UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Proyecto Final.

Videojuego: Dinosaurios Espaciales.

Física para programadores de videojuegos.

Guillen de la Torre Mario Alberto.
Valdes Escarrega Francisco Daniel.
Licenciatura en Ciencias Computacionales.

Profesor(a): Eloísa del Carmen García Canseco.

Ensenada B.C.; 26 de mayo, 2021.

Objetivo:

El objetivo principal que se tiene es lograr crear un juego que integre de una forma divertida y fluida la fisica detras de el lanzamiento de proyectiles y campos de gravedad. Uno de los rasgos principales en este enfoque es el lograr que tanto los proyectiles como los campos gravitacionales puedan tener diferentes rangos y masas, esto añade profundidad en las mecánicas y ayuda a romper mecánicas que podrían llegar a ser monótonas.

Nuestros principales desafíos se encuentran en la implementación del sistema de colisiones, las fórmulas de movimientos y fuerzas. Para solucionar estos problemas se analizaron los sistemas incluidos en el motor gráfico y las fórmulas físicas detrás de los fenómenos explorados (tiro parabólico y gravedad), extrapolando de esta forma las soluciones.

Introducción:

Gran parte de nuestra motivación a realizar este juego proviene de nuestro aprecio por los juegos afines a worms, en los cuales jugadores se posicionan estratégicamente para lanzar proyectiles a lo largo de la pantalla unos contra otros, debido a que existe una gran cantidad de juegos como estos decidimos agregar el elemento de la gravedad a nuestro diseño.

La razón por la cual se eligió el fenómeno de planetas y campos de gravedad en vez de otros fenómenos es bastante simple, cualquier tema relacionado con el espacio tiende a atraer más interés, además, esto nos permite agregar elementos visuales más interesantes. La inspiración principal para todo esto proviene del juego del 2012 Angry Birds Space, en donde se implementan de forma exitosa los fenómenos mencionados anteriormente.

El diseño final terminó siendo un compuesto, se une el concepto de lanzar proyectiles contra los otros jugadores de worms y los campos de gravedad y el caminar a lo largo de planetas de Angry Birds Space. Esto logra crear un juego original que cumpla con los objetivos declarados en los párrafos anteriores.

Descripción

Como se mencionó anteriormente el juego consiste en dos jugadores disparando proyectiles uno al otro por turnos. A continuación se explicará el bucle de juego:

1. El juego inicia con el turno del jugador uno. El jugador dos tiene todos sus movimientos y ajustes desactivados.
2. Al jugador actual se le activan las capacidades de caminar a lo largo de su planeta, actualizar el ángulo y velocidad inicial de su tiro (imagen a.1), antes de finalmente presionar el botón de disparo, cuando esto ocurre lo siguiente pasa:
    a. Un proyectil se dispara y el jugador actual pierde la habilidad de moverse o ajustar valores. El proyectil volará hasta:
        i. Colisionar con un planeta. (imagen a.2)
        ii. Colisionar con el otro jugador.
            1. Si este es el caso entonces el juego termina y se declara al jugador que lanzó el proyectil como el ganador. (imagen a.3)

iii. Hasta que pase más de 3 segundos en el aire.

b. Una vez esto suceda el proyectil explota y será el turno del otro jugador.

3. El paso 2 se repite indefinidamente hasta que un jugador sea declarado ganador, una vez que esto ocurre se presenta la pantalla final y se podrá regresar al menú principal.

A lo largo del juego se cambian las siguientes variables:

- PolarMovement.cs:
  - Vector 2 pos: Se utiliza para definir la posición del jugador en coordenadas polares, de esta forma podemos moverlo alrededor de un planeta mucho más fácil. Se modifica en el paso 2.
- Disparar.cs:
  - float Vo: Se utiliza para definir la velocidad inicial en el momento de disparar un proyectil. Se modifica en el paso 2.
  - float Angulo Se utiliza para definir el ángulo inicial en el momento de disparar un proyectil. Se modifica en el paso 2.
- ProyectileMovement1.cs:
  - Vector2 P: Se utiliza para tener una variable que maneje la posición antes de asignarlo a la posición del proyectil. Se modifica en el paso 2a. Esta variable se imprime al final del turno.
  - Vector2 V: Esta variable contiene la velocidad actual que se le sumará a la variable P. Se modifica en el paso 2a. Esta variable se imprime al final del turno.
  - Vector2 A: Esta variable contiene la aceleración actual que se le sumará a la variable V. Se modifica en el paso 2a. Esta variable se imprime al final del turno.
  - float F: En esta variable se guarda la fuerza de la gravedad que se calcula. Se modifica en el paso 2a.
  - float time: Esta variable contiene el tiempo que ha pasado desde que se lanzó un proyectil, esto sirve para encontrar el punto en el que este debe de ser destruido.

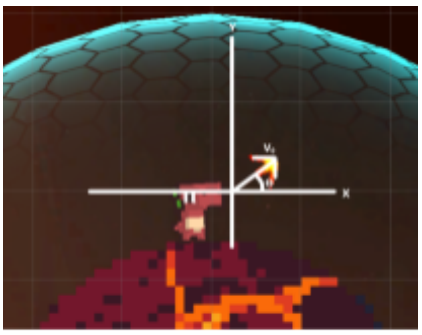


Imagen a.1 - Ajuste de disparo

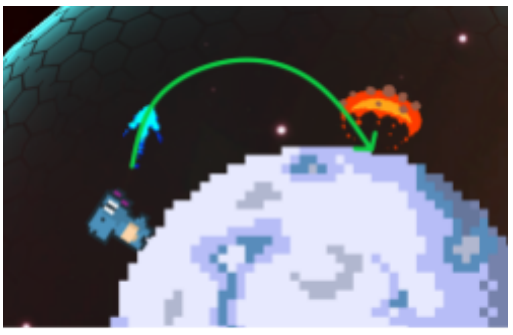


Imagen a.2 - Trayectoria de disparo.

Imagen a.3 - Pantalla final.

Desarrollo

Set up:

Para comenzar a trabajar en el proyecto primero se creó el repositorio en github. En el momento de la creación se seleccionó la opción de crear un archivo README.mb y .gitignore, este último con la opción de Unity.

Una vez creado se debe de crear una carpeta local en donde se desea guardar el repositorio localmente, y se realiza el comando git clone seguido del enlace al repositorio para enlazar la carpeta local con el github.

Para crear el juego se necesita tener instalado Unity Hub, si no se tiene instalado se puede descargar el instalador en el siguiente enlace https://unity3d.com/es/get-unity/download. Posteriormente se creó el proyecto de Unity 2D, en la versión que se desee (cabe recalcar que la versión usada por nosotros fue 2020.3.4f1), posteriormente se tiene que guardar en la carpeta creada para el repositorio. Una vez creado el proyecto se realiza un push para subir el proyecto en blanco al repositorio.

Movimiento:
Jugador:

Para el movimiento del personaje se siguieron los siguientes pasos:

- Declaración de variables.
  -
```
// Esta es la velocidad del personaje.
public float speed;
// Este es transform del planeta sobre el cual esta el personaje.
public Transform planetaCentro;
// Esta es la altura del personaje en relacion con el planeta.
public float altura;

// Este representa la posicion en valores polares
private Vector2 pos;

// Una relacion con el objeto que contiene el GameManager
private GameManager gameManager;

//Esto es referencia al objeto que representa visualmente al jugador y su respectivo control de animacion.
private Animator animator;
public GameObject playerSprite;
```

- En la función Awake(), que ocurre incluso antes que la función Start(), se asignan las referencias al animator, GameManager, se le da valor a pos y ocurre una iteración de la función GeneralPolarMovement(), esto logra posicionar a los personajes de forma correcta en un primer momento.

```
//Esta funcion ocurre en el inicio del juego, antes que la funcion Start
void Awake(){
    //Se asignan los valores correspondientes a las variables definidas anteriormente.
    animator = playerSprite.GetComponent<Animator>();
    gameManager = FindObjectOfType<GameManager>();
    pos = transform.position;
    //Se inicia rapidamente el movimiento del jugador para evitar errores visuales.
    GeneralPolarMovement();

}
```

- En la función Update() se manda a llamar a la función GeneralPolarMovement, además, se verifica si el jugador está presionando a las teclas correspondientes a movimiento, esto se usa para controlar las animaciones y la rotación del jugador, si se quiere mover a la izquierda se rota el jugador 180 grados, de no ser así se invierte esta rotación.

```
void Update()
{
    //Manda a llamar a la funcion que se encarga del movimiento en general
    GeneralPolarMovement();

    /* Esta serie de ifs se encargan de mostrar el movimiento visualmente en el juego
     * Rotacion:
     * Si el jugador esta presionando la tecla indicativa de un movimiento hacia la izquierda entonces la
     * se realiza una rotacion de acuerdo a esto, si es a la derecha se reinicia la rotacion.
     * Movimiento:
     * Si el jugador esta presionando cualquier tecla indicativa de un movimiento entonces se actualiza a verdadero
     * el booleano que controla la animacion de movimiento dentro del controllador de no ser asi se regresa a falso.
     */
    if (Input.GetAxis("Horizontal") != 0)
    {
        if (Input.GetAxis("Horizontal") >= 0)
        {
            playerSprite.transform.localRotation = Quaternion.Euler(new Vector3(0, 0, 0));
        }
        else if (Input.GetAxis("Horizontal") < 0)
        {
            playerSprite.transform.localRotation = Quaternion.Euler(new Vector3(0, 180, 0));
        }
            animator.SetBool("Moving", true);
    }
    else {
        animator.SetBool("Moving", false);
    }
}
```

- En la función GeneralPolarMovement() se obtiene la dirección que debería tomarse como "arriba" según la posición del jugador a lo largo del planeta, esta dirección se le asigna a transform.up para que se vea dicha rotación. También se hacen los cálculos del movimiento en coordenadas polares, para esto se utiliza la variable pos, esta representa el vector de posición en coordenadas polares, por eso solo se cambia su valor de "y" para representar el movimiento a lo largo del planeta, el valor de "x" es estático y solo se le asigna el valor de la altura (relativa al planeta, imagen 1a) del personaje, posteriormente se calculan las coordenadas cartesianas basándose en estos valores y se asignan al personaje.

```
//Esta funcion se encarga del movimiento en general.
void GeneralPolarMovement()
{
    //Esto obtiene la direccion que deberia tomarse como "arriba" segun la posicion del jugador en el planeta
    //cuando esto se le asigna al transform.up el jugador rota conforme a esto.
    transform.up = transform.position - planetaCentro.position;

    //Como los valores de la variable "pos" representan el vector en coordenadas polares, para poder moverme "Horizontalmente" alrededor del centro
    //Tengo que aumentar el valor del angulo, en este caso sera el valor representado por "y" dentro del vector
    //Ademas para poder moverme "Verticalmente" respecto del centro
    //Tengo que aumentar el valor del radio, en este caso sera el valor representado por "x" dentro del vector
    pos.y += Time.deltaTime * (speed) * Input.GetAxis("Horizontal");
    pos.x = altura;

    //Despues de aumentar dicho angulo aplico la funcion de coordenadas polares
    transform.position = (Vector2)planetaCentro.position + new Vector2(pos.x * Mathf.Sin(pos.y), pos.x * Mathf.Cos(pos.y));
}
```

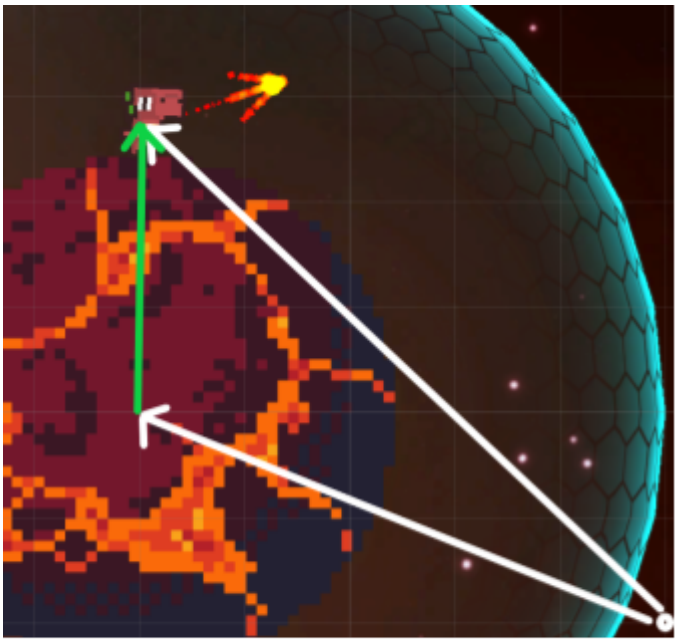


Imagen 1a

- En la función OnCollision(Gameobject coll), que ocurre cuando se detecta una colisión con el jugador, se verifica que personaje es el que recibió la colisión y con que, basándose en esto podemos declarar un ganador y "Destruir" al jugador actual.

```
//Esta funcion se ejecuta cuando se detecta una colision
public void OnCollision(GameObject coll)
{
    //Si el proyectil del jugador 2 colisiona con el jugador 1 entonces el jugador 2 gana y viceversa.
    if(coll.tag == "ProjectPla2" && this.tag == "Player1")
    {
        gameManager.Win(2);
        Destroy(this.gameObject);
    }
    else if(coll.tag == "ProjectPla1" && this.tag == "Player2")
    {
        gameManager.Win(1);
        Destroy(this.gameObject);
    }
}
```

Game Manager:

Cambio de turno:

- Primero se definen las siguientes variables:
  -
```
//Lista de todos los jugadores
//Jugador 1 = 0
//Jugador 2 = 1
public GameObject[] Jugadores;

//Aqui se guardaran la lista de los objetos que funcionaran como fondo en el juego.
    //Fondo del Jugador 1 = 0
    //Fondo del Jugador 2 = 1
public GameObject[] Fondos;
```

- Función Start():
  - Se manda a llamar la función ChangeTurn con un 0 para que el juego inicie con el primer jugador al mando.
  -
```
void Start(){
    //Se encuentra la instancia de menu principal que se encuentre en escena.
    menuPrincipal = FindObjectOfType<MenuPrincipal>();

    //Para comenzar el juego se utiliza la funcion de ChangeTurn dando control al primer jugador.
    ChangeTurn(0);
}
```

- Función ChangeTurn(int jA):
  - Se inicia la corrutina change background.
  - Se activan las funcionalidades de moverse y disparar al jugador jA.
  - Se desactivan las funcionalidades de moverse y disparar al otro jugador.
  -
```
public void ChangeTurn(int jA)
{
    //Se ejecuta la corutina definida para cambiar el fondo al del jugador correspondiente.
    StartCoroutine("changeBackground", jA);

    /*Las siguientes lineas de codigo activan el movimiento y habilidad de disparo a el jugador correspondiente
     * y lo desactivan al otro.
    */
    Jugadores[jA].GetComponent<Disparo>().enabled = true;
    Jugadores[jA].GetComponent<PolarMovement>().enabled = true;

    Jugadores[Mathf.Abs(jA - 1)].GetComponent<Disparo>().enabled = false;
    Jugadores[Mathf.Abs(jA - 1)].GetComponent<PolarMovement>().enabled = false;

}
```

- Corrutina changeBackground(int a)
  - Este contiene dos fors, uno va de 0 a 1 lentamente y el otro va de 1 a 0 lentamente, asignamos este valor del for a el alpha de las imágenes de fondo, el for que va de 0 a 1 cambia el valor del fondo correspondiente del nuevo jugador y el que va de 1 a 0 cambia el valor del otro fondo.

Esta es una corrutina para que el cambio pueda ser gradual, es por esto que se agrega la línea yield return null, cada frame dentro del juego va a llegar hasta esta línea y pasará al siguiente. Si se hubiera definido como función y no corrutina el cambio sería instantáneo.

```
IEnumerator changeBackground(int a) {

    /*Los siguientes fors aumentan el valor de alpha de la imagen indicada y disminullen el de la otra hasta que
     * el de la primera es 1 y la segunda es 0.
    */
    for (float k = 0; k <= 1; k += 0.01f){
        Fondos[a].GetComponent<SpriteRenderer>().color = new Color(1, 1, 1, k);
        yield return null;
    }
    for (float j = 1; j >= 0; j -= 0.01f){
        Fondos[Mathf.Abs(a-1)].GetComponent<SpriteRenderer>().color = new Color(1, 1, 1, j);
        yield return null;
    }


}
```

- Función DisableMovement():
  - Se desactivan las funcionalidades de moverse y disparar a ambos jugadores.

```
public void DisableMovement()
{
    /*Para desactivar el movimiento y habilidad de disparo podemos acceder directamente a los scripts
     * de cada jugador y desactivarlos.
    */
    Jugadores[0].GetComponent<Disparo>().enabled = false;
    Jugadores[0].GetComponent<PolarMovement>().enabled = false;
    Jugadores[1].GetComponent<Disparo>().enabled = false;
    Jugadores[1].GetComponent<PolarMovement>().enabled = false;
}
```

- Ganar:
  - Se definen las siguientes variables:
  - Se utiliza hace la referencia al menú principal para poder obtener las funciones de menús.

```
//Aqui se guarda el objeto que representara el menu que aparrece cuando se gana.
public GameObject menuGanar;
//Esto se utiliza para modificar el texto cuando se gana, como se utiliza otra
//paqueteria para este texto se agrega de esta forma.
public TMPro.TextMeshProUGUI textGanar;

//Se hace referencia al script de MenuPrincipal para poder utilizar sus funciones.
private MenuPrincipal menuPrincipal;
```

- Función Win(int a):
  - Se cambia el texto de ganar para que corresponda con el jugador ganador, además se muestra el menú principal.

```
public void Win(int a) {
    //Se cambia el texto para concordar con el ganador y se muestra el menu de fin de juego.
    textGanar.text = "Ganaste jugador " + (a);
    menuPrincipal.Mostrar(menuGanar);
}
```

Proyectiles

Invocación de proyectil:
Para lograr una creación nueva de un proyectil se utilizaron los siguientes puntos:

- Declaración de variables a utilizar
  - ```
    public ProjectileMovement1 projectile;
    public GameObject flechaDireccion;
    public float Vo;
    public float Angulo;
    private ColliderController colliderController;

    public TextMeshProUGUI posicion_txt;
    public TextMeshProUGUI velocidad_txt;
    public TextMeshProUGUI aceleracion_txt;

    public float SpeedOfChangeVelocity;
    public float SpeedOfChangeAngle;

    private Animator animator;
    private GameManager gameManager;

    public bool m_isAxisInUse = false;
    ```

- Función Start: Esta se manda a llamar en cuanto un objeto con el script es generado. Se busca un objeto GameManager y se asigna a una variable para tener su referencia. De igual manera con el animador y el collider manager. Y se inicializa la flechaDireccion en su posición.
  - ```
    void Start()
    {
        gameManager = FindObjectOfType<GameManager>();
        animator = GetComponentInChildren<Animator>();
        flechaDireccion.transform.localScale = new Vector3(Vo / 10, 2, 2);

        colliderController = FindObjectOfType<ColliderController>();
    }
    ```

- Función Update: Los primeros dos if que se encuentran dentro de la función, se utilizan para que el jugador solo pueda disparar una vez. Dentro del primer if se encuentran las opciones para generar el disparo, como primera acción se realiza la animación de disparo, se instancia el objeto de proyectil y se le agrega su colisionador. Posteriormente se le asignan los valores iniciales y los que se imprimirán en las etiquetas de la escena.

  Las condiciones siguientes de teclas modifican la velocidad, dada una variable si es que se desea incrementar más rápido la velocidad. Como se observa en la imagen, en las dos condicionales se cambia la velocidad inicial y el tamaño de la flechaDireccion, para mostrar gráficamente el cambio de velocidad.
  Para asignar el ángulo en el cual se disparará el proyectil se utiliza el eje "vertical" ya que se utilizarán las teclas w y s, ya que Unity las delimita como predeterminadas para este eje. De igual manera que el cambio de velocidad se tiene una variable que permite cambiar la velocidad de cambio dentro del ángulo, y se cambia la rotación de la flecha, sin embargo se modifica el centro con la opción RotateAround.

```
if (Input.GetAxis("Fire1") != 0)
{
    if (m_isAxisInUse == false)
    {
        if (Vo != 0)
        {
            animator.SetTrigger("Shoot");
            ProjectileMovement1 pro = Instantiate(projectile, transform.position, transform.rotation) as ProjectileMovement1;
            colliderController.Circles.Add(pro.GetComponent<CircleColliderSim>());
            pro.Vo = Vo;
            pro.Angulo = (((Angulo + transform.eulerAngles.z) * Mathf.PI) / 180);
            pro.posicion_txt = posicion_txt;
            pro.velocidad_txt = velocidad_txt;
            pro.aceleracion_txt = aceleracion_txt;

            gameManager.DisableMovement();

        }
        m_isAxisInUse = true;
    }



}
if (Input.GetAxisRaw("Fire1") == 0)
{
    m_isAxisInUse = false;
}
if (Input.GetKey(KeyCode.E))
{
    Vo = Mathf.Clamp(Vo + SpeedOfChangeVelocity * Time.deltaTime, 0, 30);
    flechaDireccion.transform.localScale = new Vector3(Vo/10,2, 2);
} else if (Input.GetKey(KeyCode.Q))
{
    Vo = Mathf.Clamp(Vo - SpeedOfChangeVelocity * Time.deltaTime, 10, 30);
    flechaDireccion.transform.localScale = new Vector3(Vo/10, 2, 2);
}
if (Input.GetAxisRaw("Vertical") != 0)
{
    Angulo = Angulo + Input.GetAxisRaw("Vertical") * SpeedOfChangeAngle * Time.deltaTime;
    flechaDireccion.transform.RotateAround(transform.position, flechaDireccion.transform.forward, Input.GetAxisRaw("Vertical") * SpeedOfChangeAngle * Time.deltaTime);
}
```

- Comienzo de movimiento: Para comenzar el movimiento de proyectil, se asigna un gameManager (se encarga del sistema de turnos), un vector P de posición, uno de velocidad y un peso predeterminado en 1. Posteriormente se buscan todos los camposGravitatorios que se encuentran en la escena. Y se imprimen los valores de dichos vectores.

```
void Start()
{
    gameManager = FindObjectOfType<GameManager>();
    P = new Vector2(transform.position.x, transform.position.y);
    V = new Vector2(Vo * Mathf.Cos(Angulo), Vo * Mathf.Sin(Angulo));
    peso = 1;

    //Lista en donde se encuentran todos los planetas
    camposGravitatorios = FindObjectsOfType<CampoGravitatorio>();

    posicion_txt.text = "Posicion = " + P;
    velocidad_txt.text = "Velocidad = " + V;
    aceleracion_txt.text = "Aceleracion = " + A;
}
```

- Función Update: El fragmento de código que se muestra en la imagen, verifica si el tiempo transcurrido desde el disparo es mayor al tiempo máximo de turno. Si esta condición es verdadera, se cambia de turno, realiza la animación de explosión y lo destruye.

```
//Cronometro para que la bala se destruya y cambie el turno
time += Time.deltaTime;
if (time > maxTime) {
    CambiarTurno();
    GameObject exp = Instantiate(ImpactExplosion[2], transform.position, transform.rotation) as GameObject;
    Destroy(gameObject);
}
```

- Verificación planetaria: Dentro de los condicionales anteriores se verifica si es que se encuentra dentro de un planeta o no, tomando como variable bandera, "dentro" que inicialmente es 0, se realiza un recorrido por cada campo encontrado anteriormente, y se realiza una operación de distancia para verificar si se encuentra dentro de este mismo. Si es así se cambia el valor de dentro, se asigna la variable g a la gravedad de dicho campo y se crea un vector A (aceleración) unitario con dirección al centro del campo. Si la condición resulta ser falsa significa que se encuentra dentro de un campo, y se realiza una operación de distancia para verificar si en algún momento sale de este, si llega a salir se asigna g a 0 ya que en el espacio no hay gravedad y el vector A se reinicia.

```
if (dentro == 0){
    foreach (CampoGravitatorio campo in camposGravitatorios) {
        if (Vector2.Distance(campo.transform.position, transform.position) < campo.size){
            dentro = 1;
            g = campo.gravedad;
            A = (campo.transform.position - transform.position).normalized;
            center = campo.transform;
        }
    }
}
else if (Vector2.Distance(center.transform.position, transform.position) > center.GetComponent<CampoGravitatorio>().size)
{
    P = transform.position;
    dentro = 0;
    g = 0;
    A = new Vector2(0, 0);
}
```

- A continuación se mostrarán los diferentes comportamientos dependiendo si se encuentra dentro de un planeta o fuera.

Movimiento Espacial

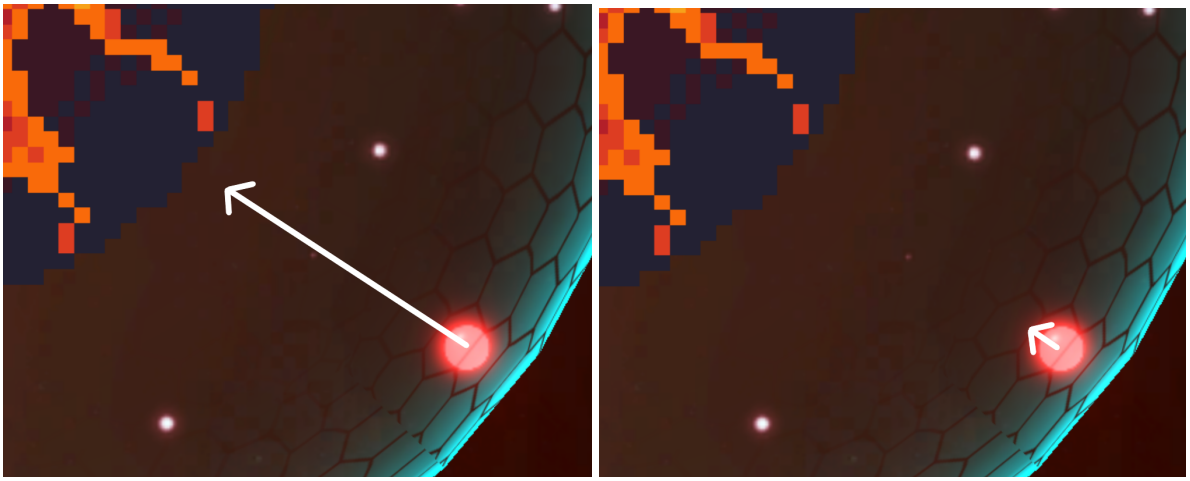
```
if (dentro == 1){PlanetMovement();} else{
    //Esto se encarga del movimiento cuando el objeto se encuentra fuera de un planeta
    P.x = P.x + V.x * Time.deltaTime;
    P.y = P.y + V.y * Time.deltaTime;
    transform.position = P;
}
```

En la imagen anterior se muestra una condición, la cual verifica con el valor bandera obtenido anteriormente, si es que se encuentra dentro de un planeta. Si el resultado es falso, las siguientes líneas muestran que debido a que se encuentra en el espacio el movimiento es lineal, y sin una aceleración, por lo tanto solo se utiliza la velocidad asignada actualizando su posición multiplicando por el tiempo entre frame y frame.

Movimiento Gravitacional

```
void PlanetMovement()
{
    A = (center.position - transform.position);
    F = g*((transform.localScale.x*center.localScale.x*60)/Mathf.Pow(A.magnitude, 2));
    V += (F * A.normalized * peso) * Time.deltaTime;
    Vector3 v3 = V;
    transform.position += v3 * Time.deltaTime;
}
```

En la primera línea lo primero a realizar es el cálculo de un vector apuntando desde la posición del proyectil hacia el centro del planeta. Posteriormente se utiliza la fórmula F = Gm1m2/r2 para calcular la fuerza de gravedad dentro del planeta, tomando como masa, la escala del objeto, y radio la magnitud del vector A. Para calcular la velocidad se toma el resultado anterior multiplicado por el vector unitario de A, para que la fuerza de gravedad apunta hacia el centro del planeta y el peso dado, todo multiplicado por el tiempo transcurrido entre frame y frame. Las últimas dos líneas se utilizan debido a que la posición se encuentra en términos de x, y y z por lo tanto se transforma la velocidad a un vector con x, y y z. Y se asigna a la posición multiplicada por el tiempo.



Colliders:

Estructura General: Para el funcionamiento general de este sistema son necesarios los siguientes scripts:

- BoxColliderSim: Este script se colocara dentro de los objetos que requieran un collider de forma cuadrada, contiene un arreglo con un par de valores flotantes, el primero representa la altura y el segundo la anchura, además contiene una función que dibujara un rectángulo verde según estos valores.

```
public class BoxColliderSim : MonoBehaviour
{
    //Este script simulara lo que realiza el box collider de unity
    //Contenera un arreglo de 2 valores flotantes:
    //    0 - Altura
    //    1 - Anchura

    public float[] sizes = new float[2];

    Mensaje de Unity | 0 referencias
    void Start()
    {
        sizes[0] = this.transform.localScale.x;
        sizes[1] = this.transform.localScale.y;
    }

    //Esta funcion dibuja el collider en la pantalla, debido a que el objeto sobre el cual este collider estara puede rotar se
    //genera una matriz de Posicion, Rotacion y Escala basandonos en el objeto original, esta se impone en el dibujo para obtener
    //la posicion y rotacion correspondiente.
    Mensaje de Unity | 0 referencias
    void OnDrawGizmosSelected()
    {
        Matrix4x4 rotationMatrix = Matrix4x4.TRS(transform.position, transform.rotation, transform.lossyScale);
        Gizmos.matrix = rotationMatrix;
        Gizmos.color = Color.green;
        Gizmos.DrawWireCube(Vector3.zero, new Vector3(sizes[0], sizes[1], 0));
    }
}
```

- CircleColliderSim: Este script se colocará dentro de los objetos que requieran un collider de forma circular, contiene solo un valor flotante, este representa el radio del círculo, además contiene una función que dibujara un círculo verde según este valor.

```
public class CircleColliderSim : MonoBehaviour
{
    //Este script simulara lo que realiza el circle collider de unity
    //Contenera un  valor flotante que simularan el radio del circulo
    public float size;

    //Esta funcion dibuja el collider en pantalla
    Mensaje de Unity | 0 referencias
    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(transform.position, size);
    }
}
```

- ColliderController: Este será el encargado de verificar si existen colisiones entre los objetos, contiene dos pares de listas, el primer par consiste en una lista conteniendo todos los BoxColliderSim en escena y otra con todos los CircleColliderSim. El otro par de listas contienen valores de strings que representan los “tags” que deberán ser ignorados en las colisiones, esto se explicará más adelante.

```
//Estas listas contienen todos los colliders que estan dentro del juego
//se requieren listas diferentes para los colliders que tienen forma de cuadrada y circular
public List<BoxColliderSim> Boxes;
public List<CircleColliderSim> Circles;
```

```
//Se ignoraran las colisiones que ocurran entre los objetos que tengan las tags contenidas en estas listas
//El sistema funciona como pares, los tags que tengan el mismo indice en las listas se excluiran entre ellos
//Para esto las listas siempre tendran que ser del mismo tamaño.
public List<string> TagsToExcludeP1;
public List<string> TagsToExcludeP2;
```

Detección de colisiones: Lo siguiente se realiza dentro del script ColliderController:

- Para la detección de colisiones entre círculos y cuadrados:
    - Se iteran las listas de CircleColliderSim y BoxColliderSim, esto nos da todos los pares de círculos y rectángulos existentes en escena.
    - Se obtiene la posición de CircleColliderSim en relación con BoxColliderSim, esto generaliza el problema y evita conflictos en el momento en el que el cuadrado rota (lo cual ocurre constantemente debido a que el jugador camina alrededor de los planetas).
    - Se encuentra el punto más cercano al círculo en el perímetro del cuadrado.
    - Calculamos la distancia entre este punto y el centro del círculo, si la distancia es menor o igual al radio del círculo entonces tenemos una colisión.

```
//Como el cuadrado puede rotar es mejor tomar las coordenadas del cuadrado como las del origen
Vector2 C = i.transform.InverseTransformPoint(new Vector3(j.transform.position.x, j.transform.position.y));

//Encuentra los valores de el punto mas cercano del cuadrado al circulo.
float Xn = Mathf.Max(-(i.sizes[0] / 2), Mathf.Min(C.x, +(i.sizes[0] / 2)));
float Yn = Mathf.Max(-(i.sizes[1] / 2), Mathf.Min(C.y, +(i.sizes[1] / 2)));

//De esta forma obtenemos el punto dentro de las coordenadas del circulo para poder calcular su distancia al mismo de forma correcta
float Dx = C.x - Xn;
float Dy = C.y - Yn;

//Si dicha distancia es menor a el radio del circulo entones existe una interseccion
if ((Dx * Dx + Dy * Dy) <= j.size * j.size)
{
    print("Colision!");
    colision_sound.Play();
    //Esta funcion busca, en todos los scripts que hereden de MonoBehaviour dentro de el objeto, una funcion con el
    //nombre "OnCollision" le manda como argumento el objeto con el que colisiono y el ultimo argumento indica que no
    //importa si no encuentra ninguna funcion con ese nombre
    j.gameObject.SendMessage("OnCollision", i.gameObject, SendMessageOptions.DontRequireReceiver);
    i.gameObject.SendMessage("OnCollision", j.gameObject, SendMessageOptions.DontRequireReceiver);
}
```

- Para la detección de círculos:
    - Obtenemos la distancia entre los centros de los círculos.
    - Si la distancia es mayor o igual que la suma de los radios de los círculos entonces tenemos una colisión.

```
//Debido a que en este caso estamos iterando sobre la misma lista que el for exterior debemos de tomar en cuenta el caso en el que
//se este comparando un circulo sobre si mismo.
if (i == j)
{
    continue;
}

//Esto encuentra la distancia entre ambos circulos.
float Dx = j.transform.position.x - i.transform.position.x;
float Dy = j.transform.position.y - i.transform.position.y;
float D = Mathf.Sqrt(Dx * Dx + Dy * Dy);

//Si la distancia entre los centros de los circulos es mayor la suma de el radio de los circulos entonces existe una colision
if (D <= j.size + i.size)
{
    print("Circle Colision!");
    colision_sound.Play();
    //Esta funcion busca, en todos los scripts que hereden de MonoBehaviour dentro de el objeto, una funcion con el
    //nombre "OnCollision" le manda como argumento el objeto con el que colisiono y el ultimo argumento indica que no
    //importa si no encuentra ninguna funcion con ese nombre
    j.gameObject.SendMessage("OnCollision", i.gameObject, SendMessageOptions.DontRequireReceiver);
    i.gameObject.SendMessage("OnCollision", j.gameObject, SendMessageOptions.DontRequireReceiver);
}
```

Uso de la colisión: En el momento en el que ocurre una colisión se busca en todos los scripts, de cada objeto involucrado en ella, una función llamada OnCollision (que recibe como argumento el objeto con el que se colisionó) y se ejecuta.

```
//Esta funcion busca, en todos los scripts que hereden de MonoBehaviour dentro de el objeto, una funcion con el
//nombre "OnCollision" le manda como argumento el objeto con el que colisiono y el ultimo argumento indica que no
//importa si no encuentra ninguna funcion con ese nombre
j.gameObject.SendMessage("OnCollision", i.gameObject, SendMessageOptions.DontRequireReceiver);
i.gameObject.SendMessage("OnCollision", j.gameObject, SendMessageOptions.DontRequireReceiver);
```

Ejemplo del uso de la función:

```
public void OnCollision(GameObject coll)
{
    //Primera mente se cambia de turno y se destruye el objeto de proyectil
    CambiarTurno();
    Destroy(this.gameObject);

    //Esto se utiliza para generar la explosion, como existen diferentes explosiones para
    //cuando el proyectil choca contra un objeto y cuando choca con un jugador, por lo que esto
    //tiene que ser verificado
    GameObject Explosion;
    Vector3 posImp = transform.position; ;
    if (coll.tag == "Player1" || coll.tag == "Player2")
    {
        posImp = coll.transform.position;
        Explosion = ImpactExplosion[1];
    }
    else {
        Explosion = ImpactExplosion[0];
    }
    GameObject exp = Instantiate(Explosion, posImp, transform.rotation) as GameObject;
    //Esto define la rotacion del explosivo para que este este de acuerdo al planeta en el que se choca.
    exp.transform.up = transform.position - center.position;
}
```

Excepciones en colisiones: Existen casos en los cuales no queremos que una colisión se registre, por ejemplo, como los proyectiles aparecen dentro del jugador al momento de disparar, si no evitamos que esta colisión ocurra los proyectiles harán daño al mismo jugador que lo lanzó, para evitar que esto suceda se hizo lo siguiente:

- Dentro de los fors que iteran entre los scripts de colliders creamos un booleano que inicie en falso.
- Se agrega otro set de fors que iteran sobre las dos listas de strings que representan los tags, dentro de este for se accede a los objetos actuales que contienen a los scripts de colliders en el par de fors exterior, se verifica que sus tags no concuerden con las de las listas de objetos a ignorar (el tag en el índice n de la lista uno no puede colisionar con el tag en el índice n de la lista dos).
- Sí concuerdan los tags entonces el booleano generado anteriormente pasa a verdadero y se utiliza para saltarnos la iteración en el set de fors externo.

```
//Esto sirve para excluir las colisiones de los objetos que contengan los tags definidos anteriormente
bool excluir = false;
for (int h = 0; h < TagsToExcludeP1.Count; h++){
    if(i.gameObject.tag == TagsToExcludeP1[h] && j.gameObject.tag == TagsToExcludeP2[h]){
        excluir = true;
    } else if (i.gameObject.tag == TagsToExcludeP2[h] && j.gameObject.tag == TagsToExcludeP1[h])
    {
        excluir = true;
    }
}
if (excluir) { continue; }
```

Visuales

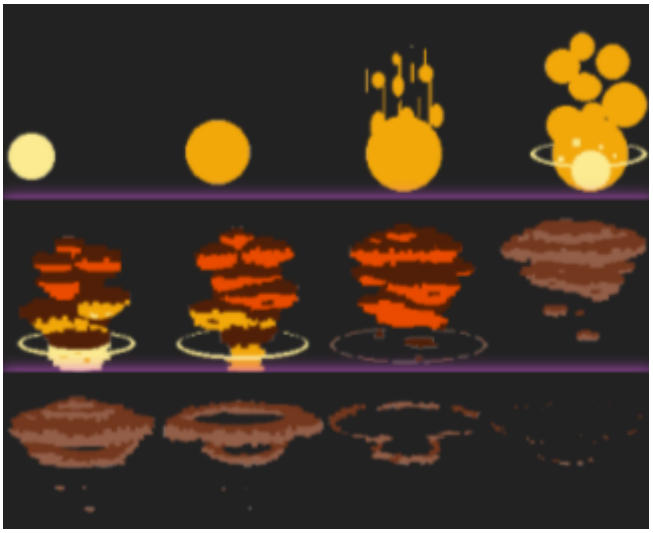
Animación: Para implementar las animaciones primero se agregaron “Animation controllers” a todos los objetos que debían estar animados, se definió un set de animaciones específico a cada uno de los objetos y en base de estos se modificaron los controladores. Se terminan definiendo los siguientes controladores y animaciones:
:

- Jugador: Este controlador, como lo dice su nombre, define las animaciones del jugador, contiene 3 estados:
    - IDLE: Esta es una animación por defecto, debe ocurrir siempre que no exista ningún input por parte del jugador.
    - Moviéndose: Esta animación se deberá incluir siempre que el jugador esté caminando.
    - Disparo: Esta animación deberá ocurrir solo en el momento en que el jugador disparó.

      Para pasar entre animaciones el controlador deben de haber variables dentro del mismo, las variables definidas son las siguientes:

    - Moving: Es una variable de tipo booleana. Cuando esta tiene valores verdaderos o positivos, el estado de animación “Moviéndose” se activará, de otra forma el estado será “IDLE”.
    - Shoot: Es una variable de tipo Trigger. Esta variable funciona de la misma forma que una variable booleana pero solo se activa lo suficiente para suscitar un cambio de estados. En su activación la animación tomará el estado Disparo.

- Explosión: Este controlador define las animaciones de las explosiones, este contiene dos estados:
    - Explotar: Esta es la animación por defecto de una explosión.
    - Placeholder: Este estado no contiene ninguna animación. Además, la transición del estado “Explotar” al estado “Placeholder” tiene un comportamiento específico, este elimina al objeto al terminar la transición, es por esto que el estado “Placeholder” no contiene ninguna animación.



Recursos: Para obtener las imágenes que se utilizaron dentro del juego se recurrieron a itch.io esta es una página web que ofrece recursos como imágenes, animaciones o sonidos de forma gratuita. Los recursos tomados se describen detalladamente en la sección de referencias.

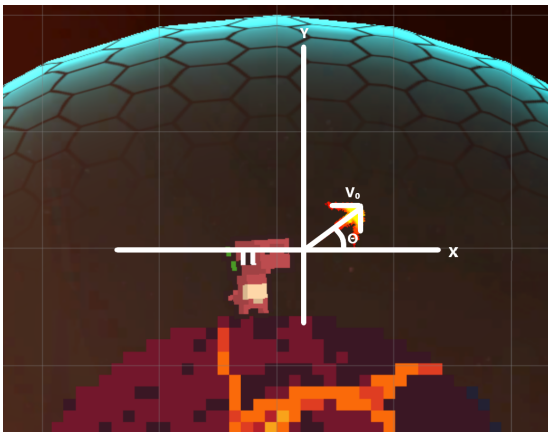
Conclusiones

Uno de los principales problemas que encontramos a lo largo del desarrollo del proyecto fue el lograr replicar los sistemas más simples de física que unity tiene implementados, como el de colisiones entre colliders, incluso aunque nuestro proyecto solo cuenta con 2 tipos de colisiones (entre círculos y entre cuadrados y círculos) fue bastante difícil lograr que el sistema funcionara de forma correcta, e incluso ahora estamos seguros de que se podría mejorar para optimizar el consumo de procesamiento.

Otra complicación que se dio en el momento de implementar el sistema de disparo y gravedad, es el cambio que ocurre en el proyectil cuando entra a un campo de gravedad y cuando sale del mismo causaba bastantes problemas y la falta de documentación para realizar estas cosas sin el sistema de física integrado de Unity no ayudaba a resolver el problema, pero al final logramos que esto funcionara, utilizando vectores para proyectiles y coordenadas polares en el caso del jugador.

A pesar de estos problemas el proceso de desarrollo fue entretenido y el observar como el juego tomaba forma a lo largo del tiempo fue bastante gratificante. Si se retomara el proyecto las mejoras que le harían sería cambiar el sistema que controla el flujo del juego para permitir una mayor cantidad de personajes.

Fotos:



Vector Unitario a centro del planeta

Vector Unitario multiplicado por F

Tomando como (0,0) el punto blanco, se obtiene un vector apuntando hacia “arriba” del planeta

Referencias

Las imágenes utilizadas para los personajes:

- Nombre del creador (Nombre virtual): Arks
    - URL del creador: https://arks.uwu.ai
    - Twitter: https://twitter.com/ScissorMarks
    - URL del recurso: https://arks.itch.io/dino-characters

Las imágenes utilizadas para las flechas:

- Nombre del creador (Nombre virtual): XYEzawr
    - URL del creador: https://itch.io/profile/xyezawr
    - Twitter: https://twitter.com/XYEzawr
    - URL del recurso: https://xyezawr.itch.io/gif-free-pixel-effects-pack-15-magick-arrows

Las imágenes utilizadas para las explosiones:

- Nombre del creador (Nombre virtual): Ansimuz
    - URL del creador: http://ansimuz.com/site/
    - Twitter: https://twitter.com/ansimuz
    - URL del recurso: https://ansimuz.itch.io/explosion-animations-pack

Las imágenes utilizadas para el fondo:

- Nombre del creador (Nombre virtual): DinvStudio
    - URL del creador: https://itch.io/profile/dinvstudio
    - Twitter: https://twitter.com/DinvStudio
    - URL del recurso: https://dinvstudio.itch.io/dynamic-space-background-lite-free

Las imágenes utilizadas para los planetas:

- Nombre del creador (Nombre virtual): Helianthus Games
    - URL del creador: https://itch.io/profile/helianthus-games
    - Twitter: https://twitter.com/HelianthusGames
    - URL del recurso: https://helianthus-games.itch.io/pixel-art-planets

La música utilizada:

- Nombre del creador: Serjo de lua | Keys of moon music
    - URL del creador: https://soundcloud.com/keysofmoon
    - URL del recurso: https://soundcloud.com/keysofmoon/voice-of-eternity-hybrid-space-music-free-download
    - Creative Commons Attribution 3.0 Unported License: https://creativecommons.org/licenses/by/3.0/
    -