TIPS & TRICKS

Animazioni in sequenza

Può capitare di voler fare delle animazioni in sequenza usando un solo tasto, come avviene per alcuni giochi d'azione/RPG dove il giocatore premendo il tasto di attacco esegue un tipo di attacco basilare e finito, ma se esso preme il pulsante di attacco ripetutamente, il personaggio, dopo il primo attacco basilare, continua ad attaccare eseguendo animazioni diverse e più complesse. Si tratta si una serie di animazioni concatenate che sono legate all'animazione in esecuzione e in fase di conclusione.

COSA FAREMO E PERCHE'

Come nella maggior parte dei casi, esistono svariati approcci e tipologie di soluzione per raggiungere questo scopo.
Noi ne vedremo uno tra i più semplici, ovvero quello che presuppone una conoscenza di livello base dell'**Animator** che sarebbe lo strumento (Finite State Machine) che gestisce le animazioni in Unity.
L'**Animator** di Unity è uno strumento potentissimo che rappresenta uno dei pilastri dell'engine. Ma come avrete sentito dire altrove; *"la potenza è nulla senza controllo".* Essendo uno strumento di per se abbastanza complesso, è necessario studiarne il funzionamento con diverse lezioni. Dunque adesso noi andremo ad usare una soluzione che delega il controllo al codice in C# (che già dovremmo conoscere abbastanza bene).

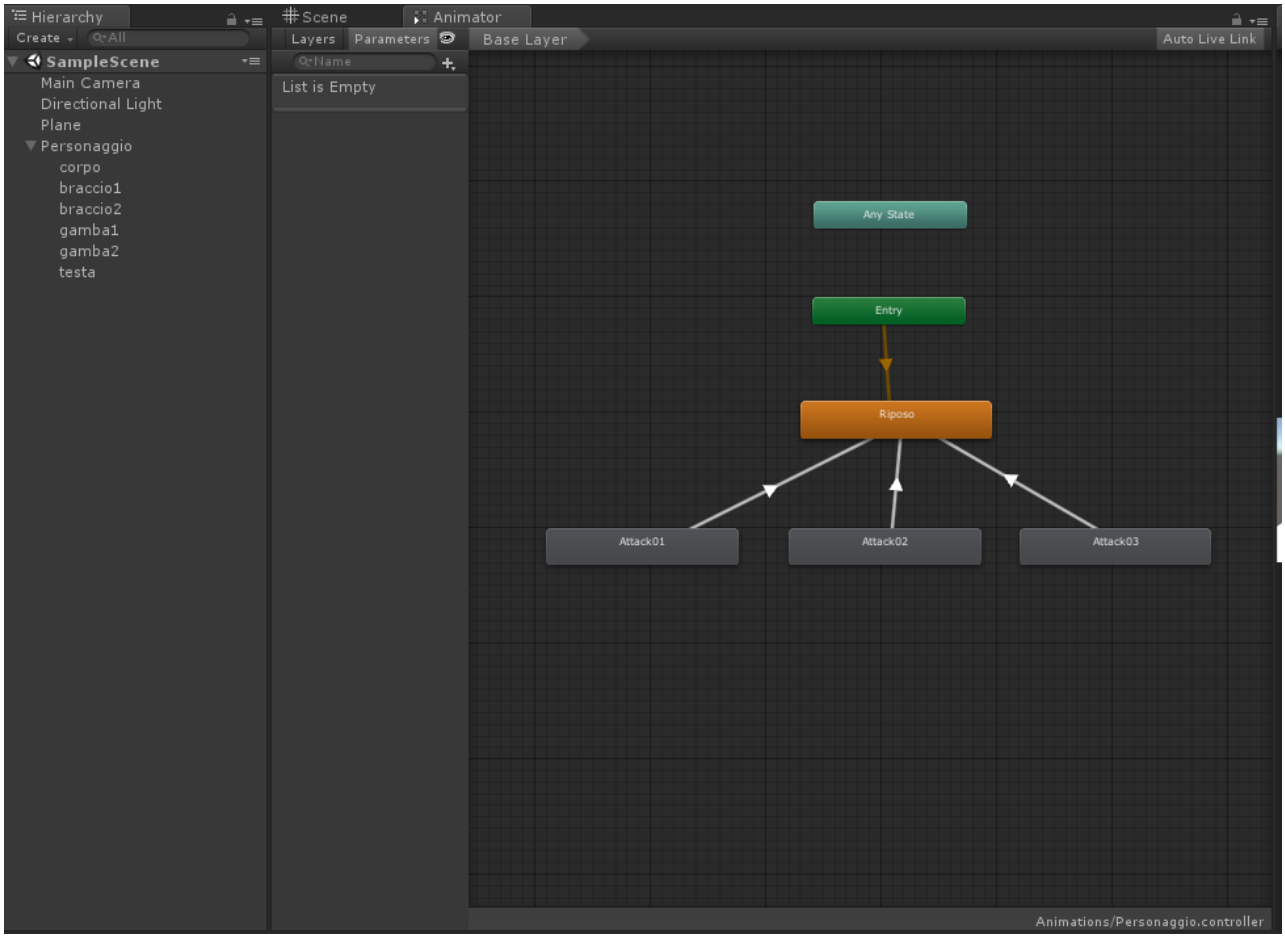
Andremo a controllare quale animazione è in esecuzione durante la pressione del tasto "*Fire01*" tramite codice, poi controlleremo anche a che punto si trova questa animazione (ovvero se sta per terminare) e andremo a far eseguire l'animazione che ne consegue tramite un CrossFade, ovvero un **passaggio graduale** dall'animazione in corso ad un'animazione successiva.

In alternativa si potrebbero usare dei valori (float, string, bool) da inviare all'**Animator** con un'istruzione tipo *"aminatore.SetFloat(nomeValore,valore)"* che ne influenzino il comportamento, delegando la maggior parte del lavoro all'**Animator**. Ma per far questo dovremmo conoscere uno strumento che non ho ancora trattato su queste pagine.
Ovviamente siete qui per imparare e devo presupporre un grado di conoscenza di Unity di livello basilare da parte di chi legge. Per questo motivo vedremo la prima soluzione, quella che secondo me è la più facile implementare e che necessita una conoscenza meno approfondita dell'Animator e che delega la maggior parte del lavoro allo script in C#.
L'argomento Animator sarà ripreso e approfondito in sede opportuna.

INIZIAMO

Se non lo abbiamo già, creiamo un Animator per il nostro personaggio e poniamo le nostre animazioni di attacco al suo interno.
Non sarà necessario **nessun passaggio** (Transitions) **tra lo stato di riposo a quello di attacco**, perché tale passaggio le effettueremo da codice. Sarà necessario solo il passaggio inverso, ovvero dai vari stati di attacco allo stato di **"Riposo"** (o il vostro stato di default che di norma potrebbe e dovrebbe essere un **Blend Tree**).
Non sarà necessario neanche specificare dei parametri particolari per il passaggio tra uno stato ad un altro. Semplicemente, al termine di una qualsiasi animazione di attacco, se non verrà premuto di nuovo il pulsante

di *Fire01*durante l'esecuzione di una delle animazioni di attacco, lo stato tornerà automaticamente su "Riposo".



Ed ecco il codice da usare per effettuare gli attacchi in sequenza.

C# 43 lines

```
//Unity3DTutorials.it
using UnityEngine;

public class AttackAnimation : MonoBehaviour {

    public Animator animator; //Il nostro Animator
    AnimatorClipInfo[] currentClipInfo; //Le info necessarie per capire quale animazione è in esecuzione
    public float clipNormalizedTime; //Il timer normalizzato (cioè da 0 a 1) del clip in esecuzione



    void Update () {

        currentClipInfo = animator.GetCurrentAnimatorClipInfo(0); //Teniamo aggiornato le info del Clip i
        clipNormalizedTime= animator.GetCurrentAnimatorStateInfo(0).normalizedTime; //Teniamo aggiornato



        if (Input.GetButtonDown("Fire1"))//Se premuto il tasto Fire1
        {


            if (currentClipInfo[0].clip.name.Equals("Riposo")) //Se in questo momento è in esecusione l'a
            {
                animator.CrossFade("Attack01", 0.1f); //Esegue l'animazione Attack01
            }

            else if (currentClipInfo[0].clip.name.Equals("Attack01"))  //Se in questo momento è in esecus
            {
                if (clipNormalizedTime > 0.75f) //Se l'animazione Attack01 è giunta almeno a tre quarti d
                    animator.CrossFade("Attack02", 0.1f);//Esegue l'animazione Attack02
            }

            else if (currentClipInfo[0].clip.name.Equals("Attack02")) //Se in questo momento è in esecusi
            {
                if (clipNormalizedTime > 0.75f) //Se l'animazione Attack02 è giunta almeno a tre quarti d
                    animator.CrossFade("Attack03", 0.1f);//Esegue l'animazione Attack03
            }

        }

    }
}
```

Come sempre il codice è commentato in ogni sua riga e istruzione, così che sia di facile comprensione.
Per chiarire l'uso dell'istruzione
if(clipNormaliedTime >0.75f)
spieghiamo che:
clipNormaliedTime è il tempo normalizzato, ovvero indica il timer del clip in corso, dove 0 è l'inizio e 1 è la fine.
Dunque quando il clip sarà a 0.5 significa che si trova alla metà esatta della sua esecuzione totale, mentre quando si trova a 0.75 significa che si trova a tre quarti della sua esecuzione totale.
In questo modo potrete controllare al millesimo di secondo il momento in cui volete che la successiva animazione dell'attacco sia eseguita.
E' altresì ovvio che potrete aggiungere una quantità infinita di animazioni nella vostra sequenza.

Potete scaricare il package di esempio funzionante (versione Unity 2018 2.2f1) che ho usato per testare questo sistema e che contiene tutto il necessario (attenzione al copyright sul modello del personaggio)da questo link: MultipleAnimationSequence.unitypackage

UI – Migliorare la fluidità di uno ScrollRect

Uno dei problemi noti delle UI di Unity sta nella scarsa fluidità degli *scrollRect* e delle UI in generale, come per esempio durante il *drag&Drop* degli elementi come icone e immagini. Il *lag* aumenta in modo esponenziale se utilizzati su dispositivi mobili sopratutto in presenza di tanti elementi all'interno di un pannello "scrollabile". Dovremmo trovare qualche stratagemma per migliorare l'effetto visivo dello scrolling. Per ovviare a questo problema ho raccolto tre metodi differenti che, se usati contemporaneamente, potranno aumentare sensibilmente le performance delle vostre UI e rendere l'aspetto del movimento più veloce e fluido.

Vi ricordo che se non conoscete già le UI, questo articolo potrebbe essere troppo avanzato e vi invito ad approfondire l'argomento nell'apposita sessione sulle UI di Unity.

Opzione 1

Abbiamo uno script alternativo allo ScrollRect, che io ho chiamato fantasiosamente *ScrollRect2* che possiamo usare al posto del normale ScrollRect.

Lo script è fondamentalmente identico all'originale ScrollRect, ma con due modifiche sostanziali.
Una alla riga 70, dove abbiamo aggiunto due righe e l'altro alla riga 356.
Non c'è molto da dire, semplicemente, usate questo al posto del normale ScrollRect e siate felici.
Usando quest'opzione i miglioramenti di velocità di scorrimento si potranno notare sopratutto su dispositivi mobili dove avremo delle viewport molto grandi, piene di elementi o con immagini molto grandi da scorrere.
In realtà non aumenteranno le performance ma potrete settare la velocità di scorrimento che di norma è molto lento sui dispositivi mobili.

C# 810 lines

```
using System;
using UnityEngine.Events;
using UnityEngine.EventSystems;

namespace UnityEngine.UI
{
    [AddComponentMenu("UI/Scroll Rect", 37)]
    [SelectionBase]
    [ExecuteInEditMode]
    [DisallowMultipleComponent]
    [RequireComponent(typeof(RectTransform))]
    public class ScrollRect2 : UIBehaviour, IInitializePotentialDragHandler, IBeginDragHandler, IEndDragHandler,
    {
        public enum MovementType
        {
            Unrestricted, // Unrestricted movement -- can scroll forever
            Elastic, // Restricted but flexible -- can go past the edges, but springs back in place
            Clamped, // Restricted movement where it's not possible to go past the edges
        }

        public enum ScrollbarVisibility
        {
            Permanent,
            AutoHide,
            AutoHideAndExpandViewport,
        }

        [Serializable]
        public class ScrollRectEvent : UnityEvent<Vector2> { }

        [SerializeField]
        private RectTransform m_Content;
        public RectTransform content { get { return m_Content; } set { m_Content = value; } }

        [SerializeField]
        private bool m_Horizontal = true;
        public bool horizontal { get { return m_Horizontal; } set { m_Horizontal = value; } }

        [SerializeField]
        private bool m_Vertical = true;
        public bool vertical { get { return m_Vertical; } set { m_Vertical = value; } }

        [SerializeField]
        private MovementType m_MovementType = MovementType.Elastic;
        public MovementType movementType { get { return m_MovementType; } set { m_MovementType = value; } }

        [SerializeField]
        private float m_Elasticity = 0.1f; // Only used for MovementType.Elastic
        public float elasticity { get { return m_Elasticity; } set { m_Elasticity = value; } }

        [SerializeField]
        private bool m_Inertia = true;
        public bool inertia { get { return m_Inertia; } set { m_Inertia = value; } }

        [SerializeField]
        private float m_DecelerationRate = 0.135f; // Only used when inertia is enabled
        public float decelerationRate { get { return m_DecelerationRate; } set { m_DecelerationRate = value; } }

        [SerializeField]
        private float m_ScrollSensitivity = 1.0f;
        public float scrollSensitivity { get { return m_ScrollSensitivity; } set { m_ScrollSensitivity = value; }

        [SerializeField]
        private RectTransform m_Viewport;
        public RectTransform viewport { get { return m_Viewport; } set { m_Viewport = value; SetDirtyCaching(); }

```

```
    /// <summary>
    /// Aggiunta allo script originale
    /// </summary>
    [SerializeField]
    private float m_ScrollFactor = 1.0f;
    public float scrollFactor { get { return m_ScrollFactor; } set { m_ScrollFactor = value; } }

    [SerializeField]
    private Scrollbar m_HorizontalScrollbar;
    public Scrollbar horizontalScrollbar
    {
        get
        {
            return m_HorizontalScrollbar;
        }
        set
        {
            if (m_HorizontalScrollbar)
                m_HorizontalScrollbar.onValueChanged.RemoveListener(SetHorizontalNormalizedPosition);
            m_HorizontalScrollbar = value;
            if (m_HorizontalScrollbar)
                m_HorizontalScrollbar.onValueChanged.AddListener(SetHorizontalNormalizedPosition);
            SetDirtyCaching();
        }
    }

    [SerializeField]
    private Scrollbar m_VerticalScrollbar;
    public Scrollbar verticalScrollbar
    {
        get
        {
            return m_VerticalScrollbar;
        }
        set
        {
            if (m_VerticalScrollbar)
                m_VerticalScrollbar.onValueChanged.RemoveListener(SetVerticalNormalizedPosition);
            m_VerticalScrollbar = value;
            if (m_VerticalScrollbar)
                m_VerticalScrollbar.onValueChanged.AddListener(SetVerticalNormalizedPosition);
            SetDirtyCaching();
        }
    }

    [SerializeField]
    private ScrollbarVisibility m_HorizontalScrollbarVisibility;
    public ScrollbarVisibility horizontalScrollbarVisibility { get { return m_HorizontalScrollbarVisibility; } set { m_HorizontalScrollbarVisibility = value; SetDirtyCaching(); } }

    [SerializeField]
    private ScrollbarVisibility m_VerticalScrollbarVisibility;
    public ScrollbarVisibility verticalScrollbarVisibility { get { return m_VerticalScrollbarVisibility; } set { m_VerticalScrollbarVisibility = value; SetDirtyCaching(); } }

    [SerializeField]
    private float m_HorizontalScrollbarSpacing;
    public float horizontalScrollbarSpacing { get { return m_HorizontalScrollbarSpacing; } set { m_HorizontalScrollbarSpacing = value; SetDirty(); } }

    [SerializeField]
    private float m_VerticalScrollbarSpacing;
    public float verticalScrollbarSpacing { get { return m_VerticalScrollbarSpacing; } set { m_VerticalScrollbarSpacing = value; SetDirty(); } }

    [SerializeField]
    private ScrollRectEvent m_OnValueChanged = new ScrollRectEvent();
    public ScrollRectEvent onValueChanged { get { return m_OnValueChanged; } set { m_OnValueChanged = value; } }
```

```
131
132         // The offset from handle position to mouse down position
133         private Vector2 m_PointerStartLocalCursor = Vector2.zero;
134         private Vector2 m_ContentStartPosition = Vector2.zero;
135
136         private RectTransform m_ViewRect;
137
138         protected RectTransform viewRect
139         {
140             get
141             {
142                 if (m_ViewRect == null)
143                     m_ViewRect = m_Viewport;
144                 if (m_ViewRect == null)
145                     m_ViewRect = (RectTransform)transform;
146                 return m_ViewRect;
147             }
148         }
149
150         private Bounds m_ContentBounds;
151         private Bounds m_ViewBounds;
152
153         private Vector2 m_Velocity;
154         public Vector2 velocity { get { return m_Velocity; } set { m_Velocity = value; } }
155
156         private bool m_Dragging;
157
158         private Vector2 m_PrevPosition = Vector2.zero;
159         private Bounds m_PrevContentBounds;
160         private Bounds m_PrevViewBounds;
161         [NonSerialized]
162         private bool m_HasRebuiltLayout = false;
163
164         private bool m_HSliderExpand;
165         private bool m_VSliderExpand;
166         private float m_HSliderHeight;
167         private float m_VSliderWidth;
168
169         [System.NonSerialized] private RectTransform m_Rect;
170         private RectTransform rectTransform
171         {
172             get
173             {
174                 if (m_Rect == null)
175                     m_Rect = GetComponent<RectTransform>();
176                 return m_Rect;
177             }
178         }
179
180         private RectTransform m_HorizontalScrollbarRect;
181         private RectTransform m_VerticalScrollbarRect;
182
183         private DrivenRectTransformTracker m_Tracker;
184
185         protected ScrollRect2()
186         {
187             flexibleWidth = -1;
188         }
189
190         public virtual void Rebuild(CanvasUpdate executing)
191         {
192             if (executing == CanvasUpdate.Prelayout)
193             {
194                 UpdateCachedData();
195             }
196
197             if (executing == CanvasUpdate.PostLayout)
198             {
199                 UpdateBounds();
200                 UpdateScrollbars(Vector2.zero);
```

```
            UpdatePrevData();

            m_HasRebuiltLayout = true;
        }
    }

    public virtual void LayoutComplete()
    { }

    public virtual void GraphicUpdateComplete()
    { }

    void UpdateCachedData()
    {
        Transform transform = this.transform;
        m_HorizontalScrollbarRect = m_HorizontalScrollbar == null ? null : m_HorizontalScrollbar.transform as RectTransform;
        m_VerticalScrollbarRect = m_VerticalScrollbar == null ? null : m_VerticalScrollbar.transform as RectTransform;

        // These are true if either the elements are children, or they don't exist at all.
        bool viewIsChild = (viewRect.parent == transform);
        bool hScrollbarIsChild = (!m_HorizontalScrollbarRect || m_HorizontalScrollbarRect.parent == transform);
        bool vScrollbarIsChild = (!m_VerticalScrollbarRect || m_VerticalScrollbarRect.parent == transform);
        bool allAreChildren = (viewIsChild && hScrollbarIsChild && vScrollbarIsChild);

        m_HSliderExpand = allAreChildren && m_HorizontalScrollbarRect && horizontalScrollbarVisibility == ScrollbarVisibility.AutoHideAndExpandViewpo
        m_VSliderExpand = allAreChildren && m_VerticalScrollbarRect && verticalScrollbarVisibility == ScrollbarVisibility.AutoHideAndExpandViewport;
        m_HSliderHeight = (m_HorizontalScrollbarRect == null ? 0 : m_HorizontalScrollbarRect.rect.height);
        m_VSliderWidth = (m_VerticalScrollbarRect == null ? 0 : m_VerticalScrollbarRect.rect.width);
    }

    protected override void OnEnable()
    {
        base.OnEnable();

        if (m_HorizontalScrollbar)
            m_HorizontalScrollbar.onValueChanged.AddListener(SetHorizontalNormalizedPosition);
        if (m_VerticalScrollbar)
            m_VerticalScrollbar.onValueChanged.AddListener(SetVerticalNormalizedPosition);

        CanvasUpdateRegistry.RegisterCanvasElementForLayoutRebuild(this);
    }

    protected override void OnDisable()
    {
        CanvasUpdateRegistry.UnRegisterCanvasElementForRebuild(this);

        if (m_HorizontalScrollbar)
            m_HorizontalScrollbar.onValueChanged.RemoveListener(SetHorizontalNormalizedPosition);
        if (m_VerticalScrollbar)
            m_VerticalScrollbar.onValueChanged.RemoveListener(SetVerticalNormalizedPosition);

        m_HasRebuiltLayout = false;
        m_Tracker.Clear();
        m_Velocity = Vector2.zero;
        LayoutRebuilder.MarkLayoutForRebuild(rectTransform);
        base.OnDisable();
    }

    public override bool IsActive()
    {
        return base.IsActive() && m_Content != null;
    }

    private void EnsureLayoutHasRebuilt()
    {
        if (!m_HasRebuiltLayout && !CanvasUpdateRegistry.IsRebuildingLayout())
            Canvas.ForceUpdateCanvases();
    }
```

```
269
270         public virtual void StopMovement()
271         {
272             m_Velocity = Vector2.zero;
273         }
274
275         public virtual void OnScroll(PointerEventData data)
276         {
277             if (!IsActive())
278                 return;
279
280             EnsureLayoutHasRebuilt();
281             UpdateBounds();
282
283             Vector2 delta = data.scrollDelta;
284             // Down is positive for scroll events, while in UI system up is positive.
285             delta.y *= -1;
286             if (vertical && !horizontal)
287             {
288                 if (Mathf.Abs(delta.x) > Mathf.Abs(delta.y))
289                     delta.y = delta.x;
290                 delta.x = 0;
291             }
292             if (horizontal && !vertical)
293             {
294                 if (Mathf.Abs(delta.y) > Mathf.Abs(delta.x))
295                     delta.x = delta.y;
296                 delta.y = 0;
297             }
298
299             Vector2 position = m_Content.anchoredPosition;
300             position += delta * m_ScrollSensitivity;
301             if (m_MovementType == MovementType.Clamped)
302                 position += CalculateOffset(position - m_Content.anchoredPosition);
303
304             SetContentAnchoredPosition(position);
305             UpdateBounds();
306         }
307
308         public virtual void OnInitializePotentialDrag(PointerEventData eventData)
309         {
310             if (eventData.button != PointerEventData.InputButton.Left)
311                 return;
312
313             m_Velocity = Vector2.zero;
314         }
315
316         public virtual void OnBeginDrag(PointerEventData eventData)
317         {
318             if (eventData.button != PointerEventData.InputButton.Left)
319                 return;
320
321             if (!IsActive())
322                 return;
323
324             UpdateBounds();
325
326             m_PointerStartLocalCursor = Vector2.zero;
327             RectTransformUtility.ScreenPointToLocalPointInRectangle(viewRect, eventData.position, eventData.pressEventCamera, out m_PointerStartLocalCursor)
328             m_ContentStartPosition = m_Content.anchoredPosition;
329             m_Dragging = true;
330         }
331
332         public virtual void OnEndDrag(PointerEventData eventData)
333         {
334             if (eventData.button != PointerEventData.InputButton.Left)
335                 return;
```

```
336
337             m_Dragging = false;
338         }
339
340         public virtual void OnDrag(PointerEventData eventData)
341         {
342             if (eventData.button != PointerEventData.InputButton.Left)
343                 return;
344
345             if (!IsActive())
346                 return;
347
348             Vector2 localCursor;
349             if (!RectTransformUtility.ScreenPointToLocalPointInRectangle(viewRect, eventData.position, eventData.pressEventCamera, out localCursor))
350                 return;
351
352             UpdateBounds();
353
354
355             var pointerDelta = localCursor - m_PointerStartLocalCursor;
356             pointerDelta *= m_ScrollFactor;    // Questa linea è nuova!
357             Vector2 position = m_ContentStartPosition + pointerDelta;
358
359
360             // Offset to get content into place in the view.
361             Vector2 offset = CalculateOffset(position - m_Content.anchoredPosition);
362             position += offset;
363             if (m_MovementType == MovementType.Elastic)
364             {
365                 if (offset.x != 0)
366                     position.x = position.x - RubberDelta(offset.x, m_ViewBounds.size.x);
367                 if (offset.y != 0)
368                     position.y = position.y - RubberDelta(offset.y, m_ViewBounds.size.y);
369             }
370
371             SetContentAnchoredPosition(position);
372         }
373
374         protected virtual void SetContentAnchoredPosition(Vector2 position)
375         {
376             if (!m_Horizontal)
377                 position.x = m_Content.anchoredPosition.x;
378             if (!m_Vertical)
379                 position.y = m_Content.anchoredPosition.y;
380
381             if (position != m_Content.anchoredPosition)
382             {
383                 m_Content.anchoredPosition = position;
384                 UpdateBounds();
385             }
386         }
387
388         protected virtual void LateUpdate()
389         {
390             if (!m_Content)
391                 return;
392
393             EnsureLayoutHasRebuilt();
394             UpdateScrollbarVisibility();
395             UpdateBounds();
396             float deltaTime = Time.unscaledDeltaTime;
397             Vector2 offset = CalculateOffset(Vector2.zero);
398             if (!m_Dragging && (offset != Vector2.zero || m_Velocity != Vector2.zero))
399             {
400                 Vector2 position = m_Content.anchoredPosition;
401                 for (int axis = 0; axis < 2; axis++)
402                 {
403                     // Apply spring physics if movement is elastic and content has an offset from the view.
404                     if (m_MovementType == MovementType.Elastic && offset[axis] != 0)
```

```
                if (m_MovementType == MovementType.Elastic && offset[axis] != 0)
                {
                    float speed = m_Velocity[axis];
                    position[axis] = Mathf.SmoothDamp(m_Content.anchoredPosition[axis], m_Content.anchoredPosition[axis] + offset[axis], ref speed, m_
                    m_Velocity[axis] = speed;
                }
                // Else move content according to velocity with deceleration applied.
                else if (m_Inertia)
                {
                    m_Velocity[axis] *= Mathf.Pow(m_DecelerationRate, deltaTime);
                    if (Mathf.Abs(m_Velocity[axis]) < 1)
                        m_Velocity[axis] = 0;
                    position[axis] += m_Velocity[axis] * deltaTime;
                }
                // If we have neither elaticity or friction, there shouldn't be any velocity.
                else
                {
                    m_Velocity[axis] = 0;
                }
            }

            if (m_Velocity != Vector2.zero)
            {
                if (m_MovementType == MovementType.Clamped)
                {
                    offset = CalculateOffset(position - m_Content.anchoredPosition);
                    position += offset;
                }

                SetContentAnchoredPosition(position);
            }
        }

        if (m_Dragging && m_Inertia)
        {
            Vector3 newVelocity = (m_Content.anchoredPosition - m_PrevPosition) / deltaTime;
            m_Velocity = Vector3.Lerp(m_Velocity, newVelocity, deltaTime * 10);
        }

        if (m_ViewBounds != m_PrevViewBounds || m_ContentBounds != m_PrevContentBounds || m_Content.anchoredPosition != m_PrevPosition)
        {
            UpdateScrollbars(offset);
            m_OnValueChanged.Invoke(normalizedPosition);
            UpdatePrevData();
        }
    }

    private void UpdatePrevData()
    {
        if (m_Content == null)
            m_PrevPosition = Vector2.zero;
        else
            m_PrevPosition = m_Content.anchoredPosition;
        m_PrevViewBounds = m_ViewBounds;
        m_PrevContentBounds = m_ContentBounds;
    }

    private void UpdateScrollbars(Vector2 offset)
    {
        if (m_HorizontalScrollbar)
        {
            if (m_ContentBounds.size.x > 0)
                m_HorizontalScrollbar.size = Mathf.Clamp01((m_ViewBounds.size.x - Mathf.Abs(offset.x)) / m_ContentBounds.size.x);
            else
                m_HorizontalScrollbar.size = 1;

            m_HorizontalScrollbar.value = horizontalNormalizedPosition;
        }

```

```
        if (m_VerticalScrollbar)
        {
            if (m_ContentBounds.size.y > 0)
                m_VerticalScrollbar.size = Mathf.Clamp01((m_ViewBounds.size.y - Mathf.Abs(offset.y)) / m_ContentBounds.size.y);
            else
                m_VerticalScrollbar.size = 1;

            m_VerticalScrollbar.value = verticalNormalizedPosition;
        }
    }

    public Vector2 normalizedPosition
    {
        get
        {
            return new Vector2(horizontalNormalizedPosition, verticalNormalizedPosition);
        }
        set
        {
            SetNormalizedPosition(value.x, 0);
            SetNormalizedPosition(value.y, 1);
        }
    }

    public float horizontalNormalizedPosition
    {
        get
        {
            UpdateBounds();
            if (m_ContentBounds.size.x <= m_ViewBounds.size.x)
                return (m_ViewBounds.min.x > m_ContentBounds.min.x) ? 1 : 0;
            return (m_ViewBounds.min.x - m_ContentBounds.min.x) / (m_ContentBounds.size.x - m_ViewBounds.size.x);
        }
        set
        {
            SetNormalizedPosition(value, 0);
        }
    }

    public float verticalNormalizedPosition
    {
        get
        {
            UpdateBounds();
            if (m_ContentBounds.size.y <= m_ViewBounds.size.y)
                return (m_ViewBounds.min.y > m_ContentBounds.min.y) ? 1 : 0;
            ;
            return (m_ViewBounds.min.y - m_ContentBounds.min.y) / (m_ContentBounds.size.y - m_ViewBounds.size.y);
        }
        set
        {
            SetNormalizedPosition(value, 1);
        }
    }

    private void SetHorizontalNormalizedPosition(float value) { SetNormalizedPosition(value, 0); }
    private void SetVerticalNormalizedPosition(float value) { SetNormalizedPosition(value, 1); }

    private void SetNormalizedPosition(float value, int axis)
    {
        EnsureLayoutHasRebuilt();
        UpdateBounds();
        // How much the content is larger than the view.
        float hiddenLength = m_ContentBounds.size[axis] - m_ViewBounds.size[axis];
        // Where the position of the lower left corner of the content bounds should be, in the space of the view.
        float contentBoundsMinPosition = m_ViewBounds.min[axis] - value * hiddenLength;
        // The new content localPosition, in the space of the view.
```

```
            float newLocalPosition = m_Content.localPosition[axis] + contentBoundsMinPosition - m_ContentBounds.min[axis];

            Vector3 localPosition = m_Content.localPosition;
            if (Mathf.Abs(localPosition[axis] - newLocalPosition) > 0.01f)
            {
                localPosition[axis] = newLocalPosition;
                m_Content.localPosition = localPosition;
                m_Velocity[axis] = 0;
                UpdateBounds();
            }
        }

        private static float RubberDelta(float overStretching, float viewSize)
        {
            return (1 - (1 / ((Mathf.Abs(overStretching) * 0.55f / viewSize) + 1))) * viewSize * Mathf.Sign(overStretching);
        }

        protected override void OnRectTransformDimensionsChange()
        {
            SetDirty();
        }

        private bool hScrollingNeeded
        {
            get
            {
                if (Application.isPlaying)
                    return m_ContentBounds.size.x > m_ViewBounds.size.x + 0.01f;
                return true;
            }
        }
        private bool vScrollingNeeded
        {
            get
            {
                if (Application.isPlaying)
                    return m_ContentBounds.size.y > m_ViewBounds.size.y + 0.01f;
                return true;
            }
        }

        public virtual void CalculateLayoutInputHorizontal() { }
        public virtual void CalculateLayoutInputVertical() { }

        public virtual float minWidth { get { return -1; } }
        public virtual float preferredWidth { get { return -1; } }
        public virtual float flexibleWidth { get; private set; }

        public virtual float minHeight { get { return -1; } }
        public virtual float preferredHeight { get { return -1; } }
        public virtual float flexibleHeight { get { return -1; } }

        public virtual int layoutPriority { get { return -1; } }

        public virtual void SetLayoutHorizontal()
        {
            m_Tracker.Clear();

            if (m_HSliderExpand || m_VSliderExpand)
            {
                m_Tracker.Add(this, viewRect,
                    DrivenTransformProperties.Anchors |
                    DrivenTransformProperties.SizeDelta |
                    DrivenTransformProperties.AnchoredPosition);

                // Make view full size to see if content fits.
                viewRect.anchorMin = Vector2.zero;
                viewRect.anchorMax = Vector2.one;
                viewRect.sizeDelta = Vector2.zero;
                viewRect.anchoredPosition = Vector2.zero;
```

```
610
611             // Recalculate content layout with this size to see if it fits when there are no scrollbars.
612             LayoutRebuilder.ForceRebuildLayoutImmediate(content);
613             m_ViewBounds = new Bounds(viewRect.rect.center, viewRect.rect.size);
614             m_ContentBounds = GetBounds();
615         }
616
617         // If it doesn't fit vertically, enable vertical scrollbar and shrink view horizontally to make room for it.
618         if (m_VSliderExpand && vScrollingNeeded)
619         {
620             viewRect.sizeDelta = new Vector2(-(m_VSliderWidth + m_VerticalScrollbarSpacing), viewRect.sizeDelta.y);
621
622             // Recalculate content layout with this size to see if it fits vertically
623             // when there is a vertical scrollbar (which may reflowed the content to make it taller).
624             LayoutRebuilder.ForceRebuildLayoutImmediate(content);
625             m_ViewBounds = new Bounds(viewRect.rect.center, viewRect.rect.size);
626             m_ContentBounds = GetBounds();
627         }
628
629         // If it doesn't fit horizontally, enable horizontal scrollbar and shrink view vertically to make room for it.
630         if (m_HSliderExpand && hScrollingNeeded)
631         {
632             viewRect.sizeDelta = new Vector2(viewRect.sizeDelta.x, -(m_HSliderHeight + m_HorizontalScrollbarSpacing));
633             m_ViewBounds = new Bounds(viewRect.rect.center, viewRect.rect.size);
634             m_ContentBounds = GetBounds();
635         }
636
637         // If the vertical slider didn't kick in the first time, and the horizontal one did,
638         // we need to check again if the vertical slider now needs to kick in.
639         // If it doesn't fit vertically, enable vertical scrollbar and shrink view horizontally to make room for it.
640         if (m_VSliderExpand && vScrollingNeeded && viewRect.sizeDelta.x == 0 && viewRect.sizeDelta.y < 0)
641         {
642             viewRect.sizeDelta = new Vector2(-(m_VSliderWidth + m_VerticalScrollbarSpacing), viewRect.sizeDelta.y);
643         }
644     }
645
646     public virtual void SetLayoutVertical()
647     {
648         UpdateScrollbarLayout();
649         m_ViewBounds = new Bounds(viewRect.rect.center, viewRect.rect.size);
650         m_ContentBounds = GetBounds();
651     }
652
653     void UpdateScrollbarVisibility()
654     {
655         if (m_VerticalScrollbar && m_VerticalScrollbarVisibility != ScrollbarVisibility.Permanent && m_VerticalScrollbar.gameObject.activeSelf != vScrollingNeeded)
656             m_VerticalScrollbar.gameObject.SetActive(vScrollingNeeded);
657
658         if (m_HorizontalScrollbar && m_HorizontalScrollbarVisibility != ScrollbarVisibility.Permanent && m_HorizontalScrollbar.gameObject.activeSelf != hScrollingNeeded)
659             m_HorizontalScrollbar.gameObject.SetActive(hScrollingNeeded);
660     }
661
662     void UpdateScrollbarLayout()
663     {
664         if (m_VSliderExpand && m_HorizontalScrollbar)
665         {
666             m_Tracker.Add(this, m_HorizontalScrollbarRect,
667                 DrivenTransformProperties.AnchorMinX |
668                 DrivenTransformProperties.AnchorMaxX |
669                 DrivenTransformProperties.SizeDeltaX |
670                 DrivenTransformProperties.AnchoredPositionX);
671             m_HorizontalScrollbarRect.anchorMin = new Vector2(0, m_HorizontalScrollbarRect.anchorMin.y);
672             m_HorizontalScrollbarRect.anchorMax = new Vector2(1, m_HorizontalScrollbarRect.anchorMax.y);
673             m_HorizontalScrollbarRect.anchoredPosition = new Vector2(0, m_HorizontalScrollbarRect.anchoredPosition.y);
674             if (vScrollingNeeded)
675                 m_HorizontalScrollbarRect.sizeDelta = new Vector2(-(m_VSliderWidth + m_VerticalScrollbarSpacing), m_HorizontalScrollbarRect.sizeDelta.y);
676             else
677                 m_HorizontalScrollbarRect.sizeDelta = new Vector2(0, m_HorizontalScrollbarRect.sizeDelta.y);
678         }
```

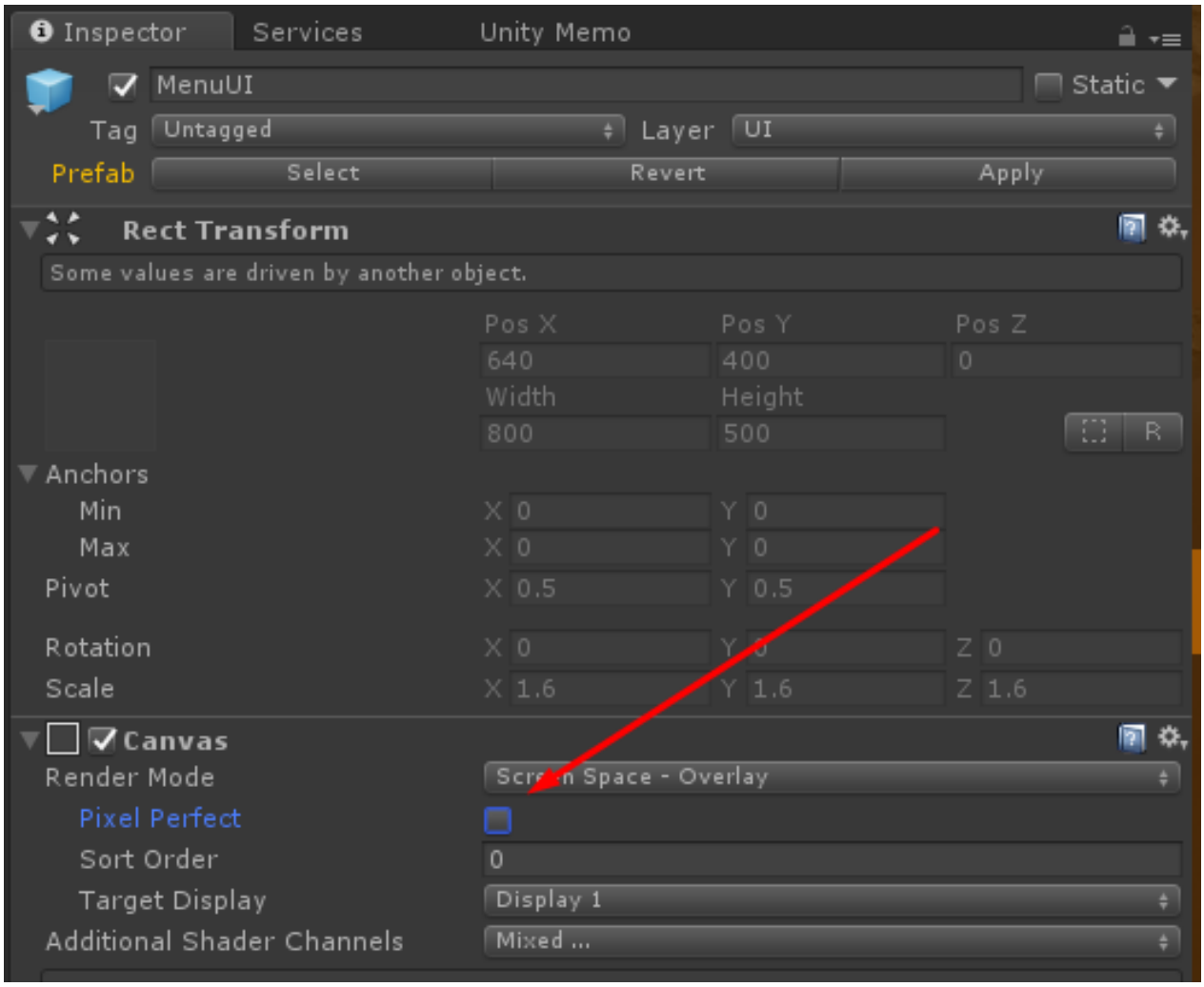
```
679
680             if (m_HSliderExpand && m_VerticalScrollbar)
681             {
682                 m_Tracker.Add(this, m_VerticalScrollbarRect,
683                     DrivenTransformProperties.AnchorMinY |
684                     DrivenTransformProperties.AnchorMaxY |
685                     DrivenTransformProperties.SizeDeltaY |
686                     DrivenTransformProperties.AnchoredPositionY);
687                 m_VerticalScrollbarRect.anchorMin = new Vector2(m_VerticalScrollbarRect.anchorMin.x, 0);
688                 m_VerticalScrollbarRect.anchorMax = new Vector2(m_VerticalScrollbarRect.anchorMax.x, 1);
689                 m_VerticalScrollbarRect.anchoredPosition = new Vector2(m_VerticalScrollbarRect.anchoredPosition.x, 0);
690                 if (hScrollingNeeded)
691                     m_VerticalScrollbarRect.sizeDelta = new Vector2(m_VerticalScrollbarRect.sizeDelta.x, -(m_HSliderHeight + m_HorizontalScrollbarSpacing));
692                 else
693                     m_VerticalScrollbarRect.sizeDelta = new Vector2(m_VerticalScrollbarRect.sizeDelta.x, 0);
694             }
695         }
696
697         private void UpdateBounds()
698         {
699             m_ViewBounds = new Bounds(viewRect.rect.center, viewRect.rect.size);
700             m_ContentBounds = GetBounds();
701
702             if (m_Content == null)
703                 return;
704
705             // Make sure content bounds are at least as large as view by adding padding if not.
706             // One might think at first that if the content is smaller than the view, scrolling should be allowed.
707             // However, that's not how scroll views normally work.
708             // Scrolling is *only* possible when content is *larger* than view.
709             // We use the pivot of the content rect to decide in which directions the content bounds should be expanded.
710             // E.g. if pivot is at top, bounds are expanded downwards.
711             // This also works nicely when ContentSizeFitter is used on the content.
712             Vector3 contentSize = m_ContentBounds.size;
713             Vector3 contentPos = m_ContentBounds.center;
714             Vector3 excess = m_ViewBounds.size - contentSize;
715             if (excess.x > 0)
716             {
717                 contentPos.x -= excess.x * (m_Content.pivot.x - 0.5f);
718                 contentSize.x = m_ViewBounds.size.x;
719             }
720             if (excess.y > 0)
721             {
722                 contentPos.y -= excess.y * (m_Content.pivot.y - 0.5f);
723                 contentSize.y = m_ViewBounds.size.y;
724             }
725
726             m_ContentBounds.size = contentSize;
727             m_ContentBounds.center = contentPos;
728         }
729
730         private readonly Vector3[] m_Corners = new Vector3[4];
731         private Bounds GetBounds()
732         {
733             if (m_Content == null)
734                 return new Bounds();
735
736             var vMin = new Vector3(float.MaxValue, float.MaxValue, float.MaxValue);
737             var vMax = new Vector3(float.MinValue, float.MinValue, float.MinValue);
738
739             var toLocal = viewRect.worldToLocalMatrix;
740             m_Content.GetWorldCorners(m_Corners);
741             for (int j = 0; j < 4; j++)
742             {
743                 Vector3 v = toLocal.MultiplyPoint3x4(m_Corners[j]);
744                 vMin = Vector3.Min(v, vMin);
745                 vMax = Vector3.Max(v, vMax);
746             }
747
```

```
            var bounds = new Bounds(vMin, Vector3.zero);
            bounds.Encapsulate(vMax);
            return bounds;
        }

        private Vector2 CalculateOffset(Vector2 delta)
        {
            Vector2 offset = Vector2.zero;
            if (m_MovementType == MovementType.Unrestricted)
                return offset;

            Vector2 min = m_ContentBounds.min;
            Vector2 max = m_ContentBounds.max;

            if (m_Horizontal)
            {
                min.x += delta.x;
                max.x += delta.x;
                if (min.x > m_ViewBounds.min.x)
                    offset.x = m_ViewBounds.min.x - min.x;
                else if (max.x < m_ViewBounds.max.x)
                    offset.x = m_ViewBounds.max.x - max.x;
            }

            if (m_Vertical)
            {
                min.y += delta.y;
                max.y += delta.y;
                if (max.y < m_ViewBounds.max.y)
                    offset.y = m_ViewBounds.max.y - max.y;
                else if (min.y > m_ViewBounds.min.y)
                    offset.y = m_ViewBounds.min.y - min.y;
            }

            return offset;
        }

        protected void SetDirty()
        {
            if (!IsActive())
                return;

            LayoutRebuilder.MarkLayoutForRebuild(rectTransform);
        }

        protected void SetDirtyCaching()
        {
            if (!IsActive())
                return;

            CanvasUpdateRegistry.RegisterCanvasElementForLayoutRebuild(this);
            LayoutRebuilder.MarkLayoutForRebuild(rectTransform);
        }

#if UNITY_EDITOR
        protected override void OnValidate()
        {
            SetDirtyCaching();
        }

#endif
    }
}<span id="mce_marker" data-mce-type="bookmark" data-mce-fragment="1"></span>
```

Create dunque un nuovo script chiamato ScrollRect2.cs e incollateci questo codice.
Sul gameObject dove avete lo ScrollRect ,toglietelo e sostituitelo con questo.

Opzione 2

Un altro metodo per migliorare le performance delle UI sta nel deselezionare la spunta PixelPerfect sul canvas.

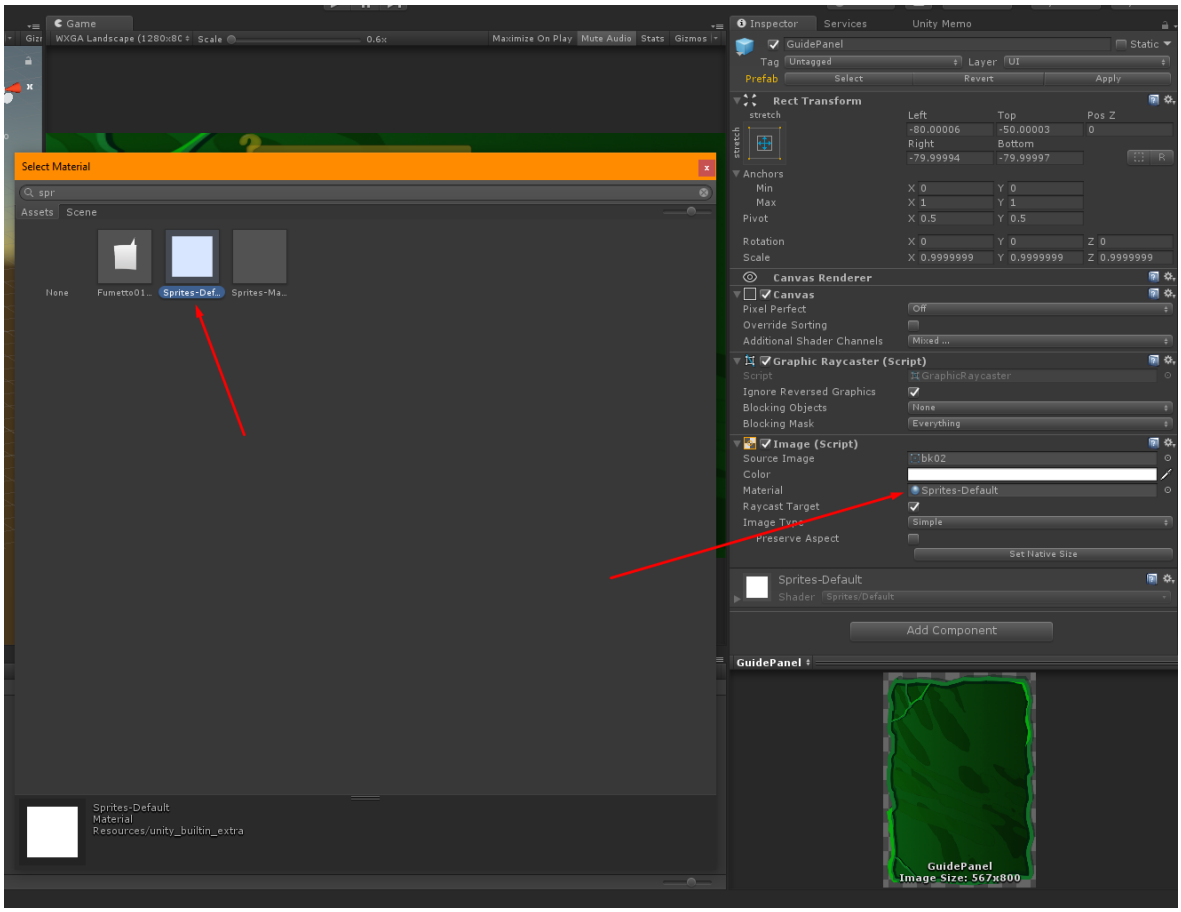


PixelPerfect forza tutti gli elementi nel canvas ad allinearsi con i pixel. Si applica solo con *renderMode* su Screen Space.

L'attivazione di pixelPerfect può rendere gli elementi più nitidi e prevenire la sfocatura. Tuttavia, se molti elementi sono ridimensionati, ruotati o animati, quest'opzione rallenta pesantemente l'aggiornamento della canvas. Potrebbe essere vantaggioso disabilitare pixelPerfect e testare il guadagno in performance.

Opzione 3

Un altro “*stratagemma*” per migliorare le prestazioni delle **UI** sta nel cambiare il materiale agli elementi dell’**UI**.



Invece di lasciarlo “vuoto”, scegliete lo **Sprites-Default**. Se non riscontrerete problemi di visualizzazione con le maschere, questo farà aumentare il *refresh* dell’elemento **UI**.

Conclusioni Finali

Utilizzando tutte e tre questi sistemi, le vostre **UI** saranno (o sembreranno) più leggere da “muovere” sia per lo *scrolling*che per il drag&Drop e potranno risultare più piacevoli e meno pesanti alla vista dell’utente.

Camera che segue il Player

Questo script è relativamente semplice ma molto utile.
In molti tipi di giochi dovremmo avere una telecamera che segue il giocatore in modo “smooth”, cioè leggermente ritardato e fluido.
Che sia da dietro e che ruoti insieme al personaggio (come in un classico gioco in terza persona), oppure da una posizione più in alto e senza che la telecamera ruoti insieme al personaggio, questo script può fare al caso vostro.

Con questo script sarà possibile utilizzare diversi tipi di visuale, a seconda delle impostazioni che sceglierete.
Inoltre potrete scegliere il tipo di aggiornamento (*UpdateSystem*) per adattarlo al meglio all’aggiornamento del gioco.
La scelta di un *UpdateSystem* adeguato si rende necessario perché, per evitare leggeri ritardi ed effetti “flickering” della telecamera, l’aggiornamento dovrebbe essere sempre uguale a quello che usa il movimento del player.
Dunque, se per esempio il vostro player si muove tramite un rigidBody userà l’aggiornamento per la fisica (*FixedUpdate*) ed altrettanto dovrà fare la telecamera.

VIDEO 01_CAMERA CHE SEGUE IL PLAYER

C# 73 lines

```
using UnityEngine;

public class CameraFollowCharacter : MonoBehaviour
{

    [SerializeField]
    private Transform target;

    [SerializeField]
    public Vector3 offsetPosition;
    public float cameraTargetHeight;
    public float cameraTargetHorizontal;

    [SerializeField]
    private Space SpaceType = Space.Self;
    Vector3 LoockAtPoint;

    public enum UpdateType { Update, FixedUpdate, LateUpdate };
    public UpdateType updateSystem=UpdateType.LateUpdate;

    [Range(1,5)]
    public float smoothMovement=1;


    //Nel caso si debba aggiornare in Update
    private void Update()
    {
        if(updateSystem.Equals(UpdateType.Update))
        Refresh(Time.deltaTime);
    }

    //Nel caso si debba aggiornare in FixedUpdate
    private void FixedUpdate()
    {
        if (updateSystem.Equals(UpdateType.FixedUpdate))
            Refresh(Time.fixedDeltaTime);
    }
    //Nel caso si debba aggiornare in LateUpdate
    private void LateUpdate()
    {
        if (updateSystem.Equals(UpdateType.LateUpdate))
            Refresh(Time.deltaTime);
    }

    //Il metodo che aggiorna le posizioni della telecamera
    public void Refresh(float updateDelta)
    {
        if (target == null)
        {
            Debug.LogWarning("Ops, No Target !", this);

            return;
        }



        //Se ruota insieme al target
        if (SpaceType == Space.Self)
        {
            transform.position =Vector3.Lerp(transform.position, target.TransformPoint(offsetPosition) , smoothMovement * updateDelta);
        }
        else
        {
            transform.position = Vector3.Lerp(transform.position, target.position + offsetPosition, smoothMovement * updateDelta);
        }

         LoockAtPoint= Vector3.Lerp(LoockAtPoint,(target.position + (target.up * cameraTargetHeight) + (target.right * cameraTargetHorizontal)), (smoothMovement*updateDelta));

         transform.LookAt(LoockAtPoint);


    }
}
```

Sta a voi fare tutte le prove che volete, sistemando gli offsets e lo smooth ad hoc, per arrivare ad avere l’effetto che serve per il vostro gioco.

Telecamera FPS con Character

Questo prefab si basa su quello già presente nello standard Asset di Unity per il movimento del player in stile FPS (prima persona).
L'unica differenza sta nel fatto che oltre alla classico Capsule Collider vuoto, potremmo vedere anche il personaggio animato, così che abbassando lo sguardo potremo ammirare le sue gambe muoversi oltre a poter vedere la sua ombra. Diciamo che questo prefab è un ibrido tra i due prefabs presenti nello standard Asset di Unity, ThirdPersonController e RigidBodyFPSController.

VIDEO_02 TELECAMERA FPS CON CHARACTER

Essendo il prefab composto da diversi elementi e diversi scripts (una leggera modifica è stata fatta anche allo script *HeadBob*) e da diversi gameObject con una gerarchia predefinita, non ho messo lo script ma direttamente l'intero package scaricabile da questo link:

DOWNLOAD – FPSwithCharacter.unitypackage -
http://unity3dtutorials.it/Download/FPSwithCharacter.unitypackage

Per un corretto funzionamento dovrete aver precedentemente importato *lo Standard Asset Character.*

Trovare GameObjects non attivi

Questa è un'esigenza che capita almeno una volta in ogni progetto.
Come sappiamo, quando abbiamo la necessità di trovare uno specifico **gameObject** nella scena possiamo andare a cercarlo tramite diversi metodi:

Facendo una ricerca tramite il nome del **gameObject** :
mygameObject=GameObject.Find("nomeOggetto");

oppure facendo una ricerca tramite un tag:
mygameObject=GameObject.FindObjectsWithTag("tagOggetto");

oppure ricercando uno specifico script che sappiamo è attaccato su di esso:
mygameObject= FindObjectOfType<myScript>().gameObject;

Tutti queste tecniche hanno però un handicap, ovvero non riescono a trovare un oggetto se non è attivo nella gerarchia.
Vi accorgerete presto che questo è un limite molto fastidioso che vi potrebbe generare diversi grattacapi per diverse ragioni.
Per ovviare a questo problema ci viene in aiuto una funzione che ci permette di fare ricerche anche su oggetti non attivi:

GetComponentInChildren<myScript>();

Questa funzione va alla ricerca di uno script nei childrens di un gameObject e permette di impostare anche la ricerca sugli oggetti inattivi, specificandolo come parametro:

GetComponentInChildren<myScript>(true);

E' ovvio che il nostro gameObject dovrà trovarsi necessariamente sotto ad un altro gameObject e che la ricerca deve essere effettuata specificatamente su di esso:

myParentGameObject.GetComponentInChildren<myScript>(true);

In questo modo potremmo andare alla ricerca di uno specifico gameObject, anche se inattivo, a patto che si trovi sotto myParentGameObject e che contenga lo script myScript.

Se volessimo ricercare un determinato oggetto senza sapere sotto a quale gameObject si trovi dovremmo fare una ricerca su tutti gli oggetti:

C# 8 lines

```
1 foreach (GameObject allObjects in FindObjectsOfType<GameObject>()){
2 if(allObjects.GetComponentInChildren<myScript>(true)){
3
4 mygameObject=allObjects;
5 return;
6
7     }
8 }
```

In questo modo avremo ricercato su tutti gli oggetti **attivi** con **FindObjectsOfType() e** se uno di essi contenesse un gameObject (anche se inattivo) che possiede lo script myScript e lo avremo *"salvato"* su **myObject**.
Appena si trova l'oggetto in questione, con return; usciamo dalla ricerca.

OnTriggerExit su oggetti distrutti

Può capitare di dover sapere quando un oggetto esce da una determinata area perché distrutto.
Come sappiamo per rilevare quando un oggetto esce da un'area **trigger** possiamo usare la funzione **OnTriggerExit**.

Ma se proviamo a cancellare (distruggere) un oggetto che era in contatto con un determinato **trigger** noteremo che la funzione **OnTriggerExit** non verrà richiamata, perché appunto, l'oggetto è stato distrutto e la rilevazione d'uscita non può più avvenire correttamente anche se in realtà l'oggetto, essendo stato distrutto, non è più a contatto con il **trigger** ed è dunque in effetti, uscito dal suo contatto.

Per ovviare a questo problema dovremmo usare uno stratagemma.
Nello script del trigger che dovrà rilevare la distruzione di un oggetto che era precedentemente in contatto con esso, ci mettiamo questo script:

C# 42 lines

```
using UnityEngine;


    public class MyObject : MonoBehaviour
    {

        bool triggered = false;
        Collider col;

        //L'entrata in contatto funziona come sempre
        //andiamo però ad impostare i due parametri necessari
        private void OnTriggerEnter(Collider other)
        {
            triggered = true;
            col = other;

            //Mettere qui il codice per OnTriggerEnter

        }

        //La nostra funzione, sostitutiva del OnTriggerExit
        private void OnTriggerDestroy()
        {
            triggered = false;

            //Mettere qui il codice per OnTriggerExit

        }



        void Update()
        {

            if (triggered && !col)
            {
                OnTriggerDestroy();
            }


        }
    }
```

Cosa succede in questo script è abbastanza intuibile.
Nel metodo **Update** (dunque ad ogni frame) andremo a fare un doppio controllo, **(triggered && !col)**, ovvero se esiste un collider impostato all'entrata in contatto (coll) e che esso sia ancora esistente. Quando esso non lo sarà più, verrà eseguita la funzione **OnTriggerDestroy();**
Dunque nel momento in cui coll sarà stato distrutto esso risulterà null e verrà eseguita la funzione **OnTriggerDestroy();**come se fosse un **OnTriggerExit();**
Tale funzione verrà eseguita una singola volta nel momento della distruzione dell'oggetto perché avremmo impostato su false la variabile ***triggered*** che tornerà su true solo ad un nuovo contatto con questo oggetto.

Sparare con un’arma

Abbiamo già visto come instanziare un oggetto (prefab) alla pressione di un tasto sull'articolo riguardante i prefabs.

C# 31 lines

```
using UnityEngine;

public class Player : MonoBehaviour {

    public GameObject PrefabOriginale;




    void Spara () {
        //Istanzia l'oggetto proiettile, creando una copia del PrefabOriginale impostando la posizione e
        GameObject proiettile = GameObject.Instantiate(PrefabOriginale,transform.position,Quaternion.iden
    }


    void Update()
    {
        // Alla pressione di Fire1, esegui il metodo Spara()
        if (Input.GetButtonDown("Fire1"))
        {
            Spara();
        }




    }



}
```

Questa funzione ci può tornare utile per creare un sistema di “fuoco” di un’arma. Il codice visto in precedenza (sull'articolo riguardante i prefabs) è però troppo generico e rudimentale perché possa essere considerato realmente utilizzabile se non per un tipo di sparo singolo.

Ora creeremo un sistema che istanzierà un proiettile alla pressione del tasto “Fire1" e faremo in modo che tenendo premuto il tasto la nostra arma sparerà di continuo, colpo dopo colpo, con una cedenza da noi stabilita.
Ma non solo... faremo in modo anche che il nostro caricatore abbia un numero limitato di colpi e ogni raffica effettuata tenendo premuto "Fire1" sia così contenuta entro un certo numero di proiettili dopo i quali bisognerà rilasciare il pulsante per rendere possibile la raffica successiva. Simuleremo così il riscaldamento dell'arma o se preferite un effetto "UZI" a raffica controllata.
Potremmo poi cambiare i valori a nostro piacimento per simulare qualsiasi tipo di arma, ad un colpo singolo, a raffica controllata o a raffica continua.

Iniziamo creando lo script che gestisce le munizioni dell'arma che chiamerò ArmaControl.

Sarebbe buona usanza usare sempre nomi di variabili in inglese, ma noi, visto che siamo qui per imparare, solo per scopo didattico e per una comprensione più immediata, useremo anche qualche parola in italiano.

Questo script è da posizionarsi su una singola arma, così da poter creare anche diverse tipologie di armi, ognuna con un fire-rate differente.

C# 121 lines

```
using UnityEngine;


public class ArmaControl : MonoBehaviour
{
    public float fireRate = 0.5f; //Tempo tra un colpo e l'altro
    public bool recargeOnMouseUp=false; //Se true, quando si rilascia il tasto di fuoco ricarica la raffica senza attesa
    public int maxAmmoRaffica = 5; //Colpi di una singola raffica
    public float pauseTime; //Tempo di attesa tra una raffica e l'altra
    public int ammoQuantity = 20; //Quantità di munizioni totali


    int ammoRaffica; //Munizioni attuali
    float nextFire; //Prossimo colpo pronto
    float pauseTimer; //Il timer della pausa
    float cooldown; //Cantatore di colpi
    bool pause =false; //In pausa




     void Start()
    {
        ammoRaffica = maxAmmoRaffica; //All'inizio il caricatore è pieno
    }



    void Update()
    {
        if (!pause && ammoQuantity>0) //Se è tra una raffica e l'altra non sparare
        {
            if (fireRate == 0 && Input.GetButtonDown("Fire1"))
            {   //Il primo colpo lo effettua senza dover attendere nulla
                Shoot();
            }
            else
            {
                if (Input.GetButton("Fire1") && Time.time > nextFire && fireRate > 0) //Pronto per sparare
                {
                    //Colpi successivi, tenendo premuto il pulsante
                    if (ammoRaffica > 0)
                    {   //Se ci sono munizioni
                        nextFire = Time.time + fireRate;
                        Shoot(); //Esegue la funzione di "sparo"

                    }
                    if (ammoRaffica == 0)
                    {   //Se non ci sono più munizioni
                        if (cooldown > Time.time)
                        {   //Se non è passato il tempo giusto
                            cooldown = Time.time + fireRate;
                        }
                    }
                }
            }

            if (Time.time > cooldown && ammoRaffica == 0)
            {  //Se il tempo di recupero (cooldown) è finito e le munizioni sono terminate
                pause = true;

            }
        }


        //Se si è scelto di far ricaricare al rilascio del tasto
```

```
68          if(recargeOnMouseUp && Input.GetButtonUp("Fire1")){
69              pauseTimer = 0;
70              pause = false;
71
72
73              if (ammoQuantity >= maxAmmoRaffica)//Se ci sono abbastanza i proiettili
74                  ammoRaffica = maxAmmoRaffica; //Ricarico per la prossima raffica
75              else
76                  ammoRaffica = ammoQuantity; //Se non sono i proiettili metti quelli disponibili
77          }
78
79         //Se in pausa, fai il conteggio del tempo tra una raffica e l'altra
80          if (pause)
81          ShootPause();
82
83      } //Fine di Update()
84
85
86      //Metodo che fa il conteggio del tempo tra una scarica e l'altra
87      void ShootPause()
88      {
89          pauseTimer += Time.deltaTime;
90          if (pauseTimer >= pauseTime)
91          {
92              pauseTimer = 0;
93              pause = false;
94
95
96              if(ammoQuantity>= maxAmmoRaffica)//Se ci sono abbastanza i proiettili
97              ammoRaffica = maxAmmoRaffica; //Ricarico per la prossima raffica
98              else
99              ammoRaffica = ammoQuantity; //Se non sono i proiettili metti quelli disponibili
100         }
101     }
102
103
104
105     void Shoot() {
106
107         ammoRaffica--; //Rimuove un colpo della raffica ad ogni sparo
108         ammoQuantity--; //Rimuove un colpo dalla quantità in possesso
109
110         print("Shooooot");//Spara!
111     }
112
113
114
115
116 } //Chiusura classe
117
118
119
120
121
```

A questo punto abbiamo uno script che ci permette di settare diversi parametri.
Se per esempio non vogliamo il "sistema a scariche", basterà impostare su 0 il pauseTime.
Abbiamo inserito anche un parametro (ammoQuantity) per gestire il numero di munizioni a disposizione del giocatore, terminate le quali non sarà più possibile sparare.

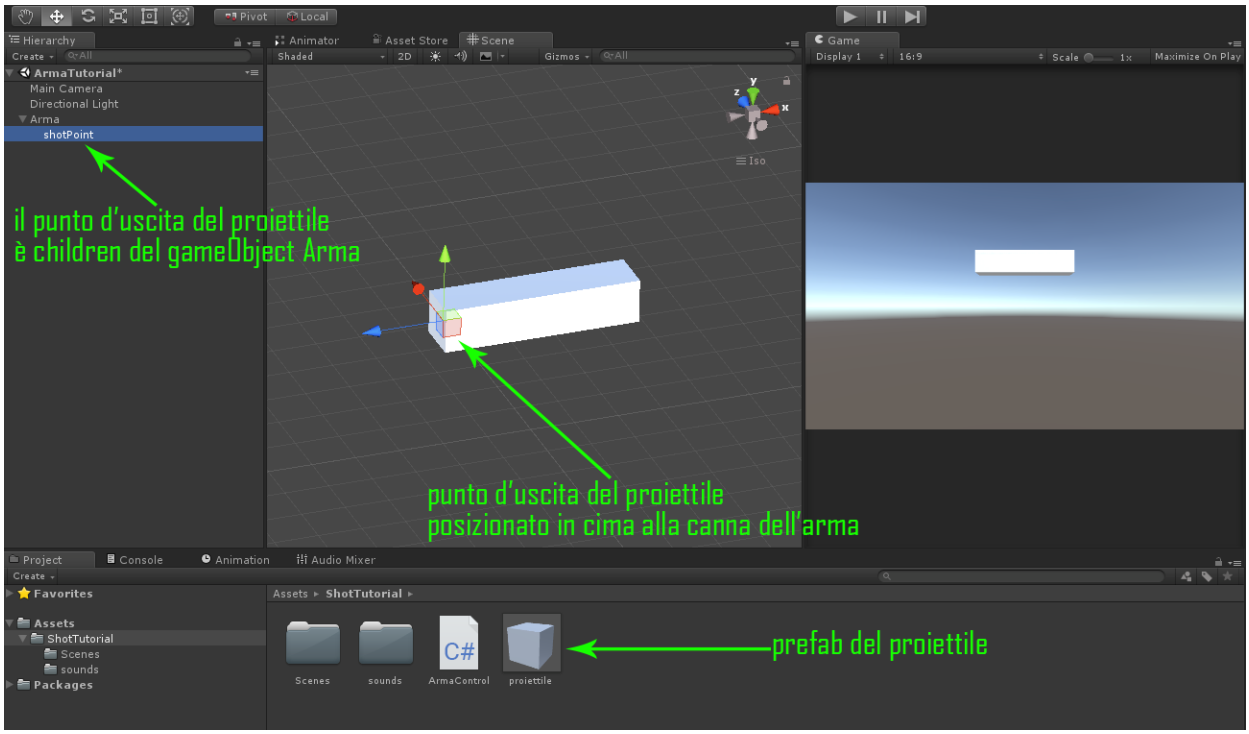
Fino a qui abbiamo creato la gestione dei colpi con cui poter creare diversi tipi di "reazione" alla pressione del nostro "grilletto".
Il metodo Spara() fin'ora non fa altro che scrivere in console la stringa Shooooot che potremmo contare per verificare che tutto funzioni correttamente.

Ora lavoriamo sul metodo Spara() per creare il nostro proiettile ed anche un effetto "fiammata" sul punto d'uscita della canna della nostra arma.
Per fare le cose per bene andremo anche ad inserire un suono che si esegue al momento dello sparo.

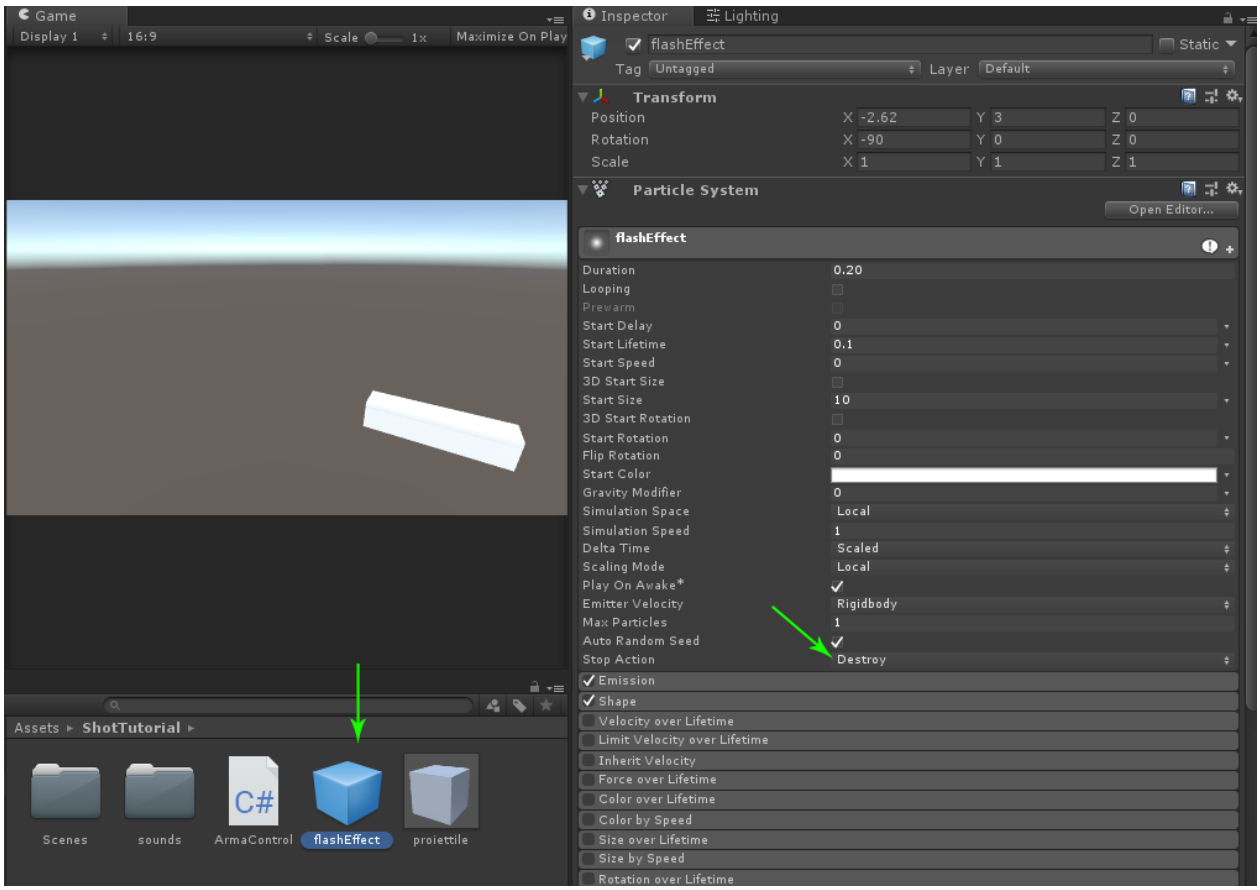
Dovremmo prima di tutto inserire la variabile del prefab che identifica il proiettile, poi dovremo identificare il punto di creazione del proiettile che sarà lo stesso del punto di creazione dell' effetto fiamma e del suono.

Per fare tutto questo si potrebbero percorrere tante strade differenti, noi useremo quella che secondo me è quella più utile a scopo didattico.

1. Creiamo un arma (Arma) , nel nostro esempio un semplice parallelepipedo.
2. Creaimo un oggetto che funga da proiettile (Proiettile) , nel nostro caso un semplice cubo. Il gameObject Proiettile non deve essere presente nella scena, ma solo in una cartella del progetto, servirà come prefab da istanziare nel momento dello sparo.
3. Creiamo un gameObject vuoto (ShotPoint) e lo mettiamo come children al gameObject dell'arma, posizionandolo là dove vorremmo far "uscire" il proiettile, in cima alla "canna" dell'arma. ShotPoint sarà anche il punto in cui verrà visualizzato l'effetto "lampo/fiammata" dell'arma.



Se vogliamo aggiungere anche un effetto "lampo/fiammata" allo sparo, creiamo un Particle System e usiamolo come prefab da generare insieme al proiettile.
Ricordiamoci di impostare su la sua Stop Action su "destroy" in modo che l'effetto sia automaticamente distrutto una volta completato (le impoostazioni del Particle System le potete vedere scaricando il package del tutorial).



NOTA SULLE PRESTAZIONI:
Come desctitto nei commenti, avremmo anche potuto usare una tecnica più performante per generare l'effetto "fiammata". Nel sistema usato ora c'è bisogno di un'ulteriore istanziamento, oltre che del proiettile, anche del Particle System. Notoriamente, gli istanziamenti sono la morte delle prestazioni, dunque dove possibile sarebbe meglio evitarli.
Si potrebbe per esempio creare un unico Particle System posto sulla punta della canna e farlo "emettere" nel momento dello sparo. Oppure dotare il proiettile di un Particle System che verrà rilasciato nel momento dello sparo... Le tecniche sono tante.

Non solo l'istanzaimento del Particle System potrebbe essere evitato... mapersino quello dei proiettili! Vedremo in seguito un sistema chiamato genericamente "pool system", ovvero una tecnica che riutilizza sempre gli stessi proiettili, nascondendoli e riattivandoli (riposizionandoli nel punto dello sparo) nel momento più opportuno, senza doverli istanziare e distruggere tutte le volte, cosa molto dispendiosa in termini di performance.

C# 162 lines

```
using UnityEngine;


public class ArmaControl : MonoBehaviour
{

    //Variabili pubbliche
    public float fireRate = 0.5f; //Tempo tra un colpo e l'altro
    public bool recargeOnMouseUp=false; //Se true, quando si rilascia il tasto di fuoco ricarica la raffica senza attesa
    public int maxAmmoRaffica = 5; //Colpi di una singola raffica
    public float pauseTime; //Tempo di attesa tra una raffica e l'altra
    public int ammoQuantity = 20; //Quantità di munizioni totali
    public GameObject proiettilePrefab;//Il prefab del proiettile
    public GameObject flashEffectPrefab;//Il prefab dell'effetto flash/fiammata
    public Transform shotPoint; //Il trasform che identifica il punto di creazione dei proiettili
    public AudioClip shotSound; //Il suono dello sparo



    //Variabili private
    int ammoRaffica; //Munizioni attuali
    float nextFire; //Prossimo colpo pronto
    float pauseTimer; //Il timer della pausa
    float cooldown; //Cantatore di colpi
    bool pause =false; //In pausa
    AudioSource audioSource; //L'audioSource che eseguirà il suono



    void Start()
    {
          AudioSourceCreation();//Crea l'AudioSource (se non presente)
          ammoRaffica = maxAmmoRaffica; //All'inizio il caricatore è pieno
    }

    //Creiamo l'AudioSource a runtime, senza doverlo inserire manualmente
    void AudioSourceCreation() {
        //Se non è stato creato manualmente
        if (!GetComponent<AudioSource>())
        {
            audioSource = gameObject.AddComponent<AudioSource>(); //Crea l'AdudioSource
        }
        else
        {
            audioSource = GetComponent<AudioSource>(); //Se già esisteva usa quello esistente
        }

        //Notare che si può creare l'audioSource anche manualmente, in modo da impostare i parametri
        //i parametri (come il volume, il pitch ed altri) direttamente da Inspector
        //In alternativa potete anche impostarli qui, nel momento della creazione, scegliendo i parametri
        //a runtime (audioSource.volume=1,  audioSource.pitch=1, ecc.)

    }



    void Update()
    {
        if (!pause && ammoQuantity>0) //Se è tra una raffica e l'altra non sparare
        {
            if (fireRate == 0 && Input.GetButtonDown("Fire1"))
            {   //Il primo colpo lo effettua senza dover attendere nulla
                Shoot();
            }
            else
            {
                if (Input.GetButton("Fire1") && Time.time > nextFire && fireRate > 0) //Pronto per sparare
```

```
            {
                //Colpi successivi, tenendo premuto il pulsante
                if (ammoRaffica > 0)
                {   //Se ci sono munizioni
                    nextFire = Time.time + fireRate;
                    Shoot(); //Esegue la funzione di "sparo"

                }
                if (ammoRaffica == 0)
                {   //Se non ci sono più munizioni
                    if (cooldown > Time.time)
                    {   //Se non è passato il tempo giusto
                        cooldown = Time.time + fireRate;
                    }
                }
            }
        }

        if (Time.time > cooldown && ammoRaffica == 0)
        {  //Se il tempo di recupero (cooldown) è finito e le munizioni sono terminate
            pause = true;

        }
    }


    //Se si è scelto di far ricaricare al rilascio del tasto
    if(recargeOnMouseUp && Input.GetButtonUp("Fire1")){
        pauseTimer = 0;
        pause = false;


        if (ammoQuantity >= maxAmmoRaffica)//Se ci sono abbastanza i proiettili
            ammoRaffica = maxAmmoRaffica; //Ricarico per la prossima raffica
        else
            ammoRaffica = ammoQuantity; //Se non sono i proiettili metti quelli disponibili
    }

    //Se in pausa, fai il conteggio del tempo tra una raffica e l'altra
    if (pause)
    ShootPause();

} //Fine di Update()


//Metodo che fa il conteggio del tempo tra una raffica e l'altra
void ShootPause()
{
    pauseTimer += Time.deltaTime;
    if (pauseTimer >= pauseTime)
    {
        pauseTimer = 0;
        pause = false;


        if(ammoQuantity>= maxAmmoRaffica)//Se ci sono abbastanza i proiettili
        ammoRaffica = maxAmmoRaffica; //Ricarico per la prossima raffica
        else
        ammoRaffica = ammoQuantity; //Se non sono i proiettili metti quelli disponibili
    }
}



void Shoot() {

    ammoRaffica--; //Rimuove un colpo della raffica ad ogni sparo
    ammoQuantity--; //Rimuove un colpo dalla quantità in possesso
```

```
136
137         //Istanzia l'oggetto proiettile, creando una copia del proiettilePrefab impostando la posizione e la rotazione
138         GameObject proiettile = Instantiate(proiettilePrefab, shotPoint.position, shotPoint.rotation);
139         //Notare che impostiamo la posizione di origine e la rotazione inziale uguali a quelle del trasform shotPoint
140
141         //Secondo istanziamento di un gameObject, necessario per l'effetto "fiammata"
142         GameObject flashEffect = Instantiate(flashEffectPrefab, shotPoint.position, shotPoint.rotation);
143         //Notare che si potrebbe usare anche un'altra tecnica, meno dispendiosa in termini di prestazioni
144         //che vedremo in seguito
145
146
147         if (shotSound)//Assicuriamoci che c'è un suono scelto per il rumore dello sparo
148         audioSource.PlayOneShot(shotSound); //Esegui il suono
149
150
151         print("Shooooot");//Spara!
152     }
153
154
155
156
157 } //Chiusura classe
158
159
160
161
162
```

A questo punto sta a voi dare dare velocità al proiettile, altrimenti tutti i proiettili saranno generati nel punto "shotPoint" senza muoversi, accumulandosi in cima alla canno dell'arma.
Per fare questo dovrete dotare il prefab proiettile di uno script apposito, che lo faccia muovere appena generato e che lo faccia autodistruggere dopo un tot di tempo, altre che ovviamente ad un sistema di collisione che rilevi quando il proiettile va a sengno con un nemico o altro.
Potremmo dare forza al proiettile anche direttamente dallo script dell'arma, ma è buona norma dividere le cose in modo da poter gestire i proiettili individualmente.
Questo argomento è influenzato da molti aspetti soggettivi che dipendono da come si vuol far interagire i proiettili con in mondo circostante e con i nemici. Esistono diverse tecniche per muovere gli oggetti, dotandoli di rigidBody o muovendoli da Transform.

C# 17 lines

```
1  using UnityEngine;
2
3  public class Proiettile : MonoBehaviour {
4
5      public Rigidbody rig;
6      public float speed=100f;
7
8      void Start () {
9
10     }
11
12
13     void Update () {
14         rig.AddForce(transform.forward* speed);
15     }
16 }
17
```

Nel package del tutorial troverete un sistema molto semplice per muovere i proiettili, a voi modificarlo ed espanderlo a vostro piacimento.

Tutorial Package: Arma Tutorial. - http://unity3dtutorials.it/Download/ArmaTutorial.unitypackage

Migliorare il PlayerPrefs

PlayerPrefs è la classe che ci permette di effettuare dei salvataggi permanenti. La locazione dei salvataggi varia a seconda del dispositivo per cui si è sviluppato il gioco ma il funzionamento a livello di codice è lo stesso. Se non conoscete il suo utilizzo, vi invito a leggere la lezione dedicata al PlayerPrefs.

Siamo qui per "potenziare" il metodo di salvataggio tramite la classe PlayerPrefs, o meglio, per creare una nuova classeche si appoggi sull'originale ma che ci permetta di salvare tipi di variabile come i Vector3, i Vector2, i Quaternioni e tutti quei dati che, se usassimo il normale PlayerPref necessiterebbero il salvataggio di ogni singolo elemento che lo compongono.
Badate bene che questo script non è un sostituto del PlayerPrefs originale, esso lo completa e si basa su di esso.

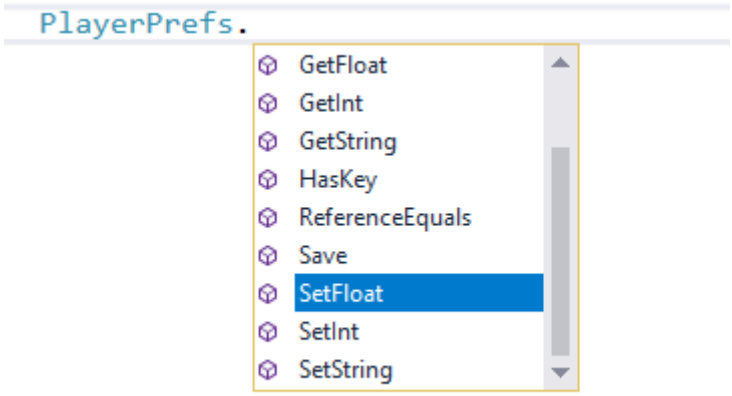
Come sappiamo, tramite PlayerPrefs potremmo salvare un dato di tipo float con la seguente istruzione:

```
C# 1 lines
1  PlayerPrefs.SetFloat("nomeNumero", mioFloat);//Salvo il numero "mioFloat" sotto il nome "nomeNumero"
```

così da poterlo caricare in qualsiasi momento con l'istruzione:

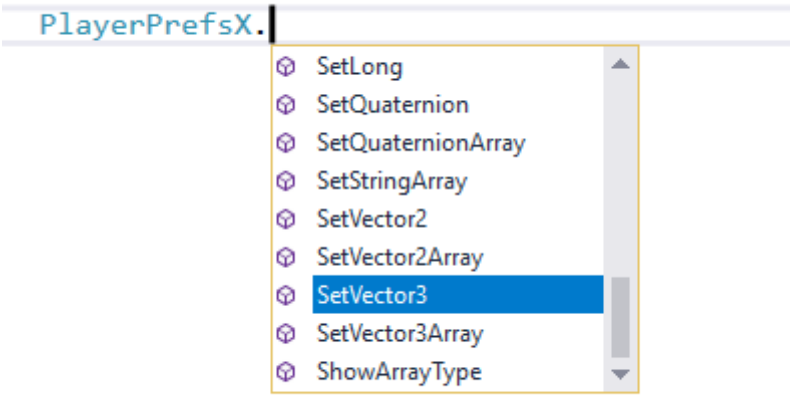
```
C# 1 lines
1  PlayerPrefs.GetFloat("nomeNumero");//Carico il numero salvato sotto il nome "nomeNumero"
```

Tutto molto semplice e funzionale… finchè si tratta di dati singoli, come un solo numero float, o int, ecc.



Come possiamo notare, il PlayerPrefs ci permette di salvare e caricare tre tipi di dati, float, int, e string. Poca roba insomma, non vengono menzionati i tipi di dati compositi, come i Vector3, che sono un tipo di dato molto usato in un gioco.
Se per esempio volessimo salvare la posizione del giocatore alla chiusura del gioco, in modo di poterlo riposizionare in quello stesso punto al riavvio, dovremmo salvare singolarmente le tre componenti X,Y,Z con

tre righe diverse e lo stesso dovremmo fare per effettuare il caricamento, andando anche a “ricomporre” il vector3 tramite i tre dati salvati.



Una bella rogna che possiamo evitare con questo splendido script, il PlayerPrefsX, dove vediamo presenti una nutrita serie di variabili composite, tutte salvabili esattamente con lo stesso metodo di prima. Ci sono perfino gli array e gli array di Vector3, i bool, i Color e molti altri tipi di variabile, tutti salvabili e ricaricabili con una singola riga! Possiamo intuire quanto diventi di importanza essenziale questa nuova classe.

Se per esempio volessimo salvare il Vector3 che rappresenta la posizione del player nel momento della chiusura del gioco (o della scena), non dovremmo far altro che usare l’istruzione *SetVector3*:

```
C# 1 lines
1  PlayerPrefsX.SetVector3( "posizionePlayer", playerTransform.position);
```

e per ricaricarla non dovremmo far altro che usare l’istruzione *GetVector3*:

```
C# 2 lines
1  playerTransform.position = PlayerPrefsX.GetVector3( "posizionePlayer");
2
```

Copiate e incollate il codice in uno script chiamato *PlayerPrefsX.cs*.

C# 649 lines 349 lines 3 lines

```
// ArrayPrefs2 v 1.4

using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;

public class PlayerPrefsX
{
    static private int endianDiff1;
    static private int endianDiff2;
    static private int idx;
    static private byte[] byteBlock;

    enum ArrayType { Float, Int32, Bool, String, Vector2, Vector3, Quaternion, Color }

    public static bool SetBool(String name, bool value)
    {
        try
        {
            PlayerPrefs.SetInt(name, value ? 1 : 0);
        }
        catch
        {
            return false;
        }
        return true;
    }

    public static bool GetBool(String name)
    {
        return PlayerPrefs.GetInt(name) == 1;
    }

    public static bool GetBool(String name, bool defaultValue)
    {
        return (1 == PlayerPrefs.GetInt(name, defaultValue ? 1 : 0));
    }

    public static long GetLong(string key, long defaultValue)
    {
        int lowBits, highBits;
        SplitLong(defaultValue, out lowBits, out highBits);
        lowBits = PlayerPrefs.GetInt(key + "_lowBits", lowBits);
        highBits = PlayerPrefs.GetInt(key + "_highBits", highBits);

        // unsigned, to prevent loss of sign bit.
        ulong ret = (uint)highBits;
        ret = (ret << 32);
        return (long)(ret | (ulong)(uint)lowBits);
    }

    public static long GetLong(string key)
    {
        int lowBits = PlayerPrefs.GetInt(key + "_lowBits");
        int highBits = PlayerPrefs.GetInt(key + "_highBits");

        // unsigned, to prevent loss of sign bit.
        ulong ret = (uint)highBits;
        ret = (ret << 32); return (long)(ret | (ulong)(uint)lowBits); } private static void SplitLong(long input, out int lowBits, out int highBits) { // unsigned everything, to prevent loss of sign bit. lowBits = (int)(uint)(ulong)input; highBits = (int)(uint)(input >> 32);
    }

    public static void SetLong(string key, long value)
    {
        int lowBits, highBits;
        SplitLong(value, out lowBits, out highBits);
```

```
67          PlayerPrefs.SetInt(key + "_lowBits", lowBits);
68          PlayerPrefs.SetInt(key + "_highBits", highBits);
69      }
70
71      public static bool SetVector2(String key, Vector2 vector)
72      {
73          return SetFloatArray(key, new float[] { vector.x, vector.y });
74      }
75
76      static Vector2 GetVector2(String key)
77      {
78          var floatArray = GetFloatArray(key);
79          if (floatArray.Length < 2)
80          {
81              return Vector2.zero;
82          }
83          return new Vector2(floatArray[0], floatArray[1]);
84      }
85
86      public static Vector2 GetVector2(String key, Vector2 defaultValue)
87      {
88          if (PlayerPrefs.HasKey(key))
89          {
90              return GetVector2(key);
91          }
92          return defaultValue;
93      }
94
95      public static bool SetVector3(String key, Vector3 vector)
96      {
97          return SetFloatArray(key, new float[] { vector.x, vector.y, vector.z });
98      }
99
100     public static Vector3 GetVector3(String key)
101     {
102         var floatArray = GetFloatArray(key);
103         if (floatArray.Length < 3)
104         {
105             return Vector3.zero;
106         }
107         return new Vector3(floatArray[0], floatArray[1], floatArray[2]);
108     }
109
110     public static Vector3 GetVector3(String key, Vector3 defaultValue)
111     {
112         if (PlayerPrefs.HasKey(key))
113         {
114             return GetVector3(key);
115         }
116         return defaultValue;
117     }
118
119     public static bool SetQuaternion(String key, Quaternion vector)
120     {
121         return SetFloatArray(key, new float[] { vector.x, vector.y, vector.z, vector.w });
122     }
123
124     public static Quaternion GetQuaternion(String key)
125     {
126         var floatArray = GetFloatArray(key);
127         if (floatArray.Length < 4)
128         {
129             return Quaternion.identity;
130         }
131         return new Quaternion(floatArray[0], floatArray[1], floatArray[2], floatArray[3]);
132     }
133
134     public static Quaternion GetQuaternion(String key, Quaternion defaultValue)
135     {
136         if (PlayerPrefs.HasKey(key))
```

```
        {
            return GetQuaternion(key);
        }
        return defaultValue;
    }

    public static bool SetColor(String key, Color color)
    {
        return SetFloatArray(key, new float[] { color.r, color.g, color.b, color.a });
    }

    public static Color GetColor(String key)
    {
        var floatArray = GetFloatArray(key);
        if (floatArray.Length < 4)
        {
            return new Color(0.0f, 0.0f, 0.0f, 0.0f);
        }
        return new Color(floatArray[0], floatArray[1], floatArray[2], floatArray[3]);
    }

    public static Color GetColor(String key, Color defaultValue)
    {
        if (PlayerPrefs.HasKey(key))
        {
            return GetColor(key);
        }
        return defaultValue;
    }

    public static bool SetBoolArray(String key, bool[] boolArray)
    {
        // Make a byte array that's a multiple of 8 in length, plus 5 bytes to store the number of entries as an int32 (+ identifier)
        // We have to store the number of entries, since the boolArray length might not be a multiple of 8, so there could be some padded zeroes
        var bytes = new byte[(boolArray.Length + 7) / 8 + 5];
        bytes[0] = System.Convert.ToByte(ArrayType.Bool);   // Identifier
        var bits = new BitArray(boolArray);
        bits.CopyTo(bytes, 5);
        Initialize();
        ConvertInt32ToBytes(boolArray.Length, bytes); // The number of entries in the boolArray goes in the first 4 bytes

        return SaveBytes(key, bytes);
    }

    public static bool[] GetBoolArray(String key)
    {
        if (PlayerPrefs.HasKey(key))
        {
            var bytes = System.Convert.FromBase64String(PlayerPrefs.GetString(key));
            if (bytes.Length < 5)
            {
                Debug.LogError("Corrupt preference file for " + key);
                return new bool[0];
            }
            if ((ArrayType)bytes[0] != ArrayType.Bool)
            {
                Debug.LogError(key + " is not a boolean array");
                return new bool[0];
            }
            Initialize();

            // Make a new bytes array that doesn't include the number of entries + identifier (first 5 bytes) and turn that into a BitArray
            var bytes2 = new byte[bytes.Length - 5];
            System.Array.Copy(bytes, 5, bytes2, 0, bytes2.Length);
            var bits = new BitArray(bytes2);
            // Get the number of entries from the first 4 bytes after the identifier and resize the BitArray to that length, then convert it to a boolean array
            bits.Length = ConvertBytesToInt32(bytes);
            var boolArray = new bool[bits.Count];
            bits.CopyTo(boolArray, 0);
```

```
            return boolArray;
        }
        return new bool[0];
    }

    public static bool[] GetBoolArray(String key, bool defaultValue, int defaultSize)
    {
        if (PlayerPrefs.HasKey(key))
        {
            return GetBoolArray(key);
        }
        var boolArray = new bool[defaultSize];
        for (int i = 0; i < defaultSize; i++)
        {
            boolArray[i] = defaultValue;
        }
        return boolArray;
    }

    public static bool SetStringArray(String key, String[] stringArray)
    {
        var bytes = new byte[stringArray.Length + 1];
        bytes[0] = System.Convert.ToByte(ArrayType.String); // Identifier
        Initialize();

        // Store the length of each string that's in stringArray, so we can extract the correct strings in GetStringArray
        for (var i = 0; i < stringArray.Length; i++) { if (stringArray[i] == null) { Debug.LogError("Can't save null entries in the string array when setting " + key); return false; } if (stringArray[i].Length > 255)
            {
                Debug.LogError("Strings cannot be longer than 255 characters when setting " + key);
                return false;
            }
            bytes[idx++] = (byte)stringArray[i].Length;
        }

        try
        {
            PlayerPrefs.SetString(key, System.Convert.ToBase64String(bytes) + "|" + String.Join("", stringArray));
        }
        catch
        {
            return false;
        }
        return true;
    }

    public static String[] GetStringArray(String key)
    {
        if (PlayerPrefs.HasKey(key))
        {
            var completeString = PlayerPrefs.GetString(key);
            var separatorIndex = completeString.IndexOf("|"[0]);
            if (separatorIndex < 4)
            {
                Debug.LogError("Corrupt preference file for " + key);
                return new String[0];
            }
            var bytes = System.Convert.FromBase64String(completeString.Substring(0, separatorIndex));
            if ((ArrayType)bytes[0] != ArrayType.String)
            {
                Debug.LogError(key + " is not a string array");
                return new String[0];
            }
            Initialize();

            var numberOfEntries = bytes.Length - 1;
            var stringArray = new String[numberOfEntries];
            var stringIndex = separatorIndex + 1;
            for (var i = 0; i < numberOfEntries; i++) { int stringLength = bytes[idx++]; if (stringIndex + stringLength > completeString.Length)
```

```
            {
                Debug.LogError("Corrupt preference file for " + key);
                return new String[0];
            }
            stringArray[i] = completeString.Substring(stringIndex, stringLength);
            stringIndex += stringLength;
        }

        return stringArray;
    }
    return new String[0];
}

public static String[] GetStringArray(String key, String defaultValue, int defaultSize)
{
    if (PlayerPrefs.HasKey(key))
    {
        return GetStringArray(key);
    }
    var stringArray = new String[defaultSize];
    for (int i = 0; i < defaultSize; i++)
    {
        stringArray[i] = defaultValue;
    }
    return stringArray;
}

public static bool SetIntArray(String key, int[] intArray)
{
    return SetValue(key, intArray, ArrayType.Int32, 1, ConvertFromInt);
}

public static bool SetFloatArray(String key, float[] floatArray)
{
    return SetValue(key, floatArray, ArrayType.Float, 1, ConvertFromFloat);
}

public static bool SetVector2Array(String key, Vector2[] vector2Array)
{
    return SetValue(key, vector2Array, ArrayType.Vector2, 2, ConvertFromVector2);
}

public static bool SetVector3Array(String key, Vector3[] vector3Array)
{
    return SetValue(key, vector3Array, ArrayType.Vector3, 3, ConvertFromVector3);
}

public static bool SetQuaternionArray(String key, Quaternion[] quaternionArray)
{
    return SetValue(key, quaternionArray, ArrayType.Quaternion, 4, ConvertFromQuaternion);
}

public static bool SetColorArray(String key, Color[] colorArray)
{
    return SetValue(key, colorArray, ArrayType.Color, 4, ConvertFromColor);
}

private static bool SetValue<T>(String key, T array, ArrayType arrayType, int vectorNumber, Action<T
{
    var bytes = new byte[(4 * array.Count) * vectorNumber + 1];
    bytes[0] = System.Convert.ToByte(arrayType);    // Identifier
    Initialize();

    for (var i = 0; i < array.Count; i++)
    {
        convert(array, bytes, i);
    }
    return SaveBytes(key, bytes);
}
```

```
345     private static void ConvertFromInt(int[] array, byte[] bytes, int i)
346     {
347         ConvertInt32ToBytes(array[i], bytes);
348     }
349
350     private static void ConvertFromFloat(float[] array, byte[] bytes, int i)
351     {
352         ConvertFloatToBytes(array[i], bytes);
353     }
354
355     private static void ConvertFromVector2(Vector2[] array, byte[] bytes, int i)
356     {
357         ConvertFloatToBytes(array[i].x, bytes);
358         ConvertFloatToBytes(array[i].y, bytes);
359     }
360
361     private static void ConvertFromVector3(Vector3[] array, byte[] bytes, int i)
362     {
363         ConvertFloatToBytes(array[i].x, bytes);
364         ConvertFloatToBytes(array[i].y, bytes);
365         ConvertFloatToBytes(array[i].z, bytes);
366     }
367
368     private static void ConvertFromQuaternion(Quaternion[] array, byte[] bytes, int i)
369     {
370         ConvertFloatToBytes(array[i].x, bytes);
371         ConvertFloatToBytes(array[i].y, bytes);
372         ConvertFloatToBytes(array[i].z, bytes);
373         ConvertFloatToBytes(array[i].w, bytes);
374     }
375
376     private static void ConvertFromColor(Color[] array, byte[] bytes, int i)
377     {
378         ConvertFloatToBytes(array[i].r, bytes);
379         ConvertFloatToBytes(array[i].g, bytes);
380         ConvertFloatToBytes(array[i].b, bytes);
381         ConvertFloatToBytes(array[i].a, bytes);
382     }
383
384     public static int[] GetIntArray(String key)
385     {
386         var intList = new List<int>();
387         GetValue(key, intList, ArrayType.Int32, 1, ConvertToInt);
388         return intList.ToArray();
389     }
390
391     public static int[] GetIntArray(String key, int defaultValue, int defaultSize)
392     {
393         if (PlayerPrefs.HasKey(key))
394         {
395             return GetIntArray(key);
396         }
397         var intArray = new int[defaultSize];
398         for (int i = 0; i < defaultSize; i++)
399         {
400             intArray[i] = defaultValue;
401         }
402         return intArray;
403     }
404
405     public static float[] GetFloatArray(String key)
406     {
407         var floatList = new List<float>();
408         GetValue(key, floatList, ArrayType.Float, 1, ConvertToFloat);
409         return floatList.ToArray();
410     }
411
412     public static float[] GetFloatArray(String key, float defaultValue, int defaultSize)
413     {
```

```
414         if (PlayerPrefs.HasKey(key))
415         {
416             return GetFloatArray(key);
417         }
418         var floatArray = new float[defaultSize];
419         for (int i = 0; i < defaultSize; i++)
420         {
421             floatArray[i] = defaultValue;
422         }
423         return floatArray;
424     }
425
426     public static Vector2[] GetVector2Array(String key)
427     {
428         var vector2List = new List<Vector2>();
429         GetValue(key, vector2List, ArrayType.Vector2, 2, ConvertToVector2);
430         return vector2List.ToArray();
431     }
432
433     public static Vector2[] GetVector2Array(String key, Vector2 defaultValue, int defaultSize)
434     {
435         if (PlayerPrefs.HasKey(key))
436         {
437             return GetVector2Array(key);
438         }
439         var vector2Array = new Vector2[defaultSize];
440         for (int i = 0; i < defaultSize; i++)
441         {
442             vector2Array[i] = defaultValue;
443         }
444         return vector2Array;
445     }
446
447     public static Vector3[] GetVector3Array(String key)
448     {
449         var vector3List = new List<Vector3>();
450         GetValue(key, vector3List, ArrayType.Vector3, 3, ConvertToVector3);
451         return vector3List.ToArray();
452     }
453
454     public static Vector3[] GetVector3Array(String key, Vector3 defaultValue, int defaultSize)
455     {
456         if (PlayerPrefs.HasKey(key))
457
458         {
459             return GetVector3Array(key);
460         }
461         var vector3Array = new Vector3[defaultSize];
462         for (int i = 0; i < defaultSize; i++)
463         {
464             vector3Array[i] = defaultValue;
465         }
466         return vector3Array;
467     }
468
469     public static Quaternion[] GetQuaternionArray(String key)
470     {
471         var quaternionList = new List<Quaternion>();
472         GetValue(key, quaternionList, ArrayType.Quaternion, 4, ConvertToQuaternion);
473         return quaternionList.ToArray();
474     }
475
476     public static Quaternion[] GetQuaternionArray(String key, Quaternion defaultValue, int defaultSize)
477     {
478         if (PlayerPrefs.HasKey(key))
479         {
480             return GetQuaternionArray(key);
481         }
482         var quaternionArray = new Quaternion[defaultSize];
```

```
        for (int i = 0; i < defaultSize; i++)
        {
            quaternionArray[i] = defaultValue;
        }
        return quaternionArray;
    }

    public static Color[] GetColorArray(String key)
    {
        var colorList = new List<Color>();
        GetValue(key, colorList, ArrayType.Color, 4, ConvertToColor);
        return colorList.ToArray();
    }

    public static Color[] GetColorArray(String key, Color defaultValue, int defaultSize)
    {
        if (PlayerPrefs.HasKey(key))
        {
            return GetColorArray(key);
        }
        var colorArray = new Color[defaultSize];
        for (int i = 0; i < defaultSize; i++)
        {
            colorArray[i] = defaultValue;
        }
        return colorArray;
    }

    private static void GetValue<T>(String key, T list, ArrayType arrayType, int vectorNumber, Action<T, byte[]> convert) where T : IList
    {
        if (PlayerPrefs.HasKey(key))
        {
            var bytes = System.Convert.FromBase64String(PlayerPrefs.GetString(key));
            if ((bytes.Length - 1) % (vectorNumber * 4) != 0)
            {
                Debug.LogError("Corrupt preference file for " + key);
                return;
            }
            if ((ArrayType)bytes[0] != arrayType)
            {
                Debug.LogError(key + " is not a " + arrayType.ToString() + " array");
                return;
            }
            Initialize();

            var end = (bytes.Length - 1) / (vectorNumber * 4);
            for (var i = 0; i < end; i++)
            {
                convert(list, bytes);
            }
        }
    }

    private static void ConvertToInt(List<int> list, byte[] bytes)
    {
        list.Add(ConvertBytesToInt32(bytes));
    }

    private static void ConvertToFloat(List<float> list, byte[] bytes)
    {
        list.Add(ConvertBytesToFloat(bytes));
    }

    private static void ConvertToVector2(List<Vector2> list, byte[] bytes)
    {
        list.Add(new Vector2(ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes)));
    }

    private static void ConvertToVector3(List<Vector3> list, byte[] bytes)
```

```
    {
        list.Add(new Vector3(ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes)));
    }

    private static void ConvertToQuaternion(List<Quaternion> list, byte[] bytes)
    {
        list.Add(new Quaternion(ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes)));
    }

    private static void ConvertToColor(List<Color> list, byte[] bytes)
    {
        list.Add(new Color(ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes), ConvertBytesToFloat(bytes)));
    }

    public static void ShowArrayType(String key)
    {
        var bytes = System.Convert.FromBase64String(PlayerPrefs.GetString(key));
        if (bytes.Length > 0)
        {
            ArrayType arrayType = (ArrayType)bytes[0];
            Debug.Log(key + " is a " + arrayType.ToString() + " array");
        }
    }

    private static void Initialize()
    {
        if (System.BitConverter.IsLittleEndian)
        {
            endianDiff1 = 0;
            endianDiff2 = 0;
        }
        else
        {
            endianDiff1 = 3;
            endianDiff2 = 1;
        }
        if (byteBlock == null)
        {
            byteBlock = new byte[4];
        }
        idx = 1;
    }

    private static bool SaveBytes(String key, byte[] bytes)
    {
        try
        {
            PlayerPrefs.SetString(key, System.Convert.ToBase64String(bytes));
        }
        catch
        {
            return false;
        }
        return true;
    }

    private static void ConvertFloatToBytes(float f, byte[] bytes)
    {
        byteBlock = System.BitConverter.GetBytes(f);
        ConvertTo4Bytes(bytes);
    }

    private static float ConvertBytesToFloat(byte[] bytes)
    {
        ConvertFrom4Bytes(bytes);
        return System.BitConverter.ToSingle(byteBlock, 0);
    }

    private static void ConvertInt32ToBytes(int i, byte[] bytes)
```

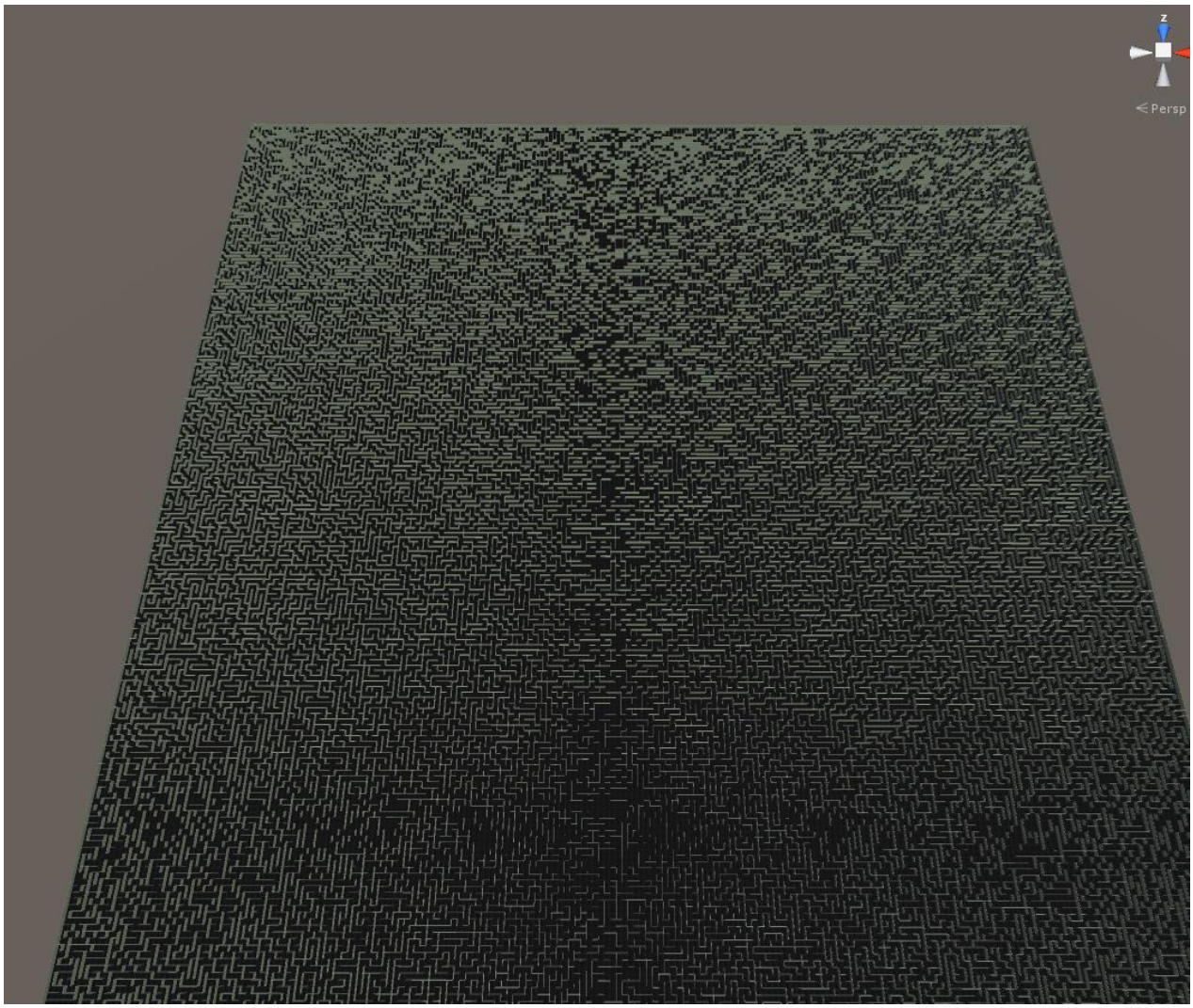
```
621     {
622         byteBlock = System.BitConverter.GetBytes(i);
623         ConvertTo4Bytes(bytes);
624     }
625
626     private static int ConvertBytesToInt32(byte[] bytes)
627     {
628         ConvertFrom4Bytes(bytes);
629         return System.BitConverter.ToInt32(byteBlock, 0);
630     }
631
632     private static void ConvertTo4Bytes(byte[] bytes)
633     {
634         bytes[idx] = byteBlock[endianDiff1];
635         bytes[idx + 1] = byteBlock[1 + endianDiff2];
636         bytes[idx + 2] = byteBlock[2 - endianDiff2];
637         bytes[idx + 3] = byteBlock[3 - endianDiff1];
638         idx += 4;
639     }
640
641     private static void ConvertFrom4Bytes(byte[] bytes)
642     {
643         byteBlock[endianDiff1] = bytes[idx];
644         byteBlock[1 + endianDiff2] = bytes[idx + 1];
645         byteBlock[2 - endianDiff2] = bytes[idx + 2];
646         byteBlock[3 - endianDiff1] = bytes[idx + 3];
647         idx += 4;
648     }
649 }
```

Se avete dato uno sguardo al codice, vi sarete accorti che **PlayerPrefsX** è costruito su PlayerPrefs.SetString. I lavoro che fa **PlayerPrefsX** non è altro che quello che dovreste fare voi, ogni volta che vi avrete la necessità di salvare una serie di dati compositi, come array e vettori.

Una volta creato questo nuovo script, potrete usare la classe **PlayerPrefsX** (con la X finale) ogni qual volta avrete la necessità di salvare un array, un Vector o un altro tipo di dato non presente tra le opzioni di salvataggio del **PlayerPrefs**.
Per le tre variabili string, int e float, potrete usare normalmente il **PlayerPrefs.**
Stesso dicasi per il metodo *HasKey()*.

Creare un Labirinto Procedurale

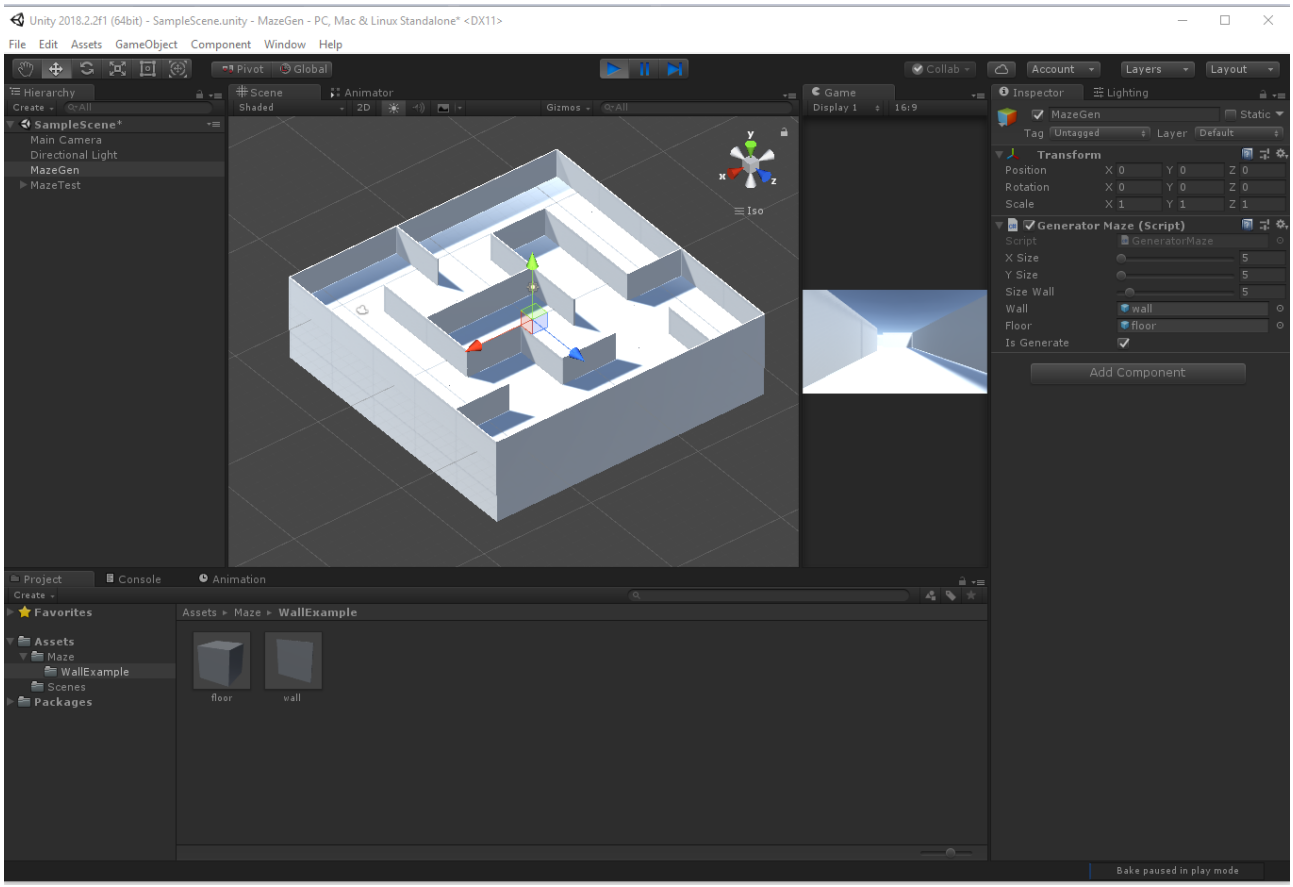
Benvenuti nel primo degli articoli scritti da voi utenti!
Il caro Simone Zambonardi ci ha inviato un interessante sistema per la creazione di labirinti randomizzati a runtime. Un ottima soluzione per la creazione di livelli sempre diversificati ad ogni avvio, un po' come succede nei vari titoli della serie Diablo e molti altri.
Ciò che vi salterà subito all'occhio sarà l'estrema velocità e semplicità di utilizzo. Impostando un paio di valori potrete generare livelli sempre diversificati. E se avrete le capacità, potrete inserire nuove features allo script, come per esempio la possibilità di randomizzare diversi tipi di muri e tante altre cose.
L'algoritmo si basa su una matrice (max X max) scelta dall'utente. Tale matrice ha il compito di creare e disporre le pareti in maniera da creare un fitto labirinto visitabile dal giocatore.



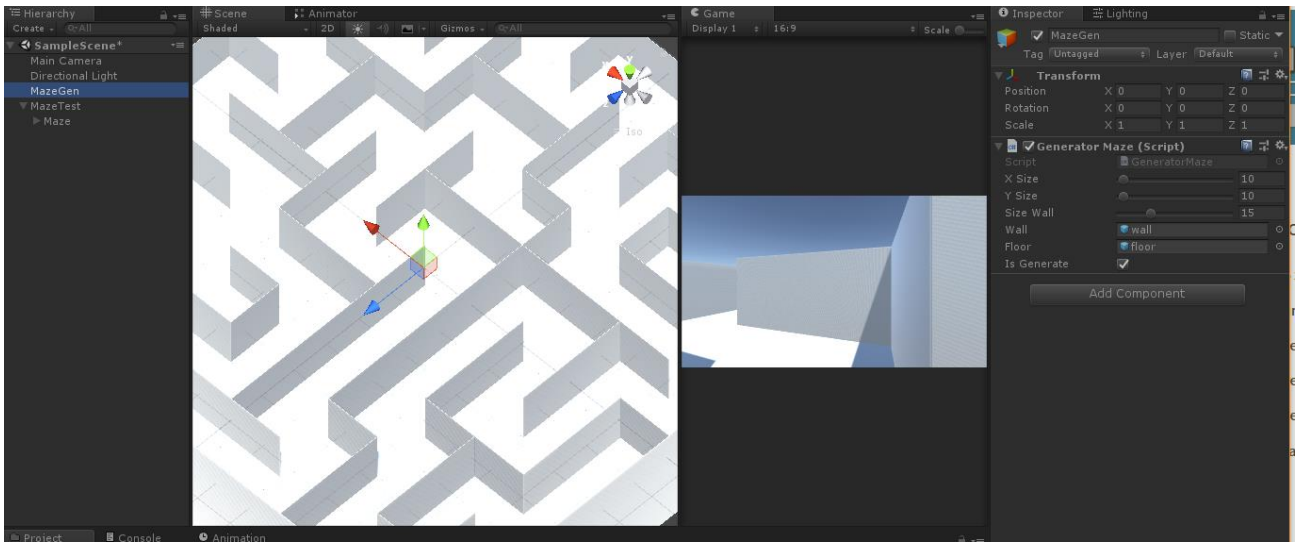
Questo è per esempio ciò che potrete generare con una matrice 200×200.

Davvero un sacco di cunicoli in cui perdersi

Con un paio di click potrete generare un livello pronto per essere esplorato dai giocatori.
Questo è per esempio un livello 5×5 con due semplici cubi come prefabs dei muri e del pavimento.



Ma potrete impostare le variabili come più vi aggradano generando livelli sempre diversificati.



Capirete da soli che potenzialità di questo script sono davvero enormi. Esistono sono ampi margini di miglioramento ma già così potrete ottenere livelli diversificati, senza dover posizionare muri e piattaforme “a mano”.
Vi lascio dunque il documento in formato .pdf scritto dall’autore dello script Simone Zambonardi e un package che contiene lo script e una scena di esempio.

Link al Package (13Kb) - http://unity3dtutorials.it/Download/MazeGen.unitypackage

Di seguito, il PDF del MAZE GENERATOR

MAZE GENERATOR

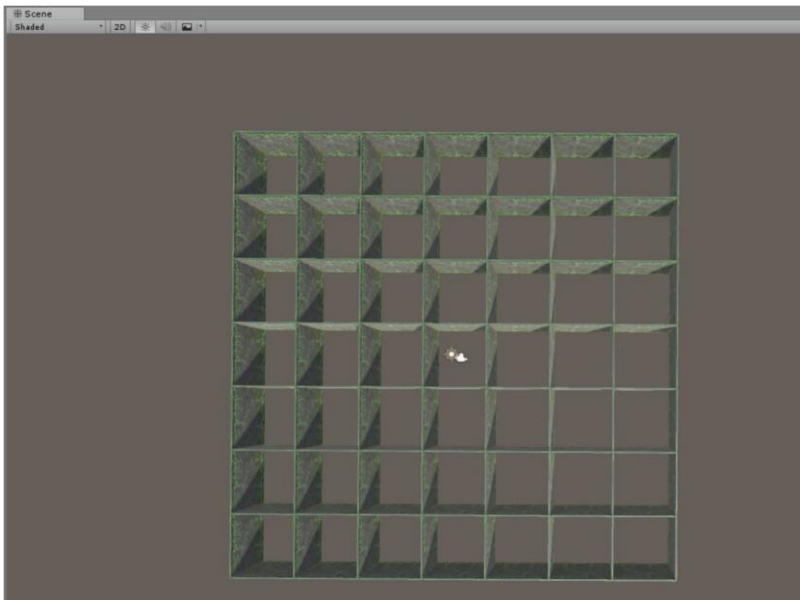
PSEUDOCODE

1) Scegli a random la cella iniziale, rendila “cella corrente” e contrassegnala come visitata
2) Trova i suoi vicini che non sono stati visitati
3) Scegli a random un suo vicino “non visitato”
4) Rendi il vicino come “cella corrente”
5) Distruggi il muro che separano queste due celle
6) Torna al punto 2 fino a quando tutte le celle non sono state visitate

L'algoritmo si basa su una matrice (max X max) scelta dall'utente. Il primo passo è creare i muri del labirinto. Con questa funzione creeremo la matrice.

```
void createWalls(GameObject allParent)
{
    Maze = new GameObject("Maze");

    //X
    for (int i = 0; i < ySize; i++)
    {
        for (int j = 0; j <= xSize; j++)
        {
            GameObject wallX = Instantiate(wall,
                                    new Vector3(initialPos.x + (j * sizeWall), 0, initialPos.z + (i * sizeWall)),
                                    Quaternion.identity) as GameObject;
            wallX.transform.parent = Maze.transform;

        }
    }


    //Y
    for (int i = 0; i <= ySize; i++)
    {
        for (int j = 0; j < xSize; j++)
        {
            GameObject wallY = Instantiate(wall,
                                    new Vector3(initialPos.x + (j * sizeWall) + middle, 0, initialPos.z + (i * sizeWall) - middle),
                                    Quaternion.Euler(0, 90, 0)) as GameObject;
            wallY.transform.parent = Maze.transform;
        }
    }
    Maze.transform.parent = allParent.transform;
}
```

RISULTATO



Come seconda cosa bisogna creare il pavimento per ogni cella

```
void createFloor(GameObject allParent)
{
    Floor = new GameObject("Floor"); //group all the Quad to keep everything in order

    for (int i = 0; i < ySize; i++)
    {
        for (int j = 0; j < xSize; j++)
        {
            var insFloor = Instantiate(floor, new Vector3((j * sizeWall) + middle, -middle, i * sizeWall), Quaternion.Euler(90, 0, 0)) as GameObject;

            insFloor.transform.parent = Floor.transform; //group all the Quad to keep everything in order
            indexCell = (i * xSize) + j; //I transform the array into an array
            insFloor.transform.name = indexCell.ToString();
        }
    }
    Floor.transform.parent = allParent.transform;
}
```

La variabile “middle” serve per allineare il pavimento in base alla grandezza dei muri scelto dempre dall'utente

Una volta creati gli elementi principali bisogna definire il termine “cella”. La cella è una classe che rappresenta ogni quadrato della matrice di muri. Questa classe comprende :

- muro EST
- muro OVEST
- muro NORD
- muro SUD
- pavimento
- booleana se è stata visitata

```
[SerializeField]
12 riferimenti
public class Cell
{
    public bool isVisited;
    public GameObject north;
    public GameObject south;
    public GameObject east;
    public GameObject west;
    public GameObject floor;
    public int current; //Debug
    public Vector3 cellPosition; //Debug


}
```

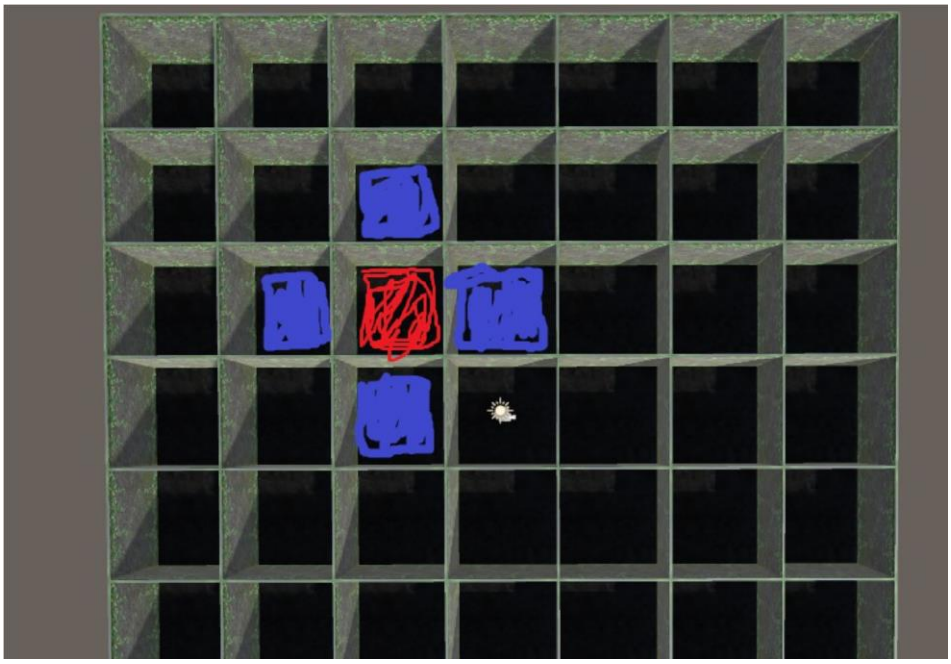
Una volta stabilita la classe bisogna creare effettivamente la cella.

```
void createCell()
{
    walls = new GameObject[Maze.transform.childCount];
    floors = new GameObject[xSize * ySize];


    int south = (xSize + 1) * ySize;
    int north = ((xSize + 1) * ySize) + xSize;

    for (int i = 0; i < Maze.transform.childCount; i++)
    {
        walls[i] = Maze.transform.GetChild(i).gameObject;
    }


    for (int i = 0; i < xSize * ySize; i++)
    {
        floors[i] = Floor.transform.GetChild(i).gameObject;
    }

    for (int i = 0; i < xSize * ySize; i++)
    {

        cells.Add(new Cell());
        currentRow = (i / xSize); //find current row for every cell


        cells[i].south = walls[i + south];
        cells[i].north = walls[i + north];
        cells[i].east = walls[(currentRow + i) + 1];
        cells[i].west = walls[currentRow + i];
        cells[i].floor = floors[i];
        cells[i].current = i;


    }


    mazeCreation(cells);



}
```

Con questo metodo abbiamo creato la cella. Ora ogni cella ha i propri muri, pavimenti ecc... Ogni cella ha dei vicini. Il seguente metodo mostra come trovare i vicini di ogni cella considerando anche le celle ai margini del labirinto.

```
void AddNears(Cell currentCell)
{


    int currentRow = (currentCell.current / xSize) + 1;
    int eastLimited = currentCell.current % xSize;
    int westLimited = eastLimited;
    int southLimited = xSize - 1;
    int northLimited = currentCell.current / xSize;



    //East
    if (eastLimited != xSize - 1)
    {
        if (cells[currentCell.current + 1].isVisited == false)
            neighboringCell.Add(cells[currentCell.current + 1]);

    }
    //West
    if (westLimited != 0)
    {
        if (cells[currentCell.current - 1].isVisited == false)
            neighboringCell.Add(cells[currentCell.current - 1]);

    }
    //South
    if (currentCell.current > southLimited)
    {
        if (cells[currentCell.current - xSize].isVisited == false)
            neighboringCell.Add(cells[currentCell.current - xSize]);

    }
    //North
    if (northLimited != ySize - 1)
    {
        if (cells[currentCell.current + xSize].isVisited == false)
            neighboringCell.Add(cells[currentCell.current + xSize]);

    }
}
```

Ora tutte le celle hanno i propri vicini.
ESEMPIO: cella rossa = cella corrente. Celle blu = vicini della cella rossa



il passo successivo è quello di creare il metodo per distrugere i muri tra la cella corrente e il suo vicino scelto a random

```
void breakWall(Cell curr, Cell choose)
{
    //East
    if (choose.current == curr.current + 1)
    {
        Destroy(curr.east);
        currentCell = cells[currentCell.current + 1];


    }

    //West
    if (choose.current == curr.current - 1)
    {
        Destroy(curr.west);
        currentCell = cells[currentCell.current - 1];


    }
    //South

    if (choose.current == curr.current - xSize)
    {
        Destroy(curr.south);
        currentCell = cells[currentCell.current - xSize];



    }
    //North
    if (choose.current == curr.current + xSize)
    {
        Destroy(curr.north);
        currentCell = cells[currentCell.current + xSize];

    }


}
```

Una volta creati tutti i metodi essenziali, si scrive l'algoritmo

```
void mazeCreation(List<Cell> allCells)
{

    int visitedCell = 0;


    int randomNeighboring;
    int random = Random.Range(0, allCells.Count); //I choose a cell at random
    currentCell = allCells[random]; //the choice cell becomes the current cell



    while (visitedCell < allCells.Count) //I run the cycle for all the cells of the matrix
    {
        AddNears(currentCell); //add to the selection cell neighboring cells
         randomNeighboring = Random.Range(0, neighboringCell.Count); //I choose a neighboring cell at random



        if (neighboringCell.Any()) //if there are cells neighboring the current cell
        {
            lastCellVisited.Add(currentCell); //becomes visited

            breakWall(currentCell, neighboringCell[randomNeighboring]); //Break the wall

            neighboringCell.Clear(); //Reset the list of neighboring cells
            visitedCell++;
            currentCell.isVisited = true;


        }

        else
        {
            //else I choose a already been visited cell
            random = Random.Range(0, lastCellVisited.Count);
            currentCell = lastCellVisited[random];

        }

    }

    isGenerate = true;

}
```

Risultato finale di una matrice 20X20