

Capítulo 3

Desarrollo del videojuego

Al iniciar un proyecto, especialmente si está enfocado al desarrollo de videojuegos, es esencial dividir el proyecto en tres etapas importantes: **preproducción**, **producción** y **postproducción**. En la primera etapa de preproducción es donde iniciamos con la conceptualización de la idea y el diseño de la misma. Al terminar la primera etapa pasamos a la producción en la cual empezamos con la implementación y por último a terminar la producción iniciamos la postproducción, la cual se enfoca en las pruebas del videojuego y su lanzamiento. El proceso de desarrollo de videojuegos, como lo muestra la figura 6, se divide en las etapas antes mencionadas y en donde cada una de las ellas puede iterar de regreso; es decir, podemos realizar cambios al diseño del juego estando en la etapa de producción o modificar aspectos de la implementación de acuerdo al resultado de nuestras pruebas.

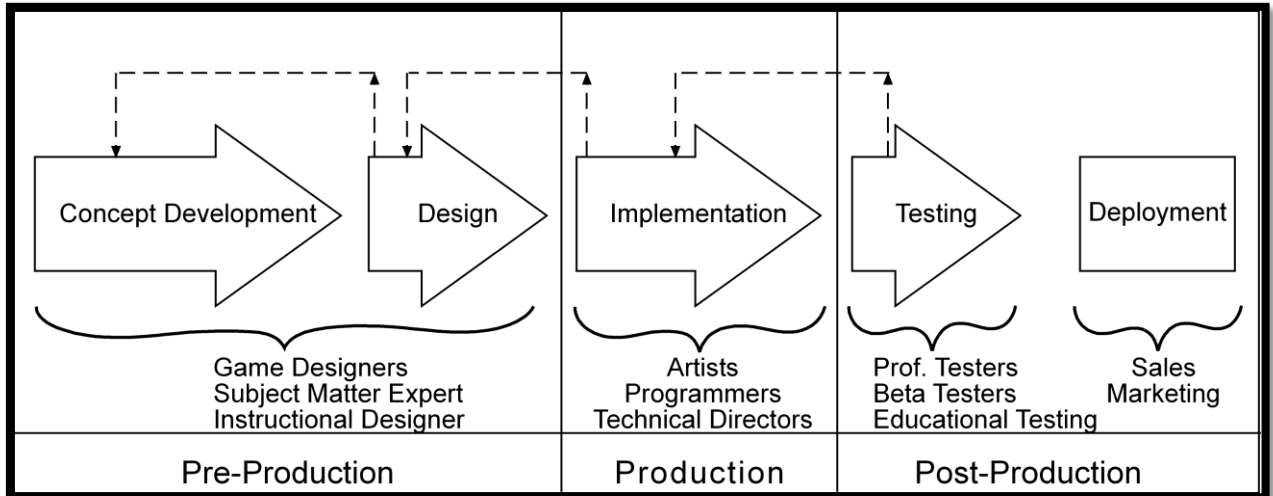


Figura 6. Proceso de desarrollo de videojuegos.

3.1 Preproducción.

La etapa de preproducción es donde se realizó la investigación sobre el tema del proyecto. Aquí buscamos referencias artísticas de cómo queremos que se vea el videojuego. Para el desarrollo del videojuego se investigó en la literatura acerca de los elementos de videojuegos de acción, especialmente en aquellos en donde el jugador controla una nave y algunos otros juegos que tuvieran este tipo de elementos. De la investigación se optó por tomar como referencia principal el juego *Tempest* de Atari Inc. (figura 7), el cual es un juego en el que su mecánica principal es sobrevivir el mayor tiempo posible, moviéndose dentro de los extremos del nivel esquivando o destruyendo enemigos. El diseño del nivel fue basado en la premisa del movimiento en los extremos del nivel, además; se usó un diseño retro, para que el videojuego diera una sensación más espacial o de ciencia ficción.





Figura 7. Nivel del videojuego Tempest.

Por otra parte, se llevó a cabo la selección de las herramientas del entorno de desarrollo del videojuego. Los factores que se tomaron en cuenta fueron los siguientes:

- Compatibilidad con Oculus Rift.
- Fácil de usar.
- Experiencia del desarrollador.
- Costo del software.

Se realizó una búsqueda de los motores de juegos que tuvieran compatibilidad con Oculus Rift. Para poder decidir cuál motor usar se realizó una la siguiente tabla comparativa para identificar ventajas y desventajas.

 UNREAL ENGINE	
<ul style="list-style-type: none"> • Tiene Costo. • Compatible con Oculus Rift. • Soporte nativo para Oculus Rift. • Ambiente de desarrollo gráfico. 	<ul style="list-style-type: none"> • Tiene Costo, pero cuenta con versión gratuita. • Compatible con Oculus Rift. • Soporte nativo para Oculus Rift.




<ul style="list-style-type: none"> • C++. • Sin experiencia del desarrollador. 	<ul style="list-style-type: none"> • Ambiente de desarrollo gráfico. • C#, UnityScript. • Con experiencia previa del desarrollador.
	
<ul style="list-style-type: none"> • Tiene Costo, pero cuenta con versión gratuita. • Compatible con Oculus Rift. • No contiene soporte nativo para Oculus Rift. • Ambiente de desarrollo gráfico. • C++, Lua. • Sin experiencia del desarrollador. 	<ul style="list-style-type: none"> • Tiene Costo. • Compatible con Oculus Rift. • No contiene soporte nativo para Oculus Rift. • Ambiente de desarrollo gráfico. • C++. • Sin experiencia del desarrollador.
	
<ul style="list-style-type: none"> • Tiene Costo • Compatible con Oculus Rift • No contiene soporte nativo para Oculus Rift. • Ambiente de desarrollo gráfico. • Javascript. • Sin experiencia del desarrollador. 	

Tabla 1. Comparación de motores de juegos.

Los dos motores preferibles fueron Unreal Engine y Unity Engine por contar con soporte de manera nativa, con lo que se facilita el trabajo, ya que no fue necesario plantear una etapa de desarrollo para añadir la característica

de compatibilidad con Oculus Rift. De estos motores se decidió usar Unity Engine por su versión gratuita y experiencia previa del desarrollador.

3.1.1 Mecánicas del videojuego

Antes de planear los elementos del videojuego y el diseño del nivel es importante tener bien marcadas y seleccionadas las mecánicas principales del juego. Las mecánicas como se mencionó al inicio de este capítulo se desarrollaron en base al juego *Tempest*, con lo que se llegó a la conclusión de que consistía en tres mecánicas que tienen una relación entre sí, como se muestra en el siguiente diagrama.

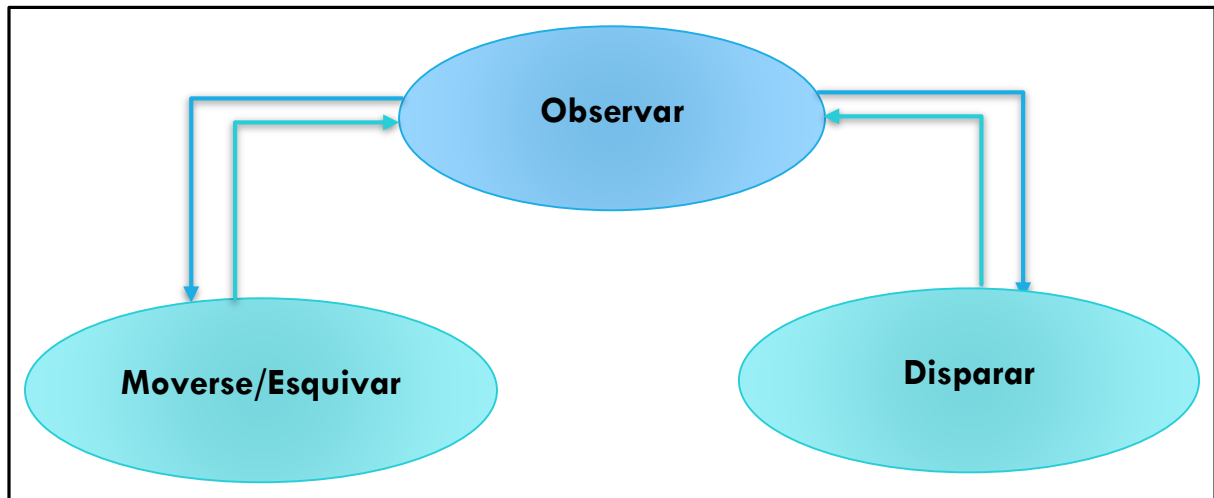


Figura 8. Flujo de las mecánicas del videojuego.

La acción que el jugador realiza más es *observar*, esto es muy común en juegos de acción, en donde la reacción es lo más importante, la cual se da mediante la observación del escenario y lo que sucede en él, para luego tener dos opciones *moverse/esquivar* o *disparar*, y estos eventos suceden en cada momento durante la sesión de juego.

A partir de los resultados de la investigación se tiene un enfoque concreto de lo que se quiere hacer y adonde se quiere llegar. Además, es importante

tener referencias visuales de los conceptos del juego, como son el nivel, enemigos y personajes. Para ello se inicia a bocetar las ideas, basándose en la imaginación del artista y las referencias obtenidas de la investigación.

3.1.2 Conceptos del videojuego.

La conceptualización de las ideas se inició con el nivel del videojuego, el cual se hizo de una manera circular y dando la ambientación al de un agujero de gusano o un viaje a través de un túnel intergaláctico (Figura 9).

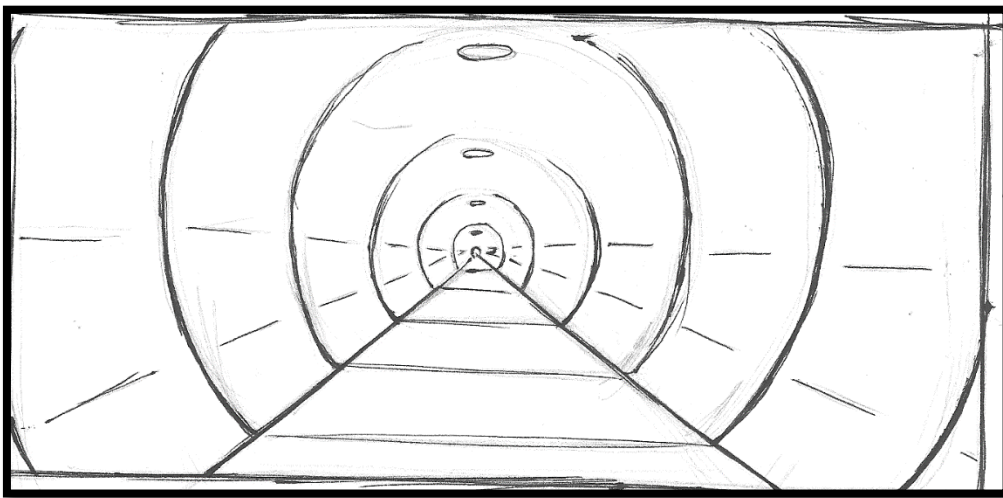


Figura 9. Primer concepto del nivel.

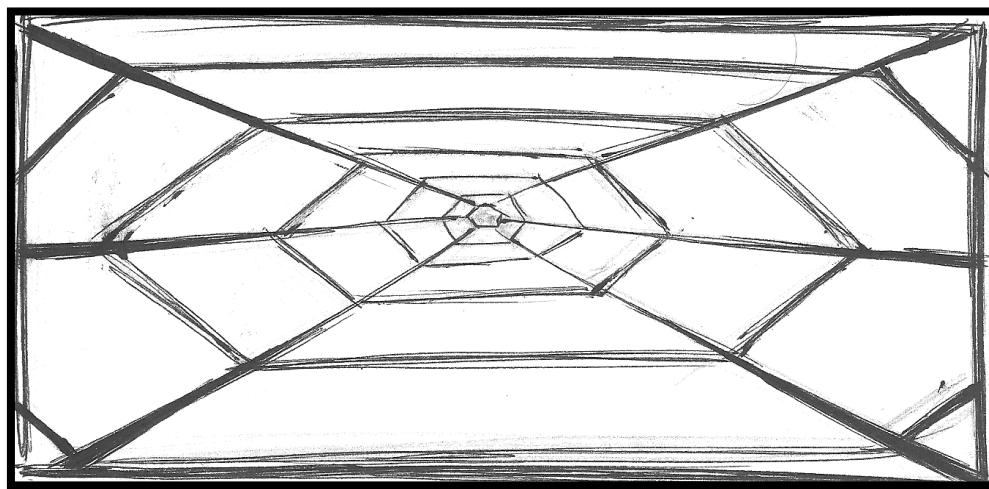


Figura 10. Segundo concepto del nivel.

Como podemos observar en los conceptos del nivel el primer concepto tiene una forma circular, pero este no daba a entender muy bien que se trataba de un nivel espacial, por ello el segundo concepto (figura 10) al agregarle una especie de pistas se lograba percibir la referencia espacial. Por ello al final se usó una combinación de ambos usando la forma circular del primero y añadiéndole las pistas del segundo.

Al terminar la conceptualización del nivel, se inició con los bocetos de los enemigos. Para estos se utilizaron figuras geométricas, ya que daban una referencia visual concreta a la meta que se estaba buscando, que se sintiera espacial y a la vez retro.

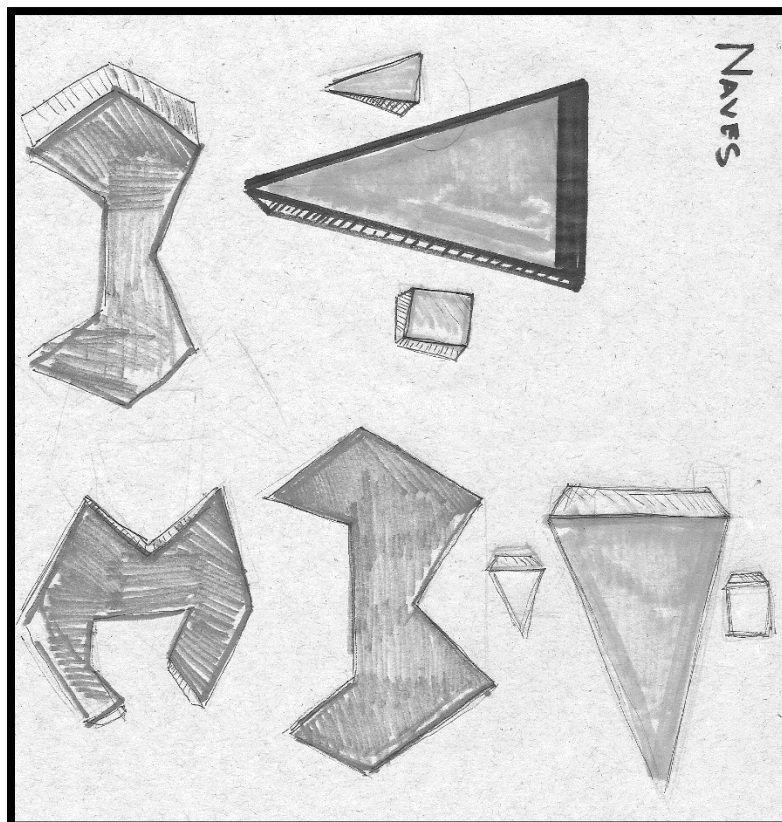


Figura 11. Arte conceptual de los enemigos.

Con los bocetos de la figura 11 fueron suficientes para obtener el diseño visual que se buscaba en los enemigos, esencialmente se hicieron dos tipos de enemigos, uno de forma regular como el triángulo, y el otro de forma irregular.

Como último boceto se dejó el del jugador, ya que este requería más trabajo por ser el puente de entrada del jugador al videojuego; era necesario que tuviera elementos que aportaran en la inmersión del jugador. Por lo cual se inició a bocetar buscando que el jugador se sintiera dentro de una nave visualizando el entorno dentro de ella (figuras 12 y 13).

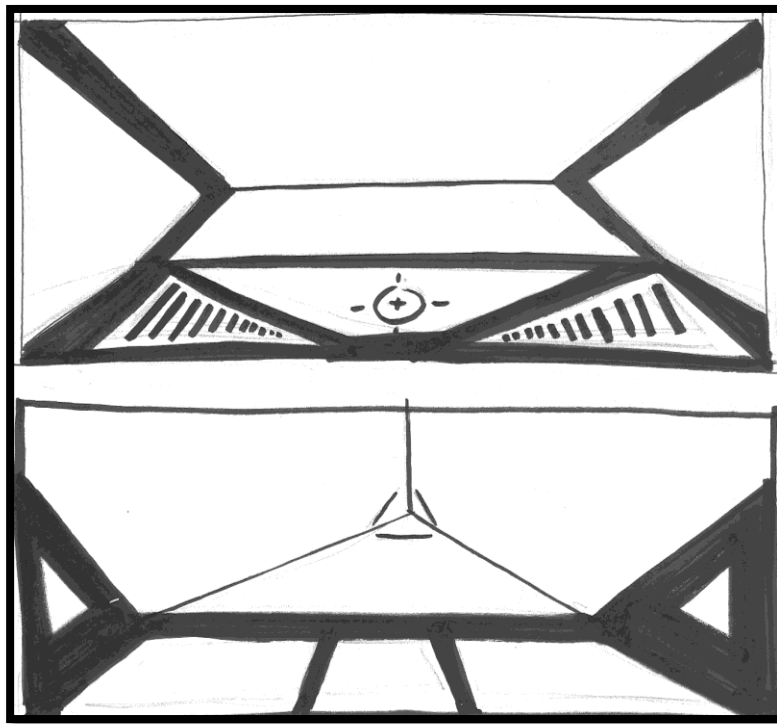


Figura 12. Arte conceptual de la nave del jugador.

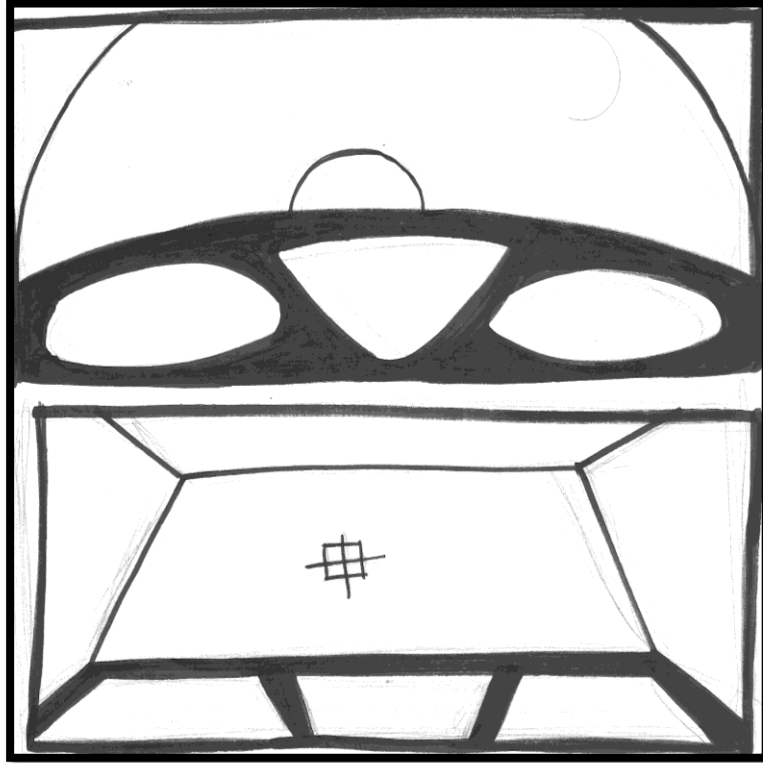


Figura 13. Arte conceptual de la nave del jugador.

En base a los conceptos de la nave, encontramos que la ventana es el factor principal en cuanto a la vista del jugador al escenario, más que los paneles de control de la misma. Por ello al final se decidió usar un panel pequeño con pocos elementos y una ventana en forma hexagonal.

Por ultimo en esta etapa de preproducción se necesita hacer el documento de diseño de videojuego. En donde se redacta de manera específica el funcionamiento del videojuego.

3.1.3 Documento de diseño de videojuego.

El game design document (GDD), es un documento necesario en cualquier producción de un videojuego, un documento de diseño de videojuego actúa como un nexo que conecta y enumera cada aspecto del juego (Sayenko, 2015).

En base a esto nuestro documento está enfocado en las mecánicas del juego y como estas deben funcionar. A continuación, describimos el documento de diseño.

Documento de diseño de videojuego

El objetivo del juego es sobrevivir el mayor tiempo posible, mientras dispara a sus enemigos y esquiva los disparos de ellos. El jugador tiene la cantidad de tres vidas.

Modo de juego.

El jugador deberá sobrevivir el mayor tiempo posible haciendo uso de sus reflejos para moverse dentro del juego. También podrá realizar la acción de disparar con la cual puede destruir a sus enemigos.

Reglas.

- El jugador pierde cuando se le acaben sus tres vidas.
- El jugador solo puede moverse a la izquierda y derecha, en forma radial.
- Por motivo de hacer el juego como medio para un experimento el cual trata de medir la inmersión del jugador, se dejó fuera el factor de victoria.
- La única manera de hacerle daño al jugador es mediante el disparo de los enemigos.
- Solamente el enemigo de color rojo puede disparar.

Flujo.

El flujo del juego es muy sencillo, al iniciar el jugador podrá moverse de forma radial por la escena, después de cierta cantidad de segundos se iniciarán a instanciar los enemigos, al inicio solo serán los enemigos amarillos. Después de otra cierta cantidad de tiempo iniciamos a instanciar enemigos en rojo con un porcentaje bajo de aparición, que conforme pase el tiempo se va aumentando.

El jugador deberá tratar de sobrevivir usando solamente su movimiento y destruyendo a los enemigos. El juego acaba cuando el jugador recibe tres disparos que equivalen a su vida; al terminar el juego el jugador podrá elegir entre repetir la sesión de juego o salirse de la misma.

3.2 Producción

Después de dar terminada la etapa de preproducción, se inició la producción del videojuego con la información generada a partir de la etapa anterior. Se inició con la creación de los assets, que son todos los elementos que contiene el videojuego. Cabe destacar que se usó el programa de Autodesk Maya para crear los modelos de los enemigos y de la nave del jugador.

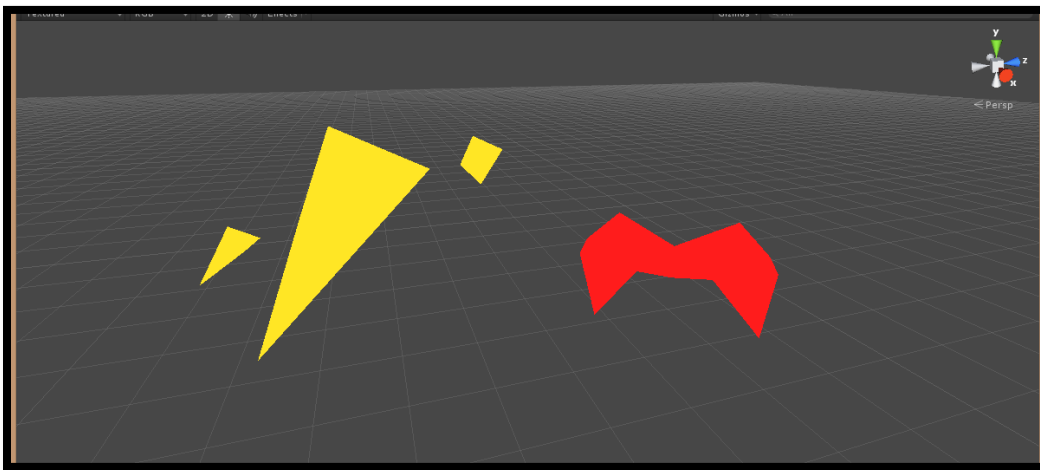


Figura 14. Modelos de las naves de los enemigos.

Como podemos observar en la figura 14 los modelos de los enemigos fueron hechos a partir de sus conceptos, quedando en formas geométricas; de igual forma para representar la diferencia de los dos enemigos se le añadió a cada uno un color representativo.

La figura 15 representa a la nave del jugador, como se había mencionado antes, se buscó una representación significativa en las ventanas y un panel con pocos elementos.

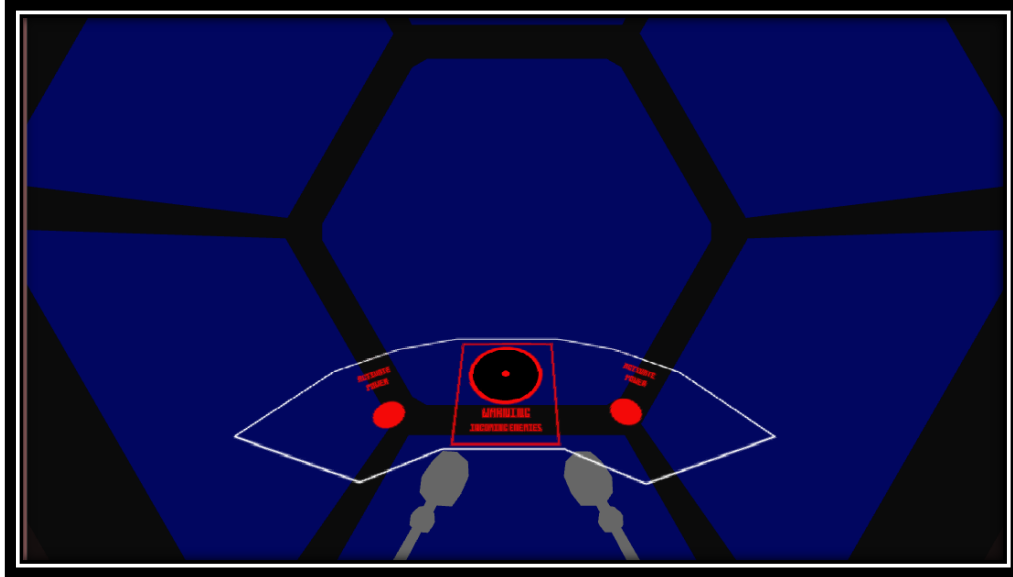


Figura 15. Modelo de la nave del jugador.

Al terminar con los assets se inició la etapa de construcción del juego empezando por el desarrollo del escenario.

3.2.1 Desarrollo del escenario.

En esta sección iniciamos la construcción del escenario y el comportamiento del mismo dentro del juego. El escenario se creó dentro de Unity usando solo el objeto primitivo quad. Los quads se usaron por su pequeña cantidad de triángulos que tienen (2 triángulos), con lo cual podemos usar varios sin preocuparnos por el impacto en el rendimiento del juego.

El desarrolló el escenario del juego fue de la siguiente manera:

1. Se crearon seis quads, uno por cada fila de nuestro círculo.
2. Cada quad se asignó a un objeto padre, el objeto padre se rotaba en 60° grados en su eje z. Quedando como resultado cada objeto a una distancia de 60° grados.
3. Los seis objetos se agregaron aun solo padre, el cual formaba una sección del escenario.

4. Por último, se duplico el objeto que formaba toda una sección treinta veces, cada sección separada en 3 unidades en el eje z.

Las figuras 16 y 17 muestran el resultado de los pasos anteriores. Se le agrego color a las secciones para distinguirlas al momento de que se movieran hacia el jugador.

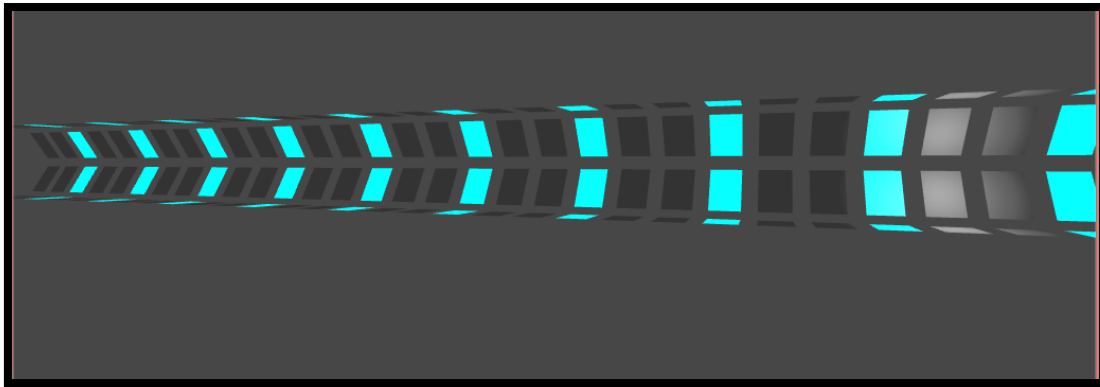


Figura 16. Escenario del videojuego, vista lateral.

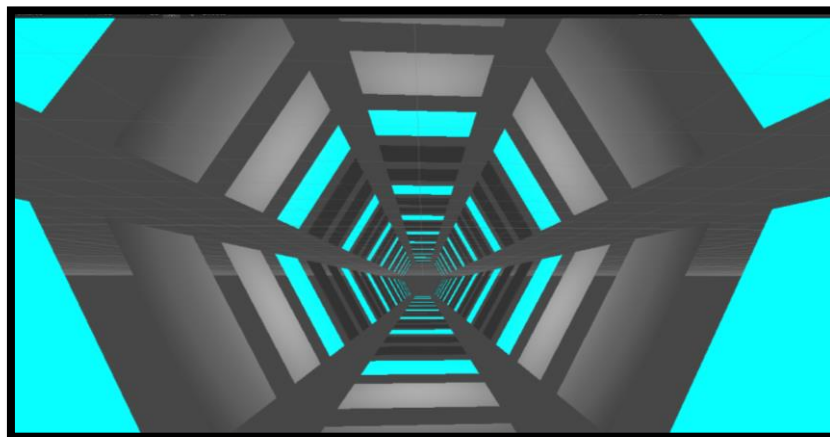


Figura 17. Escenario del videojuego, vista del jugador.

Después de tener armado el escenario, se procedió a programar su comportamiento. Para el efecto de movimiento del jugador, se optó por

mover el escenario hacia el jugador, en vez de mover al jugador hacia el escenario.

Para ello lo que hizo fue mover cada sección del escenario hacia el jugador y cuando cierta sección llegaba a un límite establecido, esta se colocaba en la posición final del escenario. El comportamiento de las secciones ressemblan al comportamiento de las colas FIFO, pero el primer componente no se elimina si no regresa a la cola en la última posición. Este comportamiento se programó con el siguiente código:

```
void Update () {  
    foreach (Transform t in tunel) {  
        t.Translate(0,0,-0.1f*velocity);  
        if(t.position.z < -9){  
            t.position = new Vector3(0,0,81);  
        }  
    }  
}
```

Figura 18. Código del movimiento de las secciones del escenario.

En esta figura dentro del método *Update*, el cual es un método interno de Unity que se ejecuta en cada frame, hacemos un ciclo con un *foreach* para obtener las transformaciones de cada objeto que se encuentren en el túnel. Los objetos que obtenemos son las secciones del túnel a los cuales le aplicamos un movimiento de translación negativo en su eje z, para que este se mueva hacia el jugador. Agregamos una condición en la que si la posición de objeto es menor a -9, entonces la posición de ese objeto se va al final de todos.

De esta manera terminamos con el desarrollo del nivel y pasamos al desarrollo del jugador.

3.2.2 Desarrollo del jugador.

Para esta parte se colocó dentro de la escena el asset de la nave que se modeló previamente, esta se ubicó al inicio de las secciones del escenario. Después de tener la nave en la posición adecuada, se colocó la cámara del juego en la ubicación del piloto dentro de la nave (figura 19). Parte importante del control del jugador era el movimiento radial que hacia moverse hacia los lados. Este efecto al igual que el movimiento se logró rotando el escenario y no al jugador.

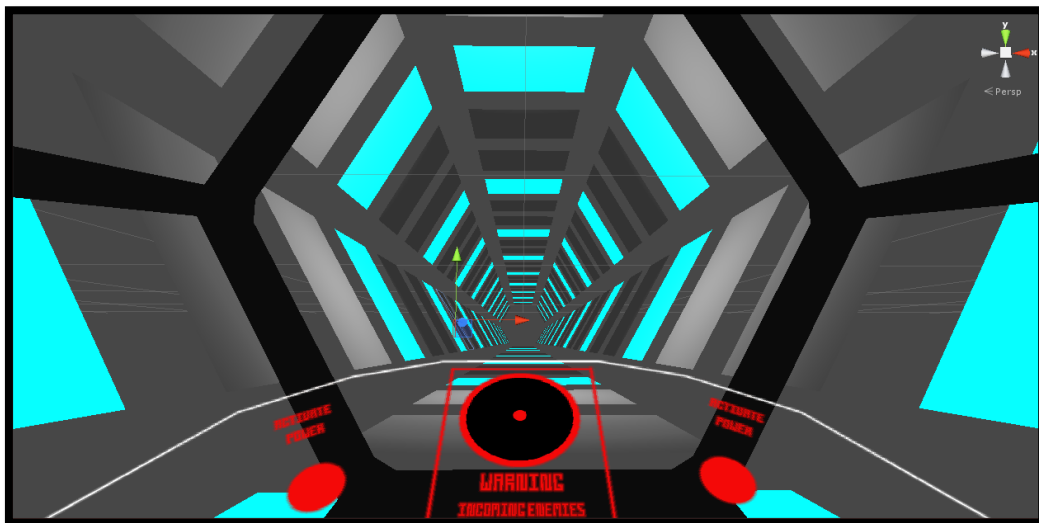


Figura 19. Cabina del jugador.

Antes de trabajar este tipo de movimiento en la escena, se realizaron unas pruebas con un prototipo. Para este prototipo se realizó un pequeño escenario que consistía en cuatro paredes con el modelo de un jugador (figura 20).

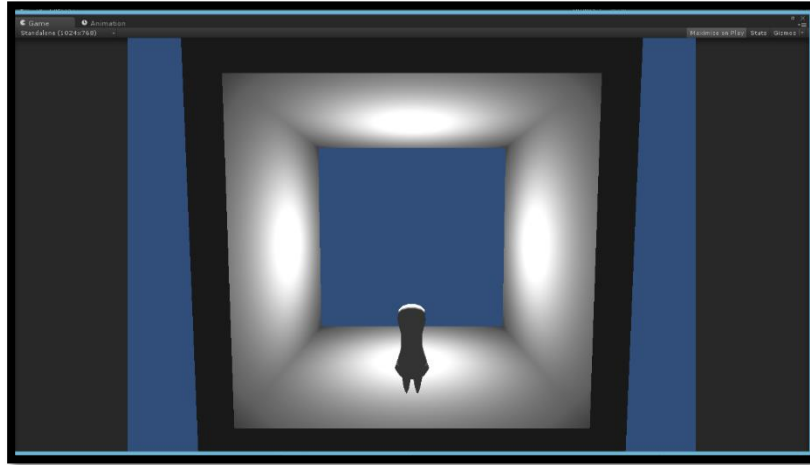


Figura 20. Prototipo del escenario.

En el prototipo el jugador puede moverse a las paredes adyacentes (figura 21), para poder esquivar obstáculos que se agregaron a las paredes mientras el jugador se mueve hacia adelante de manera automática.

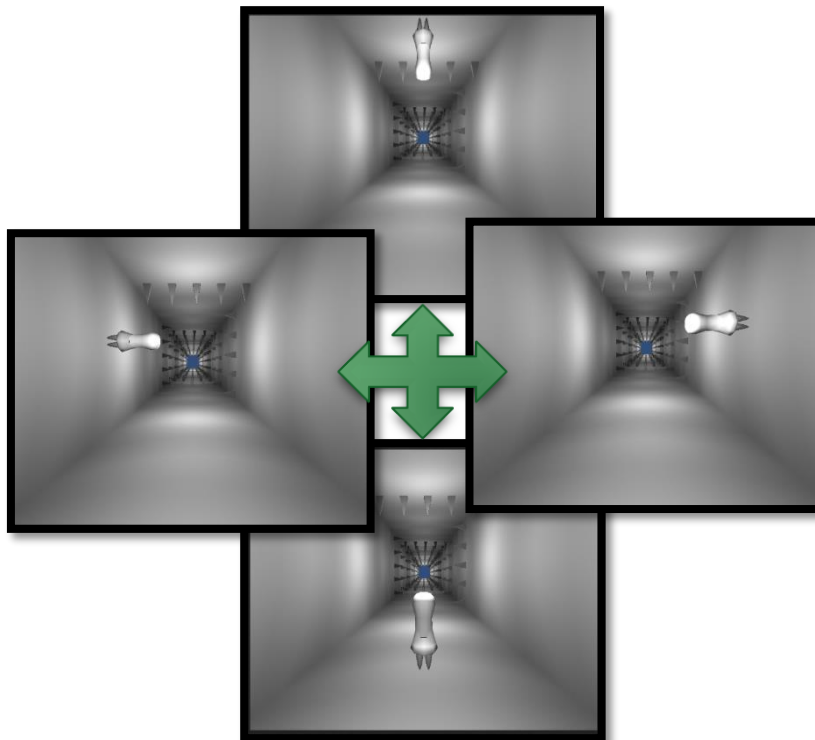


Figura 21. Prototipo realizado para la experimentación de la mecánica de movimiento radial.

Para el movimiento radial se espera al input del usuario, el cual lo hacía al presionar la tecla A o D, las cuales representaban el movimiento a la izquierda y a la derecha respectivamente. Al presionar una de las teclas se entraba a la sección de código respectivo para cada acción, dentro de esta sección se cambiaba el valor de la variable *_rotateControl*, la cual tiene almacenada la rotación actual del escenario, a esta variable se le suma o resta según sea el caso el valor de la variable *rotateValue*, el valor del cual es de 60. También se llama a una función (figura 22) la cual se encarga del movimiento de rotación del escenario.

```
if (Input.GetKeyDown(KeyCode.A))
{
    _rotateControl += rotateValue;
    InvokeRepeating("RotateTunnelRight", 0f, 0.02f);
}
else
    if (Input.GetKeyDown(KeyCode.D))
    {
        _rotateControl -= rotateValue;
        InvokeRepeating("RotateTunnelLeft", 0f, 0.02f);
    }
```

Figura 22. Código del movimiento de las secciones del escenario.

Esta rotación se lleva acabo usando la función *Quaternion.Slerp* (figura 23), la cual realiza una interpolación esférica desde posición actual a la posición nueva en cierta cantidad de tiempo.

```

void RotateTunelRight(){
    Quaternion newRotation = Quaternion.AngleAxis(_rotateControl,
    Vector3.forward);
    transform.rotation = Quaternion.Slerp(transform.rotation, newRotation,
    smooth);
}

void RotateTunelLeft(){
    Quaternion newRotation = Quaternion.AngleAxis(_rotateControl,
    Vector3.forward);
    transform.rotation = Quaternion.Slerp(transform.rotation, newRotation,
    smooth);
}

```

Figura 23. Código de métodos que realizan el movimiento radial del escenario.

Al terminar esta parte solo se le añadió la capacidad de disparo al jugador.

3.2.3 Desarrollo de los enemigos

Los enemigos tienen en común su movimiento en dirección al jugador, la única diferencia radica en la capacidad del enemigo rojo en realizar disparos. En la figura 24 perteneciente al comportamiento de la nave está programado su movimiento en el cual se ejecuta cada frame. Para el disparo se creó una variable que determinaba el tiempo entre disparos, cuando el tiempo del juego sobrepasa el tiempo de disparo, entonces, se procedía a ejecutar el código de disparo.

```

void Update () {
    transform.Translate(0,0, 0.1f * velocity);
    if(Time.time > _timeD)
    {
        _timeD = Time.time + _speedRate;
        _bullet = Instantiate(enemyBullet, spawnBullet.position,
        spawnBullet.rotation) as GameObject;
        _bullet.transform.parent = this.transform;
    }
}

```

Figura 24. Código del método para el movimiento y disparo del enemigo rojo.

Al terminar de añadir todos estos comportamientos a los enemigos, jugador y escenario, pasamos a programar el comportamiento del juego, aquel que define lo que va suceder dentro del juego.

3.2.4 Desarrollo de los controladores del juego.

Para el videojuego se desarrollaron dos controladores específicos, uno para encargarse del crecimiento de la dificultad y el otro controlador se encarga de instanciar a los enemigos y en qué posición del escenario se va a instanciar.

En la figura 25 se encuentra toda la programación dedicada a cambiar la dificultad del juego. Para ir modificando la dificultad, esta se hizo a través del tiempo de juego, mientras el tiempo aumentaba se llaman a eventos en donde se cambian variables que modifican el comportamiento de varios elementos.

Mediante *If* anidado verificábamos cada uno de los casos en donde esperábamos que el tiempo de juego fuera mayor al tiempo del evento, los cuales se establecieron en el método *Start* que se ejecuta una vez durante todo el juego.

Dentro del método *Update* están los *If* para cada caso, cada uno se ejecuta de la siguiente manera:

1. Verificábamos el primer evento el cual pregunta si el tiempo desde que inicio el juego es mayor al tiempo del evento 1 y menor al evento 2.
 - a. Si esto es cierto hacemos los siguientes cambios.
 - i. La velocidad de instanciamiento se cambia a 8, con lo cual hacemos que ninguna nave roja se instancie por el momento.

- ii. El rango de instanciamiento para las naves enemigas se pone en 1, el cual es un aumento considerable ya que iniciamos con rango de 5.
 - iii. La velocidad del enemigo amarillo es aumentada a 6.
- 2. Pasamos al segundo evento, que de la misma manera que el anterior, verificábamos si el tiempo de juego es mayor al evento 2 y si es menor al evento 3.
 - a. Si es cierto se vuelven hacer otros cambios, ajustando la velocidad de instanciamiento para que a partir de este evento se comience a instanciar a los enemigos rojos a una probabilidad baja de salir. También se modifica la velocidad del escenario para que vaya aumentado la presencia de velocidad dentro del juego.

Los eventos siguientes siguen la misma característica de los primeros dos eventos, ajustando valores. Estos valores se fueron ajustando en la etapa de pruebas de la postproducción, donde identificamos los valores adecuados para la dificultad.

```

void Start () {
    _event1 = 15;
    _event2 = 30;
    _event3 = 50;
    _event4 = 70;
    _event5 = 150;
}
void Update () {
    if(Time.timeSinceLevelLoad > _event1 && Time.timeSinceLevelLoad < _event2)
    {
        spawnController.spawnVelocity = 8;
        spawnController.spawnRate = 1f;
        enemy1.velocity = 6f;
    }
    else
        if(Time.timeSinceLevelLoad > _event2 && Time.timeSinceLevelLoad <
        _event3)
    {
        spawnController.spawnVelocity = 7;
        spawnController.spawnRate = 0.8f;
        enemy1.velocity = 6.5f;
        enenemy2.velocity = 4f;
    }
    else
        if (Time.timeSinceLevelLoad > _event3 && Time.timeSinceLevelLoad <
        _event4)
    {
        spawnController.spawnVelocity = 6;
        enemyBullet.velocity = 5f;
        spawnController.spawnRate = 0.4f;
        tunel.velocity = 6f;
    }
    else
        if(Time.timeSinceLevelLoad > _event4 && Time.timeSinceLevelLoad <
        _event5)
    {
        spawnController.spawnVelocity = 5;
        spawnController.spawnRate = 0.2f;
        enemy1.velocity = 9f;
        enenemy2.velocity = 6f;
        enemyBullet.velocity = 7f;
        tunel.velocity = 8f;
    }
    else
        if(Time.timeSinceLevelLoad > _event5)
    {
        spawnController.spawnRate = 0.05f;
        enemy1.velocity = 12f;
        enenemy2.velocity = 7f;
        enemyBullet.velocity = 9f;
        tunel.velocity = 10f;
    }
}

```

Figura 25. Código del script encargado de cambiar la dificultad del juego.

Para el otro controlador (figura 26) se usó un array de transformaciones, en donde guardamos las posiciones de los lugares donde se instanciarán a los enemigos. Dentro de este script se hizo uso de la función *Random.Range(min, max)*, la cual devuelve un valor entre el valor mínimo y el máximo-1. La función se usó en dos ocasiones para sacar de manera aleatoria una ubicación del array y a cuál enemigo instanciar. Se utilizó un *Switch* para separar al enemigo a instanciar.

```
public Transform[] spawnPoints;

void Update () {
    if (Time.timeSinceLevelLoad > _timeD)
    {
        _spawnChosen = Random.Range(0,6);
        _timeD = Time.timeSinceLevelLoad + spawnRate;
        _enemyChosen = Random.Range(0,9);
        if (_enemyChosen < spawnVelocity)
            _enemyChosen = 0;
        else
            _enemyChosen = 1;
        switch(_enemyChosen)
        {
            case 0:
                Instantiate(enemy1, spawnPoints[_spawnChosen].position,
                    spawnPoints[_spawnChosen].rotation);
                break;
            case 1:
                Instantiate(enemy2, spawnPoints[_spawnChosen].position,
                    spawnPoints[_spawnChosen].rotation);
                break;
            default:
                break;
        }
    }
}
```

Figura 26. Script encargado de seleccionar enemigo y posición para instanciarlo al juego.

3.2.5 Sonido

El sonido forma parte del videojuego como elemento que incrementa la inmersión. A través de ellos se distinguen partes del ambiente y mecánicas,

con el cual el jugador podrá identificar de mejor manera lo que está sucediendo.

Se integraron un total de cuatro sonidos al videojuego, tres de ellos son para efectos y uno ambiental.

3.2.6 Integración de Oculus Rift

Para hacer la integración en Unity se requiere tener el plugin de Oculus Rift para Unity que se encuentra <https://developer.oculusvr.com/>. Después de tener el plugin descargado, se procede a realizar la importación a Unity para hacer uso de sus componentes.

Una vez lograda la importación se tienen las carpetas necesarias para trabajar con las cámaras de Oculus Rift. Dentro de los componentes que trae el plugin, los más importantes son los *prefabs* “vr camera y vr controller”, estos son una colección de componentes que son re-utilizables a lo largo del juego. Se usan cualquiera de estos dos componentes para dotar nuestro juego con la funcionalidad para Oculus Rift.

Para el juego se utilizó el *prefab* vr camera, el cual se añadió a la escena en la posición del piloto. Para hacer las pruebas se hicieron dos ejecutables, el primero contenía la cámara normal con la cual se jugaría en una computadora y el otro ejecutable contenía la cámara para usarlo con el Oculus Rift.

3.3 Postproducción.

La postproducción fue la etapa donde se estuvo probando el juego antes de finalizar con el juego e iniciar con las pruebas. En esta etapa principalmente se ajustó la dificultad del juego y algunas otras características que surgieron al probar el juego en el Oculus Rift.

Al probar el juego con los Oculus Rift pudimos notar que nuestra interfaz de usuario que habíamos puesto para informar al usuario que lo habían golpeado, no se sentía bien. Mediante algunas pruebas e investigación de demos para Oculus Rift, notamos la ausencia de interfaz de usuario típica en 2D. En base a esto optamos por cambiar nuestros mensajes 2D a 3D. De esta manera pudimos explorar diferentes maneras de mostrar esos mensajes al jugador.

Los mensajes se colocaron en la ventana de la nave para dar el efecto de que el mensaje estaba en la ventana y dar una sensación más natural al mostrarla de esta manera.

Otro cambio fue el de hacer el movimiento radial del escenario más suave para que de esta manera no provocara mareos. Este fue un síntoma que al hacer las pruebas logramos descubrir. Al hacer el movimiento suave la sensación de mareo fue menor.

3.3.1 Dificultades.

Dentro del desarrollo del juego, se encontraron algunos problemas en el área de programación y diseño. Para la parte de programación resultó difícil conseguir el efecto deseado para el movimiento del jugador, ya que debía ser un movimiento radial de acuerdo al nivel que se propuso. En el área de diseño se dificultó el diseño de la nave del jugador, ya que se necesitaba que el modelo de la nave ayudara a la inmersión, sin perder de vista el diseño retro de la misma.

Se requirió hacer algunas modificaciones en la dificultad después de las primeras pruebas, porque se observó que la curva de dificultad era muy alta, de igual manera algunos detalles que fueron visibles al usar los Oculus Rift se tuvieron que modificar. Tomando en cuenta esta nueva información

se optimizó la dificultad para que el inicio del juego fuera más sencillo, así se daba tiempo a que el jugador se adaptara a los controles.

3.4 Resumen.

En este capítulo se explicó todo el proceso de desarrollo que se llevó a cabo para la creación del videojuego, desde los prototipos iniciales hasta el producto final. Explicando todos los componentes de la preproducción, producción y postproducción.