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

**IIC2233 Programación Avanzada (2019-1)**

# Tarea 02

## Entrega

- **Avance de tarea**
  - **Fecha y hora:** miércoles 15 de mayo de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T02/`
- **Tarea**
  - **Fecha y hora:** domingo 26 de mayo de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T02/`
- `README.md`
  - **Fecha y hora:** lunes 27 de mayo de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T02/`

## Objetivos

- Utilizar PyQt5 y conceptos de interfaces para implementar una aplicación interactiva
- Entender y aplicar los conceptos de *back-end* y *front-end*
- Implementar una separación entre el *back-end* y el *front-end*
- Aplicar conocimientos de *threading* en interfaces
- Aplicar conocimientos de señales

# Índice

# 1. Introducción a DCCivil War

¿Sabías que existen casi 6 millones de pingüinos y 600 millones de gatos en el mundo? Eso significa que si los pingüinos deciden luchar contra los gatos, a cada pingüino le tocaría pelear con 100 gatos. ¿Quién ganaría? Dada esta gran interrogante, los ayudantes han decidido simular cuál de las dos facciones de animales sería la ganadora en esta inminente batalla. Para esto deberás crear un juego de *Tower Defense*, en donde deberás seleccionar un bando (pingüinos o gatos) y, defender tu base colocando torres de manera estratégica a lo largo de tu territorio para derribar al enemigo.

# 2. DCCivil War

DCCivil War consiste en un juego del estilo *Tower Defense*[1], en donde el usuario debe defender su base militar de enemigos que buscan destruirla (en este caso **gatos** o **pingüinos**). Para ello, se compran y ubican en un mapa distintos tipos de torres, las que atacan a los enemigos que pasan por el camino cerca de ellas. Las compras se realizan utilizando monedas, que el jugador recogerá mediante un personaje controlado por medio del teclado. El objetivo del juego es defenderse y destruir enemigos por rondas, y así durar la mayor cantidad de rondas sin que derroten a tu base.

Deberás crear una aplicación que recree este juego utilizando PyQt5 y *threads*[2]. Se espera una aplicación modularizada con una correcta separación entre *back-end* y *front-end*. **Debes modelar todas las acciones de la interfaz (movimientos, ataques, etc.) mediante el uso de señales**.

A lo largo del enunciado te encontrarás con palabras escritas en mayúsculas *monospaced*. Éstas son variables de Python que deberás importar en tu aplicación. Por ejemplo: "El tiempo de duración de una moneda se mide en segundos, y está definido por el parámetro `DURACION_MONEDA`".

# 3. Parámetros

Existen datos que influyen en la dinámica del juego y que **se mantienen constantes a lo largo de toda la ejecución**. A estos datos les llamaremos **parámetros**.

Se te entregará un archivo llamado `parametros.py`, que contendrá los valores predefinidos de los parámetros del juego. Por ejemplo, si el juego tuviera dos parámetros, el archivo se vería de esta forma:

```
1 ATK_NORMAL = 10
2 SPD_NORMAL = 5
```

Debes completar este archivo agregando todos los parámetros que sean especificados en el enunciado. Si lo consideras necesario también puedes agregar algunos adicionales. De esta manera, podrán ser modificados con facilidad, permitiendo ejecutar el juego bajo distintos valores iniciales. Su tarea **no debe contener ningún valor numérico en su código**, sino que todos ellos deben estar en `parametros.py`. **No respetar esta indicación provocará un descuento**.

[1] Para más información puedes visitar el siguiente *link*.
[2] Esto incluye `QThreads`, `QTimer`, etc.

# 4. Flujo de juego

Al abrir la aplicación se debe mostrar una ventana que permita ingresar el nombre del jugador, el bando con el que desea jugar (gatos o pingüinos) y desplegar el puntaje histórico de los 10 mejores jugadores. Al comenzar a jugar, se debe abrir una interfaz con la ventana principal del juego. En esta interfaz debe ser posible visualizar gráficamente un mapa donde se aprecia la distribución espacial de los caminos, torres, enemigos y la base del jugador. También se debe poder ver la ronda actual en la que se encuentra el jugador, la cantidad de monedas que ha acumulado, el progreso de la ronda, opciones de compras de torres o mejoras, y la opción de iniciar la siguiente ronda (más detalles en Interfaz).

Una partida está compuesta por múltiples rondas. Al inicio de cada ronda habrá una fase de preparación en donde debe ser posible ver las distintas torres que se pueden comprar y, de tener las monedas necesarias, arrastrar las torres para colocarlas en los espacios disponibles del Mapa. También debe ser posible comprar mejoras que alteran el ataque de todas las torres durante el resto de la partida. Tras finalizar la fase de preparación, comienza una ronda.

Una vez que empieza una ronda, los enemigos comienzan a generarse en puntos predeterminados del mapa, moviéndose de forma aleatoria por un camino que está definido en un archivo a cargar. Una vez que las unidades enemigas entran en el rango de ataque de una torre, estas últimas comienzan a atacarlas hasta que salgan de su rango, mueran o la torre sea destruida. Cuando un enemigo muere, puede dejar Monedas en su ubicación, las que permitirán al jugador comprar mejoras o más torres. Los tipos de enemigos y torres están explicados en Enemigos y Torres.

Además de los enemigos, existe un personaje principal que puede recoger las monedas mientras están en el mapa. Este personaje se desplazará a lo largo del mapa a voluntad del jugador usando el teclado.

Tanto el personaje principal como los enemigos no son capaces de atravesar torres, la base, ni cualquier otro obstáculo ubicado en el mapa. Tampoco pueden salir de los bordes delimitados por el mapa. Los enemigos, adicionalmente, solo se pueden mover a través del camino predeterminado. No hay interacción entre el personaje principal y las unidades enemigas; es decir, no existen colisiones ni ataques entre ellos. Tampoco hay interacción entre los enemigos.

Si los enemigos logran destruir la base, el jugador pierde y la partida termina. En cambio, si logra vencer y derrotar a todos los enemigos de la ronda, tendrá la oportunidad tanto de comprar más torres y mejoras para jugar una ronda adicional, como de terminar la partida [3].

Al finalizar una partida, se despliega el resultado de dicha partida, la información de monedas obtenidas, número de ronda alcanzada y se da la opción de volver a jugar o volver al menú inicial. Más detalles en la sección Fin de ronda.

# 5. Entidades

## 5.1. Enemigos

Esta entidad corresponde a la especie opuesta a la escogida inicialmente por el jugador y es aquella que busca atacar la base militar del jugador. Los enemigos se mueven según casillas predefinidas por caminos en el mapa del juego, rotando y avanzando según lo especificado en la sección Movimiento. Un enemigo no podrá moverse fuera del camino indicado en el mapa, ni podrá atravesar obstáculos de ningún tipo.

[3] Podrás notar que nunca podrás ganar, solo completar la mayor cantidad de rondas :(

Todos los enemigos poseen un valor de *hp* (puntos vitales). Al inicio de cada ronda, todos empiezan con el mismo valor *hp*. Este valor está determinado por el parámetro `HP_ENEMIGOS`, que es un valor positivo. Un enemigo es destruido cuando su *hp* llega a 0. Luego de ser destruido, con cierta probabilidad, un enemigo puede dejar Monedas en el mapa.

Existen dos tipos de enemigos:

- **Enemigo normal.** Un enemigo común y corriente. Una vez que llegan a la base, lo que es equivalente a estar en una celda adyacente a ésta, empieza a atacarla proporcionándole una cantidad de daño fija, y con una frecuencia de ataque constante, durante todo momento en que se encuentren adyacentes. Los parámetros que determinan la cantidad de daño y frecuencia de ataque, es decir, cantidad de ataques por segundo, son respectivamente `ATK_NORMAL` y `SPD_NORMAL`.
- **Enemigo kamikaze.** Entrenado toda su vida para morir luchando por su pueblo, este tipo de enemigo posee la habilidad de explotar y autodestruirse cuando colisiona (evento definido en la sección Movimiento) con alguna torre o base enemiga. El daño causado es de tal magnitud que destruye completamente a dicha torre con la explosión. Por otro lado, si el Kamikaze colisiona con la base principal, ésta no será destruida necesariamente, sino que recibirá un daño equivalente a la constante `ATK_KAMIKAZE`. Si este valor es mayor al *hp* actual de la base, entonces la base sí será destruida.

El número de enemigos ($N_{\text{enemigos}}$) que aparecerá en cada ronda se define mediante la función:

$$N_{\text{enemigos}} = ((N_{\text{ronda}} - 1) \times \texttt{GENERACION\_1}) + \texttt{GENERACION\_2}$$

donde:

- $N_{\text{ronda}}$: Número de ronda actual (la primera ronda es la 1)
- `GENERACION_1`: Constante almacenada dentro del archivo de parámetros.
- `GENERACION_2`: Constante almacenada dentro del archivo de parámetros.

Dado un valor de $N_{\text{enemigos}}$, la proporción de enemigos normales versus kamikazes será $3:1$. Finalmente, queda a tu criterio el ver cómo se distribuye la aparición de enemigos durante una ronda. Por ejemplo, podrían aparecer todos juntos al inicio de la ronda, o podrían aparecer por grupos cada cierto intervalo de tiempo predeterminado (recuerda agregar este valor a `parametros.py`), o algún otro esquema.

### 5.2. Personaje

Es el personaje controlado por el usuario durante las rondas del juego. Puede moverse libremente por el mapa, siempre cuidando que no atraviese las torres, la base, los obstáculos y los bordes del mapa. Su movimiento es controlado por el teclado (teclas WASD), y su misión es recolectar las monedas soltadas por los enemigos y así poder comprar mejoras para las siguientes rondas. El personaje no posee un valor de *hp*, ya que no puede ser dañado ni por las torres ni por los enemigos.

### 5.3. Base

Es el centro de operaciones de cada especie, lo que la hace un elemento crucial para su supervivencia y, por ende, el principal blanco de los enemigos.

La base posee un valor *hp*, que inicialmente está determinado por el parámetro `HP_BASE`. Su *hp* puede disminuir por ataques de enemigos pero no aumenta a lo largo del juego. Al momento de iniciar una nueva partida, se crea automáticamente una base sin costo alguno para el jugador y se ubica al final del

recorrido del mapa. El juego se acaba cuando, en alguna ronda, la base es destruida (su *hp* llega a 0), en cuyo caso el jugador pierde. La base no posee modos de ataque ni defensa ante los enemigos, y no es posible comprar una en la tienda.

### 5.4. Torres

Las torres son edificaciones poseídas por el jugador. Su objetivo es detener el inminente ataque animal y proteger la base. Existen dos tipos de torres:

- **Torre francotiradora**. Gracias a su precisión mortal, esta torre se enfoca en un único enemigo hasta que éste muere, desaparece de su rango de ataque o la torre es destruida. La rapidez, medida en número de ataques por segundo, y el daño que proporciona un ataque de francotiradora están almacenados en los parámetros `SPD_FRANCOTIRADORA` y `ATK_FRANCOTIRADORA`, respectivamente. La torre francotiradora siempre elige como objetivo al primer enemigo que haya entrado dentro de su rango de ataque. Si este enemigo deja de estar disponible (por haber muerto o salido del rango efectivo) y hay más de un enemigo dentro de su rango en ese momento, elige al enemigo con menor *hp* como nuevo objetivo.
- **Torre racimo**. Esta torre ataca a **todos** los enemigos que se encuentren dentro de su rango de ataque. El daño que recibe cada enemigo afectado por una torre racimo, y su velocidad de ataque se almacenan en los parámetros `ATK_RACIMO` y `SPD_RACIMO`, respectivamente.

Todas las torres poseen el mismo rango inicial de ataque. Este rango consiste en un área cuadrada de una celda (explicadas en la sección Mapa) alrededor de la torre, ejemplificada en la Figura 1. El enemigo que se encuentra dentro del área azulada está dentro del rango y es atacado, mientras que el enemigo fuera del área no recibe ningún tipo de daño, siempre y cuando se mantenga fuera del área azulada.

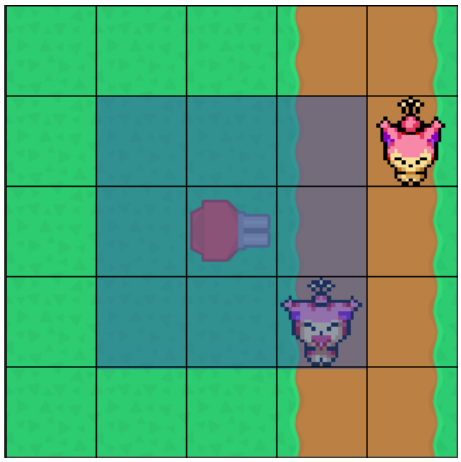


Figura 1: Rango de ataque de **Torres**

Todas las torres tienen un costo fijo en monedas, definido por el parámetro `COSTO_TORRES`.

## 6. Monedas

Dentro del juego, existen las monedas que te permiten comprar torres y adquirir mejoras para ellas de modo que te permita enfrentar de mejor forma a tus adversarios.

Las monedas aparecen ocasionalmente en cualquier parte del mapa, en periodos fijos en segundos, definido por el parámetro `TIEMPO_MONEDA`. Estas monedas estarán en el campo de batalla durante una cantidad fija de tiempo en segundos, almacenado en `DURACION_MONEDA`, y luego desaparecerán.

Alternativamente, los enemigos al ser destruidos tienen una probabilidad de entregar una moneda con una probabilidad, definida por el parámetro `PROBABILIDAD_ENEMIGO_MONEDA`.

El jugador acumula monedas a lo largo de cada ronda utilizando su personaje. Las monedas que aparezcan en el mapa deben aparecer en un lugar que no esté ocupado por un obstáculo o por una torre.

## 7. Mejoras

Las mejoras permiten aumentar algún aspecto de tus torres y pueden ser adquiridas usando monedas.

- **Aumento de ataque.** Aumenta el ataque de las torres en un porcentaje fijo, almacenado en `MEJORA_ATAQUE`. Solo se puede comprar una vez y permanece durante toda la partida. Tiene un costo fijo determinado por el parámetro `COSTO_MEJORA_ATAQUE`.
- **Aumento de rango de daño.** Aumenta el rango de alcance de las torres en `MEJORA_ALCANCE` casillas. Esta mejora solo se puede comprar una vez y permanece durante toda la partida. El aumento de alcance se aplica a los lados del área cuadrada explicada en la Figura 1. En ese ejemplo, el rango de ataque es de 3 casillas. Una vez aplicada la mejora, el nuevo rango de ataque de la torre sería $3 +$ `MEJORA_ALCANCE`. Puedes suponer que esta constante siempre será par, para que el rango total siempre sea impar y puedas ubicar la torre correctamente en el centro del cuadrado. Solo se puede comprar una vez y permanece durante toda partida. Tiene un costo en monedas dado por `COSTO_MEJORA_ALCANCE`.

## 8. Mapa

La carpeta `mapas` contiene tres archivos: `mapa_1.txt`, `mapa_2.txt` y `mapa_bonus.txt`. Estos mapas codifican, mediante letras, la forma y distribución espacial de los elementos de distintos mapas (cada letra está separada por un espacio).

Todos los elementos en el mapa tienen el mismo tamaño, delimitados por un cuadrado de `N` $\times$ `N` pixeles, donde `N` es una constante que quedará a tu criterio, pero que debe estar en `parametros.py`. El mapa, a su vez, será rectangular y tendrá dimensiones de $i \times j$ elementos. Estas dimensiones se entregan en los archivos.

Se definen **cinco tipos de espacios** que deberás considerar al momento de armar tu juego:

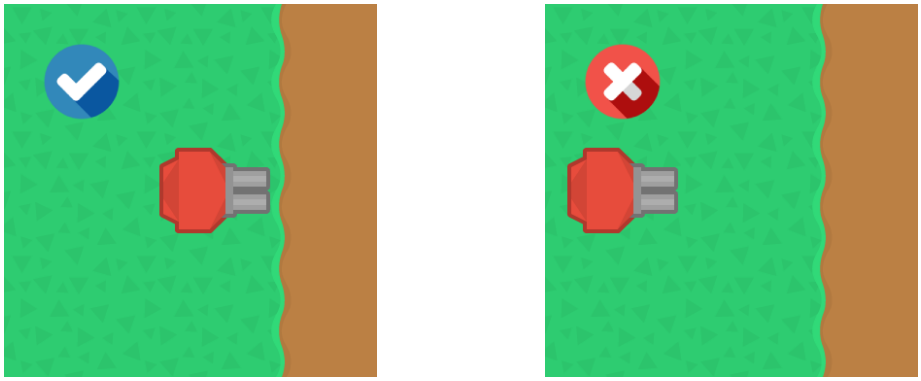
- **Camino (`C`)**: Espacios que, unidos, generan una vía por donde los enemigos y el personaje se desplazan. El punto de partida de una vía está marcado por un elemento **Inicio**, mientras que el punto de llegada está marcado por un elemento **Base**; en otras palabras, es la única vía por la que se mueven los enemigos.

  Una vía podrían separarse en varias otras, o varias vías podrían fusionarse en una, pero no formarán ciclos. No se pueden construir torretas sobre un **Camino**.

- **Inicio (`S`)**: Espacios en donde aparecen las unidades enemigas y comienzan su recorrido hacia la base. Siempre tendrán al menos un espacio contiguo de tipo **Camino**. Cada camino cuenta con dos elementos de tipo **Inicio** como se muestra Figura 3. Dentro de un mismo mapa puede haber más de un punto de partida, y éstos pueden estar distribuidos en cualquier parte del borde del mapa.

Asimismo, el total de enemigos a generar por ronda deberá repartirse equitativamente entre todos estos espacios[4]. No se pueden construir torres sobre un **Inicio**.

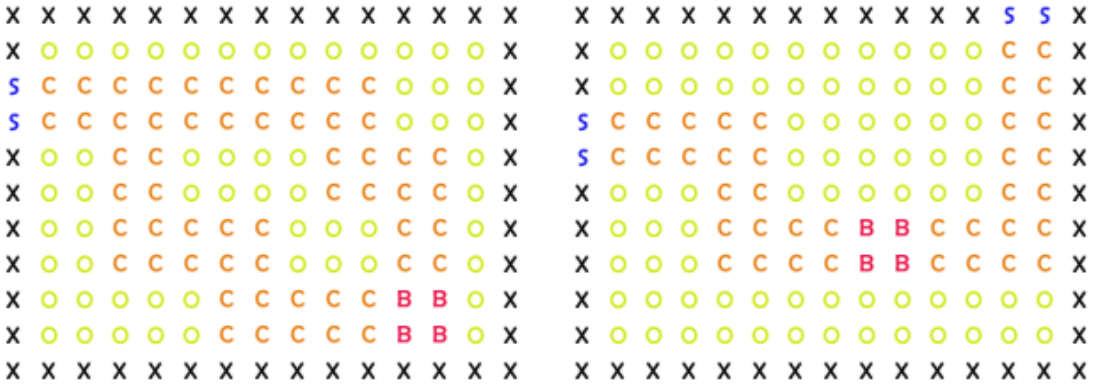
- **Base (`B`)**: Espacios ocupados por la única base del jugador, los cuales siempre serán una agrupación contigua de $2 \times 2$ celdas[5]. Este es también el destino al que las unidades enemigas tienen que llegar y marcan el final de uno o más **Caminos**. No se pueden construir torres sobre estos espacios, ni tampoco atravesarlos.
- **Espacio libre (`O`)**: Espacios destinados para la construcción de torres y para el desplazamiento del personaje. Las torres se pueden construir en un **Espacio libre** solo si hay, a lo menos, un elemento **Camino** adyacente, como se ve en la Figura 2. Al construir una torre sobre un **Espacio libre**, éste pasa a ser un elemento de tipo **Obstáculo**. Por otro lado, cuando una torre es destruida, el espacio que ocupaba pasa a ser un elemento de tipo **Espacio libre**.



(a) Torre bien ubicada (b) Torre mal ubicada

Figura 2: Ubicación de las torres con respecto a un camino.

- **Obstáculo (`X`)**: Espacios que, tanto el personaje como las unidades enemigas no pueden ocupar o atravesar, ya sea por la presencia de una torre o de algún otro objeto; tampoco se pueden construir torres sobre ellos.



(a) Con bifurcación (b) Con dos puntos de partida

Figura 3: Ejemplo de posibles mapas.

[4]Ejemplo: Si se deben generar $N$ enemigos totales y existen $M$ elementos de tipo **Inicio** en el mapa, entonces se deben generar $\frac{N}{M}$ enemigos en cada uno.

[5]**Ojo:** Esto quiere decir que la *sprite* de la base aliada ocupa los 4 espacios, no que por cada espacio haya un *sprite* de la base. Para más información con respecto a las *sprites* entregadas, revisa la sección *Sprites*.

Las Monedas pueden aparecer en los espacios de tipo **Camino** y **Espacio libre**. La aparición de una moneda no transforma un **Espacio libre** en un **Obstáculo**. No obstante, tendrás que controlar que no se generen dos o más monedas en un mismo espacio.

Para efectos del juego, deberás cargar **uno** de los archivos entregados. Debes guardar la ruta relativa del mapa a utilizar en un parámetro `RUTA_MAPA`; pudiendo cambiar el mapa utilizado por otro creado manualmente. Tu programa debe ser capaz de leer **cualquiera** de estos archivos (y otros que pudiésemos querer probar) y funcionar correctamente independiente de las características que presenten (caminos bifurcados, puntos de inicio múltiples, etc).

# 9. Puntaje

El puntaje obtenido en una ronda del juego se calcula de la siguiente forma:

$$\frac{monedas\_obtenidas \times vida\_base}{torres\_perdidas + 1} + enemigos\_derrotados$$

donde:

- *monedas_obtenidas*: Cantidad de monedas que se recogieron durante una ronda.
- *vida_base*: *hp* final con que termina la base.
- *torres_perdidas*: Cantidad de torres que se pierden durante la ronda.
- *enemigos_derrotados*: Cantidad de enemigos que fueron destruidos durante la ronda.

El puntaje total del juego, se calcula como la suma de los puntajes obtenidos por cada ronda.

## 9.1. Guardado de puntajes

Al finalizar cada partida, ya sea cerrando la sesión o perdiendo una ronda, deberás guardar en un archivo de texto plano, llamado `puntajes.txt` el puntaje total que se ha obtenido durante esta, así como también el nombre del usuario que jugó dicha partida. El formato de este archivo queda a tu criterio, pero debes poder utilizarlo sin problemas al mostrar el *ranking* de puntuaciones más altas de la sección Ventana de inicio. Ten presente que si bien para el *ranking* de los jugadores con mayores puntuaciones solo deberás mostrar a los **diez** mejores, deberás almacenar todos los registros, pudiendo ser muchos más de diez.

# 10. Interfaz

## 10.1. Introducción

Para el *front-end* de DCCivil War, debes crear una interfaz gráfica de usuario (GUI) usando la biblioteca PyQt5. La GUI constará de dos ventanas diferentes. En la primera están los menús y en la segunda debe verse la preparación de una ronda y la ejecución de la ronda en sí. Debes tener presente que tu interfaz debe ser robusta a todos los *inputs* posibles del usuario.

## 10.2. Ventana de inicio

Al abrir el juego, entrarás directo al menú de inicio, que provee varias opciones para el usuario:

1. **Ingresar jugador:** el jugador debe poder indicar su nombre y hacer *click* en un botón para iniciar su sesión.
2. **Cerrar sesión:** una vez iniciada la sesión, debe darse la opción de cerrar la sesión, lo cual permite al jugador ingresar otro nombre.
3. **Elegir bando:** el jugador debe poder seleccionar el bando al que pertenecerá en la partida. Esto se debe realizar con `RadioButton`, `CheckBox`, `Buttons` o cualquier `QObject` que no involucre ingresar texto.
4. **Jugar:** solo después de que algún usuario haya iniciado sesión y escogido un bando, se debe dar la opción de **Comenzar una partida nueva**, la que parte siempre en la primera ronda y con una puntuación inicial de 0. Cuando se hace *click* en **Comenzar una partida nueva**, se abre la ventana de juego y queda a tu criterio el qué hacer con la ventana de inicio.
5. **Ranking puntuaciones más altas:** debe mostrar una lista ordenada con los **diez** usuarios con mayores puntuaciones. Los usuarios y sus puntuaciones deben estar ordenados de mayor a menor.

## 10.3. Ventana de juego

Esta ventana muestra todo lo necesario para preparar la ronda y luego jugarla.

### 10.3.1. Preparando ronda

En esta interfaz, el usuario debe ser capaz de preparar la defensa del juego. Debes mostrar el mapa del juego con las torres que hayan sobrevivido de la ronda anterior. En caso de ser la primera ronda, el mapa empieza sin torres. También debe mostrar los tipos de torres y mejoras disponibles, además de ciertos textos con información para el jugador. En concreto, debes mostrar:

- ☐ Un icono que indique el bando elegido.
- ☐ El mapa de la ronda, tal como se describe en Mapa, incluyendo:
  - ☐ Base
  - ☐ Caminos
  - ☐ Puntos de aparición de los enemigos
  - ☐ Obstáculos
  - ☐ Torres colocadas
- ☐ Los tipos de torres disponibles para colocar en el mapa. Para ubicar una torre, debes **arrastrarla** hasta una ubicación válida. Ten en consideración, que al soltar la torre, ésta puede no coincidir exactamente con una casilla. En ese caso, la torre deber quedar en la casilla que contenga el centro de la imagen.

  Puedes colocar tantas torres en el mapa como tu dinero te permita comprar. El costo de la torre se descontará una vez que esté situada correctamente sobre el campo de batalla. Debes mostrar sus costos y actualizar la cantidad de monedas que tienes en tu poder. Además, si el usuario hace *click*

en una torre que ya se encuentra colocada en el mapa, ésta desaparece (se destruye) dejando libre dicha casilla. Las monedas que se invirtieron en construir esta torre no se devuelven.

- ☐ El total de monedas del jugador.
- ☐ Las mejoras disponibles por comprar (ver más en Mejoras). Para cada una se debe mostrar:
  - ☐ El costo de la mejora.
  - ☐ Un botón para comprarla, que debe estar bloqueado si ya fue comprada.
- ☐ La cantidad de enemigos que vendrán en la ronda que está por comenzar.
- ☐ Un botón para empezar a jugar la ronda.
- ☐ Un botón para cerrar sesión, el cual cierra tanto la ventana de juego y la sesión, y te devuelve a la ventana de inicio.

Luego de cualquier compra, el dinero mostrado debe actualizarse correctamente. El *back-end* debe ser quien compruebe que todas las acciones realizadas por el usuario sean correctas y avisar o impedirlas en el caso contrario.

### 10.3.2. Ronda

Una ronda empieza cuando haces *click* en el botón de "Empezar ronda". Debes deshabilitar todos los elementos relacionados con la preparación de la ronda, y habilitar o incluir los elementos relacionado con las funcionalidades propias de esta etapa. Se debe mostrar:

- ☐ Una barra de progreso con el avance de la ronda, medido en cantidad de enemigos muertos sobre el total de la ronda.
- ☐ Monedas totales disponibles para el jugador.
- ☐ Sobre la cabeza de cada enemigo, y también de la base, debe haber una barra que indique el porcentaje de *hp* que le queda actualmente.
- ☐ Una etiqueta[6] con la cantidad de vida restada a un enemigo o la base cuando sufran daño. Ésta debe aparecer sobre el elemento cada vez que se pierda *hp* y se debe mantener durante un tiempo fijo en segundos, definido por `TIEMPO_VIDA_PERDIDA`. Si una entidad recibe daño mientras se muestra una etiqueta de un daño anterior a esa misma entidad, se debe reemplazar la etiqueta por el nuevo daño realizado.
- ☐ Cuando una torre ataca, debe aparecer una etiqueta sobre ella que lo señalice. Ésta se debe mantener durante un tiempo fijo en segundos, definido por `TIEMPO_ATAQUE_TORRE`.
- ☐ La ronda en la que se encuentra el jugador.
- ☐ Botón de pausa. Pausa o continúa el juego. Debe ser activable también con la tecla `P`. Debes representar visualmente que el juego está pausado, sin ocultar los elementos del juego. Al pausar se detiene tanto: el personaje; los enemigos y su aparición; las monedas y su aparición; y el ataque de las torres y los enemigos.

[6] Pueden considerarlo como *label*.

### 10.3.3. Movimiento

Los personajes deben poseer movimientos animados, tanto para avanzar como para atacar. Para lograr esto, se debe diferenciar su aspecto al moverse, considerando al menos tres estados de movimiento para cada acción y dirección, como los mostrados a continuación:

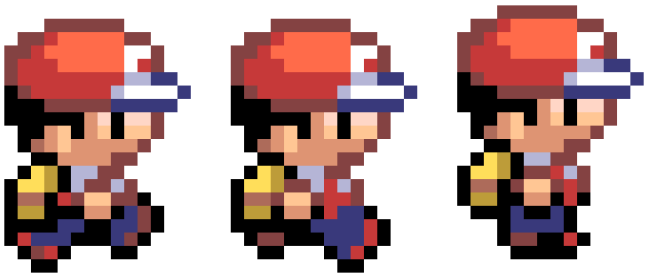


Figura 4: Estados de movimiento del personaje.

De esta forma, el movimiento se verá más fluido. Para esta sección, pueden buscar *sprites* en internet, utilizar las entregadas junto al enunciado o usar unas creadas por ustedes. Para más información con respecto a las *sprites* entregadas, revisa la sección *Sprites*.

Una **colisión**[7] se produce cuando el jugador o un enemigo intenta moverse en una dirección que, de realizarse el movimiento, lo dejaría en una posición inválida.

A continuación se explica en detalle el movimiento del personaje y los enemigos:

- **Personaje:** Su movimiento **no** se encuentra restringido por las casillas, es decir, se mueve una cantidad fija de píxeles. La cantidad de píxeles que se mueve está determinada por `SPEED_PLAYER` píxeles por segundo.

  Durante su movimiento deberás considerar que el personaje **no** podrá atravesar la base, las torres, los obstáculos, ni salir de los bordes del mapa, por lo que deberás considerar su colisión (intentar "moverse hacia", cuando está en "contacto con") con estos elementos.

  Durante una colisión el personaje del jugador **no** debe dejar de tener la animación de movimiento pese a que no avance en el mapa.

Figura 5: Movimiento del Personaje durante una colisión.

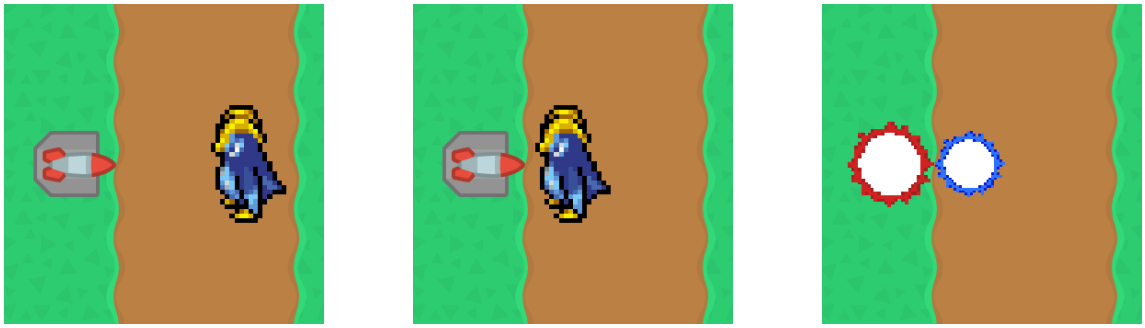
[7] **Hint:** Para comprobar las colisiones puedes utilizar la clase `QRect` de `QtCore` y el método `intersects`. Este permite obtener la intersección de dos objetos `QRect`, donde se retorna `True` si es que hay intersección y `False` en caso contrario. Existen más formas de hacerlo, en todo caso. Ésta sólo es una de varias.

- **Enemigos:** Únicamente pueden moverse por los caminos del mapa. Su movimiento y rotación son aleatorios, es decir, que eligen aleatoriamente la dirección en la que se moverán. Además, se encuentran restringidos a las celdas del mapa, es decir, al moverse deben avanzar de casilla en casilla con un movimiento animado. En ningún momento, a excepción de que se pause el juego, su movimiento puede acabar en una posición que no corresponda exactamente a la de una casilla. Los enemigos no colisionan entre sí, por lo tanto puede ocurrir que haya más de uno ocupando exactamente la misma casilla. En este caso debes mostrarlos todos.

  Cuando un Kamikaze está en una casilla adyacente a una torre o base, éste explotará destruyendo la torre o infligiendo daño a la base. En caso que sea un enemigo normal, éste sólo dañará la base si es que se encuentra en una casilla adyacente a ella.

(a) Enemigo normal antes de hacer colisión con la base. (b) Enemigo normal al hacer colisión con la base. (c) Disminución de *hp* de la base.

Figura 6: Colisión de un Enemigo normal con la base.[8]



(a) Enemigo kamikaze antes de hacer colisión con una torre. (b) Enemigo kamikaze al hacer colisión con una torre. (c) Explosión del enemigo kamikaze junto con la torre.

Figura 7: Colisión de un Enemigo kamikaze.[9]

**IMPORTANTE:** Al tener los enemigos un movimiento aleatorio, puede darse la situación que una ronda no termine porque uno o más enemigos no avancen hacia una torre o base durante un tiempo prolongado. Es tu deber como programador ver como evitar esta situación. La solución que uses queda completamente a tu criterio[10], solo debes mencionarla debidamente en tu README.

[8] Por simplicidad, no se muestra la barra de *hp* del enemigo.

[9] Por simplicidad, no se muestra la barra de *hp* del enemigo.

[10] Por ejemplo: puedes usar un temporizador, un *cheat code*, un botón especial, *aging* de los enemigos (que puedan dar una cantidad máximo de pasos y luego mueran), o cualquier opción que estimes conveniente para forzar la muerte de un enemigo y permitir que el juego continúe. Solo considera que no debe intervenir con el normal desarrollo del resto del juego impidiendo probar alguna otra funcionalidad.

#### 10.3.4. Fin de ronda

Tras finalizar una ronda, debes mostrar el resultado (victoria o derrota), y lo siguiente:

- Monedas conseguidas en la ronda que acaba de terminar.
- Monedas totales.
- Puntaje total obtenido.
- Botón para pasar a la siguiente ronda en caso de victoria.
- Botón para cerrar la sesión y ventana de juego, y volver al menú principal.

## 11. Funcionalidades extras

Con la finalidad de ~~facilitar la corrección~~ mejorar la experiencia de los jugadores al jugar DCCivil War se te pide implementar los siguientes secuencias de teclas para hacer "trampa" (cheat):

- **E + N + D**: Ingresar esta combinación de teclas debe terminar la ronda actual, considerando como ganador al jugador. No se recibe puntaje por los enemigos que quedaban en la partida.
- **M + O + N**: Ingresar esta combinación aumenta en `DINERO_TRAMPA` tus monedas.
- **Destrucción Torre (*Click*)**: No siempre se puede ganar una batalla. Esto es un mecanismo de autodestrucción. Hace que una torre explote arrasando con todo lo que encuentra a su paso, dañando a los enemigos a su alrededor y aturdiendo a las demás torres. El alcance de esta explosión es por todo el mapa, pero disminuye a medida que estás más lejos. El daño causado a los enemigos por la explosión se calcula de la siguiente forma:

$$\frac{\texttt{CONSTANTE\_DESTRUCCION}}{distancia^2}$$

  donde:

  - `CONSTANTE_DESTRUCCION`: Constante que debes definir.
  - *distancia*: La distancia euclidiana que separa a la torre del enemigo. Debes calcular esta distancia para cada enemigo que esté en el mapa (usa la casilla como unidad de distancia).

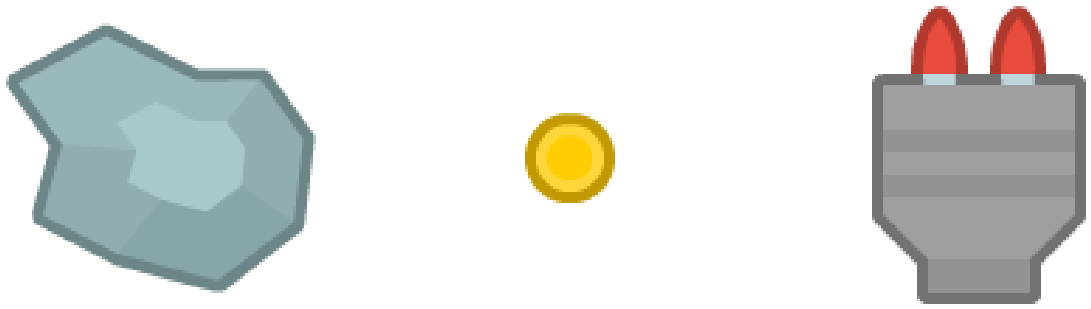
  Ten en cuenta que, una vez que la torre explota, esta queda totalmente destruida. A su vez, el resto de tus torres quedan inutilizables durante un tiempo fijo en segundos, dado por el parámetro `TIEMPO_INUTILIZABLE`. Luego de ese tiempo, todas las torres que permanecen vivas vuelven a funcionar normalmente. Para aplicarlo, debes hacer *click* sobre la torre que deseas destruir.

## 12. *Sprites*

Para esta tarea, se recomienda el uso se sprites[11] para representar gráficamente las distintas entidades del juego. Se entregará la carpeta `sprites` la cual contiene:

[11] Para más información puedes visitar el siguiente *link*.

- **`mapa`:** Carpeta que contiene las imágenes necesarias para construir el mapa del juego (espacios libres, caminos y obstáculos), las torres y las monedas.



(a) Obstáculo (b) Moneda (c) Torre

Figura 8: *Sprites* contenidas en la carpeta mapa.

- **`pinguino`**: Carpeta que contiene las *sprites* de movimiento y ataque de los pingüinos normales y kamikaze, junto con su base. Se incluye su *spritesheet* para que puedas identificar a qué movimiento corresponde cada *sprite*.

Figura 9: *Sprites* de movimiento de los pingüinos normales.

- **`gato`**: Carpeta que contiene las *sprites* de movimiento y ataque de los gatos normales y kamikaze, junto con su base. Se incluye su *spritesheet* para que puedas identificar a qué movimiento corresponde cada *sprite*.

Figura 10: Base de los gatos.

- **`personaje`**: Carpeta que contiene las *sprites* de movimiento del personaje.

Dentro en cada carpeta hay imágenes adicionales que podrían serte de utilidad para tu tarea.

En caso de que quieras hacer uso de tus propias *sprites* deberás guardarlas en carpetas con otro nombre e indicarlo en el README. Para obtener las *sprites* de los *spritesheets* se te aconseja utilizar la herramienta *ShoeBox* la cual te permite cortar, quitar el fondo y renombrar las *sprites*.

Como se indicará más adelante en la sección `.gitignore`, no debes subir la carpeta *sprites* a tu repositorio, por lo que **no debes cambiar el nombre de las imagenes**. Para facilitarte el acceso a las direcciones de las imágenes (que puedes acceder de forma más lógica) te recomendamos utilizar un diccionario como el que se muestra a continuación.

```
mov_gato_normal = {
    'down_0': 'sprites/gato/normal/normal_04',
    'down_1': 'sprites/gato/normal/normal_05',
    'down_2': 'sprites/gato/normal/normal_06'
}
```

## 13. `.gitignore`

Para esta tarea **la carpeta llamada `sprites` deberá ser ignorada** con un `.gitignore` que deberá estar dentro de tu carpeta T02. Puedes encontrar un ejemplo de `.gitignore` en el siguiente *link*.

En caso de que hagas uso de tus propias *sprites*, deberás guardarlas en otra carpeta y asegurarte que no sean ignoradas por el `.gitignore`.

Tampoco deberás subir el enunciado y los archivos contenidos en la carpeta `mapas`.

Se espera que no se suban archivos autogenerados por programas como PyCharm, o los generados por entornos virtuales de Python, como por ejemplo: la carpeta `__pycache__`.

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

## 14. Avance de tarea

El avance de esta tarea corresponde a subir una versión *alpha* del juego. En particular, esta versión debe cumplir **al menos una** de las siguientes restricciones :

- Mostrar un personaje con las imágenes dadas y un obstáculo en el mapa. El personaje se debe mover en las 4 direcciones con las teclas WASD y no permitir colisión entre el personaje y el obstáculo.
- Mostrar un enemigo con las imágenes dadas y un obstáculo en el mapa. El enemigo debe moverse automáticamente hacia el obstáculo, colisionar con él y luego moverse en otra dirección.

En cualquiera de las dos opciones, deberás entregar una interfaz gráfica diseñada con PyQt5, la cual deberá presentar un **correcto uso de señales** para realizar la comunicación entre el *back-end* y el *front-end*.

## 15. Bonus

Para esta tarea habrán **dos bonus** para los estudiantes que puedan implementar estas funcionalidades extra. Cada bonus se considerará completo **solo si fue implementado en su totalidad**.

Para poder optar a los bonus, **debes obtener una nota igual o mayor a 5.0** en la tarea, considerando los descuentos que corresponda aplicar.

### 15.1. Efectos de Sonido (3 décimas)

Para obtener este bonus debes agregar efectos de sonido[12]. Los archivos que utilices **deben estar en formato `.wav`**. Los efectos a agregar son:

- Música de fondo.
- Ataque de torres.
- Destrucción de torres
- Ataque de enemigos normales.
- Muerte de enemigos normales.
- Explosión de enemigos kamikaze.

Finalmente el sonido de tu juego **debe poder ser desactivado** mediante un botón en la pantalla de juego.

### 15.2. Camino más Corto (5 décimas)

Para obtener este bonus debes hacer que los enemigos sigan siempre el camino **más corto posible** hacia la base. Esto quiere decir que para cada punto de entrada `S` debes calcular el camino más corto para llegar a la base y hacer que los enemigos **sigan ese camino**.

Por ejemplo:

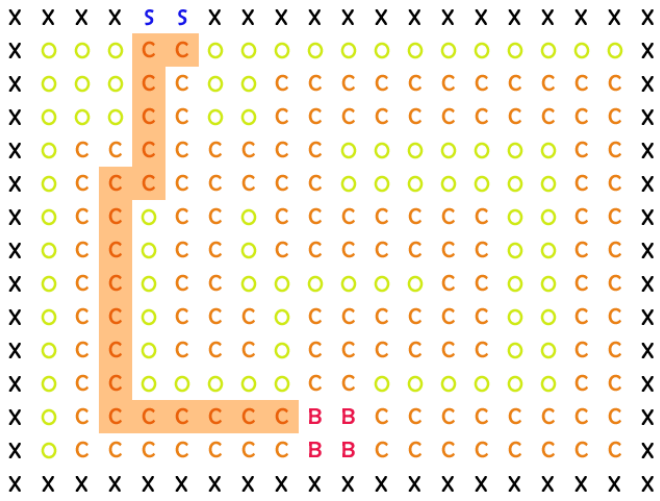


Figura 11: Camino Corto

En esta figura se ve que los enemigos tienen muchos posibles caminos para seguir, algunos **sin salida**. El camino más corto se encuentra subrayado, para obtener el bonus todos los enemigos que aparezcan deben seguir de forma **exacta y consistente** el camino más corto, para todo posible mapa.

[12] **Hint:** Hagan uso del módulo `QSound`.

## 16. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del juego será el pilar de la corrección, es decir, **solo 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.

Cuando se publique la distribución de puntajes, se señalará con un color rojizo cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado en rojizo significa que será evaluado si y solo si se puede probar con la ejecución de su tarea. En tu README deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

Finalmente, se recomienda el uso de *prints* para ver los estados del sistema (*hp* de las entidades, ataque de las torres, etc.) pero no se evaluará ningún ítem por consola. Esto implica que hacer *print* del resultado una función, método o atributo no valida, en la corrección, que dicha función esté correctamente implementada. Todo debe verse reflejado en la interfaz. Por ejemplo, si un enemigo pierde *hp*, que hagan *print* de su vida disminuyendo no será válido, para este ítem se evaluará que la barra de vida en la interfaz disminuye.

## 17. 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 3.6.
- Si no se encuentra especificado en el enunciado, asume 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. **Tendrás hasta 24 horas después del plazo de entrega** de la tarea para subir el *readme* a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 24 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

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