Criterio C: Desarrollo

Técnicas utilizadas

1. Interfaz gráfica
2. Uso de las clases Graphics y BufferStrategy
3. Pensamiento algorítmico: movimiento de Pacman
4. Clase anidada Nodo: pathfinding para el movimiento de los fantasmas

1. Interfaz gráfica

Pacman es un juego en el que es obligatorio tener un componente visual. Debido a que esto no puede ser hecho en la consola, es necesario tener una interfaz gráfica para el usuario que muestre el mapa, a Pacman, a los fantasmas, y a la comida.

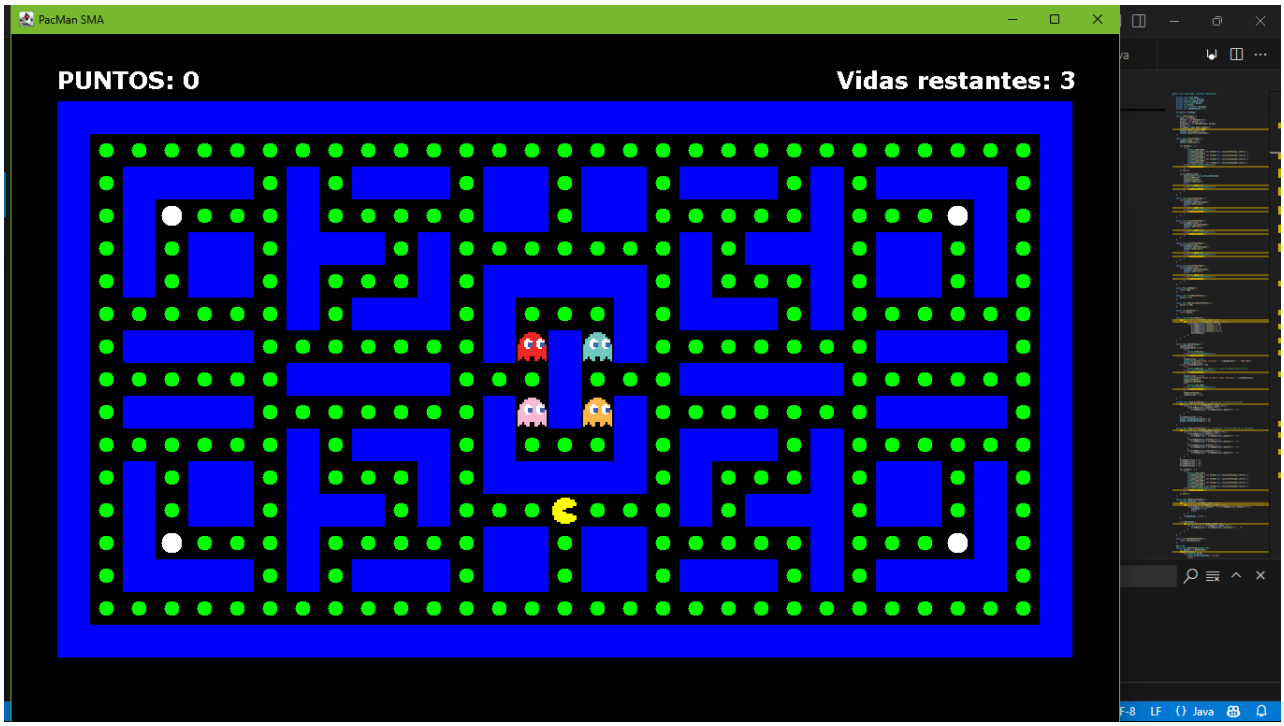

Imagen 1: Captura del producto final.

La interfaz gráfica ha sido creada utilizando la clase `JFrame` para sacar una ventana que más adelante será usada como contenedor para graficar los elementos del juego. Captura del código en la clase `Ventana`:

```
public Ventana(Controlador controlador) {
    this.controlador = controlador;
    setTitle("PacMan SMA");
    setSize(1200, 800);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    setResizable(false);
    createBufferStrategy(2);
}
```

Imagen 2: Constructor de la clase Ventana.

2. Uso de las clases Graphics y BufferStrategy

En vez de usar un `Canvas` o `JPanel`, opté por el uso de la clase `Graphics` y la clase `BufferStrategy` para mejorar y optimizar el renderizado de los elementos del juego. El proceso de renderizado se gestiona a través del método `renderizar()`, que utiliza un objeto de la clase `BufferStrategy` creado con `createBufferStrategy(2)` para implementar doble buffering:

```
public void renderizar() {
    pacmanArriba = new ImageIcon(getClass().getResource("Imágenes/pacmanArriba.png")).getImage();
    pacmanAbajo = new ImageIcon(getClass().getResource("Imágenes/pacmanAbajo.png")).getImage();
    pacmanIzquierda = new ImageIcon(getClass().getResource("Imágenes/pacmanIzquierda.png")).getImage();
    pacmanDerecha = new ImageIcon(getClass().getResource("Imágenes/pacmanDerecha.png")).getImage();

    fantasma1 = new ImageIcon(getClass().getResource("Imágenes/fantasma1.png")).getImage();
    fantasma2 = new ImageIcon(getClass().getResource("Imágenes/fantasma2.png")).getImage();
    fantasma3 = new ImageIcon(getClass().getResource("Imágenes/fantasma3.png")).getImage();
    fantasma4 = new ImageIcon(getClass().getResource("Imágenes/fantasma4.png")).getImage();

    BufferStrategy bufferStrategy = getBufferStrategy();
    if (bufferStrategy == null) {
        createBufferStrategy(2);
        return;
    }
```

Imagen 3: Inicio del método `renderizar()` e inicialización del buffering.

Para dibujar en la ventana se crea una instancia de la clase Graphics g, ya que la clase Graphics proporciona los métodos necesarios para dibujar los elementos del juego en la interfaz gráfica de usuario:

```
Graphics g = bufferStrategy.getDrawGraphics();
```

Imagen 4: Declaración de la instancia `Graphics g`.

Tras declarar la instancia g, en el proceso de renderización se empieza dibujando el fondo de negro con una recta que cubre toda la ventana:

```
if (g != null) {
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, getWidth(), getHeight());

    if (controlador.getVidasRestantes() > 0) {
        mapa = controlador.getMapa().getArrayMapa1();
    } else if (controlador.getVidasRestantes() <= 0) {
        mapa = controlador.getMapa().getArrayMapaGameOver();
    }
```

Imagen 5: Renderizar el fondo negro.

Tras el fondo, se renderizan las paredes y la comida con figuras simples (rectángulos y círculos):

```
// RENDERIZAR PAREDES
for (int i = 0; i < mapa.length; i++) {
    for (int j = 0; j < mapa[i].length; j++) {
        if (mapa[i][j].equals("X")) {
            if (controlador.getVidasRestantes() > 0) {
                g.setColor(Color.BLUE);
            } else if (controlador.getVidasRestantes() <= 0) {
                g.setColor(Color.RED);
            }
            g.fillRect(j * tamañoCelda + offsetX, i * tamañoCelda + offsetY, tamañoCelda, tamañoCelda);
        }
    }
}

// RENDERIZAR COMIDA
for (int i = 0; i < mapa.length; i++) {
    for (int j = 0; j < mapa[i].length; j++) {
        if (mapa[i][j].contains(" ")) {
            g.setColor(Color.GREEN);
            int centroX = j * tamañoCelda + offsetX + tamañoCelda / 2;
            int centroY = i * tamañoCelda + offsetY + tamañoCelda / 2;
            int radioComida = tamañoCelda / 4;
            g.fillOval(centroX - radioComida, centroY - radioComida, radioComida * 2, radioComida * 2);
        }
    }
}

// RENDERIZAR SUPERCOMIDA
for (int i = 0; i < mapa.length; i++) {
    for (int j = 0; j < mapa[i].length; j++) {
        if (mapa[i][j].contains("S")) {
            g.setColor(Color.WHITE);
            int centroX = j * tamañoCelda + offsetX + tamañoCelda / 2;
            int centroY = i * tamañoCelda + offsetY + tamañoCelda / 2;
            int radioComida = tamañoCelda / 3;
            g.fillOval(centroX - radioComida, centroY - radioComida, radioComida * 2, radioComida * 2);
        }
    }
}
```

Imagen 6: Renderizado de paredes, comida y supercomida.

Para cada elemento se revisan las celdas del array bidimensional del mapa. Si se encuentra una “X” (pared), un espacio “ “ (comida), o una “S” (supercomida), se renderiza acorde al elemento; las paredes son

bloques azules, la comida normal círculos verdes pequeños, y la supercomida círculos blancos más grandes.

Tras renderizar las figuras simples se renderiza la imagen de Pacman:

```
// RENDERIZAR PACMAN
for (int i = 0; i < mapa.length; i++) {
    for (int j = 0; j < mapa[i].length; j++) {
        if (mapa[i][j].contains("P")) {
            int centroX = j * tamañoCelda + offsetX + tamañoCelda / 2;
            int centroY = i * tamañoCelda + offsetY + tamañoCelda / 2;
            int radioPacman = tamañoCelda / 2;

            String direccion = Pacman.direccion;
            if (direccion != null) {
                switch (direccion) {
                    case "arriba":
                        pacmanDireccion = pacmanArriba;
                        break;
                    case "abajo":
                        pacmanDireccion = pacmanAbajo;
                        break;
                    case "izquierda":
                        pacmanDireccion = pacmanIzquierda;
                        break;
                    case "derecha":
                        pacmanDireccion = pacmanDerecha;
                        break;
                }
            } else {
                pacmanDireccion = pacmanDerecha;
            }

            g.drawImage(pacmanDireccion, centroX - radioPacman, centroY - radioPacman, radioPacman * 2, radioPacman * 2, null);
        }
    }
}
```

Imagen 7: Renderizado de Pacman.

Depende de la dirección en la que esté yendo Pacman, la imagen que aparece en la celda varía. La dirección de Pacman se saca de la variable `dirección` proveniente de la clase `Pacman`. Tras renderizar a Pacman, se renderizan los fantasmas:

```java
// RENDERIZAR FANTASMAS
for (int i = 0; i < mapa.length; i++) {
    for (int j = 0; j < mapa[i].length; j++) {
        if (mapa[i][j].contains("1")) {
            g.drawImage(fantasma1, j * tamañoCelda + offsetX, i * tamañoCelda + offsetY, tamañoCelda, tamañoCelda, null);
        }
        if (mapa[i][j].contains("2")) {
            g.drawImage(fantasma2, j * tamañoCelda + offsetX, i * tamañoCelda + offsetY, tamañoCelda, tamañoCelda, null);
        }
        if (mapa[i][j].contains("3")) {
            g.drawImage(fantasma3, j * tamañoCelda + offsetX, i * tamañoCelda + offsetY, tamañoCelda, tamañoCelda, null);
        }
        if (mapa[i][j].contains("4")) {
            g.drawImage(fantasma4, j * tamañoCelda + offsetX, i * tamañoCelda + offsetY, tamañoCelda, tamañoCelda, null);
        }
    }
}
```

Imagen 8: Renderizado de los fantasmas.

Para cada fantasma hay una imagen, al igual que con Pacman. Las variables de las imágenes aparecen en la Imagen 3. Al encontrar el número correspondiente, se inserta la imagen en la celda de la ventana. Tras renderizar todas esas cosas, lo último que queda por renderizar son los textos del contador de puntaje y de vidas restantes:

```java
// RENDERIZAR PUNTOS
g.setColor(Color.WHITE);
g.setFont(new Font("Verdana", Font.BOLD, 25));
g.drawString("PUNTOS: " + controlador.getPuntos(), 57, 90);

// RENDERIZAR VIDAS
g.setColor(Color.WHITE);
g.setFont(new Font("Verdana", Font.BOLD, 25));

if (controlador.getVidasRestantes() >= 0) {
    g.drawString("Vidas restantes: " + controlador.getVidasRestantes(), 890, 90);
} else {
    g.drawString("Vidas restantes: 0", 890, 90);
}
```

Imagen 9: Renderizado de textos.

3. Pensamiento algorítmico: Pacman

El movimiento de Pacman se controla usando las flechas del teclado (arriba, abajo, izquierda, derecha). Tras presionar una de estas flechas, el método keyPressed() en la clase Controlador recibe la tecla presionada. Luego pasa por un switch en el que depende que tecla haya sido presionada establece la dirección solicitada modificando la variable direccionDeseada:

```java
@Override
public void keyPressed(KeyEvent e) {
    int keyCode = e.getKeyCode();
    switch(keyCode) {
        case KeyEvent.VK_UP:
            Pacman.direccionDeseada = "arriba";
            break;
        case KeyEvent.VK_DOWN:
            Pacman.direccionDeseada = "abajo";
            break;
        case KeyEvent.VK_RIGHT:
            Pacman.direccionDeseada = "derecha";
            break;
        case KeyEvent.VK_LEFT:
            Pacman.direccionDeseada = "izquierda";
            break;
    }
}
```

Imagen 10: Cambio de dirección al presionar las teclas de flechas.

Mientras el juego está iniciado y en marcha, un bloque `while` dentro del método `iniciarJuego()` (que es llamado cuando se inicia o reinicia la partida) llama constantemente al método `mover()` de la clase `Pacman` a través de una instancia. Al llamar al método se introduce como argumento la variable `direccionDeseada`.

```java
while(juegoIniciado) {
    pacman.mover(Pacman.direccionDeseada);
    verificarMuerte();
    reaparecerComida();
    ventana.renderizar();
    try {
        Thread.sleep(120);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Imagen 11: Bloque `while` *dentro del método* `iniciarJuego()`.

El movimiento de Pacman está controlado por el método `mover()` en la clase `Pacman`. Al ser llamado desde la clase `Controlador` con la dirección solicitada en `direccionDeseada`, se itera por todo el array bidimensional en busca de Pacman ("P"). Al encontrarlo, se entra a un switch en el que depende de la dirección deseada, se comprueba si el siguiente paso con esa dirección es posible. Solo permitirá el movimiento de Pacman en caso de que haya o comida, o supercomida, o un espacio ya comido anteriormente. Si hay una pared o un fantasma, no permitirá el movimiento. Para moverse, se intercambia la constante de Pacman ("P") entre las celdas del array bidimensional.

```
// Método para mover a Pacman en la dirección deseada
public void mover(String direccionDeseada) {
    if(Controlador.juegoIniciado) {
        for(int i = 0; i < mapa.length; i++) {
            for(int j = 0; j < mapa[i].length; j++) {
                if(mapa[i][j].contains("P")) {
                    switch(direccionDeseada) {
                        case "arriba":
                            if(mapa[i - 1][j].contains(" ") || mapa[i - 1][j].contains("S") || mapa[i - 1][j].contains("C") || mapa[i - 1
                                controlador.verificarMuerte();
                                if(mapa[i - 1][j].contains(" ")) {
                                    controlador.incrementarPuntos();
                                } else if(mapa[i - 1][j].contains("S")) {
                                    controlador.SuperIncrementarPuntos();
                                }
                                mapa[i - 1][j] += "P";
                                mapa[i][j] = mapa[i][j].replaceAll(" ", "C");
                                mapa[i][j] = mapa[i][j].replaceAll("S", "C");
                                mapa[i][j] = mapa[i][j].replaceAll("P", "");
                                direccion = "arriba";
                                posicionY = i - 1;
                                posicionX = j;
                                coordenadasPacman[0] = posicionY;
                                coordenadasPacman[1] = posicionX;
                                return;
                            } else if(mapa[i - 1][j].contains("X")) {
                                direccionDeseada = direccion;
                                direccionEnEspera = "arriba";
                            }
                            break;
```

Imagen 12: Método `mover()` *en la clase* `Pacman`.

La comprobación de si Pacman está en contacto con un fantasma (por ende, debe morir) se hace en la clase Controlador, en la que se comprueba si está en la misma celda (o estará al siguiente movimiento) Pacman ("P") y al menos uno de los fantasmas ("1", "2", "3", "4"):

```
public void verificarMuerte() {
    for(int i = 0; i < arrayMapa.length; i++) {
        for(int j = 0; j < arrayMapa[i].length; j++) {
            if(arrayMapa[i][j].contains("P") &&
               (arrayMapa[i][j].contains("1") ||
                arrayMapa[i][j].contains("2") ||
                arrayMapa[i][j].contains("3") ||
                arrayMapa[i][j].contains("4"))) {
                muertePacman();
            }
        }
    }
}
```

Imagen 13: Método `verificarMuerte()` *en la clase* `Controlador`.

4. Clase anidada Nodo: pathfinding para el movimiento de los fantasmas

El movimiento de los fantasmas es una de las partes más complejas del producto. Este requiere de cálculos para perseguir al objetivo (Pacman), también conocido como pathfinding. Para calcular el siguiente paso de cada fantasma para intentar tocar a Pacman, se implementó un famoso algoritmo llamado A* (A-star), que calcula el mejor paso siguiente (calculando por nodos, cada celda) teniendo en cuenta costos (como distancias) para llegar al objetivo.

Este algoritmo tiene en cuenta el movimiento anterior y el objetivo, cosa que lo hace más eficiente para evitar movimientos en círculo o repetitivos. El algoritmo A* está implementado en la clase Fantasma, como una subclase, o clase anidada, llamada `Nodo`:

- Clase que representa un nodo en el camino. Los nodos representan las celdas en el mapa.
- Se utiliza para representar cada celda en el camino que el fantasma tomará para alcanzar a Pacman.
- Cada nodo contiene su posición (x, y), los costos g, h y f, y una referencia a su nodo padre.
- El costo g representa la distancia desde el nodo inicial hasta el nodo actual.
- El costo h es una estimación de la distancia desde el nodo actual hasta el nodo objetivo.
- El costo f es la suma de g y h, y se utiliza para determinar el nodo con el menor costo total.

```java
private class Nodo {
    int x, y; // Coordenadas del nodo en el mapa
    int g, h, f; // Costos g, h y f. Los costos son la distancia de celdas que cuesta llegar al nodo en (x, y)
    Nodo padre; // Referencia al nodo padre

    // Constructor que inicializa las coordenadas, el nodo padre y los costos g y h
    Nodo(int x, int y, Nodo padre, int g, int h) {
        this.x = x; // Coordenada x del nodo actual
        this.y = y; // Coordenada y del nodo actual
        this.padre = padre; // Nodo padre (nodo anterior al actual)
        this.g = g; // Costo desde el nodo inicial hasta este nodo
        this.h = h; // Estimación del costo desde este nodo hasta el nodo objetivo
        this.f = g + h; // Costo total (g + h)
    }

    // Método equals para comparar dos nodos
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Si los objetos son iguales, devuelve true
        if (obj == null || getClass() != obj.getClass()) return false; // Si el objeto es nulo o no es de la misma clase, devuelve false
        Nodo nodo = (Nodo) obj; // Convierte el objeto a Nodo
        return x == nodo.x && y == nodo.y; // Compara las coordenadas x e y
    }

    // Método hashCode para generar un código hash para el nodo
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

Imagen 14: Clase anidada `Nodo`*, integrada en la clase* `Fantasma`*.*

Mientras el juego está activo e iniciado, el método `moverFantasma()` es llamado constantemente.

```java
// Método para mover un fantasma en una dirección específica
private void moverFantasma(int fantasma, String idFantasma) {
    String[][] arrayMapa = mapa.getArrayMapa1();
    for (int i = 0; i < arrayMapa.length; i++) {
        for (int j = 0; j < arrayMapa[i].length; j++) {
            if (arrayMapa[i][j].contains(idFantasma)) {
                List<Nodo> camino = encontrarCamino(i, j, pacman.coordenadasPacman[0], pacman.coordenadasPacman[1]);
                if (camino != null && camino.size() > 1) {
                    Nodo siguientePaso = camino.get(1);
                    arrayMapa[siguientePaso.x][siguientePaso.y] += idFantasma;
                    arrayMapa[i][j] = arrayMapa[i][j].replaceAll(idFantasma, "");
                } else {
                    int[][] direcciones = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};
                    Collections.shuffle(Arrays.asList(direcciones));
                    for (int[] dir : direcciones) {
                        int nuevoX = i + dir[0];
                        int nuevoY = j + dir[1];
                        if (nuevoX >= 0 && nuevoY >= 0 && nuevoX < arrayMapa.length && nuevoY < arrayMapa[0].length && !arrayMapa[nuevoX][nuevoY].contains("X")) {
                            arrayMapa[nuevoX][nuevoY] += idFantasma;
                            arrayMapa[i][j] = arrayMapa[i][j].replaceAll(idFantasma, "");
                            break;
                        }
                    }
                }
                return;
            }
        }
    }
}
```

Imagen 15: Método `moverFantasma()`*.*

Mientras el juego está activo e iniciado, el método `moverFantasma()` es llamado constantemente. Al encontrar las coordenadas del fantasma, llama a otro método; el método `encontrarCamino()`:

```java
// Método para encontrar el camino desde una posición inicial a una posición final
private List<Nodo> encontrarCamino(int startX, int startY, int endX, int endY) { // endX/endY son las coordenadas de Pacman, y startX/startY las del fantasma
    PriorityQueue<Nodo> abierta = new PriorityQueue<>(Comparator.comparingInt(n -> n.f)); // Cola de prioridad para los nodos abiertos, ordenada por el costo total f
    Set<Nodo> cerrada = new HashSet<>(); // Conjunto de nodos cerrados
    abierta.add(new Nodo(startX, startY, null, 0, Math.abs(startX - endX) + Math.abs(startY - endY))); // Añadir el nodo inicial a la cola de prioridad

    while (!abierta.isEmpty()) { // Mientras haya nodos en la cola de prioridad
        Nodo actual = abierta.poll(); // Obtener el nodo con el menor costo total f
        if (actual.x == endX && actual.y == endY) { // Si el nodo actual es el nodo objetivo
            List<Nodo> camino = new ArrayList<>(); // Crear una lista para almacenar el camino
            while (actual != null) { // Mientras el nodo actual no sea nulo
                camino.add(actual); // Añadir el nodo actual al camino
                actual = actual.padre; // Moverse al nodo padre
            }
            Collections.reverse(camino); // Invertir el camino para obtener el orden correcto
            return camino; // Devolver el camino
        }

        cerrada.add(actual); // Añadir el nodo actual al conjunto de nodos cerrados

        for (int[] dir : new int[][]{{0, -1}, {0, 1}, {-1, 0}, {1, 0}}) { // Para cada dirección posible (arriba, abajo, izquierda, derecha)
            int nuevoX = actual.x + dir[0]; // Calcular la nueva coordenada x
            int nuevoY = actual.y + dir[1]; // Calcular la nueva coordenada y
            if (nuevoX < 0 || nuevoY < 0 || nuevoX >= mapa.getArrayMapa1().length || nuevoY >= mapa.getArrayMapa1()[0].length) { // Si la coordenada está fuera del mapa
                continue; // Saltar a la siguiente iteración
            }
            if (mapa.getArrayMapa1()[nuevoX][nuevoY].equals("X")) { // Si la nueva coordenada es un obstáculo
                continue; // Saltar a la siguiente iteración
            }
            Nodo vecino = new Nodo(nuevoX, nuevoY, actual, actual.g + 1, Math.abs(nuevoX - endX) + Math.abs(nuevoY - endY)); // Crear un nuevo nodo vecino
            if (cerrada.contains(vecino)) { // Si el nodo vecino ya está en el conjunto de nodos cerrados
                continue; // Saltar a la siguiente iteración
            }
            abierta.add(vecino); // Añadir el nodo vecino a la cola de prioridad
        }
    }
    return null; // Si no se encuentra un camino, devolver null
}
```

Imagen 16: Método `encontrarCamino()`.

Este método crea una cola de prioridad para los nodos abiertos, ordenada por el costo total (f) de cada uno. Cada nuevo nodo se añade a la cola. Mientras hay nodos en la cola, se encuentra el nodo con menor costo y si el nodo que se está inspeccionando es el objetivo, se añade el nodo calculado a la lista de pasos a seguir. Tras esto, cambia el nodo actual al calculado, así generando el cambio de posición. Mientras hay nodos en la cola se calculan posibles coordenadas a calcular como nodos. Se devuelve constantemente una lista de nodos como pasos a seguir para cada fantasma, que se actualiza a medida que Pacman se mueve.

Bibliografía:

- alfredonuhe. (2018). *GitHub - alfredonuhe/pacman-astar-search: Implementation of A* search algorithm for pacman game.* GitHub. https://github.com/alfredonuhe/pacman-astar-search. Consultado el 13 de febrero de 2025.
- Belwariar, R. (30 de julio, 2024). *A* Search Algorithm - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm. Consultado el 14 de febrero de 2025.
- Cox, G. (20 de noviembre, 2019). *Implementing A* Pathfinding in Java | Baeldung* (G. Piwowarek, Ed.). Www.baeldung.com. https://www.baeldung.com/java-a-star-pathfinding. Consultado el 15 de febrero de 2025.
- Pittman, J. (16 de junio, 2011). *The Pac-Man Dossier*. Archive.org. https://web.archive.org/web/20151006023914/http://home.comcast.net/~jpittman2/pacman/pacmandossier.html. Consultado el 17 de febrero de 2025.