UNIVERSIDAD INTERNACIONAL DE LAS AMÉRICAS

ESCUELA DE INGENIERÍA

INFORMÁTICA

Proyecto Final

CURSO:

Estructuras de Datos y Algoritmos

Nombre de (los) estudiante (s):

- **Joseline Chavarría**
- **Miguel Fernández Arteaga**
- **Byron García Astua**

PROFESOR

Daniel Álvarez Garro

SAN JOSÉ, COSTA RICA

Planteo del proyecto:

En el inicio del proyecto, se empezó con la escogencia del videojuego el cual en este caso fue: Donkey Kong Country: Tropical Freeze. Investigando, Gómez, J. (2021) indica que fue basado en plataformas en 2D y los encargados de su desarrollo fue Retro Studios, seguidamente Nintendo fue el encargado de realizar el lanzamiento oficial para WiiU en el 2014, luego 4 años más tarde en el 2018 se sumó a esto Nintendo Switch. (Párr., 3)

En el juego, el jugador controla a Donkey Kong y sus amigos mientras intentan detener a los invasores que han invadido su isla. Se deben completar una serie de recoleta de bananas en este caso y adicionalmente luchar contra obstáculos para poder el final de cada mundo en el nivel.

Una vez realizada la investigación sobre en que se basa el videojuego y la historia, lo primero fue poder buscar ideas de cómo realizar plataformas y cual sería la idea de interfaz gráfica donde se iba a aplicar el desarrollo, todo esto en JAVA usando el IDE Netbeans.

- **Modelo MVC:**

El patrón MVC es un enfoque arquitectónico utilizado, según Álvarez, M.A (2014), en el desarrollo de software para separar la lógica de presentación de los datos y la lógica de negocio en tres componentes distintos: el Modelo, la Vista y el Controlador. (Párr. 5)

Modelo: Representa los datos y la lógica de negocio. Es responsable de almacenar y recuperar datos, así como de procesar y aplicar las reglas de negocio sobre ellos. En nuestro proyecto se aplica con 4 clases de la siguiente forma:

1. Credits:
   Es la que se encarga de manejar y mostrar los créditos del jugador en el juego Donkey Kong, pues carga las imágenes necesarias y las utiliza para construir una imagen del puntaje actual del jugador, además tiene la capacidad de actualizar el puntaje y mostrar una imagen que representa la suma de uno a la puntuación actual.

Métodos usados:

Credits(): constructor de la clase. Carga las imágenes necesarias y actualiza el puntaje a cero.

loadImages(): método privado que carga las imágenes necesarias para mostrar los créditos.

getCreditNumber(int nro): método público que retorna la imagen que representa el número pasado como parámetro.

getPlusOne(): método público que retorna la imagen que representa la suma de uno a la puntuación actual.

updateScore(): método público que actualiza el puntaje y construye la imagen que representa el puntaje actual del jugador.

drawTitle(): método público que dibuja el título de los puntos.

drawTitle(): método público sobrecargado que dibuja el título en caso de que se le terminen los créditos.

getScore(): método público que retorna la imagen que representa el puntaje actual del jugador.

La clase tiene dos variables públicas:

- ✓ **width**: representa el ancho de los caracteres utilizados para mostrar los créditos.
- ✓ **height**: representa el alto de los caracteres utilizados para mostrar los créditos.

La clase tiene dos variables privadas:

- ✓ **imageCredits**: representa la imagen que contiene los caracteres para mostrar los créditos.
- ✓ **imageScore**: representa la imagen construida que representa el puntaje actual del jugador.

2. Jugador:

   Esta clase se encarga de definir las variables y métodos necesarios para representar al personaje del jugador en el juego, como su posición, tipo de jugador, estado actual (corriendo, saltando, cayendo, inactivo, etc.), tamaño, y las imágenes correspondientes a cada estado del jugador. La clase implementa la interfaz "Element" del controlador del juego, lo que significa que puede ser actualizada y representada en pantalla.

   Características de la clase jugador:

   - ✓ **Variables**: se definen variables de instancia para representar el tipo de jugador, el estado actual del personaje, la posición, el tamaño y las imágenes correspondientes a cada estado del jugador, entre otras.
   - ✓ **Constantes**: se definen constantes para representar los diferentes tipos de jugador y los estados del personaje.
   - ✓ **Constructor**: se define un constructor que recibe el tipo de jugador y la posición inicial como parámetros, y se encarga de cargar las imágenes correspondientes al jugador
   - ✓ **Métodos**: se definen varios métodos para actualizar el estado del jugador, establecer su estado como "corriendo", cargar las imágenes correspondientes, entre otros. También se implementan métodos de la interfaz "Element" (dentro de controlador) para que el controlador pueda actualizar y renderizar al jugador en pantalla.

3. Lives: Define una clase Java llamada "Lives" que es responsable de representar la cantidad de vidas disponibles para el jugador en un juego. La clase tiene las siguientes propiedades:

   Variables en uso:

   - ✓ "lives" que almacena la cantidad actual de vidas.
   - ✓ "title" que almacena el título de la barra de vidas.
   - ✓ Las variables "imageLives" e "imageAuxLives" que almacenan imágenes que representan la barra de vidas.

La clase tiene los siguientes métodos:

Un constructor que llama al método "drawLives()" para inicializar las imágenes de las vidas. El método "drawLives()" crea una instancia de BufferedImage con un ancho de 100 y un alto de 20 píxeles. Luego se crea un objeto a partir de esta instancia y se configura la fuente y el color de dibujo. Después se dibuja la cadena "Vidas:" seguida de la cantidad de vidas en la imagen en la posición (0,9).

Finalmente, se llama al método dispose() y se retorna la instancia de BufferedImage resultante. El método "getLives()" devuelve la imagen de las vidas almacenada en "imageAuxLives", seguidamente usa "getLivesCount()" devuelve la cantidad actual de vidas almacenada en "lives", el "setLivesCount()" establece la cantidad de vidas en "lives", asegurándose de que no sea menor que cero, adicionalmente "loseLife()" llama al método "drawLives()" para actualizar la imagen de las vidas.

4. Nodo: Representa un nodo de una lista enlazada. En una lista enlazada, cada nodo almacena un elemento y una referencia al siguiente nodo en la lista. La clase contiene con un tipo genérico T que representa el tipo de dato que se almacenará en el nodo.

   Atributos:
   - ✓ Dato: objeto de tipo T que se almacenará en el nodo.
   - ✓ Siguiente: referencia al siguiente nodo en la lista.

   Métodos:
   - ✓ Nodo (): constructor que recibe el objeto que se almacenará en el nodo.
   - ✓ getDato (): método que retorna el objeto almacenado en el nodo.
   - ✓ setDato (): método que establece el objeto que se almacenará en el nodo.
   - ✓ getSiguiente (): método que retorna la referencia al siguiente nodo en la lista.
   - ✓ setSiguiente (): método que establece la referencia al siguiente nodo en la lista.

Vista: Es la interfaz de usuario que muestra los datos al usuario final. Se encarga de presentar los datos al usuario de una manera fácil de entender y proporcionar interacción.

1. Fondo: Es responsable de cargar una imagen de fondo desde un archivo, actualizar la posición del fondo en la pantalla y proporcionar una sección de la imagen del fondo que se ajusta a las dimensiones del marco en el que se está mostrando. La clase cuenta con varias variables, como posición que representa la posición actual del fondo en la pantalla, varias variables BufferedImage que almacenan las imágenes de fondo, y una variable booleana que indica si el fondo está compuesto por una o dos imágenes.

   Además, tiene varios métodos para cargar imágenes, actualizar la posición del fondo y devolver una sección de la imagen del fondo que se ajusta a las dimensiones del marco.

2. frmPrincipal: Es el encargado de inicializar las instancias iniciales de los objetos y de definir y ejecutar las funciones del juego. Crea diferentes variables como FRAME_WIDTH, FRAME_HEIGTH, DELAY; además cuenta con instancias de clases como Fondo, Jugador y Timer que son de las principales para llevar a cabo el juego. Además, se definen valores iniciales como cantidad de créditos del jugador, cantidad de vidas, tamaño del frame y delay del timer. Utiliza métodos como drawList, paint, paintDoubleBuffered, isBoxCollision, actualizarFondo, actualizarFrutas, generarFrutas y generarCajas, que son los encargados de confirmar que el juego funcione de manera correcta haciendo las validaciones y actualizando los valores.

Controlador: Se encarga de recibir y manejar las solicitudes del usuario, procesarlas y enviarlas al Modelo para su procesamiento. El Controlador también es responsable de ayudar con la actualizar de la parte Vista con los resultados del procesamiento del Modelo, en el proyecto se usan un total de 4 clases y 1 una interfaz.

1. Element: La interfaz de element define un conjunto de métodos que debe implementar cada clase que representa un elemento en la pantalla de un juego o aplicación gráfica. Los métodos getX() y getY() devuelven las coordenadas X/Y del elemento, getWidth() y getHeight() devuelven el ancho y el alto del elemento. Además, la interfaz de usuario

define un método getImage() que devuelve la imagen asociada con el elemento en formato BufferedImage.

2. Box: Implementa la interfaz "Element", lo que significa que una caja es un elemento del juego que se puede usar con otros elementos. Además, tiene una serie de variables que definen alto, ancho, posición y tipo. También hay algunas que se utilizan para verificar si la casilla está rota y visible en el juego. Adicionalmente se agregó un constructor que se usa para crear una nueva instancia de un cuadro del tipo y ubicación dados.

   Métodos:

   - ✓ **Box (int type, Point initialPosition)**: constructor que crea una nueva instancia de la clase Box con el tipo y la posición especificados.
   - ✓ **loadImages ():** Método privado que carga las imágenes de la caja.
   - ✓ **updateBox ():** método público que actualiza el cuadro.
   - ✓ **getImage ()**: Método público que devuelve la imagen correspondiente a la caja.
   - ✓ **setY ():** método público que establece la coordenada y del playbox.
   - ✓ **isBreak ():** Método público que regresa si la caja está rota.
   - ✓ **setBreak ():** Método público que marca la caja como rota.
   - ✓ **isVisible ():** Método público que devuelve la visibilidad de la caja.

3. ListaArreglo: La clase es una implementación de una lista enlazada simple implementada con un arreglo de nodos. En el constructor, se inicializa una lista vacía con nombre cabeza = null y tamaño = 0. La clase contiene métodos para agregar y eliminar elementos de la lista, buscar un elemento en la lista y obtener el elemento en el índice dado. Además, se implementa un método que verifica si la lista está vacía.

   Esta clase usa la clase genérica Nodo para representar un nodo en una lista enlazada. El tipo de datos T definido en la declaración de clase se utiliza para representar el tipo de datos contenidos en la lista. La clase también implementa la interfaz Iterable y su método iterador (), que permite iterar sobre los elementos de una lista usando un bucle for-each.

Es decir, la clase ListaArreglo proporciona una implementación para una lista enlazada simple que arreglos de matrices de nodos y métodos para agregar, eliminar y buscar elementos en la lista.

4. IteradorListaArreglo: IteratorArrayList, que implementa la interfaz Iterator. El tipo genérico T representa el tipo de datos que se almacenarán en la lista. En la definición de clase, tenemos tres variables de instancia: lista, actual y anterior.

   - ✓ Lista: es una referencia a la lista a partir de la cual se creó el iterador.
   - ✓ Actual es una referencia para atravesar el nodo actual en la lista vinculada
   - ✓ Anterior: es una referencia al nodo anterior al actual.
   - ✓ El constructor de la clase recibe como parámetro una instancia de ArrayList , que es la lista a partir de la cual desea crear un iterador. En este constructor, se inicializan los nodos actual y anterior.

   Métodos:

   El método hasNext () es un método que devuelve verdadero si hay un siguiente elemento en la lista y falso en caso contrario. Este método solo verifica si el nodo actual está vacío. Luego, método next () es lo que devuelve el siguiente elemento de la lista y mueve el cursor al siguiente nodo. Este método devuelve los datos almacenados en el nodo actual y luego actualiza la referencia actual al siguiente nodo de la lista.

5. Weapon: Este código es una que representa el arma en el juego. Esta clase implementa la interfaz "Element", que define métodos comunes para los elementos gráficos del juego.

   Esta clase tiene las siguientes variables:

   - ✓ width y height: son números enteros que representan el ancho y el alto de la imagen del arma, respectivamente.
   - ✓ Position representa la posición del arma en la pantalla.

- ✓ Visible: es un booleano que indica si el arma es visible o no.
- ✓ imageWeapon: representa unaimagen de un arma.
- ✓ imageNumber: representa el número de imagen de animación del arma

Métodos:

- ✓ Weapon es un constructor de clase que inicializa la posición del arma
- ✓ getImage devuelve la imagen actual que el arma presenta
- ✓ isVisible indica si el objeto del arma está visible en el juego.

- ✓ setVisible cambia la visibilidad en el juego del objeto arma.
- ✓ loadImages es un método privado que carga imágenes de armas desde un archivo
- ✓ updateWeapon actualiza el estado del arma. Mueve el arma, cambia su imagen y visibilidad.

- **TAD:**

  Anteriormente en clase se vio lo que es conocido como Tipos de Datos Abstractos, por medio de ellos los desarrolladores puedes abstraer conceptos de la programación creando un tipo distinto de dato. Para el proyecto, se implementó en este caso en la clase WEAPON ubicada en la parte de controlador, en este caso podemos ver que es una implementación ya que:

  1. Logra definir qué datos guardar: posición del arma, tamaño, imagen, visibilidad.
  2. Define qué operaciones se pueden realizar con estos datos, ejemplo obtener la imagendel arma, obtener la posición, obtener el tamaño, obtener visibilidad, cambiar lavisibilidad, actualizar el estado.

```java
public class Weapon implements Element {

    //Variables
    private int width = 19; // Ancho de la imagen de la arma.
    private int height = 19; // Alto de la imagen de la arma.

    private Point position; // Posición de la arma en la pantalla.

    private boolean visible = true; // Indica si la arma es visible o no.
    private BufferedImage imageWeapon; // Imagen de la arma.

    private int imageNumber = 0; //El numero de imagen de la animacion del arma

    /**
     * Constructor inicializa la posición del arma y carga las imágenes
     * asociadas.
     *
     * @param initialPosition , Objeto Point que representa la posición inicial
     * del arma.
     */
    public Weapon(Point initialPosition) {
        this.position = initialPosition;

        loadImages();
    }
```

Fuente: Elaboración Propia

- **Uso de Estructuras de Datos:**

  En nuestro proyecto se implementaron las estructuras de datos usando "lista enlazada". Dentro de modelo esta la clase NODO, la cual cuenta con dos atributos: "dato", que es el objeto que se almacenará en el nodo, y "siguiente", que es una referencia al siguiente nodo en la lista. se definen los métodos "getDato()" y "getSiguiente()" para obtener el objeto almacenado en el nodo y el siguiente nodo en la lista, respectivamente, y los métodos "setDato()" y "setSiguiente()" para establecer el objeto que se almacenará en el nodo y el siguiente nodo en la lista, respectivamente.

```java
package donkey_kong.modelo;

/**
 * Clase que representa un nodo de una lista enlazada.
 *
 * @param <T> tipo de dato que almacenará el nodo.
 */
public class Nodo<T> {
    private T dato;
    private Nodo<T> siguiente;
    /**
     * Constructor de la clase Nodo.
     *
     * @param dato objeto que se almacenará en el nodo.
     */
    public Nodo(T dato) {
        this.dato = dato;
        this.siguiente = null;
    }
    /**
     * Obtiene el objeto almacenado en el nodo.
     *
     * @return objeto almacenado en el nodo.
     */
    public T getDato() {
        return dato;
    }
```

Fuente: Elaboración Propia.

```java
public T getDato() {
    return dato;
}
/**
 * Establece el objeto que se almacenará en el nodo.
 *
 * @param dato objeto que se almacenará en el nodo.
 */
public void setDato(T dato) {
    this.dato = dato;
}
/**
 * Obtiene el siguiente nodo en la lista enlazada.
 * @return siguiente nodo en la lista enlazada.
 */
public Nodo<T> getSiguiente() {
    return siguiente;
}
/**
 * Establece el siguiente nodo en la lista enlazada.
 * @param siguiente siguiente nodo en la lista enlazada.
 */
public void setSiguiente(Nodo<T> siguiente) {
    this.siguiente = siguiente;
}
```

Fuente: Elaboración Propia.

- **Programación Recursiva:**

Se aplica en la clase **frmPrincipal** ubicada en la parte de modelo. Representa un método recursivo llamado **generarCajas**. La idea básica de este método es crear una cantidad específica de cajas en posiciones aleatorias y verificar que no se choquen entre sí. Para poder realizar esto toma como parámetro a cantidad que indica cuantos son las cajas que se deben crear, si CANTIDAD llega 0 el método se va a finalizar, si no es de esa forma, continuaría creando cajas nuevas llamándose a sí mismo para la creación de la siguiente caja.

Siempre que se llame al método va a hacer que la cantidad de cajas por crear se reduzca en 1, entonces, por ejemplo, si se llamó inicialmente con una cantidad de 9, la primera vez creara una caja y luego de esto llamara de nuevo al método con una cantidad de 8.

Una segunda llamada sería crear de igual forma la caja y llamara esta vez con una cantidad de 7, y así sucesivamente.

```
private void generarCajas(int cantidad){
    if (cantidad == 0) {
    return;
}
// Generar posición aleatoria en el eje X
int randomX = (int) (Math.random() * 350) - 100;
int elementoX = jugador.getX() + 100 + (randomX + 80);

int posY = 390;
// Crear nueva caja y agregarla a la lista
Box nuevaBox = new Box(type: Box.BOX1, new Point(x: elementoX, y: posY));
// Verificar si la nueva caja colisiona con alguna de las cajas existentes
boolean colision = false;
for (Box cajaExistente : listBox) {
    if (isBoxCollision(elemA: nuevaBox, elemB: cajaExistente)) {
        colision = true;
        break;
    }
}
// Si no hay colisión, agregar la nueva caja a la lista
if (!colision) {
    listBox.agregarAlFinal(elemento: nuevaBox);
}
generarCajas(cantidad - 1);
}
```

Fuente: Elaboración Propia.

Referencias:

Alvarez, M. A. (2014, enero 2). Qué es MVC. Desarrolloweb.com. https://desarrolloweb.com/articulos/que-es-mvc.html

Gómez, J. (2021, junio 9). Historia de Donkey Kong: ¡Conoce todos los detalles! Tokio School. https://www.tokioschool.com/noticias/historia-donkey-kong/