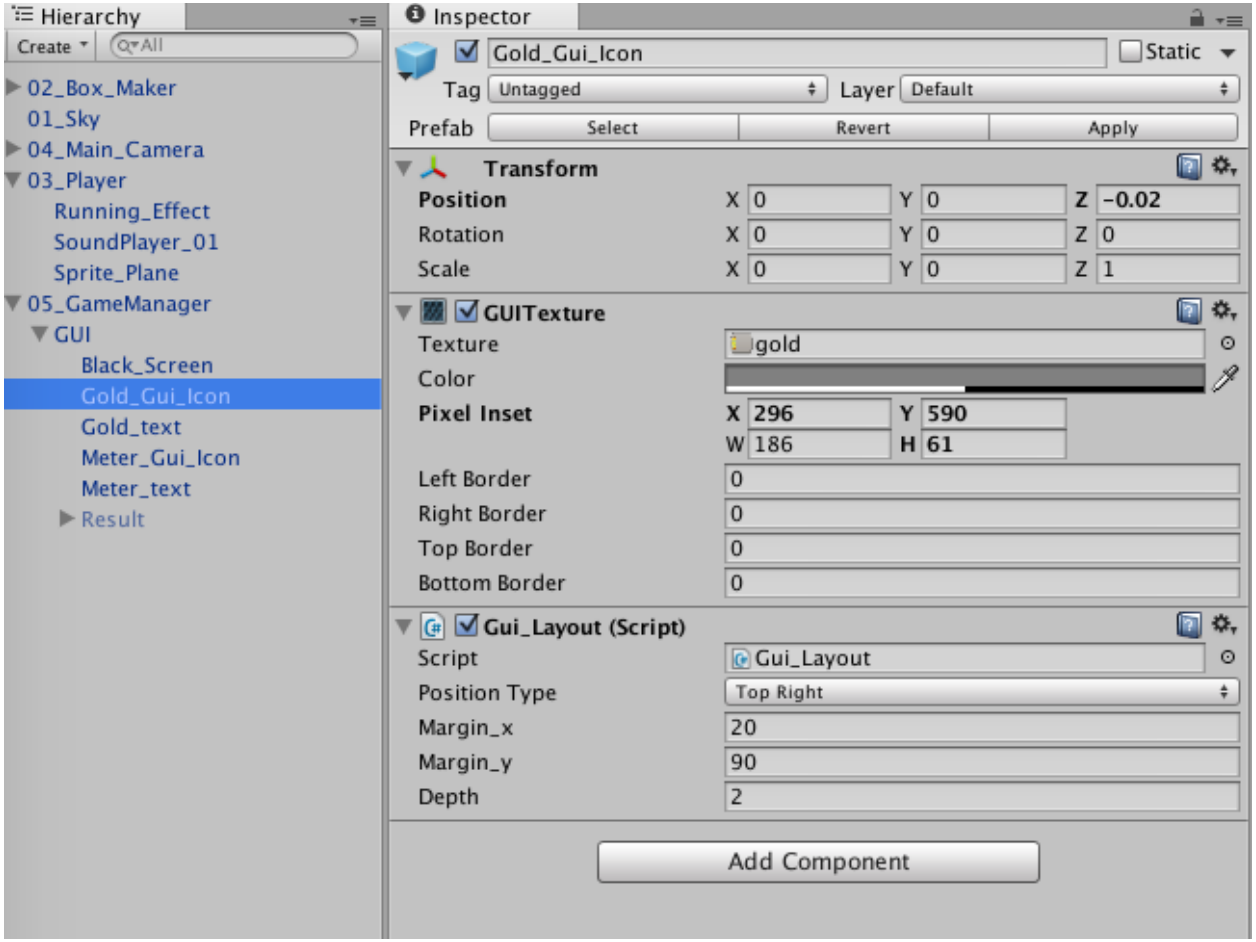
Cấu hình GUI


Hierarchy
Create
All
02_Box_Maker
01_Sky
04_Main_Camera
03_Player
Running_Effect
SoundPlayer_01
Sprite_Plane
05_GameManager
GUI
Black_Screen
Gold_Gui_Icon
Gold_text
Meter_Gui_Icon
Meter_text
Result
Inspector
GUI
Static
Tag
Untagged
Layer
Default
Prefab
Select
Revert
Apply
Transform
Position
X 50
Y 0
Z 0
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Add Component

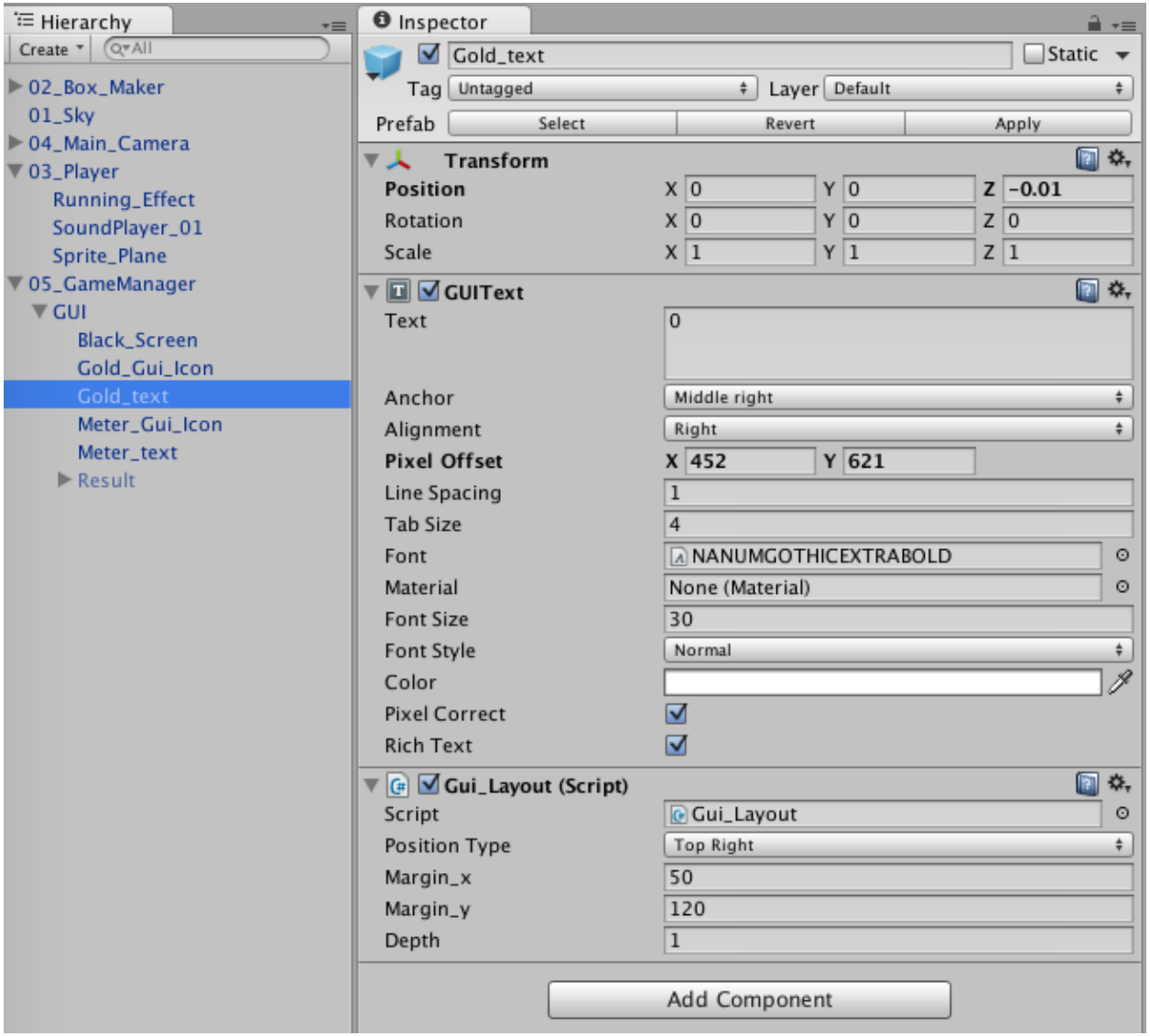
Cấu hình Black_Screen



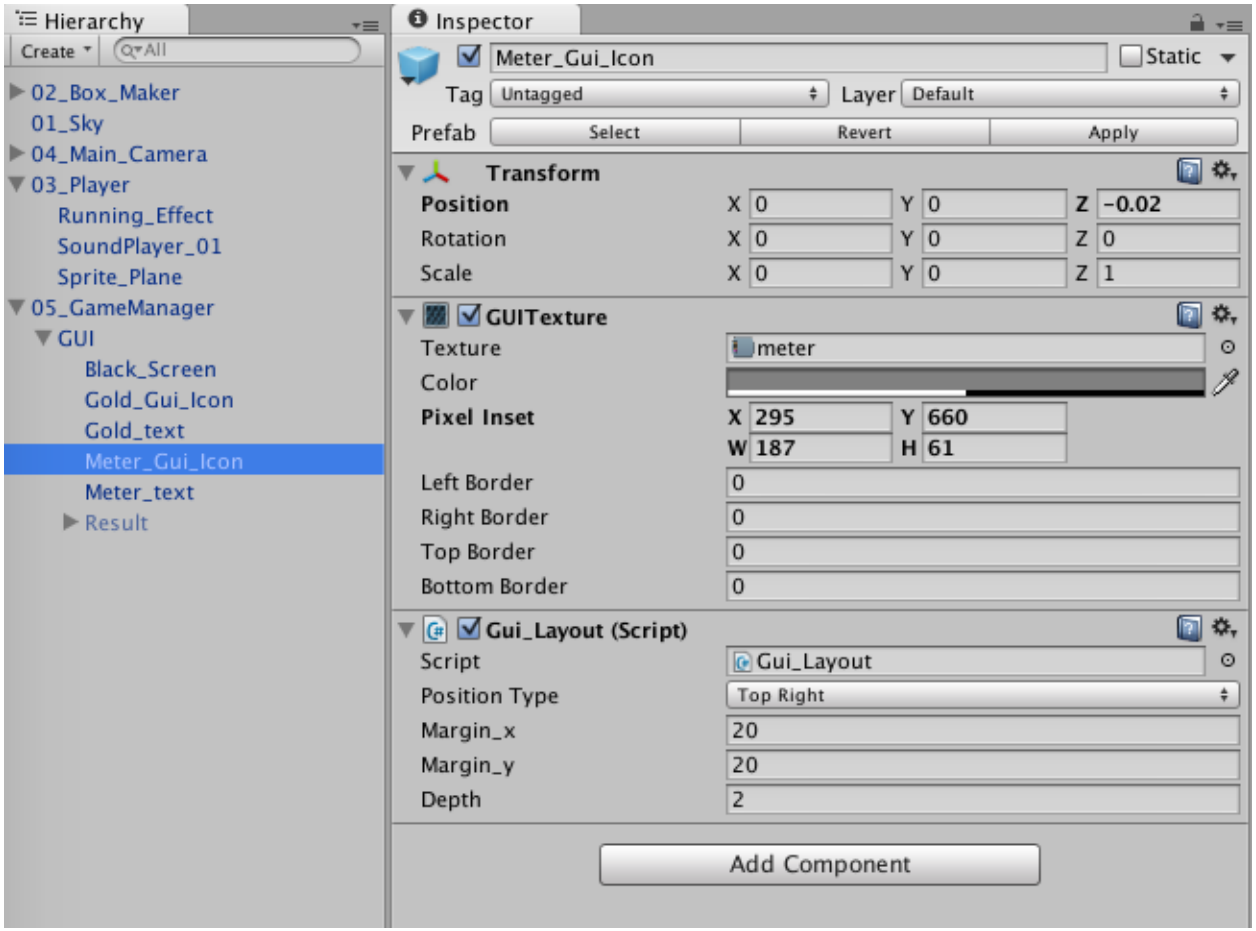
Cấu hình Gold_Gui_Icon



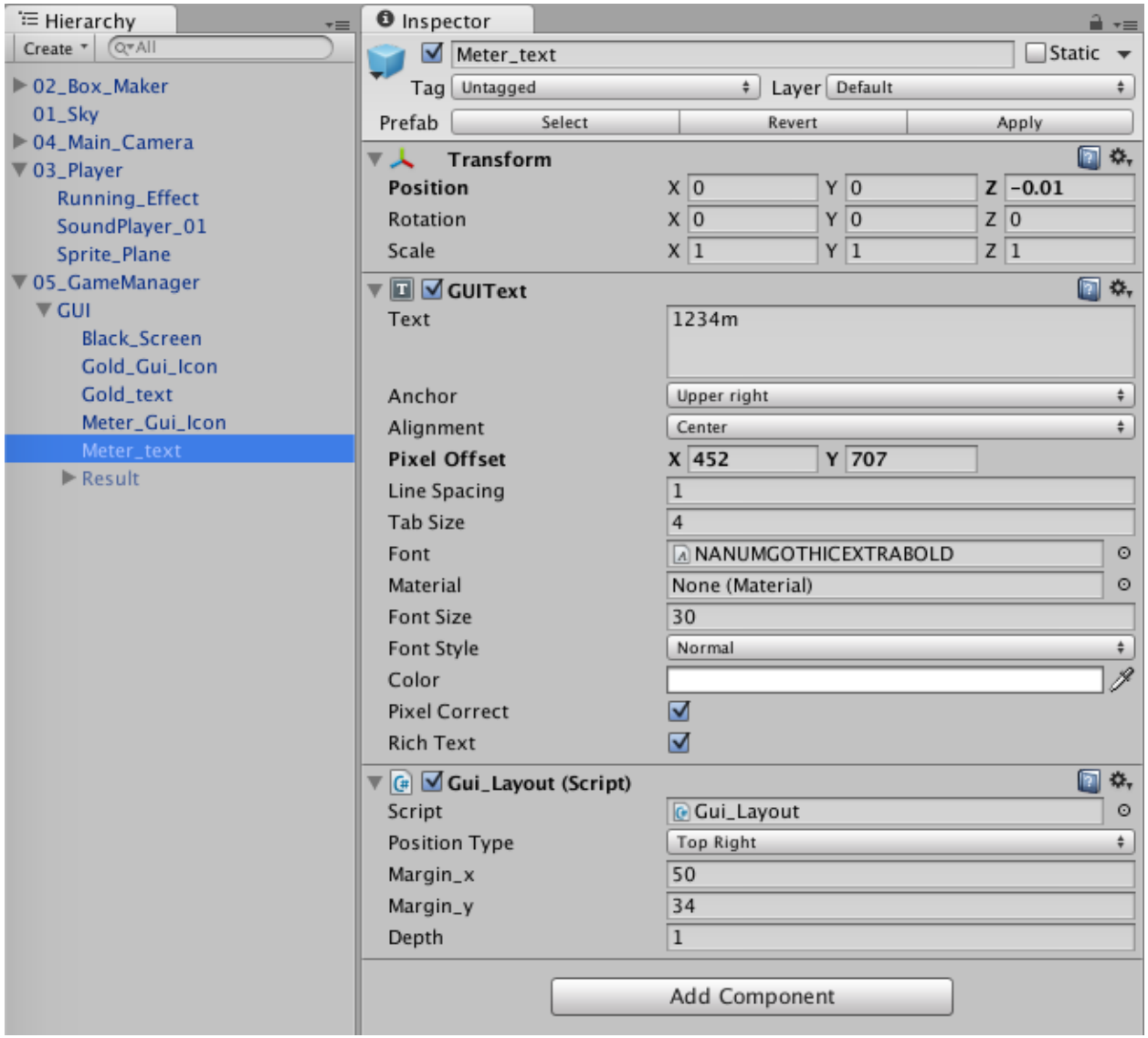
Cấu hình Gold_Text



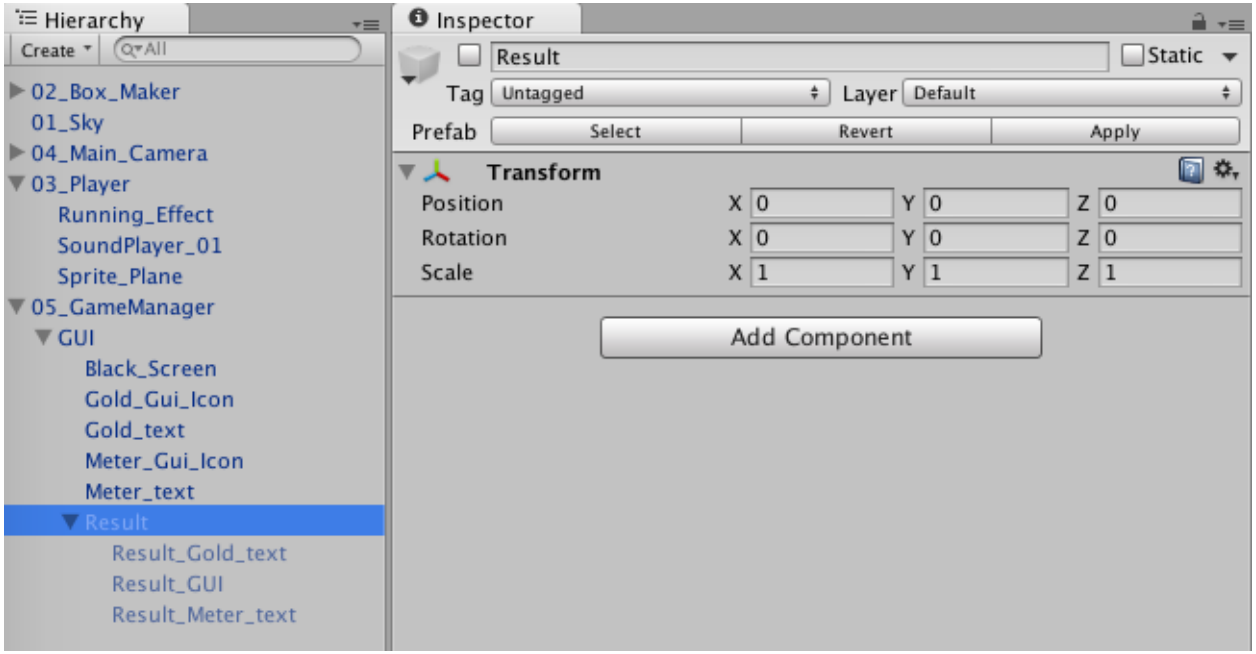
Cấu hình Meter_Gui_Icon



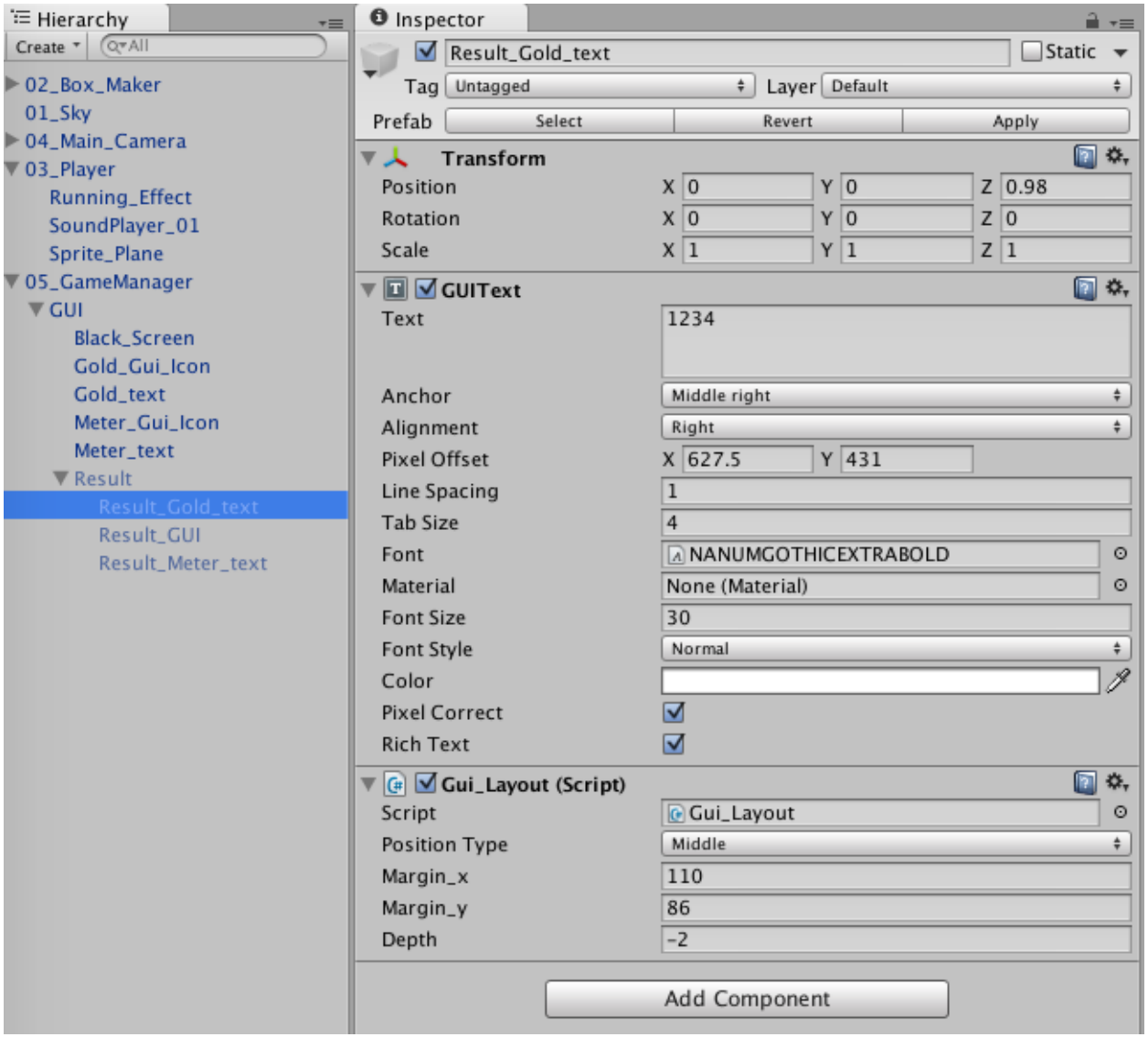
Cấu hình Meter_Text



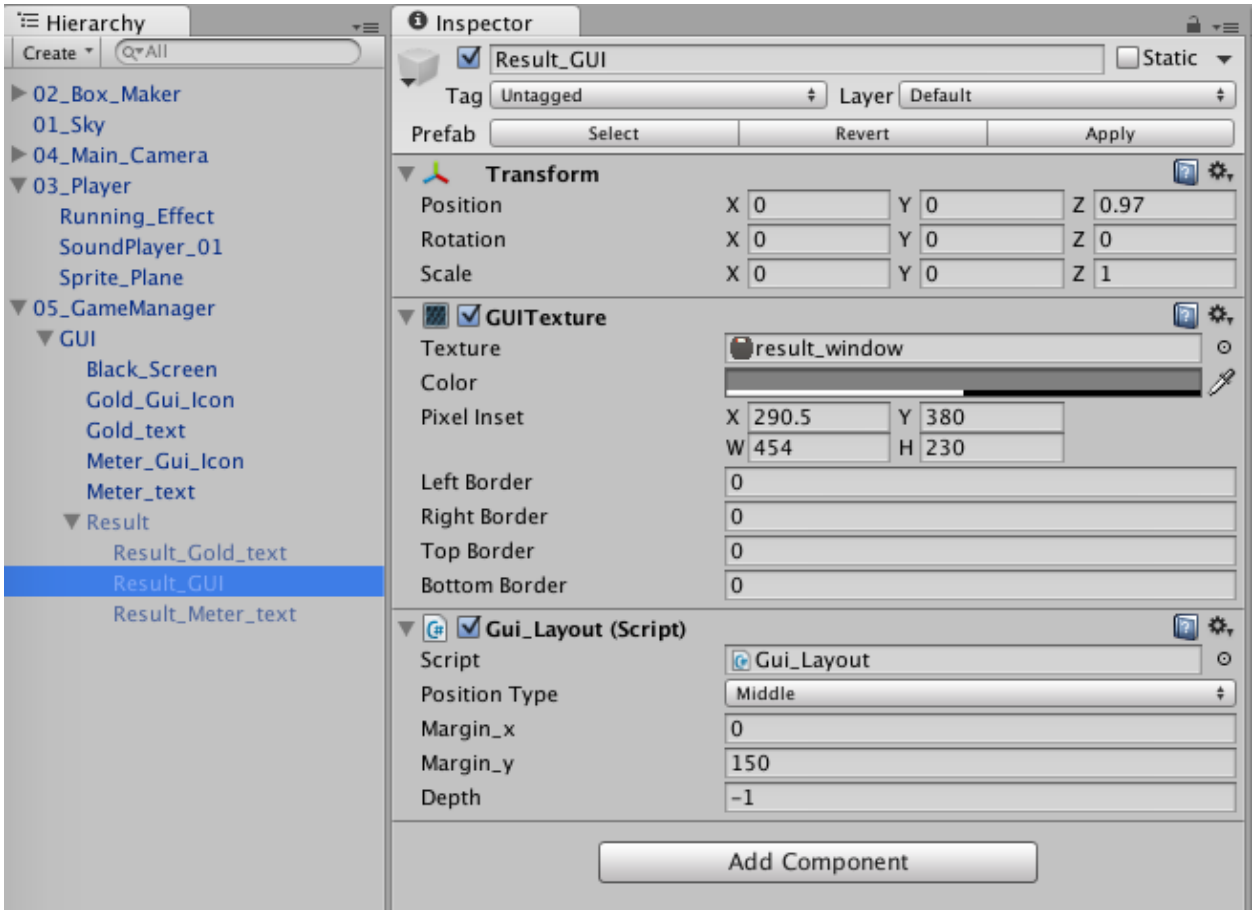
Cấu hình Result



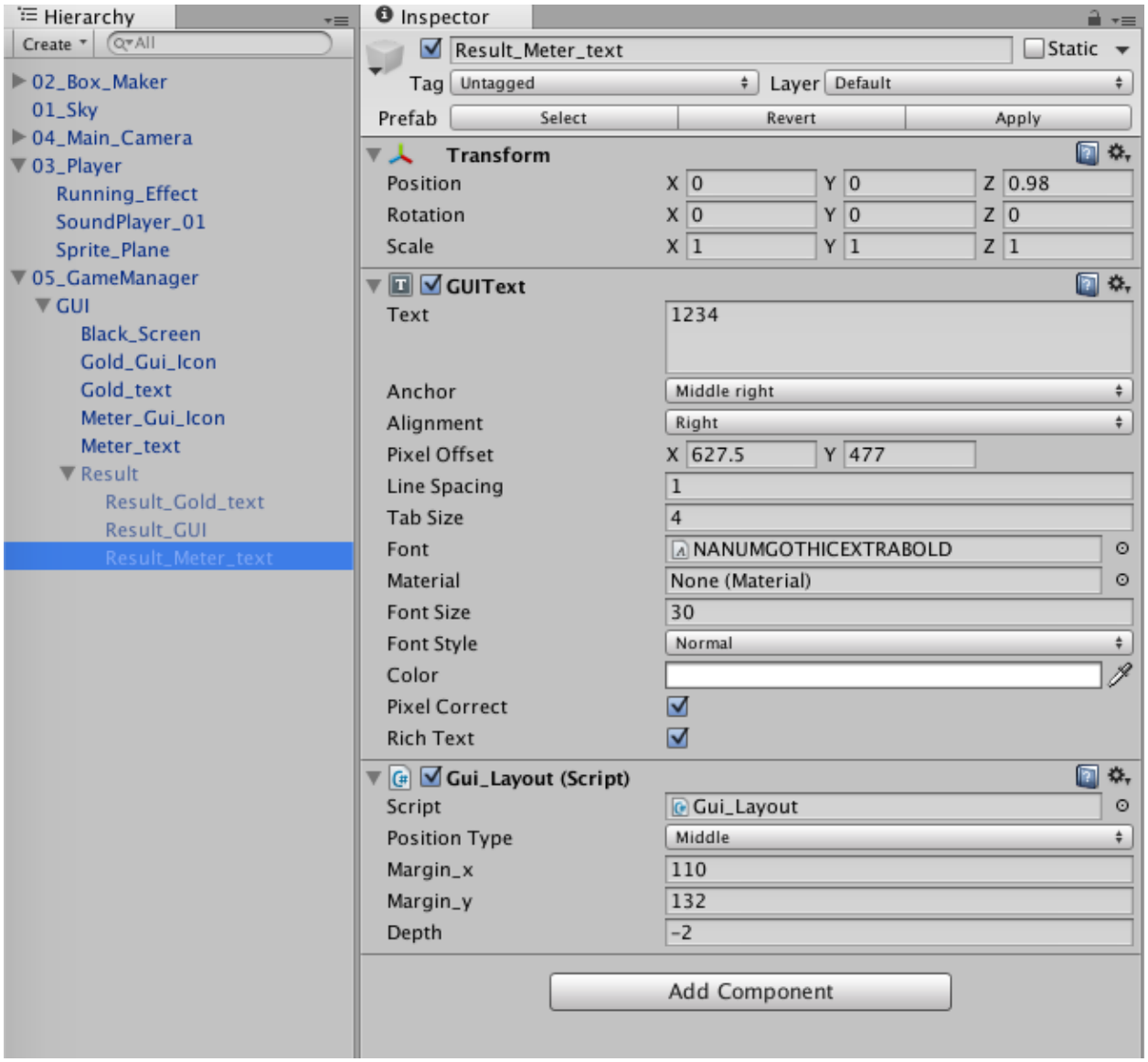
Cấu hình Result_Gold_Text



Cấu hình Result_Gui



Cấu hình Result_Meter_Text



Viết code cho Game_Manager.cs

Camera_Zoom.cs | Game_Manager.cs

```
using UnityEngine;
using System.Collections;


//Khai báo trạng thái Game
public enum GameState
{
    Play,
    Pause,
    End,
}

public class Game_Manager : MonoBehaviour
{
    /**
     *
     * @author hungnq
     */
     //Khai báo các biến cần sử dụng
    public GameState GS;
    public int GameLv;
    public float GameSpeed;
    public Box_Loop _BL;
    public Scroll_Mapping _SM;



    public float Meter;
    public int GetMoney = 0;




    public GUIText Gold_Label;
    public GUIText Meter_Label;
    public GUITexture Black_screen;
    public GUIText result_Gold_Label;
    public GUIText result_Meter_Label;
    GUIStyle guiRectStyle;
    public Fade _fade;
    public Texture Pause_btn;
    public Texture Go_btn;
    public Texture Replay_btn;
    public Texture Main_btn;
    public GameObject result_window;
    float screenX;
    float screenY;
```

Viết code cho Game_Manager.cs

```
48
49
50
51      void Start ()
52      {
53          GameObject a = GameObject.Find("02_Box_Maker");
54          if(a!=null)_BL = a.GetComponent<Box_Loop>();
55          GameObject b = GameObject.Find("01_Sky");
56          if(b!=null)_SM = b.GetComponent<Scroll_Mapping>();
57
58          GameSpeed = _BL.Speed;
59          SCREENSETTING ();
60
61
62      }
63
64      void Update ()
65      {
66          if (GS == GameState.Play) {
67              METERUPDATE ();
68          }
69
70      }
71      //Định nghĩa hàm thiết lập kích thước màn hình
72      void SCREENSETTING ()
73      {
74          screenX = Screen.width;
75          screenY = Screen.height;
76          guiRectStyle = new GUIStyle ();
77          guiRectStyle.border = new RectOffset (0, 0, 0, 0);
78          _fade.FadeIn ();
79      }
80      //Định nghia hàm cho người dùng tương tác qua giao diện
81      void OnGUI ()
82      {
83
84
85          if (GS == GameState.Play) {
86
87              if (GUI.Button (new Rect (20, 20, Pause_btn.width, Pause_btn.height), Pause_btn, guiRectStyle)) {
88                  Black_screen.color = new Color (0, 0, 0, 0.4f);
89                  GS = GameState.Pause;
90                  Time.timeScale = 0;
91              }
92          }
93
94
```

Viết code cho Game_Manager.cs

```
95
96
97          if (GS == GameState.Pause) {
98
99
100             if (GUI.Button (new Rect (screenX * 0.5f - Go_btn.width * 0.5f, screenY * 0.5f
101                 + Replay_btn.height * 0.5f + 10f, Go_btn.width, Go_btn.height), Go_btn, guiRectStyle)) {
102                 Black_screen.color = new Color (0, 0, 0, 0);
103                 Time.timeScale = 1;
104                 GS = GameState.Play;
105
106             }
107
108             if (GUI.Button (new Rect (screenX * 0.5f - Replay_btn.width * 0.5f, screenY * 0.5f
109                 - Replay_btn.height * 0.5f, Replay_btn.width, Replay_btn.height), Replay_btn, guiRectStyle)) {
110                 Time.timeScale = 1;
111                 Application.LoadLevel ("[PlayScene]");
112             }
113
114             if (GUI.Button (new Rect (screenX * 0.5f - Main_btn.width * 0.5f, screenY * 0.5f
115                 - Replay_btn.height * 0.5f - Main_btn.height - 10f, Main_btn.width, Main_btn.height),
116             Main_btn, guiRectStyle)) {
117                 Time.timeScale = 1;
118                 Application.LoadLevel ("[IntroScene]");
119             }
120         }
121
122
123
124         if (GS == GameState.End) {
125
126
127
128
129             if (GUI.Button (new Rect (screenX * 0.5f - Replay_btn.width * 0.5f, screenY * 0.5f + 10f,
130                 Replay_btn.width, Replay_btn.height), Replay_btn, guiRectStyle)) {
131                 Application.LoadLevel ("[PlayScene]");
132             }
133
134             if (GUI.Button (new Rect (screenX * 0.5f - Main_btn.width * 0.5f, screenY * 0.5f
135                 + Replay_btn.height + 20f, Main_btn.width, Main_btn.height), Main_btn, guiRectStyle)) {
136                 //Application.LoadLevel ("[IntroScene]");
137             }
138         }
139 }
```

Viết code cho Game_Manager.cs

Camera_Zoom.cs | Game_Manager.cs

```
140     //Định nghĩa hàm GameOver
141     public void GAMEOVER ()
142     {
143
144
145         GS = GameState.End;
146         _fade.FadeOut ();
147         result_window.gameObject.SetActive (true);
148         result_Gold_Label.text = string.Format ("{0:N0}", GetMoney);
149         result_Meter_Label.text = string.Format ("{0:N0}", Meter);
150     }
151     //Định nghĩa hàm ăn Coin
152     public void GETCOIN ()
153     {
154         GetMoney += 1;
155         Gold_Label.text = string.Format ("{0:N0}", GetMoney);
156     }
157 //Hàm quản lý level
158     public void METERUPDATE ()
159     {
160         Meter += Time.deltaTime * GameSpeed;
161         Meter_Label.text = string.Format ("{0:N0}<color=#ff3366> m</color>", Meter);
162
163
164
165         if (Meter >= 50 && GameLv == 1) {
166             GameLevelUp ();
167         }
168
169         if (Meter >= 100 && GameLv == 2) {
170             GameLevelUp ();
171         }
172
173         if (Meter >= 150 && GameLv == 3) {
174             GameLevelUp ();
175         }
176
177         if (Meter >= 200 && GameLv == 4) {
178             GameLevelUp ();
179         }
180
181         if (Meter >= 250 && GameLv == 5) {
182             GameLevelUp ();
183         }
184
185         if (Meter >= 300 && GameLv == 6) {
186             GameLevelUp ();
```

Viết code cho Game_Manager.cs

```
187             }
188         }
189     //Định nghĩa hàm cập nhật level
190         public void GameLevelUp ()
191         {
192             GameLv += 1;
193             GameSpeed += 3;
194             _SM.ScrollSpeed += 0.1f;
195             _BL.Speed = GameSpeed;
196         }
197     }
```

Tổng kết

- Quản lý Tag
- Collision
- Trigger
- Quản lý bộ nhớ
- Sử dụng serialization

Fpt University

FPT POLYTECHNIC

THANK YOU!

www.poly.edu.vn