TEC | Tecnológico de Costa Rica

BusCEMinas

Integrantes:
Tamara Cajiao Molina - 2024143333
Nicolás Florez Jiménez - 2024086367
David Garcia Cruz - 2024115575
Fabiola Melendez Sequeira -2023205341

Descripción de las funciones implementadas

Se implementaron diversas funciones las cuales corresponden cada una partes importantes del proyecto.

Funciones de generación del mapa

generate_map: Función principal de generación que coordina todo el proceso de creación del tablero. Recibe las dimensiones del tablero, nivel de dificultad y coordenadas del primer click para generar un mapa seguro. Retorna una matriz completamente configurada con minas y números adyacentes.

matrix: Función que crea una matriz base de dimensiones específicas inicializada con valores por defecto. Sirve como estructura inicial para el tablero del juego.

safe_bomb_placement: Implementa el algoritmo de colocación segura de minas evitando la posición del primer click del jugador. Calcula el porcentaje de minas según la dificultad y las distribuye aleatoriamente garantizando que el primer click nunca sea una mina.

fill_all_numbers: Función que calcula y asigna los números de minas adyacentes para cada celda del tablero. Itera sobre toda la matriz y para cada celda no-mina cuenta las minas en las 8 posiciones adyacentes.

Funciones de gestión de estado

get-game-config y set-game-config: Funciones para acceder y modificar la configuración global del juego incluyendo dimensiones del tablero y nivel de dificultad. Mantienen la coherencia del estado entre diferentes pantallas.

get-game-map y set-game-map: Funciones de acceso al mapa de juego activo. Permiten almacenar y recuperar la matriz del juego actual manteniendo la persistencia entre interacciones.

is-first-click y set-first-click: Controlan el estado del primer click del jugador, necesario para la generación segura del mapa. Aseguran que el mapa se genere solo después del primer click evitando minas en esa posición.

reset-game-state: Función de limpieza que reinicia todas las variables de estado del juego a sus valores por defecto. Ejecutada al iniciar nuevas partidas o cambiar configuraciones.

Funciones de manejo de celdas

get-cell-revealed y set-cell-revealed: Administran el estado de revelación de cada celda individual. Mantienen una matriz paralela que indica qué celdas han sido descubiertas por el jugador.

get-cell-flagged y set-cell-flagged: Controlan el sistema de banderas permitiendo al jugador marcar celdas sospechosas. Gestionan una matriz separada para el estado de las banderas.

reveal-cell: Función central que maneja la lógica de revelación de celdas incluyendo la expansión automática para áreas vacías. Implementa el algoritmo de propagación que revela automáticamente celdas adyacentes cuando se encuentra un área sin minas.

Funciones de utilidad

create-game-map: Función de interfaz que conecta la lógica funcional con el sistema de estados. Coordina la generación del mapa y su almacenamiento en el estado global.

difficulty-to-number: Convierte representaciones textuales o simbólicas de dificultad a valores numéricos utilizados por los algoritmos de generación de minas.

init-game-state: Inicializa las estructuras de datos auxiliares necesarias para el tracking del estado de cada celda durante la partida.

Funciones de actualización visual

update-button-image: Actualiza la representación gráfica de cada celda según su estado. Mapea estados lógicos a assets visuales correspondientes.

get-cell-asset-path: Función de mapeo que determina qué imagen mostrar para cada celda basándose en su valor y estado de revelación. Centraliza la lógica de presentación visual.

Ejemplificación de las estructuras de datos desarrolladas

Para poder ejemplificar primero es necesario entender el funcionamiento de la estructura de datos, para este caso se trabaja con 3 matrices principales, la estructura principal es la matriz que maneja la lógica o matriz "real", la segunda es la matriz de celdas reveladas la cual maneja verdaderos y falsos según lo que el jugador vaya descubriendo en el mapa y por último la matriz de banderas, la cual indica dónde el jugador ha colocado banderas. Estas se manejan de la siguiente manera, la matriz "real" que es la que contiene todos los datos de las minas y los números, por encima está la matriz de celdas y por encima de estas dos está la de banderas, si yo toco una celda esta pasa a mostrar lo que hay en la matriz real, puede ser un número o una mina y si el jugador presiona una celda en el modo bandera, es igual a poner true sobre la matriz de banderas por lo tanto muestra la bandera encima de la celda.

El siguiente ejemplo es de una matriz 4x4
Esta es la matriz principal muestra una matriz de listas donde cada elemento es un número (minas adyacentes) o una X (mina).

```
(
  (X 2 1 1)
  (2 3 X 1)
  (1 X 2 1)
  (1 1 1 0)
)
```

Después se tiene la matriz de celdas reveladas

```
(
  (#f #f #f #f)
  (#f #t #f #t)
  (#f #f #t #f)
  (#t #t #t #f)
)
```

Esta matriz representa #t como una celda que el jugador ha revelado y #f si la celda está oculta aun.

Por último, está la matriz de banderas

```
(
  (#t #f #f #f)
  (#f #f #t #f)
  (#f #t #f #f)
  (#f #f #f #f)
)
```

En esta matriz se tiene #t para las posiciones en las que el jugador coloca una bandera y #f para donde no hay ninguna bandera, note que el jugador puso las banderas en las posiciones correspondientes a las minas de la matriz principal.

Estas tres matrices corresponden a las estructuras principales del juego, cabe también mencionar la estructura que se utiliza para configurar el juego la cual es la siguiente;

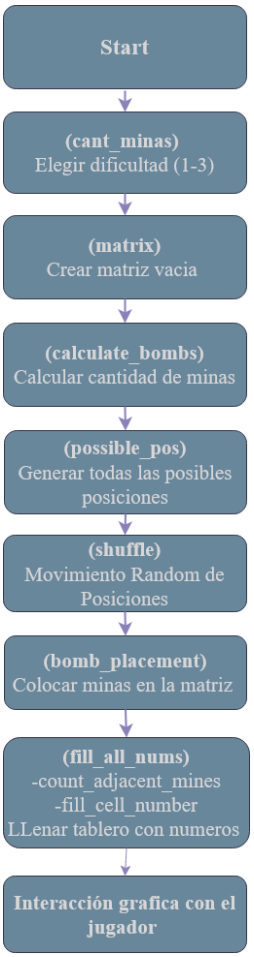
Estructura game-vars:

```
(game-vars 8 8 2)
```

el primer elemento (8) indica el número de filas, el segundo elemento (8) indica el número de columnas y el tercero (2) indica la dificultad elegida (1 = fácil, 2 = medio, 3 = difícil).

Descripción detallada de los algoritmos desarrollados

Se detallan los algoritmos principales que conforman la logica del juego:



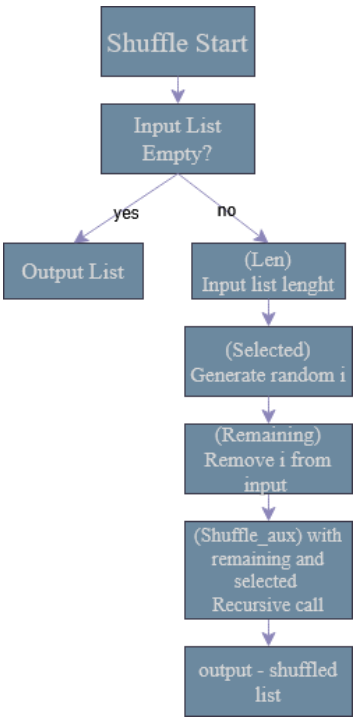
1. matrix (rows columns): Genera una matriz de un tamaño establecido como entrada (rows x columns), se establece como una lista de listas.

2. replace_in_matrix (matrix row col value): Reemplazo de un valor en un índice, ambos establecidos como entradas. Se realiza con el objetivo de poder establecer las bombas en ciertas posiciones.

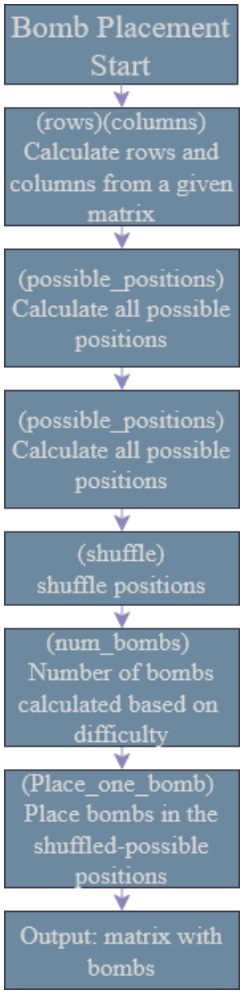
3. calculate_bombs (dificultad rows cols): Cálculo de bombas en un tablero según la dificultad y tamaño de una matriz. Fundamental para poder colocar las bombas

4. possible_positions (rows cols): Generar toda las posibles posiciones de la matriz como pares de coordenadas para poder establecer en cuáles posiciones se debe colocar una bomba.

5. shuffle (list): Algoritmo para mezclar aleatoriamente una lista de manera recursiva. Este algoritmo permite poder cambiar el orden de las bombas dependiendo de cada ejecución



6. bomb_placement (dificultad matrix): Algoritmo para colocar las bombas en la matriz segun una dificultad establecida.



7. count_adjacent_mines: Calcula cuentas minas rodean a una celda en sus 8 posiciones vecinas.

8. fill_cell_number: Coloca en una celda el numero de minas adyacentes y realiza las verificaciones necesarias. Transforma los valores de 0 establecidos inicialmente y los transforma a los valores reales por visualizar.

9. fill_all_numbers: Recorre la matriz completa y aplica fill_cell_number a cada celda para convertir la combinacion de bombas en un tablero listo para el juego.

Plan de actividades

Actividad	Descripción	Tiempo estimado de ejecución	Fecha de entrega	Responsable
Revisión de requisitos	Se analizarán las instrucciones de la tarea, se dividirán entre sub-tareas y se repartirán entre los cuatro integrantes del grupo	1 hora	20/09/25	Todos
Búsqueda de Información y familiarización con el lenguaje Racket	Investigar teoría, ejemplos y/o documentación necesaria para comenzar a desarrollar la tarea	4 horas	22/09/25	Todos
Diseño de la solución	Sugerir posibles soluciones para la elaboración (creación inicial y contenido aleatorio) del mapa del juego	2 horas	23/09/25	David y Fabiola
Desarrollo de la Interfaz Gráfica	Elaborar la interfaz gráfica con paneles, canvases, botones y controles fundamentales para jugabilidad	2 horas	24/09/25	Tamara
Definición de funciones base	Crear funciones esenciales,	2 horas	24/09/25	Nicolás

	reutilizables que permitan acceder y modificar a los valores del mapa del juego.			
Desarrollo de creación y manejo del mapa	Implementar la creación del mapa de acuerdo a la solución acordada en reuniones anteriores	3 horas	25/09/25	David y Fabiola
Integración con la Interfaz	Conectar la interfaz gráfica con las funciones lógicas del mapa de juego	2.5 horas	25/09/25	David
Pruebas, correcciones y revisión final	Ejecutar pruebas, identificar errores/gaps y aplicar correcciones/modularización codebase y producto final.	3 horas	26/09/25	Nicolás
Finiquitar Documentación	Redactar un breve informe sobre el trabajo realizado, justificación de decisiones, problemas encontrados, resultados y conclusiones	4 horas	27/09/25	Todos

Problemas encontrados

Uso de la librería gráfica de Racket (GUI)

El uso de la librería gráfica de Racket representó un reto debido a su sintaxis particular y a las limitaciones en personalización. A diferencia de frameworks modernos, la librería no provee un sistema extenso de estilos o layouts.

Ejemplo concreto: al dimensionar botones y paneles, no existen propiedades directas de flexibilidad (como flex o grid en otros lenguajes), por lo que fue necesario adaptar manualmente posiciones y tamaños.

Generación y manejo del mapa

La lógica de generación del tablero requirió pensar en términos funcionales, lo cual contrasta con la percepción visual mutable del juego.

Se implementaron tres matrices principales: mapa real (minas y números), matriz de revelación y matriz de banderas.

Problema objetivo: aunque gráficamente una celda revelada puede "cambiar de estado", internamente la estructura de datos original es inmutable. Para resolverlo, se desarrollaron funciones como replace_in_matrix y algoritmos recursivos que devuelven nuevas versiones de las matrices en cada modificación.

La sincronización visual se logró mediante la función get-cell-asset-path, que define prioridades lógicas (bandera > celda oculta > número/mina).

Manejo de estados del juego

El control de estados como primer click seguro, partida en curso, derrota o victoria resultó complejo debido a la rigidez del paradigma funcional.

Problema objetivo: mantener la coherencia entre múltiples estructuras (mapa, reveladas, banderas) sin recurrir a mutaciones directas provocaba desincronizaciones si no se propagaban los cambios correctamente.

La solución fue centralizar las variables en state.rkt mediante funciones de acceso (get y set), asegurando consistencia en la manipulación del estado global.

Inserción y gestión de imágenes

Otro punto crítico fue la integración de imágenes (tiles, minas, números, banderas).

Problema objetivo: la librería GUI de Racket no ofrece soporte nativo robusto para escalado automático o posicionamiento dinámico de imágenes.

Fue necesario estandarizar los recursos gráficos en tamaños fijos y controlar manualmente el refresco de cada celda tras cambios en el estado del juego. Esto evitó problemas de desalineación y de actualización parcial en pantalla.

Problemas sin solución

Click derecho para banderas

No se logró implementar el uso del click derecho para colocar banderas, posiblemente debido a limitaciones en el manejo de eventos de la librería GUI de Racket, que no proporciona un soporte nativo sencillo y consistente para distinguir botones del mouse en todas las plataformas. Se decidió en su lugar utilizar un toggle button para alternar entre revelar celdas y colocar banderas.

Modularización incompleta

Parte de la lógica del juego sigue centralizada en archivos como game.rkt, events.rkt o bomb.rkt, lo que genera clutter y dificulta la reutilización del código. Anteriormente, este problema era más severo y afectaba prácticamente a todos los archivos, pero actualmente la modularización es más limpia y con intención, y el código es más legible y manejable.

Aun así, la estructura sigue presentando cierta complejidad que podría dificultar la escalabilidad y el mantenimiento en versiones futuras.

Limitaciones en la UI y estilización

Aunque la interfaz es simétrica y funcional, no se logró implementar un comportamiento completamente responsivo ni mejoras estéticas adicionales. Ajustes como adaptar automáticamente los tiles al tamaño de la ventana, utilizar componentes más atractivos, implementar diferentes tipografías o mejorar el estilo visual general no fueron abordados, principalmente por falta de tiempo y por las restricciones de la librería GUI de Racket. La interfaz funciona correctamente, pero existe margen para mejorar la experiencia de usuario y la presentación visual.

Conclusiones

A lo largo de este documento se pudo observar cada una de las implementaciones y soluciones a cada uno de los problemas y objetivos planteados. Las nuevas herramientas y la complejidad de cada una de ellas fue sin lugar a duda un desafío para cada uno de los estudiantes, pero, se puede asegurar según los resultados observados que cada uno de los objetivos fue cumplido en su totalidad.
Los estudiantes aplicaron de manera efectiva cada uno de las prácticas estudiadas, el uso correcto de inputs, la integración de una interfaz para visualización de parte del usuario, el manejo correcto de estructuras de datos e implementación de conceptos de lenguaje funcional como lo es Racket. A través de la implementación de las distintas estrategias de solución, no sólo fortalecieron sus habilidades técnicas, sino que también desarrollaron un enfoque analítico para la resolución de problemas. Los resultados obtenidos evidencian que cada uno de los objetivos planteados fue alcanzado de manera satisfactoria, lo que demuestra el progreso y la adquisición de conocimientos esenciales en el área.
En conclusión, esta segunda tarea no sólo permitió la familiarización del estudiante con el lenguaje como tal, sino que también impulsó el desarrollo de habilidades clave como la resolución de problemas, la adaptación a nuevas herramientas y la gestión eficiente del tiempo; aspectos fundamentales para el éxito en proyectos futuros dentro del campo de la ingeniería.

Recomendaciones

Tras el análisis realizado y los hallazgos obtenidos, se considera pertinente sugerir lo siguiente:

• Comprensión del objetivo y funcionamiento de las partes: Antes de empezar a trabajar, es recomendable comprender en profundidad el propósito, la integración y las estructuras base que conforman el lenguaje de Racket, así como conceptos de lenguajes funcionales y estructuras de datos, con el fin de minimizar errores y favorecer un aprendizaje más efectivo.

• Trabajo en equipo: Se recomienda fomentar la comunicación efectiva y clara entre los miembros del equipo, además del apoyo conjunto del grupo, con el objetivo de alcanzar de manera asertiva cada uno de los objetivos propuestos.

• Documentación del trabajo: Dado que las tareas se encuentran divididas entre los miembros del equipo, resulta esencial que cada integrante documente de manera clara y organizada sus avances, hallazgos, dificultades y soluciones implementadas. Lo anterior no solo facilita la comunicación y el aprendizaje colectivo, sino que también garantiza una integración más fluida y eficiente de todas las partes del proyecto.

Bibliografía

[1] J. E. Helo Guzmán, *Introducción a la programación con Scheme*. Cartago, Costa Rica: Editorial Tecnológica de Costa Rica, 2000.

[2] “Buscaminas – reglas,” Buscaminas.eu [En línea]. Disponible: https://buscaminas.eu/reglas

[3] Problemas y Desafíos Matemáticos (PyDM), “Aprende a jugar al Buscaminas. Juego de Lógica gratuito,” *YouTube*, 1 mar. 2024. [Vídeo en línea]. Disponible: https://www.youtube.com/watch?v=PqA7yQcAZVc