MEMORIA

BUSCA PC, PROYECTO 1EVA

DESCRIPCIÓN BREVE

Todo sobre Busca Pc un juego didáctico de aventura, realizado con C# WPF

ROBERTO JOAO TIRÓN
2DAM DESARROLLO DE INTERFACES

FECHA

10/11/2023

Contenido

DESCRIPCION

"BUSCA PC" es un juego de aventuras en tercera persona 2D TOPDOWN VIEW que te sumerge en el mundo de la tecnología. Eres un niño curioso en busca de una emocionante misión: recolectar las 5 piezas esenciales de una computadora (Torre, PSU, RAM, GPU, CPU, Placa Base) en 5 niveles únicos, cada uno representando un entorno y un desafío diferentes. El juego comienza con un nivel de tutorial que te familiariza con los controles y las mecánicas.

Todo está orientando con el fin de familiarizar al usuario con los distintos componentes que forman un ordenador de una manera divertida y entretenida.

FUTUROS Y MERCADOS

Después de un riguroso estudio de mercado centrado en el mercado de habla española, he identificado un nicho de alto potencial que, sorprendentemente, aún no ha sido explotado por ninguna otra empresa o desarrollador. El juego que he creado se destaca por su singular género y estilo, lo que lo convierte en una propuesta única en un mercado saturado de opciones similares.

Fuentes consultadas:

https://www3.gobiernodecanarias.org/medusa/ecoescuela/recursosdigitales/tag/hardware/

https://wordwall.net/es-ar/community/hardware-software

https://es.educaplay.com/recursos-educativos/1794553-software_y_hardware.html

Este juego, a pesar de su simplicidad y enfoque básico, puede llamar la atención de los posibles consumidores. Su simplicidad y atractivo puede ser importante para aquellos que buscan una experiencia de juego sin complicaciones y de fácil acceso.

Es importante destacar que, aunque el juego tiene un gran potencial en su categoría, no tengo la intención de llevarlo a la fase de producción ni de comercializarlo. Esta creación ha sido un proyecto personal y una exploración de mercado. No tengo planes de hacerlo público ni de ponerlo a la venta en el futuro.

Si bien este juego no verá la luz comercialmente esta experiencia podría servir de base para futuros proyectos o colaboraciones en la industria de los videojuegos, aprovechando las valiosas lecciones aprendidas durante este proceso de investigación y desarrollo.

En resumen, aunque este juego se mantendrá en el ámbito personal y no se compartirá con consumidores, cabe a destacar el aprendizaje realizado durante su etapa de desarrollo permite mejorar en futuras oportunidades

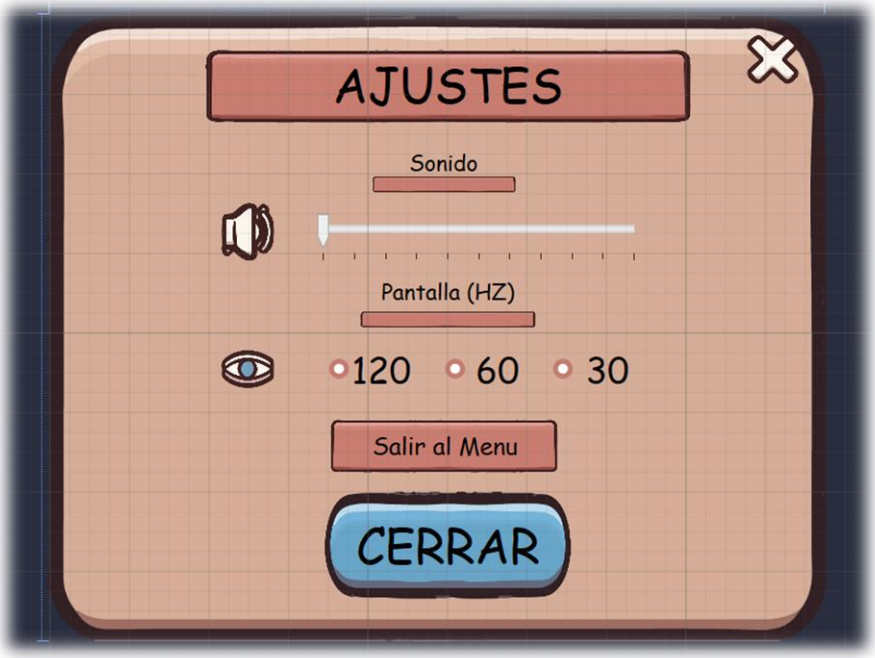
CONSTRUCCION

FUNCIONAMIENTO

El juego se compone de una ventana principal y el resto son Pages que se cargan en la ventana principal cuando sea necesario, comportamiento idéntico a los fragments en Android.

También se cuenta con 3 series de ventanas auxiliares, que son, EXIT, esta ventana abre un dialogo de salida al hacer clic en la X para cerrar la aplicación.

Ajustes, esta ventana permite al usuario modificar el nivel del sonido del juego y tiene a disposición un selector de Fotogramas Por Segundo (30-60-120)



Ayuda, la sección de ayuda guía al usuario de cómo manejar el juego y una breve explicación del objetivo general del juego

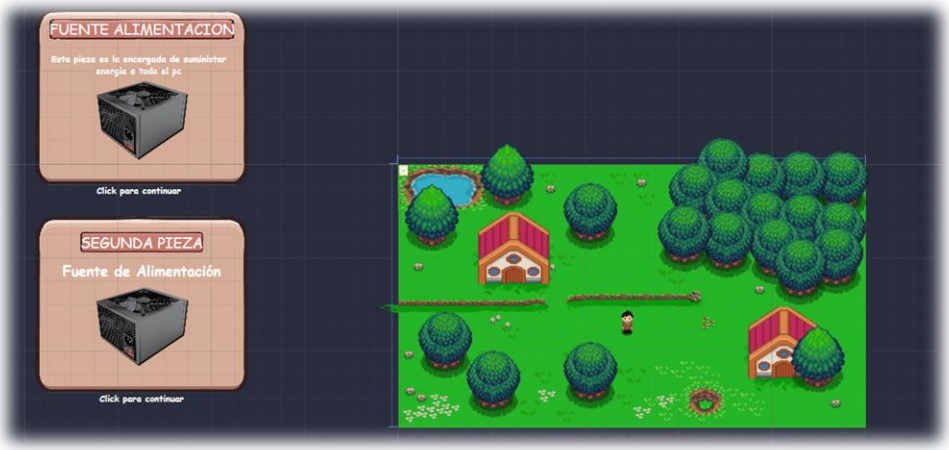
Retomando la ventana principal, la primera page que vamos a visualizar es el MENU, en la cual podemos iniciar partidas, salir, ajuste y sección de ayuda.

Las diferentes Pages nombradas anteriormente son las siguientes:

TUTORIAL: aquí se desarrolla la primea interacción con el juego por parte del usuario, guiándole un poco y explicándole ciertas mecánicas

NIVELES: cada nivel es una page diferente debido a que la lógica es independiente en cada nivel

Ejemplo Nivel 1:

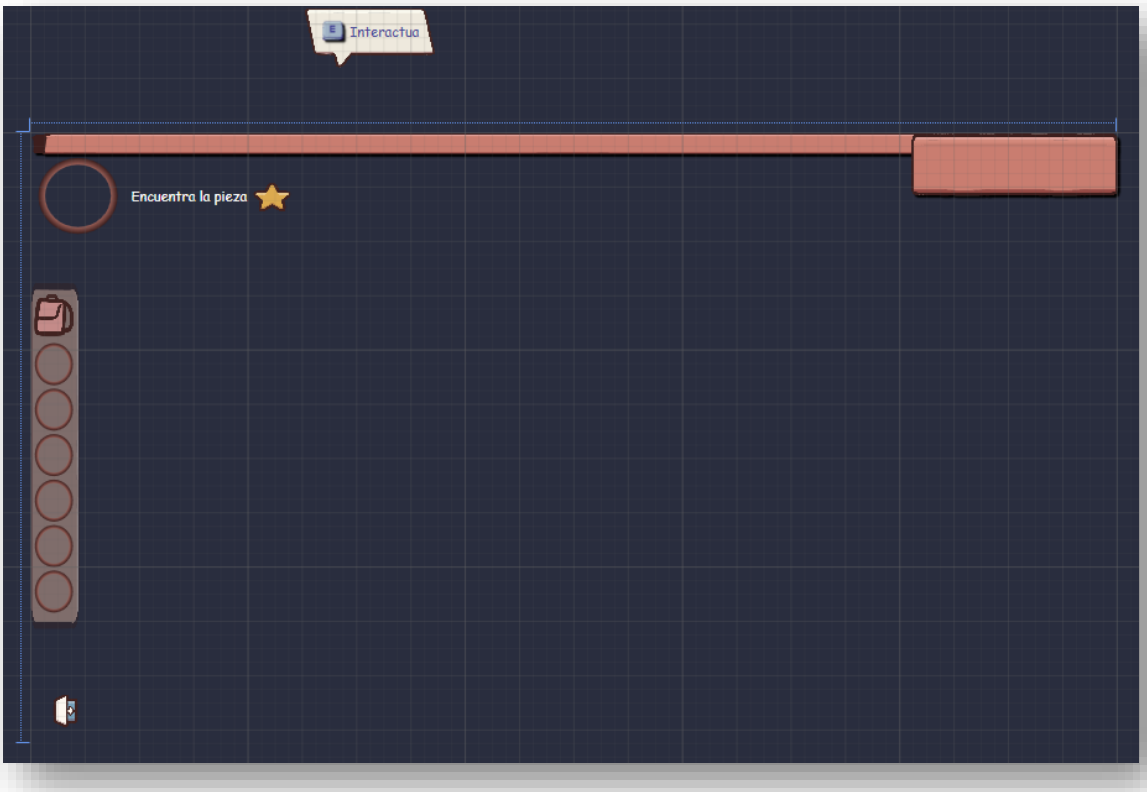


CASAS: cada casa es una page diferente debido a que la lógica es independiente en cada casa, el usuario tiene que encontrar las piezas escondidas, y cada casa alberga una pieza diferente

Casa1:

Taller, es un lugar donde accedes al recoger una pieza, un profesor aparecerá y te explicará sobre la pieza obtenida, al obtener todas las piezas podrás montar tu pc

Control de Usuario UI, actúa como un recurso global ya que puede ser llamado desde cualquier parte del código, y su principal funcionalidad es actuar como una capa en la cual se visualiza el inventario del usuario, el objetivo actual y el nivel en el cual se encuentra, en resumen, es una capa la cual representa componentes que van a ser siempre iguales y necesarios en diferentes niveles



CODIFICACION

ELEMENTOS GRAFICOS USADOS

BUTTON: Para botones en distintos lugares.

LABEL: Textos en distintos lugares o simplemente para actuar como botón

TEXTBLOCK: Es un texto el cual permite texto editable para el nombre usuario

TEXTBOX: Elemento para poder tener varia línea de texto

RECTANGLE = figura geométrica usada para marcar hit box, elementos interactivos etc.

ELLIPSE = figura para mostrar en su interior una imagen (objetivo del nivel)

IMAGE = elemento para poder representar imágenes.

PAGES: Elementos que permiten mostrar sus · elementos · en una misma ventana sin necesidad de abrir otra

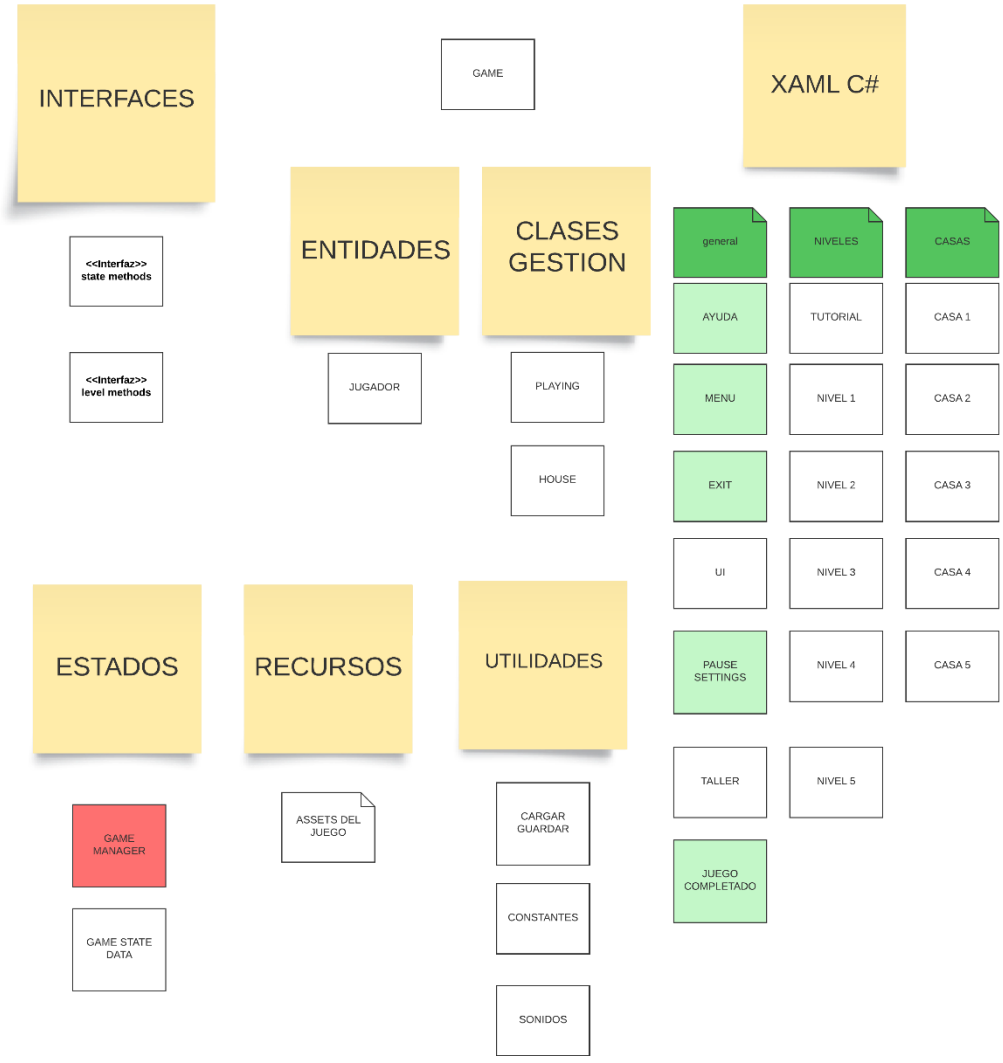
WINDOW: Elementos que se muestran en una nueva ventana ajena a la principal ventana

Las diferentes Pages y ventanas mostradas anteriormente pueden resultar extensas en cuanto a codificación por lo tanto este apartado se centra más en las mecánicas del juego, para más información acerca de todo el código se puede acceder a la siguiente dirección:

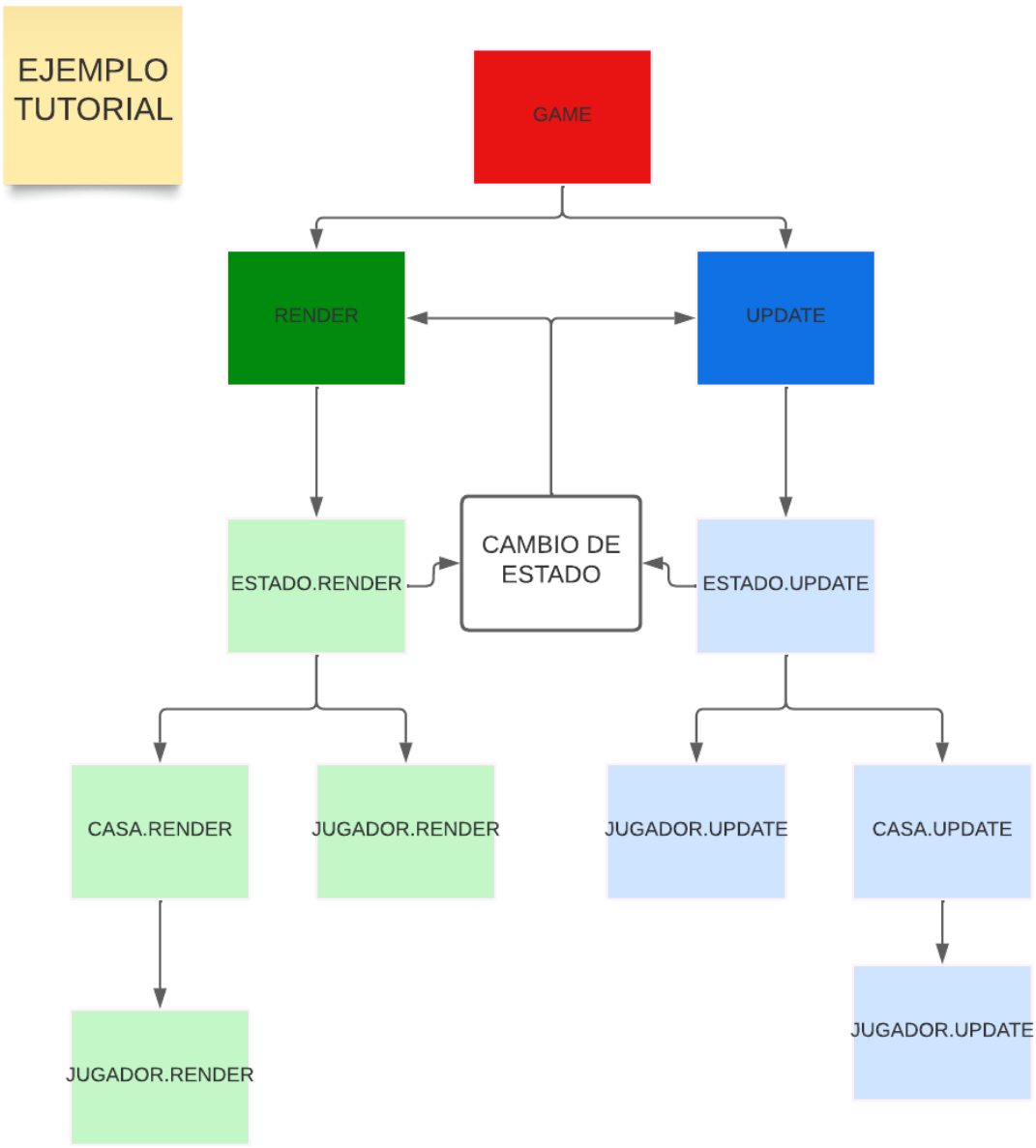
https://github.com/Rxxbertx/2DAM_DESARROLLO_INTERFACES/tree/main/1EVA/PROYECTO_1EVA_RJT

CLASES

Estructuración del código en un diagrama:



Ejemplo de ejecución del juego:



DESARROLLO DE MECANICAS JUGADOR

SISTEMA DE COLISION

Este sistema detecta el valor próximo del jugador y detecta si choca con algún elemento colider (elemento en el que chocas) , para ello el jugador en si es un canvas y está contenido en un canvas padre, que es donde se alojan todos los elementos colider, para detectarlo mejor se realiza contorno rojo a los elementos colider

Ilustración 1 INICIO DEL JUEGO (CAMBIOS SUFRIDOS)

<código>

```
private void UpdatePosition()
{
    bool avanzar = true;
    setMoving(false);

    double x = 0;
    double y = 0;


    // hitbox
    if (isFront() && Canvas.GetTop(Jugador) + Jugador.Height < 890)
    {
        y += speed * Game.DeltaTime;
        setMoving(true);
    }
    if (isBack() && Canvas.GetTop(Jugador) - Jugador.Height > 30)
    {
        y -= speed * Game.DeltaTime;
        setMoving(true);
    }
    if (isRight() && Canvas.GetLeft(Jugador) + Jugador.Width < 1600)
    {
        x += speed * Game.DeltaTime;
        setMoving(true);
    }
    if (isLeft() && Canvas.GetLeft(Jugador) > 0)
    {
        x -= speed * Game.DeltaTime;
        setMoving(true);
    }

    // Verifica colisiones antes de actualizar la posición
    Rect newHitBox = new Rect(Canvas.GetLeft(Jugador) + x, Canvas.GetTop(Jugador) + y,
Jugador.Width, Jugador.Height);

    foreach (Rectangle element in GameElementsColiders)
    {
        Rect elementRect = new Rect(Canvas.GetLeft(element), Canvas.GetTop(element), element.Width,
element.Height);

        if (newHitBox.IntersectsWith(elementRect))
        {



            setMoving(false);
            avanzar = false;
            return;
        }


    }

    if (avanzar)
    {

        double length = Math.Sqrt(x * x + y * y);
        if (length > 0)
        {
            x /= length;
            y /= length;
        }

        x *= speed * Game.DeltaTime;
        y *= speed * Game.DeltaTime;

        Canvas.SetLeft(Jugador, Canvas.GetLeft(Jugador) + x);
        Canvas.SetTop(Jugador, Canvas.GetTop(Jugador) + y);


    }



}


```

Primero, algunas inicializaciones:

avanzar es una bandera que indica si el jugador puede avanzar.

setMoving(false) indica que el jugador no se está moviendo inicialmente.

x e y son variables para almacenar la cantidad por la cual se moverá el jugador en las direcciones horizontal y vertical, respectivamente.

Luego, se verifica si el jugador puede moverse en alguna dirección sin chocar con límites del lienzo y si es así, se actualizan las variables x e y, en consecuencia.

Después, se crea un nuevo rectángulo (newHitBox) que representa la nueva posición del jugador después de intentar moverse. La función luego verifica si este nuevo rectángulo colisiona con algún elemento en GameElementsColiders, que representan objetos en el juego.

Si hay una colisión, se establece que el jugador no se está moviendo (setMoving(false)), se actualiza la bandera avanzar a false y se sale de la función.

Finalmente, si no hay colisiones, se normaliza la dirección de movimiento. La normalización convierte el vector de dirección en un vector unitario (longitud 1). Esto se hace dividiendo x e y por la longitud de ese vector, calculada con Math.Sqrt(x * x + y * y). La normalización es útil para garantizar que el jugador se mueva a una velocidad constante, independientemente de la dirección.

Después de la normalización, se multiplica la dirección por la velocidad y el tiempo delta (speed * Game.DeltaTime). Finalmente, se actualiza la posición del jugador en el lienzo horizontal y verticalmente.

En resumen, la función intenta mover al jugador en la dirección indicada por las teclas, evitando colisiones, y normalizando la dirección para mantener una velocidad constante. La parte Math.Sqrt(x * x + y * y) se utiliza para calcular la longitud del vector de dirección antes de normalizarlo.

PROBLEMAS INCIALES Y SOLUCIONES

Un problema bastante importante era la posibilidad de avanzar más rápido en diagonal que recto, entonces la solución aplicada fue la normalización del vector de movimiento

Supongamos que decides caminar 3 unidades hacia la derecha (x) y 4 unidades hacia arriba (y). Entonces:

- x * x sería 3 * 3 = 9.
- y * y sería 4 * 4 = 16.
- x * x + y * y sería 9 + 16 = 25.
- Math.Sqrt(x * x + y * y) sería la raíz cuadrada de 25, que es 5.

Así que, visualmente, si representamos esto en un gráfico:

(0,4)----- (3,4)

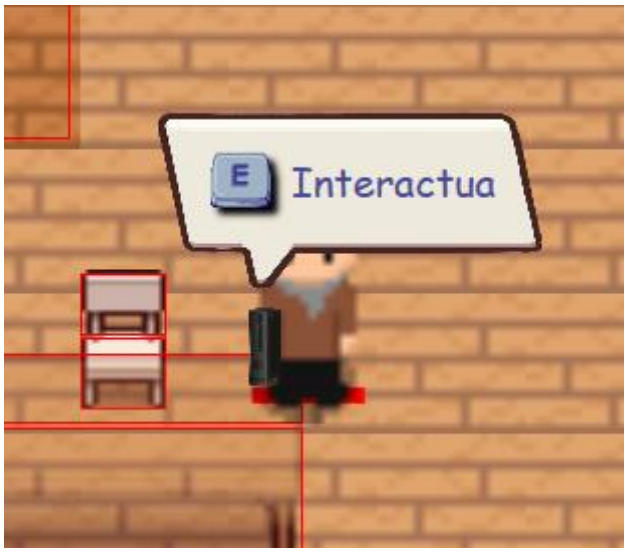
(0,0)-

Entones la normalización seria x = 3 / longitud = 5 = 0.6 en x

y 4 / longitud 5 = 0.8

SISTEMA DE INTERACCION

Este sistema detecta si atraviesa un elemento interactuable, para ellos se guardan previamente en un array de rectángulos y comprobaremos si el personaje entra dentro de ese rectángulo, si es así aparecerá un pop up de Interacción, y si el usuario pulsa la tecla E entonces se confirma esa interacción



<código>

```
private void UpdateInteractiveElements()
{
    CanvaInteractuar.Visibility = Visibility.Hidden;
    InteractiveObj = null;

    foreach (Rectangle element in GameElementsInteractive)
    {
        if (IsCollidingWith(element))
        {
            ShowInteractuable(element);


            if (isInteract())
            {

                InteractiveObj = element.Name;
                return;

            }
        }
    }


}

private bool IsCollidingWith(Rectangle element)
{

    Rect newHitBox = new Rect(Canvas.GetLeft(Jugador), Canvas.GetTop(Jugador), Jugador.Width,
Jugador.Height);
    Rect elementRect = new Rect(Canvas.GetLeft(element), Canvas.GetTop(element), element.Width,
element.Height);

    if (newHitBox.IntersectsWith(elementRect))
    {
        return true;
    }
    else
    {
        return false;
    }
}

private void ShowInteractuable(Rectangle element)
{

    CanvaInteractuar.Visibility = Visibility.Visible;
    Canvas.SetLeft(CanvaInteractuar, Canvas.GetLeft(element) + element.Width / 2 -
CanvaInteractuar.Width / 2);
    Canvas.SetTop(CanvaInteractuar, Canvas.GetTop(element) - CanvaInteractuar.Height);

}

```

UpdateInteractiveElements()

Esta función actualiza elementos interactivos en el juego. Aquí está el desglose:

1. CanvaInteractuar.Visibility = Visibility.Hidden;:
    - Oculta un elemento visual llamado CanvaInteractuar al inicio de la función.
2. InteractiveObj = null;:
    - Inicializa una variable llamada InteractiveObj a null (ningún objeto interactivo).
3. foreach (Rectangle element in GameElementsInteractive):
    - Inicia un bucle que recorre todos los elementos en la lista GameElementsInteractive, que probablemente contiene objetos interactivos en el juego.
4. if (IsCollidingWith(element)):

- Verifica si el jugador (Jugador) está colisionando con el elemento actual del bucle. Esto se verifica llamando a la función IsCollidingWith(element).

5. ShowInteractuable(element);:
   - Si hay una colisión, muestra un elemento interactivo llamado element utilizando la función ShowInteractuable(element).
6. if (isInteract()):
   - Verifica si el jugador está intentando interactuar con el elemento (posiblemente presionando una tecla específica). Esto se verifica llamando a la función isInteract().
7. InteractiveObj = element.Name;:
   - Si el jugador está interactuando, actualiza InteractiveObj con el nombre del elemento actual (element).
8. return;:
   - Sale de la función después de encontrar el primer elemento interactivo con el que el jugador colisiona.

IsCollidingWith(Rectangle element)

Esta función verifica si el jugador colisiona con un elemento rectangular. Aquí está el desglose:

1. Rect newHitBox = new Rect(Canvas.GetLeft(Jugador), Canvas.GetTop(Jugador), Jugador.Width, Jugador.Height);:
   - Crea un nuevo rectángulo (newHitBox) que representa la posición y el tamaño del jugador.
2. Rect elementRect = new Rect(Canvas.GetLeft(element), Canvas.GetTop(element), element.Width, element.Height);:
   - Crea un rectángulo (elementRect) que representa la posición y el tamaño del elemento actual en el bucle.
3. if (newHitBox.IntersectsWith(elementRect)):
   - Verifica si hay una intersección (colisión) entre el rectángulo del jugador y el rectángulo del elemento actual.
4. return true;:
   - Devuelve true si hay una colisión, indicando que el jugador está colisionando con el elemento.

ShowInteractuable(Rectangle element)

Esta función muestra un elemento interactivo

1. CanvaInteractuar.Visibility = Visibility.Visible;:
   - Hace visible el elemento CanvaInteractuar para mostrar algún tipo de indicador de interactividad.
2. Canvas.SetLeft(CanvaInteractuar, Canvas.GetLeft(element) + element.Width / 2 - CanvaInteractuar.Width / 2);:
   - Posiciona horizontalmente CanvaInteractuar en el centro del elemento element.
3. Canvas.SetTop(CanvaInteractuar, Canvas.GetTop(element) - CanvaInteractuar.Height);:
   - Posiciona verticalmente CanvaInteractuar encima del elemento element.

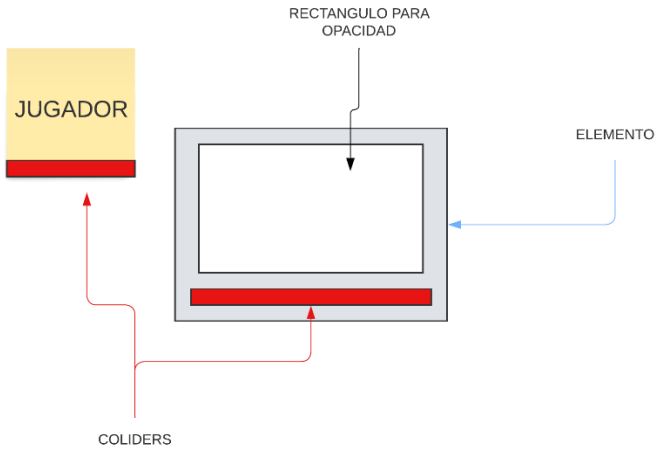
En resumen, estas funciones trabajan juntas para detectar colisiones entre el jugador y elementos interactivos, mostrar indicadores de interactividad y registrar qué objeto interactivo está siendo seleccionado por el jugador.

PROBLEMAS INCIALES Y SOLUCION

Esta mecánica se aplicó correctamente sin necesidad de replanteo

SISTEMA DE CAMBIO DE OPACIDAD DE ELEMENTOS

Este sistema permite al usuario ver su personaje aun estando detrás en un elemento, normalmente solo los elementos que pueden ser accesibles desde detrás, es decir imagina un plano 2d y quieres hacer que parezca un poco 3d, simplemente haces el elemento colider del elemento justo la parte de abajo del mismo, entones el usuario solo chocara con esa parte, por lo tanto si quieres que parezca que está detrás del elemento y no que esta encima , porque quedaría muy raro, hacemos un sistema de opacidad que es definir un rectángulo el cual si el usuario entra en él, cambiamos la opacidad del elemento respectivo, es ni más ni menos que una ilusión, *"<los desarrolladores de juegos son ilusionistas>"*



<código>

```
1 private void UpdateNormalOpacity()
2 {
3
4     List<Rectangle> listaOpacidad = GameElementsNormalOpacity[0]; //hitbox de opacidad
5     List<Rectangle> listaNormal = GameElementsNormalOpacity[1]; //rectangulo imagen
6
7     for (int i = 0; i < listaOpacidad.Count; i++)
8     {
9
10        if (IsCollidingWith(listaOpacidad[i]))
11        {
12
13            // Crea un LinearGradientBrush
14            LinearGradientBrush linearGradientBrush = new LinearGradientBrush
15            {
16                StartPoint = new Point(0.5, 0),
17                EndPoint = new Point(0.5, 0.8)
18            };
19
20
21
22            // Agrega los GradientStops al LinearGradientBrush
23            linearGradientBrush.GradientStops.Add(new GradientStop(Colors.Black, 0.827));
24            linearGradientBrush.GradientStops.Add(new GradientStop(Color.FromArgb(0x3E, 0xFF, 0xFF,
   0xFF), 0.679));
25
26            // Asigna el LinearGradientBrush como OpacityMask del Rectangle
27            listaNormal[i].OpacityMask = linearGradientBrush;
28
29
30        }
31        else
32            listaNormal[i].OpacityMask = null;
33    }
34
35 }
```

UpdateNormalOpacity()

1. `List<Rectangle> listaOpacidad = GameElementsNormalOpacity[0];:`

   - Obtiene la primera lista (`listaOpacidad`) de la lista `GameElementsNormalOpacity`. Parece ser una lista de rectángulos que actúan como hitboxes para la opacidad.
2. `List<Rectangle> listaNormal = GameElementsNormalOpacity[1];`:
   - Obtiene la segunda lista (`listaNormal`) de la lista `GameElementsNormalOpacity`. Esta lista parece contener rectángulos que representan imágenes.
3. `for (int i = 0; i < listaOpacidad.Count; i++)`:
   - Inicia un bucle que recorre todos los elementos en las listas `listaOpacidad` y `listaNormal`.
4. `if (IsCollidingWith(listaOpacidad[i]))`:
   - Verifica si el jugador (`Jugador`) está colisionando con el elemento `listaOpacidad[i]`. Esto se hace llamando a la función `IsCollidingWith(listaOpacidad[i])`.
5. `LinearGradientBrush linearGradientBrush = new LinearGradientBrush`:
   - Crea un pincel de gradiente lineal (`linearGradientBrush`). Este pincel se utiliza para crear un efecto de opacidad gradiente en el elemento.
6. `linearGradientBrush.StartPoint` y `linearGradientBrush.EndPoint`:
   - Establece el punto de inicio y el punto final del gradiente. En este caso, parece ser un gradiente vertical que comienza desde la parte superior y va hacia abajo.
7. `linearGradientBrush.GradientStops.Add(new GradientStop(Colors.Black, 0.827));` y `linearGradientBrush.GradientStops.Add(new GradientStop(Color.FromArgb(0x3E, 0xFF, 0xFF, 0xFF), 0.679));`:
   - Agrega paradas de gradiente al pincel. En este caso, crea un gradiente que va desde negro (`Colors.Black`) hasta blanco con cierta transparencia (`Color.FromArgb(0x3E, 0xFF, 0xFF, 0xFF)`) a medida que te desplazas hacia abajo.
8. `listaNormal[i].OpacityMask = linearGradientBrush;`:
   - Asigna el pincel de gradiente como una máscara de opacidad al elemento `listaNormal[i]`. Esto significa que la opacidad del elemento será controlada por el gradiente creado.
9. `else listaNormal[i].OpacityMask = null;`:
   - Si no hay colisión, se quita cualquier máscara de opacidad asignada con anterioridad al elemento `listaNormal[i]`.

En resumen, esta función ajusta la opacidad de elementos visuales en función de si el jugador está colisionando con ciertos elementos de opacidad. Utiliza un gradiente lineal para crear un efecto de opacidad gradual en los elementos afectados.

PROBLEMAS INCIALES Y SOLUCION

Esta mecánica se aplicó correctamente sin necesidad de replanteo

SISTEMA DE CARGA Y DESCARGA DE ELEMENTOS

Este sistema se basa en guardar el estado de los elementos en su última actualización, por ejemplo, si entras a una casa y vuelves a salir, te gustaría ver al personaje en la última posición antes de entrar y que el usuario vuelva a interactuar con los elementos de fuera de la casa, bien para ello se ha realizado lo siguiente:

Primero debemos realizar un diccionario para asociar cada estado con su contenido

```
private Dictionary<Enum, GameStateData> gameStates = new Dictionary<Enum, GameStateData>();
```

```
// Inicializa los estados y sus datos
gameStates[GameState.TUTORIAL] = new GameStateData();
gameStates[GameState.LVL1] = new GameStateData();
gameStates[GameState.LVL2] = new GameStateData();
gameStates[GameState.LVL3] = new GameStateData();
gameStates[GameState.LVL4] = new GameStateData();
gameStates[GameState.LVL5] = new GameStateData();
```

Clase que guarda contenido.

```
16 referencias
public class GameStateData
{
    18 referencias
    public Canvas playerHitbox { get; set; }
    18 referencias
    public List<Rectangle> CollidableElements { get; set; }
    18 referencias
    public List<Rectangle> InteractiveElements { get; set; }
    18 referencias
    public List<Rectangle>[] NormalOpacityElements { get; set; }

    12 referencias
    public GameStateData()
    {

    }
}
```

Clase donde se incorpora el guardado y carga de elementos, ej: Tutorial

Se crean las estructuras, colliders, elementos de opacidad y elementos interactivos.

```
12 referencias
public List<Rectangle> CollidableElements { get; set; }
7 referencias
public List<Rectangle>[] NormalOpacityElements { get; set; }
6 referencias
public List<Rectangle> InteractiveElements { get; set; }
```

```
CollidableElements = new List<Rectangle>();
NormalOpacityElements = new List<Rectangle>[2];
InteractiveElements = new List<Rectangle>();
```

Método de carga por si ya existen cosas.

```
public bool LoadElements()
{

    if (Game.GameManager.CurrentGameStateData == null ||
        Game.GameManager.CurrentGameStateData.playerHitbox == null ||
        Game.GameManager.CurrentGameStateData.CollidableElements == null ||
        Game.GameManager.CurrentGameStateData.InteractiveElements == null ||
        Game.GameManager.CurrentGameStateData.NormalOpacityElements == null) return false;

    hitbox = Game.GameManager.CurrentGameStateData.playerHitbox;
    CollidableElements = Game.GameManager.CurrentGameStateData.CollidableElements;
    InteractiveElements = Game.GameManager.CurrentGameStateData.InteractiveElements;
    NormalOpacityElements = Game.GameManager.CurrentGameStateData.NormalOpacityElements;

    InitPlayer();

    return true;


}
```

Método de Save para guardar

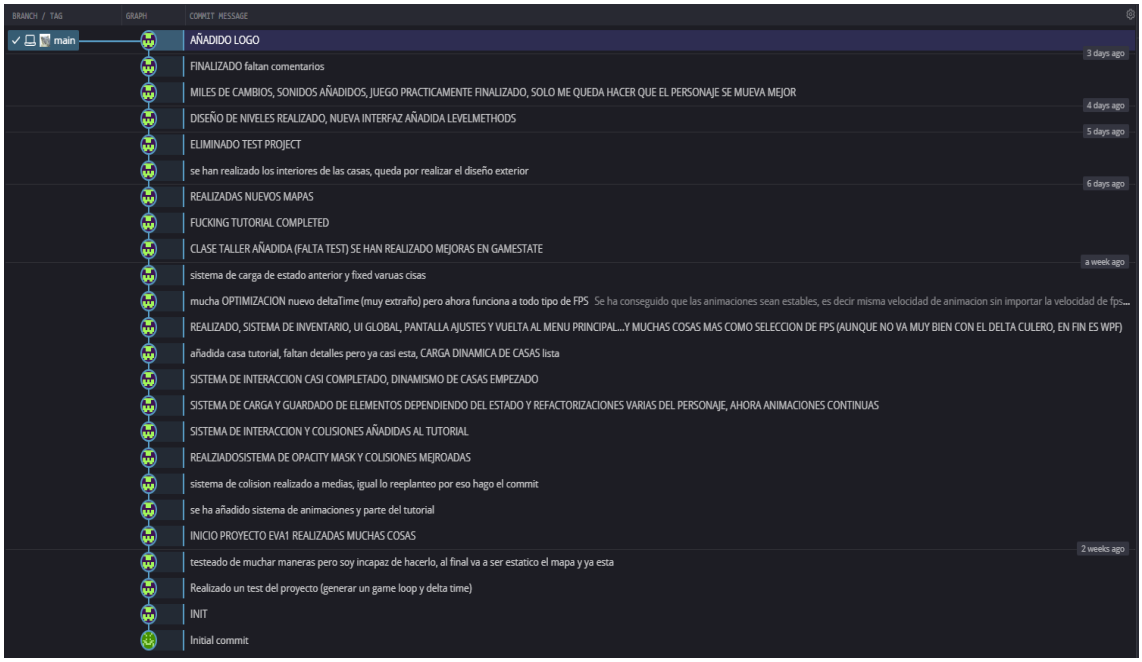
```
public void SaveElements()
{

    Game.GameManager.GameStateElements(GameState.TUTORIAL);

    if (Game.GameManager.CurrentGameStateData == null)
    {
        Game.GameManager.CurrentGameStateData = new GameStateData();
    }

    Game.GameManager.CurrentGameStateData.playerHitbox = hitbox;
    Game.GameManager.CurrentGameStateData.CollidableElements = CollidableElements;
    Game.GameManager.CurrentGameStateData.InteractiveElements = InteractiveElements;
    Game.GameManager.CurrentGameStateData.NormalOpacityElements = NormalOpacityElements;


}
```

Y método para añadir cosas nada más inicializar

```
public void AddElements()
{
    insideHouse = false;
    Focusable = true;
    this.Focus();

    Game.GameManager.GameStateElements(GameState.TUTORIAL);
    if (Game.GameManager.CurrentGameStateData != null)
    {
        if (LoadElements())
        {
            return;
        }

    }


    //image and opacity
    NormalOpacityElements[0] = new List<Rectangle>()
    {
        arbol1Opacidad
        , arbol2Opacidad
        , arbol3Opacidad
        , arbol4Opacidad
        , arbol5Opacidad
        , casa1Opacidad

    };
    NormalOpacityElements[1] = new List<Rectangle>
    {
        arbol1N
        , arbol2N
        , arbol3N
        , arbol4N
        , arbol5N
        , casa1N

    };


    //coliders

    CollidableElements.Add(arbol1HitBox);
    CollidableElements.Add(arbol2HitBox);
    CollidableElements.Add(arbol3HitBox);
    CollidableElements.Add(arbol4HitBox);
    CollidableElements.Add(arbol5HitBox);
    CollidableElements.Add(vallasHitbox);
    CollidableElements.Add(casa1Hitbox);


    //interactuables

    InteractiveElements.Add(puertaCasa1);

    InitPlayer();


}
```

PRUEBAS Y AVANCES DEL CODIGO DIA A DIA:

Para ello el proyecto se ha realizado en GitHub, así de esta manera se pueden visualizar los diferentes estados del juego en cada momento, el historial de modificaciones es el siguiente:



También se ha registrado el tiempo dedicado al proyecto

https://wakatime.com/projects/2DAM_DESARROLLO_INTERFACES?start=2023-11-01&end=2023-11-09

https://wakatime.com/projects/PROYECTO_1EVA_RJT?start=2023-11-01&end=2023-11-09

TOTAL, DE 76HRS 39MINUTOS

Además, se ha realizado un TO DO de las tareas a realizar

https://app.milanote.com/1R0luc17g9kn9E/to-do

CONCLUSIONES

A lo largo de este proyecto se ha enfrentado a varios problemas de implementación, ya que plantear e implementar nuevos mecanismos / sistemas, resulta muy laborioso y complejo.

Algunos problemas enfrentados han sido:

- EL sistema de carga y descarga de elementos resulto complejo y difícil de crear ya que, por mucho tener la idea, su implementación resultó una modificación en gran medida de otra lógica de la aplicación
- Sistema de animaciones, al principio era suficiente, pero a medida que se probaba otra configuración de FPS, o en otro equipo, la velocidad de animación era inestable.
- Colisión de elementos fue todo un desafío hasta llegar a entender el funcionamiento completo de un canvas
- Delta time, su implementación es sencilla, pero si los FPS cambian en tiempo de ejecución se vuelve inestable y errónea, por lo tanto, se ha tenido que rehacer de una manera más compleja
- La búsqueda de assets y sonidos para el juego resulto una gran pérdida de tiempo
- Se realizo varias optimizaciones y refactorizaciones a lo largo del proyecto, siendo una gran perdida de tiempo, pero una mejor modularidad, funcionamiento y comprensión
- PUNTO CLAVE se empieza sin saber y se termina sabiendo.

Cabe destacar que el juego en sus principios tenía otra idea e iba a tomar un rubo algo diferente, pero con los mismos objetivos, la primera idea resultaba un jugador que iba a estar en un mapa enorme por el cual iba a buscar piezas, esa idea no llego a producción debido a su perdida masiva de tiempo para llegar a hacer algo funcional, por lo tanto, se descartó para optar por otro sistema, ya que el contenido tenía que moverse en base al personaje

El juego ha cambiado bastante en diseño, las primeras versiones resultaron ser básicas y sin nada llamativo, por lo tanto, en su versión final se ha logrado un producto mucho más elaborado.

ANTES:

DESPUES

MEJORAS

- Assets con una calidad muy superior.
- Realización de cámara de seguimiento del personaje y poder hacer los niveles mucho más enormes.
- Adaptación a diferentes pantallas y escala de resoluciones

ENLACE PARA EXTENDER LA INFORMACION DE LA MEMORIA

Milanote: Realizado un Game Data Document Sencillo

https://app.milanote.com/1R0H8P17g8zEbu?p=bgJwR7WHRwM