PRACTICA 2

Curvas Bezier Bicúbicas + Interpolación:

He implementado las curvas y el movimiento de un objeto a través de ellas con un movimiento uniforme que no depende de la curvatura de la curva (segunda derivada = velocidad de la curva, pero no es la que queremos porque se movería más rápido a menor curvatura y más lento a mayor curvatura).

Para ello he utilizado LookUpTables con la distancia acumulada de la curva y su t correspondiente, y las actualizo cada vez que mis puntos de control cambian de posición:

```
// PointCloud Update()
if (transform.hasChanged && bezier)
{
   bezier.UpdateControlPoints();
   bezier.UpdateLineRenderer();
   transform.hasChanged = false;
}
```

Los puntos de control se obtienen como la posición de cada uno de los objetos hijo de la curva:

```
// Bezier
public void UpdateControlPoints()
        {
   ControlPoint[] controlPoints = GetComponentsInChildren<ControlPoint>();
   cpPositions = new Vector3[controlPoints.Length];
   for (int i = 0; i < controlPoints.Length; i++)
   {
      cpPositions[i] = controlPoints[i].transform.position;
   }
                UpdateBezierPoints();
}
```

Y actualizo las LookUpTables vaciándolas e iterando por la curva con incrementos de T tan pequeños como precisa quiero que sea. La distancia acumulada será la distancia del t anterior + la distancia entre el punto anterior y el nuevo punto en t.

En mi caso he utilizado una resolución de 0.01.

Además, también actualizo una LookUpTable de puntos de la curva a partir de su t, lo cual se calcularía con la ecuación de la bezier bicúbica, aunque no es tan necesario, solo reduce el tiempo de cálculo. Seguramente sea mejor opción descartar esta LookUpTable ya que mis consultas de puntos de la curva se hacen con valores de t interpolados, por lo que muy difílmente caerá precisamente en un punto ya calculado. Antes cuando no interpolaba si era útil.

```
private void UpdateBezierPoints()
        {
                // Vaciamos las LUT
                LUTpuntosT.Clear();
                LUTdistanceByT.Clear();
                LUTtByDistance.Clear();

                // El parametro de la linea t va incrementado segun la resolucion de la curva
                decimal t = 0;
```

```
                // Inicio
                LUTdistanceByT[0] = 0;
                LUTtByDistance[0] = 0;
                LUTpuntosT[0] = cpPositions[0];

                while (t <= 1)
                {
                        decimal tAnterior = t;
                        t += BezierResolution; // + 0.01

                        Vector3 point = GetBezierPointT(t);

                        LUTpuntosT.Add(t, point);

                        // distancia(t1) = ( Distancia entre p(t0) y p(t1) ) + distancia(t0)
                        float distancia = (point - LUTpuntosT[tAnterior]).magnitude +
LUTdistanceByT[tAnterior];
                        LUTdistanceByT.Add(t, distancia);
                        if (!LUTtByDistance.ContainsKey(distancia))
                                LUTtByDistance.Add(distancia, t);
                }
        }
```

He implementado como ecuación de Bezier la general y no la bicúbica pensando en si en un futuro podría escalarlo a curvas de Bezier de mayor o menor grado.

```
  private void UpdateBezierPoints()
        {
                // Vaciamos las LUT
                LUTpuntosT.Clear();
                LUTdistanceByT.Clear();
                LUTtByDistance.Clear();

                // El parametro de la linea t va incrementado segun la resolucion de la curva
                decimal t = 0;

                // Inicio
                LUTdistanceByT[0] = 0;
                LUTtByDistance[0] = 0;
                LUTpuntosT[0] = cpPositions[0];

                while (t <= 1)
                {
                        decimal tAnterior = t;
                        t += BezierResolution;

                        Vector3 point = GetBezierPointT(t);

                        LUTpuntosT.Add(t, point);

                        // distancia(t1) = ( Distancia entre p(t0) y p(t1) ) + distancia(t0)
                        float distancia = (point - LUTpuntosT[tAnterior]).magnitude +
LUTdistanceByT[tAnterior];
                        LUTdistanceByT.Add(t, distancia);
                        if (!LUTtByDistance.ContainsKey(distancia))
                                LUTtByDistance.Add(distancia, t);
```

```
		}
	}

  // COMBINATORIA (usa memoization)
	// Cache con una lista de resultados porque se van a repetir mucho
	private readonly Dictionary<Tuple<float, float>, float> _combCache = new
Dictionary<Tuple<float, float>, float>();
```

Además he implementado memoización para la Combinatoria ya que las llamadas a esta función eran muy frecuentes y las combinatorias que se calculan son poquísimas

```
	private float Combinatoria(float n, float k)
	{
		Tuple<float, float> key = new Tuple<float, float>(n, k);
		if (_combCache.ContainsKey(key))
			return _combCache[key];

		if (k.Equals(0))
			return 1;

		if (k.Equals(1))
			return n;

		// Combinatoria = n(n-1)(n-2)...(n - k+1) / k!
		float comb = n;
		for (int i = 1; i < k; ++i)
			comb *= n - i;

		// Factorial de k = 1*2*3*4*...*k
		float factorial = 1;
		for (int i = 1; i <= k; ++i)
			factorial *= i;

		_combCache.Add(key, comb / factorial);
		return _combCache[key];
	}
```

Una vez implementadas las LookUpTable necesitamos métodos para consultarlas, y dado que los valores no son continuos habrá que calcular valores intermedios por medio de Interpolación:

```
  public float GetDist(decimal t)
	{
		if (t <= 0)
			return 0;

		if (t >= 1)
			return GetLenght();

		if (LUTdistanceByT.ContainsKey(t))
			return LUTdistanceByT[t];

		// Si no esta registrado en la LUT buscamos el t mas cercano
		for (decimal t0 = 0; t0 <= 1; t0 += BezierResolution)
		{
			if (t >= t0)
			{
				decimal t1 = t0 + BezierResolution;
```

```
                                        // Remapeado del rango [t0,t1] a [d0,d1] por Interpolacion Lineal
                                        return ((float)t).Remap((float)t0, (float)t1, LUTdistanceByT[t0],
LUTdistanceByT[t1]);
                                }
                        }

                        // Si no interpolamos con distancias continuas a las malas
                        return (float)t * GetLenght();
                }

        public decimal GetT(float distance)
                {
                        // Casos Triviales:
                        // Si se sale de la curva extrapolar
                        if (distance <= 0 || distance >= GetLenght())
                        {
                                return (decimal)(distance / GetLenght());
                        }

                        // Last distance
                        float s0 = 0;
                        // Buscar por toda la tabla de distancias hasta que encaje en un hueco y interpolar la
t del segmento
                        foreach (float s1 in LUTtByDistance.Keys)
                        {
                                if (distance <= s1 && distance >= s0)
                                {
                                        decimal t0 = LUTtByDistance[s0];
                                        decimal t1 = LUTtByDistance[s1];

                                        // INTERPOLACION entre los dos puntos t (% deltaS => % deltaT)
                                        // t = (fraccion de distancia que sobrepasa) * (segmento t) + t0
                                        return ((decimal)distance).Remap((decimal)s0, (decimal)s1, t0, t1);
                                }
                                s0 = s1;
                        }

                        // Interpolamos si no encuentra en la tabla un hueco
                        return (decimal)Mathf.InverseLerp(0, GetLenght(), distance);
                }
```

La interpolación la realizo con un método Remap(a, b, t0, t1) que realiza una interpolación a los valores [a,b] a partir de t, y luego una interpolación inversa al rango [t0, t1] con el resultado, que quedaría así:

```
(value - a) / (b - a) * (t1 - t0) + t0
```

¿Por qué uso decimal para los valores de t, y float para las distancias en la curva?

En principio, implementé todo de distinta forma y tuve problemas con la precisión de t al crear la LookUpTable. Cuando incrementas un valor float con un incremento muy pequeño (0.01) tras varias iteraciones perdía precisión y cuando llegaba, por ejemplo, a 0.81, 0.82, 0.83 seguía con 0.839999, 0.849999, 0.859999, por lo que luego al consultar la tabla de valores la t no encajaba la mayoría de las veces, y como no tenía implementada la interpolación aproximaba al más cercano, lo cual daba un resultado muy tosco, como si el objeto fuera a saltos.

Una vez implementada la interpolación ya no era necesario, pero debugear me resultaba más difícil, así que lo dejé así con decimal, el cual tiene la mayor precisión. Ni siquiera el double era capaz de dar un buen resultado si se le incrementaba muchas veces en un incremento pequeño.

Ease In / Ease Out

Fui haciendo iteraciones en mi código conforme iba descifrando las matemáticas de la curva. Al principio tomaba la velocidad y la aceleración de cada Ease como input y calculaba la posición del objeto a partir de estos.

Luego conseguí implementar las ecuaciones de Ease In / Ease Out que obtienen la distancia recorrida a partir de el tiempo de animación y la fracción de curva con cada Ease.

Despejé la velocidad a partir de la última ecuación, aunque estoy seguro que se podría despejar de la primera o la segunda:

```
float v0 = 1 / (-t1/2 + 1 - (1-t2)/2);
```

El tiempo lo normalizo de rango [0, TiempoAnimación] a [0,1]:

```
float timeNormalized = Mathf.InverseLerp(0, animationTime, time);
```

Y para cada fracción de curva, la cual depende del tiempo normalizado, si está antes de t1, después y después de t2, calculo la distancia recorrida a partir de su ecuación:

```
// Ease IN
if (tiempoNormalizado < t1)
    d = v0 * timeNormalized * timeNormalized / 2 / t1;

// Ease Middle
if (tiempoNormalizado >= t1 && tiempoNormalizado <= t2)
    d = v0 * t1 / 2 + v0 * (timeNormalized - t1);

// Ease OUT
if (tiempoNormalizado > t2)
    d = v0 * t1 / 2 + v0 * (t2 - t1) + (v0 - (v0 * (timeNormalized - t2) / (1 - t2)) / 2) * (timeNormalized - t2);
```

Desnormalizo la distancia para colocarla sobre la curva y obtener el parámetro t:

```
d *= Bezier.GetLenght();

        return (float)Bezier.GetT(d);
```

Una vez tengo la t de la curva donde el objeto debe estar en un momento de tiempo puedo calcular su posición:

```
transform.position = Bezier.GetBezierPointT(t)
```

Y su rotación, que en realidad gracias a Quaternion.LookRotation(direccion), se puede obtener a partir del vector tangente de la curva, que coincide con la velocidad de la curva que se calcula como la derivada.

Esta rotación se interpola esféricamente con Quaternion.Slerp(Q0, Q1, deltaTime):

```
  private Quaternion RotateTowardsCurve(float t)
        {
                // Los cuerpos con masa dan problemas con la rotacion
                Rigidbody rb = GetComponent<Rigidbody>();
                if (rb)
                        rb.mass = 0;

                Vector3 curveVelocity = Bezier.GetVelocity(t).normalized;
                Vector3 up = Vector3.Cross(Vector3.Cross(Bezier.GetAcceleration(t).normalized,
curveVelocity), curveVelocity);

                // Apunta en direccion de la Tangente de la Curva (Derivada)
                Quaternion tangentQuat = Quaternion.LookRotation(curveVelocity, up);

                if (rb)
                        rb.MoveRotation(
                                Quaternion.Slerp(
                                        transform.rotation,
                                        tangentQuat,
                                        (Time.inFixedTimeStep ? Time.fixedDeltaTime :
Time.deltaTime) * Bezier.GetAcceleration(t).magnitude
                                )
                        );
                else
                        transform.rotation = Quaternion.Slerp(
                                transform.rotation,
                                tangentQuat,
                                (Time.inFixedTimeStep ? Time.fixedDeltaTime : Time.deltaTime) *
Bezier.GetAcceleration(t).magnitude
                        );

                return tangentQuat;
        }
```

Además, la **aceleración** de la curva, que es la 2ª derivada, nos dice **hacia donde se esta curvando**, como una fuerza que tira de ella, y nos permite calcular un **sistema de coordenadas** sacando la perpendicular con la velocidad, y la perpendicular de ésta y la velocidad, cuyo vector se puede interpretar como el **up** del objeto, lo cual si lo usamos con un avión u objetos similares da resultados mucho más **realistas**. En mi caso, al estar haciendo pruebas con el avión y doblar la curva de forma vertical, la rotación del avión era muy brusca, cuando se va a poner boca abajo pega un giro forzandolo a ponerse boca arriba. Una vez implementado el vector up, el avión se mantiene **boca abajo** de forma realista.

Además, también se puede usar la aceleración como un medidor de la curvatura en ese punto de la curva, por lo que podemos interpolar nuestras rotaciones con una velocidad angular proporcional este valor en QuaternionSlerp(Q0, Q1, deltaTime * P''(t));

Control de la Curva

Para cambiar los puntos de control se pueden mover **arrastrandolos con el ratón**.
Para que fuera cómodo he dejado 2 cámaras estáticas desde arriba y desde el lateral para poder mover los puntos de control en 3D con libertad.

Se puede alternar entre las cámaras con los botones de la esquina.

Parámetros:

En principio se cambian desde Unity. No he tenido tiempo a implementar una interfaz con sliders ni campos para modificarlos ingame.

Se puede cambiar el recorrido de una curva activando y desactivando Ease In Out Activated, para que sea a velocidad constante si está desactivado (la cámara lo tiene desactivado)

Las fracciones de Ease In y Ease Out se puede modificar entre [0,1] y además no permite superar un máximo de 1 entre la suma de los dos.

También se puede activar o desactivar la rotación y cambiar el tiempo de animación, que es el que tarda en recorrer la curva entera.

Eso está encapsulado en un script llamado "BezierMovable", el cual se debe asignar a cualquier objeto que se quiero dejar recorriendo una curva.

Por otra parte la curva tiene un script "Bezier", donde se puede activar o desactivar la visualización de las lineas y los puntos de control.

Son prefabs y se pueden reutilizar fácilmente independientemente del objeto al que se los asigne.

PRACTICA 3

He creado 3 Escenas:
- Una para la curva de Bezier y el moviimiento del avión a través de ella de la Práctica 1
- Otra para ver las animaciones creadas en esta Práctica 2
- Y una Escena Principal en la que voy a integrar todas las técnicas en siguientes Prácticas.

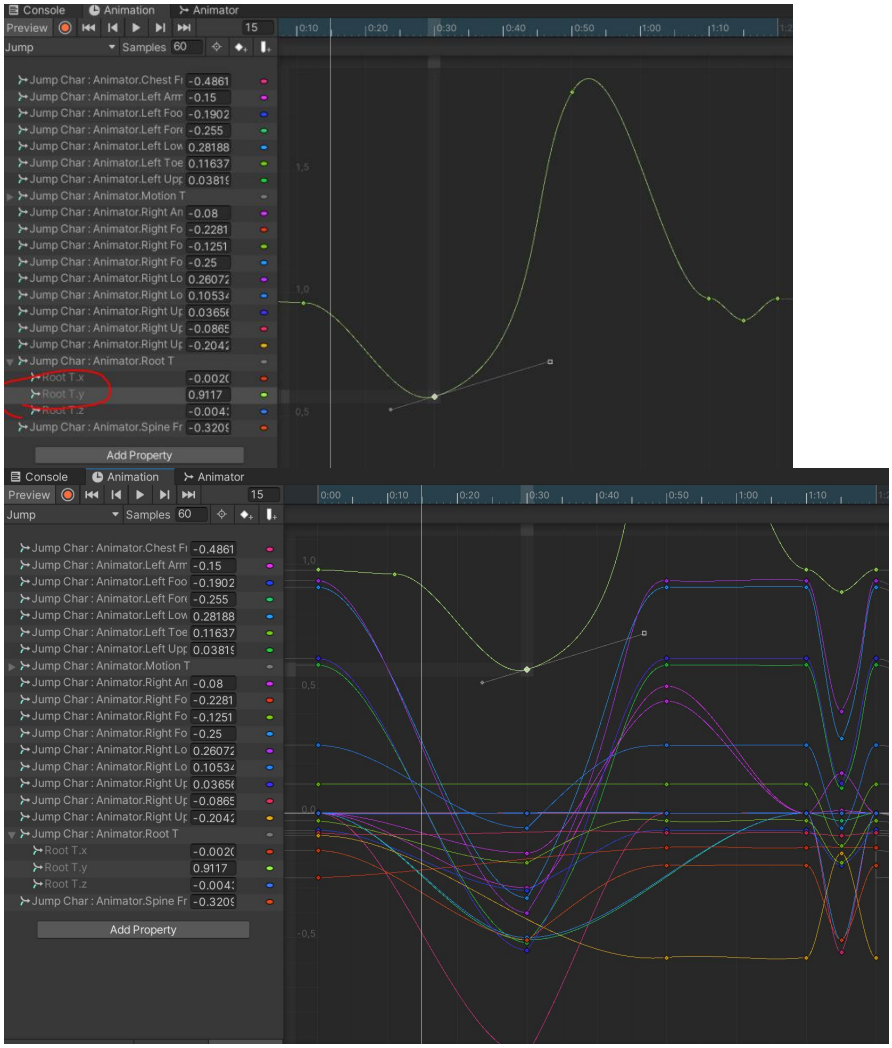
Animaciones

Andar:

La traslación no la hace la animación, sino que la controlo a través de un Script en la Escena Principal donde empieza andando hacia delante por 3 segundos antes de lanzar el avión.

Saltar:

La he creado de la misma manera que la de Andar, colocando keyframes a mano poco a poco ajustando que quede bien el movimiento. Excepto el movimmiento del modelo vertical, que se aun haciendolo con keyframes he ajustado la velocidad con una curva de forma que no necesita tantos keyframes.



Lanzar:

He hecho una animación para lanzar el avión. Simplemente empieza el brazo en alto y he ido ajustado la rotación del brazo con la curva hasta que me ha gustado el resultado.

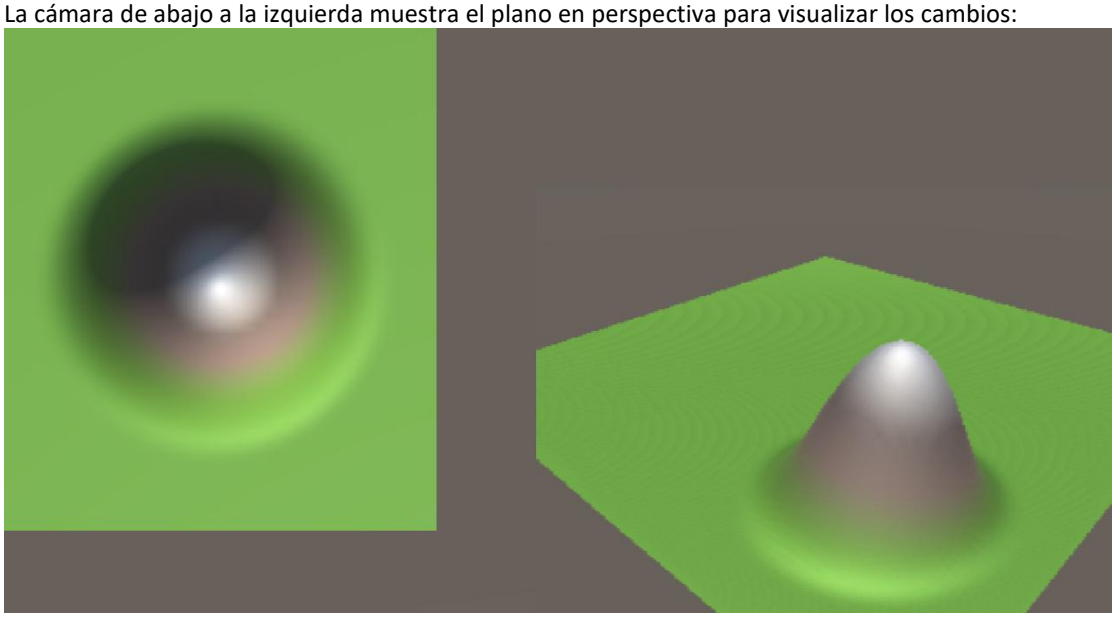
Layers:

He utilizado Layers en mi Animation Controller para la Escena Principal, que permiten superponer una animación sobre otra, por medio de una máscara de avatar, en la que bloqueo las piernas para juntar la animación de lanzamiento con la de andar, para poder andar con el avión en la mano antes de lanzarlo.

PRACTICA 4

Deformación de Objetos: Pick & Pull

Si clickas con el ratón sobre el plano se produce una deformación, hacia arriba con click izquierdo, y hacia abajo con click derecho.
La cámara de abajo a la izquierda muestra el plano en perspectiva para visualizar los cambios:



Creación de la Malla:

Creamos la malla que vamos a modificar con la técnica Pick & Pull, de forma rectangular con una anchura y altura específica, y con altura de cada vértice a 0:

```
public static class MeshGenerator
{
    public static MeshData generateMesh(int width, int height)
    {
        MeshData data = new MeshData(width, height);

        // La malla la creamos centrada en 0:
        float initX = (width - 1) / -2f;
        float initY = (height - 1) / -2f;

        int vertIndex = 0;
        for (int y = 0; y < height; y++)
        for (int x = 0; x < width; x++)
        {
            data.vertices[vertIndex] = new Vector3(x: initX + x, y: 0, z: initY + y);
            data.uvs[vertIndex] = new Vector2(x: (float)x / width, y: (float)y / height);

            // Ignorando la ultima fila y columna de vertices, añadimos los triangulos
            if (x < width - 1 && y < height - 1)
            {
                data.AddTriangle(a: vertIndex, b: vertIndex + width, c: vertIndex + width + 1);
                data.AddTriangle(a: vertIndex + width + 1, b: vertIndex + 1, c: vertIndex);
            }

            vertIndex++;
        }

        return data;
    }
```

Creamos la Mesh con los datos almacenados:

```
public Mesh CreateMesh()
{
    Mesh mesh = new Mesh
    {
        vertices = vertices,
        triangles = triangles,
        uv = uvs,
    };

    mesh.RecalculateNormals();

    return mesh;
}
```

Pick:

Para hacer el Pick & Pull primero necesitamos un vértice al que aplicarlo, que vamos a seleccionar con el ratón, obteniendo de la malla el vértice más cercanos a la posición de un punto donde hemos golpeado haciendo Raycast con el cursor:

```
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out RaycastHit hit, maxDistance: Mathf.Infinity, LayerMask.NameToLayer("DeformationMesh")))
{
    // Punto relativo al objeto
    Vector3 hitPoint = hit.point - meshFilter.transform.position;

    // Hacemos un Pick&Pull (si es acumulativo se actualiza
    MeshData newData = data.PickAndPull(
        pos: hitPoint, deformation: deformation * deformationStrength, deformationRadius,
        deformationScaleFactor, maxHeight, minHeight
    );
```

Pull:

Con **getNearVertex(pos)** cogemos el índice del vértice cuya posición es la mas cercana a *pos.*
Este vértice será la raíz de la deformación.
Sus vecinos recibirán un efecto de deformación directamente **proporcional** a la **distancia** respecto a este **vértice Raíz.**
En mi caso he implementado una versión que utiliza la **distancia** solamente, y no cuántos vecinos hay entre el vértice y la raíz, lo cual necesita un método más complejo para encontrar vecinos, que en una situación ideal la haría con una Estructura de Datos Espacial como la Winged Edge.
La distancia máxima a la que llega el efecto está limitada por "**effectRadius**", y el factor k con el que modifica la función de deformación es el "**ScaleFactor**".
"**Deformation**" es el vector hacia donde se desplazan los vértices.

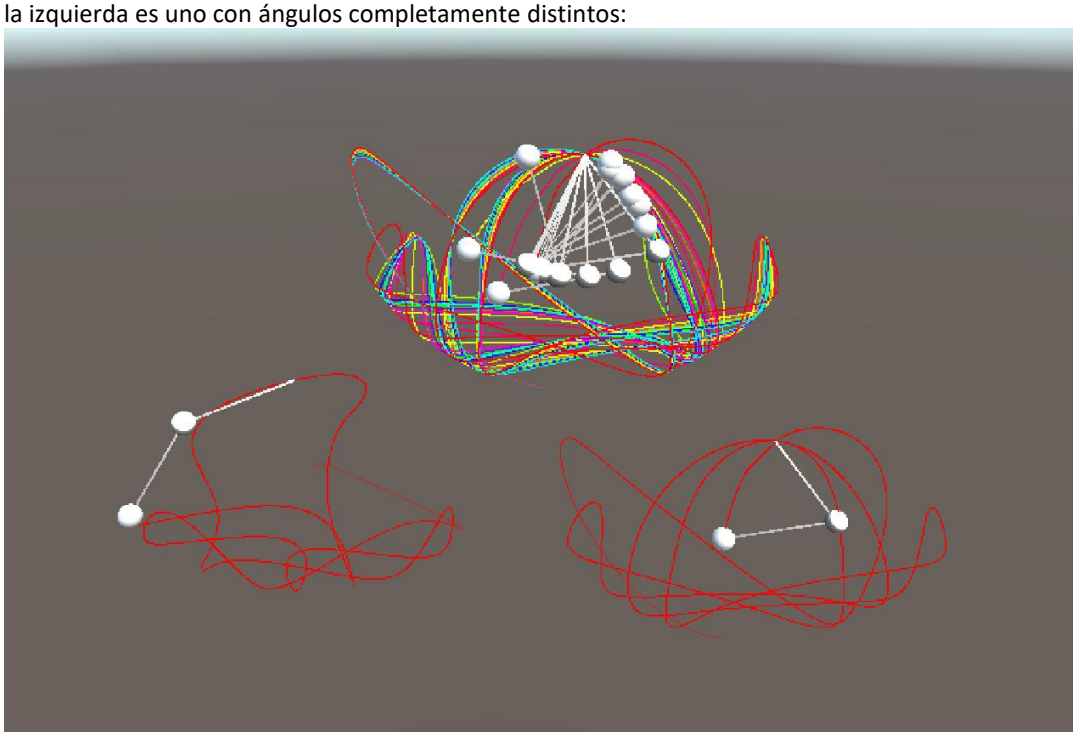
```
public MeshData PickAndPull(Vector3 pos, Vector3 deformation, float effectRadius, float scaleFactor, float max, float min)
{
    MeshData newData = new MeshData(this);

    // Buscamos el Vertice mas cercano a la posicion, va a ser el VERTICE RAIZ
    int rootVertIndex = getNearVertex(pos);

    Vector3 rootV = vertices[rootVertIndex];
    for (int i = 0; i < vertices.Length; i++)
    {
        // Distancia respecto al Vertice RAIZ
        float distance = (vertices[i] - rootV).sqrMagnitude;

        // Si entra dentro del radio de efecto recibe una fuerza
        if (distance < effectRadius)
        {
            float force = Mathf.Pow( f: 1f - distance / (effectRadius + 1), p: 1 + Mathf.Abs(scaleFactor) );

            newData.vertices[i] += deformation * force;

            // Limitamos la altura a un maximo y minimo
            newData.vertices[i].y = Mathf.Clamp( value: newData.vertices[i].y, min, max);
        }
    }

    return newData;
}
```

Los valores min y max son para limitar que no se pase de una altura específica, pero no es necesario a no ser que haga que el efecto sea acumulativo con el bool “**Accumulative**”, que tiene un efecto similar a la herramienta de modificación de terrenos:

```
if (accumulative)
    data = newData;

UpdateMesh(newData);
```

Animación Procedural

He implementado el Doble Péndulo, cuyo movimiento se basa en la aceleración angular de cada uno de los 2 extremos.
El de arriba son varios péndulos dobles superpuestos que comienzan con la misma posición, salvo el ángulo del extremo inferior, que difiere 0,5º de uno al siguiente, y están representados por un color de trazada distinto.
El de abajo a la derecha es el mismo péndulo con mismos ángulos para ver el resultado aparte, y el de la izquierda es uno con ángulos completamente distintos:



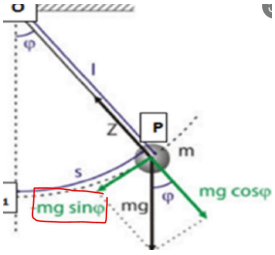
El Péndulo:

He implementado primero un péndulo básico antes del doble.
La animación tiene una velocidad asociada, un ángulo respecto con su punto de sujeción, una longitud de cuerda, y los componentes angulares de la velocidad y aceleración.

```
public class Pendulum : MonoBehaviour
{
    public float animationSpeed = .01f;

    public float angle = 60;
    public float ropeLength = 1;

    public float angularVel = 0;
    public float angularForce = 1f;

    void Awake()
    {
        Vector3 ropeDir = transform.position - transform.parent.position;
        ropeLength = ropeDir.magnitude;
        angle = Mathf.Acos(f: Vector3.Dot(lhs: ropeDir.normalized, rhs: new Vector3(x: 0, ropeDir.y, z: 0).normalized));
    }
```

El cálculo del Péndulo sencillo se realiza con la ecuación:

$$-mg\sin(\varphi)$$



Luego esta fuerza se le añade a la velocidad angular.
Y la velocidad se añade al ángulo.
Y la posición del péndulo será en
X = Sin(φ) y **Y = Cos(φ)** multiplicado por la **longitud de la cuerda:**

```
void FixedUpdate()
{
    angularForce = getSinglePendulumForce();

    angularVel += angularForce;
    angle += angularVel;


    transform.localPosition = new Vector3(x: Mathf.Sin(f: Mathf.Deg2Rad * angle), y: -Mathf.Cos(f: Mathf.Deg2Rad * angle), z: 0);
    transform.localPosition *= ropeLength;
}

float getSinglePendulumForce()
{
    Rigidbody rb = GetComponent<Rigidbody>();

    float m = rb.mass;
    float g = Physics.gravity.magnitude;

    return -m * g * Mathf.Sin(f: Mathf.Deg2Rad * angle) * Time.fixedDeltaTime * AnimationSpeed;
}
```

Para crear un doble péndulo debemos aplicar la ecuación:

$$\theta_1'' = \frac{-g\,(2\,m_1 + m_2)\sin\theta_1 - m_2\,g\,\sin(\theta_1 - 2\,\theta_2) - 2\sin(\theta_1 - \theta_2)\,m_2\,(\theta_2'^2\,L_2 + \theta_1'^2\,L_1\cos(\theta_1 - \theta_2))}{L_1\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

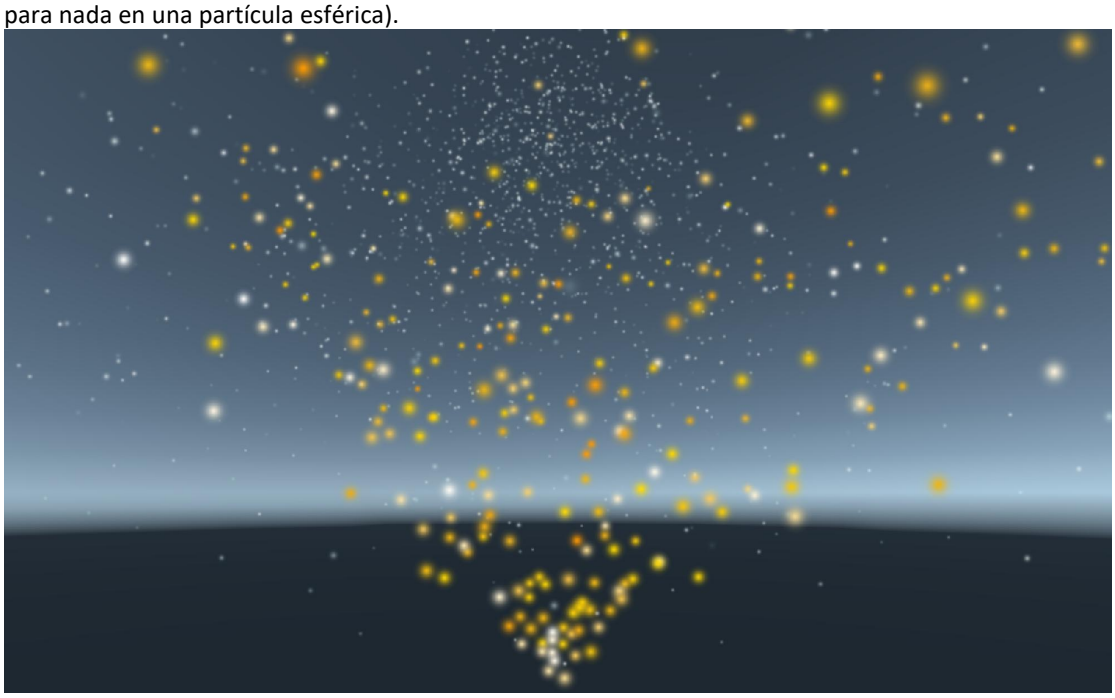
$$\theta_2'' = \frac{2\sin(\theta_1 - \theta_2)\,(\theta_1'^2\,L_1\,(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \theta_2'^2\,L_2\,m_2\cos(\theta_1 - \theta_2))}{L_2\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

Siendo θ'' la **aceleración angular** de cada péndulo.

```
float g = Physics.gravity.magnitude * Time.fixedDeltaTime * Time.fixedDeltaTime;
float m1 = p1.GetComponent<Rigidbody>().mass;
float m2 = p2.GetComponent<Rigidbody>().mass;

float l1 = p1.ropeLength;
float l2 = p2.ropeLength;

float a1 = Mathf.Deg2Rad * p1.angle;
float a2 = Mathf.Deg2Rad * p2.angle;
float v1 = p1.angularVel;
float v2 = p2.angularVel;

p1.angularForce = -g * (2 * m1 + m2) * Mathf.Sin(a1)
                  - m2 * g * Mathf.Sin(f: a1 - 2 * a2)
                  - 2 * Mathf.Sin(f: a1 - a2)
                  * m2 * (v2 * v2 * l2 + v1 * v1 * l1 * Mathf.Cos(f: a1 - a2))
                  / (l1 * (2 * m1 + m2 - m2 * Mathf.Cos(f: 2 * a1 - 2 * a2)));
p2.angularForce = 2 * Mathf.Sin(f: a1 - a2)
                    * (v1 * v1 * l1 * (m1 + m2) + g * (m1 + m2) * Mathf.Cos(a1)
                    + v2 * v2 * l2 * m2 * Mathf.Cos(f: a1 - a2))
                    / (l2 * (2 * m1 + m2 - m2 * Mathf.Cos(f: 2 * a1 - 2 * a2)));

p1.angularForce *= Time.fixedDeltaTime * animationSpeed;
p2.angularForce *= Time.fixedDeltaTime * animationSpeed;

p1.angularForce = Mathf.Clamp(value: p1.angularForce, min: -1, max: 1);
p2.angularForce = Mathf.Clamp(value: p2.angularForce, min: -1, max: 1);


p1.angularVel += p1.angularForce;
p2.angularVel += p2.angularForce;

// Limit Velocity:
p1.angularVel = Mathf.Clamp(value: p1.angularVel, minAngularVelocity, maxAngularVelocity);
p2.angularVel = Mathf.Clamp(value: p2.angularVel, minAngularVelocity, maxAngularVelocity);

p1.angle += p1.angularVel;
p2.angle += p2.angularVel;
```

Sistema de Partículas

He creado para empezar un sistema de partículas básico como base, pero más tarde, como iteración siguiente, he implementado un efecto de ***Nieve*** (lo llamo yo). Se trata de partículas que cada frame modifican su dirección de movimiento suavemente utilizando **Ruido de Perlin** bidimensional.
Una de las dimensiones para la **rotación en X** y otra para la **rotación en Z** (la rotación en Y no sirve para nada en una partícula esférica).



He tomado como referencia el sistema de partículas de Unity para usar los parámetros básicos:

```
public class MyParticleSystem : Spawner
{
    [Range(0, 10)] public float startSpd = 5f;
    [Range(0.1f, 100)] public float startLifetime = 2;
    [Range(0.1f, 10)] public float startSize = 1;
    public Gradient startColor;

    [Range(0, 10)] public float delay = 0;
    [Range(0, 10)] public float duration = 5;
    public bool looping = true;

    // Particulas / Segundo
    [Range(0, 1000)] public float rateEmission = 5;

    private float timePlaying = 0;

    // Area de Spawn
    [Range(0, 90)]public float coneAngle;
    public float coneRadius = 1;

    // Aleatoriedad
    public bool randomSpd;
    public bool randomOrientation;
    public bool randomLifeTime;
    public bool randomSize;
    public bool randomColor;

    public bool snowEffect = false;
    public bool useGravity = false;

    void Start()
    {
        StartCoroutine(routine: Play());
    }
```

Hereda de **Spawner**, porque he reutilizado un script de un proyecto de otra asignatura, cuya función en resumen es llevar un Pool del objeto prefab que le pases, que irá creciendo conforme necesite spawnear más, y que realiza la creación y eliminación de objetos de forma más eficiente por medio de una Pila, de la que los saca al spawnear y los activa, y los vuelve a meter y desactivar cuando se quieren destruir, para evitar la carga computacional y de memoria que supone usar Destroy() o Instantiate().

El funcionamiento del sistema se realiza mediante una Corutina *Play()*, donde hace las esperas que necesite, entre partículas o antes de empezar a emitir con un delay:

```
public IEnumerator Play()
{
    yield return new WaitForSeconds(delay);

    while (looping || timePlaying < duration)
    {
        yield return new WaitForSeconds(1 / rateEmission);

        SpawnParticle();

        timePlaying += Time.deltaTime;
    }
    Debug.Log(message: "Particle System Finished");
}
```

La función SpawnParticle() es la que realiza todo el proceso de posicionar y aplicar los atributos iniciales a cada partícula:

```
Particle SpawnParticle()
{
    Vector3 position = transform.position + new Vector3(
        x: Mathf.Lerp(a: -coneRadius, b: coneRadius, t: Random.value),
        y: 0,
        z: Mathf.Lerp(a: -coneRadius, b: coneRadius, t: Random.value)
        );

    Particle particle = Spawn(position).GetComponent<Particle>();

    // Set Variables
    particle.speed = randomSpd ? Mathf.Lerp(a: startSpd - 1, b: startSpd + 1, t: Random.value) : startSpd;
    particle.lifeTime = randomLifeTime ? Mathf.Lerp(a: startLifetime - 1, b: startLifetime + 1, t: Random.value) : startLifetime;
    particle.size = randomSize ? Mathf.Lerp(a: startSize - startSize/10, b: startSize + startSize / 10, t: Random.value) : startSize;
    particle.color = randomColor ? startColor.Evaluate(Random.value) : startColor.Evaluate(time: 0);
    particle.orientation = randomOrientation
        ? Quaternion.Lerp(
            a: Quaternion.Euler(x: 0, y: 0, z: coneAngle),
            b: Quaternion.Euler(x: 0, y: 0, z: -coneAngle),
            Random.value
        ) * Quaternion.Lerp(
            a: Quaternion.Euler(x: coneAngle, y: 0, z: 0),
            b: Quaternion.Euler(x: -coneAngle, y: 0, z: 0),
            Random.value
        ) * transform.up
        : transform.up;

    particle.gravity = useGravity;

    particle.snow = snowEffect;

    particle.Initialize();
```

Y la partícula se autogestiona sus atributos desde Initialize():

```
public void Initialize()
{
    Rigidbody rb = GetComponent<Rigidbody>();
    MeshRenderer meshRenderer = GetComponent<MeshRenderer>();

    rb.velocity = orientation * speed;

    rb.useGravity = gravity;

    transform.localScale = Vector3.one * size;

    meshRenderer.material.color = color;

    timeAlive = 0;
}
```

La Partícula:

Tiene todos los atributos individuales que necesita, como la velocidad, el tiempo de vida, el tamaño, el color, la orientación.
Y además dos estados que se pueden activar. Uno es “gravity” que activa la gravedad de la partícula, y el otro es “snow” que produce el efecto de nieve mencionado antes.

Además, hereda de Spawneable, que es la clase abstracta que gestiona su creación y destrucción dentro de un Spawner.

```
public class Particle : Spawneable
{
    public float speed = 1f;
    public float lifeTime = 2;
    public float size = 1;
    public Color color = Color.white;

    public Vector3 orientation = Vector3.up;

    private float timeAlive = 0;

    public bool snow = false;
    public bool gravity = false;

    protected override void OnEnable()
    {
    }

    protected override void OnDisable()
    {
    }

    void Update()
    {
        if (timeAlive < lifeTime)
        {
            timeAlive += Time.deltaTime;

            if (snow)
                snowEffect(maxAngle: 80);
        }
        else
        {
            base.Destroy();
        }
    }
```

EFECTO NIEVE

Si está activado, cada frame **rota el vector orientación** de la partícula para cambiar su trayectoria. Esta rotación se calcula con **Perlin Noise**, que utiliza como **coordenada** dentro del mapa de ruido el **tiempo de vida** de la partícula, para que a medida que la partícula va aumentando su tiempo de vida, los valores del mapa de ruido cambien de forma suave con **coherencia temporal**. Además, cojo dos valores unidimensionales, uno en el eje de la X en Y=0 para la rotación en el eje X de la orientación, y otro en el eje Y en X=0 para la rotación en el eje Z de la orientación:

```
private void snowEffect(float maxAngle)
{
    // Utilizando Perlin Noise con el valor de su tiempo de vida conseguimos valores similares
    // Por lo que si interpolamos entre la maxima y minima rotacion, en ambos ejes,
    // Conseguimos rotaciones similares a cada frame interpolando [-.5, .5]

    Quaternion xRotation = Quaternion.Slerp(
        a: Quaternion.Euler(x: maxAngle, y: 0, z: 0),
        b: Quaternion.Euler(x: -maxAngle, y: 0, z: 0),
        t: Mathf.PerlinNoise(x: timeAlive, y: 0)
        );
    Quaternion zRotation = Quaternion.Slerp(
        a: Quaternion.Euler(x: 0, y: 0, z: maxAngle),
        b: Quaternion.Euler(x: 0, y: 0, z: -maxAngle),
        t: Mathf.PerlinNoise(x: 0, y: timeAlive)
        );

    Rigidbody rb = GetComponent<Rigidbody>();
    rb.velocity = xRotation * zRotation * orientation * speed;
}
```