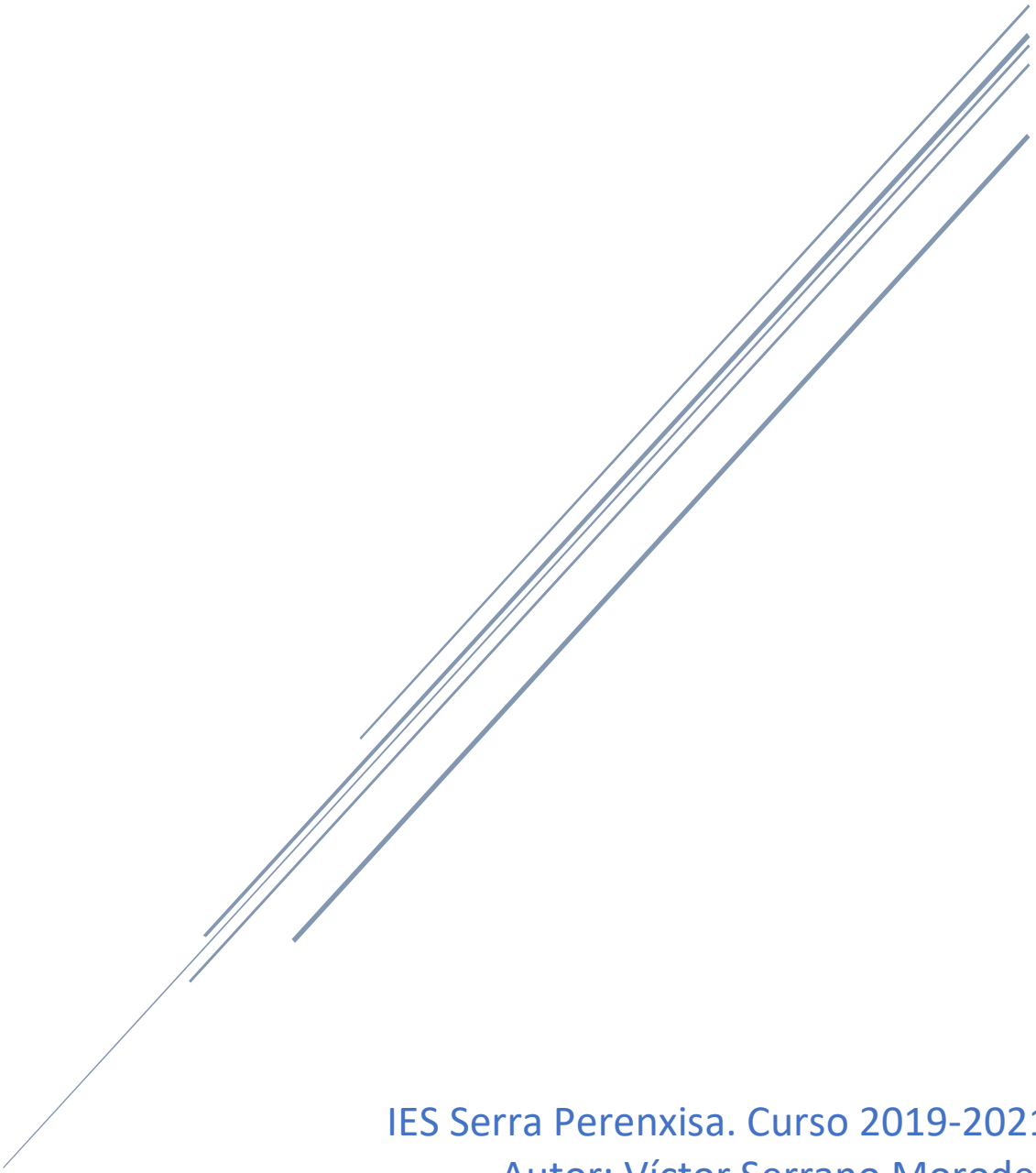
RENOISE

PROTOTIPO DE VIDEOJUEGO EN UN ESPACIO CON COMPORTAMIENTO NO EUCLIDIANO EN UNITY



IES Serra Perenxisa. Curso 2019-2021
Autor: Víctor Serrano Moroder
Tutor: Raúl Fuentes Ferrer
Memoria de proyecto de Desarrollo de Aplicaciones Multiplataforma

Agradecimientos

A mi familia por aguantarme todos estos años.

A mis amigos por animarme cuando me frustraba.

A mi perrita Lina, siempre te recordaré.

Renoise

Un prototipo de videojuego en un Espacio con comportamiento no euclidiano en Unity

2ºDAM – C.F.G.S

IES SERRA PERENXISA

Índice de Ilustraciones

1. Introducción

1.1 Contexto y justificación del trabajo

El mundo del videojuego es muy variopinto, en este ecosistema coexisten miles de títulos y géneros diferentes, con objetivos, diseños y conceptos muy diferentes. Desde el género simulador, donde se pretende buscar lo más parecido a la realidad, hasta el mmorpg, o plataformas. Es un mundo que destaca por su originalidad y abundancia de ideas novedosas, y es que no hacen falta mecánicas complicadas ni costosas, lo más importante es la idea.

Para llegar a estas ideas hay quienes rompen con lo estándar, o cambian conceptos o sentidos tan obvios que no nos los planteamos en un primer lugar como objeto para una mecánica. ¿Qué pasaría si cambiáramos de dirección la gravedad? ¿Y si la luz proyecta oscuridad y la oscuridad proyecta luz? ¿Qué ocurre si ignoras al narrador de una historia y tienes más opciones? Estas mecánicas personalmente siempre han sido objeto de fascinación y de asombro para mí, porque hacen uso de pequeños cambios en algo ya asumido por todos para crear experiencias y finalmente un gameplay definido.

En este proyecto se va a profundizar en un tipo de mecánicas poco conocidas, son las que juegan con la geometría y el espacio. Dan lugar a que el jugador se plantee rutas y que piense de forma muy diferente a la que está acostumbrado y que salga de su zona de confort teniéndose que cuestionar todo.

Hay un género en especial que, además de eso, juegan y rompen algo tan familiar y a la vez es algo diferente. Estos son los juegos de Geometría No-Euclidiana.

La idea de este trabajo viene de la mera curiosidad de entender cómo funcionan, diseccionar sus mecánicas y comprender sus componentes. Por ello se ha decidido por un prototipo, un prototipo es una demostración a nivel bajo de las ideas y mecánicas que podrían formar parte de un videojuego. Aquí hablamos de gameplay a nivel muy básico, pero con una base robusta que puedan formar parte de un núcleo de mecánicas.

1.2 Objetivos del proyecto

El objetivo del proyecto es diseccionar, analizar y explicar algunas de las mecánicas utilizadas por los videojuegos de Geometría No-Euclidiana para presentarlas en un prototipo donde se den ejemplos de su uso. Por lo tanto, podemos definir este trabajo de fin de grado como un proyecto de investigación sobre este tipo de mecánicas.

Esto además una vez terminado se va a subir a github para ser parte de un porfolio y su acceso será público de forma que si alguien quisiera usar estas mecánicas en sus juegos aquí encontraría una guía de cómo hacerlo con ejemplos y código explicado.

Al proyecto se le ha dado una estructura modular, y las mecánicas trabajan de forma independiente entre si lo cual las hace fáciles de entender y de explicar.

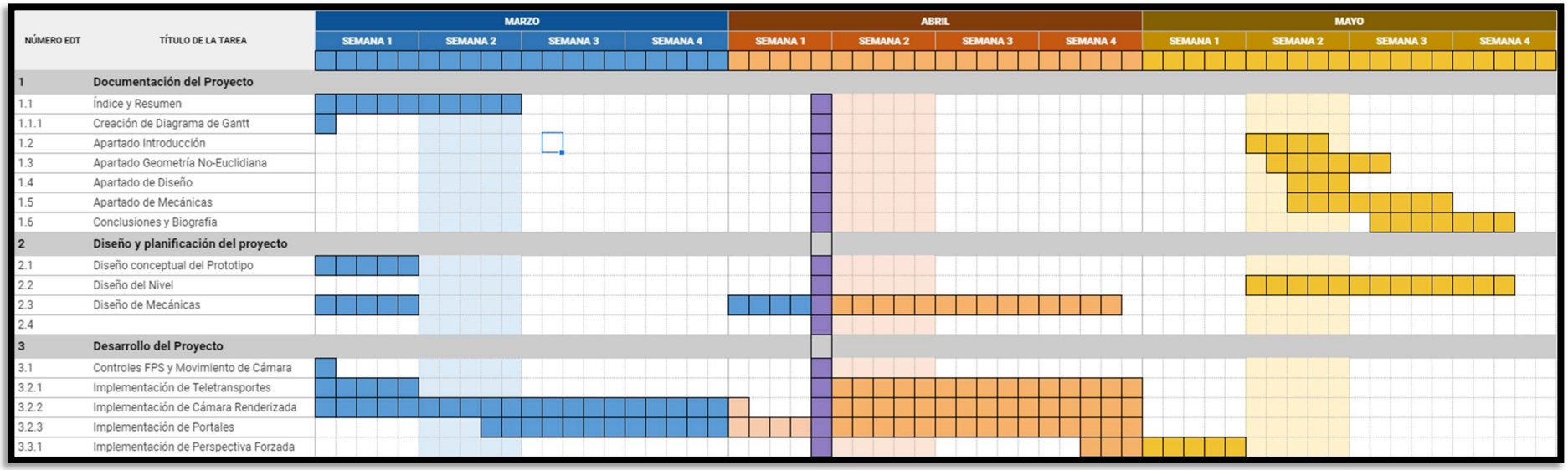
El prototipo se compone de 3 módulos diferentes:

1. Demostración de Portales
2. Demostración de Perspectiva forzada
3. Prototipo de Nivel

Los módulos de demostración serán expuestos en la presentación para mostrar el funcionamiento de las mecánicas, por ofrecer una visión más clara de cómo funcionan. Mientras que el prototipo de nivel presenta ideas de cómo podrían formarse estos niveles y cómo esconder su verdadero funcionamiento.

1.3 Planificación del Trabajo

Este es el diagrama de Gantt donde se ve el desarrollo del prototipo a lo largo de los 3 meses, fíjese que en el mes de abril hay una línea morada, esta indica cuando se realizó la reescritura y la reestructuración del proyecto, debido a razones que se mencionarán más adelante.



1.4 Enfoque y Método de Trabajo

Como se ha mencionado antes el proyecto se va a estructurar de forma modular. Para conseguirlo se van a usar técnicas de gestión de proyectos 'Agile Scrum'. Estas utilizan un enfoque incremental e iterativo para gestionar proyectos. Las unidades de tiempo y de trabajo se llaman sprints y duran unas pocas semanas para las que se fijan una serie de objetivos y metas a cumplir. Tras cada sprint hay una fase de revisión y corrección que una vez acabada iniciará el siguiente sprint.

Esta metodología es perfecta para proyectos de desarrollo como este ya que se trabaja en entornos y en campos que presentan cambios constantes y donde, al ser un desarrollo incremental, cada fase se asienta sobre lo que ya está hecho. De forma que se sigue un patrón de desarrollo secuencial con solapamiento para asegurar que las diferentes fases se integren bien en el programa.

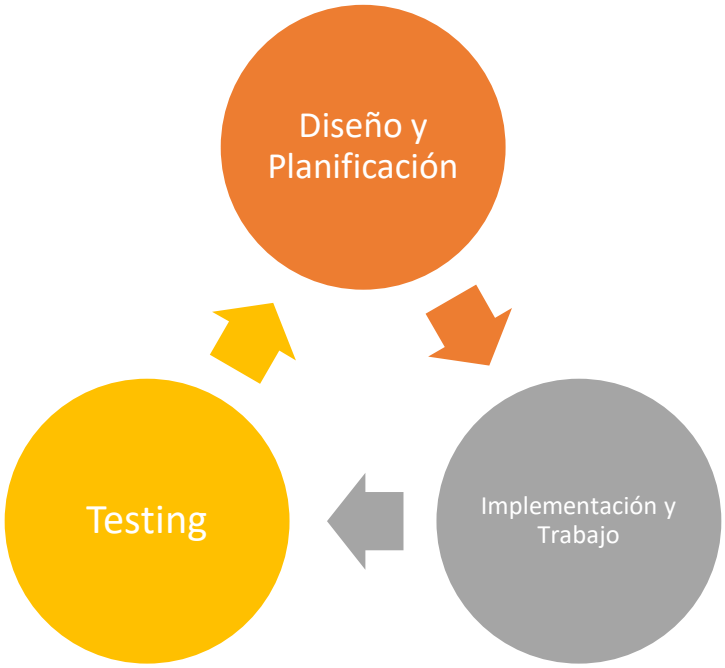


Ilustración 1 - Diagrama 'Agile Scrum'

Gracias a esto el progreso del proyecto avanza de forma gradual e incremental, y se pueden revisar las diferentes partes e ir solucionando problemas y bugs que vayan desarrollándose por el camino. Esto ha sido muy útil en este proyecto ya que a medida que se iba progresando se ha necesitado dedicar sprints a la corrección de errores y para perfilar detalles y asperezas que han ido surgiendo.

2. Geometría No Euclidiana

2.1 Conceptos Básicos y Postulados de Euclides

Se denomina Geometría No Euclidiana a aquel sistema de geometría que viole uno o más de los postulados de Euclides establecidos en su obra “Los Elementos”.

1º Postulado: Una línea recta puede ser trazada al unir 2 puntos.

2º Postulado: Cualquier recta puede ser extendida indefinidamente en una línea recta.

3º Postulado: Se puede trazar una circunferencia dados un centro y un radio cualquiera

4º Postulado: Todos los ángulos rectos son iguales entre si

5º Postulado: Si una línea recta corta a otras dos, de tal manera que la suma de los dos ángulos interiores del mismo lado sea menor que dos rectos, las otras dos rectas se cortan, al prolongarlas, por el lado en el que están los ángulos menores que dos rectos.

A efectos prácticos podemos reformular estos postulados para crear 3 reglas generales:

1º Regla: Una línea recta es la distancia más corta entre dos puntos.

2º Regla: La distancia que separa dos rectas paralelas siempre es y será constante entre ambas.

3º Regla: La suma de los ángulos internos que forman un triángulo siempre serán 180º.

Esto crea diferentes tipos de geometrías dependiendo qué postulados se cumplan y del ángulo de curvatura de su espacio.

El primero, el espacio euclídeo. Se define como aquél que cumple todos los postulados de Euclides y, por ende, tiene curvatura 0.

El segundo, el espacio hiperbólico. Satisface los cuatro primeros postulados de Euclides y tiene una curvatura positiva. En este tipo de espacios la suma de los ángulos de cualquier triángulo sea menor que 180º.

El, tercero, el espacio elíptico o esférico. Al igual que el hiperbólico, sólo satisface los 4 primeros postulados de Euclides, pero tiene una curvatura negativa. Esto significa que los ángulos de cualquier triángulo son mayores de 180º.

En este proyecto se va a trabajar sobre un cuarto tipo de espacio, denominado “Espacio con Comportamiento No Euclidiano”. Es un espacio regular, aparentemente euclidiano. Pero exhibe, en momentos determinados, distorsiones y deformaciones geométricas, que violan uno o más de los postulados de Euclides.

Por lo tanto, podemos introducir comportamientos no intuitivos en espacios familiares, nos referiremos a estos como “mecánicas” con las que el jugador tendrá que trabajar para avanzar.

2.2 Mundos de Riemann

Hasta el siglo XIX la única geometría aceptada era la Euclídea, reforzada por lo bien que trabajaba con el modelo físico de caja de Newton, capaz de describir y confirmar con gran precisión las observaciones del universo. Sin embargo, a lo largo de los años fueron apareciendo otras propuestas de geometrías, una de las que mejor describe la realidad fue la concebida por Bernhard Riemann.

Riemann cuestionó la concepción newtoniana, que afirmaba que el universo era una caja infinita. El planteamiento de Riemann lo llevó a entender que, dependiendo de la perspectiva, una esfera podría ser un modelo completamente válido y que encajaría con la descripción de un universo finito, pero ilimitado (finito porque tiene cierto volumen, ilimitado porque no tiene bordes). Estas observaciones sobre la perspectiva ayudarían a fundamentar las bases y las ideas de la Teoría de la relatividad.

En geometría la esfera es matemáticamente un objeto de 2 dimensiones, represable en 3 dimensiones. Podemos describir la posición de cualquier elemento sobre ésta con tan sólo 2 dimensiones, latitud y longitud. Si nos situáramos en la superficie de una esfera lo suficientemente grande, nuestra perspectiva nos diría que estamos sobre un mundo plano sin bordes. Estas observaciones son, actualmente, algunas de las descripciones más acertadas del universo en el que vivimos. Esta imagen del universo se conoce actualmente como hiperesfera o esfera tridimensional.

El estudio que realizó Riemann de las variedades diferenciales (variedad como objeto geométrico que generaliza la noción intuitiva de una curva y superficie sobre una dimensión o cuerpos diversos), se conoce como Geometría de Riemann. Dando lugar y definición a los tipos de espacios no euclidianos mencionados anteriormente. Por eso mismo se les considera variaciones o casos particulares de Geometrías Riemannianas, o Mundos de Riemann. Algunos artistas surrealistas del siglo XX tomaron inspiración de las oportunidades que brindaban estos mundos, que desafiaban las leyes de la lógica, entre ellos Picasso y Dalí.

El artista más reconocido por el uso de comportamientos No Euclidianos en su obra es sin duda Maurits Cornelis Escher, o M.C Escher.

La obra de Escher juega con la perspectiva y el espacio para ofrecer objetos imposibles, como pueden ser las escaleras infinitas en su obra “Ascending And Descending”.

Estas escaleras infinitas se conocen como “Escalera de Penrose”. Se trata de una ilusión óptica descrita por los matemáticos Lionel Penrose y su hijo Roger Penrose, publicada junto a otros objetos imposibles como el “Triángulo de Penrose”. Ambas pertenecen a una serie de formas imaginarias llamadas, “objetos imposibles”, que presentan comportamientos no euclidianos.

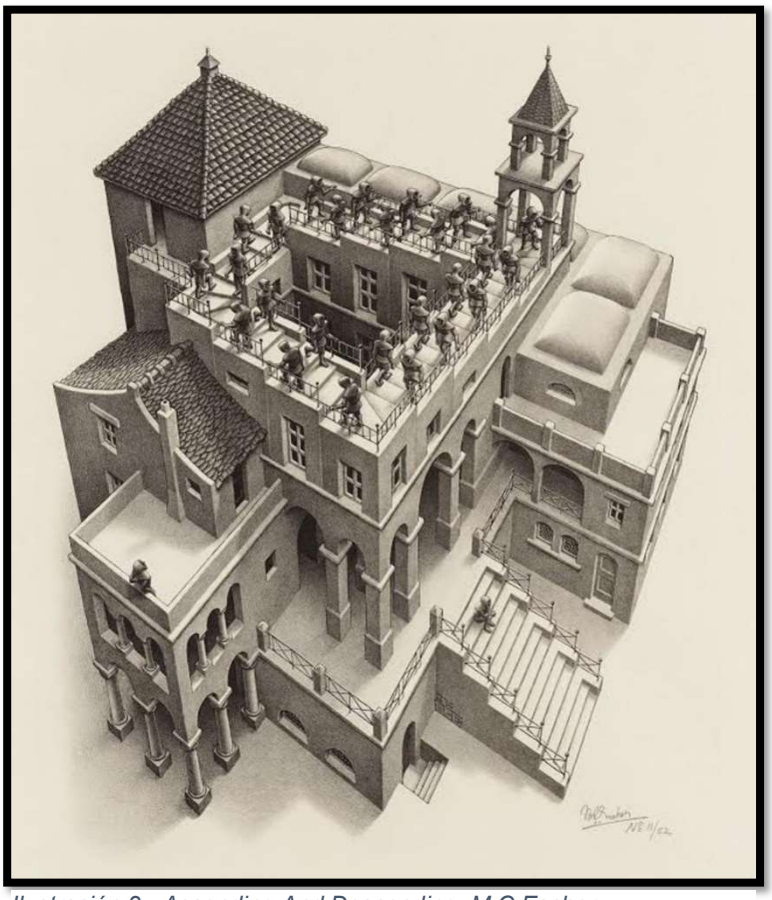


Ilustración 2 - Ascending And Descending. M.C.Escher

M.C Escher trabajó también con geometría hiperbólica en su serie “Circle Limit”. Fue inspirado y aconsejado por el geométra Harold Scott MacDonald Coxeter, que extendió y ayudó a definir la obra de Riemann.

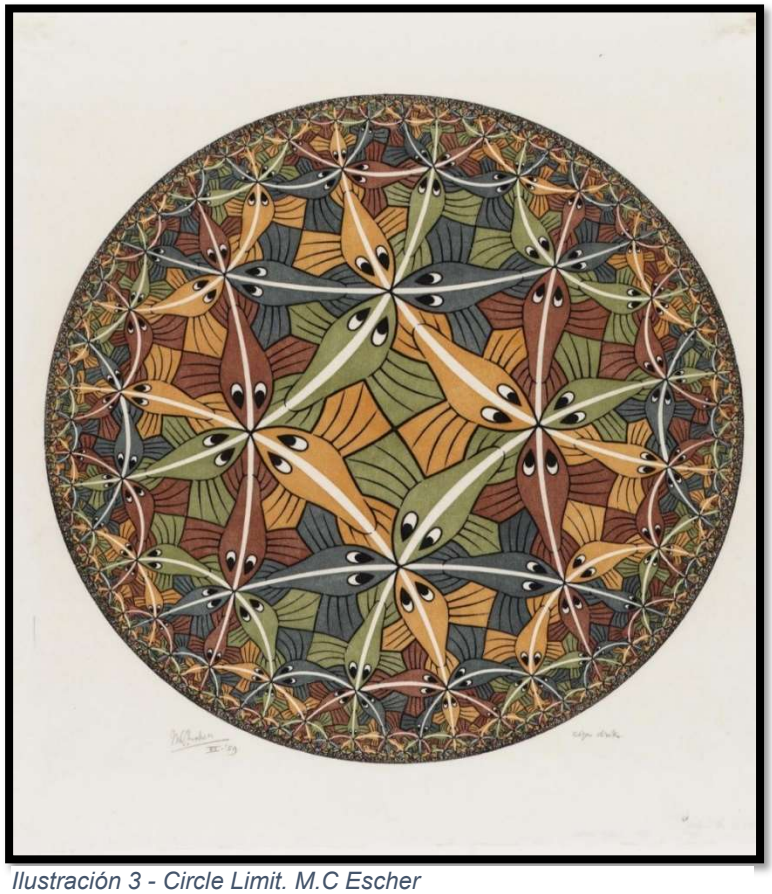


Ilustración 3 - Circle Limit. M.C Escher

Estas obras han influenciado también al mundo del cine. Algunos ejemplos se pueden encontrar, más recientemente, en la película del británico Christopher Nolan, Origen (Inception 2010). En una escena aparece la Escalera de Penrose, que juega un papel clave como catalizador de la narrativa visual. Se explican ejemplos de arquitectura paradójica para engañar a la mente, como si se tratara de un sueño imposible.

Engaño es la palabra clave aquí para replicar “Comportamientos No Euclidianos”. En un espacio normal tenemos que hacer uso de engaños, de perspectivas inteligentes y jugar con la profundidad. En definitiva, son ilusiones.

El mundo de los videojuegos también ha sido fuertemente inspirado por estos mundos, así ha surgido su propio género, Juegos de Geometría No Euclidiana. Este proyecto considerará que tanto los videojuegos que usan Espacios Geométricos No Euclidianos Reales, como los que presentan comportamientos No Euclidianos, son Juegos de Geometría No Euclidiana. Aclarado este punto podemos pasar al mayor referente del género, Antichamber.

Antichamber es un videojuego de acertijos y exploración en primera persona, donde el jugador ha de explorar un mundo. A medida que va avanzando, sus expectativas de cómo funciona el juego se van desmoronando. Antichamber marcó un antes y un después en la industria y dio pie a una oleada de indies de este género, que harían enriquecer aún más la oferta y la percepción sobre el género. Entre ellos, destaca uno llamado Rooms, que ha servido de inspiración para empezar este proyecto.

3. Diseño del Prototipo

3.1 Planteamiento de Mecánicas

El prototipo presentará un comportamiento No-Euclidiano a través de mecánicas y comportamiento del mundo. Se van a replicar los comportamientos más típicos de este tipo de juegos, se han elegido, los Portales y la Perspectiva Forzada.

La razón por la cual se han escogido estos dos es por ser los representantes más directos de este género y por las oportunidades que generan, los portales expanden las posibilidades de diseño del mundo y la Perspectiva Forzada introduce interacciones y la posibilidad de introducir puzles o desafíos al jugador.

La primera decisión de diseño que se tuvo fue si hacer que los portales fueran móviles o estáticos, se decantó por la segunda opción para que el jugador no tuviera control sobre su entorno y hacer que solo sea un mero observador en él.

3.2 Diseño del Menú

El programa se inicia en el menú principal, entre las opciones está:

1. Jugar el Nivel
2. Jugar Demo Portales
3. Jugar Demo Perspectiva Forzada
4. Volumen
5. Salir

Las demos llevan a entornos controlados donde se ve de forma más sencilla la implementación de las mecánicas mientras que en el nivel está más disimulado y pasado por filtros y post procesado.

El menú contiene una barra de volumen que determina el volumen de la música de toda la aplicación, en el script MainMenu se ve implementado como se va guardando el valor de la barra de volumen.

La función Start inicializa el volumen si es la primera vez que se ejecuta la aplicación, si no es el caso tomará el valor guardado de la anterior sesión:

```
private void Start()
    {

        Cursor.lockState = CursorLockMode.Confined;
        Cursor.visible = true;
        firstPlayInt = PlayerPrefs.GetInt(FirstPlay);

        if(firstPlayInt == 0)
        {
            backgroundFloat = .5f;
            musicSlider.value = backgroundFloat;
            PlayerPrefs.SetFloat(BackgroundPrefs, backgroundFloat);
            PlayerPrefs.SetInt(FirstPlay, -1);
        }
        else
        {
            backgroundFloat = PlayerPrefs.GetFloat(BackgroundPrefs);
            musicSlider.value = backgroundFloat;
        }
    }
```

Las funciones Update y saveVolumeSettings van actualizando los valores del volumen:

```
    private void Update()
    {
        backgroundAudio.volume = musicSlider.value;
        saveVolumeSettings();
    }

    public void saveVolumeSettings()
    {
        PlayerPrefs.SetFloat(BackgroundPrefs, musicSlider.value);
    }
```

Ahora, cada vez que se entre a un nivel se ejecuta la función Start de la cámara principal, cargando el volumen seleccionado de la barra y reproduciendo la música, esta funcionalidad se encuentra en el MainCameraScript:

```
    private void Start()
    {
        Cursor.visible = false;
        backgroundFloat = PlayerPrefs.GetFloat(BackgroundPrefs);
        audioSource.volume = backgroundFloat;
        audioSource.Play();
    }
```

Por último, los botones del menú principal se activan por eventos OnClick que ejecutan las siguientes funciones:

```
public void PlayMainLvl()
{
    SceneManager.LoadScene("Level Concept");
}

public void PlayDemoPortales()
{
    SceneManager.LoadScene("PortalDemo");
}

public void PlayDemoPerspectiva()
{
    SceneManager.LoadScene("ForcedPerspectiveDemo");
}

public void QuitGame()
{
    Application.Quit();
}
```

3.3 Diseño del Nivel, Guía y Explicación

El objetivo del nivel es simple, avanzar. Sólo hay una ruta posible y el jugador ha de navegar por él y buscar la ruta que le lleve más lejos.

El nivel comienza en la habitación del Caballero (Llamada así por la pieza de ajedrez enorme que ocupa la habitación) hay dos caminos, el primero, debajo de las escaleras que te lleva al inicio, y un segundo camino que te lleva a una habitación imposible, es una habitación con 3 mitades que al final te lleva a una habitación con una ventana a una habitación con un alfil y dos puertas, si cruzas cualquiera de las dos puertas acabarás en la puerta contraria, por lo tanto, es un bucle. Para avanzar el jugador tiene que fijarse y asomarse por la ventana, y cuando se aleje se dará cuenta de que se encuentra en esa habitación. Y podrá avanzar.

La habitación del Alfil es muy similar a la del caballero, esta tiene 2 caminos, sin embargo, ambos te llevan a la misma habitación, donde se comunican estos 2 caminos, la habitación contiene unas escaleras y 3 cubos, 2 verdes y uno rojo, además de una puerta que está en un lugar muy alto e inaccesible para el jugador. Los verdes son cubos normales que el jugador puede coger y colocar normalmente, sin embargo, el rojo hace

uso de la perspectiva forzada, el objetivo de este puzle es que el jugador use estos cubos para llegar a la puerta inaccesible y continuar con el nivel.

El nivel continúa en una habitación grande donde hay varios bloques y piezas de ajedrez que el jugador, usando la perspectiva forzada, tiene que usar para acceder a una plataforma con un pilar, si el jugador pasa por detrás del pilar acaba en una nueva habitación con las mismas dimensiones que la anterior.

Esta habitación no está acabada ya que depende de la recursividad para que funcione correctamente, sin embargo, es una idea interesante que en este estado del programa ya se puede experimentar. La habitación contiene una caída libre infinita y a lo largo de esta se va repitiendo la habitación, al otro lado de esta hay una plataforma con un hoyo donde el jugador se debe de tirar. Y finalmente llega a la última habitación del nivel, donde si el jugador explora un poco llega a una ventana que da al inicio del nivel para volver al inicio.

4. Implementación de Mecánicas

4.1 Movimiento del Jugador y la Cámara

Los movimientos del jugador son procesados en el script Player Movement, antes que nada, se va a definir de qué se compone un jugador o que objetos/módulos se atribuyen a él. Un jugador tiene, una cámara, un cuerpo y un groudCheck. El input del usuario es recogido por las funciones GetAxis y con ayuda del módulo CharacterController de Unity. Mediante estos podemos registrar los movimientos de los ejes X y Z y sumar estos deltas para formar un vector de movimiento.

```
Vector3 move = transform.right * x + transform.forward * z;

controller.Move(move * speed * Time.deltaTime);
```

Un GroundCheck es un objeto vacío que se sitúa a los pies del jugador, esto nos será útil para determinar si se encuentra en el aire, en cuyo caso por efecto de la gravedad deberá de caer, o, si se encuentra en el suelo, que en cuyo caso la aceleración en el eje vertical será 0 (en la práctica la velocidad es -2f para preparar las funciones con un valor inicializado distinto de 0). Unity contiene la función CheckSphere, que proyecta una esfera invisible alrededor del objeto (en nuestro caso el GroundCheck), si esta intersecta con algún otro objeto con la máscara indicada (en nuestro caso el suelo y los objetos interactuables que podamos), retornará True.

```
isGrounded = Physics.CheckSphere(groundCheck.position, radius,
groundMask);

if (isGrounded && velocity.y < 0){
	velocity.y = -2f;
}
```

Ahora con los casos límite. Para calcular la velocidad en caída libre se va sumando lo que recorre el cuerpo multiplicado por el tiempo de tick de reloj (Unity te da acceso a esto a través del objeto Time con la función Time.deltaTime).

```
velocity.y += gravity * Time.deltaTime;
```

Y por último el salto. Comprobaremos si el jugador está presionando el botón de salto, en nuestro caso la barra espaciadora y si el jugador está en el suelo. Para realizar el salto, usaremos las definiciones de Energía Cinética y Potencial asociadas a un sistema. La energía Potencial representa a aquella asociada a la posición relativa entre dos puntos del sistema mientras que la Energía Cinética es aquella asociada con el movimiento. La combinación de ambas forma la Energía Mecánica de un sistema.

La magnitud física relacionada directamente con la energía de los cuerpos es el Trabajo, medido en julios. A partir de la definición de trabajo podemos sacar la energía mecánica de un sistema:

$$W = \int_{P1}^{P2} \vec{F} * \overrightarrow{dr} = Ec + Ep$$

Si consideramos que el primer punto es de reposo (estando en el suelo la velocidad vertical es 0). Entonces el cuerpo ha ganado una energía mecánica igual a la energía cinética alcanzada a la velocidad y por tanto obtenemos la fórmula de la energía cinética y de la energía Potencial:

$$Ec = \frac{1}{2}mv^2 \qquad\qquad Ep = mgh$$

Despejamos la velocidad y la adecuamos al sistema:

$$v = \sqrt{2gh}$$

Se puede apreciar la aplicación de la fórmula en el script PlayerMovement:

```
if(Input.GetButtonDown("Jump") && isGrounded){
	velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
}
```

Finalmente, todo el movimiento en el eje vertical se manda al controlador multiplicando el vector obtenido por el tick del reloj:

```
controller.Move(velocity * Time.deltaTime);
```

4.2 Diseño e Implementación de un Portal

4.2.1 Concepto y Objetivo de la Mecánica

Cuando se le hace mención a este término la primera idea que se nos viene a la mente es la obra de Valve, Portal. Obviamente no es el único ejemplo que hay, en el clásico Spyro también podemos encontrar, sin embargo, la idea de Valve iba más allá. Mientras que en Spyro los portales no eran más que pequeños vistazos al skybox usados como transiciones, en Portal se hizo que la jugabilidad se centrara en estos y en vez de llevar a otro nivel o escena, este lleva a otra localización dentro del nivel.

Para nuestra definición de Portal tomaremos la idea de valve. A partir de aquí se van a diferenciar 2 tipos de implementación de portales.

1- Stencil-Based Portals (Buffer de Plantilla)
2- Matrix-Based Portals (Basados en Matrices)

El Buffer de Plantilla sirve para guardar valores de referencia píxel a pixel, lo que permite aplicar geometrías imposibles mediante shaders entre otras cosas. Un primer shader escribe los valores de referencia en el Buffer de Plantilla y un segundo shader escribe las imágenes capturadas por la cámara virtual como parte de un efecto de post procesado. De esta forma el shader leerá el buffer y solo dibujará los pixeles que tengan el valor de referencia adecuados. Sin embargo, este método para hacer portales trae consigo problemas cuando se siga desarrollando el portal y en caso de que se quieran implementar portales con recursividad la dificultad para introducirlos aumenta exponencialmente.

Como se piensa continuar avanzando con el proyecto incluso después de su entrega y entre esos objetivos está la recursividad, se ha decidido por los Portales basados en Matrices.

Primero se va a explicar y presentar un diagrama de flujo que represente el funcionamiento de esta mecánica posteriormente se explicará paso a paso los detalles de implementación donde se examinará el código de las funciones mencionadas en el diagrama.

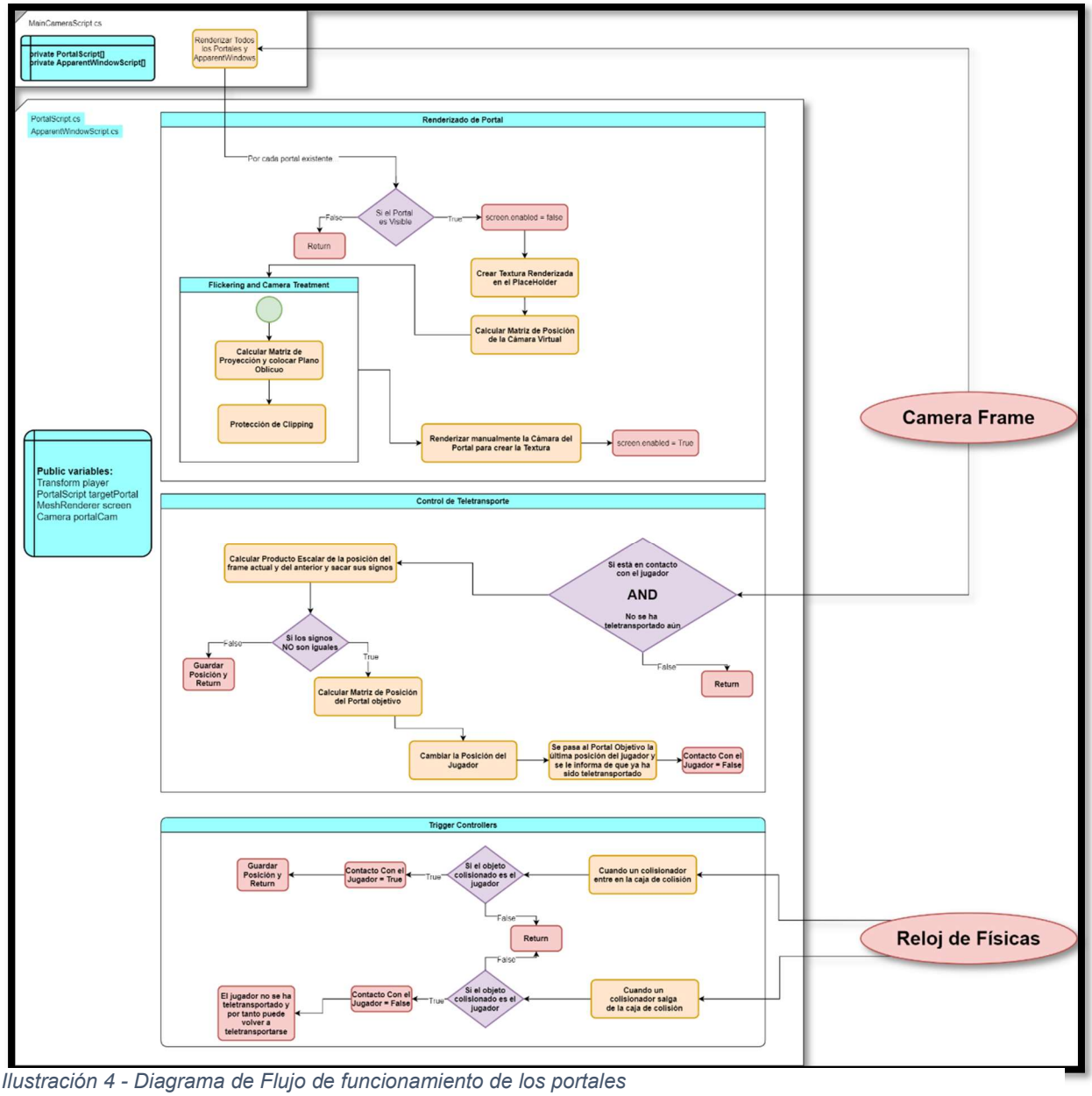


Ilustración 4 - Diagrama de Flujo de funcionamiento de los portales

El Script comienza en el MainCameraScript.cs donde se recogen todos los Portales y se mandan a renderizar al PortalScript. Si el portal es visible para el jugador entonces se desactiva la pantalla del portal y se coloca la textura renderizada en la textura PlaceHolder del portal objetivo. Este proceso lo podemos ver en la función CreateViewTexture().

```
void CreateViewTexture(){
	if(viewTexture == null || viewTexture.width != Screen.width ||
			viewTexture.height != Screen.height){
		if(viewTexture != null){
			viewTexture.Release();
		}

	viewTexture = new RenderTexture(Screen.width, Screen.height, 0);
	portalCam.targetTexture = viewTexture;
	targetPortal.screen.material.SetTexture("_MainTex", viewTexture);

	}
}
```

Una vez con la textura renderizada lista calculamos la matriz de posición para la cámara virtual. Posteriormente se ejecutan los métodos de tratamiento de cámara y resolución de Clipping y Flickering para después renderizar manualmente la cámara del portal y habilitar la pantalla. Luego para el teletransporte se comprueba si el jugador está en el portal y si aún no se ha teletransportado, si esto es cierto se comprueba en qué lado del portal se encuentra el jugador si no ha pasado de lado se guarda la posición y si se ha pasado de lado se produce el teletransporte. Por último, está la sección de trigger controllers que controlan los disparadores de colisiones, para determinar si el jugador está en contacto o no con el portal y a partir de ahí hace sus operaciones pertinentes.

Y esta es su disposición gráfica, lo que queremos que renderice la cámara es la sección rayada de colores:

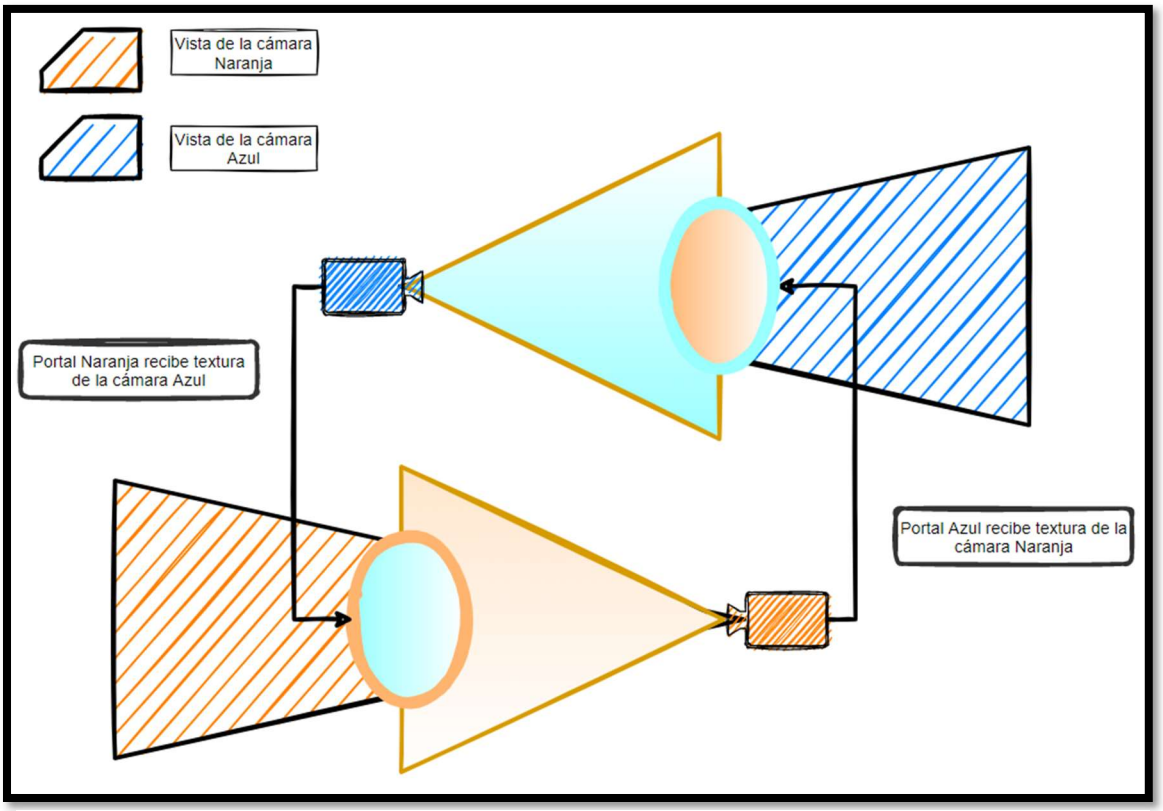


Ilustración 5 - Diagrama gráfico de funcionamiento de los portales

4.2.2 Cámara de Portales y Efecto Visual

Sin embargo, hay muchos aspectos y detalles que no se ven a simple vista desde la ilustración, por eso vamos a explicar en profundidad que ocurre.

A partir del ejemplo anterior se explicará el funcionamiento de las cámaras virtuales. Primero se van a definir los elementos que participan cuando miramos un portal, tenemos un observador (el jugador) como cámara principal, el portal origen (portal naranja en este caso) y un portal destino (portal azul) con su cámara virtual.

La primera idea que viene a la mente es colocar la cámara virtual en la posición del portal. Sin embargo, rápidamente salen los problemas, primero, la imagen es estática y segundo, la imagen carece de profundidad.

Se necesita que la imagen del portal sea dinámica y relativa a la posición del jugador respecto al portal origen. Esto es más entendible si en vez de pensar en portales se piensa en ventanas. Cuando uno se coloca mirando una ventana de un edificio desde la calle no se está viendo lo que hay al otro lado como si se estuviera justo enfrente, sino que lo que se ve depende de donde se observa, de si se está mirando desde la calle o desde una ventana del edificio de enfrente, y depende de lo cerca que se esté de la ventana. Este efecto es el que se busca replicar, con la dificultad de que el otro lado de “la ventana” está físicamente en otra parte.

En otras palabras, para que la ilusión funcione la cámara virtual ha de posicionarse a una distancia y ángulos relativos al portal objetivo, equivalentes a los del jugador con el portal origen.

Se puede calcular la posición de la cámara virtual con el mismo razonamiento de antes multiplicando sus transformadas.

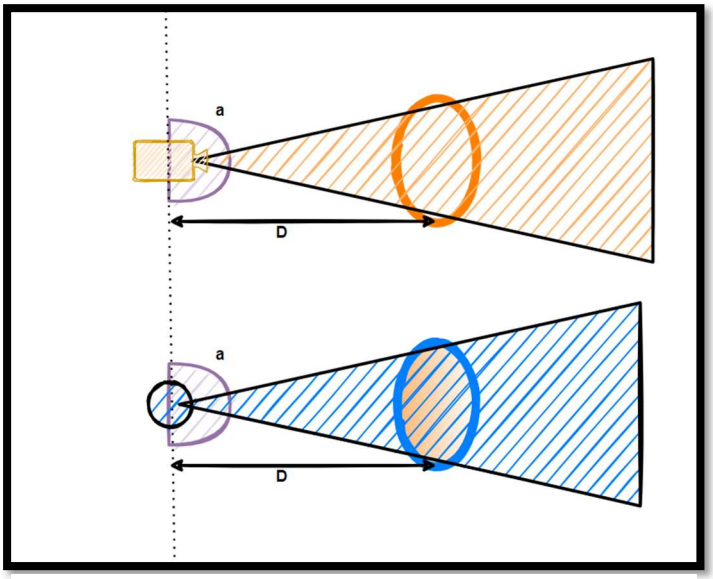


Ilustración 6 - Cámara y Jugador comparten ángulo y distancia con el jugador

```
var matrix =transform.localToWorldMatrix
      * targetPortal.transform.worldToLocalMatrix *
       playerCam.transform.localToWorldMatrix;
```

Y asignarle la posición y la rotación a la cámara usando la función SetPositionAndRotation de la clase Transform.

```
portalCam.transform.SetPositionAndRotation(matrix.GetColumn(3),
      matrix.rotation);
```

Con esto la cámara virtual ya se mueve de forma dinámica con respecto a los movimientos del jugador.

Sin embargo, al introducir la profundidad de campo se produce un nuevo efecto no deseado. La cámara virtual se mueve con respecto a la distancia y al ángulo del jugador con respecto al portal original, eso es correcto, sin embargo, la cámara está renderizando todo el campo de visión, eso significa que lo que haya detrás del portal objetivo también se va a ver a medida que vayamos cambiando la distancia, y en el caso de que haya una pared, solo veremos la pared.

En esta ilustración se observa que el cubo rojo es lo que está delante del portal, y por tanto es lo que deberíamos de poder ver desde el azul, pero debido a que el jugador está a cierta distancia lo que estamos viendo es la pared que hay detrás. Esto rompe la ilusión.

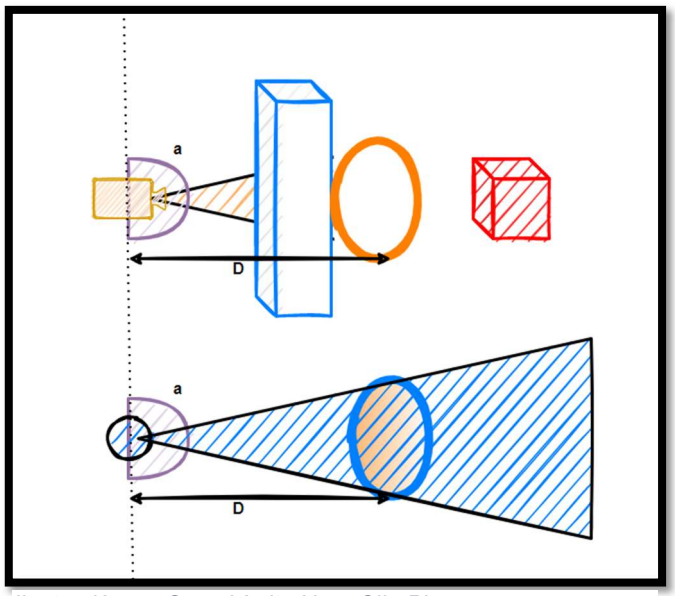


Ilustración 7 - Caso Límite Near Clip Plane

Hay que asegurarse de que solo se renderice aquello que esté por delante del portal objetivo. Para ello calcularemos la matriz oblicua correspondiente que ajustará el plano.

Para solucionar el problema se debe entender primero que es el FOV. El campo visual o FOV se puede describir como los planos troncales que se proyectan desde la cámara y toman la forma de un tronco cuadrangular (al proyectarlo sobre un plano 2D se crea un triángulo, que es la forma que usamos para estos ejemplos), estos planos troncales contienen todo aquello que percibirá y renderizará la cámara.

Por lo tanto, en un campo de visión existen 2 bases, la base superior (la más cercana a la cámara) o **Near Clip Plane** y la base inferior o **Far Clip Plane**, ajustando estos planos se decide desde donde hasta donde y con qué rotación queremos que trabaje la cámara.

Para este propósito tenemos que llevar el Near Clip Plane hasta la posición del portal y cambiar su rotación a la del portal. Este procedimiento está implementado en el proyecto en la función SetNearClipPlane en el PortalScript.cs.

```
void SetNearClipPlane(){
       Transform clipPlane = transform;
       int dot = System.Math.Sign(Vector3.Dot(clipPlane.forward,
       transform.position - portalCam.transform.position));

       Vector3 camSpacePos =
       portalCam.worldToCameraMatrix.MultiplyPoint(clipPlane.position);
       Vector3 camSpaceNormal =
       portalCam.worldToCameraMatrix.MultiplyVector(clipPlane.forward) * dot;
       float camSpaceDst = -Vector3.Dot(camSpacePos, camSpaceNormal);
       Vector4 clipPlaneCameraSpace = new Vector4(camSpaceNormal.x,
       camSpaceNormal.y, camSpaceNormal.z, camSpaceDst);

       portalCam.projectionMatrix =
       playerCam.CalculateObliqueMatrix(clipPlaneCameraSpace);

    }
```

Con el plano oblicuo bien colocado en el portal objetivo sólo queda tratar el último caso límite de la cámara.

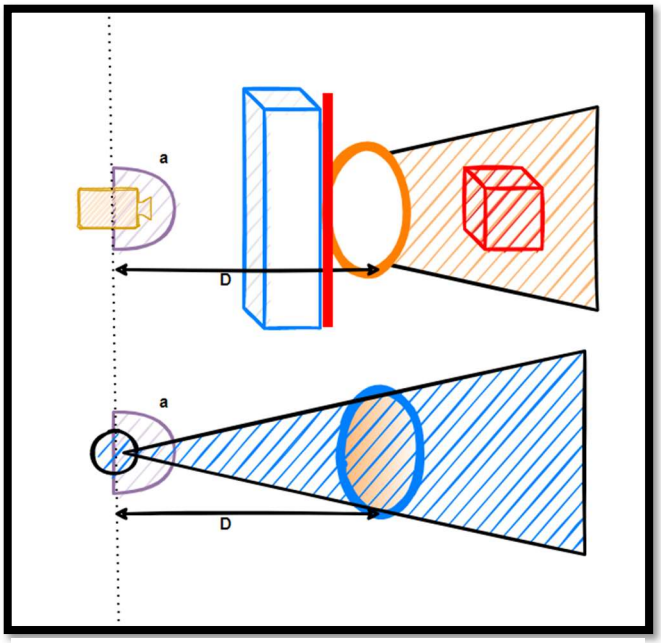


Ilustración 8 - Portal con Plano Oblicuo ajustado

Como se puede apreciar, el ancho del campo visual, al ser con forma de pirámide (triangular en este caso), varía según la altura, de forma que cuanto más cerca estás de la base, más ancha es la pirámide. En el caso actual esto es un problema ya que para nosotros la altura de la pirámide es la distancia del campo visual, y se está usando la distancia para mover la cámara, de forma que, en la mayor parte de los casos, gran parte del campo visual no debería de poder ser visible.

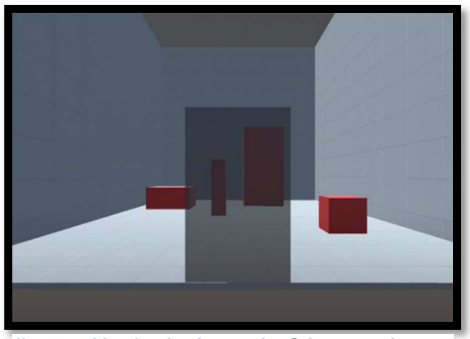


Ilustración 9 - Imágen de Cámara sin recortar

Por eso mismo se debe de recortar la sección de la imagen que no interesa y dejar la que pertenece al área que ocupa el portal. Para hacer esto se creará un Shader que usará la técnica de Texture Mapping. El mapeado de texturas establece cómo se sitúa la textura sobre un objeto al momento de proyectarse, podemos usar esto para recortar la parte de la imagen que cae fuera del campo visual y quedarnos con la que nos interesa. Para mapear la textura dinámica usaremos la propiedad MeshUV, que es un método de representación donde un vector bidimensional representa un punto o píxel de una textura. Siendo el (0,0) la esquina inferior izquierda y el (0.5,0.5) el punto medio de la textura y viceversa.

Primero se toman los valores de posición de cada píxel:

```
v2f vert(appdata v)
{
	v2f o;
	o.vertex = UnityObjectToClipPos(v.vertex);
	o.screenPos = ComputeScreenPos(o.vertex);
	return o;
}
```

Y, por último, usando la función tex2D se crea una nueva textura a partir de la textura que nos renderiza la cámara y el mapeado realizado.

```
fixed4 frag(v2f i) : SV_Target
{
	float2 screenSpaceUV = i.screenPos.xy / i.screenPos.w;
	return tex2D(_MainTex, screenSpaceUV);
}
```

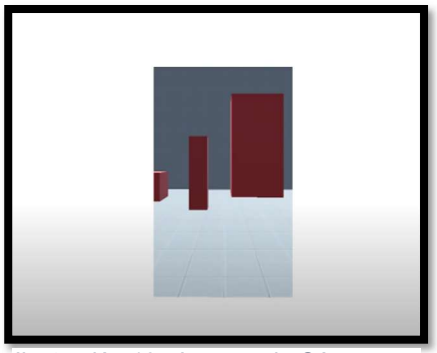


Ilustración 10 - Imagen de Cámara Recortada

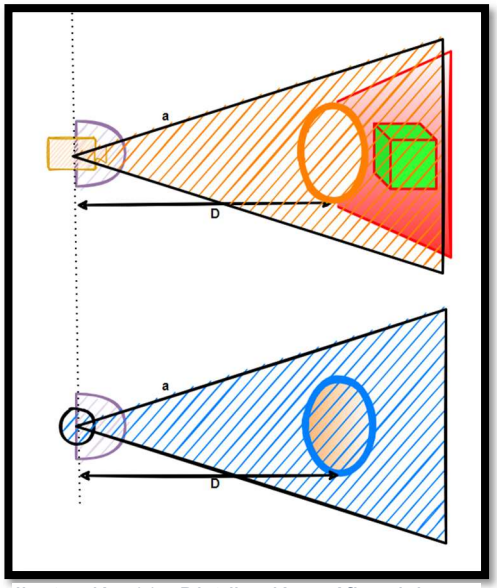


Ilustración 11 - Distribución gráfica del recorte de cámara

4.2.3 Implementación del teletransporte

Para completar la ilusión de portal falta el poder "pasar" por él. Hasta el momento la pantalla del portal no interactúa con el jugador, cuando pase por él se debe teletransportar al portal con el que está emparejado. El comportamiento que queremos que presente es el siguiente: cuando el jugador entre por el portal A este se teletransporta al portal B.

Primero hay que definir qué significa entrar por un portal. El jugador tiene que pasar de un lado a otro del umbral del portal para entrar por él. El umbral es la caja de colisiones que define la región previa a la pantalla renderizada.

En este proyecto se usan colisiones para detectar que el jugador entra y sale del umbral, para ello se usan las funciones OnTriggerEnter, que se dispara cuando entra en contacto con otro collider y el OnTriggerExit que se dispara cuando el jugador abandona la caja de colisión.

```
private void OnTriggerEnter(Collider other)
    {
        Debug.Log("Player Contact");

        if (other.tag == "Player")
        {

            playerContact = true;
            OnTravellerEnterPortal();
        }
    }
```

Ahora bien, se podría ejecutar el teletransporte simplemente desde este trigger, sin embargo, esto producirá un efecto llamado flickering, ocurre porque Unity tiene el motor de físicas (que controla el tick del movimiento del jugador) y la cámara trabajando con relojes que no están sincronizados. Por ello se ha puesto la funcionalidad de teletransporte en la función LateUpdate, que se dispara con cada fotograma de la cámara. Este flickering seguirá apareciendo, pero con menor frecuencia y se explicará cómo evitarlo más adelante.

```
void LateUpdate()
{

	if(playerContact&&!hasAlreadyTeleported){

		Vector3 offsetFromPortal = player.position - transform.position;
		int portalSide = System.Math.Sign(Vector3.Dot(offsetFromPortal,
		transform.forward));
		int portalSideOld =
		System.Math.Sign(Vector3.Dot(previousOffsetFromPortal,
		transform.forward));

	if (portalSide != portalSideOld)
	{
		var m = targetPortal.transform.localToWorldMatrix *
		transform.worldToLocalMatrix * player.localToWorldMatrix;

		teleportPlayer(transform, targetPortal.transform, m.GetColumn(3),
		m.rotation);

		targetPortal.OnTravellerEnterPortal();
		targetPortal.hasAlreadyTeleported = true;
		playerContact = false;
	}
	else
	{
		previousOffsetFromPortal = offsetFromPortal;
	}
}
}
```

Tras cada fotograma salta la función y se determina si el jugador está o no está en contacto con el portal y si ya se ha teletransportado, si está en el umbral del portal (que es donde hasta dónde llega nuestro collidable 3D), y no se ha teletransportado recientemente, se comprueba si lo ha atravesado, si, efectivamente, lo ha atravesado teletransportará al jugador al portal objetivo y marcará como que ha sido teletransportado recientemente.

Hay un detalle que resolver, toca definir cuando se cruza por el portal. Para ello usamos las funciones Dot y comparamos el retorno Dot de este fotograma con el del anterior. La función Dot, también conocida como producto escalar nos es útil para determinar a qué lado nos encontramos del portal. Siendo A la posición actual y B la del anterior fotograma:

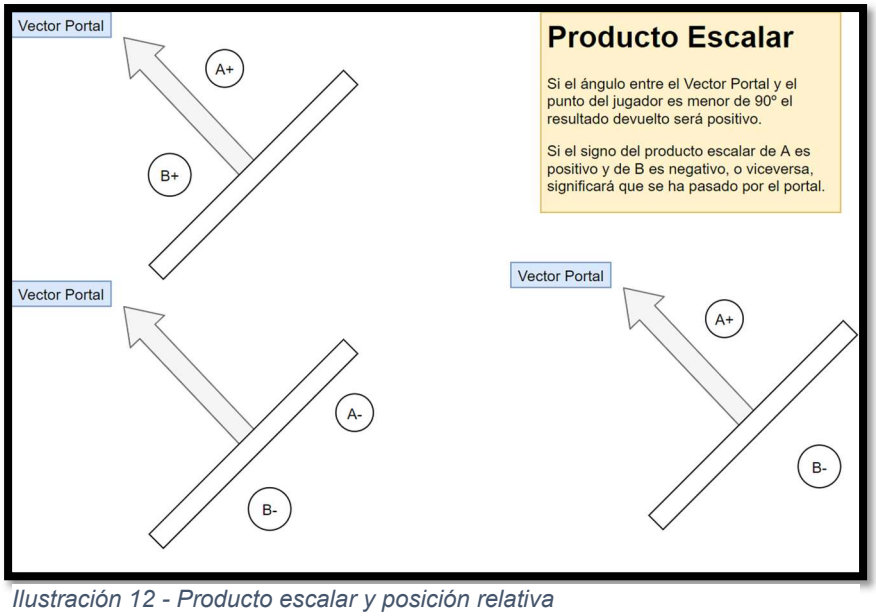


Ilustración 12 - Producto escalar y posición relativa

4.2.4 Tratamiento de Flickering

Se ha mencionado en varias ocasiones el concepto de flickering a lo largo del proyecto y se ha explicado por qué se produce. Y es que no hay un solo motivo para la aparición de este efecto. En principio el parpadeo de la pantalla ocurre por estos motivos:

1. Desincronización de la cámara
2. Atravesar una textura plana
3. Teletransporte tardío, no controlado.

La desincronización de la cámara ya se ha explicado en el apartado del teletransporte, el motor de físicas que regula el movimiento y el momento de los objetos, incluidos los del jugador y la tasa de refresco de la cámara funcionan a destiempo. Hay ocasiones donde estos dos relojes coinciden y la ilusión funciona, sin embargo, no es fiable depender de que ambos relojes estén sincronizados. Tras mucho investigar la función LateUpdate nos ofrece la capacidad de unir el teletransporte a la tasa de la cámara de forma que la cámara renderiza al mismo tiempo que se produce el teletransporte permitiendo que la ilusión sea robusta.

El tema de la textura plana es mucho más sutil, en un principio la pantalla donde se renderiza la cámara virtual era plana, esto funcionaba siempre y cuando la tasa de FPS fuera alta, a más FPS más refrescos por segundo, sin embargo, si los FPS sufrieran una caída entonces el motor de físicas vuelve a molestar. A pesar de que se ha derivado el tick del teletransporte a la cámara el movimiento del jugador sigue funcionando con el reloj del motor de físicas de Unity. Por lo tanto, con bajos FPS el jugador puede seguir encontrándose el molesto flickering.

Además, usando una textura plana ocurrirá que la textura se cruza con el NearClippingPlane de la cámara permitiendo ver lo que hay de verdad al otro lado del portal y no la ilusión que queremos proyectar.

La solución que se le ha encontrado al problema fue desechar la idea de la pantalla plana, en lugar de eso las pantallas de los portales son un rectángulo hueco, sin caja de colisión, donde las caras interiores están pintadas con la textura renderizada tal y como lo están las exteriores. Esto produce que si el jugador entra en el rectángulo seguirá viendo la textura renderizada (o sea, lo que hay al otro lado del portal) sin estar ahí, dándole a la función de teletransporte un poco de espacio de oportunidad para que se ejecute.

Las dimensiones del cubo se calculan de forma dinámica para que éste sea del tamaño correcto en la función ProtectScreenFromClipping :

```
public void ProtectScreenFromClipping(Vector3 viewPoint){
       float halfHeight = playerCam.nearClipPlane *
       Mathf.Tan(playerCam.fieldOfView * 0.5f * Mathf.Deg2Rad);
       float halfWidth = halfHeight * playerCam.aspect;
       float dstToNearClipPlaneCorner = new Vector3(halfWidth, halfHeight,
       playerCam.nearClipPlane).magnitude;
       float screenThickness = dstToNearClipPlaneCorner;

       Transform screenT = screen.transform;
       bool camFacingSameDirAsPortal = Vector3.Dot(transform.forward,
       transform.position - viewPoint) > 0;
       screenT.localScale = new Vector3(screenT.localScale.x,
       screenT.localScale.y, dstToNearClipPlaneCorner);
       screenT.localPosition = Vector3.forward* screenThickness *
       ((camFacingSameDirAsPortal) ? 0.5f : -0.5f);

}
```

4.2.5 “Apparent Windows”

Un apparent window, o ventana aparente, es una ventana a otra ubicación donde si el jugador se asoma por ella, de forma inadvertida, acabará teletrasportado a esa ubicación. Como se puede ver trabaja con elementos y módulos muy parecidos a los portales, sin embargo, contiene unas diferencias críticas que lo diferencian de los portales.

Para mantener la ilusión se necesita definir una ventana activada (donde se podrá ver lo que hay al otro lado, como si se tratara de un portal) y una ventana desactivada (donde sólo se trabaja con el marco de la ventana para que el jugador no note el teletransporte y la cámara que enviará la imagen a la ventana activada). Esta última es clave para mantener la ilusión.

El comportamiento de estas ventanas está escrito en el ApparentWindowScript. Guarda mucho parecido con el PortalScript, sin embargo, contiene algunas diferencias críticas que cambian el funcionamiento de estos.

La ventana activada y desactivada se decide por un nuevo booleano, y cambiará el comportamiento de las funciones principales.

```
public bool windowEnabled;
```

En la función Render:

```
if (!VisibleFromCamera(targetWindow.screen, playerCam))
 {
     if (windowEnabled)
     {
         targetWindow.screen.enabled = false;
     }

     return;

 }
```

Y en el LateUpdate:

```
if (playerContact && !hasAlreadyTeleported && windowEnabled)
{
       Transform linkedPortalTransform = targetWindow.transform;

       Vector3 offsetFromPortal = player.position - transform.position;
       int portalSide = System.Math.Sign(Vector3.Dot(offsetFromPortal,
       transform.forward));
       int portalSideOld =
       System.Math.Sign(Vector3.Dot(previousOffsetFromPortal,
       transform.forward));

       var m = targetWindow.transform.localToWorldMatrix *
       transform.worldToLocalMatrix * player.localToWorldMatrix;
       teleportPlayer(transform, targetWindow.transform, m.GetColumn(3),
       m.rotation);
       targetWindow.OnTravellerEnterPortal();
       targetWindow.hasAlreadyTeleported = true;
       playerContact = false;
}
```

Los cambios en estas funciones crean el nuevo comportamiento, permitiéndonos “desactivar” la pantalla de una de las ventanas mientras que la otra opera con normalidad. Además, al tratarse de una ventana ya no existe el concepto de atravesar un umbral. Por lo tanto, el teletransporte se vale únicamente del disparador de entrada de colisionables que, como el jugador ya no tiene que atravesar polígonos, en el caso de las ventanas no generará los problemas que podría generar con los portales.

4.3 Diseño e Implementación de la Perspectiva Forzada:

4.3.1 Concepto y Objetivo de la mecánica

La perspectiva forzada es una técnica usada en campos como la fotografía, el cine y la arquitectura que hace que un objeto parezca más lejano, más cercano, más grande o más pequeño de lo que realmente es.

Un ejemplo muy famoso de este efecto es la típica foto que se hacen los turistas sujetando la torre de Pisa. Incrementar la distancia de la cámara con la torre hace que se vea más pequeña, mientras que reduciéndola el tamaño aparente aumenta. Por lo tanto, si el turista se coloca a una distancia concreta de la cámara y de la torre, desde el punto de vista de la cámara ambos serán del mismo tamaño. En otras palabras, sus tamaños angulares serán equivalentes.

Para entender cómo funciona la perspectiva forzada se debe estudiar como nuestros ojos perciben el espacio, más específicamente, la profundidad. Este efecto visual se aprovecha de las herramientas y pistas que usan nuestros ojos para percibir la profundidad, tales como, el tamaño angular, la iluminación y sombreado y el tamaño aparente entre otras. Simplificando, el tamaño de un objeto depende únicamente del tamaño de la imagen proyectada en la retina, que ésta depende del ángulo creado por los rayos que vienen de la parte superior e inferior del objeto. Cuanto más grande sea el ángulo más grande nos parecerá el objeto y viceversa. Con esta premisa dos objetos con tamaños reales radicalmente diferentes pueden tener un tamaño aparente equivalente. Igual que, dos objetos de tamaños reales semejantes pueden ser colocados a distancias diferentes y por lo tanto tendrán tamaños aparentes muy dispares.

Finalmente se puede deducir que el tamaño angular o aparente se determina a través de 2 variables:

1. El tamaño o escala real de un objeto.

2. La distancia de la cámara al objeto.

Contando con que el tamaño angular se define como el ángulo que un objeto subtiende visto desde un observador. Observamos que éste proviene de un diámetro medido como un ángulo de una esfera cuyo centro es la cámara, y se mide en arcominutos y arcosegundos. Derivando de la función de longitud de cuerda de un segmento circular, despejando su ángulo y simplificando sacamos la siguiente fórmula, que define el tamaño angular:

$$\delta = 2\arctan\left(\frac{d}{2D}\right)$$

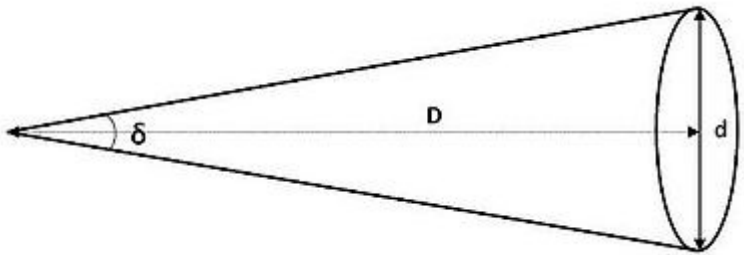


Ilustración 13 - Diagrama Tamaño angular

Donde δ es el tamaño angular, d es la escala real del objeto y D es la distancia que separa al observador del objeto.

El efecto que se quiere conseguir es que al interactuar con un objeto éste siempre mantenga el tamaño angular a la vez que el objeto es arrastrado siempre a la máxima distancia posible de la cámara hasta colisionar con una pared u otro collidable. O sea, que el objeto mantenga constante su tamaño angular independientemente del resto de variables.

4.3.2 Implementación

Dado que el tamaño angular se mantiene constante, y la distancia viene determinada por la distancia entre la cámara y el collidable más cercano con el que se cruce, lo que se ha de cambiar es la escala.

Primero se determinará la distancia entre la cámara y el collidable más cercano, esto se consigue usando una técnica llamada RayTracing muy utilizada para mejoras de iluminación virtual y gráficos. En este caso, lo usaremos para lanzar un rayo en dirección perpendicular al frente de la cámara y sacar la distancia que recorre éste.

En este proyecto su implementación se encuentra en el Script RayTracingScript.cs

```
void FixedUpdate()
    {
        int layerMask = 1 << 8;
        RaycastHit hit;

       if(Physics.Raycast(transform.position,transform.TransformDirection(
       Vector3.forward),out hit,Mathf.Infinity,layerMask))
        {
            hitDistance = hit.distance;
        }
    }
```

Mediante Getters daremos acceso a la información que se necesite.

Una vez obtenida la nueva distancia ya podemos continuar para despejar la nueva escala del objeto. Si tenemos 2 tamaños angulares (uno del frame original y otro del actual) y queremos que sean iguales, simplemente se igualan, siendo A el frame de estado original y B el actual:

$$2\arctan\left(\frac{Ad}{2*AD}\right) = 2\arctan\left(\frac{Bd}{2*BD}\right)$$

Que simplificando queda:

$$\frac{Ad}{AD} = \frac{Bd}{BD}$$

Y despejamos la escala final del objeto:

$$Bd = Ad * \left(\frac{BD}{AD}\right)$$

En el proyecto podemos ver la implementación de esta fórmula en el Script ObjectInteractionScript.cs

```
private void OnMouseDrag()
    {
        Vector3 mousePosition = new Vector3(Input.mousePosition.x,
       Input.mousePosition.y, rayTracingInfo.getHitDistance()-1);
        Vector3 objPosition = Camera.main.ScreenToWorldPoint(mousePosition);
        transform.position = objPosition;
        body.velocity = new Vector3();


        transform.localScale = originalScale * (rayTracingInfo.getHitDistance() /
        originalDistance);
    }

    private void OnMouseDown()
    {
        //Guardar Distancia original y escala
        originalScale = transform.localScale;
        originalDistance = Vector3.Distance(rayTracingInfo.getCameraPosition(),
        transform.position);

    }
```

Conclusiones

Pienso que la creatividad es una herramienta esencial en el mundo actual, que viene acompañada de interés y curiosidad.

Se le ha dedicado muchas horas a este proyecto, no tanto al código sino a entender y leer, estudiar cómo llegar a las soluciones y cómo otra gente ha enfrentado los mismos problemas que yo.

Sin duda la mayor dificultad encontrada durante el proyecto fue el debugging y el tratamiento de flickering. A principios de abril ya se contaba con portales funcionales sin embargo tanto los métodos de cámara como los de teletransporte contaban con bugs que hacían muy complicado colocar los portales de forma cómoda en el mundo. Al ver que el diseño del mundo era tan complicado se decidió reescribir todo y buscar soluciones a los problemas que habían surgido antes de continuar con el proyecto. Fue en esta etapa donde se pasó el teletransporte al LateUpdate y cuando se introdujeron los métodos de tratamiento de flickering, tras 3 frustrantes semanas por fin se logró crear portales fáciles de usar y de colocar en un mundo y robustos a prueba de errores. Conseguir esto es sin duda el mayor logro del proyecto, y al haberlo hecho modular se logró que fuera muy sencillo añadir los portales al mundo.

Otro reto fue la distancia que se requiere para hacer funcionar la perspectiva forzada. A pesar de que la perspectiva forzada no es una mecánica relativamente complicada si que se tuvo que investigar por cuenta propia el cálculo de la distancia. En muchos foros hacen uso de entornos controlados para que funcione la mecánica, pero era demasiado complicado para algo tan simple que era lo que buscaba. Por ello personalmente me enorgullece haber sacado adelante la idea del raytracing y, gracias a la documentación de Unity, a pesar de no haber usado nunca esta técnica se pudo conseguir aquello que se quería.

El prototipo no está terminado, se puede mejorar el teletransporte para que sea más robusto y los portales no son recursivos, de forma que si pones uno en frente del otro se va a romper la ilusión. A pesar de que existen soluciones, están muy mal explicadas y personalmente si no lo entiendo y no soy capaz de explicarlo no se iba a incluir en el proyecto. De modo que estas son las tareas pendientes.

Luego el diseño del mundo podría ser mucho más original y el post procesado más limpio pero debido a la falta de experiencia y que este proyecto no se centra en estos se dejaron unas versiones simples pero funcionales.

Ha sido toda una experiencia construir esto desde 0 y desde luego me ha motivado para empezar nuevos proyectos parecidos a este para intentar resolver las dudas personales que tengo sobre cómo funcionan ciertas cosas. El siguiente proyecto se empezará nada mas se termine este, se va a desarrollar un motor de ajedrez, los detalles de cómo hacerlo los ignoro, pero con una idea general ya se puede empezar a investigar. Hay pocas cosas que motiven tanto como la curiosidad.

Bibliografía

Geometría No-Euclidiana:

http://www.mat.ucm.es/~ccorrale/pdfs/mundosposibles.pdf

https://verne.elpais.com/verne/2015/07/13/album/1436801897_490586.html

http://arts.recursos.uoc.edu/referents-dibuix/es/maurits-cornelis-escher/

https://elperfilmenoshumano.com/2010/08/20/origen-o-la-escalera-de-penrose/

http://www.centroedumatematica.com/aruiz/libros/Historia%20y%20Filosofia/Parte6/Cap21/Parte02_21.htm

http://www.mathematicsdictionary.com/spanish/vmd/full/n/non-euclideangeometry.htm

https://youtu.be/lFEIUcXCEvI

Portales:

https://danielilett.com/2019-12-01-tut4-intro-portals/

https://forum.unity.com/threads/shaders-whats-the-return-value-of-tex2d.592699/

https://answers.unity.com/questions/777302/what-is-mesh-uv.html#:~:text=uv%20is%20an%20array%20of,the%20middle%20of%20the%20texture.&text=For%20example%2C%20they%20allow%20you,side%20is%20a%20different%20texture.

https://www.youtube.com/watch?v=_SmPR5mvH7w&ab_channel=DigiDiggerDigiDigger

https://youtu.be/cWpFZbjtSQg

Perspectiva Forzada:

https://en.wikipedia.org/wiki/Angular_diameter

https://en.wikipedia.org/wiki/Forced_perspective

https://mathworld.wolfram.com/CircularSegment.html

http://www.elcerradodelosfrailes.com/3s29txi/angular-size-formula-605326

https://www.reddit.com/r/Unity3D/comments/3xy9av/i_tried_to_emulate_that_forced_perspective/