Paradigmas de Programación
Práctica II – Curso 2024/25

Pargammon

1. Definición y Objetivos

Esta práctica es una continuación de la práctica anterior en la que vamos a aprovechar las capacidades de las Interfaces Gráficas de Usuario (GUI) para crear una aplicación orientada a entorno de ventanas que implemente el juego descrito en la primera práctica. En el Campus Virtual se ha depositado una **solución de la primera práctica** adaptada a las nuevas condiciones que se indican a continuación, para que podáis enfocaros únicamente en la adaptación del juego a un entorno gráfico.

La librería que utilizaremos es **wxPython** (https://wxpython.org/). Las reglas del juego y la forma de juego son las mismas salvo en los siguientes aspectos:

- Ahora el número máximo de jugadores y de dados se establece a 4. Es posible diseñar la interfaz de usuario teniendo en cuenta estas limitaciones (creando, por ejemplo, cuatro controles `StaticBitmap` para representar los dados y ocultando los que no se necesiten).
- El número máximo de columnas y de fichas iniciales ahora **no está acotado**. La idea es que en este caso la interfaz se adapte **dinámicamente** al número de columnas y fichas que se indica para cada partida. **Nota**: No es necesario que os preocupéis por el hecho de que si se elige un número muy grande de columnas o fichas no se pueda visualizar todo el tablero.
- Se elimina la regla **[J3]** de la práctica anterior: Ahora es posible que en una jugada se salten todos los dados (no se realice ningún movimiento) independientemente de si hay otras jugadas posibles o no.
- En la práctica anterior solo se jugaba una única partida en cada ejecución, ahora es posible jugar un número ilimitado de partidas, hasta que se cierre la aplicación.
- Por último, en la práctica anterior en cada turno el jugador introducía la jugada completa. En un entorno gráfico la forma natural de proceder es mediante **movimientos**: El jugador va indicando cada movimiento de la jugada (haciendo clic en la columna deseada o en un botón "Saltar dado") individualmente hasta completar su jugada.

Los objetivos de la aplicación se van a clasificar en **genéricos** y **específicos**. Los objetivos genéricos indican propiedades generales de la aplicación que se pueden cumplir en mayor o menor medida y afectan a la nota en su conjunto. Los objetivos específicos indican requisitos concretos de la aplicación, y pueden ser obligatorios (necesarios para que se evalúe la práctica) u opcionales (si no se implementan se tiene una penalización).

2. Objetivos Genéricos

[G1] **Facilidad de uso, Interfaz gráfica e intuitiva**. Que muestre **en el entorno gráfico** el estado del juego en todo momento y que la disposición de controles para realizar las operaciones sea lo más cómoda y natural posible, adaptada a un entorno gráfico (por ejemplo, en un entorno gráfico es mucho más natural indicar un movimiento haciendo clic en la ficha o columna del tablero que introduciendo una letra en un cuadro de texto). Se puede suponer que el usuario conoce perfectamente el juego (no es necesario añadir tutoriales, guías visuales o indicaciones de acciones válidas).

[G2] **Retroalimentación.** Cada operación debe tener un efecto inmediato visible en todos los elementos de la interfaz gráfica a los que afecte.

[G3] **Robustez.** La aplicación debe tolerar errores del usuario (por ejemplo si intenta un movimiento no válido) indicando discretamente el error, y evitar en lo posible que el usuario cometa errores. Por ejemplo, si no tiene sentido el usar un control en un momento dado de la ejecución, ese control debería estar desactivado.

[G4] **Diseño e Interactividad.** Se considerarán de forma positiva la inclusión de elementos gráficos, animaciones y sonidos siempre que no superen los límites del buen gusto.

3. Objetivos Específicos

3.1. Obligatorios

Estos objetivos deben ser implementados en su totalidad para que la práctica sea evaluada. Por supuesto existen otros requisitos (que el juego se implemente correctamente, que no existan errores de ejecución, que se informe al usuario si un parámetro o movimiento no es válido, etc.) que se consideran evidentes y no se indican.

[F1] Se deben poder jugar tantas partidas como desee el usuario, pero antes del comienzo de cada partida debe ser posible el indicar el número de jugadores (entre 2 y 4), de columnas (≥ número de jugadores), de fichas iniciales (≥ 1) y de dados (entre 2 y 4). También se debe poder indicar **para cada jugador** si es Humano, Máquina Tonta o Máquina Lista.

[F2] Durante el desarrollo de una partida no debe ser posible el cambiar los parámetros del juego (número de jugadores, columnas, etc.). Existirá una forma de detener irreversiblemente la partida actual (típicamente un botón) para que se puedan cambiar (o no) los parámetros y comenzar una nueva partida.

[F3] Los jugadores (tanto humanos como automáticos) no indican jugadas completas, sino **movimientos**. Cuando un jugador ha realizado **D**[1] movimientos **correctos** se cambia el turno al siguiente jugador. Debe existir la posibilidad de indicar que se quiere saltar el dado (no realizar movimiento).

[F4] Se debe mostrar la información, para cada jugador, del **número de fichas que ha sacado del tablero** y de su **puntuación**[2]. Esa información debe actualizarse tras la realización de cualquier movimiento.

3.2. Opcionales

Estos objetivos son opcionales, si no se implementan se produce una penalización (sobre 10 puntos) y sobre la nota resultante se aplican las minoraciones del cumplimiento de los objetivos genéricos.

[O1] Ampliando los datos obtenidos en **[F1]**, para cada jugador se debe poder elegir un nombre y la imagen de las fichas con las que va a jugar **[Penalización: -1 pto]**.

[O2] Los movimientos se indican pulsando en elementos gráficos de la interfaz, no introduciendo la letra de la columna en un control de texto. **[Penalización: -3 ptos]**.

[O3] En la interfaz existe una lista de las jugadas realizadas (en el formato de la práctica anterior), y haciendo doble-click sobre una jugada de la lista **se deshacen las jugadas posteriores** a ella. **[Penalización: -2 ptos]**

[O4] El usuario puede averiguar si hay un movimiento válido o no desde una columna cualquiera (y saber cuál es la columna destino) al pasar el cursor del ratón sobre la columna o un control asociado a ella. **[Penalización: -3 ptos]**.

4. Criterios de Evaluación

Los alumnos depositarán el código de la práctica en una tarea del Campus Virtual de la asignatura. El depósito del código no proporciona ninguna calificación, es simplemente un requisito previo para unificar la evaluación de subgrupos y realizar comprobaciones de autoría. La calificación se obtendrá en la **defensa**, en la que se propondrá una **modificación** del código presentado.

Importante: La defensa se llevará a cabo en los ordenadores del aula, sin acceso a internet, y usando el entorno de desarrollo **IDLE** junto con el programa **wxGlade**.

[1] **D** es el número de dados.
[2] En el código de la solución de la primera práctica existe un método, `info_jugadores`, que proporciona esa información.

Se deben cumplir las siguientes **condiciones** para que la práctica **sea evaluada** (en caso contrario se calificará con un cero):

- La práctica se puede realizar individualmente o en pareja (dos alumnos).
- Si se realiza en pareja, es posible que la calificación de ambos sea distinta, según el desarrollo de la defensa.
- El código presentado debe haber sido desarrollado **en su totalidad** por el/los alumno(s), sin ayudas humanas o de IA's.
- En la realización de la práctica se puede usar la solución de la primera práctica que se publicará en el Campus Virtual, modificada de la forma en que creáis conveniente. El resto del código presentado debe haber sido desarrollado **en su totalidad** por el/los alumno(s) (salvo la parte generada por **wxGlade**, por supuesto).
- El código debe poder evaluarse sin errores durante la defensa. En el caso de errores triviales los alumnos deben ser capaces de corregirlos in situ inmediatamente.
- Durante la defensa se puede solicitar a los alumnos que **modifiquen su código** para implementar un cambio sencillo en las condiciones de la práctica. Para que se evalúe la práctica los alumnos deben ser capaces de realizar la modificación y obtener un código funcional.
- Las únicas librerías permitidas son **wx** (y sus sublibrerías), **math**, **time** y **random**. No son necesarias ni se permite el uso de ninguna otra librería. Si necesita usar alguna primero debe contar con la aprobación del profesor.

En la evaluación de la práctica se tendrán en cuenta, entre otros, los siguientes aspectos:

- La correcta resolución del problema y la implementación de los distintos aspectos mencionados en el enunciado.
- El uso de las técnicas impartidas en la asignatura.
- La documentación interna. Deben existir al menos comentarios que indiquen el propósito y la forma en que se debe usar cada función y clase del programa, así como los comentarios adecuados para cada parte del código donde se vaya a realizar alguna tarea no trivial.

5. Presentación y Evaluación de la práctica

La evaluación de la práctica se divide en dos etapas:

1. Presentación electrónica del fichero **practica2.zip**[3] que contiene el código de la práctica (incluyendo el o los ficheros de definición de interfaz de usuario de Glade (*.wxg) y el código solución de la primera práctica, **pargammon.py**, si se ha utilizado), junto con el resto de los ficheros (imágenes, etc.) necesarios para su ejecución. El fichero de código que sirve de punto de ejecución de la práctica se llamará **practica2.py**[3]. Se habilitará en el Campus Virtual de la asignatura una tarea de subida de ficheros cuya fecha límite será el **domingo 18 de mayo a las 23:59.** Al principio de todos los ficheros de código debe aparecer un comentario con el nombre de quienes la han desarrollado.

2. Evaluación **presencial**, en laboratorio, ante el profesor. Se realizará en el lugar, día y hora correspondiente al horario de prácticas del subgrupo al que pertenezca durante la semana del 19 al 22 de mayo.

[3] Debe tener exactamente ese nombre y extensión o no se corregirá.

Anexo I: Solución de la primera práctica

En el Campus Virtual se ha deposito un fichero Python con la solución de la primera práctica, que podéis utilizar en el desarrollo de la práctica actual. Su uso **no es obligatorio** pero si es recomendable porque se ha adaptado específicamente para que su aplicación a un entorno de ventanas sea lo más sencillo posible. En la solución se definen tres clases:

La clase auxiliar **JugPtos** sirve para representar una posible jugada y su puntuación asociada. Tiene tres atributos para poder acceder a la jugada (en formato vector y formato texto) y a la puntuación. Los métodos **jugada_azar** y **jugada_mejor** de la clase **Pargammon** devuelven un objeto perteneciente a esta clase:

Clase JugPtos	
Atributos	
jugada: [int]	Jugada en formato vector de enteros (índices de columnas)
str_jugada: str	Jugada en formato texto
ptos: int	Puntuación del jugador actual

La clase auxiliar **Columna** representa una columna de fichas del tablero. Almacena información sobre la posición de la columna en el tablero, el número de fichas que contiene y a que jugador pertenecen. Se le ha añadido la posibilidad de añadir un callback (una referencia a una función externa) que se llamará cada vez que cambie su número de fichas (esto puede ser muy conveniente si creamos nuestro propio control para representar gráficamente una columna, así podemos hacer que se redibuje automáticamente cuando cambie su contenido).

Clase Columna	
Atributos	
ind: int	Índice de la columna (0-based) en el tablero
jug: int	Índice del jugador (0-based) que tiene fichas en la columna, -1 si está vacía
num: int [property]	Número de fichas de la columna. Si se cambia este valor y *callback_activos* vale True se llama a la función almacenada en *callback* con este objeto columna como argumento.
callback: func	Función que se llamará cuando cambie el valor de *num* (si está asignada, claro)
callback_activos: bool	Flag para poder desactivar que se llame al callback (necesario para que el análisis recursivo de las jugadas válidas no provoque que se redibujen los gráficos de la aplicación)

La clase **Pargammon** representa el estado de una partida. Aunque se incluyen todos los atributos y métodos necesarios para la primera práctica, para la realización de la segunda solo son relevantes los que se indican en la tabla.

Clase Pargammon	
Atributos	
N: int	Número de columnas del tablero
M: int	Número inicial de fichas por jugador
D: int	Número de dados
J: int	Número de jugadores
turno: int	Índice (0-based) del jugador actual
dados: [int]	Valores de los dados
Acceso a las columnas (objetos de clase Columna)	Acceso indexado: Si por ejemplo `juego` es la variable que almacena el objeto Pargammon, al escribir `juego[i]` accedemos a la columna (objeto de clase Columna) *i*-ésima. También se pueden recorrer todas las columnas en un bucle: `for col in juego:` …

Métodos	
Constructor	Igual que en la primera práctica, con 4 parámetros: N, M, D y un string con los caracteres de las fichas. La longitud de ese string determina el número de jugadores.
cambiar_turno( ) **-> bool**	Comprueba si se ha terminado la partida y en caso contrario actualiza turno al siguiente jugador, genera una nueva tirada de dados y analiza las jugadas posibles. Devuelve un valor lógico indicando si se ha terminado la partida.
fin_partida( ) **-> bool**	Devuelve un valor lógico indicando si se ha terminado la partida.
haz_movim(col**: int**, desp**: int**) **-> str | None**	Intenta mover una ficha de la columna **col** a la columna **col + desp**. Si col = -1 no se realizan cambios pero cuenta como movimiento válido. Si el movimiento es válido se actualiza la información adecuada para deshacerle. Devuelve None si el movimiento es válido y un string con el mensaje de error si no lo es.
haz_jugada(jugada**: [int]**) **-> str | None**	**Atención**: Hay que llamar a este método con una lista vacía (`haz_jugada([])`) tras cada llamada a `cambiar_turno` (y al comienzo del juego)[4]. Aparte de eso no debéis usar este método.
deshacer(num**: int**)	Deshace las *num* jugadas anteriores.
info_jugadores( ) **-> [(int, int)]**	Devuelve una lista con una tupla de dos enteros por cada jugador. El primer valor de la tupla indica las fichas sacadas del tablero y el segundo la puntuación del jugador.
jugada_azar( ) **-> JugPtos**	Devuelve un objeto JugPtos que representa una jugada escogida al azar de entre las posibles
jugada_mejor( ) **-> JugPtos**	Devuelve un objeto JugPtos que representa la mejor jugada entre las posibles

Uso de callbacks: La clase Pargammon define 3 callbacks (referencias a funciones) para permitir que se ejecuten 3 de vuestras funciones cuando se produzca un movimiento de fichas en el juego (por efecto de una llamada a **haz_movim**, ya sea de un jugador humano o automático). La función asignada a **on_sacar** se llama cuando el movimiento provoca que una ficha salga del tablero, la función asignada a **on_captura** se llama cuando el movimiento captura una ficha contraria y la función asignada a **on_mover** se llama cuando es un movimiento normal (de una ficha a otra).

Estas funciones reciben como parámetros las columnas afectadas por el movimiento, y por lo tanto son muy útiles cuando queráis realizar acciones que (a) sean diferentes según el tipo de movimiento y (b) necesiten conocer las columnas afectadas, lo que suele corresponderse con la realización de **animaciones** al mover las fichas. A continuación se muestra un ejemplo de esquema de código que utiliza los callbacks (se supone que los métodos se encuentran dentro de la clase de la ventana principal):

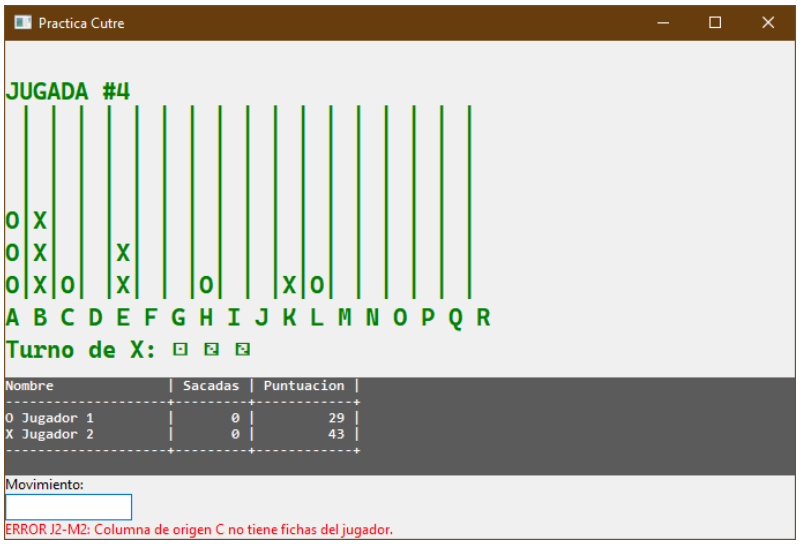
```
def crea_tablero(self):
  …
  self.juego = Pargammon(...)
  self.juego.on_sacar = self.on_sacar
  self.juego.on_movim = self.on_movim
  self.juego.on_captura = self.on_captura

def on_sacar(self, col: int):
  # Se llamará cuando se realice un movimiento de captura con origen columna col

def on_movim(self, col1: int, col2: int):
  # Movimiento normal de columna col1 a columna col2

def on_captura(self, col1: int, col2: int, col3: int):
  # Captura: Movimiento de columna col1 a columna col2 y la ficha contraria que
  # estaba en col2 se mueve a col3
```

[4] El motivo de tener que hacer esta llamada es el de actualizar la información para deshacer jugadas.

Anexo II: Ejemplos de Diseño

5.1. Aplicación minimalista

El ejemplo más sencillo de aplicación que podría conseguir algo de puntuación sería algo parecido a esto:

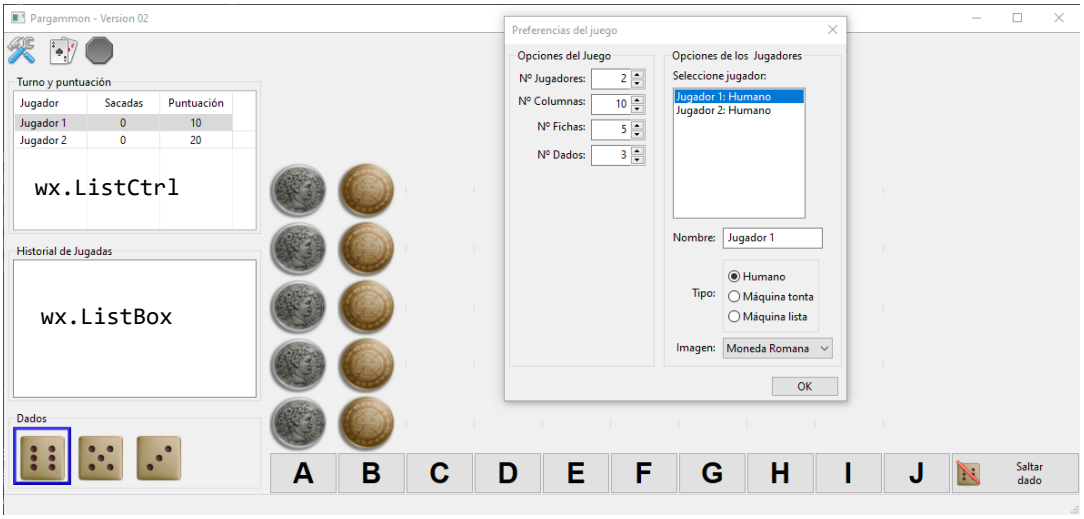


Todas las entradas (parámetros, tipo de jugadores y movimientos) se obtienen de un único control de texto (`TextCtrl`) y la partida, su información asociada, los mensajes de error y la información de que hay que introducir en el control se representan mediante etiquetas de texto (`StaticText`). El "dibujo" de la partida utiliza el método `__repr__()` de la primera práctica.

Si se implementa adecuadamente se pueden cumplir todos los requisitos obligatorios, pero no se cumplirían ninguno de los requisitos opcionales por lo que tendría una penalización de -9 y aunque se considerara que se han cumplido plenamente los objetivos genéricos (cosa que no es cierta) la nota no podría superar 1 punto sobre 10.

5.2. Aplicación basada en cuadrícula y botones

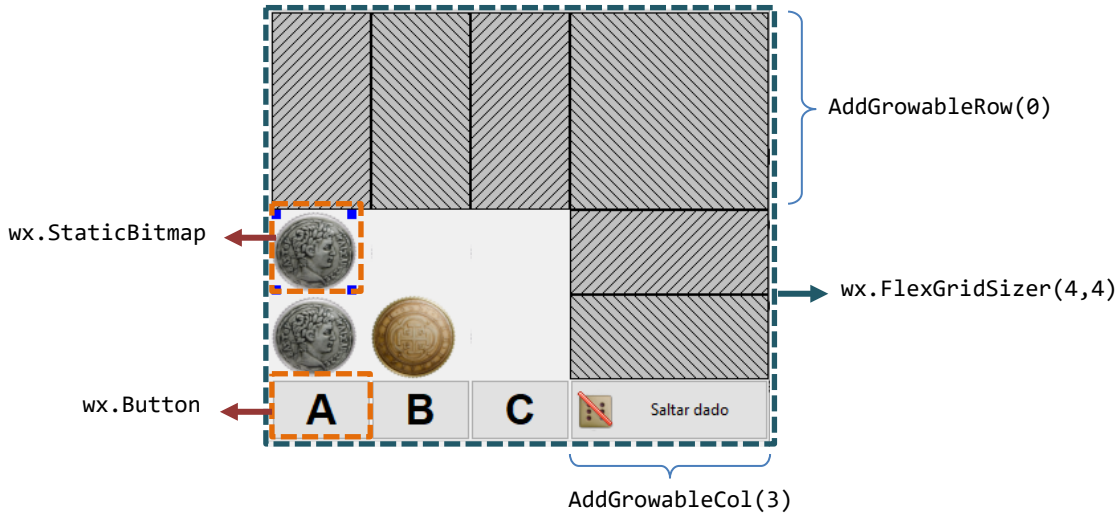
En esta aplicación se cumplen todos los requisitos obligatorios y opcionales, aunque respecto de los objetivos generales flojea en **[G1]** (el método elegido para cumplir los objetivos **[O2]** y **[O4]** no es intuitivo) y también en **[G4]** (no utiliza animación).



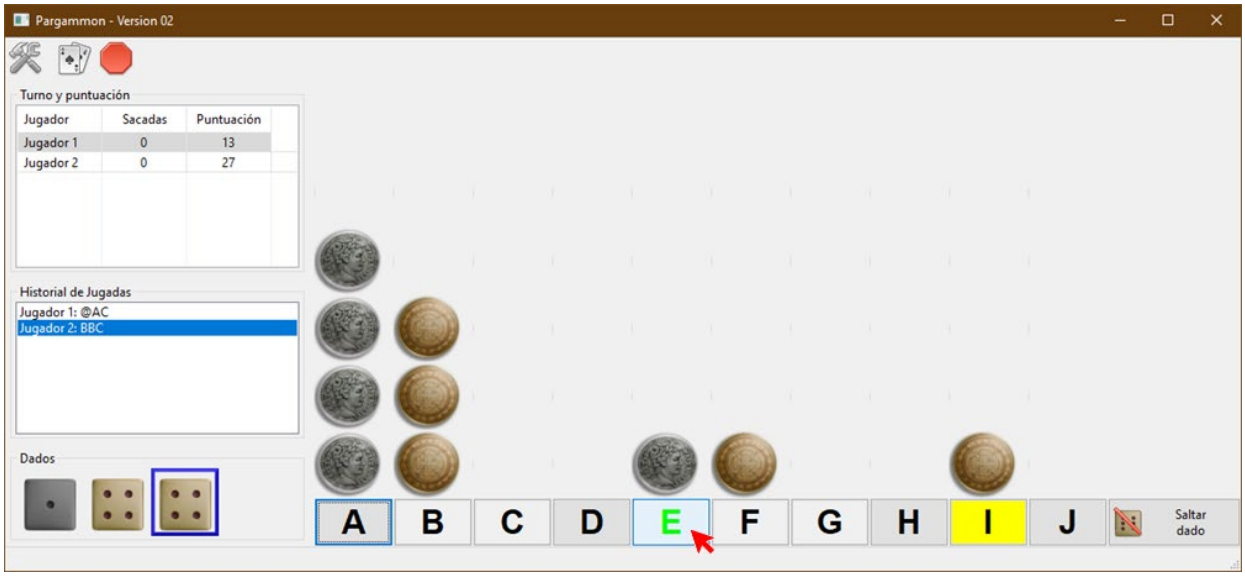
Las técnicas utilizadas son:

- Para la introducción de los parámetros de la partida se utiliza un diálogo **no modal**. Cualquier cambio en los parámetros cambia la visualización del tablero (esto se consigue pasando un callback al objeto diálogo).

- Para representar el tablero se utiliza un panel con un sizer de tipo cuadrícula (grid), y en cada celda se incluye un control de tipo imagen (`StaticBitmap`) o botón (`Button`) (la fila inferior, para poder indicar las columnas):



- Para cumplir el objetivo **[O4]** se añaden eventos EVT_ENTER_WINDOW y EVT_LEAVE_WINDOW a los botones, de forma que cuando el usuario pase el cursor sobre alguno de los botones que representan las columnas se pueda cambiar el color de primer plano de los botones para indicar si se puede realizar el movimiento y cual sería la columna de destino:



5.3. Aplicación basada en la creación de un control columna

Una posibilidad más sofisticada que cumple todos los requisitos (salvo el de animación) es la de definir nuestro propio control que represente una columna de fichas. Le asociamos a un objeto Columna por agregación o herencia de forma que pueda dibujar tantas fichas como tenga la columna y gracias a la posibilidad de tener un callback en los objetos de clase Columna, podemos automatizar el redibujado cuando cambie su contenido. Si lo hacemos heredar de `wx.Panel`[5] podemos incorporar los eventos EVT_ENTER_WINDOW, EVT_LEAVE_WINDOW y EVT_LEFT_DOWN para implementar **[O4]** y detectar que se ha pulsado en una columna.

El diseño del tablero es más sencillo ya que ahora no es necesario utilizar un sizer de tipo cuadrícula (`GridSizer` o `FlexGridSizer`) sino que podemos usar uno secuencial (`BoxSizer`) con el que es más sencillo la realización de modificaciones.

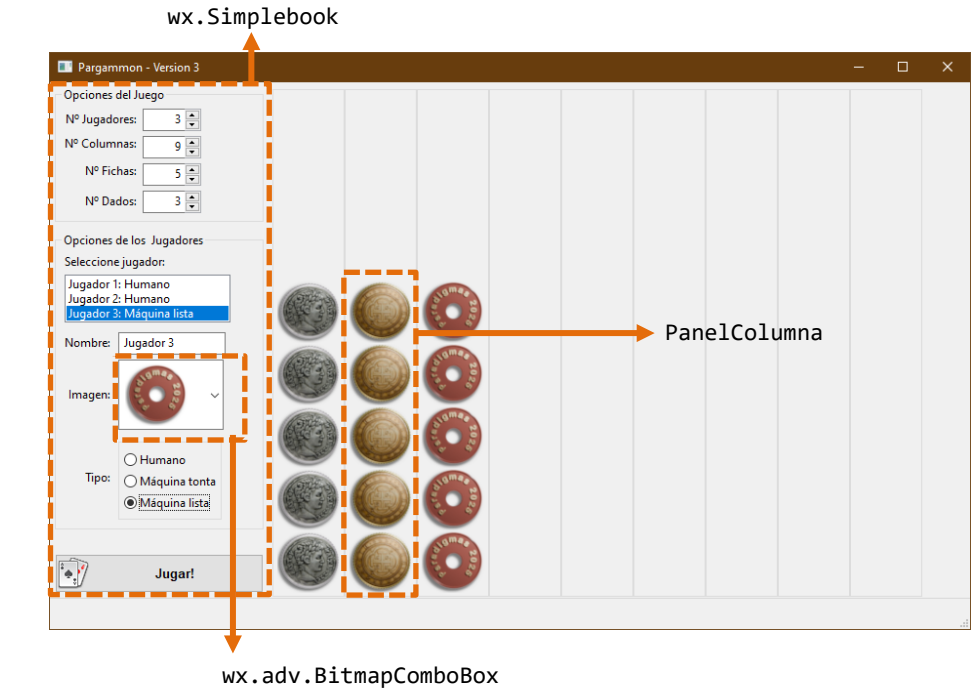
[5] Los controles `wx.StaticBitmap` no pueden recibir eventos de ratón.

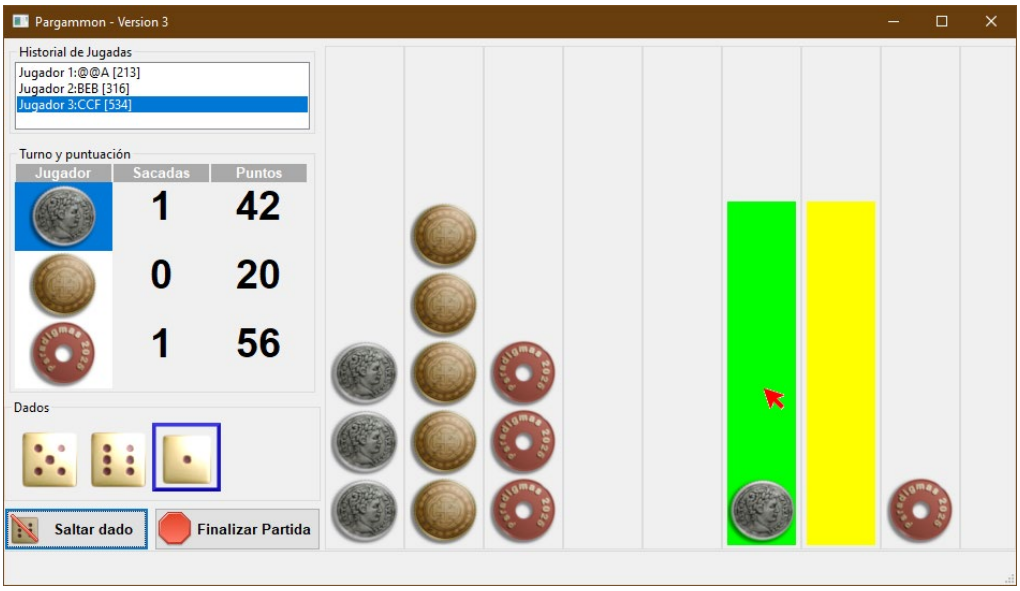
En este ejemplo no se utiliza un diálogo de preferencias, el panel izquierdo de la ventana principal es un `wx.Simplebook`, que permite crear varias *páginas* de controles y en el código podemos decidir cuál de ellas se muestra en un momento dado[6].

El esquema del nuevo control sería el siguiente:

```
class PanelColumna(wx.Panel):
    def __init__(self, parent, col: Columna, …):
        super().__init__(parent, wx.ID_ANY, size=…)
        self.col = col
        self.col.callback = self.on_cambio
        self.SetBackgroundStyle(wx.BG_STYLE_PAINT)
        self.Bind(wx.EVT_PAINT, self.on_paint)
        …

    def on_cambio(self, obj):
        self.Refresh()

    def on_paint(self, event):
        dc = wx.AutoBufferedPaintDC(self)
        dc.Clear()
        # … Dibujo de imágenes
        # con dc.DrawBitmap …
```

La apariencia de la aplicación sería:





[6] Glade no tiene en su paleta el control `Simplebook`, debemos usar un control de tipo `Notebook` (que tiene las pestañas visibles) y luego en el código cambiar la creación del objeto por `Simplebook` (con pestañas invisibles). No es necesario cambiar nada más, los métodos de ambos son idénticos.

5.4. Aplicación basada en control de tablero (con animación)

Si se quiere implementar la animación de las fichas entonces hay que ir un paso más respecto a la aplicación anterior y definir un control nuevo que represente a todo el tablero y dibujar en el todas las fichas, os podéis basar en el código del apartado anterior teniendo en cuenta que el único elemento de dibujo imprescindible es el método `dc.DrawBitmap(imagen,x,y)`.

Los tamaños del control y de las imágenes se pueden obtener con la propiedad `Size`, y será necesario añadir varios atributos que indiquen si se está realizando una animación y en ese caso en que etapa nos encontramos, que ficha se está moviendo y en que posición se debe dibujar. Se aconseja que el método `on_paint` realice un dibujado **completo** del estado del tablero en todas las circunstancias (incluyendo cuando se realiza una animación)[7].

Información relevante:

- Solo se puede dibujar en el método `on_paint`, y este método no se puede ejecutar directamente. Si intentáis llamarlo en cualquier parte de vuestro código veréis que se genera un error en la línea `dc = wx.AutoBufferedPaintDC(self)`. El motivo del error es que en un entorno gráfico no puedes dibujar libremente en la pantalla, solo se puede hacer cuando el controlador envía el mensaje adecuado y se ejecuta el manejador asociado (en este caso el mensaje es EVT_PAINT).
- Los mensajes EVT_PAINT se generan automáticamente cuando el controlador determina que debe redibujarse el control (por ejemplo si se mueve una ventana que lo ocultaba y ahora pasa a estar visible). También podemos hacer que se genere llamando al método `Refresh()` del control.

Hay que tener en cuenta la forma en que trabaja un entorno orientado a eventos para poder implementar una animación. Supongamos que queremos crear un control al que se le da una lista de imágenes y queremos que cuando se pulse en él vaya mostrándolas en sucesión con un intervalo de 0.1 segundos entre cada una:

```
class MiControl(wx.Panel):
    def __init__(self, parent, imgs: [wx.Bitmap]):
        super().__init__(parent, wx.ID_ANY, …)
        self.SetBackgroundStyle(wx.BG_STYLE_PAINT)
        self.imgs = imgs   # Lista de imágenes
        self.ind_img = 0   # Indice de la imagen que se dibuja
        self.Bind(wx.EVT_PAINT, self.on_paint)
        self.Bind(wx.EVT_LEFT_DOWN, self.on_lclick)

    def on_paint(self, event):
        dc = wx.AutoBufferedPaintDC(self)
        dc.Clear()
        dc.DrawBitmap(self.imgs[self.ind_img], 0, 0)

    def on_lclick(self, event):
        for i in range(len(self.imgs)):
            self.ind_img = i
            self.Refresh()
            time.sleep(0.1)
```

Si ejecutáis el código (con una lista de por ejemplo 20 imágenes) veréis que tras pulsarle no pasa nada durante 2 segundos (y además el control está bloqueado) y después muestra únicamente la última imagen. El motivo es el siguiente: Dentro del bucle, cada llamada a `Refresh` genera un mensaje wx.EVT_PAINT que **se añade a la cola de mensajes** de la aplicación pero al estar la aplicación ocupada con la ejecución del bucle estos mensajes se quedan esperando en la cola. Al terminar el bucle, se van procesando los 20 mensajes[8], cada uno de ellos genera una llamada a `on_paint` pero en todas ellas la variable `ind_img` tiene el mismo valor (el asignado en la última iteración del bucle).

La manera habitual de resolver este problema es usar un **temporizador**. Un temporizador genera mensajes wx.EVT_TIMER a intervalos prefijados, y si en su manejador llamamos a `Refresh` entonces en la cola de mensajes se irán intercalando eventos

[7] La otra posibilidad (el dibujar únicamente los cambios – dibujo basado en *sprites*) no es fácil de realizar en wxPython, habría que usar pyGame o similar.

[8] Existen entornos que "colapsan" los eventos de redibujado consecutivos de la cola en uno solo, ignoro si wxPython actúa de esa forma, pero el efecto final es el mismo.

EVT_TIMER y EVT_PAINT que, al no estar bloqueada la aplicación, se irán procesando adecuadamente. El código quedaría así (solo se muestran los cambios):

```
def __init__(self, parent, imgs: [wx.Bitmap]):
    …
    self.timer = wx.Timer(self)
    self.Bind(wx.EVT_TIMER, self.on_timer, self.timer)

def on_lclick(self, event):
    self.ind_img = -1
    self.timer.Start(100)

def on_timer(self, event):
    self.ind_img += 1
    if self.ind_img >= len(self.imgs):
       self.timer.Stop()
       self.ind_img = 0
    else:
       self.Refresh()
```

Otro posible enfoque es el de crear un evento personalizado que represente una etapa de la animación. En wxPython podemos crear nuevos eventos con la función `wx.lib.newevent.NewEvent()`, que devuelve una tupla con la clase que representa el nuevo evento (para que podamos crear objetos que lo representen) y un identificador del nuevo mensaje. Si usamos ese método el código sería:

```
def __init__(self, parent, imgs: [wx.Bitmap]):
    …
    self.animando = False  # Flag para saber si hay animación o no
    self.EventoAnimacion, self.EVT_ANIMACION = wx.lib.newevent.NewEvent()
    self.Bind(self.EVT_ANIMACION, self.on_animacion)

def on_paint(self, event):
    …
    if self.animando:
       time.sleep(0.1)  # Retardo
       # Generamos el evento para la siguiente etapa de la animación
       wx.PostEvent(self, self.EventoAnimacion())

def on_lclick(self, event):
    self.ind_img = -1
    self.animando = True
    self.Refresh()

def on_animacion(self, event):
    self.ind_img += 1
    if self.ind_img >= len(self.imgs):
       self.animacion = False
       self.ind_img = 0
    else:
       self.Refresh()
```

Anexo III: Guía rápida de wxPython

La página web https://docs.wxpython.org/wx.1moduleindex.html contiene una referencia de todos los elementos de wxPython, y también podemos obtener información muy útil examinando el código que genera wxGlade. La tabla siguiente proporciona una referencia rápida a todos los métodos que necesitamos conocer para la realización de la práctica:

Elemento	Clases	Acciones	Código
Color	`wx.Colour`	Creación	`wx.Colour(R: int, G: int, B: int)`
Imagen	`wx.Bitmap`	Creación	`wx.Bitmap(fich: str)`
		Obtener tamaño	`lx, ly = img.Size`
Sonido	`wx.adv.Sound`	Creación	`wx.adv.Sound(fich: str)`
		Reproducción	`sonido.Play()`
Todos los controles		Mostrar / ocultar	`ctrl.Show(mostrar: bool)`
		Actualizar	`ctrl.Refresh()` `ctrl.Parent.Layout()`
		Foco de teclado	`ctrl.SetFocus()`
Controles con texto	`wx.StaticText` `wx.Button`	Cambiar texto	`ctrl.Label = txt`
Controles con imágenes	`wx.StaticBitmap` `wx.Button`	Cambiar imagen	`ctrl.SetBitmap(img: wx.Bitmap)`
Controles booleanos	`wx.Chekbox`	Leer/cambiar valor	`ctrl.Value : bool`
Controles enteros	`wx.SpinButton`	Leer/cambiar valor	`ctrl.Value : int`

Controles selección	`wx.RadioBox`
	`wx.ComboBox`
	`wx.ListBox`
	`wx.adv.BitmapComboBox`
		Leer/cambiar selec.	`ctrl.Selection : int`

		Número elementos	`ctrl.Count : int`
Controles lista	`wx.ListBox`	Eliminar elems.	`ctrl.Clear()`
		Añadir lista elems.	`ctrl.Append(lis: [string])`
		Añadir elemento	`ctrl.Append(txt: str)`
		Cambiar elemento	`ctrl.SetString(fil: int, txt: str)`
		Borrar elemento	`ctrl.Delete(fil: int)`
	`wx.ListCtrl`	Eliminar filas	`ctrl.DeleteAllItems()`
		Añadir fila	`ctrl.Append(tupla strings)`
		Cambiar elemento	`ctrl.SetItem(fil,col: int, txt: str)`
		Seleccionar fila	`ctrl.Select(fil: int)`
Contenedor multiple	`wx.Notebook` `wx.Simplebook`	Cambiar página	`ctrl.SetSelection(pagina: int)`

Temporizador	`wx.Timer`	Creación	`self.timer = wx.Timer()`
			`self.Bind(wx.EVT_TIMER,`
			`self.on_timer,`
			`self.timer)`

		Comienzo	`self.timer.Start(miliseg: int)`
		Detención	`self.timer.Stop()`

Nota final: wxPython parece adoptar un enfoque "perezoso" a la hora de ajustar la interfaz cuando se añaden, eliminan o se realizan cambios en el contenido de los controles. Para evitar problemas, hacer lo siguiente:

- Si cambiáis la fuente, el color de fondo o de primer plano de algún control, justo después llamad a *`control`*`.Refresh()`.
- Al añadir o eliminar controles del panel de juego, llamad justo después a *`panel_juego`*`.Layout()`. Si con esto no funciona, usad *`panel_juego`*`.Parent.Fit()` y justo después *`ventana_principal`*`.Layout()`
- Si tenéis un control de texto centrado y veis que al actualizar el texto ya no está centrado, ejecutad *`control`*`.Parent.Layout()`.