SPACE BRAWL

Trabajo Individual de Realidad y Accesibilidad Aumentada

Curso 2020-2021

Jesús Quesada Matilla
UO263624@uniovi.es

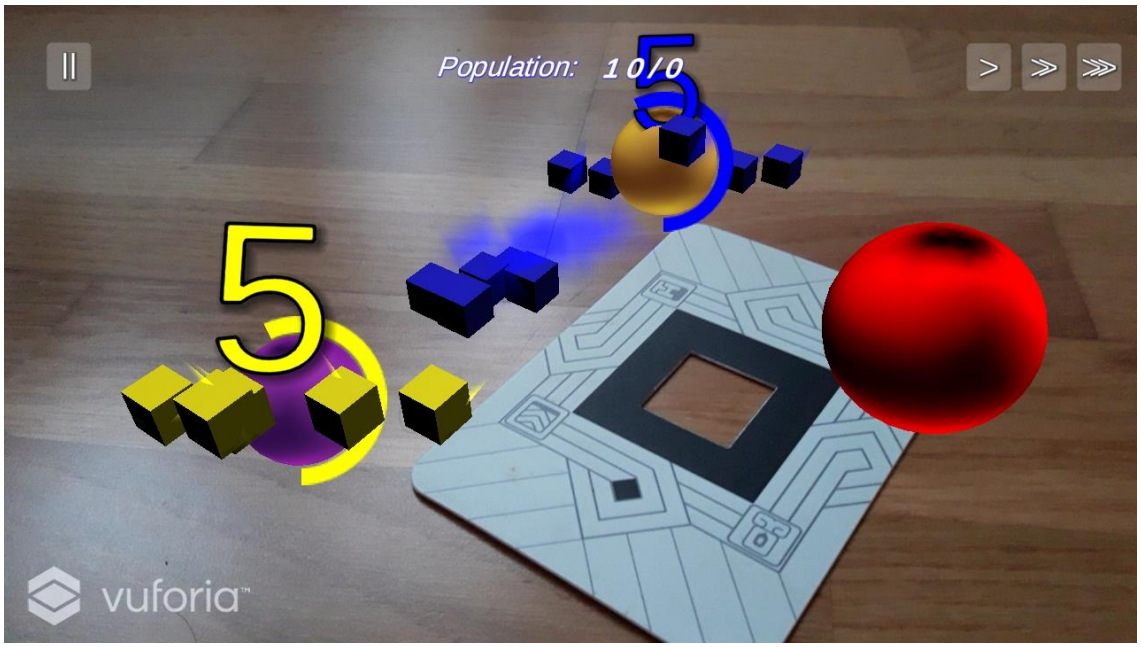
Índice

Introducción

Este documento explica el funcionamiento y creación de la mayoría de los elementos que componen “Space Brawl”, un videojuego de Realidad Aumentada para Android desarrollado para la asignatura de Realidad y Accesibilidad Aumentada de la carrera de Ingeniería Informática del Software de Oviedo.

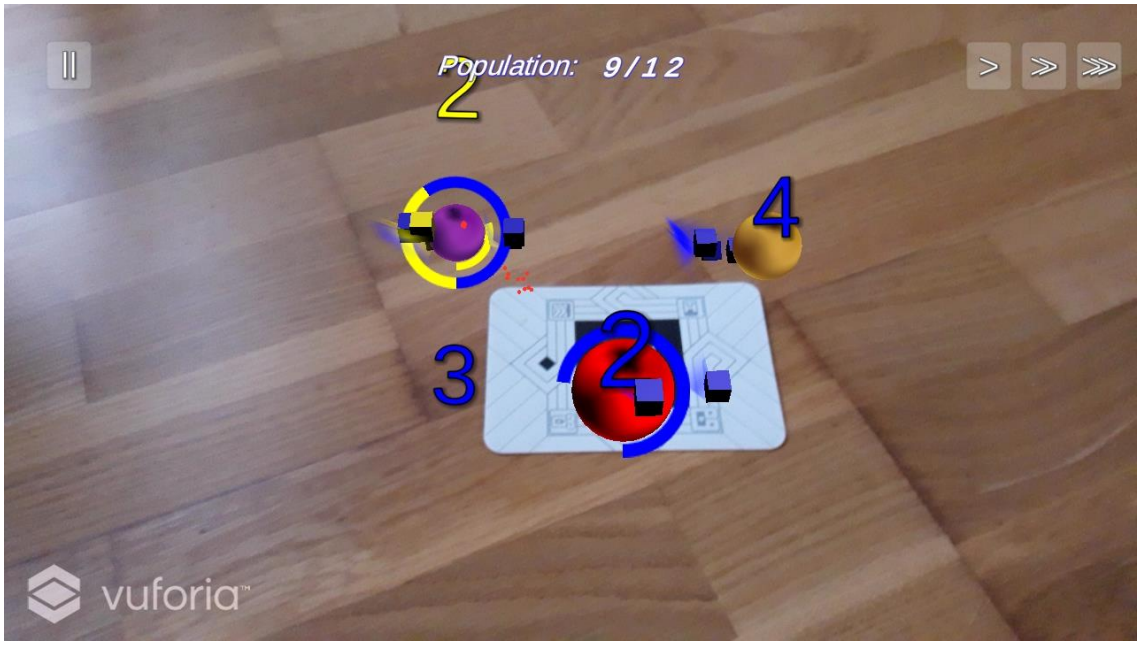
El código del proyecto está disponible en Github de forma pública (https://github.com/jesQM/SpaceBrawlAR) y existe una build para Android en la primera “release” (https://github.com/jesQM/SpaceBrawlAR/releases/tag/v1)

El juego



Space Brawl es un juego de estrategia en tiempo real en el que los jugadores deberán conquistar planetas para aumentar su flota de naves espaciales. Con estas naves pueden tanto conquistar los planetas de otros jugadores como defender los propios.

El jugador ganador es aquel que logra ser el último con tropas vivas.



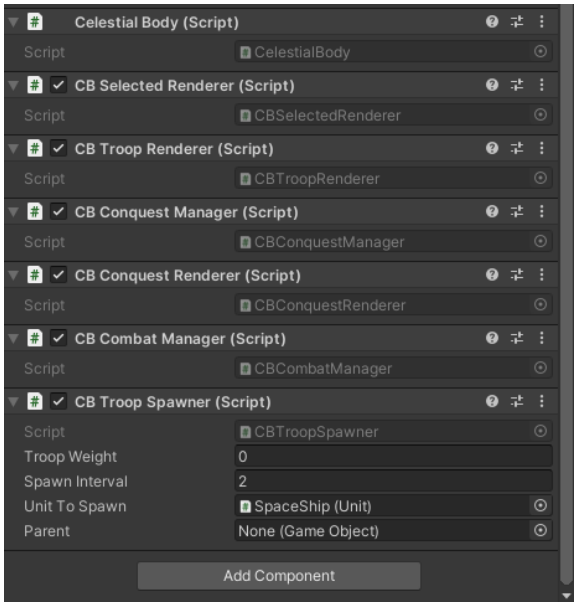
Idea original

La idea original del juego es un videojuego creado en Flash que se llama Solarmax. Para esto proyecto se han emulado la mayoría de las mecánicas base.

Lógica del juego

En esta sección se explicará cómo se han creado algunas de las mecánicas más relevantes del juego.

Planetas



Los planetas (Celestial Bodies o CB como diminutivo) se componen de muchos comportamientos, todos ellos dependientes del script “Celestial Body”.

El nombre de cada uno explica su función, es decir, SelectedRenderer se encarga de mostrar un anillo blanco alrededor de los planetas que son seleccionados, Conquest Manager se encarga de administrar la conquista/de conquista del planeta.

```
// Events
public Action<Team> OnNewTeamArrival;
public Action<Team> OnTeamLeave;
public Action<ITroop> OnTroopArrival;
public Action<Team> OnPlanetConquest;
public Action OnPlanetUnconquest;

public Action OnSelected;
public Action OnDeselected;

// Attributes
public Team Owner { private set; get; }
public (Team team, float percentage) ConquestPercentage { get; private set; } = (null ,0);

public bool IsAtPeace { get; private set; } = true;
public List<Team> CurrentTeamsInPlanet { private set; get; } = new List<Team>();
public Dictionary<Team, List<ITroop>> Troops { private set; get; } = new Dictionary<Team, List<ITroop>>();
```

Los planetas disponen de diversos eventos que son llamados en algunas situaciones, por ejemplo, si el dueño (Owner) del planeta cambia, se llama al evento “OnPlanetConquest” o “OnPlanetUnconquest”.

Las clases anteriormente mencionadas, se suscriben a estos eventos y realizan acciones dependiento del evento lanzado, por ejemplo, la clase que se encarga de generar naves espaciales (también llamadas Tropas o Unidades) se suscribe a este evento para generar tropas cuando el planeta se conquista por un jugador y dejar de generarlas cuando el jugador pierde el planeta.

```
void Start()
{
    this.planet = GetComponent<CelestialBody>();

    planet.OnPlanetConquest += (team) => {
        teamOwner = team;
        teamOwner.MaxTroopCount += TroopWeight;
        StartSpawn();
    };
    planet.OnPlanetUnconquest += () => {
        if (teamOwner != null) teamOwner.MaxTroopCount -= TroopWeight;
        teamOwner = null;
        EndSpawn();
    };
}
```

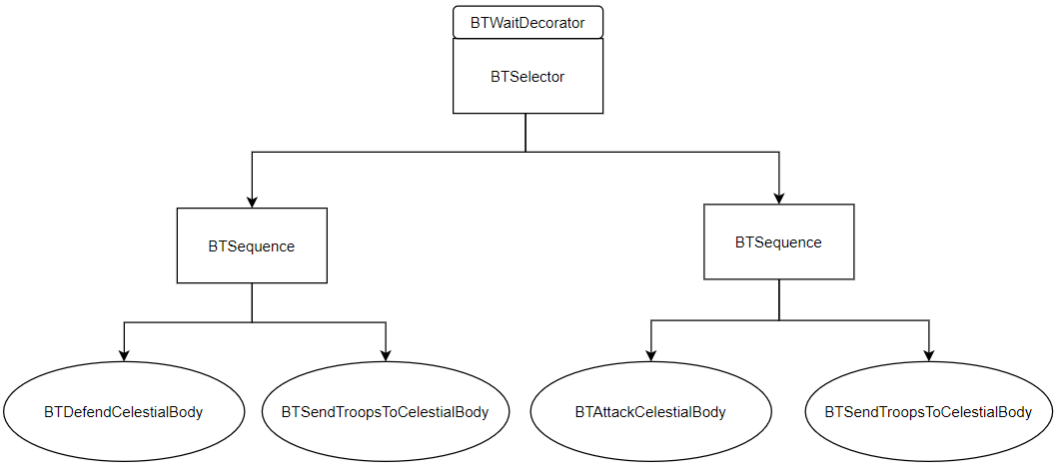
Troop Renderer y Conquest Manager funcionan de esta forma también.

Por ejemplo, Conquest Manager se encarga de conquistar planetas que sean neutrales y des conquistar los planetas de los enemigos cuando un jugador llega a un planeta (si el planeta es del jugador no se realiza ninguna acción). Estas acciones se realizan mediante un patrón Strategy (una clase conquista y otra des conquista), el evento es encargado de asegurarse de que se está ejecutando el correcto.

Inteligencia Artificial

```
public class SpaceBrawlBehaviourTree : BehaviourTree
{
    public Team team;

    protected override void BlackboardConfiguration()
    {
        Blackboard.Add("Team", team);
        Blackboard.Add("CelestialBodyTarget", null);
    }

    protected override void NodesConfiguration()
    {
        BTNode root = new BTWaitDecorator(this,
                new BTSelector(this, new BTNode[]{
                    new BTSequence(this, new BTNode[] { new BTDefendCelestialBody(this), new BTSendTroopsToCelestialBody(this) }), // Defend Sequence
                    new BTSequence(this, new BTNode[] { new BTAttackCelestialBody(this), new BTSendTroopsToCelestialBody(this) }) // Defend Sequence
                }) // End selector
            , 2f); // End wait
        this.Root = root;
    }
}
```

El juego contiene también una inteligencia artificial implementada con un “Behaviour Tree”. Como se puede ver en la imagen, contiene los elementos básicos del árbol: nodos hoja que realizan las acciones, un nodo selector para que la IA defienda o ataque, secuencias para realizar varias acciones seguidas y por último un decorator sobre la raíz para que la IA se tome descansos y no sea demasiado violenta.



La implementación base del árbol está inspirada en un tutorial de ChrisGD[1] y posterior mejorada con un vídeo informativo de Tommy Thompson[2] y la documentación de Unreal, ya que el primer vídeo no mostraba todos los elementos que requería.

Los nodos hoja tienen implementaciones muy sencillas, tomemos “AttackCelestialBody“ como ejemplo, ya que es el más complejo.

```
public override Result Execute()
{
    var s = celestialBodies.Where(b => !team.Equals(b.Owner));
    if (s.Count() == 0) return Result.Failure; // Nowhere to attack

    int startIdx = Random.Range(0, s.Count()-1);
    int currentIdx = startIdx;
    bool stop = false;

    CelestialBody target = null;
    while (!stop) // Select a viable target
    {
        target = s.ElementAt(currentIdx);
        int enemyCount = target.Troops.ToList()
            .Where(pair => !pair.Key.Equals(team))
            .Aggregate(0, (int add, KeyValuePair<Team, List<ITroop>> pair) => pair.Value.Count + add);

        if (enemyCount < team.CurrentTroopCount/2f) break;

        currentIdx++;
        if (currentIdx >= s.Count()) currentIdx = 0;
        if (currentIdx == startIdx) return Result.Failure;
    }

    Tree.Blackboard["CelestialBodyTarget"] = target;
    return Result.Success;
}
```

La tarea principal que realiza es elegir un planeta al azar y comprobar que la cantidad de unidades enemigas sea menor a la mitad de las tropas que el equipo tiene, si es así lo ataca, si no, comprueba con otro planeta. Si tras comprobar todos los planetas ninguno es válido el ataque no ocurre, pero si determina un objetivo se selecciona el planeta como objetivo en el blackboard y se enviarán las unidades a ese planeta en el siguiente nodo (BTSendTRoopsToCelestialBody).

[1] https://www.youtube.com/watch?v=aVf3awPrVPE
[2] https://www.youtube.com/watch?v=6VBCXvfNlCM

Elementos visuales

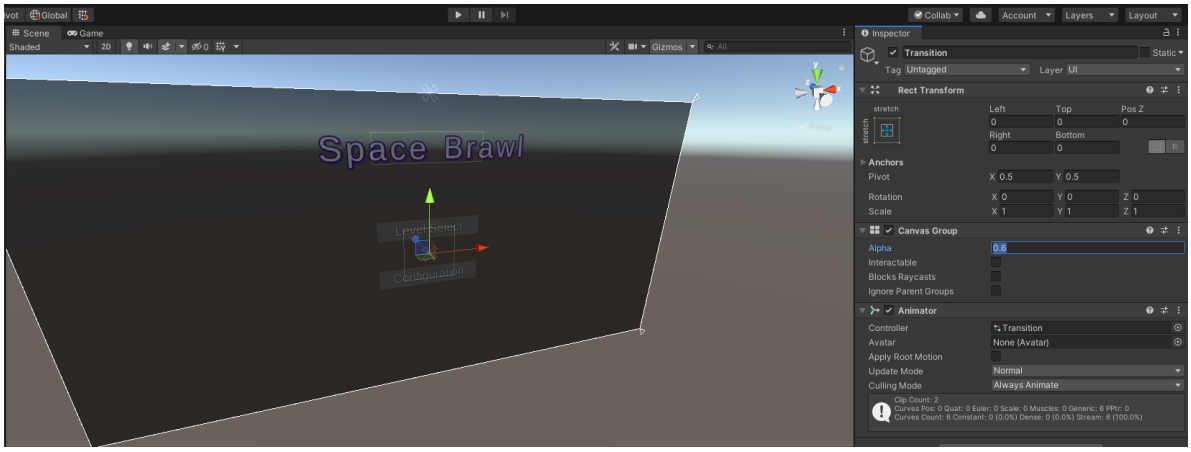
Texturas de los planetas

Las texturas utilizadas para los planetas fueron creadas con Paint Tool SAI y son todas versiones de distintos colores de una misma.

Las texturas han sido creadas de forma que el inicio y el final se enlacen perfectamente, es decir, al envolver a los planetas no se distingue una línea que indique el inicio y el fin de la textura.

Transiciones

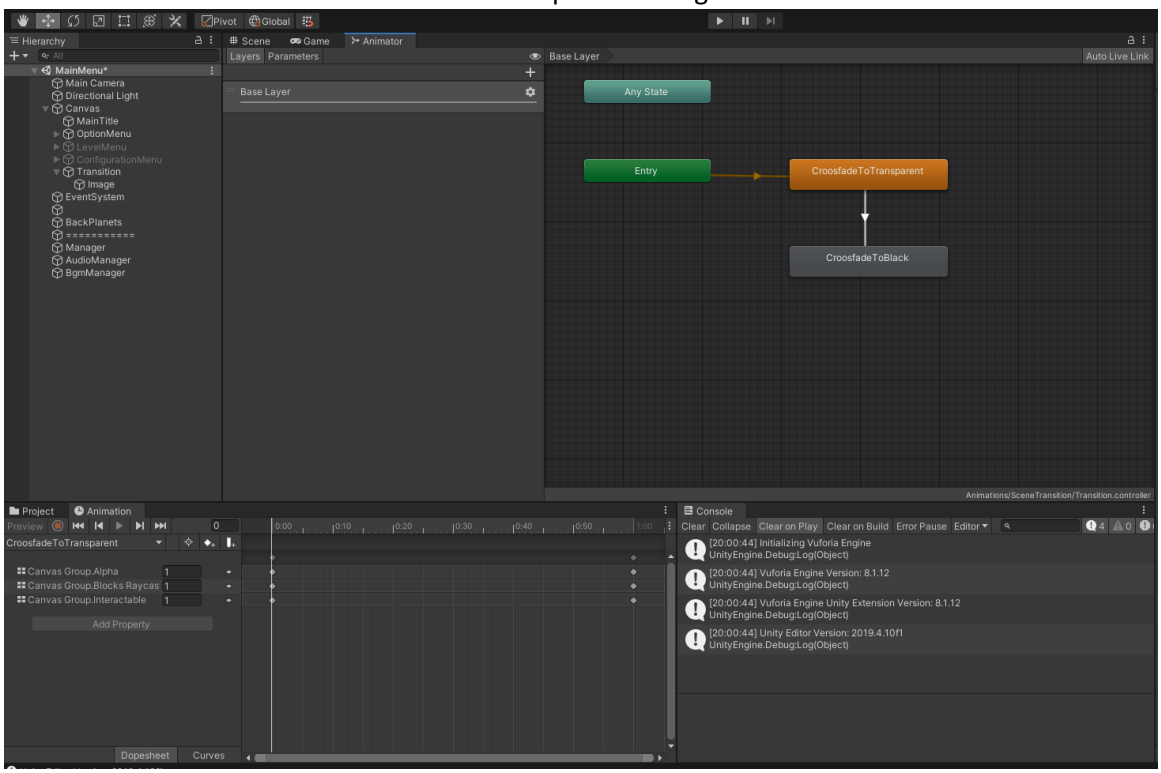
En los cambios de escena existe un fundido a negro para disimular el cambio. Dichos cambios se realizaron siguiendo un tutorial de Brackeys[3].



Consiste en una imagen negra que cubre toda la pantalla y a la que se le modifica la transparencia de 0 a 1 para que se vea la imagen o de 1 a 0 al inicio de la escena para ocultar la imagen.

[3] https://www.youtube.com/watch?v=CE9VOZivb3I

Para modificar el valor se utiliza un animator que se encarga de alternar de un estilo al otro



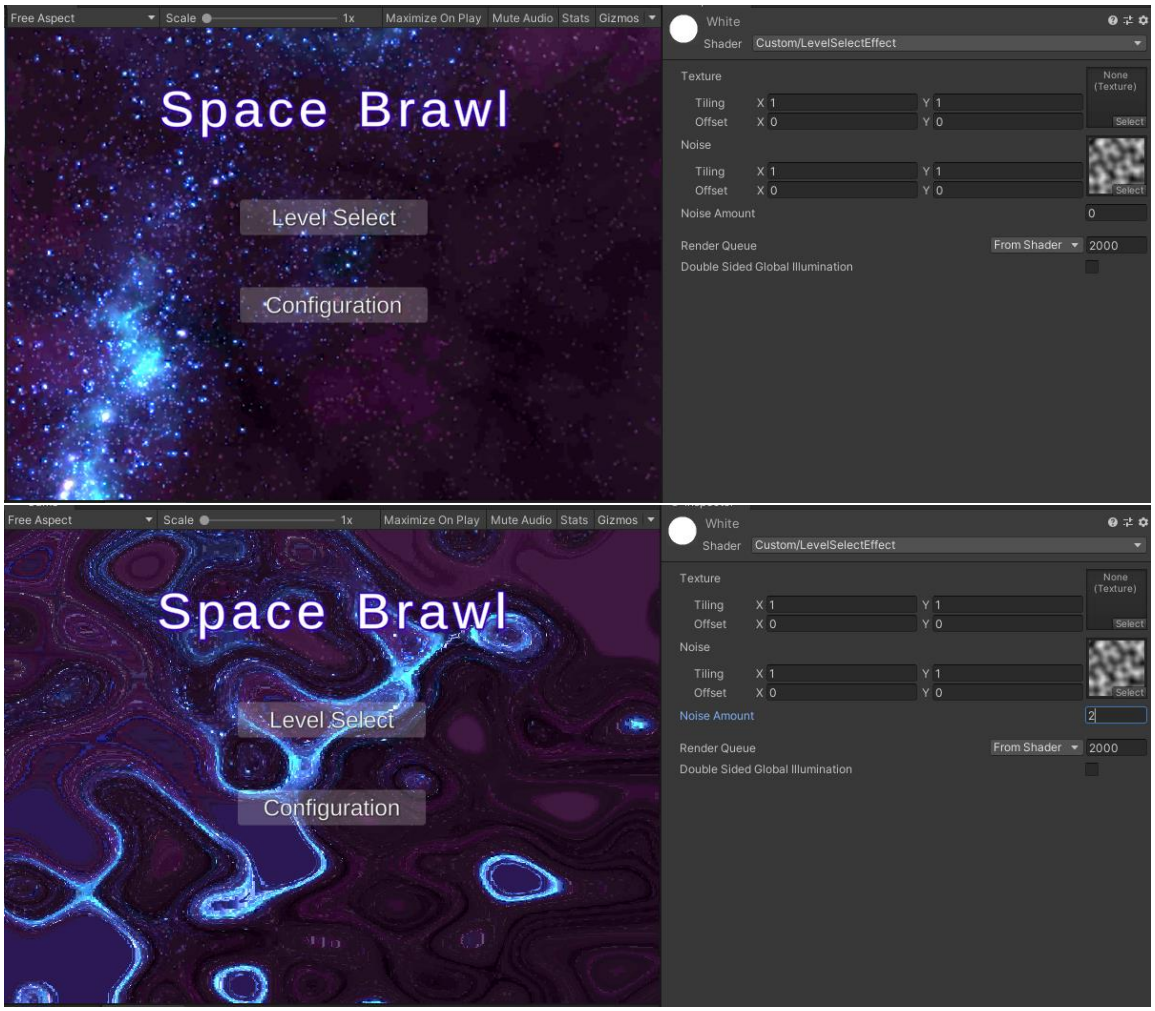
El animador empieza por defecto con un fundido desde negro a transparente y realiza la animación del fundido a negro en el cambio de escana, que ocurre en la clase LevelLoader.

```
public void ReloadLevel() {
    LoadLevel(SceneManager.GetActiveScene().name);
}

public void LoadLevel(string sceneName)
{
    StartCoroutine(LoadScene(sceneName, transitionTime));
}

IEnumerator LoadScene(string name, float transitionTime)
{
    transition.SetTrigger("Start");

    yield return new WaitForSeconds(transitionTime);

    SceneManager.LoadScene(name);
}
```

Shaders

Image Effect Shader

Aprovechando la transición de cambio de escena, se ha realizado un Image Effect Shader que modifica lo que la cámara del menú principal ve cuando se selecciona un nivel y el fundido a negro ocurre.



Este efecto ocurre a lo largo del tiempo que dura el fundido, de forma que da un efecto acuoso al fondo de pantalla.

La idea de este efecto la vi en un vídeo de Michael `Jasper` Ashworth[4]. El efecto era usado para dar el estilo al agua de una escena en un juego de Nintendo, usando una textura de ruido para mover los pixeles de otra textura:

*Fotogramas del vídeo de Michael `Jasper` Ashworth "*How scrolling textures gave Super Mario Galaxy 2 its charm*"*

Para realizar un efecto similar se creó una textura de ruido Perlin a partir de un tutorial de Brackeys[5].

[4] https://www.youtube.com/watch?v=8rCRsOLiO7k
[5] https://www.youtube.com/watch?v=bG0uEXV6aHQ

Con lo que se obtuvo una textura de este aspecto:

En Unity, se creó un script (CustomImageEffect) que aplica un material con nuestro shader a lo que ve la cámara.

```
[ExecuteInEditMode]
public class CustomImageEffect : MonoBehaviour
{
    public Material EffectMaterial;

    private void OnRenderImage(RenderTexture source, RenderTexture destination)
    {
        Graphics.Blit(source, destination, EffectMaterial);
    }
}
```

El shader que aplica este efecto es bastante sencillo:

Primero definimos la textura de ruido que vamos a utilizar y una cantidad que medirá cuanto aplicamos.

```
Shader "Custom/LevelSelectEffect"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Noise ("Noise", 2D) = "white" {}
        _Amount ("Noise Amount", float) = 0
    }
    SubShader
    {
        // No culling or depth
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            CGPROGRAM
```

Y para terminar en la función “Fragment” hacemos operaciones sencillas que selecionan el pixel del ruido y dependiendo de si es negro o blanco escogen un pixel de abajo a la izquierda o arriba a la derecha.

```
    sampler2D _MainTex;
    sampler2D _Noise;
    float _Amount;

    fixed4 frag (v2f i) : SV_Target
    {
        fixed4 offset = tex2D(_Noise, i.uv);

        float x = i.uv.x + (offset.r - 0.5) * _Amount;
        float y = i.uv.y + (offset.r - 0.5) * _Amount;
        fixed4 col = tex2D(_MainTex, float2(x,y) );
        return col;
    }
    ENDCG
}
```

Unlit Shader

El juego contiene también otro shader que se utiliza en los planetas, para resaltar el color del jugador que lo ha conquistado.

El shader tan solo se encarga de hacer una suma al color extraído de la textura con un color asignado como parámetro mediante el código.

```
fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    col += _Color;

    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
ENDCG
```

Dicho parámetro color se modifica paulatinamente con Color.Lerp en una corrutina en el script CelestialBody.

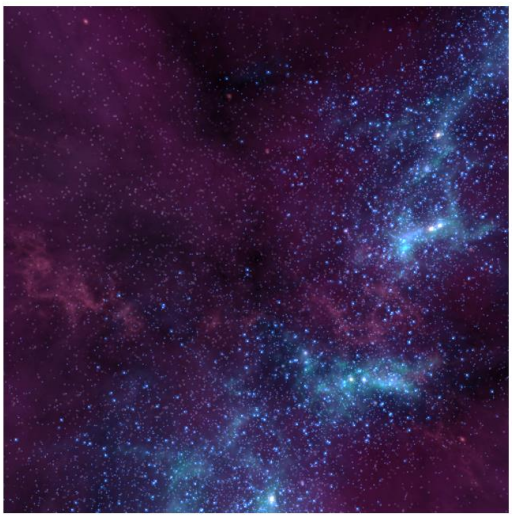
```
IEnumerator BlinkMesh(GameObject go, Color color)
{
    Renderer rend = go.GetComponent<Renderer>();
    Color originalColor = rend.material.GetColor("_Color");

    float percentage = 0f;
    float speed = 1;
    while (percentage >= 0)
    {
        percentage += Time.deltaTime * speed;
        rend.material.SetColor("_Color", Color.Lerp(originalColor, color, percentage));
        if (percentage >= 1) speed *= -1;
        yield return null;
    }
}
```

La corrutina se ejecuta cuando se llaman a los eventos del planeta OnPlanetConquest y OnPlanetUnconquest.

```
// Set Planet Unconquered
OnPlanetUnconquest += () => {
    if (this.Owner != null) StartCoroutine(BlinkMesh(transform.Find("Mesh").gameObject, Color.gray));
    this.Owner = null;
    this.ConquestPercentage = (null, 0);
};

// Set Planet Conquered
OnPlanetConquest += (team) => {
    this.Owner = team;
};

//Planet blink on conquerors colour
OnPlanetConquest += (team) => {
    StartCoroutine(BlinkMesh(transform.Find("Mesh").gameObject, team.Colour));
};
```

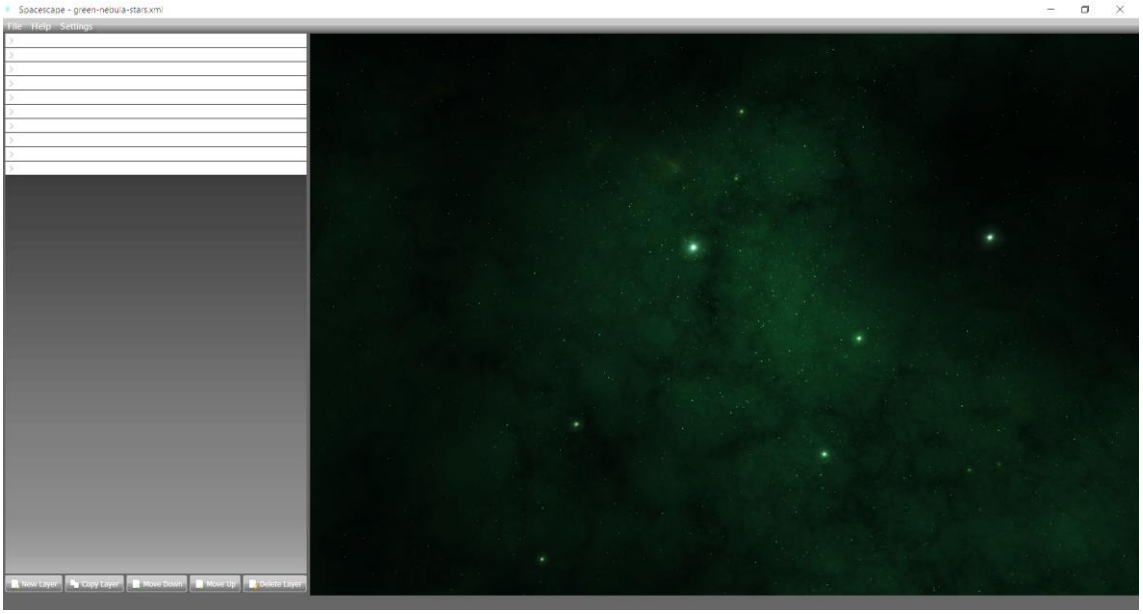
Skybox

Durante los niveles, el fondo de pantalla será lo que la cámara visualiza, pero en el menú principal se ha sustituido la skybox por defecto con otra que contiene un estilo espacial.



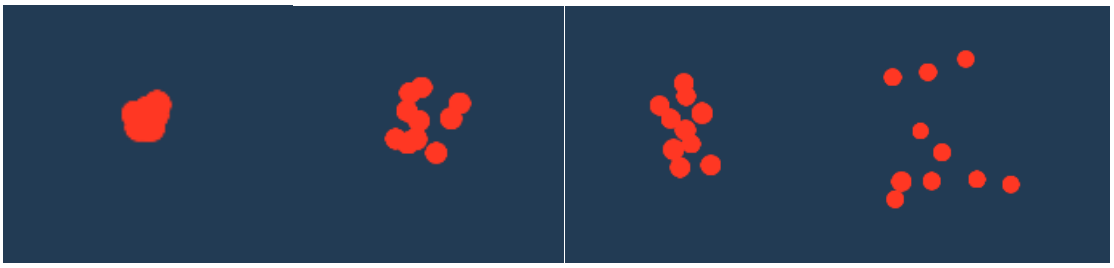
Para dar un mejor efecto durante el menú, la cámara rota lentamente para no mostrar una imagen estática del skybox.

El skybox se ha generado utilizando la herramienta Spacescape[6], que permite generar skybox en formato xml para posteriormente exportarlas como 6 imágenes o con el formato dds.



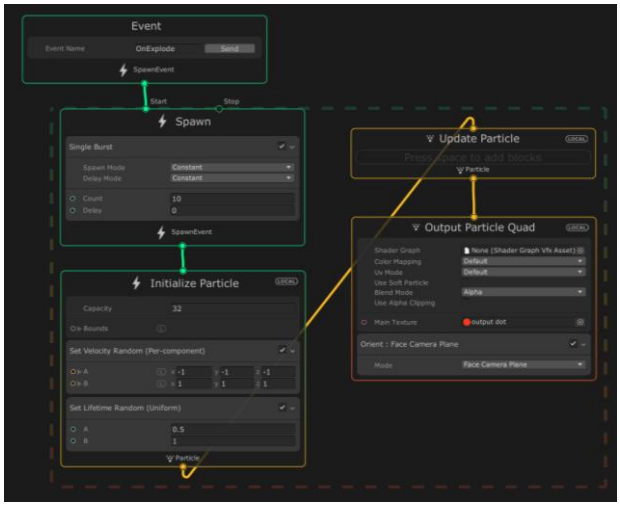
Explosiones

Las explosiones ocurren cuando una nave espacial es destruida en combate.



Las explosiones son descargas de esferas que surgen en un centro y se expanden de forma esférica alrededor de dicho centro.

El efecto ha sido generado con el paquete Visual Effect Graph. Lo único que realiza es lanzar un "burst" de partículas con diferentes velocidades, de forma que se expandan a la redonda.



[6] http://alexcpeterson.com/spacescape/

Audio

Efectos de sonido

El juego contiene tan solo tres efectos de sonido: dos para botones y uno para las explosiones de las naves. Los tres efectos han sido generados con el programa FL Studio.

Para los sonidos de los botones se ha utilizado el sintetizador Sytrus. Tras juguetear con las opciones se obtuvo un sonido sin ataque (el sonido empieza nada más tocar el botón) con un timbre que le da un estilo futurista.

Luego, para dar un efecto de “correcto” y “erróneo” (o confirmar y rechazar) en el sonido se utilizan las notas Mi y Si agudo para correcto y Mi y Si grave para erróneo.

Para el sonido de las explosiones se utilizaron el bombo y la caja de un kit de percusión. A continuación, los sonidos fueron tratados, modificando el pitch y la velocidad de reproducción. Y finalmente se les agregaron efectos como reverberación y ecualización.

Música

El tema que suena de fondo en el juego lo compuse expresamente para este proyecto. Lo interpreté en un teclado midi, para que la grabación quedase en ese formato y así poder modificar cualquier fallo que hubiese cometido sin tener que regrabar fragmentos de la obra.

Para agregar algo de detalle se duplicó la pista del piano y se le puso un instrumento sintético con un timbre etéreo que simula cuerdas o coros.

Para implementar la música en el juego se utiliza la clase “BackgroundMusicManager”, implementada como un singleton mediante el uso de “DontDestroyOnLoad” para que el objeto se conserve entre escenas diferentes y el audio no se corte.

```
[RequireComponent(typeof(AudioSource))]
public class BackgroundMusicManager : MonoBehaviour
{
    public AudioClip music;
    private AudioSource source;

    void Awake()
    {
        // Singleton
        BackgroundMusicManager[] objs = GameObject.FindObjectsOfType<BackgroundMusicManager>();

        if (objs.Length > 1)
        {
            Destroy(this.gameObject);
        }

        DontDestroyOnLoad(this.gameObject);
    }

    void Start()
    {
        source = GetComponent<AudioSource>();
        source.volume = 0.5f;

        source.clip = music;
        source.loop = true;
        source.Play();
    }
}
```