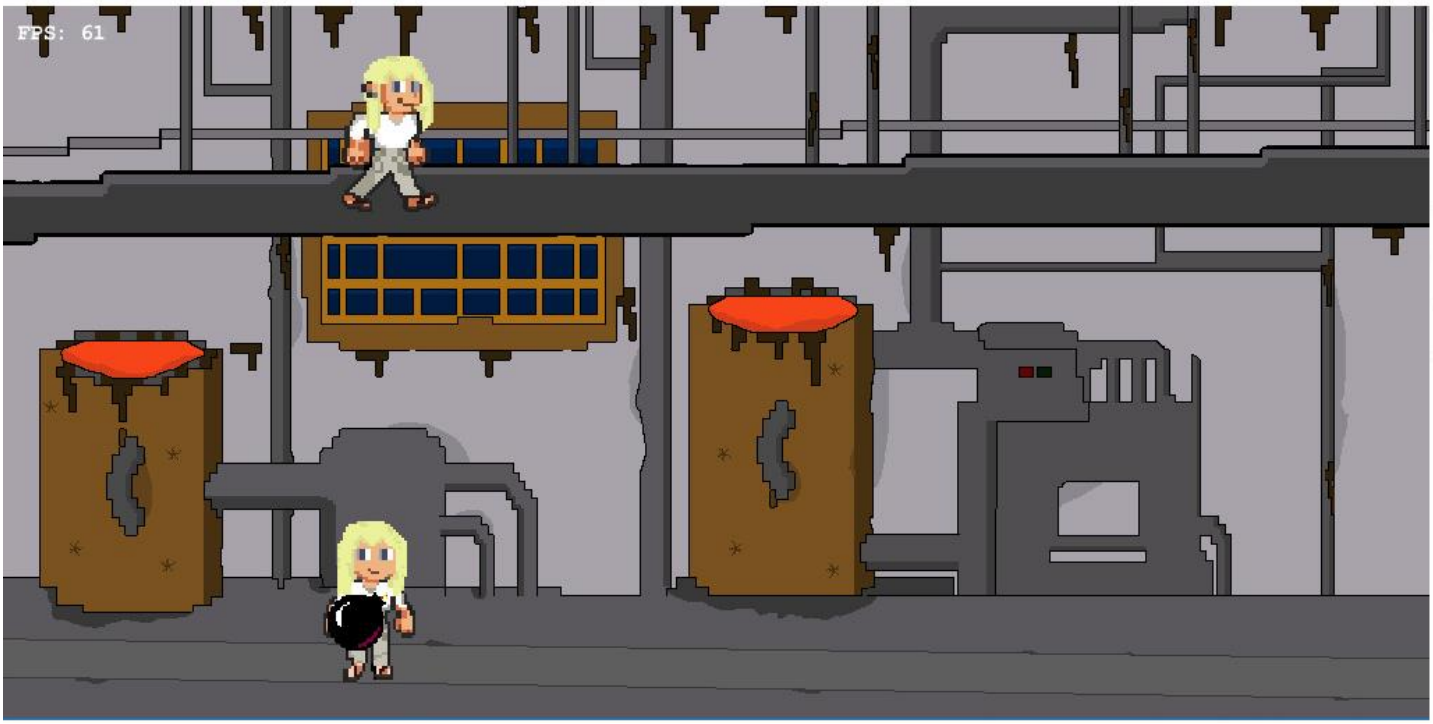
Adventure party



GRUPO F:

Álvaro Pérez Giménez
Ana Rodríguez Zarzuela
Álvaro Iglesias Salinero
Sergio Sánchez Campo
Yosua Martínez Sánchez
Pablo Gonzalez Jalvo

Índice

Escenarios……pág. 3

Minijuegos……pág. 8

Sprites……pág. 11

Análisis y perspectiva técnica……pág.14

GitHub y Estructura del proyecto……pág. 27

Escenarios

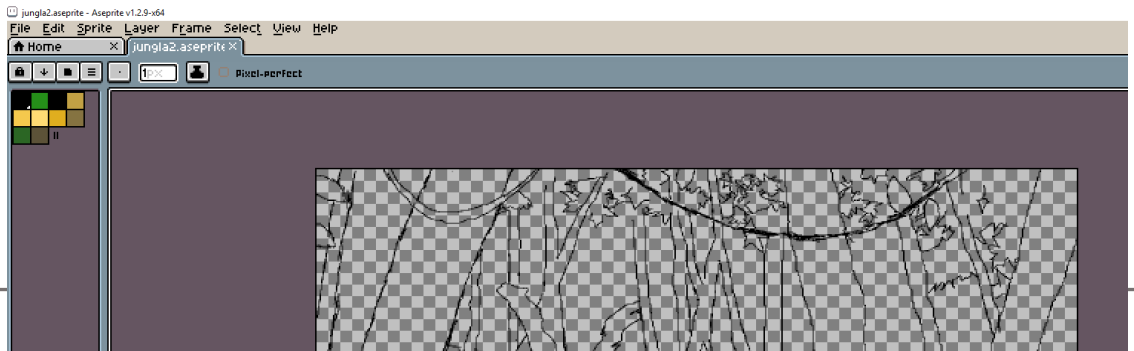
Para el proceso de diseño y dibujado de escenarios en primer lugar decidimos el bioma del escenario que empezaríamos a dibujar, los primeros paisajes que recorrería el personaje estarían basados en la selva, así que primero comenzamos haciendo algunos bocetos de este bioma con diferentes estilos artísticos para observar cual se ajustaba mejor a nuestra idea de videojuego.

Probamos a realizar escenarios de diferentes estilos artísticos.

Finalmente nos decidimos quedar con el estilo pixelado (imagen de la derecha), ya que se ajustaba mejor con nuestra idea de diseño pues nos permitía crear escenarios y personajes vistosos sin necesidad de ser grandes artistas y en una cantidad de tiempo viable.

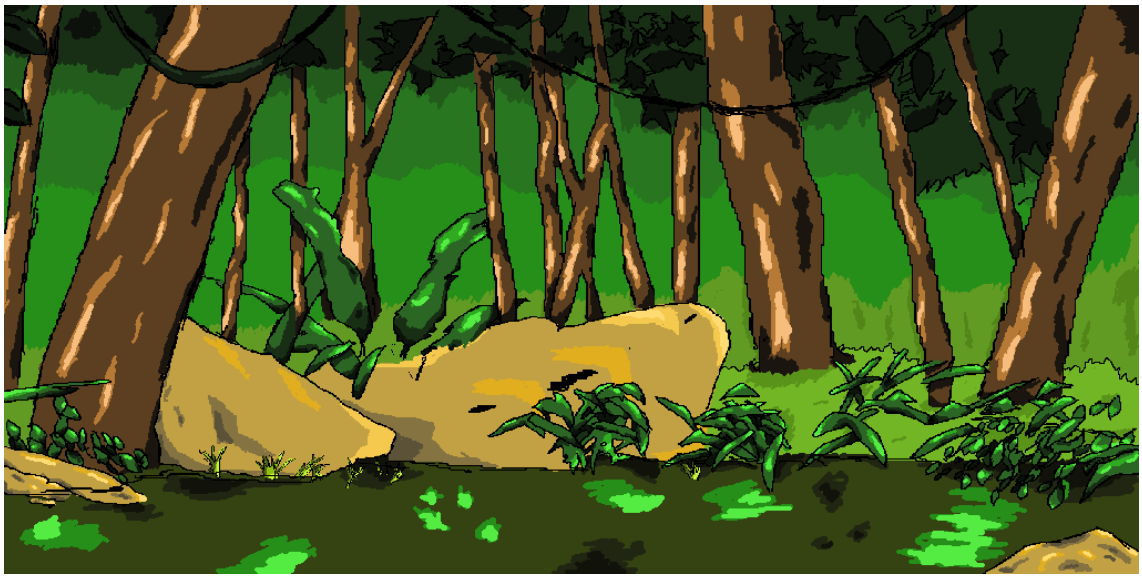
Para realizar los escenarios empleamos el programa aseprite, ya que nos permitía realizar escenarios de una manera muy sencilla, con pocas herramientas por lo que era fácil de aprender, pero nos permitía realizar todo lo que queríamos.

Para simular el estilo pixel empleamos la herramienta lápiz con un tamaño de 2 píxeles. Dividimos el proceso en tres fases, en primer lugar dibujábamos los contornos de los elementos con color negro, a continuación le aportábamos su color básico y finalmente las sombras y relieves. Cada elemento era tratado en una capa diferente para simplificar el proceso y para que en el caso de eliminar o modificar un elemento en concreto fuese mucho más sencillo. El tamaño empleado para los escenarios fue de 1024 X 512.

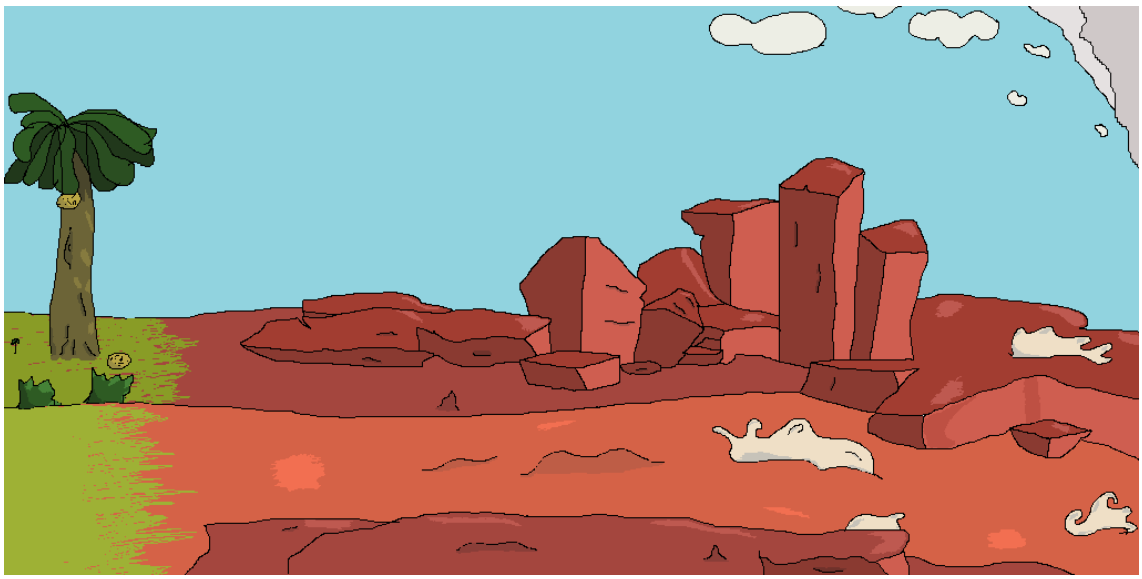


Tras haber acordado el estilo artístico que emplearíamos decidimos fijar el número de mapas diferentes que contendría cada área. Iban a existir 3 biomas diferentes y teniendo en cuenta el tiempo del que contábamos fijamos hacer 3 mapas para cada bioma. Además, nos dividimos el trabajo de manera que parte del grupo se encargara de dibujar el escenario y otras personas se encargaban de darle color.

A la hora de colorear el segundo escenario de jungla se comenzó aportando color al fondo, empleando tonalidades verdosas para mostrar que era un escenario predominado por plantas. Se le aportó diferentes tonalidades de verde según la distancia para dar mayor sensación de realismo y de profundidad. A continuación, pasamos a colorear elementos concretos del ambiente, comenzamos por los elementos inorgánicos, que serían solamente tres piedras para mostrar que es un escenario con gran fertilidad y en el que prácticamente todo está vivo, además a estas piedras se les dio una tonalidad marrón anaranjada muy intensa y colorida para dar sensación de vida y energía. Tras acabar esto pasamos a los elementos orgánicos, que estarían compuestos tanto por plantas herbáceas como leñosas. Las leñosas estarían compuestas por árboles de gran altura y con copas muy frondosas, ya que es un ambiente con una gran cantidad de árboles y con una alta competitividad para captar la luz. La herbáceas serían principalmente helechos con grandes hojas que les permitieran captar la escasa luz que llega a las partes bajas de la selva. Tras haber acabado de colorear todos los elementos del entorno pasamos a aportarles sombras y luminosidad para aportarles realismo. Para las sombras de cada elemento empleamos el color base mezclado con un poco de negro y se lo aplicamos a los lugares donde la luz incidiría menos, y al contrario para las zonas con mayor luminosidad.



El segundo bioma que comenzamos a diseñar fue el volcán y para evitar un cambio muy brusco entre ambos dibujamos un escenario que contuviese parte del bioma de la selva para mostrar gráficamente esta transición entre biomas y no pareciese que el videojuego estuviese compuesto simplemente por un conjunto de biomas distintos sin ninguna relación entre ellos.

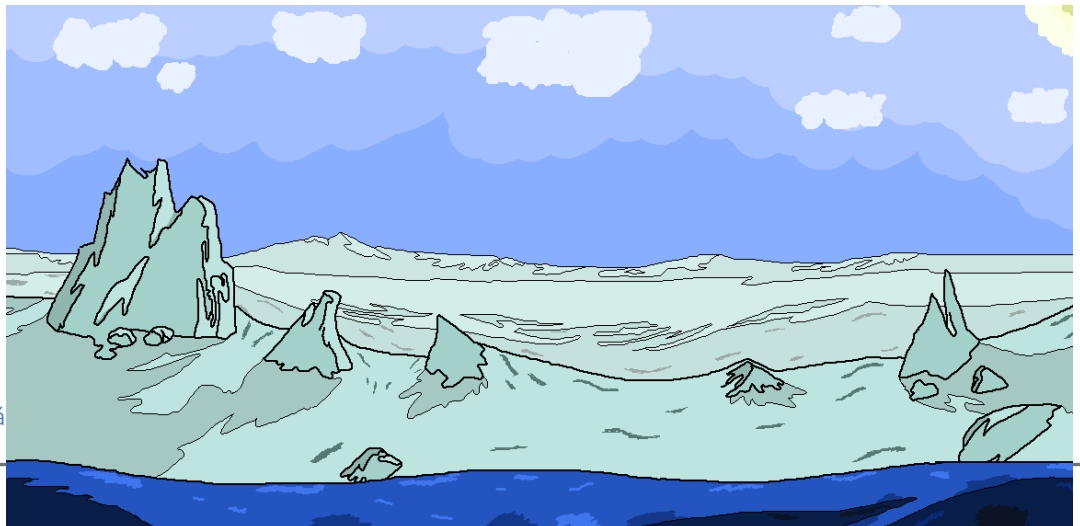


Para el área volcánica dibujamos escenarios predominantemente rocosos, con gran cantidad de piedras e incluso nubes de humo para simular la extrema escasez de humedad y vegetación

de la zona. Para los colores empleamos una paleta de colores cálidos, marrón muy rojizo en esta ocasión para las rocas para simular las altas temperaturas de la zona, color blanco para las nubes y las nubes de polvo, azul para el cielo y verde y marrón para las escasas plantas que se asoman por la parte izquierda del mapa y que muestra como las plantas van desapareciendo progresivamente según se avance en el escenario volcánico y que no vuelven a aparecer en este bioma al ser una zona inhabitable. Finalmente, se le aportaron las sombras y luces de la misma manera que en el anterior mapa.

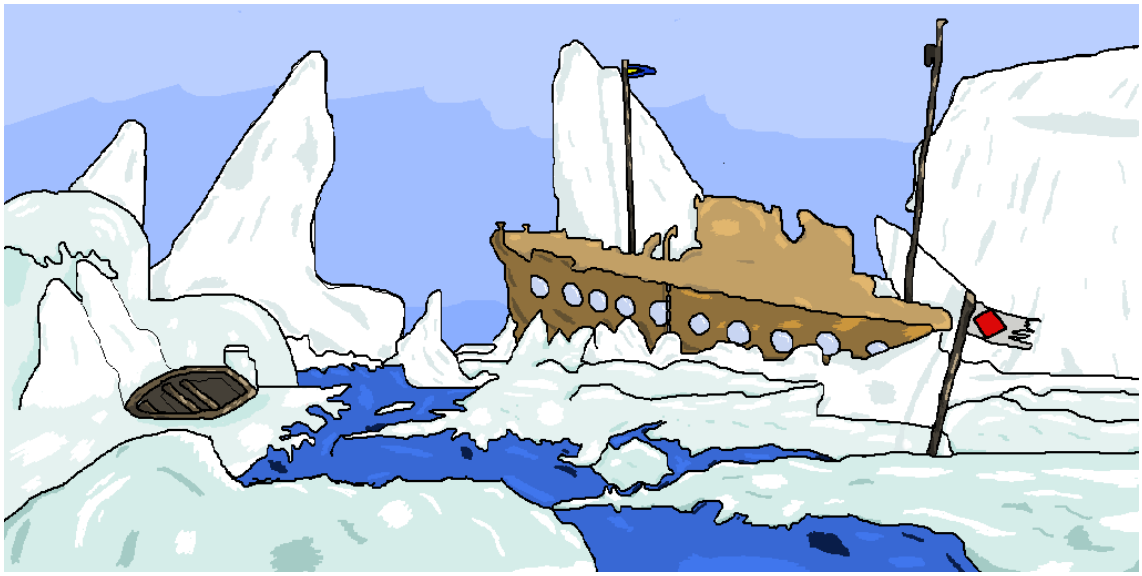
El último bioma que escogimos para nuestro videojuego fue el de nieve. Para esta clase de escenarios buscamos representar también terrenos prácticamente inhóspitos, pero en contraposición con el anterior bioma en este caso se debería a las ínfimas temperaturas. Para simular esto empleamos una paleta de colores fríos, empleando para todo el escenario solamente tonos de azules y blancos.

Para colorear el agua empleamos tonos de azules oscuros para mostrar profundidad, que son aguas peligrosas y que no es un territorio agradable. Para el terreno dibujamos grandes planicies con algunas piedras, y toda su superficie recubierta de nieve mostrando que es un territorio muy frío y prácticamente virgen en el que es muy complicado sobrevivir. El hielo fue coloreado empleado como base el color blanco y aportándole una tonalidad un poco más oscura y ligeramente azulada al terreno más cercano al espectador. La primera capa de nieve fue donde se aportó mayor definición y donde se dibujaron las piedras y sombras, pues es la más cercana al personaje y por lo tanto la que visualizaría con mayor claridad en esa situación; al contrario, las capas más lejanas de nieve están compuestas por un color lineal sin apenas detalles pues al encontrarse a mayor distancia del personaje este simplemente apreciaría un terreno constante.



Para simular las sombras se colorearon con mayor oscuridad las zonas que no se encontrarían directamente enfocadas por el sol, y para evitar estatismo y rigidez también se dibujaron ondulaciones en la nieve y en el mar para simular que no es un terreno totalmente plano.

Finalmente, para aportar mayor sensación de inhospicidad en el segundo escenario además se dibujaron dos barcos encallados, uno de ellos un pequeño barco pesquero y el otro un gran buque, que muestra que es un lugar peligroso en el que es fácil quedarse atrapado y no conseguir escapar con vida. Al buque se le colorearon dos banderas, la rectangular blanca con un rombo rojo que significa “Foxtrot”: barco averiado, por favor establezca comunicación. Y la triangular azul y amarilla que muestra el país de procedencia.



Tras haber acabado los escenarios, pasamos a realizar parte del HUB, que permitiría al jugador tener información relevante en todo momento, pero que además resultase agradable a la vista y que la información fuese fácil de leer e interpretar.

Para ello empleamos formas geométricas simples, con colores vistoso y que la información estuviese comprimida y fuese muy sencilla de comprender. Como por ejemplo en los turnos que empleamos simplemente dos palabras escritas en grande que permitiría al jugador rápidamente saber en qué turno se encuentra, y también se le aporto dos colores fácilmente diferenciables (rojo y azul) a cada jugador para que a los pocos minutos el jugador pudiese reconocer incluso más fácilmente el turno simplemente observando el color del interior del rectángulo.

Minijuegos

Un apartado importante en este videojuego son los minijuegos que lo componen. En un inicio la intención era realizar minijuegos temáticos en función de la zona en la que te situases. Aunque esta idea se ha llevado a cabo, en algunos casos, se ha intentado realizar videojuegos mas genéricos para poder reaprovechar el trabajo realizado. Un ejemplo de videojuego genérico lo tenemos en el minijuego 1

La idea del minijuego es pulsar la tecla correspondiente a la flecha que llega al panel de arriba, si se pulsa justo en el momento en que la flecha toca el panel de flechas se puntuará positivamente, de lo contrario se restarán puntos. El minijuego está basado en un videojuego de recreativa llamado PUMP IT UP NX el cual en vez de tener un teclado para pulsar las flechas debes pulsar con los pies de este modo para conseguir la máxima puntuación tenías que realizar la coreografía correctamente

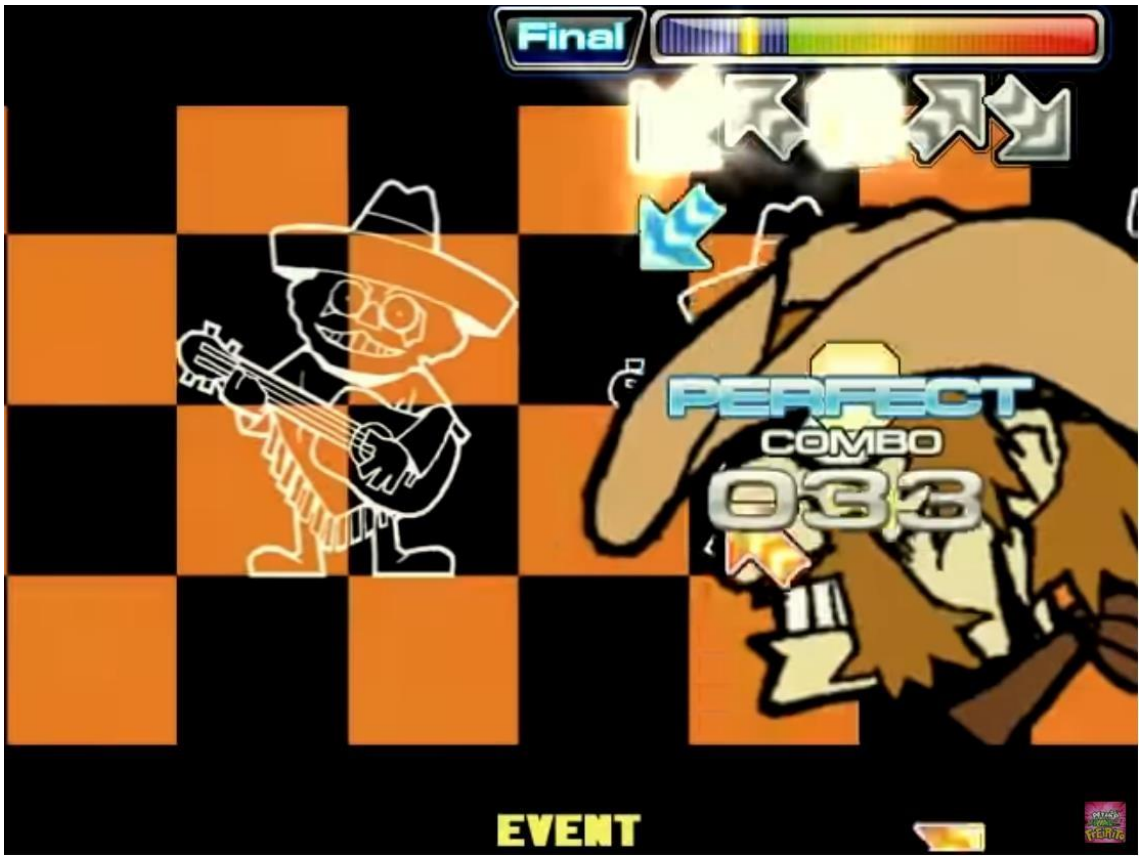
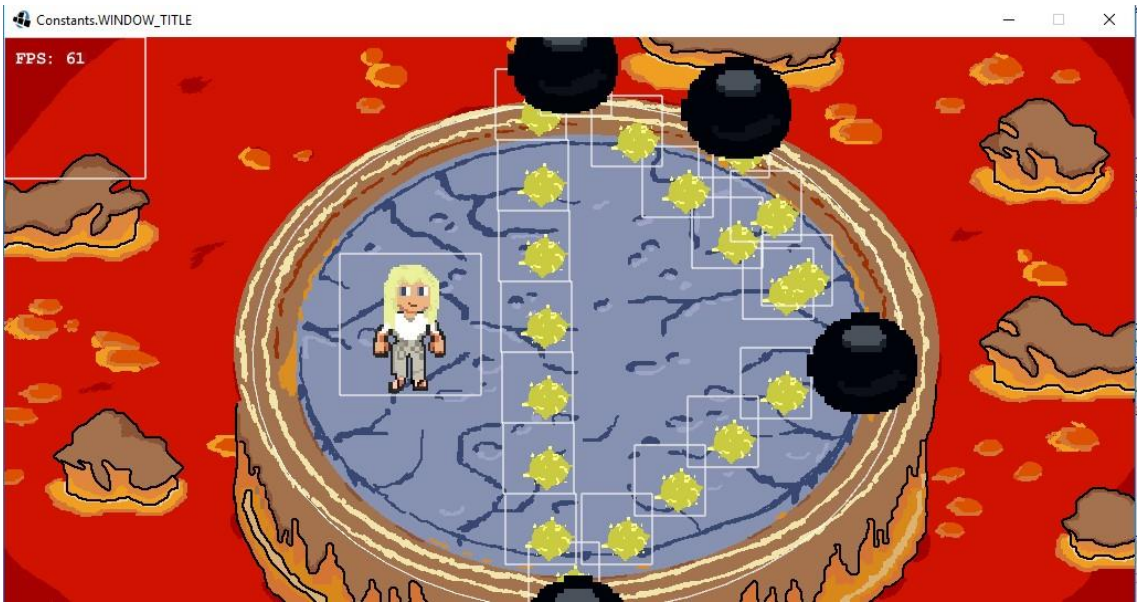


Imagen extraída del videojuego oficial

En el otro lado un minijuego más tematizado en concreto con la zona ígnea tenemos este minijuego



Este minijuego está basado en uno similar del videojuego Mario party nds, la idea del videojuego es esquivar la zona eléctrica que generan las bolas negras cuantas mas rondas dures sin tocar esas chispas más puntuación obtendrás.

Sprites

Hemos usado una aplicación llamada “aserpite” para realizar los “sprites”.

Teniendo en cuenta el tipo de minijuego y el tipo de escenario necesitábamos diferentes “sprites” de personajes y objetos, en unos cogíamos imágenes de internet y las modificábamos con esta aplicación para adaptarlas al juego y en otras ocasiones los “sprites” se dibujaban a mano para conseguir los personajes que queríamos.

Los escenarios y los fondos de pantalla del juego también los hicimos con esta aplicación y los dibujamos a mano entre todos los miembros del grupo, utilizando diferentes capas para lograr un resultado de profundidad.

También dibujamos en esta aplicación algunos elementos para los diferentes minijuegos o para diferentes partes del juego, como casillas, teclas para pulsar o el propio menú del videojuego.

Por último, para que todos los miembros del grupo pudiéramos dibujar sin que pareciera que cada dibujo tenía un estilo diferente, creamos una paleta de colores determinados y pusimos una norma de trazos; para que cuando diferentes miembros del grupo pintaran diferentes zonas del juego, fueran todas del mismo estilo de dibujo.

Análisis y perspectiva técnica

Una vez explicada la primera división entre minijuegos, procederemos a explicar aquellos factores técnicos más relevantes de los diferentes minijuegos que integran nuestro juego.

Para empezar, cabe destacar la organización de éstos. Cada minijuego tiene asignado una id y mediante una máquina de estados se van invocando a estos según se va avanzando por el mapa(“tablero”)

```
private final int menuStateId = 0;
private final int boardStateId = 1; // TODO
private final int minigameTestStateId = 2;
private final int minigame1Id= 3;
private final int minigame2Id = 4;
```

En este apartado no entraremos a explicar como va decidiendo la máquina de estados que minijuego corresponde aquí nos centraremos en los factores técnicos de los minijuegos en sí.

Minijuego 0

El minijuego 0 ha sido la base con la que hemos diseñado el resto de videojuegos.

Este consiste de un personaje que se mueve solo de derecha a izquierda que tiene que ir recogiendo objetos que caen desde arriba, gana puntos cuantos mas objetos recoge.

```
public void update(GameContainer gc, StateBasedGame sbg, int delta) throws SlickException {
    player.updateX(x += keyboard.getXMovementPl1() * delta / 200f); // Set values as constants

    if (elapsedTime++ > spawnSpeed) {
        arrayBananas.add(createBanana());
        elapsedTime = 0;
    }

    for (GameObject go : arrayBananas) {
        go.updateYByIncrease(speedDificulty);
        if (player.getCollisionBox().intersects(go.getCollisionBox())) {
            //go.setDeleted(true);
        }
    }

}

/*
 * Create Bananas
 */
private GameObject createBanana() {
    return new GameObject(bananaImage, ThreadLocalRandom.current().nextInt(0, 1024), 0, 0.1f); // Set values as constants
}
```

La programación es sencilla un bloque temporizador controlado por la variable elapsedTime que cuando llega al valor de spawnSpeed se creara otro objeto que al chocar contra el protagonista provocará que este gane puntos.

Minijuego 1

```
    /*
     * Update
     */
    @Override
    public void update(GameContainer gc, StateBasedGame sbg, int delta) throws SlickException {


        if (elapsedTime++ > spawnSpeed) {
            arrayArrow.add(createArrow());
            elapsedTime = 0;
        }

        for (int i=0;i<arrayArrow.size();i++) {
            arrayArrow.get(i).updateYByIncrease(-speedDificulty);
        }


        if(arrayArrow.size()>1)
        {
            scoreColission(arrayArrow,gc);

            if(arrayArrow.get(0).getY() <= 0)
            {
                arrayArrow.remove(0);
            }
        }
}
```

El minijuego se compone de un temporizador que es el primer bloque que va creando una flecha que se añadirá a la lista de flechas activas las cuales serán eliminadas si llegan al final de la pantalla o si pulsamos alguna tecla, en caso de que la tecla sea la correcta en el momento adecuado aportará puntos en caso contrario,restará. Esto último se puede ver en la función scorecolission

```
    private void scoreColission(ArrayList<GameObject> arrayarrow,GameContainer gc){

        GameObject firstelement=arrayarrow.get(0);
        if(keyboard.getPressedpl1(gc)!= ""){
            if (keyboard.lastpressedp1==firstelement.getdirection()){
                if(arrayarrow.get(0).getCollisionBox().intersects(keys.getCollisionBox())){
                    puntuacion+=20;
                    }
                else{
                    puntuacion-=20;
                }
                arrayarrow.remove(0);
             }
            else
            {
                puntuacion-=20;
                arrayarrow.remove(0);
            }

    }
}
```

Cada flecha tiene asignada un parámetro que será la dirección de ésta que se le dará al ser creada y que se comprobará a la hora de evaluar si la tecla pulsada corresponde con la flecha que toca.

```
     * Create Bananas
     */
    private GameObject createArrow() {
        GameObject arr1;

        int arrowcolor=ThreadLocalRandom.current().nextInt(0, 4);
        spawnSpeed=ThreadLocalRandom.current().nextInt(20, 60);
        int posx=0;
        String direction="";

        Animation arrowaux = null;
        switch(arrowcolor) {

        case 0:
            arrowaux=arrowimagedown;
            direction="Down";
            posx=64;
            break;
        case 1:
            arrowaux=arrowimageup;
            direction="Up";
            posx=128;
            break;
        case 2:
            arrowaux=arrowimageright;
            direction="Right";
            posx=192;
            break;
        case 3:
            arrowaux=arrowimageleft;
            direction="Left";
            posx=256;
            break;
        }

        arr1 = new GameObject(arrowaux, posx, 512, 1.0f); // Set values as constants
        arr1.setdirection(direction);
        return arr1;

    }
```

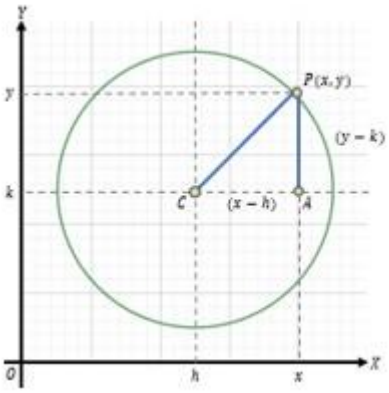
Minijuego 2

A nivel técnico el minijuego 2 es el que más interés tiene, éste tiene 2 factores que destacan sobre el resto. El primero es el como conseguir un movimiento circular de las esferas que provocan los rayos y el segundo es el como conseguir trazar los rayos que unen cada esfera.

En un primer momento el movimiento circular lo intenté realizar mediante la ecuación explícita de la circunferencia centrada en el origen

$$x^2 + y^2 = r^2$$

Dado que la parte visible de nuestro videojuego no tiene ni valores de *x* negativos ni de y negativos, la ecuación debe ser adaptada para desplazar nuestra circunferencia hacia valores tanto de *x* como de *y* siempre positivos



Con Centro fuera Del Origen

$$(x-h)^2+(y-k)^2=r^2$$

Un primer enfoque podría ser intentar despejar de esta ecuación y de modo que dando valores *a x* obtengamos valores para *y*, de esta forma obtendríamos la coordenada *x* de nuestra esfera y nuestra coordenada *y* así se movería de forma circular.

Con esta solución tenemos el problema de que al despejar y, obtenemos una expresión complicada con varias soluciones lo cual dificulta su implementación.

Una forma de evitar esto es parametrizando la ecuación de la circunferencia si en vez de expresar la ecuación de forma explícita la expresamos de forma paramétrica podremos solucionar este problema.

Una parametrización posible de la ecuación de la circunferencia es la siguiente:

$$\mathrm{p}\begin{cases} x = c_1 + r\cdot\cos\theta \\ y = c_2 + r\cdot \ sen\,\theta \end{cases}$$

siendo $C=(c_1,c_2)$ el centro y θ el ángulo del punto

Demostración:

Si volvemos al caso de la circunferencia cuyo centro está en el (0,0) sustituyendo la parametrización en la ecuación explícita obtendríamos:

$$(rcos\theta)^2+(rs\mathrm{e}n)^2=r^2$$

$$(cos^2\theta+s\mathrm{e}n^2\theta)r^2=r^2$$

Ya que

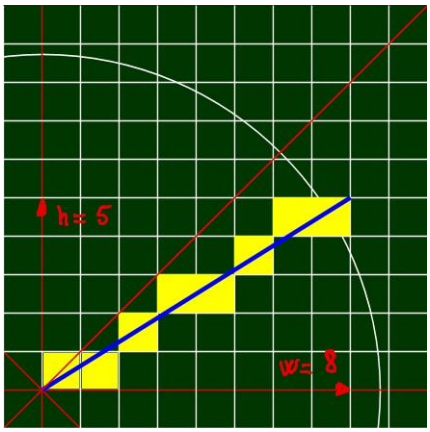
$$cos^2\theta + sen^2\theta = 1$$

Para cualquier valor de theta se cumple y por lo tanto la parametrización es válida.

```
private void updateMove() {
    double posX, posY;
    double theta=0;
    int r = 256;

    for(int i=0;i<numberBalls;i++) {

        //Evito colisiones entre bolas
        theta=ThreadLocalRandom.current().nextDouble(i*2*Math.PI/numberBalls,(i+1)*2*Math.PI/numberBalls);

        posX = r * Math.cos(theta) + 460;
        posY = r * Math.sin(theta) + 220;


        ballarray[i].setX((int)posX);
        ballarray[i].setY((int)posY);
    }
```

En el código como se puede observar vemos que para cada esfera se genera un numero entre los rangos de (0,pi/2)(pi/2,pi)(2pi/3,2pi) esto se hace así para evitar solapamientos entre esferas ya que si a todas las esferas le damos el mismo rango que sería de 0 a 2pi se moverían se forma circular pero podría darse el caso de que dos queden muy pegadas.

El trazado de rectas lo implementé utilizando el algoritmo de bresenham,



El cual dados dos puntos en función de la pendiente va iterando en la coordenada *x* o en la *y* y va realizando una aproximación de una recta en un entorno en el que los puntos donde la recta solo pueden tomar valores enteros.

Dados que los puntos de nuestra recta deben tener una distancia de 64 pixeles para evitar solapamientos en nuestro algoritmo de bresenham según se va iterando compruebo que el punto anterior diste como mínimo 64 pixeles del siguiente.

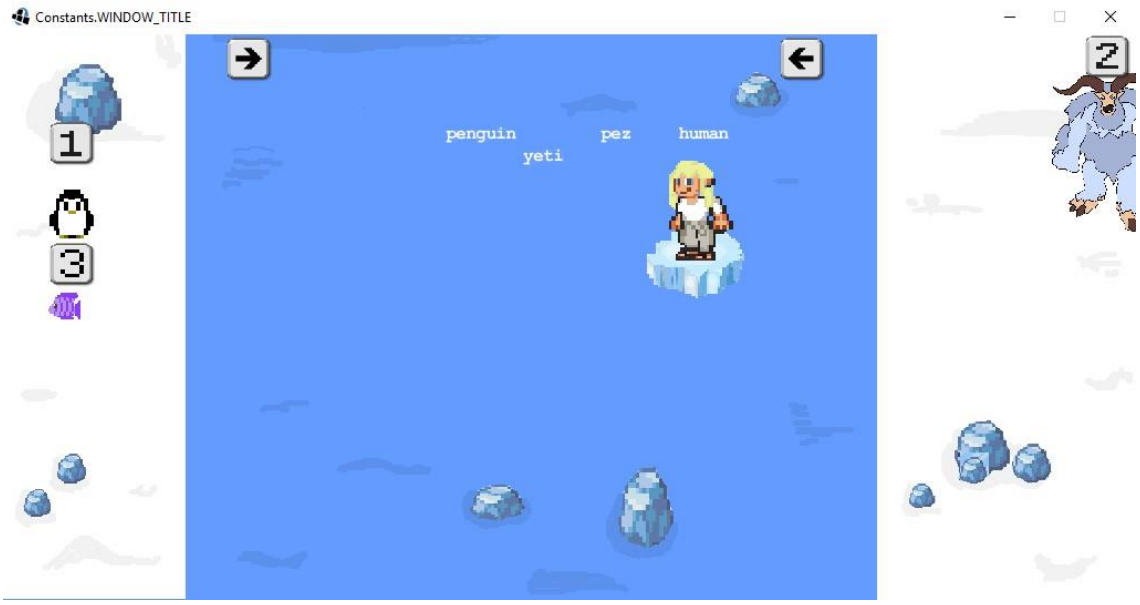
```
private void coordenadas_recta_disparo(int x0, int y0 , int x1 , int y1)
{
        int x2,y2;
        int stepX, stepY, p;
        int dx = x1 - x0;
        int dy = y1 - y0;

        if (dx < 0) {
            dx *= -1;
            stepX = -1;
        } else {
            stepX = 1;
            dx = x1 - x0;
        }
        if (dy < 0) {
            dy *= -1;
            stepY = -1;
        } else {
            stepY = 1;
        }
        x2=x0;
        y2=y0;
        if (dx > dy) {
            p = 2 * dy - dx;
            while (x2 != x1) {
                x2 += stepX;
                if (p < 0) {
                    p += 2 * dy;
                } else {
                    p +=  2 * (dy - dx);
                    y2 += stepY;
                }
                if( ( Math.sqrt( Math.pow(x2-x0 ,2) + Math.pow(y2-y0 ,2)) )>64)
                {
                    coordinates.add(new Coordinates(x2, y2));
                    x0=x2;
                    y0=y2;
                }
            }
        }
        else {
            p = 2 * dx - dy;

            while (y2 != y1) {
                y2 += stepY;
                if (p < 0) {
                    p += 2 * dx;
                } else {
                    p +=  2 * (dx - dy);
                    x2 += stepX;
                }
                if( ( Math.sqrt( Math.pow(x2-x0 ,2) + Math.pow(y2-y0 ,2)) )>64)
                {
                    coordinates.add(new Coordinates(x2, y2));
                    x0=x2;
                    y0=y2;
                }
            }
        }
    }
```

Los puntos que cumplen las características se guardan en una lista de coordenadas que se genera entre cada esfera

```
if (estado_juego==estados.Calculo){
    for(int i=1;i<numberBalls;i++)
    {

        for(int j=0;j<(numberBalls-i);j++)
        {
                if(j!=1)
                {
                    coordenadas_recta_disparo(ballarray[j].getX(),ballarray[j].getY(),ballarray[i].getX(),ballarray[i].getY());
                }
                //Se crean objetos en los puntos calculados
                for(int k=0;k<coordinates.size();k++)
                {
                    GameObject aux=new GameObject(thunderZoneImage,coordinates.get(k).getX(), coordinates.get(k).getY(), 1.0f);
                    aux.updateCurrentAnimation(0, 1, 1.25f);
                    zonethunder.add(aux);

                }
                coordinates.clear();
            }

        }
    estado_juego=estados.Espera;
```

Finalmente se muestran por pantalla recorriendo la lista aux donde se guardan todos los puntos de las rectas generadas.

Minijuego 3



El minijuego 3 está basado en el problema del lobo la oveja y la col y consiste en no dejar solos al yeti con el pingüino ni al pingüino con el pez.

La lógica del videojuego básicamente consiste en dos arrays un array de elementos del lado izquierdo y un array de elementos del lado derecho ,estos arrays se van actualizando según el personaje pulsa una tecla con la que lleva a uno de los personajes al otro lado

,continuamente se comprueba si se da alguno de los casos perdedores si no se cumple el juego sigue si se cumple el juego acaba.

```
switch(keyboard.getPressedp11(gc)){
    case("One"):
        if((vuelta==0 && Arrays.stream(personajesizq1).anyMatch("penguin"::equals)) || (vuelta==1 && Arrays.stream(personajesder1).anyMatch("penguin"::equals)))
        {
            aux1[0]=personajesizq1[0];
            aux1[3]=personajesizq1[3];
            personajesizq1[0]= personajesder1[0];
            personajesizq1[3]=personajesder1[3];
            personajesder1[0]= aux1[0];
            personajesder1[3]= aux1[3];
            aux1=new String[4];

            iceblock.changeAnimation(bloquepenguinImage,1.5f);
            iceblock.setY(80);
            iceblock.updateCurrentAnimation(0,0,1f);
            penguin.setVisible(false);
            tecla1.setVisible(false);
            player.setVisible(false);
            GameObject.paso=1;
            vuelta^=1;
            pulsado=true;
        }


        break;
```

Ejemplo de mover al pingüino

```
        //Comprobacion victoria o derrota
        for(int i=0;i<personajesizq1.length;i++)
        {
            if(personajesizq1[0]=="penguin" && personajesizq1[2]=="pez" && personajesizq1[3]!="human")
            {
                System.out.println("Derrota");
            }
            if(personajesizq1[1]=="yeti" && personajesizq1[0]=="penguin" && personajesizq1[3]!="human")
            {
                System.out.println("Derrota");
            }
            if(personajesder1[0]=="penguin" && personajesder1[2]=="pez" && personajesder1[3]!="human")
            {
                System.out.println("Derrota");
            }
            if(personajesder1[1]=="yeti" && personajesder1[0]=="penguin" && personajesder1[3]!="human")
            {
                System.out.println("Derrota");
            }
            if(personajesder1[0]=="penguin" && personajesder1[1]=="yeti" && personajesder1[2]=="pez" && personajesder1[3]=="human")
            {
                System.out.println("Victoria");
            }
        }
```

Comprobación de Derrota o victoria

Minijuego 4

El minijuego 4 consiste en pasar la bomba a tu compañero, éste es un juego competitivo. El personaje uno se controla con los controles W,A,S,D y espacio para pasar la bomba el personaje 2 se controla con las flechas de dirección y el numero 0 del teclado numerico para pasar la bomba.

Transcurrido un tiempo quien tenga la bomba pierde.

En este juego se juega con los extremos, que al ser cruzados te pueden llevar al azar o a la parte de arriba o la parte de abajo.

Es interesante no controlar donde puedes acabar para evitar la huida infinita.

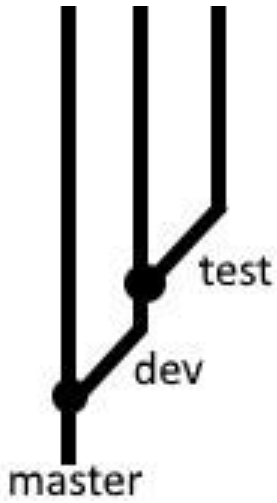
```
private void desplazamientospl1(GameContainer gc,int delta)
{
    //Desplazamientos al llegar al final
    if(player1.getX()>gc.getWidth() && player1.getY()==360)
    {
        player1.setY(getRandom(posArray));
        player1.updateX(xp1=10);

    }
    if(player1.getX()<0 && player1.getY()==360)
    {
        player1.setY(20);
        player1.updateX(xp1=100);
    }
    if(player1.getX()>gc.getWidth() && player1.getY()==20)
    {
        player1.setY(360);
        player1.updateX(xp1=gc.getWidth()-20);
    }
    if(player1.getX()<0 && player1.getY()==20)
    {
        player1.setY(360);
        player1.updateX(xp1=gc.getWidth()-20);
    }
}

    private void desplazamientospl2(GameContainer gc,int delta)
    {
        ////////////////////////////////////////////////////////////////
        if(player2.getX()>gc.getWidth() && player2.getY()==360)
        {
            player2.setY(getRandom(posArray));
            player2.updateX(xp2=10);

        }
        if(player2.getX()<0 && player2.getY()==360)
        {
            player2.setY(20);
            player2.updateX(xp2=100);
        }
        if(player2.getX()>gc.getWidth() && player2.getY()==20)
        {
            player2.setY(360);
            player2.updateX(xp2=gc.getWidth()-20);
        }
        if(player2.getX()<0 && player2.getY()==20)
        {
            player2.setY(360);
            player2.updateX(xp2=gc.getWidth()-20);
        }
    }

public static int getRandom(int[] array) {
    int rnd = new Random().nextInt(array.length);
    return array[rnd];
```

Implementacion llegada a cada borde para cada jugador

El resto de la lógica se centra en el movimiento de cada personaje y como activar cada animación según el movimiento que corresponda, la cual no tiene interés al ser muy repetitiva y similar a otra presentada ya anteriormente en otros minijuegos.

GitHub

Para llevar un control del desarrollo del videojuego, se ha trabajado con ramas, estableciendo una jerarquía, en la cual habría tres ramas principales: *master*, *dev* y *test*. Además, a parte de estas ramas principales, se van creando ramas para abordar los desarrollos de las distintas funcionalidades (sistema de turnos, minijuegos, mapa, menú, etc), las cuales son eliminadas una vez son *mergeadas* con *dev*.



Emplear esta estructura nos permite desarrollar el videojuego, con una cierta seguridad de que el código desde el que parten todos los desarrollos o tareas que se vayan a realizar, van a partir de un código correcto.

Cada rama tiene su función, *master* pasa a ser la rama en la que se suben versiones del videojuego para el público, es decir, *releases* del videojuego. La rama de *dev* es la más importante de cara al desarrollo del juego, pues todas las ramas que se vayan a crear para desarrollar las distintas funcionalidades del juego van a partir del código que se encuentre en esta rama. La rama de *test*, como indica su nombre, es una rama para *testear* los nuevos desarrollos del videojuego.

De cara a abordar un nuevo desarrollo (un minijuego, por ejemplo), se crea una rama a partir de *dev*, y se lleva a cabo todo el desarrollo en la nueva rama. Una vez está listo, se hace una *Pull Request* contra *test*, se hace una validación del código en la página de GitHub, y si todo está correcto, se *mergea* contra *test*. En test se prueba que funcione correctamente, y en caso de no detectarse ningún fallo, se vuelve a lanzar una *Pull Request* contra *dev*, se valida (esta vez la revisión es muy por encima, porque ya ha sido validado en test, pero por si acaso se pasó algo) y se *mergea*. Una vez la rama ha sido *mergeada* con *dev*, ya puede ser eliminada, tanto en local (en el ordenador de quien haya realizado el desarrollo) como la copia de la nube. A partir de ahora, todos los desarrollos nuevos partirán con el código del nuevo desarrollo como base.

Con esto, nos aseguramos en gran medida, que desarrollos incompletos, sin *testear*, o con algún fallo, sirvan como base a desarrollos posteriores, arrastrando así el error.

Estructura del proyecto

Para realizar el proyecto, desde el primer momento se ha planteado una estructura de clases que se iban a seguir, aunque debido a cambios en el rumbo del proyecto, algunas han tenido que sufrir ciertas modificaciones.

Lo más importante es que todo estuviera bien definido y separado, para que cada clase tuviera solo el código que le corresponde, fomentando así la reutilización de código.

Se ha establecido una estructura de paquetes, donde ir colocando las clases, manteniendo un orden lógico e intuitivo que nos facilitara el desarrollo del juego.

Como clases reseñables, contamos por ejemplo con las siguientes:

-ResourceLoader, una clase con métodos estáticos, los cuales utilizamos para cargar los sprites, que se vayan a utilizar desde los distintos sitios del videojuego.

-GameMode: contiene las reglas del juego.

-GameState: contiene el estado global de la aplicación en cada momento.

-GameObject: es la clase principal para los objetos y entidades que se van a emplear en los minijuegos. Se ha hecho lo suficientemente modular y genérica como para que nos permita utilizarla para todo tipo de minijuegos.

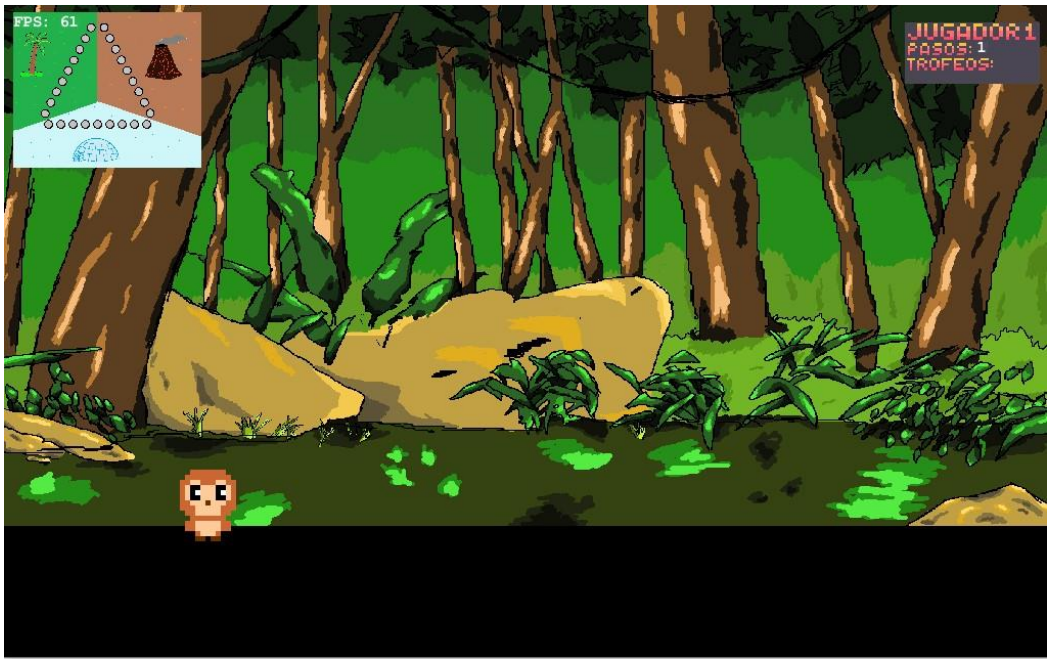
La utilidad que nos otorga esta estructura es poder reutilizar clases como GameObject en muchos sitios, así como contar con un código lo más limpio posible y tratando de buscar una buena eficiencia.

Mapa

El mapa está diseñado de tal forma que consultando el *GameState*, obtenga el jugador del turno actual y cargue el mapa en el que se encuentra el jugador para representarlo y colocar al jugador en la casilla correspondiente. El jugador puede avanzar tantas casillas como pasos disponga, los cuales obtendrá al completar los minijuegos (un paso si pierde y dos si gana).

Una vez los dos jugadores terminan su turno, el mapa hace una llamada a un método del *MainManager* para iniciar un minijuego. Al finalizar el minijuego, se volverá al mapa y comenzará de nuevo el turno del primer jugador, donde podrá moverse con los pasos que disponga.

Además, cuando los jugadores se encuentran en el mapa, disponen de un *HUD* el cual les muestra los pasos disponibles, los trofeos (funcionalidad no terminada) y un mini mapa, donde poder ver la posición en la que se encuentra en el mapa del juego.


FPS: 61
JUGADOR 1
PASOS: 1
TROFEOS: