UNIVERSIDAD TECNOLÓGICA DE SANTIAGO, UTESA

SISTEMA CORPORATIVO

Facultad Arquitectura e Ingeniería

Carrera de Ingeniería en Sistemas Computacionales

PROGRAMACION DE VIDEOJUEGOS

PROYECTO FINAL

Profesor:

Iván Mendoza

Integrantes:

Guillermo Santos 2-18-0494

28 de abril, 2022

Santiago de los Caballeros,

Rep. Dominicana

INDICE

INTRODUCCION

Ahora le damos inicio al proyecto final de programación de videojuegos, en este caso, un juego que nació de una idea traída por un juego que me intereso mucho cuando lo vi.

En este documento se mostrarán la idea, motivaciones, objetivos, y todo el desarrollo del juego hasta su culminación.

CAPITULO I: VIDEOJUEGO Y HERRAMIENTAS DE DESARROLLO

1.1 Descripción

Este juego se llama “Defend The Core”, se trata de un tower defense de estilo sci-fi con componentes modernos, donde el jugador se verá envuelto en la necesidad de construir torres y otros tipos de estructuras para defender el “CORE”, núcleo de la central minera de la empresa SANCTUSCORE, de los ataques de monstruos provenientes de CORES no purificados en el planeta.

1.2 Motivación

El subgénero de estrategia llamado tower defense siempre ha sido uno de mis géneros favoritos de los videojuegos, y con la observación del juego ROGUE TOWER, un juego tower defense que salió hace unos meses, y sus mecánicas pensaba en cómo sería un tower defense que incorpore ciertas de sus mecánicas, pero que a la vez las reestructure para una mayor dificultad.

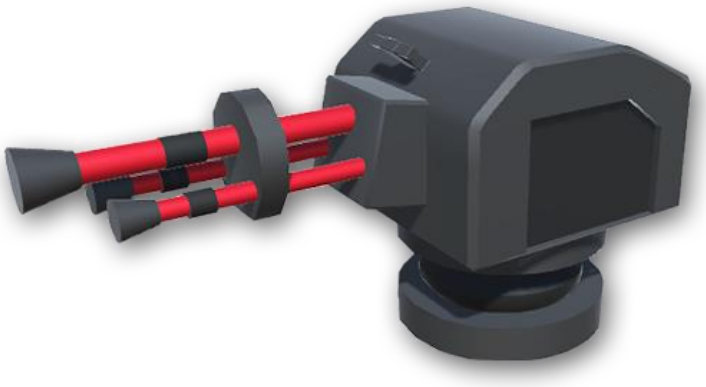
Por lo que cuando me vi en la necesidad de pensar en que desarrollar como proyecto final dije “¿Porque no?” y desarrollé esta idea que tenía aún más.

1.2.1 Originalidad de la Idea

Primeramente, remplazamos la mecánica de uso de dinero por una de recolección de recursos en el mapa en tiempo real, y el “Mana” por “Energía” para que este más acorde con la temática Sci-fi del juego y no con la temática de fantasía del juego que use como inspiración. También se cambió por completo la transición entre oleadas, de una sesión de estrategia donde podías esperar hasta que le des clic al botón de listo para iniciar la oleada, a una cuenta regresiva que te obliga a pensar constantemente fuera y durante las oleadas como reestructurar las defensas para el CORE y las otras estructuras del mapa. Para esto surgió otra idea original, y es el agregado de un tipo de monstruos cuyo comportamiento es diferente al resto, en lugar de seguir la ruta para llegar al CORE y destruirlo, este se sale del camino y se dirige a las estructuras de producción de recursos más cercanas para destruirlas, lo cual te obliga a crear defensas descentralizadas y a no construir sin un plan en mente.

1.2.2 Estado del Arte

Debido a mi inexperiencia con herramientas de diseño gráfico y el modelado 3D, se utilizarán 3 paquetes de assets distintos para el desarrollo del juego, uno usado en la serie de video tutoriales usado para crear la plantilla base del juego mostrada en tareas anteriores, y las otras 2 provenientes de la “Assets Store de Unity”, aquí muestro algunas capturas:



1.3 Objetivo general

Crear un tower defense de alta dificultad con mecánicas únicas y otras utilizadas en el juego que sirve como inspiración.

1.4 Objetivos específicos

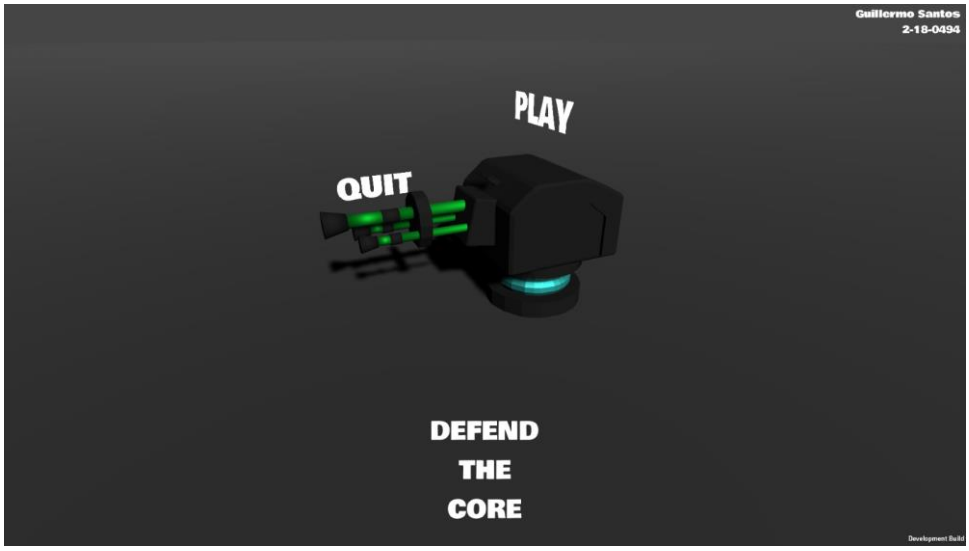
- Crear un juego multiplataforma.
- Crear un tower defense entretenido.
- Crear mecánicas desafiantes que obliguen a pensar al jugador.
- Diseñar un juego de estilo Sci-Fi futurista.

1.5 Escenario

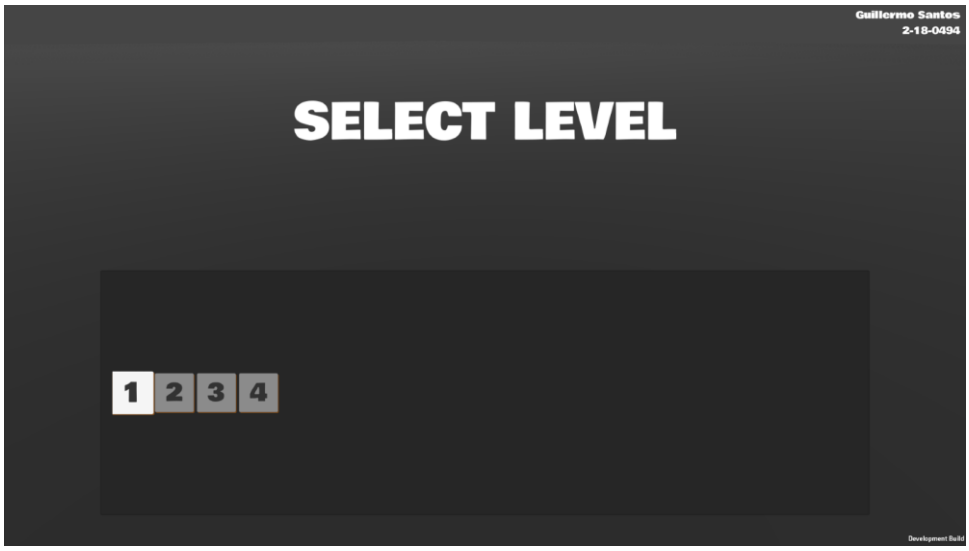
El juego constara de 4 tipos de escenas distintas, la escena del menú principal donde se muestran las opciones de “QUIT” para salir del juego “JUGAR” para entrar al menú de nivel juego, y “UPGRADES” para realizar ciertas mejoras al CORE y desbloquear estructuras (Actualmente en desarrollo).

El escenario de niveles será un cuadrado con varias alturas y recursos dispersos en el mapa, siendo esto distinto para cada nivel, aunque esta parte aún sigue en desarrollo por lo que no se tiene visual de esta todavía.

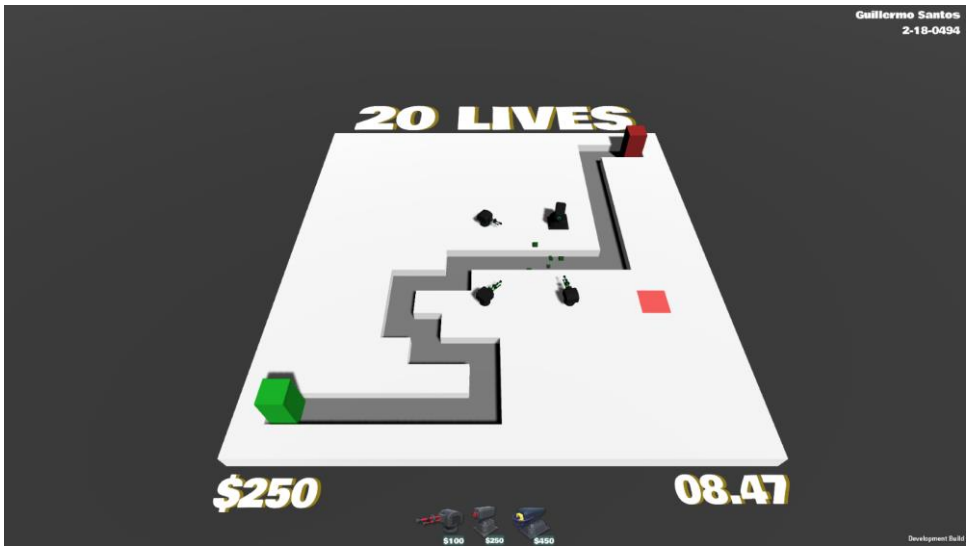
1.5.1 Menú principal



1.5.2 Selector de niveles



1.5.3 Plantilla de escenario de niveles



1.6 Contenidos

El contenido del juego se divide en los distintos componentes que dividen el nivel:

1.6.1 Ambiente

El ambiente está compuesto por 4 elementos principales, el fondo del nivel, que por lo general es un gris cercano a negro, el camino a seguir de los enemigos, el CORE, el EnemySpawner de donde se originan los enemigos y finalmente los nodos.

Existen 2 tipos de nodos:

- **Los construibles:** que son aquellos donde podrás colocar las torres, estos tienen una mecánica única que incrementa el rango de las torres según su altura.
- **Los no construibles:** son los nodos donde no podrás colocar torres, sirven de obstáculos en ciertos niveles para reducir el área en que tus torres son capaces de disparar y son el lugar donde se posicionan los recursos que deberás recolectar en el mapa.

1.6.2 Torres

Algo que nunca falta en los tower defense, las torres, estas están dividas en 2 grupos, las defensivas y las estratégicas.

1.6.2.1 Torres defensivas.

Son las torres principales a construir para defender el CORE, están compuestas por las torres que atacan a los enemigos o tienen algún tipo de efecto sobre ellas. Estas son:

- **StandardTurret:** La torre básica para la defensa de la base.
- **MissileLauncher:** Una torre de defensa con daño de área.
- **LaserBeamer:** Una torre que hace daño constante, además de ralentizar los enemigos.
- **ElectricTower:** Una torre que disminuye la velocidad de enemigos dentro de su área.

1.6.2.2 Torres estratégicas.

Son torres incapaces de atacar enemigos pero que cumplen una función específica, se necesita al menos una de estas para superar un nivel del juego y estas son:

- **HealerTurret:** Una torreta capaz de curar al CORE y otras torres estratégicas dentro de su rango.
- **EnergyTurret:** Una torreta que recolecta energía de los almacenes energéticos en el mapa, solo si uno de estos está en su rango.
- **MiningTurret:** Una torre que se encarga de la recolecta de recursos en el mapa, sin estas los recursos de construcción se acabarán rápidamente.

1.6.3 Enemigos

Los enemigos son una lista variada de los oponentes que aparecerán en las oleadas, Estos constan de 4 tipos y una variante llamada “BOSS” de cada uno de estos.

1.6.3.1 Rail

Es el enemigo base, y cuenta con las estadísticas estándar para los enemigos, se verá presente en todos los niveles.

1.6.3.2 Coloso

Una variante más lenta, pero con mayor vida y tamaño que la variante “Rail”, su dureza hace que se requieran varias torres para abatirlos y si se juntan varios pueden ser un dolor de cabeza para el jugador.

1.6.3.3 Flash

Una variante de “Rail” con menor vida, pero con una velocidad muy superior, dependiendo de la estructura de la defensa, estas pueden convertirse en algo mortal para el jugador.

1.6.3.4 Trickster

Este se puede considerar como el más peligroso entre los ya mencionados, a diferencia que el resto, comparte las funciones de ataque de las torres (En lugar de morir luego de chocar con el CORE, este le ataca a la distancia), pero lo que resulta verdadera mente problemático es que en lugar de seguir el camino junto con las demás torres, esta ataca a las torres estratégicas hasta que no queden ninguna, solo así se dirigirá al CORE, tiene ligeramente más vida y mayor velocidad que la variante "Rail" lo que sumado a su particular comportamiento lo vuelve un gran dolor de cabeza y un factor del cual hay que cuidarse en varios niveles, pues pueden arruinar completamente la partida.

1.6.3.5 Variante jefe

Cada uno de los enemigos antes listados tienen una variante jefa, con estadísticas muy superiores, son el enemigo final en varios niveles. El hecho de que no se limitan a solo uno por nivel más, la posibilidad de la variable Trickster, lo vuelven un enemigo a temer.

1.7 Metodología

La metodología seguida para el desarrollo del juego ha sido la incremental, donde cada incremento está compuesto por una serie de funcionalidades del juego y sus componentes, en total serian 3 incrementos:

1.7.1 Incremento I: Plantillas para el desarrollo

Este es el primer y más pequeño incremento, consta de la creación de las bases para el diseño de los niveles y la composición del juego base o plantilla ya presentados en tareas anteriores.

1.7.2 Incremento II: Torretas, enemigos, componentes interactivos y mejora de las mecánicas bases

La creación de las torretas de defensa, las torretas estratégicas, los distintos enemigos y los recursos del mapa, así como la interacción entre todos ellos, además de cambios en las mecánicas bases del juego, como el sistema de oleadas, sistema de objetivos y pathfinding de enemigos, y condiciones de victoria o derrota de los niveles.

1.7.3 Incremento III: Diseño de niveles y mecánica “Upgrades”

Con los componentes de los incrementos anteriores se diseña cada nivel que tendrá el juego, además se plantea la creación del sistema de mejoras que se utilizará para desbloquear las torretas y realizar mejoras, como la cantidad de recursos iniciales, vida del CORE y reducciones de costos.

1.8 Arquitectura de la aplicación

La arquitectura de desarrollo de la aplicación será la arquitectura en capas, donde el juego estará divido en distintas capas que representan un componente en específico, como los materiales usados para dar color a los modelos, audio y audiomixer para controlar los volúmenes de los audios, Prefabs para los Prefabs que componen los objetos del juego, Scripts para la lógica y controladores del juego y ScriptableObjects que representan la base para el comportamiento de los objetos del juego.

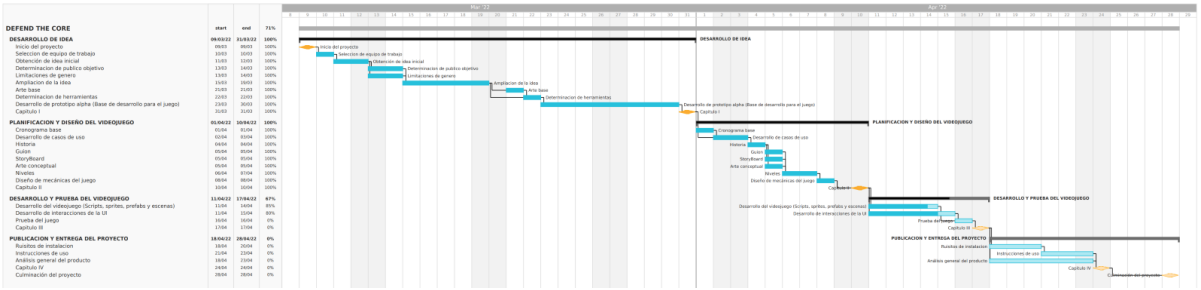
1.9 Herramientas de desarrollo

Se utilizará una serie de herramientas para el desarrollo del juego, como son:

- Motor grafico: Unity
- IDE: Visual Studio 2022
- Fuente de audios: https://elements.envato.com
- Unity Assets Store

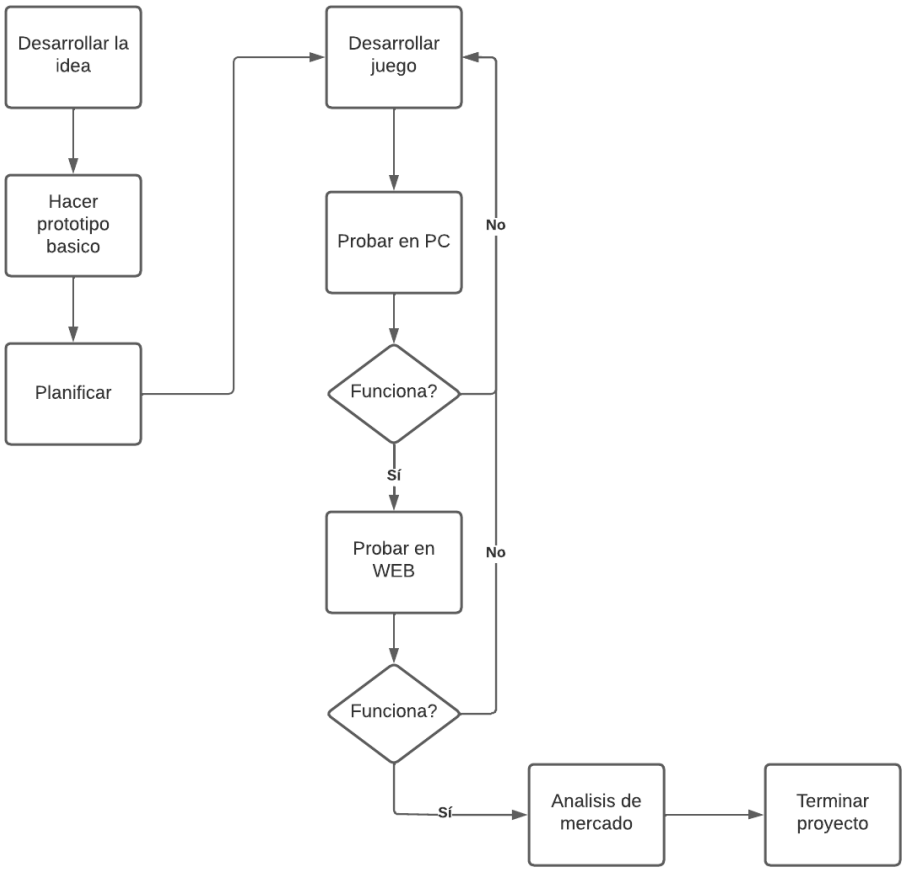
CAPÍTULO II: DISEÑO E IMPLEMENTACIÓN

2.1 Planificación (Diagrama de Gantt)

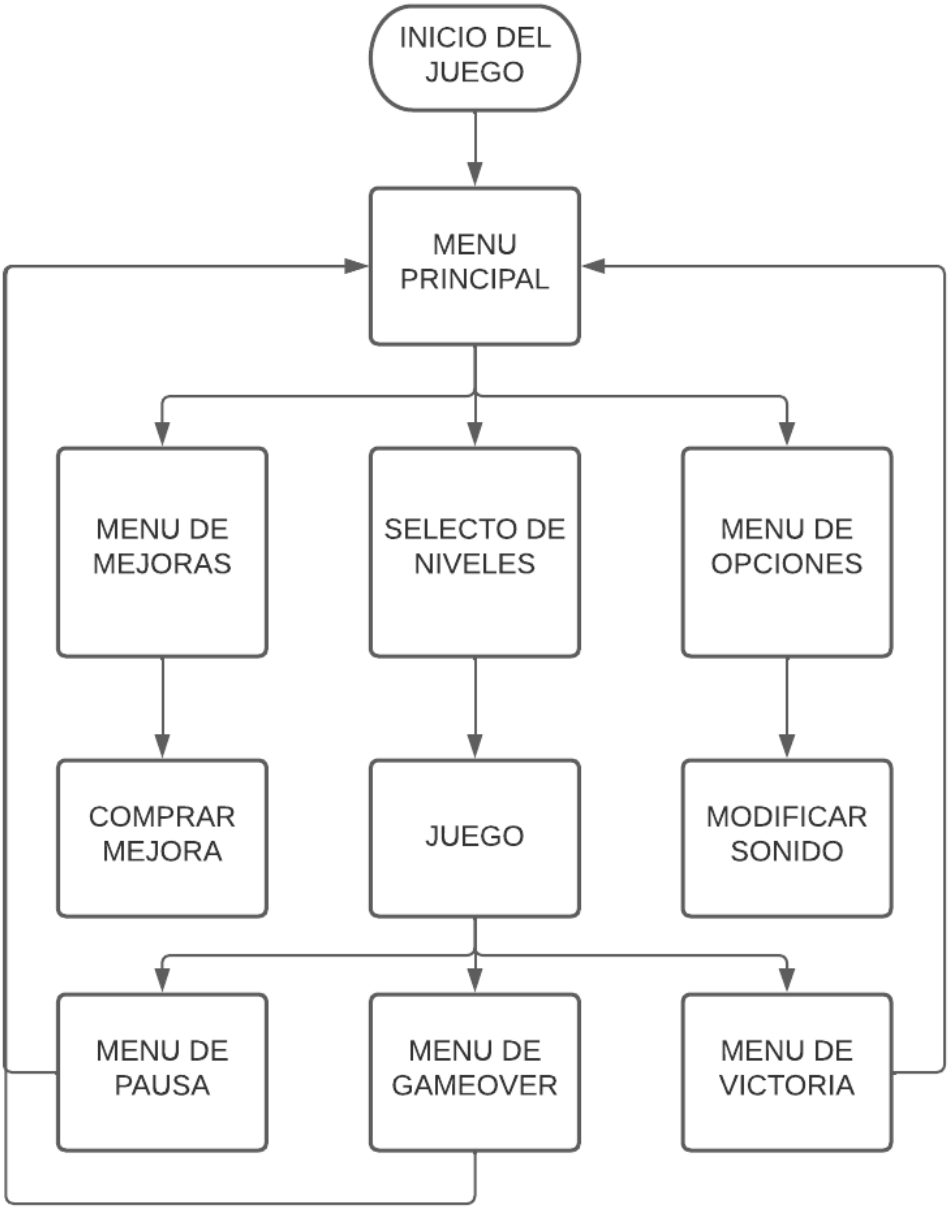


2.2 Diagramas y Casos de Uso

2.2.1 Diagrama del proyecto

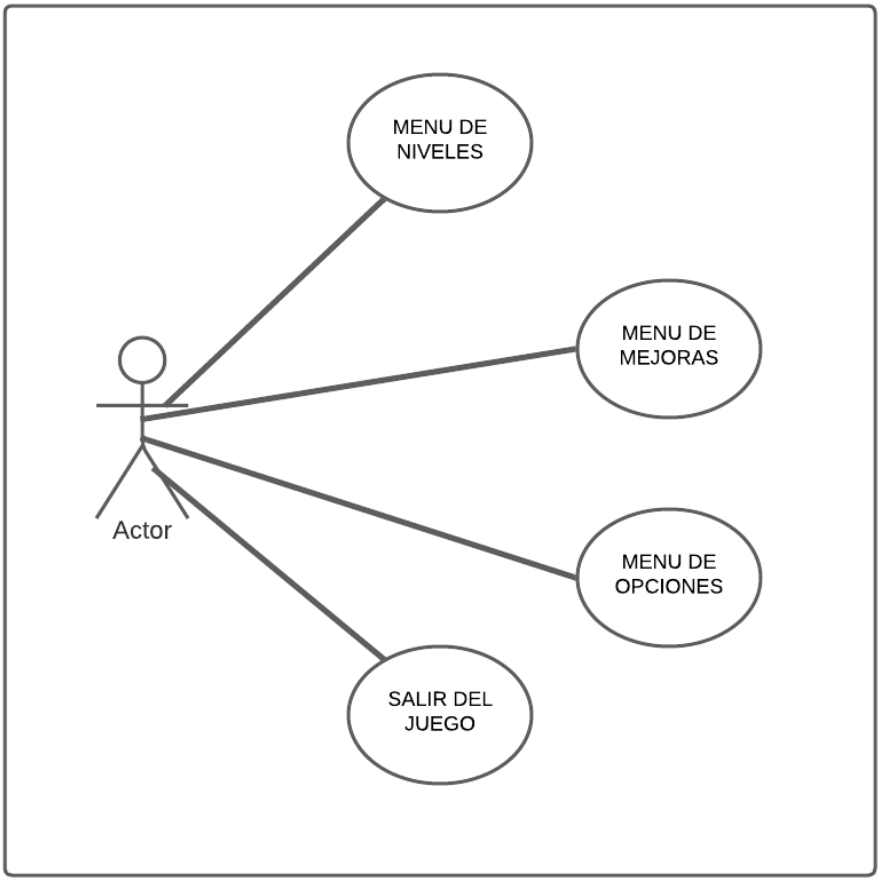


2.2.2 Diagrama del juego

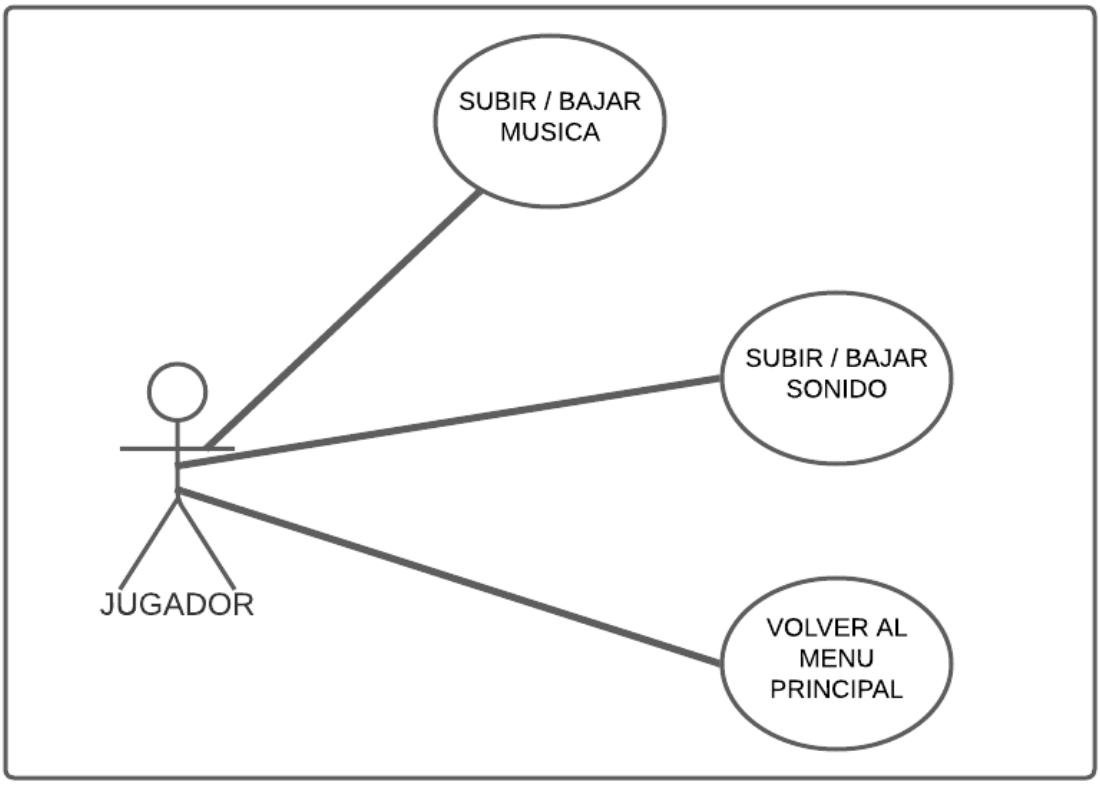


2.2.3 Casos de uso

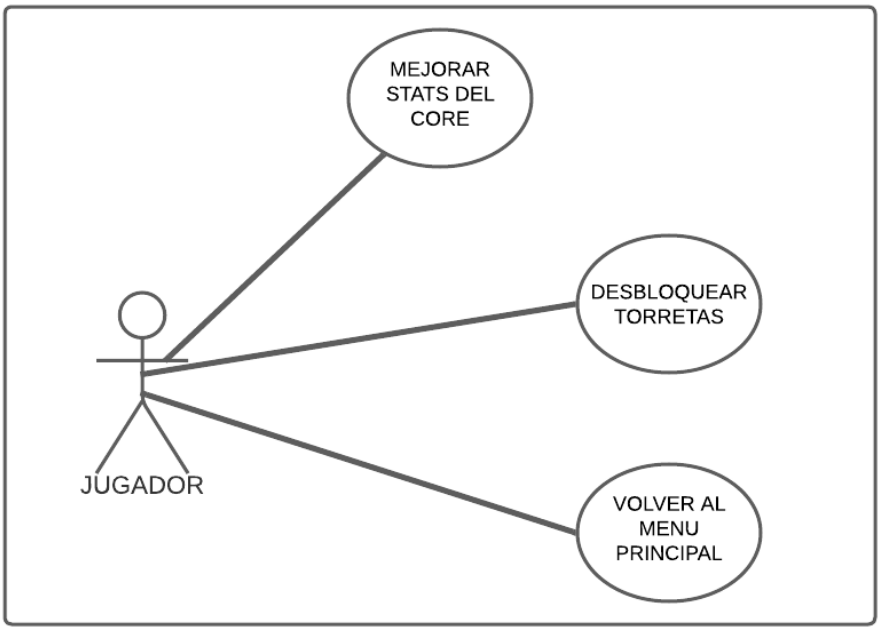
2.2.3.1 Menú principal



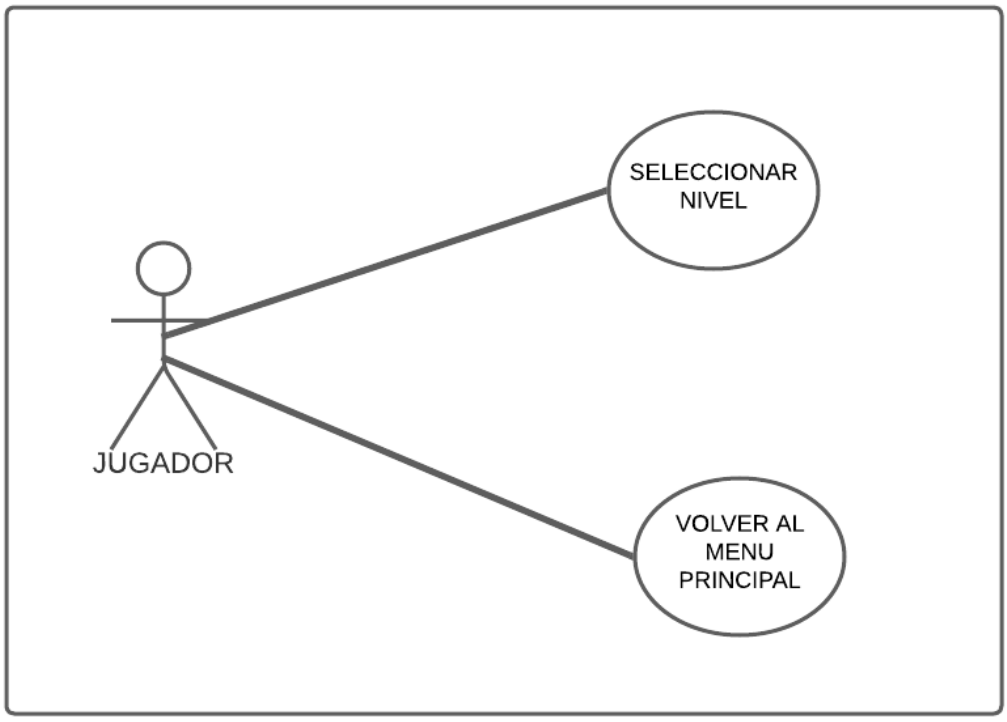
2.2.3.2 Menú de opciones



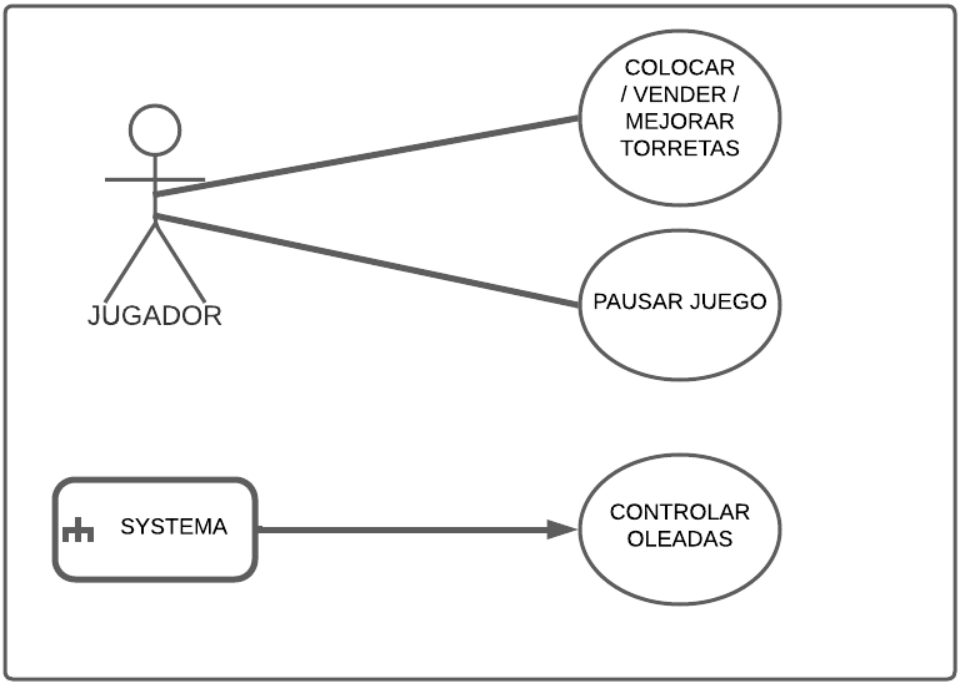
2.2.3.3 Menú de mejoras



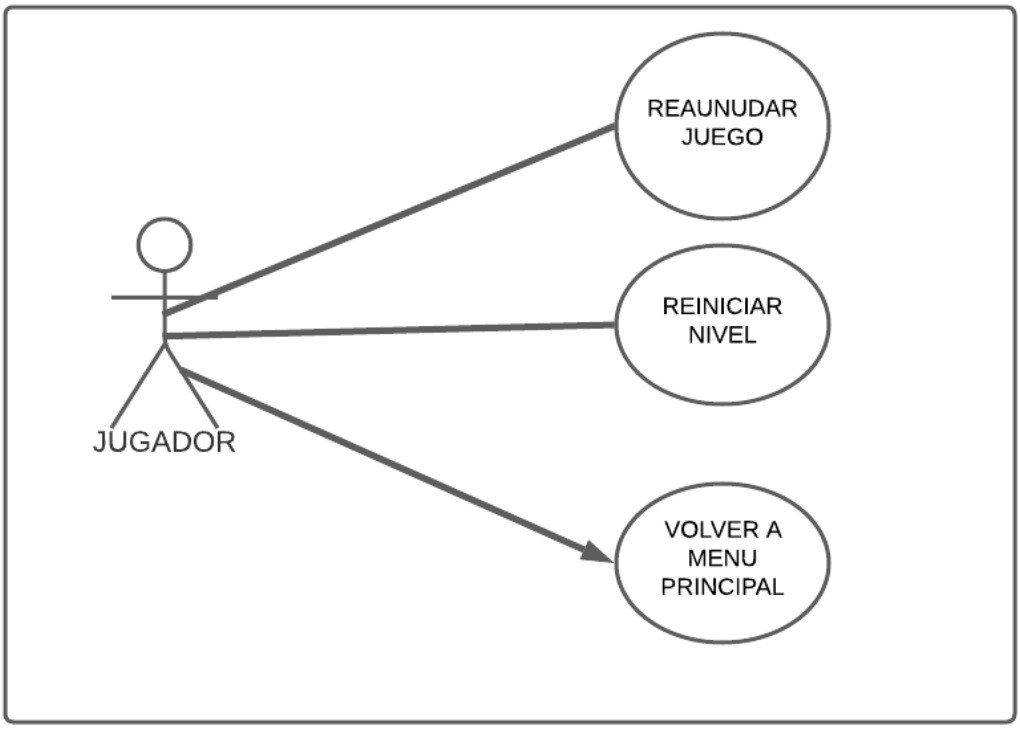
2.3.3.4 Menú selector de niveles



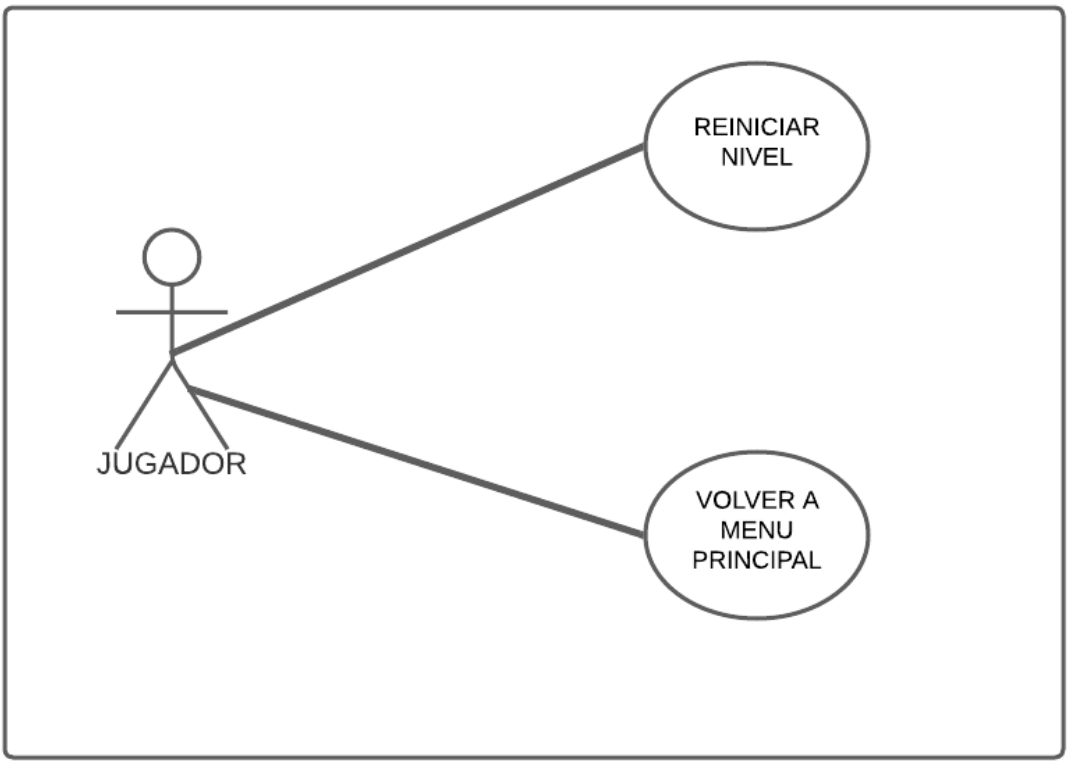
2.3.3.5 Dentro del nivel



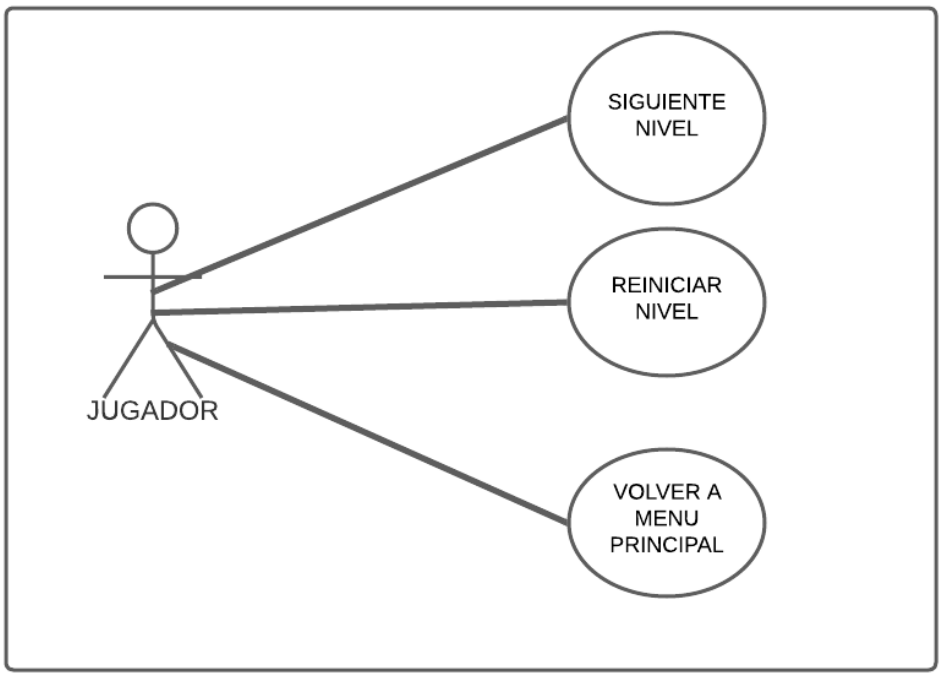
2.3.3.6 Menú de pausa



2.3.3.7 Menú de GameOver



2.3.3.8 Menú de victoria



2.3 Plataforma

Las plataformas a las que apunta el juego es la WEB (a través de ITCH.IO), y PC, específicamente WINDOWS.

2.4 Género

El género es el de estrategia, el subgénero de **tower defense** para ser específicos.

2.5 Clasificación

No tiene nada de contenido grafico por lo que podría ser de clasificación E, pero debido a su dificultad se decidió clasificarlo como E10+.

2.6 Tipo de Animación

Su tipo de animación será la 3D, aunque varios componentes de los menús contaran con animación 2D.

2.7 Equipo de Trabajo

Ingenieros de audio: Guillermo Santos

Diseñadores: Guillermo Santos

Ilustradores: Guillermo Santos

Programadores: Guillermo Santos

2.8 Historia

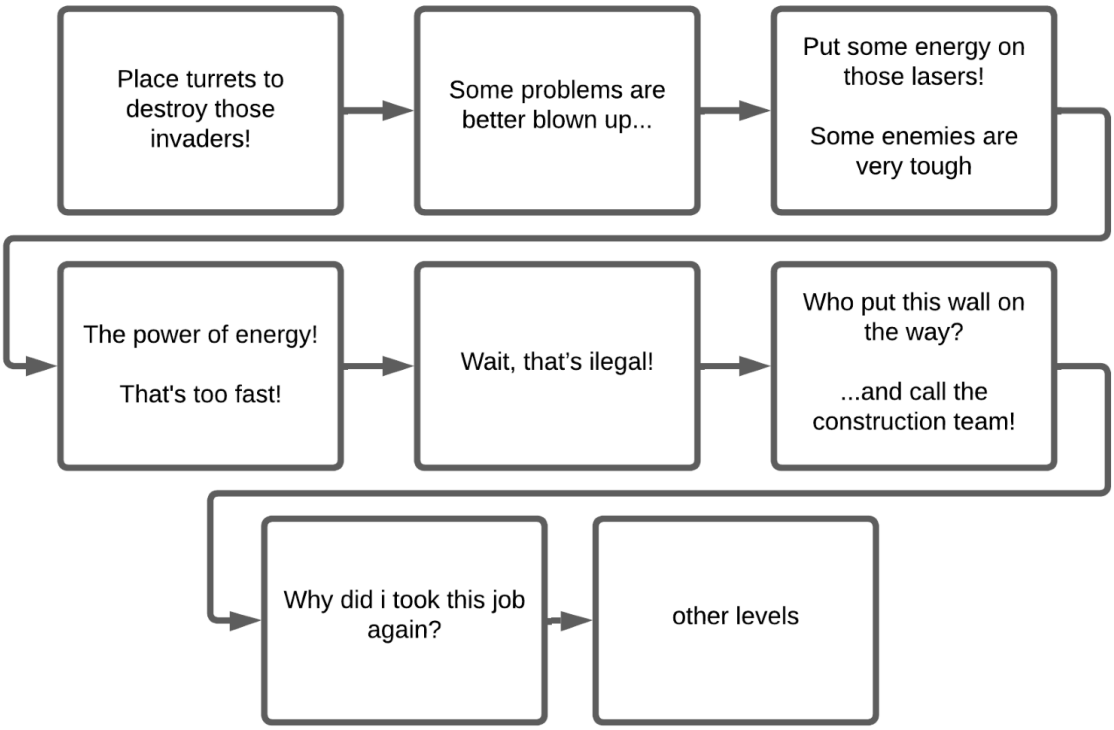
Se le ha dado el puesto de gerente en una de las operaciones mineras de SANCTUSCORP, como misión principal tendrá la de defender el CORE, estructura principal de estas operaciones, a la vez que sacar la mayor cantidad posible de minerales posibles del planeta, pero no estará solo, existen competidores que no jugaran limpio con usted, deberá desviar parte de los recursos obtenidos de la operación para defenderse de estos competidores hasta que se queden sin recursos.

2.9 Guion

El juego no tendrá guion como tal, aun así, ofrecerá varios textos en los niveles iniciales, como son:

- **Nivel 1:** “Place turrets to destroy those invaders!”
- **Nivel 2:** “Some problems are better blown up...”
- **Nivel 3:** “Put some energy on those lasers!” y “Some enemies are very tough”
- **Nivel 4:** “The power of energy!” y “That's too fast!”
- **Nivel 5:** “Wait, that’s ilegal!”
- **Nivel 6:** “Who put this wall on the way?” y “...and call the construction team!”
- **Nivel 7:** “Why did i took this job again?”

2.10 Storyboard



2.11 Personajes

2.11.1 Torreta estándar

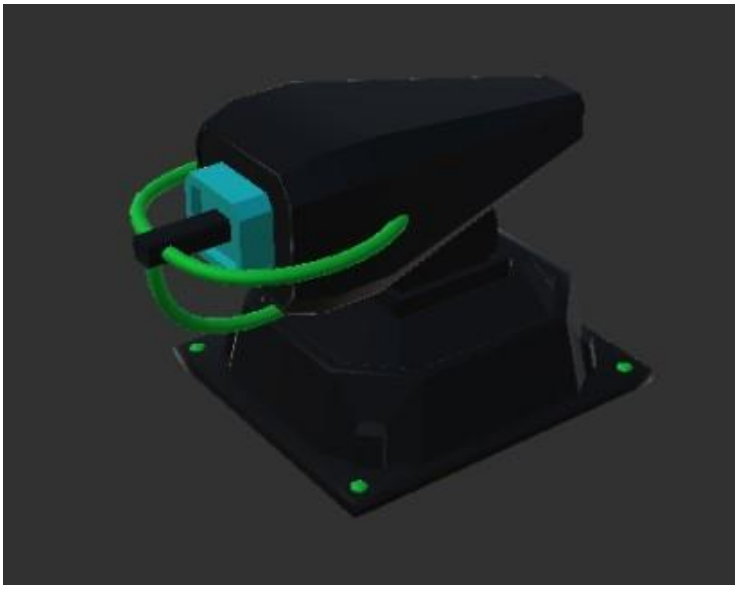
La torreta básica para defender el CORE, es la menos costosa y una con daño estándar.

2.11.2 Lanzador de misiles

Es una torreta la cual lanza misiles que infringen un gran daño en un área amplia, perfecta para quitarse grandes cantidades de enemigos de encima.

2.11.3 Rayo laser

Una torreta que realiza daño constante a un enemigo, otra de sus funciones es la de ralentizar los enemigos a los que disparan.



2.11.4 Torreta eléctrica

Una torreta que, aunque no causa daño, ralentiza a los enemigos dentro de su área, es la única torreta con 2 mejoras y cada mejora aumenta en gran medida su área de impacto.

2.11.5 Torreta minera

Una torreta que se encarga de recolectar los minerales del mapa, es una de las torretas principales a tener para pasar los niveles.

2.11.6 Torreta de energía

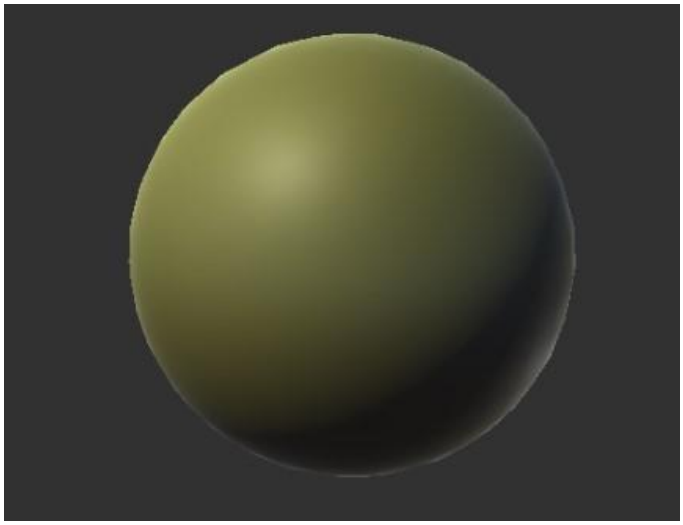
Una torreta que se encarga de recolectar los minerales energéticos utilizados para operar todas las otras torretas en el juego, sin estas, tienes un “GameOver” definitivo.

2.11.7 Torreta de reparación

Una torreta nacida de la amenaza de los tricksters, si una torreta en su área sufre daños, esta se encarga de repararla, lamentablemente una sola de estas es incapaz de superar a un trickster por lo que necesitaras otras medidas alternativas para vencerlos.

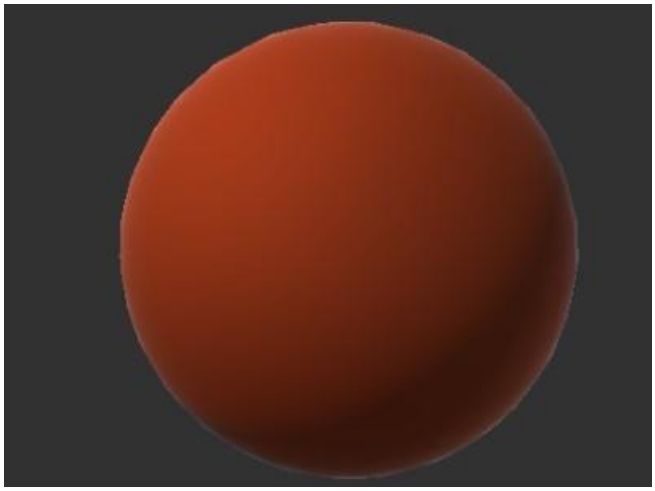
2.11.8 Rail

Es el enemigo base, y cuenta con las estadísticas estándar para los enemigos, se verá presente en todos los niveles.



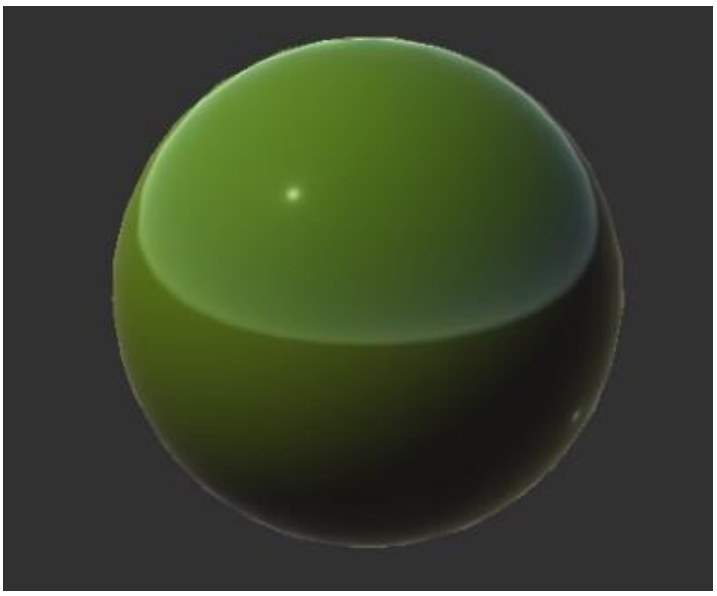
2.11.9 Coloso

Una variante más lenta, pero con mayor vida y tamaño que la variante “Rail”, su dureza hace que se requieran varias torres para abatirlos y si se juntan varios pueden ser un dolor de cabeza para el jugador.



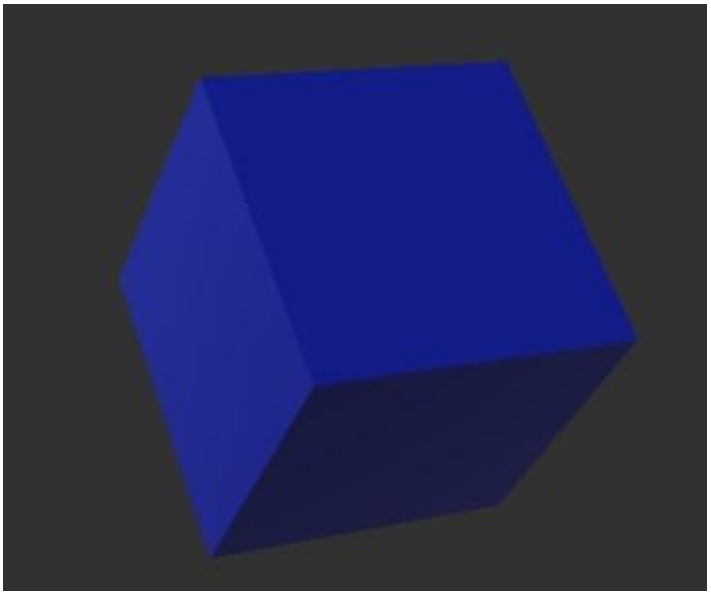
2.11.10 Flash

Una variante de “Rail” con menor vida, pero con una velocidad muy superior, dependiendo de la estructura de la defensa, estas pueden convertirse en algo mortal para el jugador.



2.11.11 Trickster

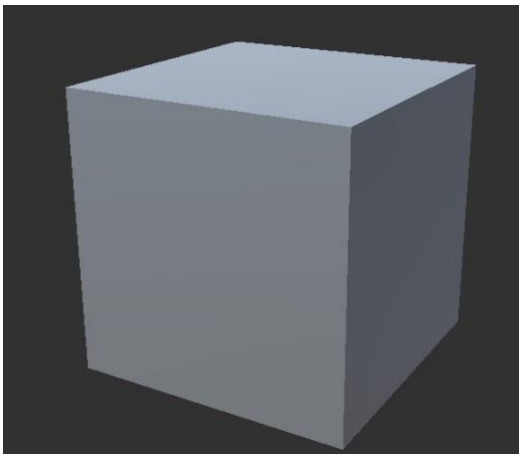
Este se puede considerar como el más peligroso entre los ya mencionados, a diferencia que el resto, comparte las funciones de ataque de las torres (En lugar de morir luego de chocar con el CORE, este le ataca a la distancia), pero lo que resulta verdadera mente problemático es que en lugar de seguir el camino junto con las demás torres, esta ataca a las torres estratégicas hasta que no queden ninguna, solo así se dirigirá al CORE, tiene ligeramente más vida y mayor velocidad que la variante "Rail" lo que sumado a su particular comportamiento lo vuelve un gran dolor de cabeza y un factor del cual hay que cuidarse en varios niveles, pues pueden arruinar completamente la partida.



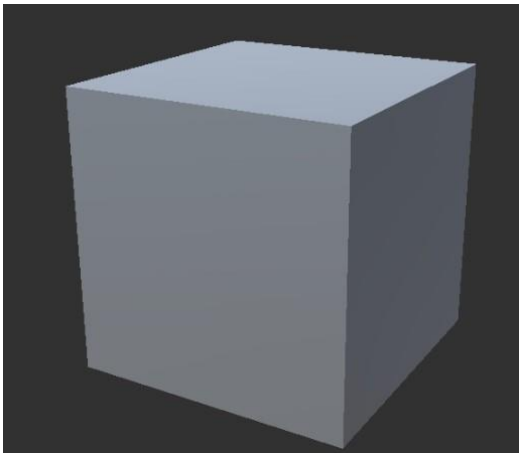
2.12 Niveles

Los niveles estarán compuestos de 8 elementos (con algunas excepciones) son los siguientes:

- **Nodo:** es el elemento principal presente absolutamente en todos los niveles, es el lugar donde puedes construir todas las torretas del juego.

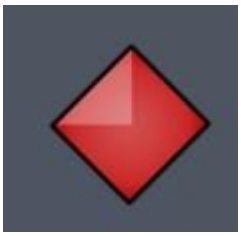


- **Obstáculo:** es un elemento extremadamente similar al nodo, lamentablemente no se puede construir en este y en varios niveles limita la visibilidad de las torretas, volviendo la defensa aún más difícil.

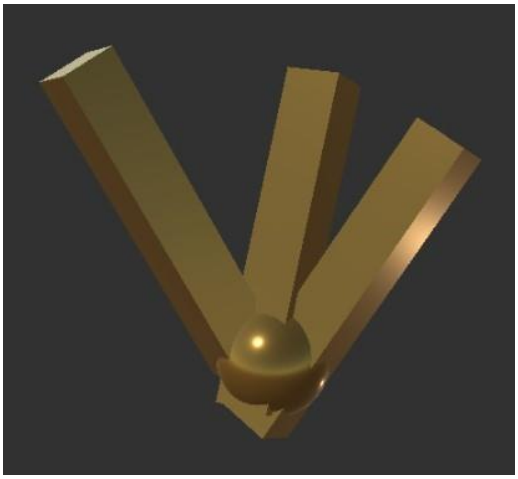


- **Caminos:** es un elemento que le indica al jugador la ruta que tomaran los enemigos.

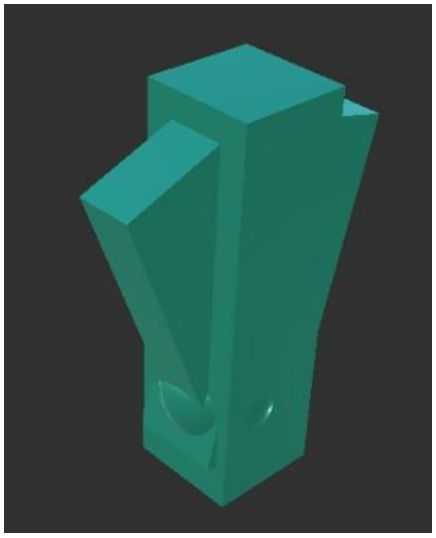
- **Puntos de camino (Waypoints):** es un elemento que le indica a los enemigos la ruta que deben seguir.



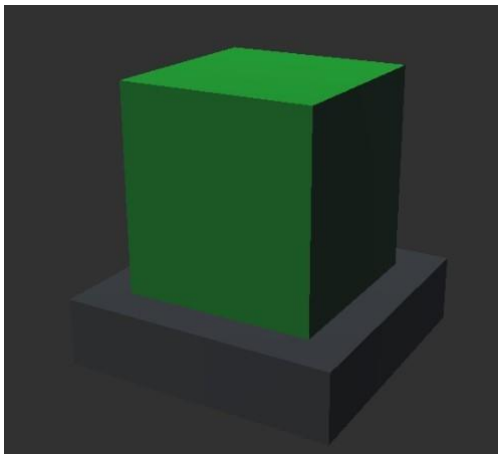
- **Mineral:** es un elemento presente en casi todos los niveles, como la historia del juego lo dice, tu objetivo es minarlo y usar este recurso para la construcción de tu defensa.



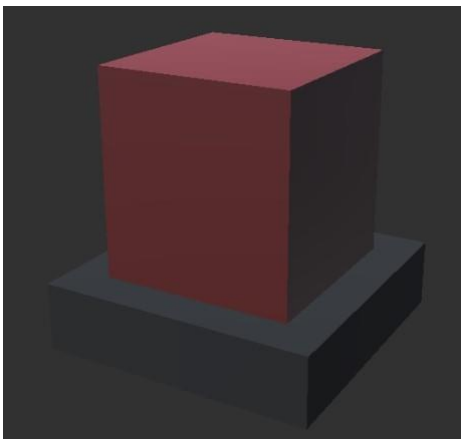
- **Mineral energético:** es el mineral mediante el cual opera toda la operación minera y defensas, está presente en todos los niveles y, aunque existe un límite de energía a almacenar, es un elemento que necesita ser constantemente minado.



- **CORE:** como ya se ha visto, es el elemento principal de la operación minera, este produce minerales y energía, es el elemento que debes defender a toda costa para el éxito de tu misión.



- **EnemyCORE:** es el CORE enemigo, lugar de donde se originan los enemigos que te atacan, aunque no puedes atacarlo al menos puedes destruir las unidades que te envían y robar los recursos que están en los restos de estos, lo cual es un beneficio para ti un costo mayor para tus enemigos.



2.13 Mecánica del Juego

El juego tiene muchas mecánicas, varias de ellas ni serán visibles para el jugador. Aquí se muestra una explicación de estas:

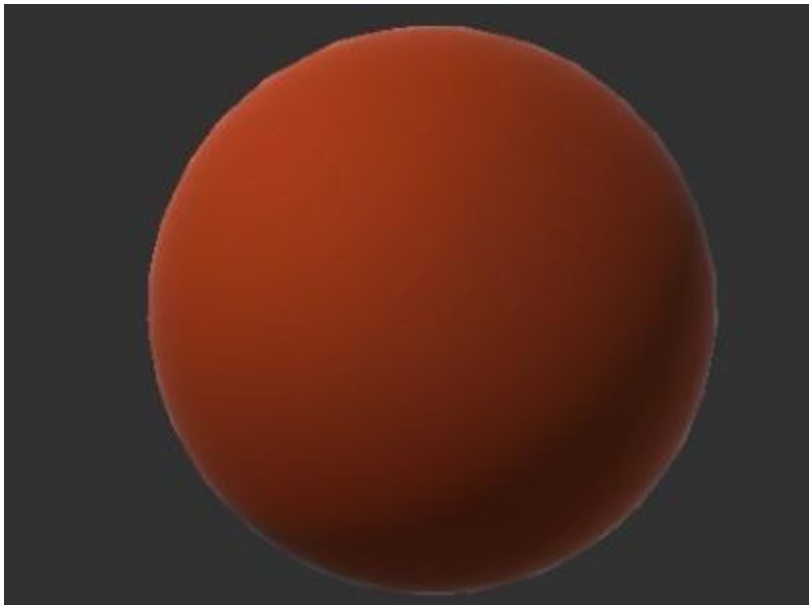
- **Colocación de torretas:** esta mecánica utiliza los nodos como elemento principal, trata de ver si es posible colocar una torreta al mirar varios elementos como la cantidad de materiales y si el nodo ya tiene otra torreta en posición.
- **Campo visual:** una mecánica usada por los enemigos y torretas por igual, aunque con distintos propósitos, una torreta ve si tiene el enemigo en la mira antes de decidir disparar, mientras tanto los enemigos miran la distancia que tienen con el suelo, nodos y obstáculos para así poder navegar por el mapa, también tenemos al enemigo trickster que utiliza ambos lados de esta mecánica para funcionar.
- **Mejora y venta de torretas:** esta mecánica nos permite, si es posible, mejorar y vender las torretas colocadas por el mapa, donde solo recibimos la mitad de su valor, una torreta mejorada tiene cambios en sus estadísticas y, en el caso de las defensivas, apartado visual. El resultado de vender una torreta mejorada es la mitad de su coste de mejora más la mitad del coste de la torreta base.
- **Oleadas:** es la mecánica que controla la aparición de enemigos en el mapa, decide los tipos de enemigo que van a aparecer, el radio de aparición, tiempo entre oleadas y numero de enemigos.
- **Producción:** mecánica principal, pues sin esta las torretas no funcionan, decide que se produce, ya sea energía, minerales o incluso cura del daño de las torretas (cortesía de los trickters), una producción deficiente tiene impactos catastróficos en el desarrollo de la partida por lo que se debe tener presente cuando estes planteando tu estrategia.

CAPITULO III: DESARROLLO

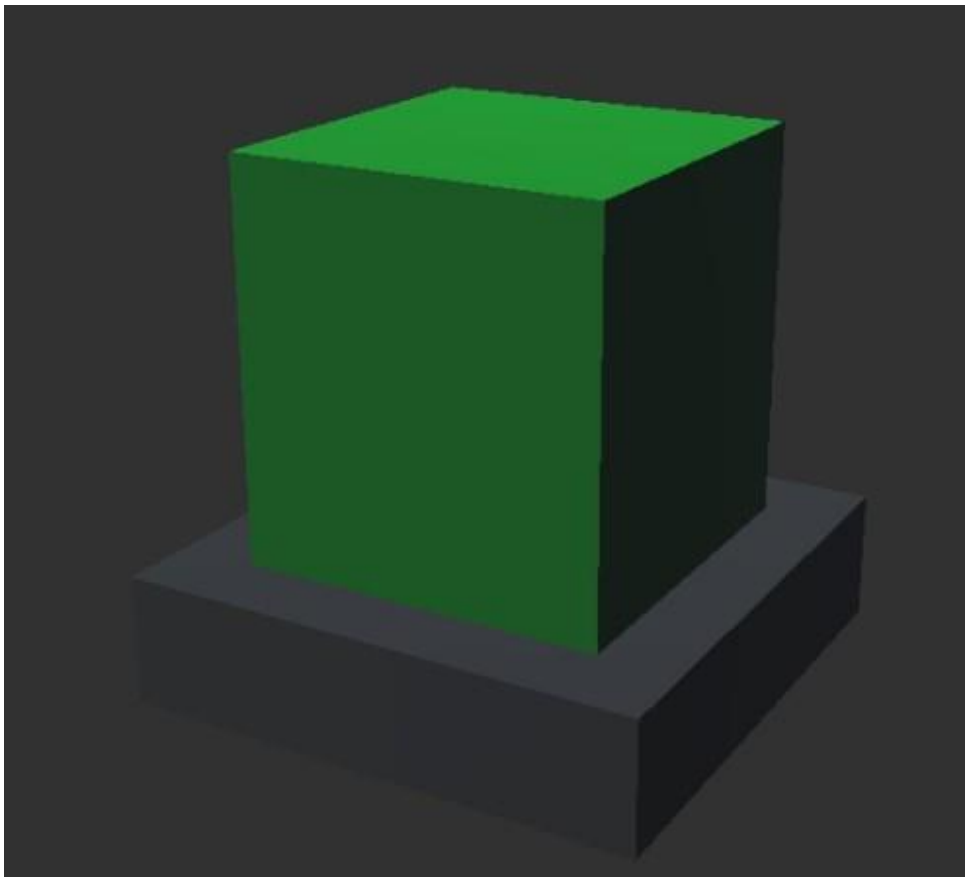
3.1. CAPTURAS DE LA APLICACION

3.1.1. PREFABS

3.1.1.1 ENEMIGO FUERTE (COLOSO)

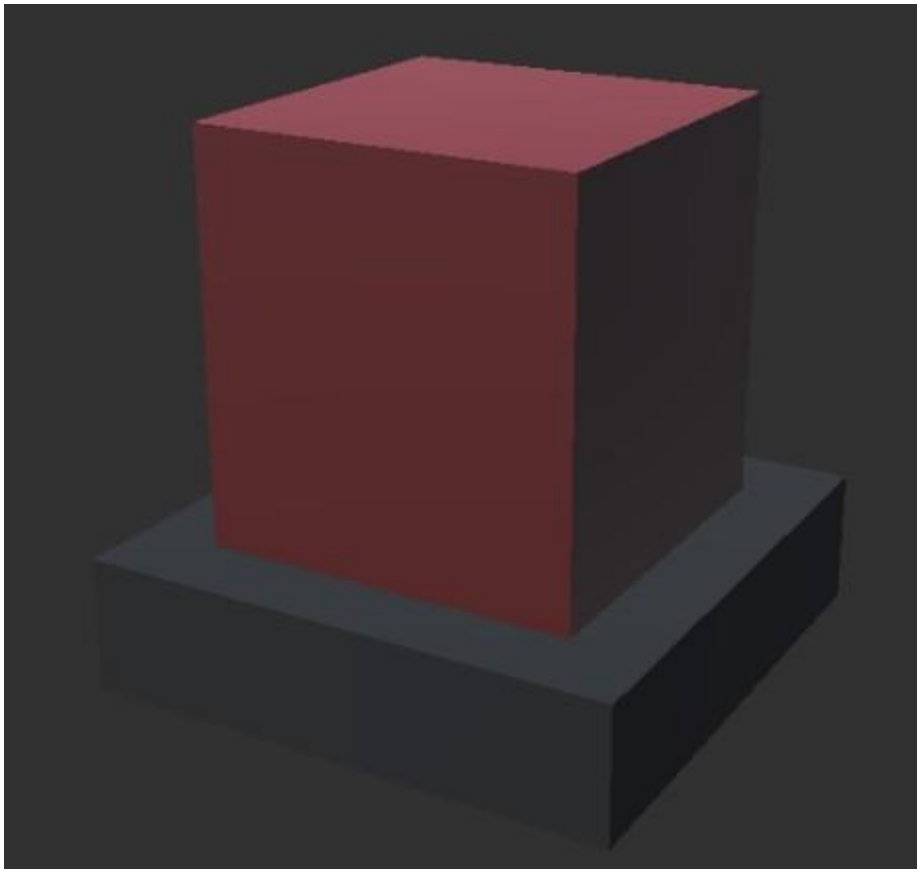


3.1.1.2. CORE

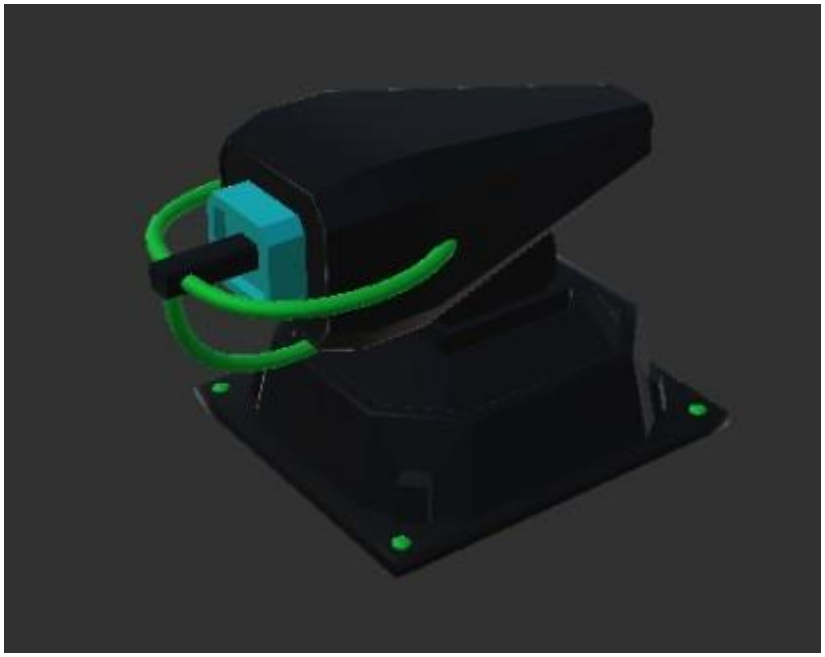


3.1.1.3. ELECTRICTURRET

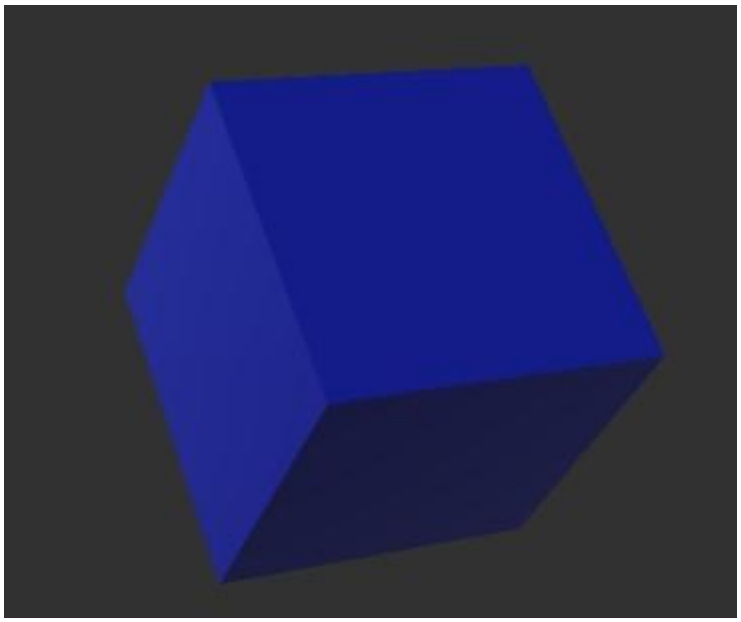
3.1.1.4. ENEMYCORE



3.1.1.5. LASERBEAMER

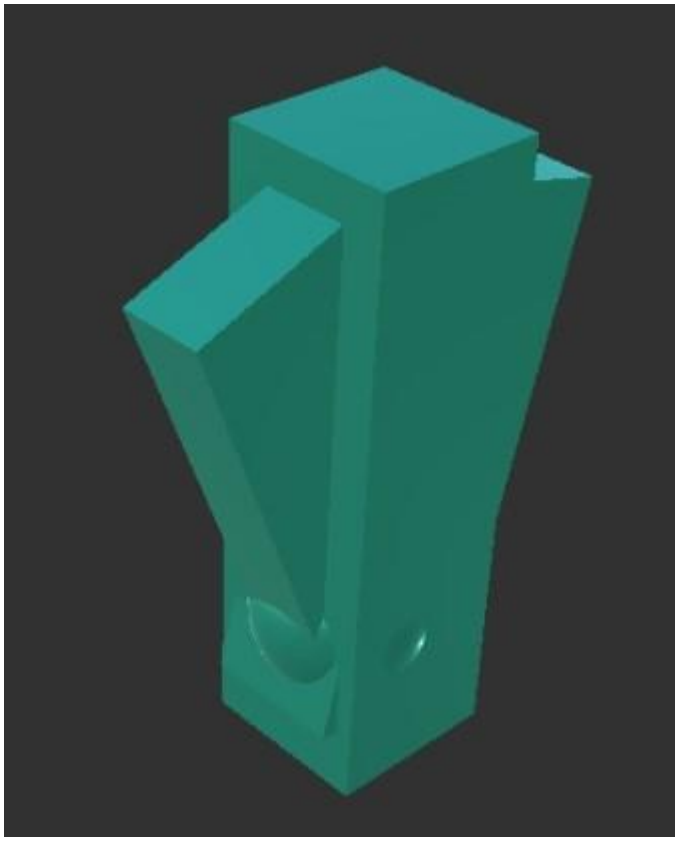


3.1.1.6. TRICKESTER

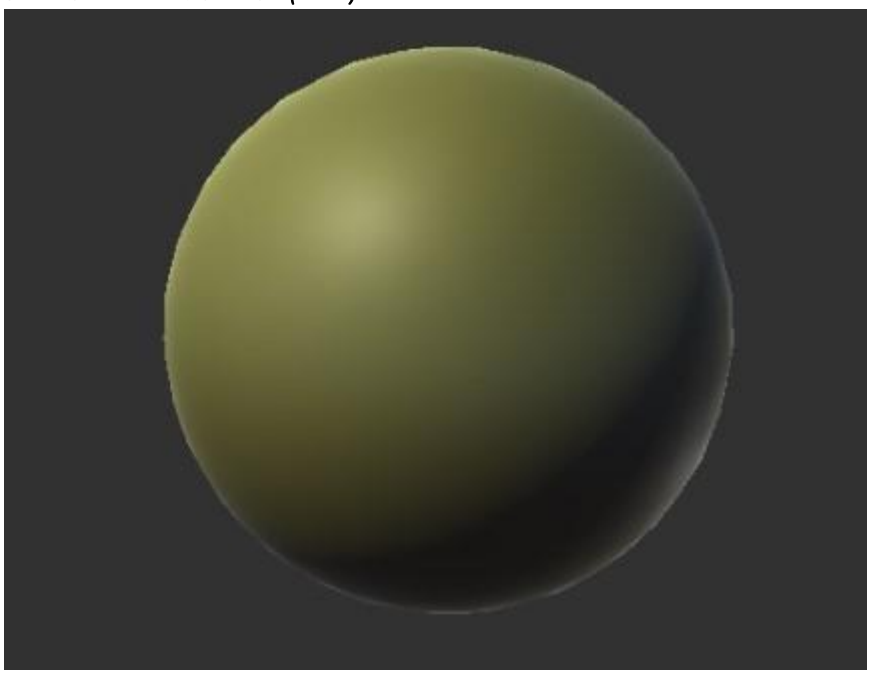


3.1.1.7 WAYPOINT

3.1.1.8. ENERGY ORE

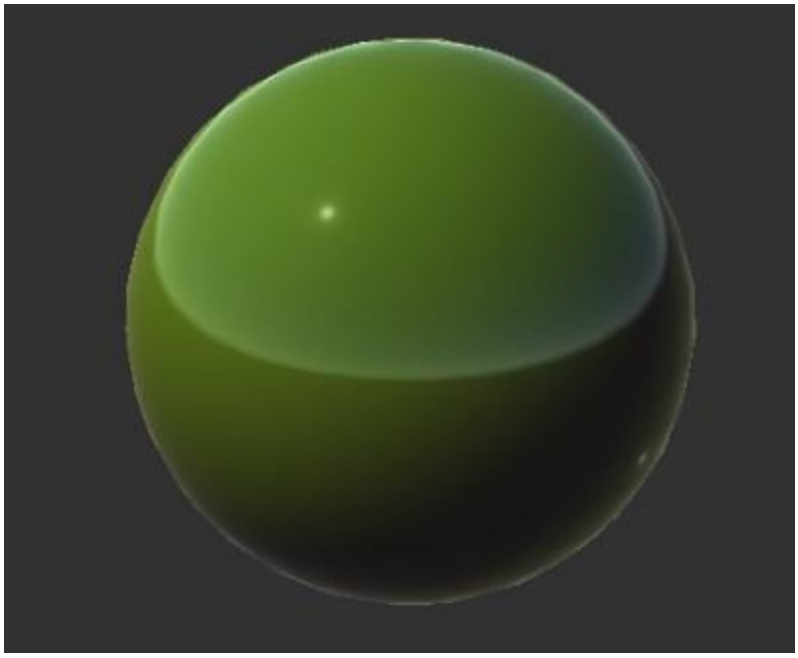


3.1.1.9. ENEMIGO BASE (RAIL)



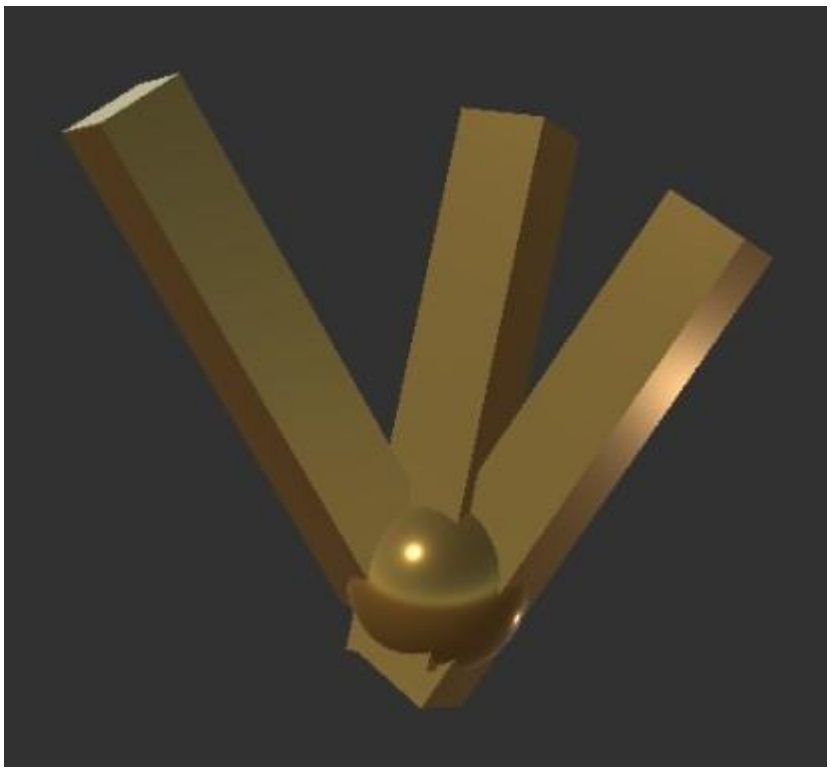
3.1.1.10. TORRETA DE ENERGIA

3.1.1.11. ENEMIGO RAPIDO (FLASH)

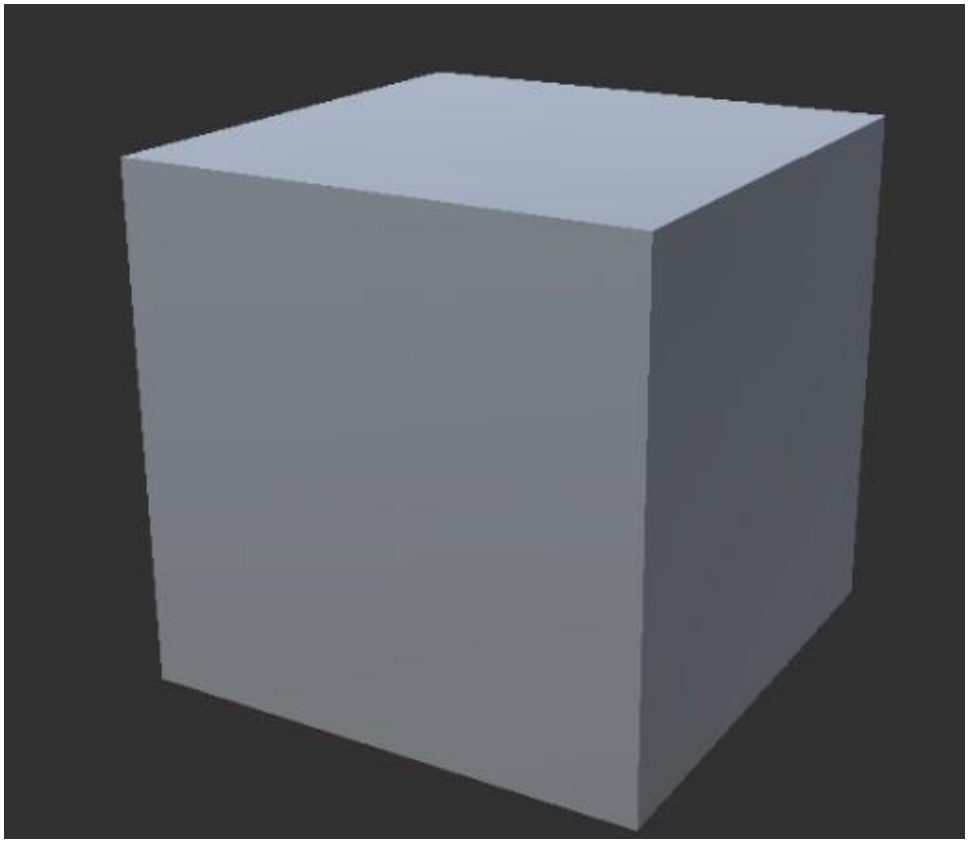


3.1.1.12. TORRETA MINERA

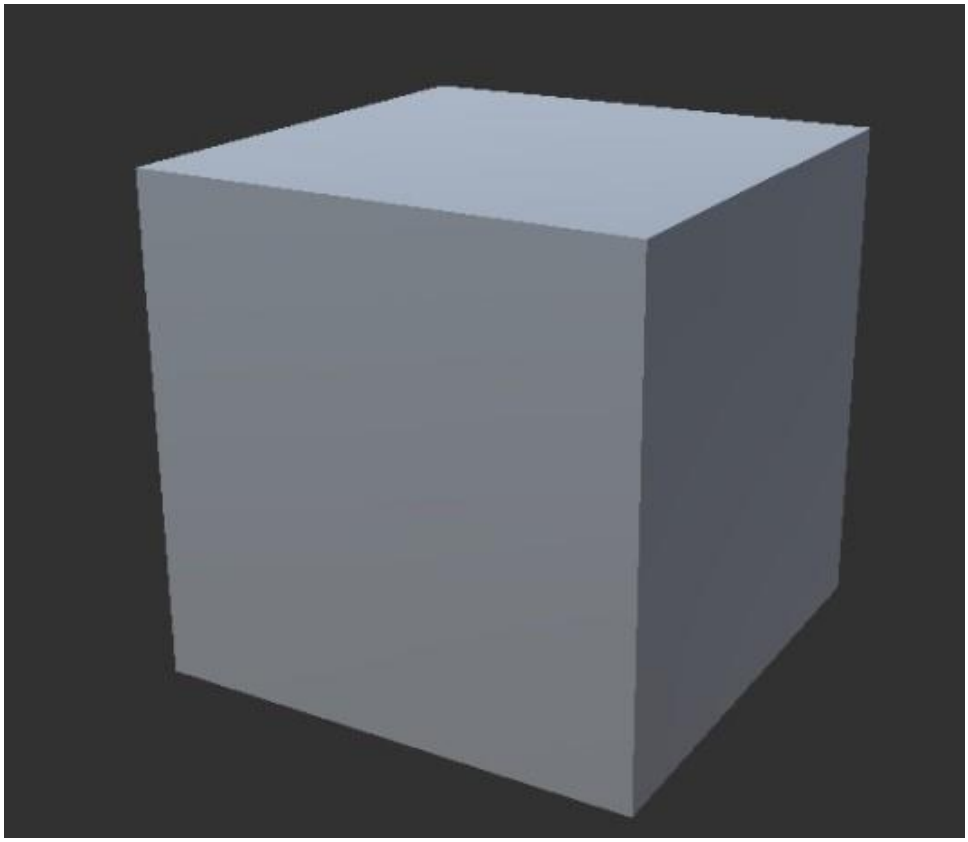
3.1.1.13. MINERAL ORE



3.1.1.14. NODO



3.1.1.15. OBSTACULO

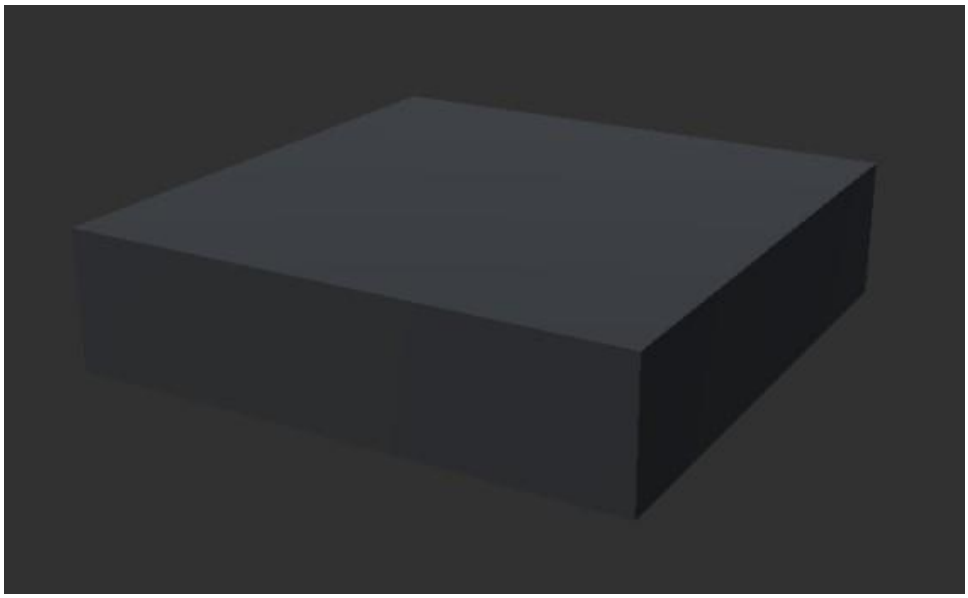


3.1.1.16. LANZA MISILES

3.1.1.17. TORRETA REPARADORA

3.1.1.18. TORRETA ESTANDAR

3.1.1.19. SUELO



3.1.2. SPRITES

3.1.2.1. TORRETA MINERA

3.1.2.2. TORRETA ESTANDAR

3.1.2.3. SIMBOLO DE ENERGIA

3.1.2.4. SIMBOLO DE MINERALES

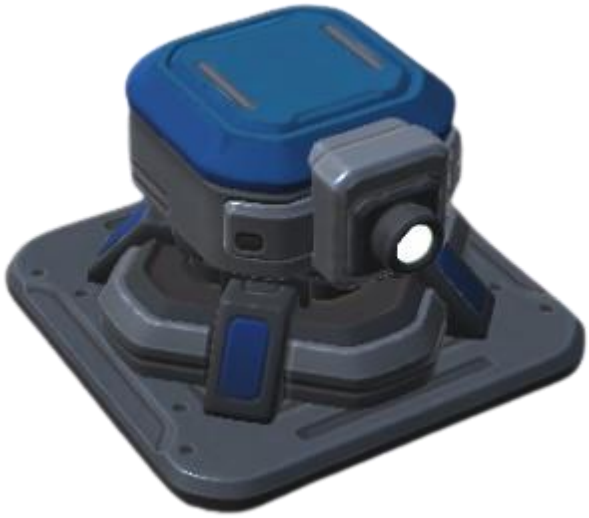
3.1.2.5 TORRETA LASER

3.1.2.6 TORRETA REPARADORA

3.1.2.7 LANZADOR DE MISILES

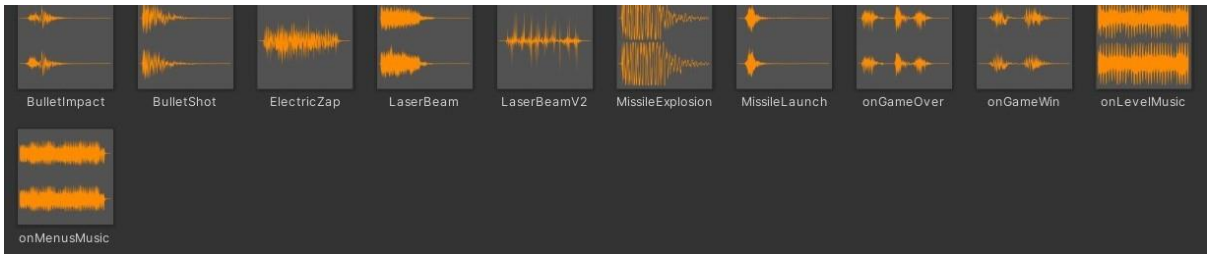
3.1.2.8 TORRETA ELECTRICA

3.1.2.9 TORRETA DE ENERGIA



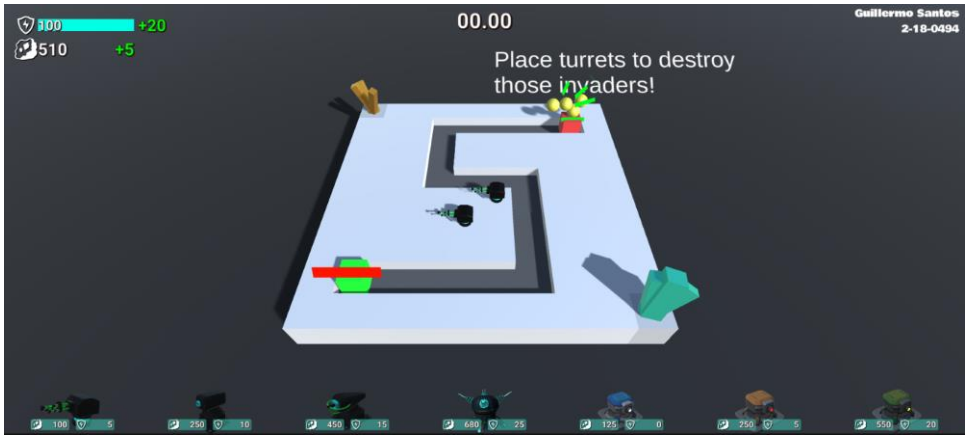
3.1.2.10 CUADRADO BLANCO

3.1.3. SONIDOS



3.1.4. NIVELES

3.1.4.1. NIVEL 1

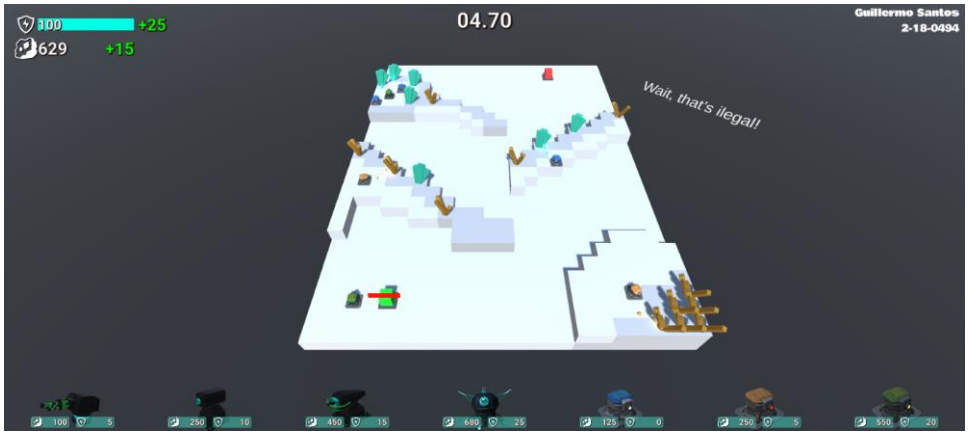


3.1.4.2. NIVEL 2

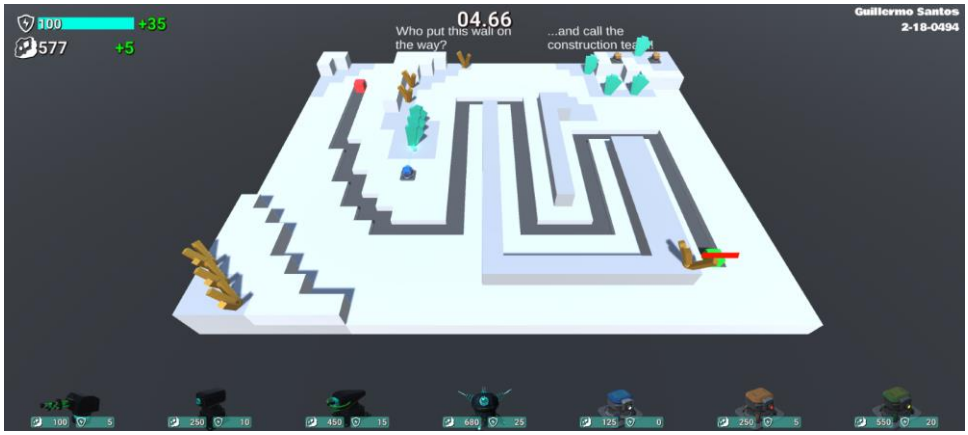
3.1.4.3. NIVEL 3

3.1.4.4. NIVEL 4

3.1.4.5. NIVEL 5



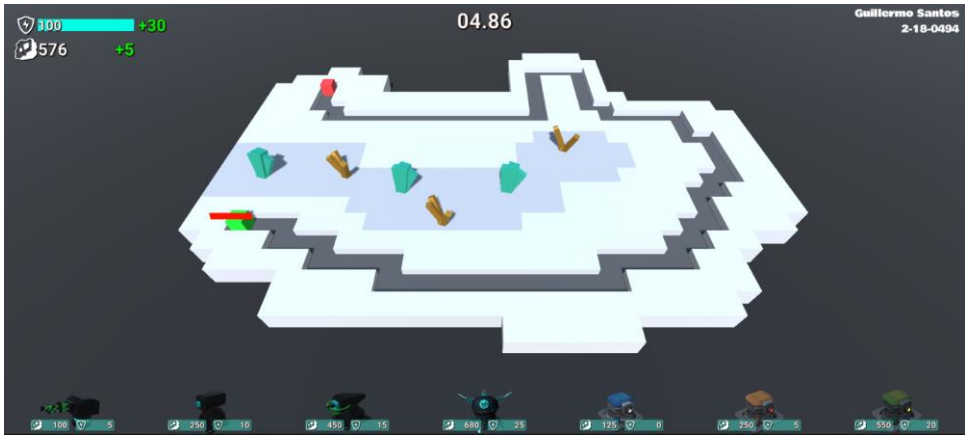
3.1.4.6. NIVEL 6



3.1.4.7. NIVEL 7

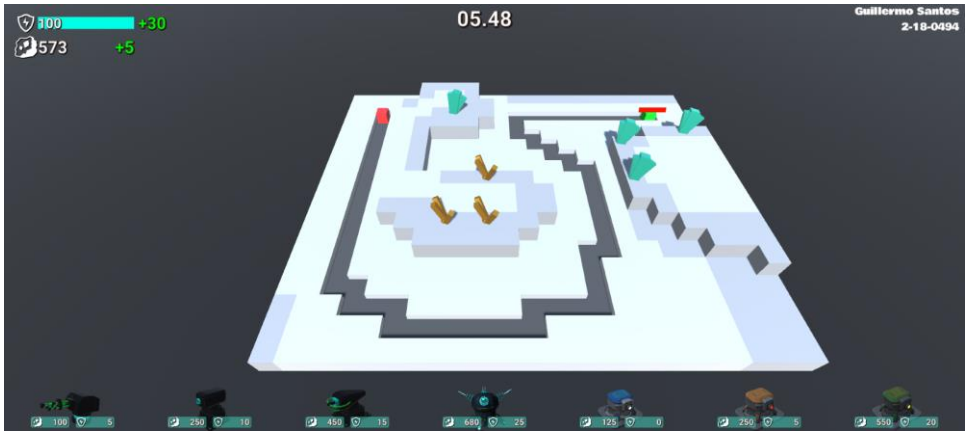
3.1.4.8. NIVEL 8

3.1.4.9. NIVEL 9



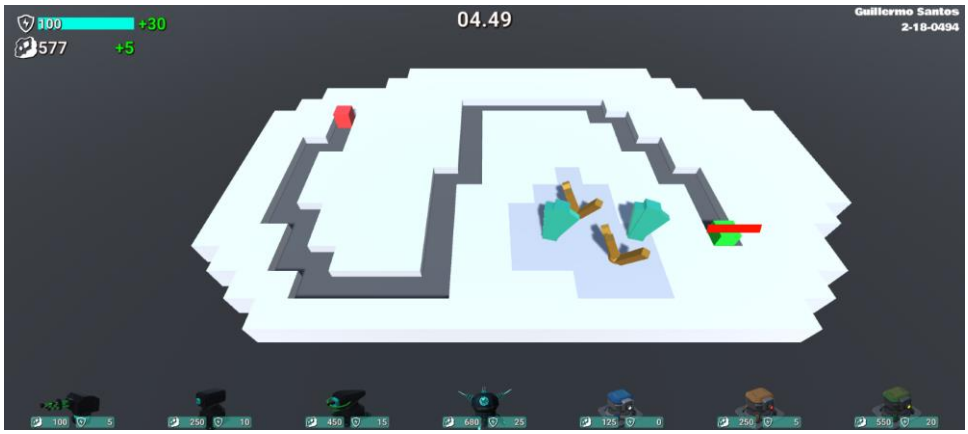
3.1.4.10. NIVEL 10

3.1.4.11. NIVEL 11

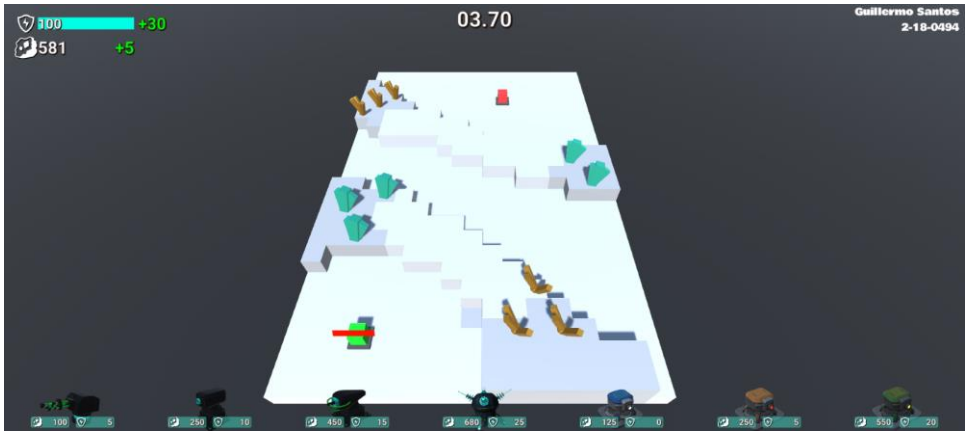


3.1.4.12. NIVEL 12

3.1.4.13. NIVEL 13



3.1.4.14. NIVEL 14



3.1.4.15. NIVEL 15

3.1.5. SCRIPTS

3.1.5.1. Enemigos

3.1.5.1.1 Ataque de enemigos

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

public class EnemyAtack : MonoBehaviour
{
    EnemyStats stats;
    Transform target;
    StructureStats targetStats;

    void Start()
    {
        stats = GetComponent<EnemyStats>();
        InvokeRepeating("UpdateTarget", 0f, 0.5f);
    }

    void UpdateTarget()
    {
        GameObject[] turrets =
GameObject.FindGameObjectsWithTag(stats.Objetive_Tag);
```

```
        if (turrets.Length <= 0)
            turrets = GameObject.FindGameObjectsWithTag(stats.CoreTag);
        float MinDistance = Mathf.Infinity;
        float distanceToTurret;
        GameObject nearestTurret = null;
        foreach (GameObject turret in turrets)
        {
            distanceToTurret = Vector3.Distance(transform.position,
turret.transform.position);
            if (distanceToTurret < MinDistance)
            {
                MinDistance = distanceToTurret;
                nearestTurret = turret;
            }
        }

        if (nearestTurret != null && MinDistance <= stats.range)
        {
            target = nearestTurret.transform;
            stats.canMove = false;
            stats.canAtack = true;
            targetStats = nearestTurret.GetComponent<StructureStats>();
        }
        else
        {
            target = null;
            stats.canAtack = false;
            stats.canMove = true;
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (!stats.canMove && stats.canAtack)
        {
            Utility.LookOnTarget(transform,(target.position -
transform.position).normalized,stats.speed);
            if (stats.fireCountdown <= 0)
            {
                Shoot();
                stats.fireCountdown = 1f / stats.fireRate;
            }

            stats.fireCountdown -= Time.deltaTime;
        }
    }


    void Shoot()
```

```
    {
        GameObject Bullet_O = Instantiate(stats.bullet,
stats.firePoint.position, stats.firePoint.rotation);
        EnemyBullet bullet_sc = Bullet_O.GetComponent<EnemyBullet>();
        if (bullet_sc != null)
            bullet_sc.Seek(target);
    }
}
```

3.1.5.1.2 Bala de enemigos

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

public class EnemyBullet : MonoBehaviour
{
      private Transform target;

      public int damage = 15;
      public float speed = 10f;
      public float explosionRadius = 0f;
      public GameObject ImpactEffect;

      private AudioSource impactSource;

      public void Seek(Transform _target)
      {
            target = _target;
      }

    private void Start()
    {
            impactSource = GetComponent<AudioSource>();
      }

    // Update is called once per frame
    void Update()
      {
            if (target == null)
            {
                  Destroy(gameObject);
                  return;
            }

            Vector3 dir = target.position - transform.position;

            float distanceFPS = speed * Time.deltaTime;

            if (dir.magnitude <= distanceFPS)
            {
```

```
                HitTarget();
                return;
            }

            transform.Translate(dir.normalized * distanceFPS,
Space.World);
            transform.LookAt(target);
      }

      void HitTarget()
      {
            impactSource.Play();
            GameObject effect = (GameObject)Instantiate(ImpactEffect,
transform.position, transform.rotation);
            effect.transform.GetComponent<ParticleSystem>().Play();
            Destroy(effect, 5f);
            if (explosionRadius > 0f)
            {
                  Explode();
            }
            else
            {
                  Damage(target);
            }
            Destroy(gameObject);
      }
      void Damage(Transform enemy)
      {
            StructureStats e = enemy.GetComponent<StructureStats>();
            if (e != null)
            {
                  e.TakeDamage(damage);
            }
      }

      void Explode()
      {
            Collider[] colliders =
Physics.OverlapSphere(transform.position, explosionRadius);
            foreach (Collider collider in colliders)
            {
                  if (collider.tag == "StrategicTurret")
                  {
                        Damage(collider.transform);
                  }
            }
      }

      private void OnDrawGizmosSelected()
      {
```

```
            Gizmos.color = Color.red;
            Gizmos.DrawWireSphere(transform.position, explosionRadius);
      }
}
```

3.1.5.1.3 Movimiento de enemigos

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

public enum Targets
{
    WayPoints,
    StrategicTurrets
}

[RequireComponent(typeof(EnemyStats))]
public class EnemyMovement : MonoBehaviour
{
    private Transform target;

    private int wavepointIndex;

    private EnemyStats stats;
    private Vector3 Direction;
    // Start is called before the first frame update
    void Start()
    {
        stats = GetComponent<EnemyStats>();
        if(stats.Target == Targets.WayPoints)
        {
            resetTarget();
        }
        else if(stats.Target == Targets.StrategicTurrets)
        {
            InvokeRepeating("SearchForTurretTargets", 0f, 0.5f);
        }
    }

    void resetTarget()
    {
        wavepointIndex = 0;
        target = WayPoints.points[wavepointIndex];
        setDirection();
    }

    void SearchForTurretTargets()
    {
        GameObject[] turrets =
GameObject.FindGameObjectsWithTag(stats.Objetive_Tag);
```

```
        if(turrets.Length <= 0)
            turrets = GameObject.FindGameObjectsWithTag(stats.CoreTag);
        float MinDistance = Mathf.Infinity;
        float distanceToTurret;
        GameObject nearestTurret = null;
        foreach (GameObject turret in turrets)
        {
            distanceToTurret = Vector3.Distance(transform.position,
turret.transform.position);
            if (distanceToTurret < MinDistance)
            {
                MinDistance = distanceToTurret;
                nearestTurret = turret;
            }
        }

        if (nearestTurret != null)
        {
            target = nearestTurret.transform;
        }
        else
        {
            stats.canMove = false;
            target = null;
        }

        if(stats.canMove)
            setDirection();

    }

    void Update()
    {
        HeightControl();

        if (stats.canMove)
        {
            Move();
            Utility.LookOnTarget(transform, Direction, stats.speed);

            if (stats.Target == Targets.WayPoints)
            {
                if (Vector2.Distance(new Vector2(transform.position.x,
transform.position.z), new Vector2(target.position.x, target.position.z))
<= 0.1f)
                {
                    NextAction();
                }
            }
        }
```

```
        else
        {
            Stop();
        }


        stats.speed = stats.startSpeed;
    }

    void NextAction()
    {
        StructureStats targetstats;
        if (target.TryGetComponent<StructureStats>(out targetstats))
        {
            EndPath(targetstats);
            return;
        }
        getNextWaypoint();
    }

    void HeightControl()
    {
        DownControl();
        FwdControl();

    }

    void DownControl()
    {
        Vector3 down = transform.TransformDirection(Vector3.down);
        RaycastHit Downhit;
        if (Physics.Raycast(transform.position, down * 2, out Downhit, 2,
stats.moveOverMask))
        {
            if (Downhit.collider != null)
            {
                float Distance =
Vector3.Distance(Downhit.collider.transform.position, transform.position);
                if (Distance < stats.floorDistance)
                {
                    Direction.y = Direction.y + stats.floorDistance;
                    MoveUp(new Vector3(transform.position.x, Direction.y +
stats.floorDistance, transform.position.z));
                    Debug.DrawRay(transform.position, down *
Downhit.distance, Color.yellow);
                }
                else
                {
                    Stop();
                    setDirection();
```

```
                    Debug.DrawRay(transform.position, down *
Downhit.distance, Color.green);
                }
            }
            else
            {
                Debug.DrawRay(transform.position, down *
stats.floorDistance, Color.red);
            }

        }
        else
        {
            Debug.DrawRay(transform.position, down * stats.floorDistance,
Color.red);
        }
    }

    void FwdControl()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);
        RaycastHit Fwdhit;

        if (Physics.Raycast(transform.position, fwd * 2, out Fwdhit, 2,
stats.moveOverMask))
        {
            if (Fwdhit.collider != null)
            {
                Direction.y = Direction.y + stats.floorDistance;
                MoveUp(new Vector3(transform.position.x, Direction.y +
stats.floorDistance, transform.position.z));
                Debug.DrawRay(transform.position, fwd * Fwdhit.distance,
Color.yellow);
            }
            else
            {
                Stop();
                setDirection();
                Debug.DrawRay(transform.position, fwd * Fwdhit.distance,
Color.green);
            }

        }
        else
        {
            Debug.DrawRay(transform.position, fwd * stats.floorDistance,
Color.red);
        }
    }
```

```
    void Move()
    {
        transform.Translate(Direction.normalized * stats.speed *
Time.deltaTime, Space.World);
    }

    void MoveUp(Vector3 pos)
    {
        transform.Translate(pos.normalized * stats.speed * Time.deltaTime,
Space.World);
    }

    private void Stop()
    {
        transform.Translate(Direction.normalized * 0, Space.World);
    }

    void setDirection()
    {
        if (target == null)
        {
            Stop();
            return;
        }
        Direction = (target.position - transform.position).normalized;
    }

    void getNextWaypoint()
    {
        //as this method is called when the waypoint is not a core or
structure, this if
        //prevent any errors by restaring the waypoints so that the enemy
move on a loop when there is not a core
        //on the way
        if (wavepointIndex >= WayPoints.points.Count - 1)
        {
            resetTarget();
            return;
        }

        wavepointIndex++;
        target = WayPoints.points[wavepointIndex];
        setDirection();
    }

    void EndPath(StructureStats target)
    {
        target.TakeDamage(stats.impactDamage);
        WaveSpawner.EnemiesAlive--;
        Destroy(gameObject);
```

```
        // This line is for testing. it make the enemy to restart from the
firstwaypoint
        //resetTarget();
    }

}
```

3.1.5.1.4 Estadísticas de enemigo

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class EnemyStats : MonoBehaviour
{
    [Header("Enemy Stats")]
    public float startHealth = 100f;
    public int MoneyDrop = 50;
    public float startSpeed = 2f;
    public int impactDamage = 5;

    [Header("Enemy Movement")]
    [Range(0f, 2f)]
    public float floorDistance = 1f;
    public LayerMask moveOverMask;
    public Targets Target = Targets.WayPoints;
    public bool canMove = true;
    public bool canAtack = false;
    [Header("Enemy Atack")]
    public Transform firePoint;
    public LayerMask targetMask;
    public string Objetive_Tag;
    public Transform lastObjetive;
    public GameObject bullet;
    [Range(0f,10f)]
    public float range;
    public float fireRate = 1f;
    public float fireCountdown = 1f;

    [Header("Unity Objects")]
    public GameObject DeadEffect;
    public Image healthBar;

    [HideInInspector]
    public float health;
    [HideInInspector]
    public float speed;
    [HideInInspector]
    public string CoreTag;
    private bool isDead = false;
```

```
    private void Start()
    {
        speed = startSpeed;
        health = startHealth;
        if(lastObjetive != null)
            CoreTag = lastObjetive.tag;
    }

    public void TakeDamage(float amount)
    {
        health -= amount;
        healthBar.fillAmount = health / startHealth;
        if (health <= 0 && !isDead)
        {
            Die();
        }
    }

    public void Slow(float pct)
    {
        speed = startSpeed * (1f - pct);
    }

    void Die()
    {
        isDead = true;

        GameObject effect =
Instantiate(DeadEffect,transform.position,Quaternion.identity);
        Destroy(effect,1.5f);
        PlayerStats.materials += MoneyDrop;
        WaveSpawner.EnemiesAlive--;
        Destroy(gameObject);
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, range);
    }
}
```

3.1.5.2. Managers

3.1.5.2.1 Manager de construcción

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

```
public class BuildManager : MonoBehaviour
{
    private TurretBlueprint turretToBuild;
    private Nodo selectedNode;

    public static BuildManager instance;
    public GameObject BuildEffect;
    public GameObject SellEffect;
    public NodoUI nodoUI;
    public bool CanBuild { get { return turretToBuild != null; } }
    public bool HasMoney { get { return PlayerStats.materials >=
turretToBuild.materialCost; } }
    void Awake()
    {
        if (instance != null)
        {
            Debug.LogError("More than one BuildManager in scene!");
            return;
        }
        instance = this;
    }
    public void SelectNode(Nodo nodo)
    {
        if (selectedNode == nodo)
        {
            DeselectNode();
            return;
        }
        selectedNode = nodo;
        turretToBuild = null;
        nodoUI.SetTarget(nodo);
    }

    public void DeselectNode()
    {
        selectedNode = null;
        nodoUI.Hide();
    }

    public void SelectTurretToBuild(TurretBlueprint turret)
    {
        turretToBuild = turret;
        DeselectNode();
    }

    public TurretBlueprint GetTurretToBuild()
    {
        return turretToBuild;
    }
```

```
}
```

3.1.5.2.2 Controlador de cámara

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.InputSystem;

public class CameraController : MonoBehaviour
{
    public float panSpeed = 30f;
    public float panBorderScreen = 30f;
    public float ScroolSpeed = 0.5f;
    public LayerMask NodeMask;

    [Header("Limits")]
    public float minY = 3.5f;
    public float maxY = 30f;

    private PlayerInput playerInput;
    private Vector2 mousePos;
    private RaycastHit lastHit;
    private IMouse lastMouseMove;

    private void Awake()
    {
        playerInput = new PlayerInput();
    }

    private void OnEnable()
    {
        playerInput.Enable();
        playerInput.Camera.MousePosition.performed += Move;
        playerInput.Camera.Scrolling.performed += scroll;
        playerInput.GamePlay.MouseLeftClick.performed += MouseLeftClick;
    }

    private void OnDisable()
    {
        playerInput.Disable();
        playerInput.Camera.MousePosition.performed -= Move;
        playerInput.Camera.Scrolling.performed -= scroll;
        playerInput.GamePlay.MouseLeftClick.performed -= MouseLeftClick;
    }

    // Update is called once per frame
    void Update()
    {
```

```
        if (GameManager.isGameOver)
        {
            this.enabled = false;
            return;
        }

        if (playerInput.Camera.ForwardMove.IsPressed() || mousePos.y >=
Screen.height - panBorderScreen)
        {
            transform.Translate(Vector3.forward * panSpeed *
Time.deltaTime, Space.World);
        }

        if (playerInput.Camera.BackwardMoce.IsPressed() || mousePos.y <=
panBorderScreen)
        {
            transform.Translate(Vector3.back * panSpeed * Time.deltaTime,
Space.World);
        }

        if (playerInput.Camera.RightMove.IsPressed() || mousePos.x >=
Screen.width - panBorderScreen)
        {
            transform.Translate(Vector3.right * panSpeed * Time.deltaTime,
Space.World);
        }

        if (playerInput.Camera.LeftMove.IsPressed() || mousePos.x <=
panBorderScreen)
        {
            transform.Translate(Vector3.left * panSpeed * Time.deltaTime,
Space.World);
        }
        /*
        */
    }

    void Move(InputAction.CallbackContext ctx)
    {
        mousePos = ctx.ReadValue<Vector2>();
        if (ctx.phase == InputActionPhase.Performed &&
            Physics.Raycast(
                Camera.main.ScreenPointToRay(mousePos),
                out RaycastHit hit, NodeMask))
        {
            IMouse mouseMove =
hit.collider.gameObject.GetComponent<IMouse>();
            if (!hit.collider.Equals(lastHit.collider))
            {
                mouseMove?.OnMouseEnter();
```

```
                lastMouseMove?.OnMouseExit();
            }

            lastHit = hit;
            lastMouseMove = mouseMove;
        }
    }
    void scroll(InputAction.CallbackContext ctx)
    {
        //float scroll = Input.GetAxis("Mouse ScrollWheel");
        float scroll = ctx.ReadValue<float>();
        Vector3 pos = transform.position;

        pos.y -= scroll * 1000 * ScroolSpeed * Time.deltaTime;
        pos.y = Mathf.Clamp(pos.y, minY, maxY);
        transform.position = pos;
    }

    public void MouseLeftClick(InputAction.CallbackContext ctx)
    {
        if (ctx.phase == InputActionPhase.Performed &&
            Physics.Raycast(
                Camera.main.ScreenPointToRay(
                    Mouse.current.position.ReadValue()),
                out RaycastHit hit, NodeMask))
        {
            IMouse mouseDown =
hit.collider.gameObject.GetComponent<IMouse>();

            mouseDown?.OnMouseDown();
        }
    }

}
```

3.1.5.2.3 Completacion de nivel

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class CompleteLevel : MonoBehaviour
{
    public SceneFader sceneFader;
    public string menuSceneName = "MainMenu";
    public string nextLevl = "Lv2";
    public int levelToUnlock = 2;

    public void Continue()
    {
```

```
        PlayerPrefs.SetInt("levelReached", levelToUnlock);
        sceneFader.FadeTo(nextLevl);
    }

    public void Menu()
    {
        sceneFader.FadeTo(menuSceneName);
    }

}
```

3.1.5.2.4 GameManager

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

public class GameManager : MonoBehaviour
{

    public GameObject gameOverUI;
    public AudioClip gameOverClip;
    public GameObject completeLevelUI;
    public AudioClip completeLeveClip;
    public GameObject CORE;
    public SceneFader sceneFader;
    public bool Editor = false;
    [HideInInspector]
    public static bool isGameOver;
    public static bool isEditor;
    private AudioSource audioSource;
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        isGameOver = false;
        isEditor = Editor;
    }

    // Update is called once per frame
    void Update()
    {
        if (isGameOver)
            return;
        if(CORE == null)
        {
            EndGame();
            return;
        }
    }

    void EndGame()
```

```
    {
        if (isGameOver)
            return;
        audioSource.loop = false;
        audioSource.clip = gameOverClip;
        audioSource.Play();
        isGameOver = true;
        gameOverUI.SetActive(true);
    }

    public void WinLevel()
    {
        if(isGameOver)
            return;
        audioSource.loop = false;
        audioSource.clip = completeLeveClip;
        audioSource.Play();
        isGameOver = true;
        completeLevelUI.SetActive(true);
    }
}
```

3.1.5.2.5 GameOver

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameOver : MonoBehaviour
{

    public SceneFader sceneFader;
    public string menuSceneName = "MainMenu";



    public void Retry()
    {
        sceneFader.FadeTo(SceneManager.GetActiveScene().name);
    }

    public void Menu()
    {
        sceneFader.FadeTo(menuSceneName);
    }
}
```

3.1.5.2.6 Selector de niveles

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.UI;

public class LevelSelector : MonoBehaviour
{
    public SceneFader sceneFader;

    public Button[] levelButtons;

    void Start()
    {
        int levelReached = PlayerPrefs.GetInt("levelReached", 1);

        for (int i = 0; i < levelButtons.Length; i++)
        {
            if( (i + 1) > levelReached)
            {
                levelButtons[i].interactable = false;
            }
            else
            {
                levelButtons[i].interactable= true;
            }

        }

    }

    public void Select(string levelname)
    {
        sceneFader.FadeTo(levelname);
    }
}
```

3.1.5.2.7 Menu principal

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
    public string levelToLoad = "MainLevel";
    public SceneFader sceneFader;
    public void Play()
    {
        sceneFader.FadeTo(levelToLoad);
    }
    public void Quit()
    {
```

```
        Application.Quit();
    }
}
```

3.1.5.2.8 Menu de pausa

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class PauseMenu : MonoBehaviour
{
    public GameObject ui;
    public SceneFader sceneFader;
    public string menuSceneName = "MainMenu";

    private PlayerInput playerInput;

    private void Awake()
    {
        playerInput = new PlayerInput();
    }

    private void OnEnable()
    {
        playerInput.Enable();
        playerInput.GamePlay.Pause.performed += Pause;
    }

    private void OnDisable()
    {
        playerInput.Disable();
        playerInput.GamePlay.Pause.performed -= Pause;
    }

    private void Pause(InputAction.CallbackContext ctx)
    {
        Toggle();
    }

    public void Toggle()
    {
        //if false = true, if true = false
        ui.SetActive(!ui.activeSelf);

        if (ui.activeSelf)
        {
            Time.timeScale = 0f;
        }
```

```
        else
        {
            Time.timeScale = 1f;
        }
    }

    public void Retry()
    {
        Toggle();
        sceneFader.FadeTo(SceneManager.GetActiveScene().name);
    }

    public void Menu()
    {
        Toggle();
        sceneFader.FadeTo(menuSceneName);
    }


}
```

3.1.5.2.9 SceneFader

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class SceneFader : MonoBehaviour
{
    public Image Image;
    public AnimationCurve fadeCurve;
    void Start()
    {
        StartCoroutine(FadeIn());
    }

    public void FadeTo(string scene)
    {
        StartCoroutine(FadeOut(scene));
    }

    IEnumerator FadeIn()
    {
        float t = 1f;

        while(t > 0f)
        {
            t -= Time.deltaTime;
            float a = fadeCurve.Evaluate(t);
            Image.color = new Color(0f,0f,0f,a);
            yield return 0;
        }
    }
```

```
    IEnumerator FadeOut(string scene)
    {
        float t = 0f;

        while (t < 1f)
        {
            t += Time.deltaTime;
            float a = fadeCurve.Evaluate(t);
            Image.color = new Color(0f, 0f, 0f, a);
            yield return 0;
        }
        SceneManager.LoadScene(scene);
    }

}
```

3.1.5.3. Torretas

3.1.5.3.1. Controlador de torreta electrica

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

public class ElectricTurretController : MonoBehaviour
{
    public StructureStats stats;

    public string Objetive_Tag = "Enemy";
    [Range(0f, 1f)]
    public float SlowPct = .75f;
    [HideInInspector]
    public List<EnemyStats> targets;
    private AudioSource audioSource;
    // Start is called before the first frame update
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        stats = GetComponent<StructureStats>();
    }

    // Update is called once per frame
    void Update()
    {
        if (stats.isWorking)
        {
            UpdateTarget();
            SlowTargets();
        }
    }

    void UpdateTarget()
    {
```

```
        targets.Clear();
        GameObject[] enemies =
GameObject.FindGameObjectsWithTag(Objetive_Tag);
        float distanceToEnemy;
        foreach (GameObject enemy in enemies)
        {
            distanceToEnemy = Vector3.Distance(transform.position,
enemy.transform.position);
            if (distanceToEnemy <= stats.range)
            {
                targets.Add(enemy.GetComponent<EnemyStats>());
            }
        }
    }

    void SlowTargets()
    {
        if (targets.Count > 0)
        {
            if(!audioSource.isPlaying)
                audioSource.Play();
            foreach (EnemyStats target in targets)
            {
                target.Slow(SlowPct);
            }
        }
    }
}
```

3.1.5.3.2. Controlador de torreta estrategica

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

public class StrategicTurretController : MonoBehaviour
{
    //Important variables
    private Transform target;
    private StructureStats targetStats;
    [HideInInspector]
    public StructureStats stats;

    [Header("Lase effects")]
    public LineRenderer lineRenderer;
    public ParticleSystem impactEffect;
    public Light impactLight;

    [Header("Unity Setups")]
    public List<string> Objetive_Tag;
    public Transform edge;
```

```
    public float edge_speed = 4f;
    public Transform fireLine;
    public Transform firePoint;
    public LayerMask targetMask;
    public LayerMask obstacleMask;


    private bool canSee;
    private bool isProducing;
    private AudioSource audioSource;

    // Start is called before the first frame update
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        stats = GetComponent<StructureStats>();
        InvokeRepeating("UpdateTarget", 0f, 0.5f);
    }

    GameObject[] posibleTargets()
    {
        List<GameObject> pTargets = new List<GameObject>();
        foreach(string tag in Objetive_Tag)
        {
            GameObject[] found = GameObject.FindGameObjectsWithTag(tag);
            if(found != null && found.Length > 0)
                foreach(GameObject target in found)
                    if(target != null)
                        pTargets.Add(target);
        }
        return pTargets.ToArray();
    }

    void UpdateTarget()
    {
        if (stats.isWorking)
        {
            GameObject[] targets = posibleTargets();
            float MinDistance = Mathf.Infinity;
            float distanceToTarget;
            GameObject nearestTarget = null;
            foreach (GameObject target in targets)
            {
                //this condition prevent the healer turret to select it
selft
                if (!this.gameObject.Equals(target))
                {
                    distanceToTarget =
Vector3.Distance(transform.position, target.transform.position);
                    if (stats.isHealer)
```

```
                    {
                        StructureStats structure;
                        if (target.TryGetComponent<StructureStats>(out
structure))
                        {
                            if ((distanceToTarget < MinDistance) &&
(structure.health < structure.maxHealth))
                            {
                                MinDistance = distanceToTarget;
                                nearestTarget = target;
                            }
                        }
                    }
                    else if (distanceToTarget < MinDistance)
                    {
                        MinDistance = distanceToTarget;
                        nearestTarget = target;
                    }
                }
            }

            if (nearestTarget != null && MinDistance <= stats.range)
            {
                target = nearestTarget.transform;
                canSee = true;
            }
            else
            {
                target = null;
                canSee = false;
            }
        }
        else {
            if (target != null)
            {
                target = null;
                canSee = false;
                StopProduction();
            }
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (target == null)
        {
            if (lineRenderer.enabled)
            {
                lineRenderer.enabled = false;
```

```
                impactEffect.Stop();
                impactLight.enabled = false;
            }
            return;
        }
        //Tracking target
        Utility.LookOnTarget(edge, (target.position -
edge.position).normalized, edge_speed);
        //Tracking();
        //Fire
        if (canSee)
        {
            Laser();
            Produce();
        }else
        {
            StopProduction();
        }
    }

    void Laser()
    {
        if(!audioSource.isPlaying)
            audioSource.Play();
        //Visual
        if (!lineRenderer.enabled)
        {
            lineRenderer.enabled = true;
            impactEffect.Play();
            impactLight.enabled = true;
        }
        lineRenderer.SetPosition(0, firePoint.position);
        lineRenderer.SetPosition(1, target.position);

        Vector3 iEdir = firePoint.position - target.position;
        impactEffect.transform.position = target.position +
iEdir.normalized * .3f;
        impactEffect.transform.rotation = Quaternion.LookRotation(iEdir);
    }

    void Produce()
    {
        if (!isProducing)
        {
            foreach (Product product in stats.products)
            {
                if (product.productionType == Production.Energy)
                {
                    PlayerStats.EnGeneration += product.production;
                    isProducing = true;
```

```
                }
                else if (product.productionType == Production.Material)
                {
                    PlayerStats.MatGeneration += product.production;
                    isProducing = true;
                }
                else if (product.productionType == Production.Health)
                {
                    targetStats = target.GetComponent<StructureStats>();
                    if (targetStats.health < targetStats.maxHealth)
                        targetStats.heal(product.production *
Time.deltaTime);
                }
            }
        }
    }

    void StopProduction()
    {
        if (isProducing)
        {
            foreach (Product product in stats.products)
            {
                if (product.productionType == Production.Energy)
                {
                    PlayerStats.EnGeneration -= product.production;
                }
                else if (product.productionType == Production.Material)
                {
                    PlayerStats.MatGeneration -= product.production;
                }
            }
            isProducing = false;
        }
    }

    private void OnDestroy()
    {
        StopProduction();
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position,
GetComponent<StructureStats>().range);
    }
}
```

3.1.5.3.3. Estadistica de estructura

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

//a little list of production types
public enum Production
{
    Damage, // not implemented
    Health,
    Energy,
    Material
}

public class StructureStats : MonoBehaviour
{
    [Header("Structure Stats")]
    public float maxHealth = 100f;
    public float range = 3f;
    public int energyCost = 5;

    [Header("Production")]
    public bool isHealer = false;
    public bool isCore = false;
    public List<Product> products = new List<Product>();

    [Header("Unity Objects")]
    public Image healthBar;
    public GameObject ExplosionEffect;
    [HideInInspector]
    public float health;
    public bool isWorking = false;
    bool isDestroyed = false;
    bool isCoreProducing = false;
    // Start is called before the first frame update
    void Start()
    {
        health = maxHealth;
        PlayerStats.EnConsumption += energyCost;
    }

    void Update()
    {
        //turn off the structure if there is no energy.
        if(PlayerStats.Energy <= 0 && energyCost > 0)
        {
            isWorking = false;
            //if is core then stop production
            if (isCore)
```

```
            {
                StopCoreProduction();
            }

        }else if (!isWorking) // if there is energy and the structure is
turned off then turn it on
        {
            isWorking = true;
            // if is core then start production (production of turrets are
handled by the strategic turret controller script)
            if (isCore)
            {
                CoreProduce();
            }
        }

        //update healthbar if exist.
        if(healthBar != null)
        {
            healthBar.fillAmount = health / maxHealth;
        }

    }

    /// <summary>
    /// method called when the structure recive damage.
    /// </summary>
    /// <param name="amount"> Amount of damage that the structure will
recive</param>
    public void TakeDamage(float amount)
    {
        health -= amount;
        if(health <= 0 && !isDestroyed)
        {
            Explode();
        }
    }

    /// <summary>
    /// method called when the structure is healed / repaired
    /// </summary>
    /// <param name="amount"> amount of heal to recive</param>
    public void heal(float amount)
    {
        if((health + amount) > maxHealth)
        {
            heal(maxHealth-health);
            return;
        }
        health += amount;
```

```
    }

    /// <summary>
    /// Method called when the structure will explode / die
    /// </summary>
    void Explode()
    {
        isDestroyed = true;

        GameObject effect = Instantiate(ExplosionEffect,
transform.position, Quaternion.identity);
        Destroy(effect, 1.5f);
        Destroy(gameObject);
    }

    /// <summary>
    /// Method called to start the production of the core if the structure
is a core
    /// </summary>
    void CoreProduce()
    {
        if (!isCoreProducing)
        {
            foreach (Product product in products)
            {
                if (product.productionType == Production.Energy)
                {
                    PlayerStats.EnGeneration += product.production;
                }
                else if (product.productionType == Production.Material)
                {
                    PlayerStats.MatGeneration += product.production;
                }
            }
            isCoreProducing = true;
        }
    }

    /// <summary>
    /// Method called to stop the production of the core.
    /// </summary>
    void StopCoreProduction()
    {
        if (isCoreProducing)
        {
            foreach (Product product in products)
            {
                if (product.productionType == Production.Energy)
                {
                    PlayerStats.EnGeneration -= product.production;
```

```
                }
                else if (product.productionType == Production.Material)
                {
                    PlayerStats.MatGeneration -= product.production;
                }
            }
            isCoreProducing = false;
        }
    }

    /// <summary>
    /// Called when the structure is destroyed
    /// </summary>
    private void OnDestroy()
    {
        PlayerStats.EnConsumption -= energyCost;
        StopCoreProduction();
    }

    /// <summary>
    /// Called when you select the prefab on scene to show the structure
range.
    /// </summary>
    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, range);
    }

}
```

3.1.5.3.4. Controlador de torreta

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

public class TurretController : MonoBehaviour
{
    private Transform target;
    private EnemyStats targetEnemy;
    [HideInInspector]
    public StructureStats stats;

    [Header("Use Bullets (default)")]
    public float fireRate = 1f;
    private float fireCountdown = 1f;
    public GameObject bullet;

    [Header("Use Laser")]
    public bool useLaser = false;
```

```
    public int damageOverTime=30;
    [Range(0f, 1f)]
    public float SlowPct = .5f;

    public LineRenderer lineRenderer;
    public ParticleSystem impactEffect;
    public Light impactLight;

    [Header("Unity Setups")]
    public string Objetive_Tag = "Enemy";
    public Transform edge;
    public float edge_speed = 4f;
    public Transform fireLine;
    public Transform firePoint;
    public LayerMask targetMask;
    public LayerMask obstacleMask;

    private bool canSee;
    private AudioSource audioSource;
    // Start is called before the first frame update
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        stats = GetComponent<StructureStats>();
        InvokeRepeating("UpdateTarget", 0f, 0.5f);
    }

    void UpdateTarget()
    {
        if (stats.isWorking)
        {
            GameObject[] enemies =
GameObject.FindGameObjectsWithTag(Objetive_Tag);
            float MinDistance = Mathf.Infinity;
            float distanceToEnemy;
            GameObject nearestEnemy = null;
            foreach (GameObject enemy in enemies)
            {
                distanceToEnemy = Vector3.Distance(transform.position,
enemy.transform.position);
                if (distanceToEnemy < MinDistance)
                {
                    MinDistance = distanceToEnemy;
                    nearestEnemy = enemy;
                }
            }

            if (nearestEnemy != null && MinDistance <= stats.range)
            {
                target = nearestEnemy.transform;
```

```
                targetEnemy = nearestEnemy.GetComponent<EnemyStats>();
            }
            else
            {
                target = null;
            }
        }else
        {
            if (target != null)
            {
                target = null;
            }
        }
    }


    void Shoot()
    {

        if (!audioSource.isPlaying)
            audioSource.Play();
        GameObject Bullet_O = (GameObject)Instantiate(bullet,
firePoint.position, firePoint.rotation);
        Bullet bullet_sc = Bullet_O.GetComponent<Bullet>();
        if (bullet_sc != null)
            bullet_sc.Seek(target);
    }

    // Update is called once per frame
    void Update()
    {
        if (target == null)
        {
            if (useLaser)
            {
                if (lineRenderer.enabled)
                {
                    lineRenderer.enabled = false;
                    impactEffect.Stop();
                    impactLight.enabled = false;
                }
            }
            return;
        }

        Utility.LookOnTarget(edge, (target.position -
edge.position).normalized, edge_speed);
        Tracking();
        //Fire
        if (canSee)
```

```
        {
            if (useLaser)
            {
                Laser();
            }
            else
            {
                if (fireCountdown <= 0)
                {
                    Shoot();
                    fireCountdown = 1f / fireRate;
                }

                fireCountdown -= Time.deltaTime;
            }
        }

    }

    void Tracking()
    {
        // fl = Fire line Direction
        Vector3 fld = fireLine.TransformDirection(Vector3.forward);
        RaycastHit Obstaclehit;
        RaycastHit enemyhit;

        if (Physics.Raycast(fireLine.position, fld * stats.range, out
Obstaclehit, stats.range, obstacleMask))
        {
            if (Obstaclehit.collider!= null)
            {
                canSee = false;
                Debug.DrawRay(fireLine.position, fld *
Obstaclehit.distance, Color.red);
            }
            else
            {
                canSee = false;
                Debug.DrawRay(fireLine.position, fld *
Obstaclehit.distance, Color.red);
            }

        }else if (Physics.Raycast(fireLine.position, fld * stats.range,
out enemyhit, stats.range, targetMask))
        {
            if (enemyhit.collider.CompareTag(Objetive_Tag))
            {
                canSee = true;
                Debug.DrawRay(fireLine.position, fld * enemyhit.distance,
Color.green);
```

```
            }
            else
            {
                canSee = false;
                Debug.DrawRay(fireLine.position, fld * enemyhit.distance,
Color.red);
            }

        }
        else
        {
            canSee = false;
            Debug.DrawRay(fireLine.position, fld * stats.range,
Color.red);
        }

    }

    void Laser()
    {
        if(!audioSource.isPlaying)
            audioSource.Play();
        //Laser damage

        targetEnemy.TakeDamage(damageOverTime * Time.deltaTime);
        targetEnemy.Slow(SlowPct);
        //Visual
        if (!lineRenderer.enabled)
        {
            lineRenderer.enabled = true;
            impactEffect.Play();
            impactLight.enabled=true;
        }
        lineRenderer.SetPosition(0,firePoint.position);
        lineRenderer.SetPosition(1,target.position);

        Vector3 iEdir = firePoint.position - target.position;
        impactEffect.transform.position = target.position +
iEdir.normalized * .3f;
        impactEffect.transform.rotation = Quaternion.LookRotation(iEdir);
    }

}
```

3.1.5.4. UI

3.1.5.4.1 Estadisticas del juego UI

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

```
using TMPro;

public class GameStatsUI : MonoBehaviour
{
    public PlayerStats stats;

    public TextMeshProUGUI EnergyAmount;
    public TextMeshProUGUI EnergyGeneration;
    public TextMeshProUGUI MineralAmount;
    public TextMeshProUGUI MineralGeneration;

    public Image EnergyBar;

    // Update is called once per frame
    void Update()
    {
        EnergyBar.fillAmount = stats.getEnergyAmount();
        EnergyAmount.text = string.Format("{0:N0}", PlayerStats.Energy);
        EnergyGeneration.text = textFormater(stats.getEnProduction());
        MineralAmount.text = string.Format("{0:N0}",
PlayerStats.materials);
        MineralGeneration.text = textFormater(stats.getMatProduction());
    }

    string textFormater(float amount)
    {
        if(amount == 0)
        {
            return "<color=#" + ColorUtility.ToHtmlStringRGB(Color.white)
+ "> - </color>";
        }
        else if(amount > 0)
        {
            return "<color=#" + ColorUtility.ToHtmlStringRGB(Color.green)
+ "> +" + amount + "</color>";
        }
        else
        {
            return "<color=#" + ColorUtility.ToHtmlStringRGB(Color.red) +
"> " + amount + "</color>";
        }
    }
}
```

3.1.5.4.2 IMouse

```
/// <summary>
/// <para>
///     Interface to use for mouse movement over gameobject colliders.
/// </para>
/// Used to beat the limitation of the new input system where this
```

```
/// options on the monobehavior do not work by default.
/// </summary>
public interface IMouse
{
    /// <summary>
    /// Method to use for mouse clicks.
    /// </summary>
    void OnMouseDown();
    /// <summary>
    /// Method to use when mouse is over the gamebject.
    /// </summary>
    void OnMouseEnter();
    /// <summary>
    /// Method to use when the mouse isn't over the gameobject anymore.
    /// </summary>
    void OnMouseExit();
}
```

3.1.5.4.3 Nodo UI

```
using UnityEngine;
using UnityEngine.UI;

public class NodoUI : MonoBehaviour
{
    public GameObject ui;
    [Header("Turret Info")]
    public Text turretName;
    public Text turretRange;
    public Text turretDamage;
    [Header("Upgrade Info")]
    public Button upgradeButton;
    public Text upgradeCost;
    [Header("Sell Info")]
    public Button sellButton;
    public Text sellGains;

    private Nodo target;
    public void SetTarget(Nodo nodo)
    {
        target = nodo;
        transform.position = target.GetBuildPosition();
        if (!target.isUpgraded)
        {
            upgradeCost.text = "$" + target.turretBlueprint.upgradedCost;
            upgradeButton.interactable = true;
        }
        else
        {
            upgradeCost.text = "DONE";
            upgradeButton.interactable = false;
```

```
        }
        sellGains.text = "$" + target.turretBlueprint.GetSellAmount();


        turretName.text = target.turretBlueprint.Name;
        TurretController turret;
        ElectricTurretController electricTurret;
        if (nodo.turret.TryGetComponent<TurretController>(out turret))
        {
            SetTurretInfo(turret);
        }
        else if(nodo.turret.TryGetComponent<ElectricTurretController>(out
electricTurret))
        {
            SetTurretInfo(electricTurret);
        }
        else{
            SetTurretInfo(nodo.turret.GetComponent<StructureStats>());
        }
        ui.SetActive(true);
    }

    public void SetTurretInfo(TurretController turret)
    {
        turretRange.text = "Range: " + turret.stats.range;
        if (turret.useLaser)
        {
            turretDamage.text = "Damage: " + turret.damageOverTime + "/s";
        }
        else
        {
            turretRange.text = turretRange.text + "\nFire rate: " +
turret.fireRate + "/s";
            turretDamage.text = "Damage: " +
turret.bullet.GetComponent<Bullet>().damage;
        }
    }

    public void SetTurretInfo(ElectricTurretController turret)
    {
        turretRange.text = "Range: " + turret.stats.range;
        turretDamage.text = "Slownes: " + turret.SlowPct * 100 + "%";
    }

    public void SetTurretInfo(StructureStats turret)
    {
        turretRange.text = "Range: " + turret.range;
        turretDamage.text = "Prodution: " +
turret.products[0].productionType.ToString() + "\nAmount: " +
turret.products[0].production;
```

```
    }
    public void Hide()
    {
        ui.SetActive(false);
    }

    public void Upgrade()
    {
        target.UpgradeTurret();
        BuildManager.instance.DeselectNode();
    }

    public void Sell()
    {
        target.SellTurret();
        BuildManager.instance.DeselectNode();
    }

}
```

3.1.5.4.4 Oleadas sobrevividas

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class RoundsSurvived : MonoBehaviour
{

    public Text roundText;

    void OnEnable()
    {
        StartCoroutine(AnimateText());
    }

    IEnumerator AnimateText()
    {
        roundText.text = "0";
        int round = 0;

        yield return new WaitForSeconds(.7f);

        while (round < PlayerStats.Rounds)
        {
            round++;
            roundText.text = round.ToString();

            yield return new WaitForSeconds(.05f);
        }
```

```
    }

}
```

3.1.5.4.5 Tienda

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Shop : MonoBehaviour
{
    BuildManager buildManager;
    //BuildingCreator buildingCreator;
    public TurretBlueprint[] TurretBlueprints;
    public Button ButtonPrefab;
    public Transform ShopContent;
    private void Start()
    {
        //buildingCreator = BuildingCreator.GetInstance();
        buildManager = BuildManager.instance;
        foreach(TurretBlueprint turret in TurretBlueprints)
        {
            Button button = Instantiate(ButtonPrefab, ShopContent);
            button.name = turret.Name;
            button.image.sprite = turret.Sprite;

button.GetComponent<ShopButtonComponents>().setComponents(turret.materialC
ost.ToString(),getEnCost(turret.prefabs[0]).ToString());
            button.onClick.AddListener(delegate { SelectTurret(turret);
});
        }
    }

    int getEnCost(GameObject turret)
    {
        return turret.GetComponent<StructureStats>().energyCost;
    }

    public void SelectTurret(TurretBlueprint Blueprint)
    {
        //buildingCreator.BlueprintSelected(Blueprint);
        buildManager.SelectTurretToBuild(Blueprint);
    }
}
```

3.1.5.4.6 Componentes de boton de tienda

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

```
using TMPro;

public class ShopButtonComponents : MonoBehaviour
{

    public TextMeshProUGUI materialCost;
    public TextMeshProUGUI energyCost;

    public void setComponents(string materialCost, string energyCost)
    {
        this.materialCost.text = materialCost;
        this.energyCost.text = energyCost;
    }

}
```

3.1.5.5. Oleadas

3.1.5.5.1 Oleada

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

[System.Serializable]
public class Wave
{
    public List<WaveEnemy> enemies;
    public int count;
    public float spawnRate;
}
```

3.1.5.5.2 Enemigo de oleada

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

[Serializable]
public class WaveEnemy
{
    public GameObject enemy;
    [Range(1, 5)]
    public int priority;
}
```

3.1.5.5.3 Iniciador de oleada

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
```

```
public class WaveSpawner : MonoBehaviour
{

    public static int EnemiesAlive = 0;

    public Wave[] waves;

    public Transform SpawnPoint;
    public TextMeshProUGUI WaveCountDownText;
    public GameManager gameManager;

    public float WaveCountDown = 10f;
    public float CountDown = 30f;
    private int waveIndex = 0;
    private int partitions;


    // Update is called once per frame
    void Update()
    {
        if(EnemiesAlive > 0)
        {
            return;
        }else if (waveIndex >= waves.Length)
        {
            gameManager.WinLevel();
            this.enabled = false;
            return;
        }
        if (CountDown <= 0f)
        {
            StartCoroutine(SpawnWave());
            CountDown = WaveCountDown;
            return;
        }
        CountDown -= Time.deltaTime;
        CountDown = Mathf.Clamp(CountDown,0f,Mathf.Infinity);
        WaveCountDownText.text = string.Format("{0:00.00}",CountDown);
    }

    private struct rarity
    {
        public int index;
        public int minPartitions;
        public int maxPartitions;
    }

    IEnumerator SpawnWave()
```

```
    {
        PlayerStats.Rounds++;

        Wave wave = waves[waveIndex];

        EnemiesAlive = wave.count;

        partitions = 0;

        List<rarity> rarityList = new List<rarity>();

        for(int cont = 0; cont < wave.enemies.Count; cont++)
        {
            rarity rarity = new rarity();
            rarity.index = cont;
            rarity.minPartitions = partitions;
            rarity.maxPartitions = partitions +
wave.enemies[cont].priority;
            rarityList.Add(rarity);
            partitions += wave.enemies[cont].priority;
        }
        for (int i = 0; i < wave.count; i++)
        {
            int partition = Random.Range(0, partitions);
            foreach (rarity rarity in rarityList)
            {
                if (partition <= rarity.maxPartitions && partition >=
rarity.minPartitions)
                {
                    SpawnEnemy(wave.enemies[rarity.index].enemy);
                    break;
                }
            }
            yield return new WaitForSeconds(wave.spawnRate);
        }
        waveIndex++;
    }

    void SpawnEnemy(GameObject enemy)
    {
        Instantiate(enemy, SpawnPoint.position, SpawnPoint.rotation);
    }


}
```

3.1.5.6. Waypoints

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

```
public class WayPoints : MonoBehaviour
{
    public static List<Transform> points;
    private List<Transform> pointsList = new List<Transform>();
    void Awake()
    {
        InvokeRepeating("updatePoints",0f,2f);
    }

    void updatePoints()
    {
        pointsList.Clear();

        int points = transform.childCount;
        for (int i = 0; i < points; i++)
        {
            this.pointsList.Add(transform.GetChild(i));
        }

        WayPoints.points = pointsList;
    }

}
```

3.1.5.7. ScriptableObjects

3.1.5.7.1 Plano

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

[CreateAssetMenu(menuName = "MyGame/BluePrints/NodeBlueprint", fileName =
"NodeBlueprint")]
public class Blueprint : ScriptableObject
{
    public string Name;
    public List<GameObject> prefabs;
    public Sprite Sprite;
}
```

3.1.5.7.2 Plano de torreta

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[CreateAssetMenu(menuName = "MyGame/BluePrints/TurretBlueprint", fileName
="TurretBlueprint")]
public class TurretBlueprint : Blueprint
{
    public int materialCost;
```

```
    public int upgradedCost;

    public int GetSellAmount()
    {
        return materialCost / 2;
    }

    public int MaxLevel => prefabs.Count;

}
```

3.1.5.8. Variados

3.1.5.8.1 Bala

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

public class Bullet : MonoBehaviour
{

      private Transform target;

      public int damage=15;
      public float speed = 10f;
      public float explosionRadius = 0f;
      public GameObject ImpactEffect;

      private AudioSource impactSource;
      private bool hasHit = false;
      public void Seek (Transform _target)
      {
            target = _target;
      }

    private void Start()
    {
        impactSource = GetComponent<AudioSource>();

      }

    // Update is called once per frame
    void Update()
      {
            if (target == null)
            {
                  Destroy(gameObject);
                  return;
            }

            Vector3 dir = target.position - transform.position;
```

```
            float distanceFPS = speed * Time.deltaTime;

            if ( dir.magnitude <= distanceFPS)
            {
                  HitTarget();
                  return;
            }

            transform.Translate(dir.normalized * distanceFPS,
Space.World);
            transform.LookAt(target);
      }

      void HitTarget()
      {
            if (hasHit)
                  return;
            hasHit = true;


            GameObject effect =
(GameObject)Instantiate(ImpactEffect,transform.position,transform.rotation
);
            effect.transform.GetComponent<ParticleSystem>().Play();
            Destroy(effect, 5f);
            if(explosionRadius > 0f)
        {
                  Explode();
        }
        else
        {
                  Damage(target);
        }
            if(!impactSource.isPlaying)
                  impactSource.Play();

            Destroy(gameObject, impactSource.clip.length);
      }
      void Damage(Transform enemy)
      {
            EnemyStats e = enemy.GetComponent<EnemyStats>();
            if (e != null)
        {
                  e.TakeDamage(damage);
            }
      }

      void Explode(){
```

```
            Collider[] colliders =
Physics.OverlapSphere(transform.position, explosionRadius);
            foreach(Collider collider in colliders)
        {
                  if (collider.tag == "Enemy")
            {
                        Damage(collider.transform);
            }
        }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color=Color.red;
            Gizmos.DrawWireSphere(transform.position, explosionRadius);
    }
}
```

3.1.5.8.2 Nodo

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.InputSystem;

public class Nodo : MonoBehaviour, IMouse
{
    public Color hoverColor;
    public Vector3 positionOffset;
    public Color BlockedColor;
    [HideInInspector]
    public GameObject turret;
    public TurretBlueprint turretBlueprint;
    public bool isUpgraded = false;

    private int upgradeIndex;
    private Renderer rend;
    private Color StartColor;

    BuildManager buildManager;

    public Vector3 GetBuildPosition()
    {
        return transform.position + positionOffset;
    }

    void Start()
    {
        rend = GetComponent<Renderer>();
        StartColor = rend.material.color;
```

```
        buildManager = BuildManager.instance;
        upgradeIndex = 0;
        if (turretBlueprint != null)
        {
            buildManager.SelectTurretToBuild(turretBlueprint);
            BuildTurret(turretBlueprint, true);
        }
    }

    void BuildTurret(TurretBlueprint blueprint, bool isFree)
    {
        if (!buildManager.HasMoney && !isFree)
        {
            return;
        }

        if (!isFree) {
            PlayerStats.materials -= blueprint.materialCost;
        }

        GameObject _turret = Instantiate(blueprint.prefabs[0],
GetBuildPosition(), blueprint.prefabs[0].transform.rotation);
        turret = _turret;
        turretBlueprint = blueprint;

        if(!(blueprint.prefabs.Count > 1))
            isUpgraded = true;

        GameObject effect = Instantiate(buildManager.BuildEffect,
GetBuildPosition(), Quaternion.identity);
        Destroy(effect, 2f);
    }

    public void UpgradeTurret()
    {
        upgradeIndex++;
        if (PlayerStats.materials < turretBlueprint.upgradedCost)
        {
            return;
        }
        PlayerStats.materials -= turretBlueprint.upgradedCost;
        //Destroy old turret
        Destroy(turret);
        //Build Upgraded turret
        GameObject _turret =
Instantiate(turretBlueprint.prefabs[upgradeIndex], GetBuildPosition(),
turretBlueprint.prefabs[upgradeIndex].transform.rotation);
        turret = _turret;
        turretBlueprint.materialCost += turretBlueprint.upgradedCost;
```

```
        GameObject effect = Instantiate(buildManager.BuildEffect,
GetBuildPosition(), Quaternion.identity);
        Destroy(effect, 2f);
        if(upgradeIndex >= turretBlueprint.prefabs.Count - 1)
            isUpgraded = true;
    }
    public void SellTurret()
    {
        PlayerStats.materials += turretBlueprint.GetSellAmount();

        GameObject effect = Instantiate(buildManager.SellEffect,
GetBuildPosition(), Quaternion.identity);
        Destroy(effect, 2f);

        //Destroy old turret
        Destroy(turret);
        turretBlueprint = null;


        upgradeIndex = 0;
        isUpgraded = false;
    }

    void IMouse.OnMouseEnter()
    {

        if (EventSystem.current.IsPointerOverGameObject())
            return;
        if (!buildManager.CanBuild)
            return;
        if (buildManager.HasMoney)
        {
            rend.material.color = hoverColor;
        }else
        {
            rend.material.color = BlockedColor;
        }
    }

    void IMouse.OnMouseExit()
    {
        rend.material.color = StartColor;
    }

    void IMouse.OnMouseDown()
    {

        if (EventSystem.current.IsPointerOverGameObject())
```

```
            return;
        if (turret != null)
        {
            buildManager.SelectNode(this);
            return;
        }
        if (!buildManager.CanBuild)
            return;

        BuildTurret(buildManager.GetTurretToBuild(),false);
    }


}
```

3.1.5.8.3 Estadisticas de jugador

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

public class PlayerStats : MonoBehaviour
{
    public int startMaterials = 150;
    public int maxEnergy = 100;
    [HideInInspector]
    public static float materials;
    public static int Rounds;
    public static float Energy;
    public static int MatGeneration = 0;
    public static int EnGeneration = 0;
    public static int EnConsumption = 0;
    // Start is called before the first frame update
    void Start()
    {
        materials = startMaterials;
        Energy = maxEnergy;
        Rounds = 0;
    }

    // Update is called once per frame
    void Update()
    {
        EnergyGeneration();
        MaterialGeneration();
    }

    public float getEnProduction()
    {
        return EnGeneration - EnConsumption;
    }
```

```
    public float getMatProduction()
    {
        return MatGeneration;
    }

    public float getEnergyAmount()
    {
        return Energy / maxEnergy;
    }
    void MaterialGeneration()
    {
        materials += (MatGeneration * Time.deltaTime);
    }

    void EnergyGeneration()
    {
        EnGenerate((getEnProduction() * Time.deltaTime));
    }

    void EnGenerate(float amount)
    {
        if ((Energy + amount) > maxEnergy)
        {
            Energy = maxEnergy;
            return;
        }
        if((Energy + amount) < 0)
        {
            if(Energy != 0)
                EnGenerate(-Energy);
            return;
        }

        Energy += amount;
    }
}
```

3.1.5.8.4 Producto

```
using System;

[Serializable]
public class Product
{
    public Production productionType;
    public int production;
}
```

3.1.5.8.5 Utileria

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

public static class Utility
{
    /// <summary>
    /// NOT IMPLEMENTET
    /// Shuffle the array on random
    /// </summary>
    /// <typeparam name="T"> type of the array </typeparam>
    /// <param name="array"> array of data to sort </param>
    /// <param name="seed"> seed for randomizer </param>
    /// <returns></returns>
    public static T[] ShuffleArray<T>(T[] array, int seed)
    {
        System.Random random = new System.Random(seed);

        for(int i = 0; i < array.Length-1; i++)
        {
            int randomIndex = random.Next(i,array.Length-1);
            T result = array[randomIndex];
            array[randomIndex] = array[i];
            array[i] = result;
        }

        return array;
    }
    /// <summary>
    /// Rotate the GameObject to look on target
    /// </summary>
    /// <param name="obj"> Is the transform of the GameObject to be
rotated</param>
    /// <param name="Direction">Is the direction the transform has to
look</param>
    /// <param name="speed">Is the speed of the rotation</param>
    public static void LookOnTarget(Transform obj, Vector3 Direction,
float speed)
    {
        Quaternion lookRotation =
Quaternion.LookRotation(Direction.normalized);
        Vector3 rotation = Quaternion.Lerp(obj.rotation, lookRotation,
Time.deltaTime * speed).eulerAngles;
        obj.rotation = Quaternion.Euler(rotation.x, rotation.y,
obj.rotation.z);
    }
```

}

3.2. PROTOTIPOS

Los prototipos del juego se han dividido en 2 categorías, Lo-Fi, Hi-Fi. Y se expresan de la siguiente forma:

- **Lo-Fi:** un prototipo compuesto solo por un nivel básico, este prototipo nunca verá la luz del día pues solo sirve como base para el desarrollo.
- **Hi-Fi:** 2 prototipos;
    - El primer prototipo Hi-Fi incorpora el menú principal, menú selector de niveles y 4 niveles básicos.
    - El segundo prototipo Hi-Fi extiende las funcionalidades al agregar los sonidos, más mecánicas, enemigos, torretas y niveles.

3.3. PERFILES DE USUARIOS

El público objetivo es:

- Personas mayores de 10 años.
- Personas que busquen algo para pasar el rato.
- Personas que busquen un desafío.
- Personas que le gusten los juegos de estrategia.

3.4. USABILIDAD

Cuando inicias el juego lo primero que vez es el menú de inicio, un pequeño menú donde encuentras las opciones de salir del juego y jugar.

Si presionas la opción de jugar, entraras al menú de niveles, aquí se encuentran 15 botones, 1 por nivel, más un botón para regresar al menú principal, los botones de los niveles que aún no ha alcanzado estarán bloqueados, por lo que deberás acceder a los niveles jugables para poder desbloquearlos todos.

En la escena del juego encontraras el mapa del juego con el core a defender y el enemy core, además de todos los otros aspectos del nivel, aquí tendrás 7 botones referentes a cada tipo de torre, un contador de recursos, un contador de tiempo para las oleadas y la barra de energía que decide si las torretas funcionan y no. Para los recursos y energía, se encuentran unos contadores especiales que indican cuanto de esto generas por segundo.

Finalmente tenemos el menú de victoria con 3 opciones, continuar, reintentar y salir al menú principal. Mientras el menú de derrota cuenta con la opción de volver a intentar y salir al menú principal.

3.5. TEST

Se realizo un test con dos personas, estos fueron los resultados

Sexo	Masculino
Edad	24
Nivel de estudios	Universitario
Aficiones	Mecánica automotriz

Tareas	Puntuación	Comentarios
Jugabilidad	4	
Control del personaje	4	
Guía del usuario	5	Bastante intuitivo
Diseño visual	3	¿Cuál es la diferencia entre donde puedo construir y no?
Mecánica del juego	5	

Con este individuo obtuvimos una puntuación de 4.2, se detectó un grabe fallo, y es que como los obstáculos tenían el mismo color que los nodos normales, se podría malinterpretar y pensar que no puedes construir por fallo del juego. Por lo que se ha propuesto la tarea de retocar este punto en antes de la salida del juego.

Sexo	Masculino
Edad	22
Nivel de estudios	Universitario
Aficiones	Computación

Tareas	Puntuación	Comentarios
Jugabilidad	5	
Control del personaje	4	
Guía del usuario	5	
Diseño visual	4	Me gusta como se ve el juego
Mecánica del juego	5	Me gustan los tricksters

Con la segunda persona obtuvimos una puntuación de 4.6 gracias a los cambios realizados obtuvimos una mejor puntuación, aun se debe mejorar el diseño visual y el control de personajes.

3.6. VERSIONES DE LA APLICACION

El juego cuenta con 3 versiones:

- **Versión Alpha:** una versión no disponible para el público conformada por un solo nivel básico.
- **Versión Beta:** una versión que cuenta con el menú principal, menú de selección de niveles y 4 niveles básicos.
- **Versión 1.0:** versión completa del juego, con 15 niveles, 4 variantes de enemigos, 7 tipos de torretas y la implementación de sonidos y nuevas mecánicas en el juego.

CAPITULO IV: PUBLICACION

4.1 Requisitos de instalación

Sistema operativo: Windows 7 – Windows 11

RAM: 2 GB

Tarjeta gráfica: Envidia GTX 1650

4.2 Instrucciones de uso

En el juego el jugador tendrá distintas opciones para mover la cámara, para que este pueda jugar como mejor le parezca, estas opciones son:

- Movimiento con WASD
- Flechas
- Moviendo el mouse a uno de los costados.

Contará con un medio para hacer zoom usando la rueda del mouse.

Y podrán seleccionar, colocar y acceder al menú de mejora/venta de torretas mediante el clic derecho.

Además, el juego cuenta con una función de pausado la cual se activa al presionar la tecla ESC.

4.3 Bugs

El juego cuenta con pequeños bugs:

1. El primero es una distorsión de los audios, hay algunos audios que terminan siendo más alto de lo configurado o directamente no se escuchan.
2. Los enemigos y torretas cuentan con una mecánica para detectar obstáculos y objetivos, mientras las torretas las usan para apuntar, los enemigos la usan para evitar obstáculos y llegar a su objetivo, precisamente en esta parte hay problema, hay algunos niveles que esta función hace que el enemigo haga movimientos extraños.

4.4 Proyección a futuro

El juego actualmente se distribuye en ITCH.IO para WEB y Windows, pero este planea avanzar a más tiendas como la Microsoft Store y también dirigirse a otras plataformas, como la de móviles, iniciando su distribución primero en la Play Store de Android y con el tiempo dirigirse a la App Store de Apple.

4.5 Presupuesto

Puesto	Personal	Pago/Hora	Duracion	Sub total
INGENIERO DE AUDIO	Guillermo Santos	180	48 horas	8,640
DISEÑADOR		220	180 horas	39,600
ILUSTRADOR		180	56 horas	10,080
PROGRAMADOR		230	220 horas	50,600
			Coste de Herramientas:	2,500
			Total:	111,420

4.6 Análisis de mercado

El mercado internacional del videojuego creció un 9,6 % en el año 2019 con respecto al año anterior, alcanzando una facturación total de 152.100 millones de dólares (133.670 millones de euros), según las estimaciones de Newzoo.

La región de Asia y el Pacífico siguió liderando de forma indiscutible la industria del videojuego, con el 47% de la facturación total, seguida de Norteamérica (26%) y Europa, Oriente Medio y África (23%). A nivel específico, el mercado de videojuegos continuó encabezado por Estados Unidos (32.400 millones de euros), China (32.000 millones de euros) y Japón (16.700 millones de euros).

4.7 Viabilidad

El juego actualmente se puede jugar en dispositivos con sistema operativo Windows o desde cualquier navegador mediante la plataforma ITCH.IO, el juego no está sujeta a ninguna licencia o necesidad de pago.

CONCLUSION

Con esto damos conclusión a lo que fue el proyecto de programación de videojuegos, la verdad que fue una experiencia interesante el dar rienda suelta y desarrollar un videojuego como este. Aprendí mucho sobre lo que es la programación de videojuegos y el trayecto que conlleva la creación de uno. Concluyo con la espera de poder aplicar de manera practica todo lo aprendido en esta materia.

REFERENCIAS BIBLIOGRAFICAS

- Juego inspirado en: Rogue Tower(Steam)
- Video tutoriales, juego plantilla y assets para 3 de las torres de defensa: Serie
- Assets Store, componente 1, para torres estratégicas: Robo's turret (free sample) | 3D Sci-Fi | Unity Asset Store
- Assets Store, componente 2, para la torre eléctrica: Electric Turret | 3D Characters | Unity Asset Store
- Fuente de audio: Download FREE Sound Effects (zapsplat.com)
- **LINK DE ITCH.IO:** Defend the core by SanctusForce (itch.io)
- **LINK DEL PROYECTO:** **Guillermo-Santos/DefendTheCore (github.com)**
- **LINK DE CAPITULOS:** **DefendTheCore/Capitulos del proyecto at santo · Guillermo-Santos/DefendTheCore (github.com)**