Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2022-1

5 de Mayo de 2022
Actividad Sumativa

Actividad Sumativa 3

Interfaces Gráficas

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AS3/`
- **Hora del *push*:** 16:40

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Importante: Debido a que en esta actividad se usarán archivos más pesados de lo normal (imágenes y archivos `.ui`), es importante el uso correcto del archivo `.gitignore` para ignorar la carpeta `frontend/assets/` y el enunciado.

Introducción

¡Un aumento en la población de conejos cerca de los cultivos de zanahorias ha estado causando estragos en la ciudad! Si la situación sigue así, pronto las zanahorias habrán desaparecido por completo. Es por esto que el sindicato de DCCultivadores le ha pedido al DCC que haga algo al respecto. Te han elegido a ti, estudiante de Programación Avanzada, para poder crear una interfaz donde los DCCultivadores puedan entrenar ~~su camino ninja~~ sus reflejos con tal de atrapar a los conejos lo más pronto posible. Y, como está comprobado científicamente ~~créannos~~ que se aprende mejor jugando, ¡Darás vida a DCCrush & Smack!

Flujo del Programa

El programa comienza con una ventana de inicio, introduciendo el usuario al juego. Aquí este debe ingresar un nombre de usuario y una contraseña. Si el nombre de usuario es conforme a los requerimientos y la contraseña es correcta se pasa a la ventana de juego. En esta ventana el jugador debe apretar las teclas en el momento correcto para aplastar a esos molestos conejos. Cada acierto va agregando al puntaje total hasta que se acabe el tiempo y se pasa a la ventana final. Esta última ventana mostrará el puntaje del jugador y le dará la opción de salir o volver a la pantalla inicial.

Archivos

Los archivos relacionados con la interfaz gráfica del programa se encuentran en la carpeta `frontend/`. Esta carpeta contiene los archivos `ventana_inicio.py`, `ventana_juego.py` y `ventana_postjuego.py`. Para completar los aspectos gráficos de DCCrush & Smack deberás modificar los dos primeros archivos. La carpeta `assets/` contiene los elementos gráficos del programa (imágenes y archivos `.ui` de *Qt Designer*)

La lógica del programa se encuentra en la carpeta `backend/`. Esta carpeta contiene a los archivos `logica_inicio.py` y `logica_juego.py`**, los cuales deberás modificar para completar los aspectos de lógica del programa**.

El archivo `main.py`, es el archivo principal que inicia la aplicación y maneja las conexiones de señales entre *front-end* y *back-end* utilizadas en el programa.

Por último, se encuentra el archivo `parametros.py`, el cual contiene todos los parámetros fijos del programa así como también las rutas de los distintos componentes gráficos.

Parte 0: Uso de `.gitignore`

Para esta actividad está incluido el `.gitignore` que deben utilizar para poder ignorar correctamente la carpeta `frontend/assets` y el enunciado. Es muy importante asegurarse de que el `.gitignore` esté en el repositorio **antes** de realizar el primer *push* para que no se suban las imágenes y los archivos `.ui`.

Parte 1: Ventana de Inicio

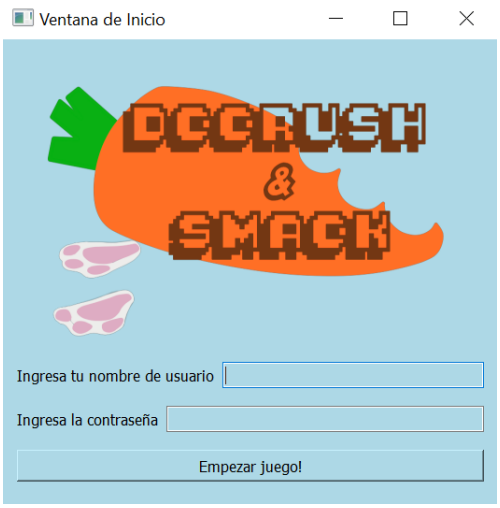
En esta primera parte, tendrás que implementar la ventana de inicio de DCCrush & Smack! Te dejamos libertad creativa pero debes incluir los siguientes elementos: el logo de DCCrush, un campo de texto para ingresar un nombre de usuario, un campo para poner la contraseña y un botón para ingresar al juego. Además debes implementar la lógica que permite comprobar que los datos ingresados sean correctos y conectar y emitir las señales para comunicar el *front-end* con el *back-end*.

Métodos de front-end

El archivo en donde deberás trabajar es `frontend/ventana_inicio.py`

- **Métodos ya implementados**
  - `def __init__(self)`: Inicializa la ventana de inicio y llama al método `crear_elementos(self)`. No debes modificarlo
- **Métodos que deberás implementar**
  - `def crear_elementos(self)`: Este método agrega todos los elementos visuales e interactivos a la ventana. Dentro de este debes crear: Debes modificarlo
    - Un `QLabel` que contenga al logo.
    - Un `QLabel` indicando al usuario que ingrese su nombre.
    - Un `QLineEdit` para ingresar el usuario.
    - Un `QLabel` indicando al usuario que ingrese la contraseña.
    - Un `QLineEdit` para ingresar la contraseña. Esta debe esconder la contraseña para que solo los jugadores exclusivos de DCCrash & Smack puedan entrar. Para hacer esto debes utilizar su método `setEchoMode` que recibe como argumento exacto `QlineEdit.Password`
    - Un `QPushButton` para enviar información de *login* y poder comenzar el juego. Debes conectar la señal `clicked` de este con el método `enviar_login`.

    Si quieres puedes usar *layouts* para que se vea bonito, pero basta con que incluyas los elementos pedidos. Aquí te dejamos un ejemplo de cómo se podría ver la pantalla (no tiene por qué ser igual).



- `def enviar_login(self)`: Este método emite la señal para que se realice la verificación de los datos de *login* en el *back-end*. Debes utilizar la señal `self.senal_enviar_login` para emitir una señal que contenga una tupla con el nombre de usuario y contraseña ingresada. Debes modificarlo
- `def recibir_validación(self, valid: bool, errores: list)`: Recibe un `bool` desde el *back-end* indicando el resultado de la validación (`True` si los datos cumplen con los requerimientos) y una lista de *strings* con posibles errores en caso que la validación no haya sido exitosa. Si la validación fue exitosa, entonces debes esconder la ventana de inicio. En caso contrario, debes indicar qué errores hubo con la verificación. Si `'Usuario'` está en la lista `errores` deberás avisar que el usuario ingresado es inválido. Si `'Contraseña'` está en la lista `errores` deberás avisar que la contraseña

es inválida. Para indicar el tipo de error deberás cambiar el texto del `QLineEdit`[1] del nombre de usuario o de la contraseña según corresponda. **Debes modificarlo**

Métodos de back-end

El archivo en donde deberás trabajar para esa parte es `backend/logica_inicio.py`.

- **Métodos ya implementados**
  - `def __init__(self)`: Inicializa la clase `LogicaInicio`. **No debes modificarlo**
- **Métodos que deberás implementar**
  - `def comprobar_usuario(self, tupla_respuesta: tuple)`: Este método recibe una tupla que contiene un `str` con el nombre de usuario y otro `str` con la contraseña ingresada. Primero deberás chequear si el usuario y contraseña son válidos, y luego deberás emitir la señal `self.senal_respuesta_validacion` que contenga un `bool` indicando si el *login* fue válido y una lista con los posibles errores de *login*. Si el nombre de usuario no es alfanumérico o el largo es mayor que `MAX_CARACTERES`[2], entonces el *login* es inválido y debes agregar `'Usuario'` a la lista de errores. Por otra parte, si la contraseña es distinta a `PASSWORD` el *login* es inválido y debes agregar `'Contraseña'` a la lista de errores. Por último, sólo si el *login* fue válido, debes emitir la señal `self.senal_abrir_juego` que contenga el nombre de usuario. Es importante que la señal `self.senal_respuesta_validacion` debe ser emitida independiente del resultado de la validación. **Debes modificarlo**

Señales

Deberás conectar las señales con los respectivos métodos en el archivo `main.py`

- **Señales de `VentanaInicio`**
  - `senal_enviar_login`: Esta señal envía una tupla con el nombre de usuario y contraseña. Debes conectarla con el método `comprobar_usuario` de la clase `LogicaInicio`. **Debes modificarlo**
- **Señales de `LogicaInicio`**
  - `senal_respuesta_validacion`: Esta señal envía un *bool* indicando el estado de la validación y una lista con los posibles errores. Debes conectarla con el método `recibir_validacion` de la clase `VentanaInicio`. **Debes modificarlo**
  - `senal_abrir_juego`: Esta señal envía un *string* con el nombre de usuario. Debes conectarla con el método `mostrar_ventana` de la clase `VentanaJuego`. **Debes modificarlo**

Parte 2: Ventana de juego

Luego de que se ingresan correctamente los datos en la ventana de inicio, se procederá a abrir la ventana de juego. En esta parte deberás hacer que se muestre correctamente la ventana de juego y completar los métodos necesarios para el funcionamiento del juego. En específico deberás crear *timers* y completar el movimiento del martillo para que se mueva según las teclas apretadas. Por último, deberás conectar y emitir las señales que permiten la comunicación entre el *front-end* y el *back-end*.

[1] Para esto puedes vaciar el campo de texto mediante el método `setText` de `QLineEdit` y luego indicar el error mediante el método `setPlaceholderText` de `QLineEdit`

[2] Las palabras en `ESTE_FORMATO` son parámetros que debes importar del archivo `parametros.py`

Nota: Es posible que los `labels` de la ventana de juego se vean superpuestos, esto se puede deber al escalado de Windows, por lo que se recomienda ponerlo en 100 % para la actividad.

Métodos de front-end

El archivo en donde deberás trabajar es `frontend/ventana_juego.py`

- **Métodos ya implementados**
  - `def init_gui(self):` Agrega el título a la ventana y conecta el botón de salir del juego. No debes modificarlo
  - `def actualizar_topos(self, topos: list):` Este método recibe una lista de objetos `Topo` y los incluye en la ventana, los hace visibles y los mueve a la posición correcta. No debes modificarlo
  - `def mover_martillo(self, martillo: Martillo):` Este método recibe un objeto `Martillo` y lo incluye en la ventana, lo hace visible y lo mueve a la posición correcta. No debes modificarlo
  - `def actualizar_datos(self, tiempo: str, puntaje: str):` Actualiza el tiempo y el puntaje en la ventana de juego. No debes modificarlo
  - `def salir(self):` Cierra la ventana. No debes modificarlo
- **Métodos que deberás implementar**
  - En primer lugar, deberás hacer que la clase cargue correctamente el archivo `ventana_juego.ui` (ubicado en la carpeta `frontend/assets`) que fue generado en *Qt Designer*. La ruta a este archivo se encuentra en el parámetro `RUTA_UI_VENTANA_JUEGO`. Debes modificarlo
  - `def __init__(self):` Inicializa la ventana de juego. Deberás llamar al método que inicializa la interfaz contenida en el archivo `.ui` generado con *Qt Designer*, y luego llamar al método `self.init_gui`. Debes modificarlo
  - `def mostrar_ventana(self, usuario: str):` Este método recibe el nombre de usuario y actualiza la ventana para mostrar correctamente la información inicial del juego. Primero deberás mostrar la ventana, mostrar el nombre de usuario en `self.casilla_nombre`, mostrar el puntaje inicial `PUNTAJE_INICIAL` en `self.casilla_puntaje` y mostrar el tiempo `TIEMPO_JUEGO` en `self.casilla_tiempo` (todas estas casillas son `QLabel`). Luego deberás emitir la señal `self.senal_iniciar_juego`. Debes modificarlo
  - `def keyPressEvent(self, event: QKeyEvent):` Envía señales al *back-end* cada vez que se presiona cualquiera de las teclas correspondientes a `TECLA_ARRIBA`, `TECLA_IZQUIERDA`, `TECLA_DERECHA`, `TECLA_ABAJO` (inicialmente estas corresponden a las teclas WASD, pero si lo prefieres las puedes cambiar en `parametros.py`). Para esto deberás emitir la señal `self.senal_tecla` entregándole como parámetro un `str` con la letra que indica la dirección a moverse (`"U"` para moverse hacia arriba, `"R"` para moverse a la derecha, `"L"` para moverse a la izquierda y `"D"` para moverse hacia abajo). Debes modificarlo

Métodos de back-end

El archivo en donde deberás trabajar para esa parte es `backend/logica_juego.py`.

- **Clase `Topo`: Métodos ya implementados**
  - `def __init__(self):` Inicializa la clase `Topo`. No debes modificarlo
  - `def start_timer(self):` Inicia el timer `timer_salir`. No debes modificarlo

- `def check_hit(self)`: Llama al método `toggle_hide` cada vez que aplastes a un topo. No debes modificarlo
- `def toggle_hide(self)`: En caso que el topo esté afuera se esconderá, en caso contrario saldrá de uno de los agujeros en la tierra. No debes modificarlo

- **Clase `Topo`: Métodos que deberás implementar**
  - `def instanciar_timer(self)`: En este método deberás instanciar dos `QTimer`. Primero deberás crear el *timer* `self.timer_salir` que hará que los topos aparezcan cada cierto tiempo. A este *timer* le debes asignar un intervalo de tiempo entre cada ejecución de `self.tiempo_salir` milisegundos. El segundo *timer* que deberás crear es `self.timer_afuera` que será el encargado de hacer que los topos desaparezcan. A este *timer* debes asignarle un intervalo de `self.tiempo_afuera` milisegundos. Además, ambos *timers* deben estar conectados al método `self.toggle_hide` y debes definirlos como *single-shot timers* [3]. Debes modificarlo
- **Clase `Martillo`: Métodos ya implementados**
  - `def __init__(self)`: Inicializa la clase `Martillo`. No debes modificarlo
  - `def reset(self)`: Vuelve al martillo a su posición inicial. No debes modificarlo
- **Clase `Martillo`: Métodos que deberás implementar**
  - `def mover(self, dir: str)`: Actualiza la posición del martillo dependiendo de la tecla que haya sido apretada y recibe como argumento un `str` que indica la dirrección del movimiento. De esta manera, si `dir` es igual a `'L'` deberás cambiar las coordenadas[4] de `self.pos_martillo` por (`MARTILLO_L_X`, `MARTILLO_L_Y`). Si `dir` es igual a `'R'` deberás cambiarlas por (`MARTILLO_R_X`, `MARTILLO_R_Y`). Si `dir` es igual a `'U'` deberás cambiarlas por (`MARTILLO_U_X`, `MARTILLO_U_Y`). Y si `dir` es igual a `'D'` deberás cambiarlas por (`MARTILLO_D_X`, `MARTILLO_D_Y`). Debes modificarlo
- **Clase `LogicaJuego`: Métodos ya implementados**
  - `def __init__(self, martillo: Martillo)`: Inicializa la clase `LogicaJuego`. No debes modificarlo
  - `def generar_topos(self)`: Crea cuatro instancias de `Topo`, uno para cada posición. No debes modificarlo
  - `def mover_martillo(self)`: Este método se llama cada vez que se aprieta una tecla para mover al martillo. Se encarga de mover al martillo y verificar si aplastó a algún topo o no. No debes modificarlo
  - `def reset_martillo(self)`: Llama al método `reset` de `Martillo` para que vuelva a su posición inicial. No debes modificarlo
  - `def actualizar_juego(self)`: Envía señales para actualizar distintos elementos de la ventana de juego. No debes modificarlo
  - `def terminar_juego(self)`: Envía una señal para cerrar la ventana de juego y una para abrir la ventana de postjuego. No debes modificarlo
- **Clase `LogicaJuego`: Métodos que deberás implementar**
  - `def instanciar_timer(self)`: En este método deberás instanciar tres `QTimer`. Primero deberás crear el *timer* `self.timer_juego` que hará que el juego termine después de un tiempo. A este *timer* debes asignarle un intervalo de `TIEMPO_JUEGO` milisegundos, conectarlo al método

[3] Para esto puedes utilizar el método `setSingleShot` de `QTimer`. Este método recibe como argumento un *bool* que indica si el *timer* es *single-shot*. Para más información puedes ir aquí

[4] Para cambiar las coordenadas puedes utilizar el método `moveTo` de `QRect`. Este método recibe como primer argumento la coordenada x y como segundo argumento la coordenada y a la cual se quiere mover el objeto `QRect`

`self.terminar_juego` y definirlo como un *single-shot timer*. El segundo *timer* que deberás crear es `self.timer_actualizar_juego` que será el encargado de actualizar elementos de la ventana de juego. A este *timer* debes asignarle un intervalo de `ACTUALIZAR_JUEGO` milisegundos y conectarlo al método `self.actualizar_juego`. El tercer *timer* que deberás crear es `self.timer_martillo` que hará que el martillo vuelva a su posición inicial luego de un tiempo. Debes asignarle un intervalo de `RESET_MARTILLO` milisegundos y conectarlo al método `self.reset_martillo`. Además, debes agregar cada uno de los tres *timers* a la lista `self.timers`. **Debes modificarlo**

- `def iniciar_juego(self)`: Este método inicia el puntaje del juego en cero, llama al método `self.generar_topos` e inicia los timers. Para completarlo, deberás iniciar cada uno de los timers guardados en la lista `self.timers`. **Debes modificarlo**

Señales

Deberás conectar las señales con los respectivos métodos en el archivo `main.py`

- **Señales de `VentanaJuego`**
  - `senal_iniciar_juego`: Esta señal inicia el funcionamiento del juego. Debes conectarla con el método `iniciar_juego` de la clase `LogicaJuego`. **Debes modificarlo**
  - `senal_tecla`: Esta señal envía un *string* con la dirección del movimiento del martillo. Debes conectarla con el método `mover_martillo` de la clase `LogicaJuego`. **Debes modificarlo**
- **Señales de `LogicaJuego`**
  - `senal_martillo`: Esta señal envía una instancia de `Martillo`. Debes conectarla con el método `mover_martillo` de la clase `VentanaJuego`. **Debes modificarlo**
  - `senal_actualizar`: Esta señal envía dos *string* con información del estado del juego. Debes conectarla con el método `actualizar_datos` de la clase `VentanaJuego`. **Debes modificarlo**
  - `senal_topos`: Esta señal envía una lista con instancias de `Topo`. Debes conectarla con el método `actualizar_topos` de la clase `VentanaJuego`. **Debes modificarlo**
  - `senal_termino_juego`: Esta señal envía un *string* con el puntaje del jugador. Debes conectarla con el método `abrir` de la clase `VentanaPostJuego`. **Debes modificarlo**
  - `senal_cerrar_ventana_juego`: Esta señal se envía para cerrar la ventana de juego. Debes conectarla con el método `salir` de la clase `VentanaJuego`. **Debes modificarlo**

Parte 3: Ventana Postjuego

Una vez se acabe el tiempo aparecerá una ventana con el puntaje del usuario y unos botones que le permiten volver a la ventana de inicio o salir del juego. No debes modificar ningun archivo aquí, solo debes asegurarte de conectar las señales correspondientes dentro del archivo `main.py`.

Señales

Deberás conectar las señales con los respectivos métodos en el archivo `main.py`

- **Señales de `VentanaPostJuego`**
  - `senal_abrir_inicio`: Esta señal se envía para abrir nuevamente la ventana de inicio y poder volver a jugar. Debes conectarla con el método `show` de la clase `VentanaInicio`. **Debes modificarlo**

- `senal_cerrar_juego`: Esta señal se envía para terminar la ejecución del programa. Debes conectarla con el método `exit` de la clase `QApplication`. Recuerda que ya creamos una instancia de `QApplication`, por lo que es importante que llames al método de esta instancia. **Debes modificarlo**

Notas

- Para la ventana de inicio recuerda que puedes poner *layouts* dentro de otros *layouts* para que quede más ordenado. Sin embargo, también recuerda que lo importante es mostrar los elementos pedidos y no es necesario que se vea como en el ejemplo.
- Recuerda que es buena práctica conectar todas las señales en el archivo `main.py`.

Requerimientos

- (2.00 pts) Parte 1: Ventana de inicio:
  - (0.75 pts) Completa correctamente el método `crear_elementos`
  - (0.75 pts) Completa correctamente el método `comprobar_usuario`.
  - (0.50 pts) Completa correctamente la conexión de señales
- (3.00 pts) Parte 2: Ventana de juego:
  - (0.50 pts) Carga correctamente el archivo `.ui`.
  - (1.00 pts) Implementar correctamente el método `keyPressEvent` de la clase `VentanaJuego` y el método `mover` de la clase `Martillo`.
  - (1.00 pts) Implementar correctamente los *timers* de la clase `Topo` y `LogicaJuego`.
  - (0.50 pts) Completa correctamente la conexión de señales
- (1.00 pts) Completa ventana de post-juego
  - (0.50 pts) Conectar señal para volver a jugar.
  - (0.50 pts) Conectar señal para volver a la ventana de inicio.