Documento adjunto del Videojuego

Iván Castro Martínez
Ivancastro.info@gmail.com

SUICIDE TANKS

Proyecto propuesto por IMMERSIA.

CONTENIDO

INTRODUCCIÓN

Resumen

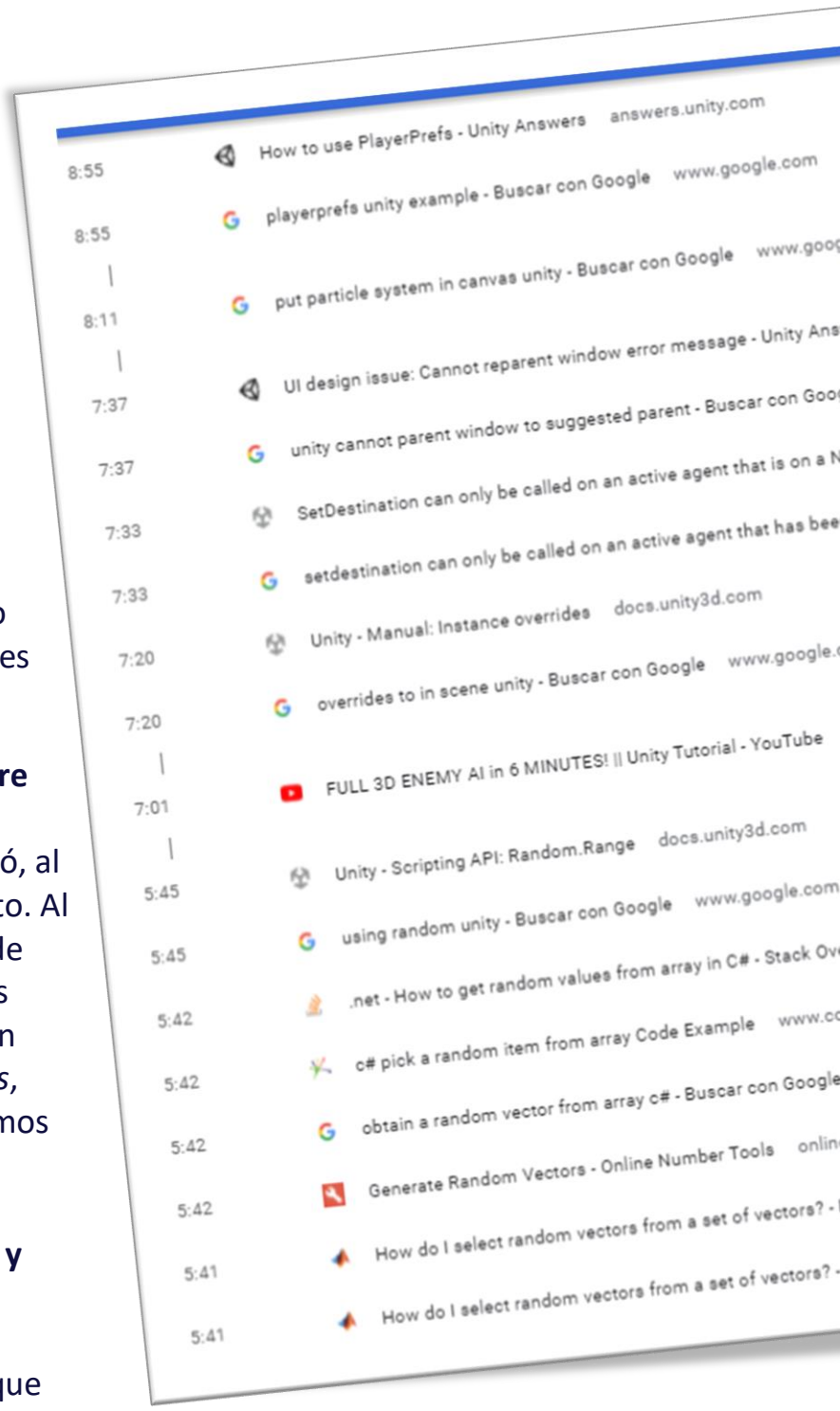
Suicide Tanks es un videojuego de tankes *3D* desarrollado en *Unity 3D* programado en *C#* para plataforma móvil (Android). Ha sido desarrollado como prueba de selección de *IMMERSIA*. Se trata de un videojuego del género *shooter* cuya jugabilidad va regida por turnos. Es un minijuego en el que deberemos vencer con disparos y estrategia al tanke enemigo controlado por inteligencia artificial. Aunque el juego es pequeño y sencillo, toca casi toda la movilidad fundamental a desarrollar en los inicios del desarrollo de un videojuego.

OBJETIVO

El objetivo principal del desarrollo de este videojuego es obtener el cargo deseado. Lo primero que se me vino a la mente con esas mecánicas, fue desarrollarlo en 2D. Sin embargo (y aprovechando la ocasión) quise dar la oportunidad de ganar más experiencia personal y tomarlo como un verdadero reto, estudiando nuevos formatos de videojuego como es programar controles para Android y desarrollarlo en 3D con una panorámica de tercera persona.

El fundamento del juego es: desarrollar la movilidad del tanke personal, la movilidad del tanke de la IA, el sistema de disparos regulable, un área limitada de movimiento y un sistema por turnos.

Fases del Desarrollo



Aprendizaje y planificación

Antes que nada, decir que se hará hincapié en las fases principales. Lo primero fue desglosar todas las fases que llevaría su realización:

- **Formación y programación sobre controles de Android**.
  Sin duda la que más tiempo llevó, al no tener claro su funcionamiento. Al inicio parecía muy complicado de entender como dejar de lado las teclas del PC y sustituirlas por un *Jockstick*, un *Toucfield* y *Botones*, pero resultaron ser casi los mismos vectores que para PC.

- **Programación de toda la lógica y física del controlador de movimiento del tanque.**
  Esta fase fue complicada dado que se debía tener en cuenta, por un lado, el movimiento del cuerpo del tanque (y su distancia de movimiento limitada) y el control

IMPORTANTE. La mecánica para el movimiento del tanke debía ser a través de clicks (y es por eso por lo que imaginé que iba pensado para PC), pero dado que fue desarrollado para Android, preferí que el usuario pueda controlar el tanke y moverlo mediante el Joystick. Lo consideré un juego más llamativo y entretenido si puedes manejarlo. Además, la idea de diseñar un sistema de tercera persona siguiendo a tu tanque al estilo "*Battleship*" es algo que tenía que intentar.

Para crear el límite de zona, dibujé un círculo alrededor del tanque cuyo centro se actualizaba con cada turno y lo crucé con su función de movimiento a través de la medición de la distancia continua entre el centro y el tanque.

```
//If we are moving, we add the new position to the actual position vector of tank
1 referencia
private void TankMovement()
{
    float verticalJoystick = LeftJoystick.Vertical;

    if ((verticalJoystick > 0.2f || verticalJoystick < -0.2f || vertical != 0))
    {
        //If the distance is more than the radius, we limit his movement
        if (distance > radiusLimit)
        {
            Vector3 fromOriginToObject = transform.position - centerPosition; //Vector of distance from tank to center o
            fromOriginToObject *= radiusLimit / distance; //Multiply by radius //Divide by Distance
            transform.position = centerPosition + fromOriginToObject; //Position of tank + Math calcules
        }
        //If the distance is less than the radius, we can move free into
        else
        {
            Vector3 moveTank = (transform.forward * LeftJoystick.Vertical * moveSpeed + transform.forward * vertical * m
            rigidbody.MovePosition(rigidbody.position + moveTank);

            if (!controlAudio)
            {
                controlAudio = true;
                gameObject.GetComponent<AudioSource>().clip = inMovement;
                gameObject.GetComponent<AudioSource>().Play();
            }
        }
```

- **Sistema de disparo.** Esta parte dio bastante problema a la hora de como desarrollarlo. Había que estar en movimiento con el tanque mientras apuntas con el cañón a través de los controles del teléfono. Escogí poner el Joystick en la parte izquierda del teléfono para controlar el movimiento del tanque, y con la mano derecha, moviendo el TouchField apuntaríamos y dispararíamos. Dado que se pedía la

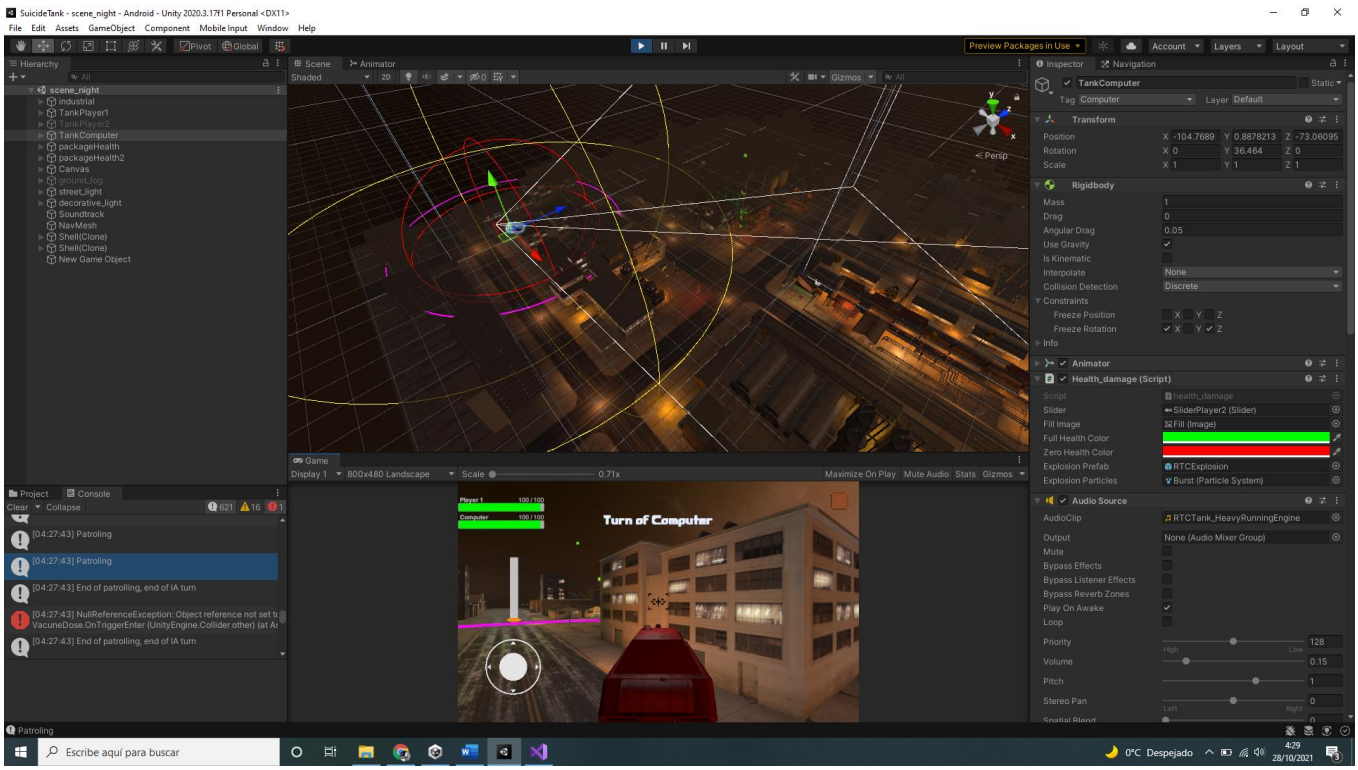
mecánica de poder controlar la fuerza con la que disparamos, creé un *Slider* visual en la parte izquierda para que podamos ver con que fuerza estamos disparando.

Para disparar en la posición del cañón, se hizo uso de un *gameObject* en la punta del mismo desde el cual instanciaríamos la munición. Podría haberse desarrollado con un *raycast* pero quería probar esta alternativa. A través de lo que mantengamos apuntando, cargaremos más las variables haciendo que la bala llegue más lejos. Una vez fijado el alcance, usaremos la función *Fire().*

```
2 referencias
private void Fire()
{
    // Set the fired flag so only Fire is only called once.
    m_Fired = true;

    // Create an instance of the shell and store a reference to it's rigidbody.
    GameObject shellInstance = Instantiate(m_Shell, m_FireTransform.position, tower.rotation);
    Rigidbody rb = shellInstance.GetComponent<Rigidbody>();
    // Set the shell's velocity to the launch force in the fire position's forward direction.
    rb.velocity = m_CurrentLaunchForce * m_FireTransform.forward;

    fireParticle.Play();
    // Change the clip to the firing clip and play it.
    m_ShootingAudio.clip = m_FireClip;
    m_ShootingAudio.Play();

    // Reset the launch force.  This is a precaution in case of missing button events.
    m_CurrentLaunchForce = m_MinLaunchForce;

    Destroy(shellInstance, 2.5f);

    gameObject.GetComponent<TankController>().DestroyNewCircle();
    StartCoroutine(WaitingTheTurn());
}
```

- **Creación de turnos**. Esta mecánica jamás la había programado pero la idea de hacer un semáforo entre dos jugadores no fue complicado de lograr. Le toca a uno, silenciamos el otro. Para darle un toque de gracia adicional, una vez disparamos, el jugador que ha disparado tiene 3 segundos exactos para esconderse o alejarse lo máximo posible del enemigo antes de que comience su turno.

- **Creación de la IA del enemigo.** A pesar de ser una tarea sencilla, dado que Unity te proporciona casi todo lo necesario con el componente *navMeshAgent*, el hecho de tener una zona limitada de movimiento enrevesaba un poco las cosas, dado que la IA se volvía loca cuando le mandas ir a por el jugador pero teniendo en cuenta los límites.



Funciones principales de la IA

Como bien podemos ver en la imagen anterior, la IA maneja dos anillos de detección: uno para la función *Patroling* (el amarillo) y otro para la función *Attack* (el rojo). Veamos en que consisten.

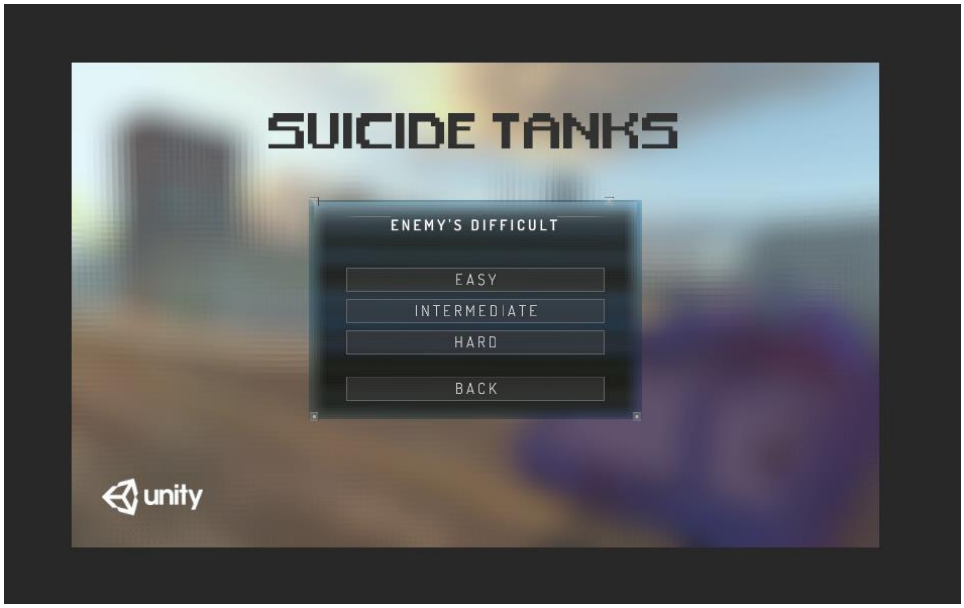
- **Función *Patroling*().** En esta fase, estaremos fuera del alcance de visión de la IA, por lo cual se dedicará a pasear. Esto se consigue con su función manejadora *SearchWalkPoint*(), la cual le otorga un punto aleatorio del mapa al que ir, y si se le acaba, le otorga uno nuevo.
- **Función *ChasePlayer*().** Esta función aunque sencilla, muy importante, hará que la IA comience a seguirnos al entrar en su alcance de visión (anillo amarillo).
- **Función *Attack*().** Si la IA consigue estar tan cerca nuestro que traspasa su anillo rojo, estaremos en su rango de ataque y nos atacará. El daño, fuerza y precisión con que lo haga vendrá dado por la selección de dificultad de la IA.

```
1 referencia
private void Patroling() //meanwhile you not arrive to that random place you have to continue
{
    if (!walkPointSet) SearchWalkPoint();

    if (walkPointSet)
        agent.SetDestination(walkPoint);

    Vector3 distanceToWalkPoint = transform.position - walkPoint;

    //Walkpoint reached
    if (distanceToWalkPoint.magnitude < 1f)
        walkPointSet = false;

    //If we pass 8 segs patroling, we shoot and change the turn
    StartCoroutine(WaitingASeconds());
}
1 referencia
private void SearchWalkPoint() //Function useful to create a random point to follow it meanwhile pa
{
    //Calculate random point in range
    float randomZ = Random.Range(-walkPointRange, walkPointRange);
    float randomX = Random.Range(-walkPointRange, walkPointRange);

    walkPoint = new Vector3(transform.position.x + randomX, transform.position.y, transform.positio

    if (Physics.Raycast(walkPoint, -transform.up, 2f, whatIsGround))
        walkPointSet = true;
}

1 referencia
private void ChasePlayer() //Function hunter. The IA follow you because youre in his sight
{
    Debug.Log("Chasing the player");
    agent.SetDestination(player.position);
}

2 referencias
private void AttackPlayer(int difficulty) //Function IA attack. 2 options: you are too near or the
{
    //Make sure enemy doesn't move
```

- **Creación de los niveles de dificultad de la IA.** Esta tarea más que difícil, fue difícil pensar como desarrollarla. La solución por la que yo opté, fue modificar la fuerza y puntería a la que dispararía la IA, a través de sumarle un vector aleatorio entre valores prefijados (la IA más sencilla tiene aplicado un vector más grande por lo que errará más. Por otro lado, la IA difícil, prácticamente acierta siempre, así que para vencerla hay que jugar muy estratégicamente.

```
//If IA can attack
2 referencias
private void AttackPlayer()
{
    //Make sure enemy doesn't move
    //agent.SetDestination(transform.position);

    int difficulty = PlayerPrefs.GetInt("difficulty");

    Debug.Log("Atacamos");
    agent.SetDestination(transform.position);
    tower.LookAt(player.position);
    Vector3 direction = player.gameObject.transform.GetChild(4).position - m_FireTransform.p

    Rigidbody rb = Instantiate(m_Shell, m_FireTransform.position, tower.rotation).GetCompone

    //IA mode easy, shoot with not much precission
    if (difficulty == 1)
    {
        direction += new Vector3(Random.Range(-5.0f, 5.0f), Random.Range(-5.0f, 5.0f), 0);

        rb.AddForce(direction * 32f, ForceMode.Impulse);
        rb.AddForce(transform.up * 8f, ForceMode.Impulse);
    }
    //IA intermediate, shoot normal and normal damage
    else if (difficulty == 2)
    {
        direction += new Vector3(Random.Range(-2.0f, 2.0f), Random.Range(-2.0f, 2.0f), 0);

        rb.AddForce(direction * 34f, ForceMode.Impulse);
        rb.AddForce(transform.up * 9f, ForceMode.Impulse);
    }
    //IA difficult, shoot with high precission and high damage
    else if (difficulty == 3)
    {
        direction += new Vector3(Random.Range(-1.0f, 1.0f), Random.Range(-1.0f, 1.0f), 0);

        rb.AddForce(direction * 36f, ForceMode.Impulse);
        rb.AddForce(transform.up * 10f, ForceMode.Impulse);
    }
```

Esta dificultad podrá ser seleccionada en el menú inicial del juego.



- **Sistema de vida del personaje.** Para la detección de las balas se han utilizado *colliders* y *OnTriggerEnter*(). Una vez podemos recibir daño, sólo queda implementar las barras de vida y hacer un buen diseño de interfaz atractivo y sencillo.

- **Interfaz, diseño, *assets* y sonido.** Una vez está toda la mecánica obligatoria implementada, sólo queda darle vida a nuestro videojuego con una buena interfaz, un menú sencillo y unos buenos *assets*. Me habría encantado tener el tiempo para diseñar yo mismo el mapa, me apasiona esa tarea, pero tenemos un *asset* de un mapa industrial increíble que pega perfecto con la temática. En el menú tendremos la opción de jugar de día o de noche con una interfaz y banda sonora que te hará meterte en la batalla.

SECCIÓN ADICIONAL

Dada la mecánica del juego, mientras iba desarrollándolo se me ocurrían numerosas formas de optimizarlo y darle más interés con tareas que realmente no llevaban mucho tiempo. Y muchas están deseando desarrollarse a continuación.

- **Creación del modo *Jugador VS Jugador.*** Jugar contra la IA es divertido, pero, que mejor forma de aprovechar un videojuego por turnos que jugando contra un amigo. Dado que debíamos crear el movimiento, rotación y disparo del tanque del jugador, me pareció interesante aprovechar ese código y hacer el modo de juego por turnos.
- **Posición inicial aleatoria.** Es muy aburrido siempre aparecer en el mismo sitio, y no hay nada mejor que un poco de incertidumbre. Quien sabe, ¡tal vez apareces justo en frente del enemigo! a partir de ese momento, tu tomarás las decisiones y a donde quieres moverte.
- **Creación de un sistema de curas.** Para darle más durabilidad y posibilidades de salvarse al jugador, aparecerán dos botiquines aleatoriamente por el mapa que te sanarán un 40% de la vida. Estos botiquines serán sencillos de encontrar porque desprenden un halo verde en el cielo.

  Añadir que **puedes saber la posición** aproximada **del enemigo**. Mirando al cielo, verás que en esa zona hay fuegos artificiales.

  ¿Qué mecánicas se podrían añadir a futuro?
  - Que cada turno tenga un contador de tiempo, y si lo pasas, pierdes dos turnos seguidos.
  - Zona que se reduce, estilo *Fortnite*. Cada turno que pasa, un área circular va cerrando el mapa, obligando a los jugadores a encontrarse, o salirse de la zona y morir.
  - Recoger un objeto aleatorio en el mapa que te otorgue un turno de más.

CONCLUSIÓN

Antes que nada, agradecer a IMMERSIA por ofrecerme esta oportunidad, la verdad me encantó programarlo, me abrió un abanico de posibilidades y ya tengo ganas de volver a crear otro juego en Android.

Otra enorme lección que me llevo es “no juzges un videojuego por su descripción”, al final, lo que aparentemente puede parecer una mecánica aburrida de programar y con pocas expectativas, con un poco de imaginación puede salir un juegazo.

A la pila de proyectos en realidad virtual que quiero diseñar, habrá que sumar algunos de Android. Sin duda alguna me seguiré informando sobre el desarrollo en estos dispositivos así como de su renderización.