Buenas Prácticas para Desarrollo de Videojuegos con Unity y C#

Por: Irving Malcolm

Una breve reseña sobre mí

Soy Ingeniero Eléctrico-Electrónico, graduado de la Universidad Tecnológica de Panamá.

Estudié una carrera de 3 años de Desarrollador de Juegos en IMAGE Campus, en Buenos Aires, Argentina. Esta carrera ponía énfasis en Programación y Game Design, con algunas otras materias adicionales de arte y negocio.

Llevo 10 años programando, 3 a nivel académico y 7 a nivel profesional. Desarrollé juegos en C++ con engines de mis profesores, Java para celulares, C# y Xna para PC en el instituto. Actionscript 3.0 para web en empresas. Unity3d en proyectos indie y laborales desde hace 5 años.

Programar vs Programar bien

* Al inicio nos importa programar para que funcione.

* Algunos somos rebeldes ante consejos de programadores con más experiencia. Queremos hacerlo a nuestra manera.

* A medida que el tiempo pasa, nos interesamos más por programar bien, para lograr código eficiente, claro, ordenado, reutilizable, y finalmente, en sentirnos orgullosos de la arquitectura de nuestro sistema.

* En ocasiones comprendemos tarde la razón de los consejos que nos dieron, o bien los seguimos desafiando hasta crear un nuevo paradigma.

Razones para las buenas prácticas de programación

- **Eficiencia**: Que la lógica logre lo mismo en menos pasos o accesos a memoria. Que se haga un uso más óptimo de los recursos disponibles.
- **Claridad**: Que el código se entienda mejor por terceros o uno mismo en el futuro.
- **Mantenibilidad**: Que resulte fácil poder cambiar, mejorar, probar o separar el código.
- **Orden**: Para poder encontrar lo que se busca más fácilmente.
- **Extensión**: Reducir la cantidad de líneas de código.

Diferenciar los nombres de variables: Públicas, _privadas y locales

- **Razón**: Al estar leyendo cualquier parte del código, permite saber cuando una variable puede ser accesada desde afuera, o asignada por inspector, y sí es local, creada solo para usarse dentro de una función o desde cualquier parte de la clase.
- **Ejemplo**:

```
 public GameObject TargetReference;
private Vector3  _velocity;


private void handleHorizontalVelocity()
{
   private float velocityH = _velocity.x;
   //…...
}
```

Ordenar el código por grupos lógicos

Razón: Orden. Poder encontrar más fácil variables, métodos o enums sabiendo más o menos dónde están en el cuerpo de la clase.

Ejemplo:

```
public enum Names
{
    Punch,
    Kick
}

const int ATTACKS_AMOUNT = 6;

public int MaxCombo = 5;

protected float _attackRange = 2;

private float _attackSpeed = 3.2f;
```

```
public float AttackSpeed { get { return _attackSpeed; }
}

void Start() { }

public Attack GetCurrentAttack() { }

protected void updateStats() { }

private void checkDistance() { }

class Attack  //Nested class
{
    //…..
}

struct MoveInfo
{
    //…...
}
```

Dejar espacio entre operandos y operadores

- **Razón**: Claridad

Ejemplo:

```
movementX=SpeedX*Time.deltaTime;
```

vs

```
movementX = SpeedX * Time.deltaTime;
```

Dejar espacio entre variables u operaciones no relacionadas

Razón: Claridad.

Ejemplo:

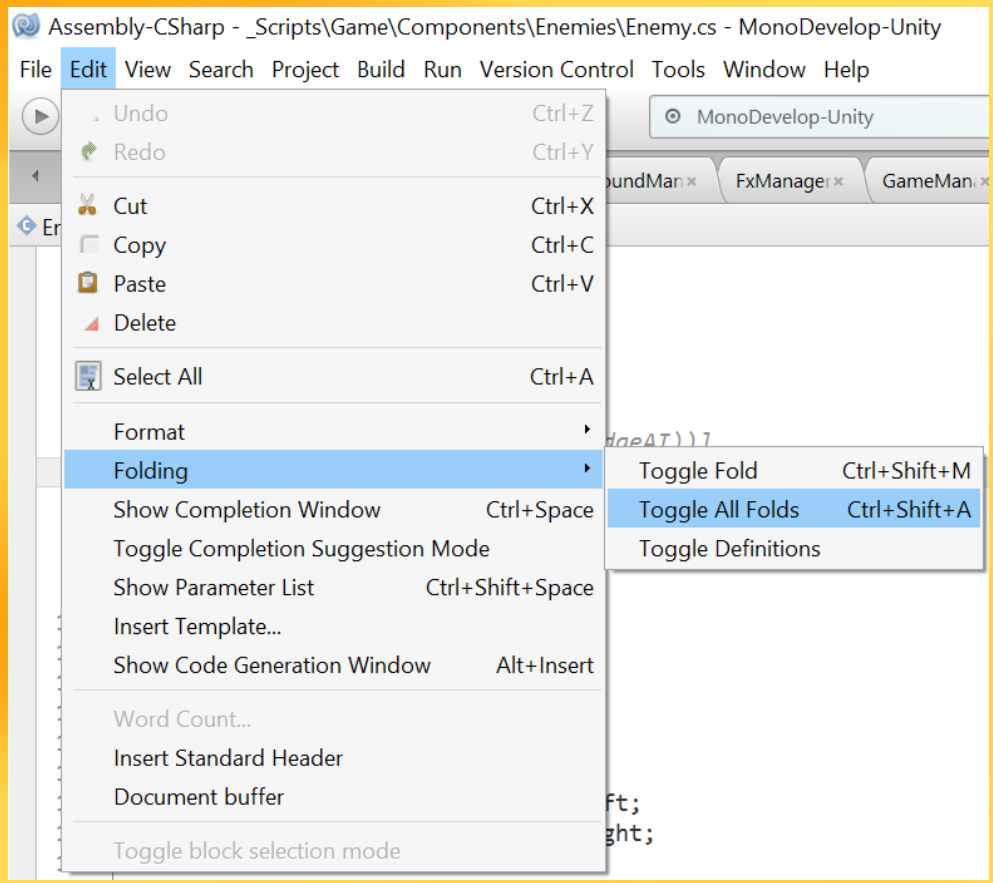
```
private Vector3 _velocity;
private float _gravity;

private Dictionary<Attack.Names, Attack> _attacksByName;


public void Initialize ()
{
    _velocity.y = 0;
    _gravity = GameConfig.Gravity;

    _attacksByName = new Dictionary<Attack.Names, Attack>();
    foreach(Attack attack in Attacks)
    {
        if(attack.Name != Attack.Names.None)
            _attacksByName.Add(attack.Name, attack);
    }

    if(HitBox != null)
        HitBox.SetActive(false);
}
```

Utilizar regiones para agrupar y esconder grandes bloques de código.

Razón: Claridad, Orden, Mantenibilidad.

Aparte de servir comor referencia para navegar el código, se pueden esconder algunas o todas las regiones utilizando Folding como se muestra en la imagen derecha.

```
public abstract class Enemy : Actor
{
    Enums

    Public fields

    Constants

    #region Protected fields
    protected Fsm _fsm;

    protected float _patrolLimitLeft;
    protected float _patrolLimitRight;

    protected float _activationTimeCount;
    protected float _perceptionTimeCount;

    protected int _currentCounterAttackPercentage = 50;
    protected int _currentComboContinuePercentage = 50;
    protected int _currentComboLimit = 3;
    protected int _comboCount = 0;

    protected SimpleCallback _currentMovementHandler;
    #endregion
```



No utilizar “Números mágicos”

- **Razón**: Claridad y mantenibilidad. Indicar qué son esos números y poder cambiarlos en un sólo lugar.
- **Ejemplo**:

```
const float GROUND_SPEED_FACTOR = 0.1f;
const float GROUND_SPEED_OFFSET = 0.2f;

public float BaseSpeed = 4;
private float _speed = 0;

private void updateGroundSpeed()
{
        _speed = BaseSpeed * 0.1f + 0.2f;
        //VS
        _speed = BaseSpeed * GROUND_SPEED_FACTOR +
GROUND_SPEED_OFFSET;
}
```

No utilizar “Números mágicos”

- **Razón**: Claridad y mantenibilidad. Indicar qué son esos números y poder cambiarlos en un sólo lugar.
- **Ejemplo**:

```
const float GROUND_SPEED_FACTOR = 0.1f;
const float GROUND_SPEED_OFFSET = 0.2f;

public float BaseSpeed = 4;
private float _speed = 0;

private void updateGroundSpeed()
{
        _speed = BaseSpeed * 0.1f + 0.2f;
        //VS
        _speed = BaseSpeed * GROUND_SPEED_FACTOR +
GROUND_SPEED_OFFSET;
}
```

Utilizar variables locales para acortar el código y reducir los accesos.

Razón: Eficiencia, extensión y claridad.

Ejemplo:

```
public GameObject MyGameObject;

public Vector3 StartingPos;
public Quaternion StartingRot;
public Vector3 StartingScale;

void Start ()
{
    MyGameObject.GetComponent<Transform>() .position = StartingPos;
    MyGameObject.GetComponent<Transform>() .rotation = StartingRot;
    MyGameObject.GetComponent<Transform>() .localScale = StartingScale;


    //VS


    Transform myTransform = MyGameObject.GetComponent<Transform>() ;
    myTransform.position = StartingPos;
    myTransform.rotation = StartingRot;
    myTransform.localScale = StartingScale;
}
```

Utilizar variables de clase para acortar el código y reducir los accesos.

Razón: Eficiencia, extensión.

```
private SpriteRenderer _myWindowRenderer;

void Awake()
{
    _myWindowRenderer =
GameObject.Find(“Window”).GetComponent<SpriteRenderer>();
}

public void Show()
{
    _myWindowRenderer.SetActive(false);
}

public void Hide()
{
    _myWindowRenderer.SetActive(true);
}
```

Reutilización de Código.
Extraer métodos de código similar o igual.

- **Razón**: Claridad, extensión y mantenibilidad. Siempre que una serie de operaciones sea similar/igual, es recomendable extraer un método.

```
public float X1, X2, Y1, Y2, W1, W2, H1, H2;

public Texture2D SourceTexture1, SourceTexture2;

private Texture2D _subTexture1, _subTexture2;

void Start()

{

    Color[] colorPixels = source1.GetPixels(X1, Y1, W1, H1);
    subTexture1 = new Texture2D(W1, H1);
    subTexture1.SetPixels(colorPixels);
    subTexture1.Apply();

    colorPixels = source2.GetPixels(X2, Y2, W2, H2);
    subTexture2 = new Texture2D(W2, H2);
    subTexture2.SetPixels(colorPixels);
    subTexture2.Apply();

}
```

```
public float X1, X2, Y1, Y2, W1, W2, H1, H2;

public Texture2D SourceTexture1, SourceTexture2;

private Texture2D _subTexture1, _subTexture2;

void Start()

{
    _subTexture1 = extractSubTexture(SourceTexture1, X1, Y1, W1,
H1);

    _subTexture2 = extractSubTexture(SourceTexture2, X2, Y2, W2,
H2);

}

private Texture2D extractSubTexture(Texture2D source, int x, int y,

                                    int w,   int h)

{
   Color[] colorPixels = source.GetPixels(x, y, w, h);
   Texture2D subTexture = new Texture2D(w, h);
   subTexture.SetPixels(colorPixels);
   subTexture.Apply();

   return subTexture;
}
```

No usar brackets para if, else, for, foreach de una sola línea.
Usar en cambio, un espacio abajo.

Razón: Extensión.

```
if(_life < MinLife)
    _life = MinLife;
else if(_life > MaxLife)
    _life = MaxLife;

for(int i = 0; i < HeroesAmount; i++)
    _heroes[i].Initialize();

foreach(AttackData attackData in AttackDatas)
        attackData.Reset();
```

Utilizar brackets abriendo siempre a la izquierda.

Razón: Claridad. Se ve más claramente donde cierra cada bracket que abre.

```
public void HandleAttackInput(bool isAttacking, int facing){
    if(_inputUser.GetPunchButtonDown())
        handleInputAttack(isAttacking, facing);
    else if(!isAttacking){
        if(_inputUser.GetSpecialButtonDown()){
            float moveH = _inputUser.GetMoveH();

            if((Mathf.Abs(moveH) > 0) && (Mathf.Sign(moveH) == facing))
                _attackCallback(Attack.Names.TornadoKick);
        }
    }
}
```

```
    public void HandleAttackInput(bool isAttacking, int facing)
    {
        if(_inputUser.GetPunchButtonDown())
            handleInputAttack(isAttacking, facing);
        else if(!isAttacking)
        {
            if(_inputUser.GetSpecialButtonDown())
            {
                float moveH = _inputUser.GetMoveH();

                if((Mathf.Abs(moveH) > 0) && (Mathf.Sign(moveH) ==
facing))
                    _attackCallback(Attack.Names.TornadoKick);
            }
        }
    }
```

Nombres claros de acuerdo a tipo de variable o método. Bool adjetivo/condición, variable/clase sustantivo, método verbo, colecciones en plural, enums en plural.

Razón: Claridad y mantenibilidad. Entender mejor de qué representa o hace la variable o método. El procesamiento se hace más rápido a través del tiempo. El código no se hace más claro.

```
public enum InGameStates
{
    Restarting,
    Playing,
    GameOver,
}
```

```
public List<ComboAttack> ComboAttacks;

private bool _grounded = false;

private bool _requiresUpdate = true;

private float _speed = 5.5f;

public void CalculateBounds() { ... }
```

Escribe comentarios para aclarar lo que hace código complejo.

- **Razón**: Claridad y mantenibilidad. Esto permitirá a otros y a ti mismo en el futuro entender lo que hiciste y/o por qué.

```
    static public Vector3 GetAimToWorldPoint(Vector3 planeNormal, Vector3 planePoint, Vector3
screenPoint)
    {
        Vector3 targetPoint = new Vector3();

        // Generate a plane that intersects the release spot transform's position with a forward
normal.
        Plane bowPlane = new Plane(planeNormal, planePoint);

        // Generate a ray from the cursor position
        Ray ray = Camera.main.ScreenPointToRay(screenPoint);

        // Determine the point where the cursor ray intersects the plane.
        // This will be the point that the object must look towards to be looking at the mouse.
        // Raycasting to a Plane object only gives us a distance, so we'll have to take the
distance,
        //   then find the point along that ray that meets that distance.  This will be the
point
        //   to look at.
        float hitdistance = 0.0f;
        // If the ray is parallel to the plane, Raycast will return false.
        if(bowPlane.Raycast(ray, out hitdistance))
        {
            // Get the point along the ray that hits the calculated distance.
            targetPoint = ray.GetPoint(hitdistance);
        }

        return targetPoint;
```

Escribir código de Warning y Error.

Razón: Claridad. Ayudar y prevenir el mal uso del código.

```
private int getFxNameIndex(FxAndSounds.FxsNames fxName)
{
    int index = -1;

    if(_indexByFxName.ContainsKey(fxName))
        index = _indexByFxName[fxName];
    else
        Debug.LogError("FxName: " + fxName.ToString() + " not
found.                                  Please check if name was
fxName was added to the                                 FXs array.");

    return index;
}
```

```
    private void open()
    {
        if(!IsOpen)
        {
            _timeCount = 0;
            _state = FadeStates.FadingIn;
        }
        else
            Debug.LogWarning("Attempted to open an already
open                                    pop up.");
    }
```

Usar Namespaces para diferenciar clases de otras con el mismo nombre.

Razón: Claridad y orden. Se utiliza para diferenciar clases que se llamen igual en diferentes contextos.

Cómo se crean:

```
namespace ThirdPersonGame
{
    public class HeroController
    {
            public void Initialize() { /*...*/ }
    }
}

namespace FirstPersonGame
{
    public class HeroController
    {
            public void Initialize() { /*...*/ }
    }
}
```

Cómo se usa:

```
using ThirdPersonGame;

public class GameManager
{
        private void HeroController _heroController;

        public void Initialize()
        {
                _heroController = new HeroController();
                _heroController.Initialize();
        }
}
```

==========

```
public class GameManager()
{
        private ThirdPersonGame.HeroController _heroController;

        public void Initialize()
        {
            _heroController = new ThirdPersonGame.HeroController();
            _heroController.Initialize();
        }
}
```

Evitar escribir funciones y procedimientos demasiado largos.

Razón: Claridad y mantenibilidad. Según estudios piscológicos, la mente humana no es capaz de procesar más de 6 ó 7 detalles diferentes a la vez. Las funciones demasiado largas suelen contener un número de detalles superior a este límite. Ello dificulta su legibilidad y comprensión y por tanto, su mantenimiento. No importa que las expectativas iniciales de reutilización de estas funciones sean prácticamente nulas. En general, si dos trozos de código pueden aparecer juntos en una sola función o separados en dos subfunciones, la opción recomendada siempre es separarlos. Ejemplo:

```
private void handleLocomotion()
{
   handlePlayerMovement();
   handleGrounding();
   handleGroundJump();
}

private void handlePlayerMovement() { … }

private void handleGrounding() { … }

private void handleGroundJump() { … }
```

Checar por ciertas condiciones en intervalos de tiempo más extensos y no cada frame (en Update).

- **Razón:** Eficiencia**.**

  Ejemplo:

```
public float CheckEnemyNearDelay;

private float _timeCount = 0;

void Start ()
{
     _timeCount = CheckEnemyNearDelay;
}

void Update()
{
     _timeCount -= Time.deltaTime;
     if(_timeCount <= 0)
     {
          checkEnemyNear();
          _timeCount = CheckEnemyNearDelay;
     }
}

void checkEnemyNear ()
{
     //......
}
```

Evitar la constante creación y destrucción de GameObjects iguales en tiempo de ejecución.

Razón: Eficiencia. Pueden provocar inestabilidad en el framerate al pedir y liberar memoria. **Ejemplo:**

```
 public int PoolSize = 5;

 public GameObject PoolObjectPrefab;
public Transform PoolContainer;

private List<GameObject> _poolInstances = new List<GameObject>();

void Start ()
{
    createPool();
}

public void createPool()
{
    for (int i = 0; i < PoolSize; i++)
        createInstance();
}

private GameObject createInstance()
{
    GameObject instance = Instantiate(PoolObjectPrefab, Vector2.zero,
                                      Quaternion.identity) as GameObject;
    instance.SetActive(false);
    instance.transform.parent = PoolContainer;
    _poolInstances.Add(instance);

    return instance;
}
```

```
 public GameObject NextObject(Vector3 pos)
 {
    GameObject nextInstance = null;

    foreach (GameObject instance in _poolInstances)
    {
        if(instance.activeSelf == false)
            nextInstance = instance;
    }

    if(nextInstance == null)
        nextInstance = createInstance();

    nextInstance.transform.position = pos;
    nextInstance.SetActive(true);

    return nextInstance;
}

public class PoolObject : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        other.gameObject.SetActive(false);
    }
}
```

Hacer todos los cálculos de cada frame (en Update) relativos al tiempo.

Razón: Esto evita que en equipos que vayan a mayor o menor framerate, los cálculos se hagan más o menos seguido, resultando en comportamiento distinto.

Ejemplo:

```
public float SpeedX = 10;

void Update()
{
    float movementX = Time.deltaTime * SpeedX;
    transform.Translate(movementX, 0, 0);
}
```

Se debe, en lo posible, hacer una vez todo cálculo que sea periódico, y sobretodo cada frame.

De ser posible, cambiar las divisiones por multiplicaciones.

Razón: Eficiencia.

Ejemplo:

```
const float HALF_FACTOR = 0.5f;

public float BaseSpeed = 10;
public float SpeedDivider = 3;

private float _speedFactor;

void Start()
{
    _speedFactor = 1/SpeedDivider;
}

void Update()
{
    float speed = BaseSpeed * HALF_FACTOR * _speedFactor *
Time.deltaTime;
    //.......
}
```

De ser posible, utilizar distancias cuadradas en comparaciones.

Razón: Eficiencia. Evitar cálculos de raíz cuadrada.

Ejemplo:

```
public Transform Target;
public float CloseSquareDistance = 5.0F;

void Update()
{
    if (Target)
    {
        Vector3 vectorToTarget = Target.position -
transform.position;
        float sqrLen = vectorToTarget.sqrMagnitude;

        if (sqrLen < CloseSquareDistance)
            print("The other transform is close to me!");

    }
}
```

Algunos conceptos importantes para programar bien

Evitar las dependencias circulares. Esto es, cuando la clase A usa a la B y B usa a A.

Algunas razones:

- **Crean alto acople**. Ambas clases se deben **recompilar** cada vez que alguna se cambia.
- **Previene el static linking** (copiar rutinas de librería en el ejecutable) porque B depende de A pero A no puede ser ensamblado hasta que B se complete.
- Objetos con una gran cantidad de referencias circulares son usualmente **objetos todopoderosos**. Aún si no lo son, tienen a conducir a **código sphaghetti** (códido difícil de seguir).
- Son en general **confusas** e incrementan la carga cognitiva cuando alguien intenta entender cómo funciona el programa.

Modularidad

La modularidad es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo.

Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de unas entradas y salidas bien definidas.

Evitar objetos todopoderosos

La idea básica detrás de la Programación Estructurada es: un gran problema se divide en muchos pequeños problemas (estrategia Divide y Vencerás) y las soluciones son creadas para cada uno de ellos. Una vez que los pequeños problemas han sido resueltos, el gran problema ha sido resuelto como un todo. Sin embargo hay un solo objeto el cual necesita saber todo: el objeto en sí. De esta manera, hay un solo grupo de problemas que el objeto debe resolver: sus propios problemas.

El Código del Objeto todopoderoso no sigue esta regla. En su lugar, la funcionalidad entera del programa está codificada en un solo objeto que hace todo, el cual mantiene toda la información del programa entero y contiene todos los métodos y subrutinas para manipular los datos. Como el objeto contiene muchos datos y requiere muchos métodos, su rol en el programa se convierte en Objeto Todopoderoso (Abarca todo). En lugar de objetos comunicándose entre ellos directamente, los objetos en el programa se cuelgan del Objeto Todopoderoso para manejar su información e interacción. Como el Objeto Todopoderoso es referenciado por casi todo el código, el mantenimiento se vuelve mucho más difícil, que el diseño del código de un programa mejor dividido.

Acoplamiento informático

El acoplamiento informático indica el nivel de dependencia entre las unidades de software de un sistema informático, es decir, el grado en que una unidad puede funcionar sin recurrir a otras; dos funciones son absolutamente independientes entre sí (el nivel más bajo de acoplamiento) cuando una puede hacer su trabajo completamente sin recurrir a la otra. En este caso se dice que ambas están desacopladas.

Un ejemplo simple de acoplamiento es cuando un componente accede directamente a un dato que pertenece a otro componente. En ese caso, el resultado del comportamiento del componente A dependerá del valor del componente B, por lo tanto, están acoplados.

Buscar bajo acoplamiento

Cuanto menos dependiente sean las partes que constituyen un sistema informático, mejor será el resultado. Sin embargo, es imposible un desacoplamiento total de las unidades.

Por ello, el objetivo final del diseño de software es reducir al máximo el acoplamiento entre componentes. Para ello, lo más importante es saber eliminar el acoplamiento que no sea funcional o arquitectónico.

El caso del acoplamiento funcional, puede ser por ejemplo que un componente de cálculo de probabilidades dependa de un componente de cálculo matemático básico, ya que para calcular probabilidades será necesario aplicar fórmulas matemáticas.

El bajo acoplamiento permite:

* Mejorar la mantenibilidad de las unidades de software.

* Aumentar la reutilización de las unidades de software.

* Evitar el efecto onda, ya que un defecto en una unidad puede propagarse a otras, haciendo incluso más difícil de detectar dónde está el problema.

* Minimiza el riesgo de tener que cambiar múltiples unidades de software cuando se debe alterar una.

Cohesión

La cohesión tiene que ver con que cada módulo del sistema se refiera a un único proceso o entidad. A mayor cohesión, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

En el diseño estructurado, se logra alta cohesión cuando cada módulo (función o procedimiento) realiza una única tarea trabajando sobre una sola estructura de datos. Un test que se suele hacer a los módulos funcionales para ver si son cohesivos es analizar que puedan describirse con una oración simple, con un solo verbo activo. Si hay más de un verbo activo en la descripción del procedimiento o función, deberíamos analizar su partición en más de un módulo, y volver a hacer el test.

Las clases tendrán alta cohesión cuando se refieran a una única entidad. Podemos garantizar una fuerte cohesión disminuyendo al mínimo las responsabilidades de una clase: si una clase tiene muchas responsabilidades probablemente haya que dividirla en dos o más. Y el test a aplicar sería ver si podemos describir a la clase con una oración simple que tenga un único sustantivo en el sujeto.

Ejemplo de código de alta cohesión

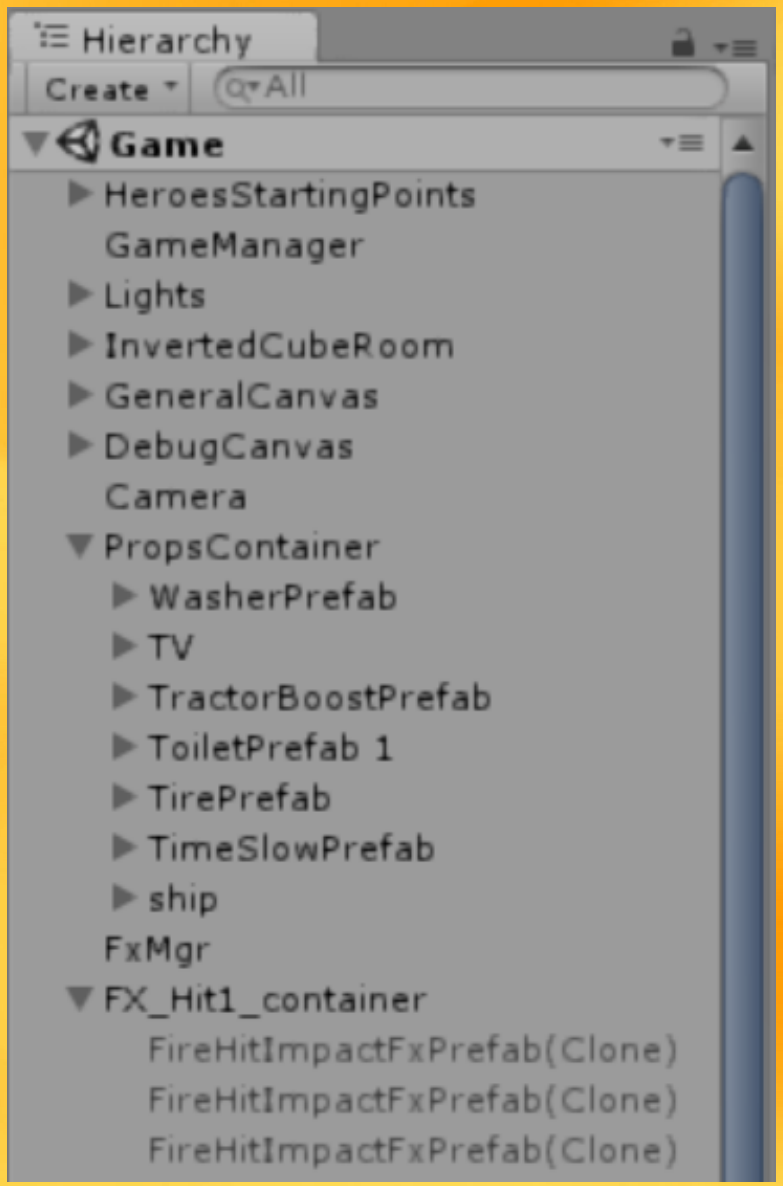
```
class EmailMessage
{
    private string _sendTo;
    private string _subject;
    private string _message;


    public EmailMessage(string to, string subject, string message)
    {
        _sendTo = to;
        _subject = subject;
        _message = message;
    }


    public void SendMessage()
    {
        // send message using sendTo, subject and message
    }
}
```

Ejemplo de código de baja cohesión

```
class EmailMessage
{
    private string _sendTo;
    private string _subject;
    private string _message;
    private string _username;


    public EmailMessage(string to, string subject, string message)
    {
        _sendTo = to;
        _subject = subject;
        _message = message;
    }


    public void SendMessage()
    {
        // send message using sendTo, subject and message
    }


    public void Login(string username, string password)
    {
        this.username = username;
        // code to login
    }
}
```

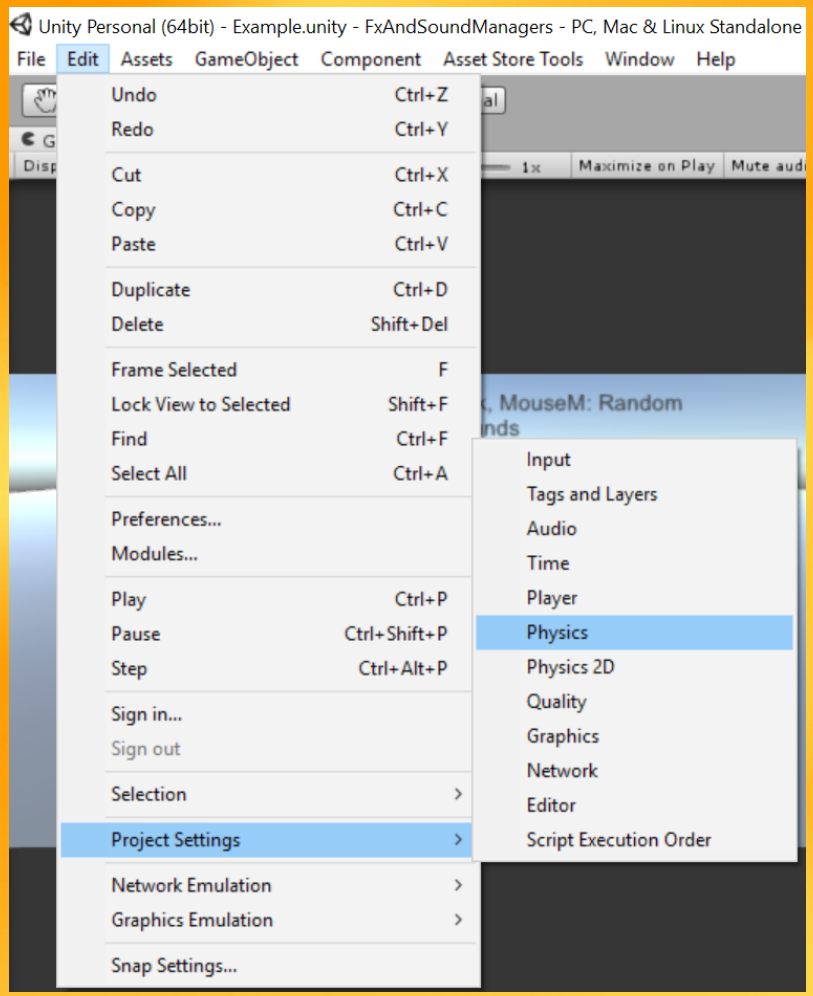
Algunas buenas prácticas para trabajar en el Editor de Unity3d

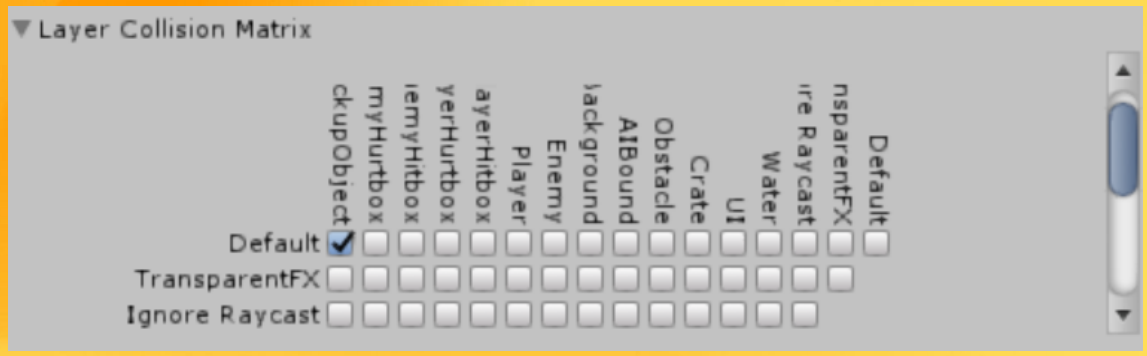
Utilizar GameObject contenedores para mantener el orden en la escena.

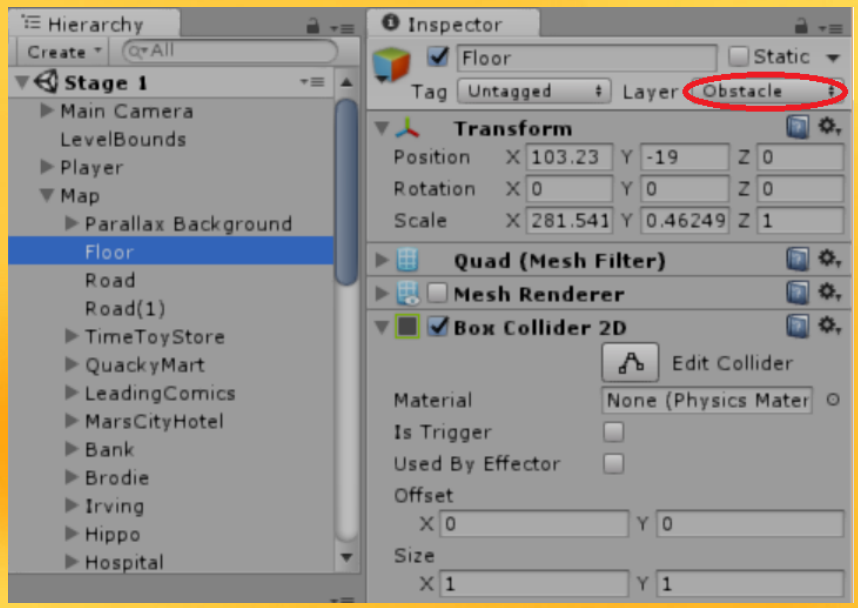


Utilizar layers específicos para controlar las colisiones.

Razón: Se hace más eficiente el chequeo de colisiones y en ocasiones, se evita tener que checar por tags.

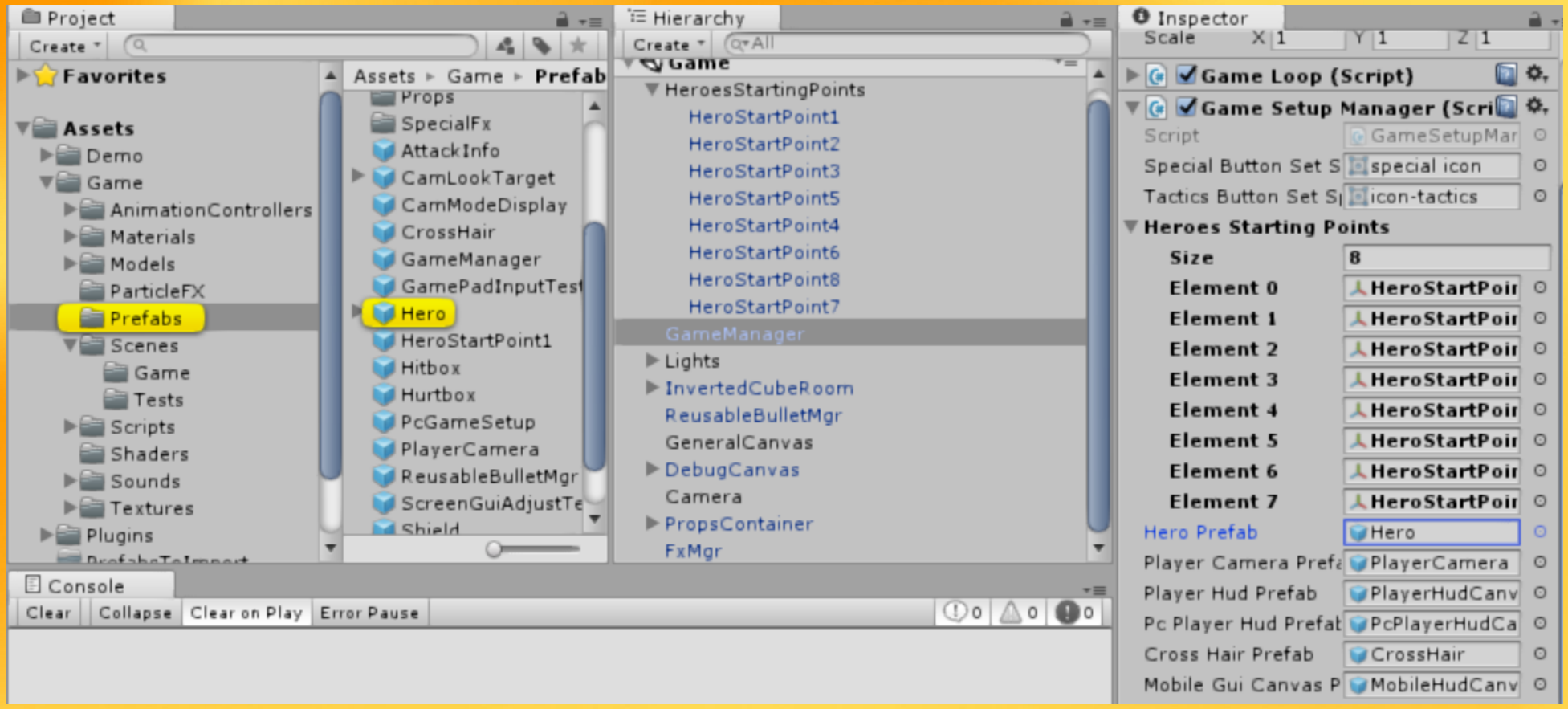






Agregar los assets necesarios desde el inspector y salvar como prefab de ser posible.

Razón: Se evitan tiempos de búsqueda y tener que agregar código con el Resources.Load( ) y su folder Resources que siempre carga al build todo lo que hay en él, se use o no.



Usar [HideInInspector] para esconder las variables públicas que no se pretende que sean modificadas desde el Inspector

```
public class ExampleClass : MonoBehaviour
{
    [HideInInspector]
    public int MyHiddenPublicVariable = 5;
}
```

Usar [SerializeField] para mostrar variables privadas en el Inspector en vez de hacerlas públicas para ello.

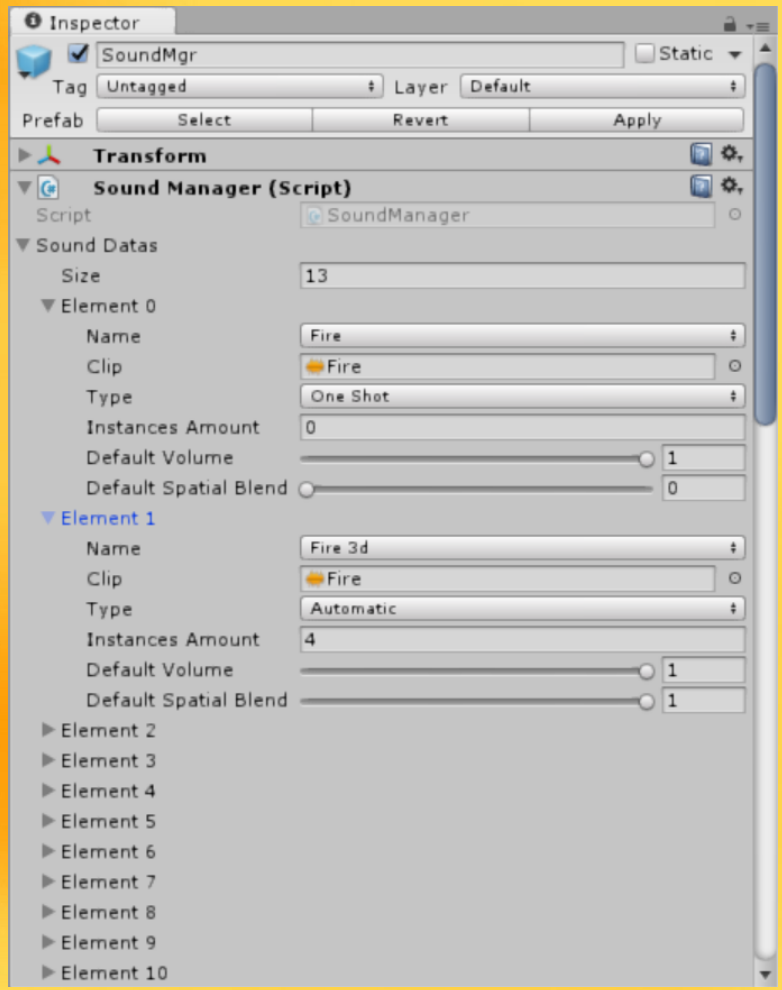
```
public class ExampleClass : MonoBehaviour
{
    [SerializeField]
    private int _myInspectablePrivateVariable = 5;
}
```

Utilizar System.Serializable para configurar elementos compuestos desde el Inspector.

Razón: Extensión, claridad, orden.

Sin conocer esta técnica lo que solemos usar son arreglos y listas en los cuales utilizamos el índice para relacionar los datos. Así es mucho más cómodo y entendible para el usuario del script. Debe ser un clase que ya que necesita heredar de object, y solo se mostrarán los mismos tipos que aparecen en el inspector en un componente normal. Se puede usar una clase serializable dentro de otra.

```
[System.Serializable]
public class SoundData
{
    public enum Types
    {
        OneShot,
        Automatic,
        Manual
    }

    public FxAndSounds.SoundsNames Name = FxAndSounds.SoundsNames.None;
    public AudioClip Clip;
    public Types Type = Types.OneShot;
    public int InstancesAmount = 3;

    [Range(0,1)]
    public float DefaultVolume = 1;

    [Range(0,1)]
    public float DefaultSpatialBlend = 0;
}
```



¡Muchas gracias!

¿Preguntas? ¿Observaciones? ¿Aportes?

Mi correo: morph.vgx@gmail.com

Mi blog: http://morphvgx.blogspot.com/