Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2023-2)

Tarea 2

Entrega

Esta evaluación cuenta con dos entregas:

- **Entrega intermedia**
  - **Fecha y hora:** lunes 23 de octubre de 2023, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T2/entrega_intermedia`
  - **Corrección:** Mediante únicamente uso de **tests.**
- **Entrega final y** `README.md`
  - **Fecha y hora oficial (sin atraso):** lunes 6 de noviembre de 2023, 20:00
  - **Fecha y hora máxima (2 días de atraso):** miércoles 8 de noviembre de 2023, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T2/entrega_final`
  - **Corrección: Manual** por parte del equipo docente.
- **Ejecución de tarea**: Durante el proceso de corrección, se cambiará el nombre de la carpeta "T2/" por otro nombre y se ubicará la terminal dentro de esta carpeta antes de ejecutar la tarea. **Los *paths* relativos utilizados en la tarea deben ser coherentes con esta instrucción.**

Objetivos

- Utilizar conceptos de interfaces y PyQt6 para implementar una aplicación gráfica e interactiva.
- Traspasar decisiones de diseño y modelación en base a un documento de requisitos.
- Diseñar e implementar una arquitectura cliente-servidor. En el cliente, se debe entender y aplicar los conceptos de back-end y front-end.
- Aplicar conocimientos de *threading* en interfaces (Qtimer y/o QThread).
- Aplicar conocimientos de señales.
- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores. *(networking)*
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCConejoChico*

Gatochico original (de nuestra dimensión), quedó estupefacta al darse cuenta que una de sus tantas versiones del multiverso de las gatochico era una entusiasta del ajedrez. Esto fue chocante pues su *evento canónico* es quedar traumada por este juego. Es por esto que decide, en un acto osado y contra los tabúes de nuestra dimensión, construir una máquina para poder viajar interdimensionalmente.

Para su mala suerte, las reglas de nuestra dimensión no pueden romperse, por lo que al momento de activar su máquina interdimensional las coordenadas se corrompieron y terminaron trasladando su consciencia hasta **Conejochico**, una de las variantes más bizarras de **Gatochico**, pues es un conejo cautivo por **Pacalein**, quien la privó de su libertad para mantenerla como mascota, encerrada en un laberinto de alta complejidad para que nunca pueda escapar.

Es por esto que ahora **Gatochico** en el cuerpo de **Conejochico**, te manda mensajes por telepatía para pedirte que la ayudes a salir de entre las garras de **Pacalein**, y de esa forma, recuperar su libertad, mediante una simulación del laberinto donde se encuentra encerrada, *DCConejoChico*.

Figura 1: Logo de *DCConejoChico*

2. Flujo del programa

DCConejoChico es un juego que tiene como objetivo principal atravesar diferentes niveles de laberintos, con el propósito de rescatar a **Gatochico** de las garras de **Pacalein**. En esta misión, el jugador deberá esquivar cañones de zanahoria y derrotar a los lobos que colaboran con **Pacalein** para mantener a **Conejochico** prisionera, con el fin de completar los laberintos y alcanzar la liberación de **Gatochico** en la mente de **Conejochico**. Para cumplir esta misión, se deberá crear una interfaz gráfica que permita controlar a **Conejochico** mediante las teclas **WASD** y, además, interactuar con ciertos objetos del juego mediante *clicks*, como por ejemplo, los objetos coleccionables (Items).

El programa debe contar tanto con una interfaz gráfica como con un servidor funcional que permita almacenar partidas (nivel y puntaje), registrar los puntajes y validar que no se cuelen usuarios bloqueados del juego. Lo primero que debe ejecutarse en la tarea es el servidor, seguido del cliente. El jugador debe ser una instancia de la clase del cliente, la cual está encargada de la interfaz gráfica del jugador y será el único medio de comunicación con el servidor del programa.

Al iniciar el programa, aparecerá la Ventana de Inicio, la cual debe tener el logo del programa, un *input* de texto para que el cliente ingrese su usuario, un botón para comenzar el juego, otro botón para salir del programa y también un "Salón de la Fama", que exhibirá los mejores puntajes registrados en el servidor.

Para dar inicio al juego se deberá ingresar el nombre de usuario, el cual debe ser **alfanumérico, de longitud acotada, con al menos una mayúscula, un número y no vacío (largo entre 3 y 16, inclusive)**. Al presionar el botón para comenzar el juego, debe validarse en el servidor que el usuario no esté en la lista de usuarios bloqueados. En caso de pasar la validación, se cierra la ventana actual y se abre la Ventana de Juego, en donde se muestra un laberinto preconstruido, estadísticas y elementos mínimos que deberá contener el mapa. Además, durante el juego, el usuario tendrá una cantidad limitada de vidas que se mostrarán en la interfaz. Estas vidas podrán disminuir si el jugador colisiona con obstáculos o

enemigos dentro del laberinto. Cada vez que el jugador pierda una vida, se reflejará en las estadísticas en pantalla. También se tendrá a disposición un inventario al que se incorporan los objetos recogidos en el camino, el cual se visualiza en la Ventana de Juego. En caso de no pasar la validación, se deberá avisar, en la ventana, que el nombre ingresado no pasa las validaciones.

Una vez comenzado el nivel, se interponen lobos y cañones de zanahorias a lo largo de todo el camino, los cuales dificultan el paso y deben evitarse para atravesar el laberinto con éxito. En el camino, pueden recogerse items que deberás almacenar en el inventario y se utilizan a elección del jugador para enfrentar a los enemigos. El nivel se acaba cuando el jugador alcanza la meta, o agota sus vidas.

En caso de alcanzar la meta de manera exitosa, se da inicio al siguiente nivel, de lo contrario, se indica la derrota y se cierra el programa. Al completar el último nivel se indica la victoria y el puntaje final. Si se sale del programa en el transcurso de un nivel, se guarda la información del puntaje de la partida en el servidor. Es importante destacar que los niveles se extraerán a partir de archivos con la información necesaria para cargar cada nivel, y se deberá realizar **sin ocupar librerías no permitidas, ni ir contra alguna otra indicación.**

3. Mecánicas del Juego

3.1. Laberinto

El formato del tablero consistirá en un archivo compuesto por 16 líneas, de 16 columnas cada una. En este tablero, las diferentes entidades se representarán con una simbología específica, la cual se detalla en la sección laberintos donde también se encuentra información adicional sobre los tableros.

Cada laberinto tiene una entrada, e inicialmente **Conejochico** aparece en una casilla aledaña a esta, pero en caso de reiniciarse el nivel por pérdida de una vida, **Conejochico debe aparecer sobre la entrada del laberinto.**

3.2. Niveles y dificultad

El juego consta de 3 niveles. Al iniciar el juego, el nivel tendrá una duración de `DURACION_NIVEL_INICIAL` [1] segundos y los lobos tendrán una velocidad de `VELOCIDAD_LOBO_INICIAL`. Cada vez que se avance de nivel, el tiempo del nivel disminuirá y la velocidad de los lobos aumentará. Se tendrá un `PONDERADOR_LABERINTO` con un valor entre 0.8 y 0.9 que afectará a la velocidad de los lobos y el tiempo del nivel:

$$duracion\_nivel = duracion\_nivel\_anterior \times \texttt{PONDERADOR\_LABERINTO}^{2} \tag{1}$$

$$velocidad\_lobo = \frac{velocidad\_lobo\_nivel\_anterior}{\texttt{PONDERADOR\_LABERINTO}}^{3} \tag{2}$$

Cabe destacar que, en caso de que un usuario **antiguo** quiera jugar de nuevo, se debe buscar el último nivel jugado y cargar la partida desde ahí. Esto significa que si **Lily416** juega completa el segundo nivel un día, pero no completa el tercero, al volver a jugar debe cargarse el laberinto del tercer nivel. En caso de que el usuario haya completado el tercer nivel y quiera volver a jugar, comienza desde el primer nivel.

[1] Todo lo que esté en `ESTE_FORMATO` se considerará un parámetro. Esto se explica en detalle en las sección de `parametros.py`.

[2] Por ejemplo, para calcular la duración que tendrá el nivel 2, se deberá multiplicar la duración que haya tenido el nivel 1 con el parámetro `PONDERADOR_LABERINTO`.

[3] Por ejemplo, para calcular la velocidad del lobo en el nivel 2, se deberá dividir la velocidad del lobo en el nivel 1 por el parámetro `PONDERADOR_LABERINTO`.

3.3. Puntaje

Al comenzar el nivel se iniciará una cuenta regresiva del tiempo correspondiente al nivel en segundos, según lo expuesto en Niveles y dificultad. Si esta cuenta regresiva llega a 0 segundos, se pierde una vida y el nivel se reinicia, permitiendo que **Conejochico** comience desde el inicio del nivel. Lo mismo ocurre si **Conejochico** colisiona con una zanahoria o un lobo. Eliminar lobos otorgará una cantidad `PUNTAJE_LOBO` de puntos.

En caso de completar el nivel con éxito, el puntaje se calculará en función del tiempo, en segundos (`int`), vidas restantes (`int`), y la cantidad de lobos eliminados en el nivel (`int`). Para el puntaje final se debe considerar todo lobo eliminado durante el nivel, esto incluye aquellos que fueron eliminados antes de un reinicio debido a la pérdida de una vida. La fórmula para calcular el puntaje del nivel es la siguiente, **siempre redondeada al segundo decimal**:

$$puntaje\_nivel = \frac{tiempo\_restante \times vidas\_restantes}{cantidad\_lobos\_eliminados \times \texttt{PUNTAJE\_LOBO}}^{4}. \tag{3}$$

En caso de no haber eliminado ningún lobo en el nivel, el puntaje de este será 0.

Cada vez que se termine un nivel se deberá enviar el nombre de usuario, nivel completado y puntaje actual al servidor, y este se guardará un archivo llamado `puntaje.txt`. Cabe destacar que en este archivo debe existir **una única línea** por cada usuario que juegue, actualizando tanto el puntaje acumulado como el último nivel completado. Esto quiere decir, independiente de comenzar una nueva partida, los puntajes siempre se suman al valor ya guardado, por ejemplo, si **Gewwy0325** jugó un día y logró 1000 puntos, su línea querdaría como `Gewwy0325,1,1000`. Si al siguiente día juega y hace 500 puntos al completar el nivel 2, su línea se actualizaría a `Gewwy0325,2,1500`.

3.4. Fin del nivel

Una vez que se llega al otro extremo del laberinto, el nivel finaliza se envía la información al servidor del nivel logrado, que corresponde al usuario, nivel recién finalizado y el puntaje de dicho nivel. Luego, da comienzo el siguiente nivel.

En caso de continuar inmediatamente desde un nivel anterior en la misma partida, se conserva la cantidad de vidas e inventario que se tenía al finalizar el nivel anterior. En otro caso, si se retoma una partida previamente existente, se comience desde el principio del nivel que corresponda, según lo último registrado en el juego, y dicho nivel comenzará **con el inventario vacío y una cantidad de vidas igual a** `max(1,CANTIDAD_VIDAS - nivel_a_jugar + 1)`.

En el caso de salir del programa a mitad del nivel, dado que este no fue completado exitosamente, no se enviará ninguna información de dicho nivel. En otras palabras, solo se enviará información del nivel logrado una vez se finaliza exitosamente dicho nivel.

3.5. Fin del juego

El juego llegará a su término cuando se terminen todas las vidas o cuando se llegue al fin del tercer nivel. Al terminar el juego se reproducirá un audio que indicará la victoria o derrota de la partida. Se deberá reproducir el audio `victoria.mp3` o el audio `derrota.mp3` en caso de victoria o derrota, mostrando el resultado en la ventana de Juego antes de que se cierre. Además, se mostrará un mensaje con el nombre del jugador y resultado del juego. Finalmente, habrá un botón de "salir" que permitirá cerrar el programa.

[4] Por ejemplo, al finalizar el nivel 2 con 8 segundos restantes y 2 vidas restantes, el numerador de la ecuación sería igual a 16. Si, en ese nivel, comencé con 3 vidas, se eliminó 1 lobo, luego se pierde una vida y se reinicia el nivel, y finalmente, al completarlo, se eliminan los 3 lobos presentes, el valor de *cantidad_lobos_eliminados* será igual a 4. Entonces, para calcular el denominador de la ecuación, multiplicaremos 4 por el parámetro `PUNTAJE_LOBO`

4. Entidades

Dentro de *DCConejoChico* existirán diversas entidades que intentarán frenar a toda costa a **Conejochico** de llegar a la meta de cada nivel del laberinto. Sin embargo, **Conejochico** es muy listo y podrá aprovechar distintos objetos e ítems dentro del laberinto para lograr su cometido.

4.1. Conejochico

Conejochico es el principal personaje del juego. Este puede ser controlado mediante el uso del teclado. Posee `CANTIDAD_VIDAS` vidas inicialmente, y estas se verán reducidas si colisiona [5] con un *Lobo* o con una *Zanahoria*. Si **Conejochico** es herido cuando no le quedan vidas, el juego terminará y se deberá notificar al usuario de la terrible noticia.

El movimiento de este personaje es limitado por las paredes del laberinto, y su tipo de movimiento es en una sola dirección hasta colisionar en una pared, es decir, cuando sea presionada una tecla de movimiento (solo una vez, no es necesario mantener presionada la tecla) deberá recorrer todas las casillas del camino sin posibilidad de quedarse detenido en medio. Su movimiento es gatillado por el uso de las teclas `WASD` donde la tecla `W` es moverse hacia arriba, `A` hacia la izquierda, `S` hacia abajo y `D` hacia la derecha. **Conejochico** se moverá a una velocidad de `VELOCIDAD_CONEJO`, entonces, el tiempo, en segundos, que se demora en avanzar hacia otra casilla se calcula mediante la siguiente fórmula:

$$tiempo\_movimiento\_conejo = \frac{1}{\texttt{VELOCIDAD\_CONEJO}} \tag{4}$$

Las animaciones de este personaje son continuas, de tal manera que se vean fluidas. Para esto el movimiento debe mostrarse como una animación correspondiente al sentido de su movimiento, con una cantidad de **3** ***sprites*** los cuales deben intercalarse entre sí para animar este movimiento entre casillas, explicado en la sección Movimiento de **Conejochico**.

4.2. Lobo

Los lobos son el enemigo principal en este juego, e intentarán acabar con **Conejochico** para que no salga del laberinto. Si **Conejochico** colisiona con un lobo y no posee vidas restantes, el juego terminará. En caso contrario, **Conejochico** perderá una vida y reaparecerá en el inicio del laberinto.

El movimiento de los lobos es entre casillas de manera continua con una cantidad de 3 *sprites* de la sección 5.1. Su movimiento es independiente del movimiento de **Conejochico**, y se moverá a una `VELOCIDAD_LOBO`, multiplicada por un `PONDERADOR_LABERINTO`, correspondiente a cada nivel del laberinto.

Existen dos tipos de lobos, el vertical y horizontal. Cada uno se mueve en su dirección correspondiente, y al colisionar con una pared estos deberán cambiar su sentido de movimiento.

4.3. Cañón de zanahorias

Dentro de *DCConejoChico*, existen los cañones de zanahorias. Estos disparan zanahorias cada ciertos segundos dentro de un camino del laberinto, describiendo una trayectoria recta hasta chocar con una pared. Las zanahorias viajan a una velocidad de `VELOCIDAD_ZANAHORIA`, lo cual implica que se moverán cada ciertos segundos entre casillas, ocupando la siguiente fórmula:

$$tiempo\_movimiento\_zanahorias = \frac{1}{\texttt{VELOCIDAD\_ZANAHORIA}}$$

[5] Ocupa una casilla que está inmediatamente después, es decir, a su izquierda, derecha, arriba o abajo.

Si **Conejochico** es impactado por una zanahoria mientras posea vidas restantes, deberá perder una vida y reaparecer automáticamente al inicio del nivel, de lo contrario se deberá informar al cliente el fin del juego y cerrar el programa.

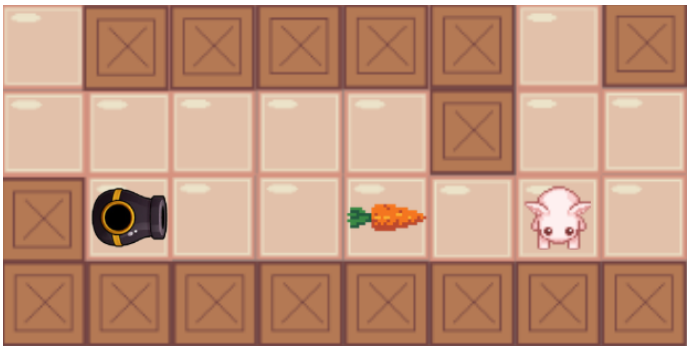


Figura 2: Ejemplo cañón

4.4. Items

Dentro de cada laberinto existirán distintos items que pueden ser usados por **Conejochico**. Cuando **Conejochico** se sitúe sobre un ítem, puede añadirlo a su inventario presionando la tecla `G` del teclado, o ignorarlo y pasar de largo. Estos items son **obtenibles** por **Conejochico**, por lo cual no lo afectan directamente.

4.4.1. Bombas de manzana

Uno de los items usables en *DCConejoChico* es la bomba de manzanas. Estas al ser colocadas por **Conejochico** son capaces de detonar todo lo que esté a su alcance, en las 4 direcciones posibles hasta chocar con una pared.

Cuando la bomba de manzana explote, deberá dejar en llamas todas las casillas de su alcance durante `TIEMPO_BOMBA` segundos, y cualquier lobo que cruce el fuego durante este intervalo de tiempo, deberá ser eliminado del laberinto.

4.4.2. Bombas de congelacion

De manera similar a las bombas de manzanas, el item de bomba de congelación es capaz de ralentizar el movimiento de cualquier lobo que sea afectado por la explosión de esta bomba. Estas bombas al ser colocadas por **Conejochico** podrán cubrir de hielo todas las casillas que estén a su alcance[6], en las 4 direcciones posibles hasta que la explosión alcance una pared.

Cuando ocurra la explosión, se aplicará un efecto de hielo durante `TIEMPO_BOMBA`. Para cada lobo que se sitúe en alguna casilla[7] afectada dentro del `TIEMPO_BOMBA`, verá reducida su velocidad en un 25 % durante todo el nivel. Si **Conejochico** es alcanzado por un lobo o una zanahoria, deberá reiniciarse el nivel, junto con los efectos de relentización aplicado en los lobos y descontando una vida.

[6] Para la animación, basta con cubrir las casillas con la sprite correspondiente (Congelación o Explosión, sección 8.1).
[7] Visualmente, se deberá superponer el lobo sobre la sprite correspondiente de congelación o explosión.

(a) Ejemplo bomba de manzana

(b) Ejemplo bomba de congelación

Figura 3: Ilustración de las bombas

5. Interacción

5.1. Movimiento de los lobos

Para el juego, se deberá simular el movimiento continuo de los lobos, tanto verticales como horizontales. Se entregaran distintos *sprites* (ver sección de Archivos) los cuales deberán intercalarse entre sí para lograr realizar la animación de estos.

(a) Parte 1

(b) Parte 2

(c) Parte 3

Figura 4: Animación de los lobos horizontales

(a) Parte 1

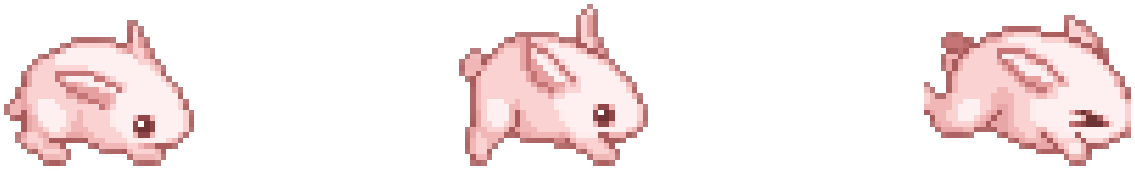
(b) Parte 2

(c) Parte 3

Figura 5: Animación de los lobos verticales

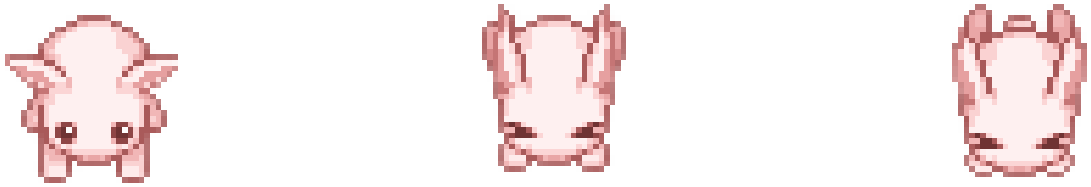
5.2. Movimiento de Conejochico

Al igual que para los lobos, el movimiento de **Conejochico** también deberá ser simulado por medio del uso intercalado de *sprites*, dependiendo de la dirección en que vaya el personaje. En el caso de tocar una zanahoria o algún lobo, deberá desaparecer y reaparecer en el punto de partida original, además de descontar una vida. En caso de estar quieto, no se deberá modelar ningún tipo de movimiento, vale decir, será un *sprite* estático.



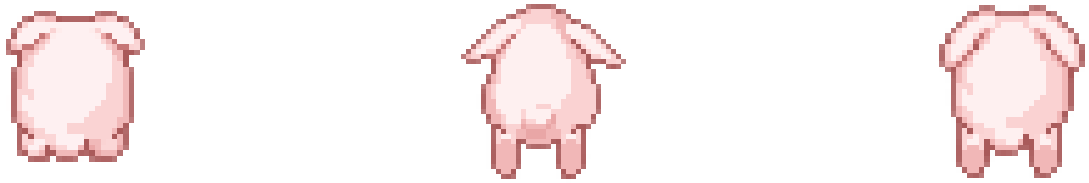
(a) Parte 1 (b) Parte 2 (c) Parte 3

Figura 6: Animación de la caminata horizontal de **Conejochico**



(a) Parte 1 (b) Parte 2 (c) Parte 3

Figura 7: Animación de la caminata hacia abajo de **Conejochico**



(a) Parte 1 (b) Parte 2 (c) Parte 3

Figura 8: Animación de la caminata hacia arriba de **Conejochico**

5.3. *Click*

La interacción con los botones del juego se deberá hacer por medio de *clicks* del cursor del computador. Además de los botones, podrás interactuar con otros objetos presentes en el juego:

5.3.1. Poner ítems en el mapa

Si se decide utilizar un ítem dentro del juego, se deberán seguir los siguientes pasos:

1. Hacer *click* sobre la entidad del panel de ítems que se desea agregar.
2. Hacer *click* sobre la casilla del mapa en donde se desea agregar el ítem.

3. Revisar si la casilla seleccionada es válida, vale decir, que no esté ocupada por otra entidad. En caso de ser inválida, se deberá notificar al jugador **en la misma ventana**, no mediante consola. [8]

5.4. Cheatcodes

Para esta tarea, y con el fin de ~~facilitar la corrección~~ mejorar la jugabilidad, se deberá poder presionar de manera consecutiva ciertas combinaciones de teclas durante la partida, una tecla tras otra, las cuales deberán llevar a cabo ciertas acciones:

- K + I + L: esta combinación de teclas elimina a todos los villanos del mapa.
- I + N + F: esta combinación de letras congela el contador de la partida, dando tiempo indefinido para terminarla. También otorga vidas infinitas a **Conejochico**, es decir, si **Conejochico** es herido, el nivel se resetea, pero no se restan vidas. En este caso, el puntaje de dicho nivel es por defecto `PUNTAJE_INF`.

5.5. Pausa

En la ventana del juego, deberá existir un botón de pausa para detener o reanudar el juego, el cual también deberá activarse pulsando la tecla **P**. Al pausar el juego, se deberá detener el contandor del tiempo transcurrido, además de cualquier otro tipo de movimiento, incluyendo a los villanos e imposibilitando al personaje **Conejochico** de moverse. En caso de volver a hacer *click* sobre el botón de pausa o volver a presionar la tecla P, el juego y todos sus elementos se deberán reanudar.

6. Interfaz gráfica

6.1. Modelación del programa

Se evaluará, entre otros, los siguientes aspectos:

- Correcta **modularización** del programa, lo que incluye una adecuada separación entre **back-end** y **front-end**, con un **diseño cohesivo** y de **bajo acoplamiento**.
- Correcto uso de **señales** y ***threading*** para modelar todas las interacciones en la interfaz.
- Presentación de la información y funcionalidades pedidas (puntajes o *cheatcodes*, por ejemplo) a través de la **interfaz gráfica**. Es decir, **no se evaluarán** *items* que solo puedan ser comprobados mediante la terminal o por código, a menos que la pauta lo explicite.
- Generación de las ventanas mediante código programado por el estudiante. Es decir, no se permite la creación de ventanas con el apoyo de herramientas como QtDesigner, entre otros.

6.2. Ventanas

Dado que *DCConejoChico* se compone de diversas etapas, se espera que se distribuyan en al menos **dos ventanas principales** que serán mencionadas a continuación, junto con los **elementos mínimos** que se solicitan en cada una. Los ejemplos de ventanas expuestos en esta sección son simplemente para que te hagas una idea de cómo se deberían ver y no es necesario que tu tarea se vea exactamente igual.

[8] Puedes hacerlo mediante un `QLabel`, o investigar sobre `QMessageBox`, entre otros *widgets* de PyQt6.

6.2.1. Ventana de Inicio

Es la primera ventana que se debe mostrar cuando el usuario ejecute el programa, la cual deberá contener como mínimo:

- El logo de *DCConejoChico*.
- Una línea de texto editable para ingresar el **nombre de usuario**.
- Un botón para comenzar la partida.
- Una sección para el "**Salon de la Fama**", que muestre el *ranking* de los mejores **puntajes** históricos guardados en el servidor.
- Un botón para salir del programa.

Cuando el usuario presione el botón para comenzar la partida, se deberá verificar antes que el nombre de usuario sea **alfanumérico**, contenga **al menos una mayúscula y un número, que no sea vacío (con un largo entre 3 y 16 caracteres)** y no se encuentre entre los usuarios **bloqueados**. Esta última verificación se debe llevar a cabo en el servidor, mientras que todo lo demás se verifica mediante el **back-end** del cliente.

En caso de no cumplir con alguna condición, **se deberá notificar al cliente** mediante un mensaje de error o *pop up* en la interfaz, mencionando el motivo del error. Si cumple todos los requerimientos para el nombre de usuario, se cerrará la ventana y automáticamente se abrirá la ventana de juego. Por último, si se selecciona la opción de salir, se deberá cerrar la ventana de inicio y cerrar el programa.

El "**Salón de la Fama**"deberá mostrar a los cinco mejores jugadores del desafiante *DCConejoChico*, incluyendo su nombre de usuario y puntaje obtenido. Los nombres de usuario y los puntajes deberán ser ordenados de manera descendente, es decir, el primer puntaje será el más alto y el último puntaje será el más bajo.

Figura 9: Ejemplo de ventana de Inicio

6.2.2. Ventana de Juego

En esta ventana se desarrolla el flujo del juego y debe estar compuesta por 3 secciones principales: **estado de la partida**, **inventario** y el **mapa**.

La primera de ellas deberá mostrar el **tiempo** restante del nivel, junto con las **vidas restantes** y el **puntaje actual**, además de dos botones, uno para **pausar** y otro para **salir** del programa.

La segunda sección es un recuadro que muestre todos los **items** disponibles en el **inventario del usuario**, representados por la *sprite* y cantidad respectiva del item, además **debe estar presente** un botón o *label* para usarlo. Estos items pueden ser utilizados en cualquier momento del juego, según se explicó en la seccion 5.3.

Por último, la tercera sección se compone del mapa del juego, donde se llevarán a cabo todas las mecánicas del juego, estando presentes los distintos elementos y entidades del juego. Siempre que el usuario posea vidas, quede tiempo restante y no haya terminado los 3 niveles, este podrá seguir jugando, de lo contrario se deberá mostrar un mensaje o *pop up* en la ventana indicando el término del juego y cerrando el programa junto con las ventanas.

A medida que el usuario complete los niveles, se mantendrá en la misma ventana y deberá cambiar el laberinto según la etapa correspondiente, junto con un nuevo tiempo restante. Una vez que el usuario finalice el juego se deberá mostrar un mensaje o *pop up* señalando su victoria y el puntaje final. [9]

Figura 10: Ejemplo de ventana de Juego

[9] El siguiente ejemplo es uno simplificado y demostrativo, en la implementación final las cajas deberían ser paredes grises, deberían haber obstáculos/villanos, y el inventario podría (o no) tener ítems.

7. Networking

Para poder validar los usuarios bloqueados y guardar tanto la partida de *DCConejoChico* como los puntajes de los jugadores, tendrás que usar todos tus conocimientos de *networking*. Deberás desarrollar una arquitectura `cliente` - `servidor` con el modelo **TCP/IP** haciendo uso del módulo **socket**.

Tu misión será implementar dos programas separados, uno para el **`servidor`** y otro para el **`cliente`**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar uno o más clientes. Debes tener en consideración que la comunicación es siempre entre cliente y servidor.

7.1. Arquitectura cliente-servidor

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando se cumpla con lo solicitado y no contradiga nada de lo indicado. Si algo no está especificado o si no queda completamente claro, puedes **preguntar aquí**.

7.1.1. Separación funcional

El cliente y el servidor deben estar separados. Esto implica que deben estar en directorios diferentes e independientes, uno llamado **`cliente`** y otro llamado **`servidor`**. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal **`main.py`**, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

```
T2
├── entrega_intermedia
│   ├── funciones_cliente.py
│   ├── funciones_servidor.py
│   ├── tests_todos.py
│   └── tests
│       └── ...
├── entrega_final
│   ├── servidor
│   │   ├── main.py
│   │   └── ...
│   └── cliente
│       ├── main.py
│       ├── frontend
│       │   └── ...
│       └── backend
│           └── ...
├── .gitignore
└── README.md
```

Si bien las carpetas asociadas al `cliente` y al `servidor` se ubican en el mismo directorio (T2), la ejecución del `cliente` **no debe depender de archivos en la carpeta del `servidor`**, y la ejecución del `servidor` **no debe depender de archivos y/o recursos en la carpeta del `cliente`**. Esto significa que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La Figura 13 muestra una representación esperada para los distintos componentes del programa:

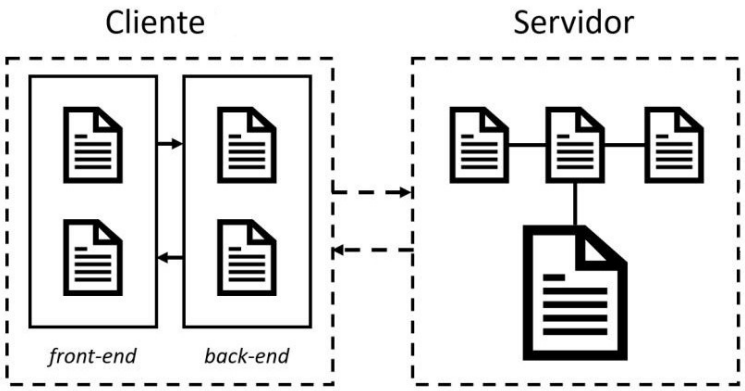


Figura 11: Separación cliente-servidor y *front-end*/*back-end*.

Cabe destacar que **solo el cliente tendrá una interfaz gráfica**. Por lo tanto, todo cliente debe contar con una separación entre *back-end* y *front-end*, mientras que la comunicación entre el `cliente` y el `servidor` debe realizarse mediante *sockets*. Ten en cuenta que la separación deberá ser de carácter **funcional**. Esto quiere decir que **toda tarea** encargada de **validar y verificar acciones** deberá ser **elaborada en el servidor**, mientras que el cliente deberá solamente recibir esta información y actualizar la interfaz.

7.1.2. Conexión

El `servidor` contará con un archivo de formato `JSON`, ubicado en la carpeta del `servidor`. Este archivo debe contener el host para instanciar un *socket*. El archivo debe llevar el siguiente formato:

```
{
    "host": <direccion_ip>,
    ...
}
```

Por otra parte, el `cliente` deberá conectarse al socket abierto por el `servidor` haciendo uso de los datos encontrados en el archivo `JSON` de la carpeta del `cliente`. La selección del puerto del *socket*, tanto para el `cliente` como para el `servidor`, se deberá hacer por medio de **argumento de consola**, pasando el puerto como argumento al ejecutar el archivo `.py`. Por ejemplo, si se quiere elegir el puerto 8000, se deberá ejecutar el archivo en consola de la siguiente forma: `python main.py 8000`. Es importante recalcar que el `cliente` y el `servidor` **no deben usar el mismo archivo** `JSON` para obtener los parámetros.

7.1.3. Método de codificación

Cuando se establece la conexión entre el **cliente** y el **servidor**, deberán encargarse de intercambiar información constantemente entre sí. Por ejemplo, el servidor deberá enviar la información de los *rankings* históricos al cliente, y el cliente deberá enviar el puntaje obtenido por el jugador al servidor para ser almacenado en este. Como podría ocurrir el caso de que un jugador intente hackear un mensaje para alterarlo, deberemos asegurarnos de encriptar el contenido antes de enviarlo. De este modo, si alguien logra interceptar un mensajes, no podrá decifrarlo. Luego de encriptar el contenido, deberás codificar los mensajes enviados entre el cliente y el servidor según la siguiente estructura:

- Los primeros 4 *bytes* indican el **largo del mensaje encriptado**, los cuales deben ser enviados en el formato *big endian*[10]. Este valor será útil para desencriptar el mensaje.
- Después, se debe separar el mensaje encriptado en *chunks* de 36 *bytes*, los cuales deben ser precedidos por otros 4 *bytes* que indiquen el número de bloque enviado partiendo desde el uno y codificados en *big endian*.
- Si para el último bloque no alcanza el mensaje para completar los *chunk* con 36 *bytes*, deberás llenar con bytes ceros (`b'\x00'`).

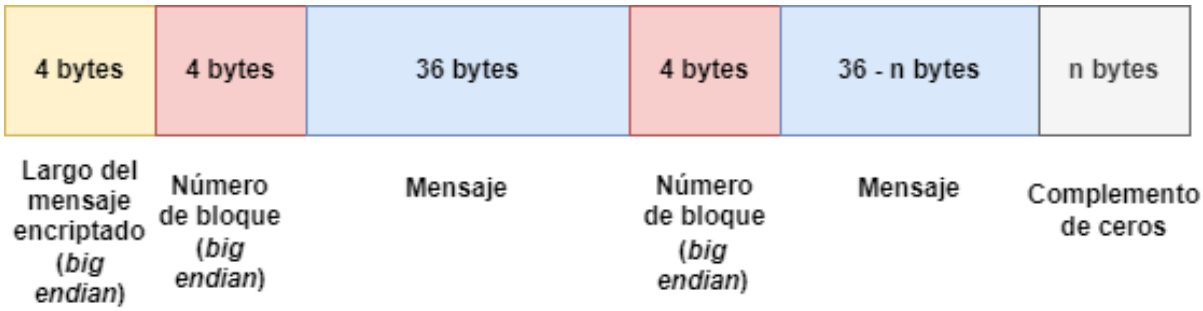


Figura 12: Ejemplo de un *bytearray* codificado.

7.1.4. Método de encriptación

Finalmente, el **método de encriptación** que deberás utilizar para esconder el contenido de tus mensajes y evitar *hackeos* es el siguiente:

- El mensaje deberá ser separado en **tres partes**. Se puede asumir que el mensaje tendrá como mínimo tres *bytes*. Cada parte será construida de la siguiente manera: se irán asignando los tres primeros *bytes* en orden desde la **primera** parte hasta la **tercera** parte, para luego asignar los tres *bytes* siguientes desde la **tercera** parte hasta la **primera** parte, alternando el orden de las partes (desde la primera parte a la tercera, o de la tercera parte a la primera) cada tres bytes asignados. El orden de asignación ilustrado sería algo como lo siguiente:

  1. Se agrega el primer *byte* a la **primera** parte.
  2. Se agrega el segundo *byte* a la **segunda** parte.
  3. Se agrega el tercer *byte* a la **tercera** parte.
  4. Se agrega el cuarto *byte* a la **tercera** parte.
  5. Se agrega el quinto *byte* a la **segunda** parte y así hasta agregar todos los *bytes* del mensaje.

  Llamaremos a la primera, segunda y tercera parte A, B y C, respectivamente. Sea X la secuencia de *bytes* a encriptar. La separación quedaría de la siguiente manera:

$$X = b_0\, b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, b_8\, b_9 \tag{5}$$

$$A = b_0\, b_5\, b_6 \tag{6}$$

$$B = b_1\, b_4\, b_7 \tag{7}$$

[10] El *endianness* es el orden en el que se guardan los *bytes* en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de *bytes* que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

$$C = b_2\, b_3\, b_8\, b_9 \tag{8}$$

Notar que A, B y C no tienen por que tener el mismo largo.

- Luego, se deberá sumar el primer *byte* de A, el último *byte* de B y el primer *byte* de C.
  - Si la suma de estos valores es un número par, el resultado encriptado será:

    nACB

    con n igual a 1 para indicar que se cumple esta condición.
  - Si la suma de estos valores es un número impar, el resultado encriptado será:

    nBAC

    con n igual a 0 para indicar que se cumple esta condición.

  Por último, para completar la comunicación segura entre el cliente y el servidor, deberás **diseñar una forma para desencriptar el mensaje** desde el *receptor* (`cliente` o `servidor`).

7.1.5. Ejemplo encriptación

Para ayudarte a entender de mejor forma el metodo de encriptación que debes implementar, utilizaremos como ejemplo el siguiente contenido de mensaje (previo a la encriptación):

b'\x05\x08\x03\x02\x04\x03\x05\x07\x05\x06\x01'

- Separamos el **bytearray** en las tres partes correspondientes:

  A = b'\x05\x03\x05'

  B = b'\x08\x04\x07\x01'

  C = b'\x03\x02\x05\x06'

- Luego, tomamos el primer *byte* de A, el último *byte* de B y el primer *byte* de C y los sumamos.

  $$5_A + 1_B + 3_C = 9$$

- Como el resultado es impar, el resultado de la encriptación sería el siguiente:

  nBAC

  b'\x00\x08\x04\x07\x01\x05\x03\x05\x03\x02\x05\x06'

7.1.6. Ejemplo de codificación

Para ayudarte a entender el método de codificación explicado anteriormente, supongamos que ser quiere enviar el mensaje encriptado que se mostró en el ejemplo anterior desde el `servidor` hacia el `cliente`:

- Según lo que se indicó, primero se enviaran 4 *bytes* indicando el largo del mensaje. Estos deben estar en formato *big endian*. En este caso, el largo del mensaje es de 12 *bytes*.
- Luego se separa el contenido del mensaje en bloques de 36 *bytes*. En este caso el largo del mensaje es de 12 *bytes*, por lo que tendremos que llenar el resto del bloque con *bytes* ceros hasta llenar el bloque.
- Para cada uno de los bloques de 36 *bytes* deberás enviar el número del bloque correspondiente en un mensaje de 4 *bytes* en formato *big endian*. Posterior a este mensaje se debe enviar el bloque de 36 *bytes* que contiene parte del mensaje.

- Una vez que envíes todos los bloques, el proceso de envío de mensaje habrá terminado. En este caso se envía un único bloque y luego finaliza el proceso.

En la siguiente figura se puede observar la estructura para enviar el mensaje ya encriptado en el ejemplo anterior:

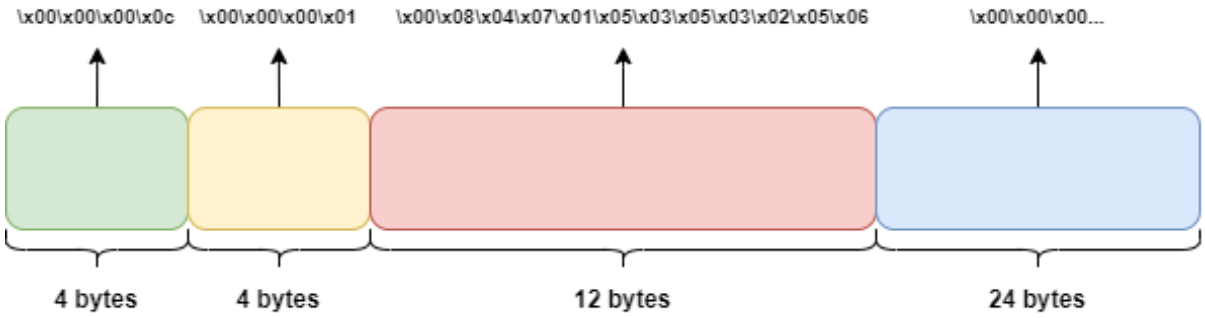


Figura 13: Ejemplo de la estructura del *bytearray* para un mensaje codificado.

7.1.7. *Logs* del servidor

Como el servidor no cuenta con interfaz gráfica, debes indicar constantemente lo que ocurre en él mediante la consola. Estos mensajes se conocen como mensajes de *log*, es decir, deberás llamar a la función `print` cuando ocurra un evento importante en el servidor. Estos mensajes se deben mostrar para cada uno de los siguientes eventos:

- Se conecta un cliente al servidor.
- Cuando el jugador pasa de nivel. Se debe indicar el puntaje obtenido en dicho nivel y el nombre del jugador.
- Cuando se finaliza la partida. Se debe indicar el puntaje acumulado por el jugador y su nombre.

Es de **libre elección** la forma en que representes los *logs* en consola. Un ejemplo de esto es el siguiente formato:

```
     Juegador    |        Evento          |      Detalles
-----------------|------------------------|---------------------------
  nombre_usuario |       Conectarse       |          -
-----------------|------------------------|---------------------------
  nombre_usuario |       Pasar nivel      |   Puntaje nivel: 13
-----------------|------------------------|---------------------------
  nombre_usuario |   Término de partida   |   Puntaje acumulado: 1113
-----------------|------------------------|---------------------------
```

7.1.8. Desconexión repentina

En caso de que algún cliente se desconecte, ya sea por error o a la fuerza, tu programa debe reaccionar para resolverlo:

- Si es el **`servidor`** quien se desconecta, cada cliente conectado debe mostrar un mensaje explicando la situación antes de cerrar el programa.
- Si es un **`cliente`** quien se desconecta, se descarta su conexión. Todo el avance en el nivel desde el último guardado hecho, se perderá.

7.2. Roles

A continuación, se detallan las funcionalidades que deben ser manejadas por el `servidor` y las que deben ser manejadas por el `cliente`:

7.2.1. Servidor

- **Procesar y validar** las acciones realizadas por los clientes. Por ejemplo, cuando el jugador ingresa su nombre de usuario, el servidor deberá verificar que dicho nombre no se encuentra en la lista de nombres bloqueados, para luego enviarle una respuesta al cliente según si este es válido o inválido.
- **Distribuir y actualizar** los puntajes históricos más altos en el salón de la fama, actualizando a medida que lleguen nuevos puntajes más altos.
- **Almacenar y actualizar** la información de cada cliente apenas se pase de nivel. La información a almacenar es el nombre de usuario, el último nivel que superó, y el puntaje acumulado hasta ese momento.

7.2.2. Cliente

Todos los clientes cuentan con una interfaz gráfica que le permitirá interactuar con el programa y enviar mensajes al servidor. Maneja las siguientes acciones:

- **Enviar la información de los niveles** completados por el usuario hacia el servidor.
- **Recibir e interpretar** las respuestas y actualizaciones que envía el servidor.
- **Actualizar** la interfaz gráfica de acuerdo a las respuestas que recibe del servidor.

8. Archivos

Para el desarrollo de la tarea, deberás hacer uso de los siguientes archivos entregados en la carpeta `assets`:

- *sprites*: Corresponden a los elementos visuales que se encuentran en la carpeta `sprites`.
- sonidos: Corresponde a los sonidos utilizados al ganar y perder el juego, explicados en Fin del nivel. Estos archivos están incluidos en la carpeta `Sonidos`.
- laberintos: Corresponde a los laberintos que deberán cargar en los distintos niveles. Estos archivos están incluidos en la carpeta `laberintos`.

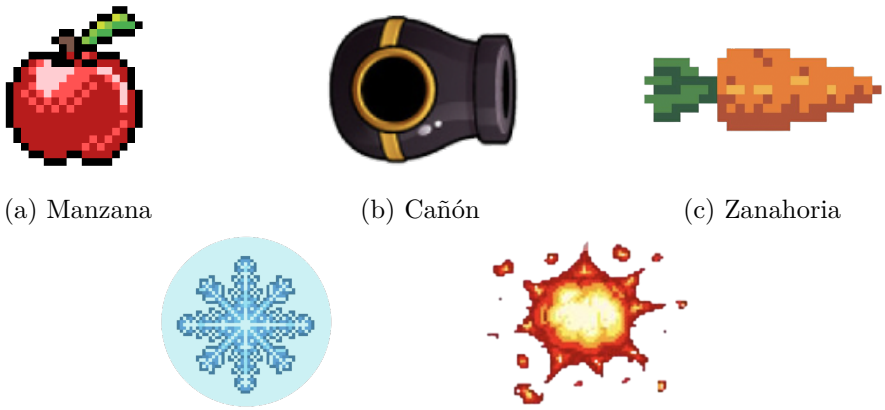
Adicionalmente, tu programa debe ser capaz de leer los siguientes archivos:

- `parametros.py`: En esta tarea se entregará este archivo con los parámetros en `ESTE_FORMATO` señalados a lo largo del enunciado para que uses en tu tarea. Además, puedes agregar cualquier otro que veas pertinente añadir, como los `PATHS` o rutas de archivos.

8.1. *sprites*

Esta carpeta contiene todas las diferentes imágenes en formato `.png` que se ocuparán para tu tarea, entre ellos se encuentran:

- `Elementos`: los cuatro elementos necesarios para construir el mapa, además del logo del juego.



(a) Manzana (b) Cañón (c) Zanahoria

(d) Poder de Congelación (e) Explosión

Figura 14: Elementos del mapa.

- **`Personajes`**: los 3 personajes a utilizar para desarrollar la tarea.

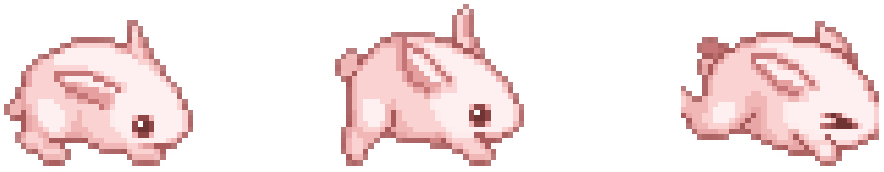


Figura 15: Caminata de **Conejochico**

Figura 16: Avance del Lobo horizontal

Figura 17: Avance del lobo vertical

8.2. sonidos

Esta carpeta contendrá los sonidos que pueden implementarse en el juego, todo detallado en la sección de Fin del nivel. Se encuentran `derrota.mp3` que se utilizará cuando el jugador pierda y `victoria.mp3` para cuando se gane el juego. Por otra parte, se deberá implementar `item.mp3` cuando se recolecte un ítem y `explosion.mp3` cuando se haga explotar una manzana.

8.3. laberintos

Esta carpeta contendrá los laberintos de los distintos niveles (fácil, intermedio, difícil) para jugar *DCConejoChico*. El formato de cada laberinto será una secuencia de 16 elementos por línea separados por comas y en total serán 16 líneas, dando un total de 16x16. Hay que recalcar que las líneas exteriores del mapa son por completo paredes, a excepción de la entrada y la salida. En caso de que **Conejochico** pierda una vida, debe re-aparecer en la **entrada** del laberinto.

La simbología de cada entidad en el mapa es la siguiente:

- `"-"`: casilla vacía.
- `"C"`: casilla con **Conejochico**.
- `"P"`: casilla con pared.
- `"BM"`: casilla con bomba de manzana.
- `"BC"`: casilla con bomba de congelación.
- `"LH"`: casilla con lobo horizontal.
- `"LV"`: casilla con lobo vertical.
- `"CU"`: casilla con un cañon apuntando hacia arriba.
- `"CD"`: casilla con un cañon apuntando hacia abajo.
- `"CL"`: casilla con un cañon apuntando hacia la izquierda.
- `"CR"`: casilla con un cañon apuntando hacia la derecha.
- `"E"`: entrada del laberinto.
- `"S"`: salida del laberinto.

Un ejemplo (de dimensión reducida) de como se implementaría esto:

```
P,P,P,P,P,P,P,P,
E,-,-,C,P,P,P,P,
P,-,P,-,-,-,-,P,
P,-,-,-,-,P,-,P,
P,-,P,-,-,-,P,P,
P,-,-,-,P,-,-,P,
P,P,P,P,P,-,-,S,
P,P,P,P,P,P,P,P,
```

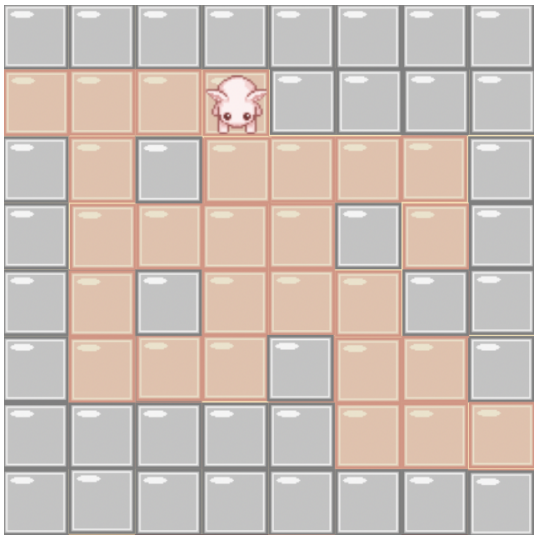


Figura 18: Ejemplo del mapa generado por el archivo anterior

8.4. `parametros.py`

En este archivo se encontrarán el ancho del laberinto como `ANCHO_LABERINTO` y el largo del laberinto como `LARGO_LABERINTO`, los cuales **no debes modificar**. Además, se encontrarán los demás parámetros mencionados anteriormente en el enunciado en `ESTE_FORMATO` para una corrección más estandarizada. Cada línea de este archivo almacena una constante con su respectivo valor, por lo que en tu tarea deberás **importar**[11] correctamente este archivo y utilizar los parámetros almacenados.

Además de incluir los parámetros descritos anteriormente, es importante incluir todo tipo de valor que será constante en tu programa, como los *paths* de archivos que se utilizarán y todo valor constante creado por ti. Es importante que los nombres de los parámetros sean declarativos, de lo contrario se caerá en una mala práctica.

Este archivo debe ser **importado como módulo** y así usar sus valores almacenados. Puedes colocar el archivo de parámetros en la posición que más te acomode en tu tarea, e incluso tener una copia para el cliente y otra para el servidor. En caso de que el enunciado no especifique un valor de algunos de los parámetros, deberás asignarlo a tu criterio, procurando que no dificulte la interacción con el juego y considerando que los ayudantes podrán modificarlo para poder corroborar que el juego está implementado de forma correcta.

Por otro lado, parámetros predeterminados como la ruta de los archivos o las dimensiones de la ventana y sus elementos nunca serán modificados, de tal manera que no interfiera con el funcionamiento del juego.

Es imperante que no ignores este archivo o tu tarea no podrá ser corregida.

9. Entregas

Esta evaluación consta de dos entregas:

Entrega 1 **Intermedia:** Completar los archivos `funciones_cliente.py` y `funciones_servidor.py` que contienen funciones que son beneficiosas para la implementación de *DCConejoChico*. Para la entrega intermedia, **ambos archivos deben estar dentro de un directorio llamado `entrega_intermedia`**. Además, se te entregará un conjunto de *tests*, los cuales estarán junto a los archivos a completar, y que validarán la correcta implementación de las funciones.
Esta parte será corregida automáticamente mediante el uso de *tests* privados, los cuales son distintos a los públicos.

Entrega 2 **Final:** Integrar los archivos `funciones_cliente.py` y `funciones_servidor.py` en la interfaz gráfica que creen, y completar el funcionamiento de *DCConejoChico*.
Esta parte será corregida únicamente por los ayudantes.

9.1. Entrega intermedia (25 %)

En esta parte deberás completar diversas funciones que te serán de provecho para el desarrollo de la tarea, tanto para las mecánicas de juego como para el uso de *networking*. La corrección completa de esta parte será mediante *tests*, es por esto que debes asegurarte que cada función se pueda ejecutar correctamente, y que el archivo no presente errores de sintaxis.

Para cada función indicada a continuación, **no puedes modificar su nombre, agregar nuevos argumentos o argumentos por defecto**. Puedes crear nuevas funciones y/o archivos si estimas conveniente, pero debes mantener el formato de las funciones entregadas. En caso de no respetar lo indicado, la evaluación presentará un fuerte descuento.

[11] Para mas información revisar el material de modularización de la semana 0.

Esta entrega no tiene política de atraso, ni de cambio de líneas, y los *test*s tienen un tiempo de ejecución acotado, en el que si superan ese tiempo, el test se considerará fallado. Cada archivo de *test* debe ejecutarse en **5 minutos máximo.**

A continuación se describen las 10 funciones que deberás completar.

Cabe destacar que el inventario **sólo por esta entrega** está conformado por los nombres de los ítemes que representan, para más adelante se espera que sean instancias de objetos.
Ej: `["manzana", "congelador", "congelador"]`

- Modificar `def validacion_formato(nombre: str) -> bool:`
  Esta función recibe el nombre de usuario y verifica que cumpla con las condiciones de ser alfanumérico, con al menos una mayúscula y un número. Además, verifica que el largo sea mayor o igual a 3, y menor o igual a 16. Retorna un bool señalando si el nombre de usuario es válido o no.

  Por ejemplo, `validacion_formato("Gatochico")` retorna `False` porque ese nombre no tiene un número.

- Modificar `def usuario_permitido(nombre: str, usuarios_no_permitidos: list) -> bool:`
  Esta función es parte del servidor, y recibe tanto el nombre de usuario como una lista de los usuarios no permitidos. Verifica que el nombre de usuario no esté entre los no permitidos, y retorna un bool `True` si el usuario es válido y `False` en caso de que no.

  Ejemplo, `usuario_permitido("Hernan444", ["Hernan444", "Ab1", "1aB", "DCCin1"])` retorna `False` porque está en la lista de usuarios no permitidos.

- Modificar `def riesgo_mortal(laberinto: list[list]) -> bool:`
  Recibe un laberinto con las posiciones de las entidades en ella, y se asume que **Conejochico no se mueve**. Retorna `True` si existe el riesgo de que **Conejochico** pierda una vida por colisionar con un lobo o una zanahoria, y `False` si no. Recordar la existencia de lobos verticales y horizontales así como zanahorias en distintas direcciones.

  El siguiente ejemplo es con un laberinto simplificado para mayor entendimiento. En caso de tener `riesgo_mortal([['-', '-', 'C'],['-', '-', '-'],['-', '-', 'LV']])`, la función debe retornar `True` pues **Conejochico** tiene el riesgo de morir (el lobo vertical podría llegar a su ubicación), mientras que `riesgo_mortal([['-', '-', 'C'],['-', '-', '-'],['-', '-', 'LH']])` retorna `False` por que **Conejochico** no corre riesgo de morir.

- Modificar `def usar_item(item: str, inventario: list) -> tuple(bool, list):`
  Recibe el nombre de un ítem y una lista de `str` donde cada elemento es el nombre de un objeto. Verifica si el nombre del item ingresado se encuentra en el inventario. En caso de que no haya, se retorna `False` junto a una copia de la lista original no modificada del inventario, y en caso de que sí, elimina la primera instancia que aparezca del ítem en el inventario y retorna una copia de la lista original modificada `True` junto a la lista actualizada del inventario.

  Ejemplo, `usar_item("manzana", ["congelador", "manzana", "congelador", "manzana"])` retorna `(True, ["congelador", "congelador", "manzana"])`

- Modificar `def calcular_puntaje(tiempo: int, vidas: int, cantidad_lobos: int, PUNTAJE_LOBO: int) -> float:`
  Retorna un `float` con el puntaje calculado con la fórmula del enunciado, redondeando el resultado a dos decimales.

  Ejemplo, para `calcular_puntaje(17, 2, 1, PUNTAJE_LOBO)` con `PUNTAJE_LOBO` = `3` retornaría `11.33`.

- Modificar `def validar_direccion(laberinto: list[list], tecla: str) -> bool:`
  Recibe un laberinto con la posición de **Conejochico** y una tecla de movimiento (`WASD`). Retorna `True` si la dirección es válida (o sea, **Conejochico** puede moverse en esa dirección) y `False` si **Conejochico** choca con una pared inmediatamente.

  El siguiente ejemplo es con un laberinto simplificado para mayor entendimiento. En caso de recibir `validar_direccion([['-', 'C', 'P'],['-', '-', '-']], "A")`, la función debe retornar `True` pues **Conejochico** puede avanzar hacia la izquierda sin chocar, mientras que `validar_direccion([['-', 'C', 'P'],['-', '-', '-']], "D")` retorna `False` por que **Conejochico** no puede ir hacia la derecha, choca con una pared.

- Modificar `def serializar_mensaje(mensaje: str) -> bytearray:`
  Recibe un mensaje y lo devuelve codificado como `bytearray`, usando el método `encode` para string de python con `big endian` y `UTF-8`.

  Por ejemplo, `serializar_mensaje('Hola mundo')` retornaría `bytearray(b'Hola mundo')`.

- Modificar `def separar_mensaje(mensaje: bytearray) -> list[bytearray]:`
  Ocupa el método de separación explicado en encriptación y retorna las secuencias `A`, `B` y `C` en una lista.

  Ejemplo: el resultado de `separar_mensaje(bytearray(b'\x03\x06\x02\x01\x05\x07\x04\x09\x01\x08'))` sería la siguiente lista:

  ```
  [
   bytearray(b'\x03\x07\x04'),
   bytearray(b'\x06\x05\x09'),
   bytearray(b'\x02\x01\x01\x08')
  ]
  ```

  donde los índices 0, 1 y 2 de la lista corresponden a las partes A, B y C, respectivamente.

- Modificar `def encriptar_mensaje(mensaje: bytearray) -> bytearray:`
  Esta función ocupa `separar_mensaje` para completar el proceso de encriptación y retornar la secuencia `nACB` o `nBAC` según corresponda.

  Ejemplo: `encriptar_mensaje(bytearray(b'\x03\x06\x02\x01\x05\x07\x04\x09\x01\x08'))` retornaría el siguiente *bytearray*:

  `bytearray(b'\x01\x03\x07\x04\x02\x01\x01\x08\x06\x05\x09')`

- Modificar `def codificar_mensaje(mensaje: bytearray) -> list[bytearray]:`
  Esta función recibe un mensaje a codificar en el formato pedido por el enunciado y retorna una lista de `bytearray`, en donde el primer elemento de la lista corresponde al `bytearray` que indica **el largo del mensaje**, y los elementos que siguen son, de forma alternada, el `bytearray` que representa el número de bloque y el `bytearray` que representa a un bloque con parte del mensaje original.

  Ejemplo: el resultado de `codificar_mensaje(bytearray(b'\x44\x44'))` sería la siguiente lista (en donde el tercer `bytearray` fue cortado en 3 líneas para que entrara en el documento, en tu computador se verá como solo una línea continua):

  ```
  [
   bytearray(b'\x00\x00\x00\x02'),
   bytearray(b'\x00\x00\x00\x01'),
  ```

```
            bytearray(b'\x44\x44\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                        \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                        \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
                      )
        ]
```

9.2. Entrega final (75 %)

En la entrega final de la tarea, se evaluará que se integren correctamente las funciones implementadas en la Entrega Intermedia, terminando todas las funcionalidades necesarias para el correcto funcionamiento del programa, tales como: levantamiento del servidor, desencriptación, levantar por completo la interfaz gráfica y conectar señales, entre otros, todo según los requerimientos dados por el enunciado.

Esta parte será corregida de forma manual, por lo que no se revisará nuevamente el contenido de las funciones evaluadas por *tests* en la Entrega Intermedia.

10. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T2/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Los elementos a ignorar para esta tarea son:

- El enunciado.
- La carpeta `assets/`.
- La carpeta `tests/`.

Recuerda **no ignorar archivos vitales de tu tarea como los que tú debes modificar, o tu tarea no podrá ser revisada**.

Es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos **deben** no subirse al repositorio debido al uso correcto del archivo `.gitignore` y no debido a otros medios.

11. Entregas atrasadas

La entrega intermedia no permite ningún tipo de atraso, mientras que la entrega final sigue la política de atrasos del curso. Posterior a la fecha de entrega de la tarea se abrirá un formulario de Google Form, en caso de que desees que se corrija un *commit* posterior al recolectado, deberás señalar el nuevo *commit* en el *form*.

El plazo para rellenar el *form* será de 24 horas, en caso de que no lo contestes en dicho plazo, se procederá a corregir el *commit* recolectado.

12. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y *push* en sus repositorios.

En el **siguiente enlace** se encuentra la distribución de puntajes. En esta señalará con color **amarillo** cada ítem que será evaluado a nivel funcional y de código, es decir, aparte de que funcione, se revisará que el código esté bien confeccionado, y se señalará con color **azul** cada ítem a evaluar con el correcto uso de señales. Todo aquel que no esté pintado de amarillo o azul significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.

Importante: Todo ítem corregido por el cuerpo docente será evaluado únicamente de forma ternaria: cumple totalmente el ítem, cumple parcialmente o no cumple con lo mínimo esperado. Luego, todo ítem corregido automáticamente será evaluado de forma binaria: pasa todos los *tests* de dicho ítem o no los pasa todos. Finalmente, todos los descuentos serán asignados manualmente por el cuerpo docente.

Una vez recopilado todo el contenido de la entrega, esta se ubicará en un directorio con nombre distinto a T2/. Por lo tanto, *paths* que hagan uso de las rutas "`T2/....`" o "`Tareas/T2/....` no se ejecutarán de forma correcta y el programa no funcionará. Es por esto que los *paths* usados deben ser relativos, asumiendo que al ejecutar el programa se ubicará la terminal dentro de la misma carpeta que contiene el contenido de la entrega, y que dicha carpeta no se llama T2/.

Para terminar, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante de Bienestar de tu sección. El correo está en el **siguiente enlace**.

13. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.10.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py` que estén correctamente ordenados por carpeta.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. El no incluir este archivo o bien incluirlo pero que se encuentre vacío conllevará un **descuento** en tu nota.
- Esta tarea se debe desarrollar **exclusivamente** con los contenidos liberados al momento de publicar el enunciado. No se permitirá utilizar contenidos que se vean posterior a la publicación de esta evaluación.
- Se encuentra estrictamente prohibido citar código que haya sido publicado **después de la liberación del enunciado**. En otras palabras, solo se permite citar contenido que ya exista previo a la publicación del enunciado.
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro que sí sea especificado por enunciado.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0)**.**