**Tarea 1: Revisar Gloss Familiarízate con la biblioteca Gloss para el desarrollo de gráficos 2D y aplicaciones interactivas en Haskell. Debes estudiar los conceptos fundamentales incluyendo: tipos básicos (Picture, Color, Display), transformaciones geométricas (Translate, Rotate, Scale), los tres modos de visualización (display, animate, play), y el sistema de manejo de eventos de teclado y ratónV**

Gloss es una biblioteca de Haskell para crear gráficos 2D y animaciones.

**Tipos básicos:**

-Picture: Representa una imagen o figura 2D.

| **Constructor** | **Parámetros** | **Descripción** | **Ejemplo de uso** | **Resultado visual** |
| --- | --- | --- | --- | --- |
| Blank | — | Imagen vacía (no dibuja nada). | Blank | No se muestra nada. |
| Polygon | [(Float, Float)] | Polígono definido por sus vértices. | Polygon [(0,0),(50,100),(100,0)] | Triángulo. |
| Line | [(Float, Float)] | Línea formada por varios puntos unidos. | Line [(0,0),(100,100)] | Línea diagonal. |
| Circle | Float | Círculo relleno con radio dado. | Circle 80 | Círculo de radio 80. |
| ThickCircle | Float (radio), Float (grosor) | Círculo hueco o anillo. | ThickCircle 60 10 | Anillo de radio 60, grosor 10. |
| Arc | Float (inicio), Float (fin), Float (radio) | Arco entre dos ángulos (grados). | Arc 0 180 80 | Semicírculo. |
| Text | String | Dibuja texto en pantalla. | Text "Hola Gloss" | Muestra texto. |

-Color:Define el color de una figura (Picture).

* Colores predefinidos: white, black, red, green, blue, yellow, orange, purple, grey, etc.
* Colores personalizados: makeColor r g b a, donde cada componente va de 0 a 1.

-Display**:** Indica cómoy dónde se mostrará la ventana.

| **Constructor** | **Ejemplo** | **Descripción** |
| --- | --- | --- |
| InWindow | InWindow "Título" (ancho, alto) (x, y) | Ventana normal. |
| FullScreen | FullScreen | Pantalla completa. |
| InPlace | (raro de usar) | Renderizado sin ventana. |

**Transformaciones geométricas y combinación de imágenes:**

Las transformaciones permiten modificar o combinar figuras ya existentes.

| **Función** | **Parámetros** | **Descripción** | **Ejemplo de uso** | **Efecto visual** |
| --- | --- | --- | --- | --- |
| Translate | Float (x), Float (y), Picture | Desplaza una figura horizontal y verticalmente. | Translate 100 50 (Circle 30) | Mueve el círculo 100 px a la derecha y 50 px hacia arriba. |
| Rotate | Float (grados), Picture | Rota la figura un ángulo (en grados, sentido antihorario) respecto al origen (0,0). | Rotate 45 (RectangleSolid 100 50) | Gira el rectángulo 45°. |
| Scale | Float (x), Float (y), Picture | Escala la figura en el eje X e Y. | Scale 2 0.5 (Circle 40) | Estira el círculo al doble de ancho y la mitad de alto. |
| Pictures | [Picture] | Combina varias figuras en una sola imagen. | Pictures [Circle 50, RectangleWire 100 100] | Pictures [Circle 50, RectangleWire 100 100] |

\*\* Las transformaciones se pueden combinar aplicándose de aplican de derecha a izquierda.

Ejemplo: figura1 = Translate 100 0 $ Rotate 45 $ Circle 30

**Modos de visualización**

-Display: crea una imagen estática.

-Animate: crea animaciones basadas en el tiempo. La función recibe el tiempo (en segundos) y devuelve un Picture distinto en cada frame.

-Simulate: permite mantener y actualizar un estado

**Play**

Permite crear aplicaciones interactivas, con eventos de teclado/ratón y actualizaciones del estado cada frame.

play

:: Display -- Tipo de ventana

-> Color -- Color de fondo

-> Int -- Número de frames por segundo (FPS)

-> world -- Estado inicial del juego

-> (world -> Picture) -- Función de dibujo (renderizado)

-> (Event -> world -> world) -- Función de manejo de eventos

-> (Float -> world -> world) -- Función de actualización del estado

-> IO ()

Un evento representa una acción que puede cambiar el estado del programa.

-EventKey: evento de teclado y ratón. Para describir el contenido de un EventKey, Gloss usa otros tipos:

-Char:Tecla de carácter alfanumérico.

-SpecialKey: Tecla especial del teclado.

-MouseButton: Botón del ratón.

Tarea 2: Desarrollar un juego simple con monigote Implementa un juego básico usando Gloss donde aparezca un personaje (monigote) que se pueda controlar mediante el teclado. El monigote debe poder desplazarse horizontalmente (izquierda/derecha) usando las teclas A/D o las flechas del teclado, y realizar un salto con física realista al presionar W o flecha arriba. El juego debe incluir: un tipo de datos para representar el estado del juego (posición, velocidad, dirección), una función de dibujo que represente gráficamente al personaje de forma lateral o frontal, un manejador de eventos que capture las pulsaciones del teclado, y una función de actualización que aplique física básica (gravedad, detección de suelo, límites de pantalla). El personaje debe animarse al caminar y voltear su orientación según la dirección del movimiento. Ver video de ejemplo en la carpeta de la tarea.

**Explicación detallada (punto por punto) — para presentar y estudiar**

**1) Organización general**

* main usa play de Gloss, que necesita:
  + Display (config. ventana),
  + color de fondo,
  + FPS,
  + estado inicial,
  + función draw :: estado -> Picture,
  + manejador de eventos Event -> estado -> estado,
  + función de actualización Float -> estado -> estado (dt en segundos).
* Separé claramente **estado**, **dibujo**, **eventos** y **actualización** (esto es buena práctica).

**2) Tipo GameState**

Contiene:

* posX, posY: posición del monigote.
* velX, velY: velocidades en x e y.
* facing: orientación (izq/der), usada para voltear el dibujo.
* onGround: para detectar si puede volver a saltar.
* leftDown, rightDown: flags para saber qué teclas horizontales están presionadas (permite mantener pulsación).
* animTimer: temporizador para la animación de pasos.

Esto hace que el estado del juego sea explícito y fácil de razonar.

**3) Eventos de teclado (handleEvent)**

* Detecta KeyDown y KeyUp para:
  + A/Left (izquierda): leftDown
  + D/Right (derecha): rightDown
  + W/Up: salto (onGround -> aplica velY = jumpImpulse)
* Cuando pulsas la tecla de salto (un Down), si onGround es True se aplica velY = jumpImpulse y onGround = False.

Por qué guardamos leftDown/rightDown:

* Permite movimiento suave y detectar cuando la tecla es liberada (importantísimo para juegos).
* Si solo respondieras a Down y movieras la posición directamente, el movimiento sería por pasos y no continuo.

**4) Física en update dt**

* targetVX = velocidad horizontal deseada según teclas.
* newVx se actualiza suavemente con una aproximación velX += (target - velX) \* factor \* dt — esto simula aceleración y evita cambios instantáneos bruscos.
* velY recibe gravity \* dt (velocidad cambia con la gravedad).
* Se integran posiciones: x += vx \* dt, y += vy \* dt.
* Colisión con suelo: si y <= groundY, forzamos y = groundY y vy = 0, onGround = True.
* Limitamos x para que el personaje no salga de la pantalla.

Parámetros ajustables (constantes arriba):

* gravity, jumpImpulse, moveSpeed, vxResponse permiten ajustar la "sensación" del salto y movimiento.

**5) Dibujo y animación**

* drawGame construye un Picture usando Translate, Scale, rectangleSolid, circleSolid, etc.
* Para el volteo usamos Scale (-1) 1 cuando el facing == FaceL.
* Animación de caminar: animTimer aumenta cuando el personaje está en el suelo y se mueve, y usamos sin(t\*8) para mover piernas (y brazos ligeramente). Esto da apariencia de paso simple sin sprites.
* Sombra y colores para legibilidad.

**6) Orientación (flip)**

* Si facing == FaceL aplicamos Scale (-1) 1 a todo el Picture — así el monigote se refleja horizontalmente.
* También actualizamos facing cuando hay una dirección de movimiento activa (comparando targetVX).

**7) ¿Cómo funciona el salto "realista"?**

* No hay teletransporte: al saltar se establece velY = jumpImpulse. En cada frame velY se reduce por gravity\*dt. La posición y se integra con vy.
* Al llegar al suelo se detiene la caída. Esto produce una trayectoria parabólica clásica.

**8) Posibles extensiones (ideas para exponer)**

* Añadir animaciones con sprites (usar gloss-juicy para cargar imágenes).
* Añadir colisiones con plataformas (varias alturas).
* Añadir aceleración/ fricción más realista (coeficiente de fricción).
* Mejorar el sistema de entrada con combinación de teclas (por ejemplo, permitir sostener tecla mientras se pulsa otra).

**9) Notas para la exposición (puntos clave)**

* Explica la separación estado / dibujo / eventos / update.
* Muestra el GameState y por qué cada campo es necesario.
* Señala cómo el handleEvent distingue Down y Up (esto es importante para controles continuos).
* Explica la integración de la física (vx, vy, gravity, dt).
* Demuestra la animación: compara animTimer > 0 (caminando) vs 0 (quieto).

**Instrucciones para compilar & ejecutar (rápido)**

1. Asegúrate de tener gloss instalado:
   * con cabal: cabal update && cabal install gloss
   * o usando stack añade gloss a las dependencias del proyecto.
2. Guardar el archivo como Monigote.hs.
3. Ejecutar:
   * runhaskell Monigote.hs
   * o compilar: ghc Monigote.hs -o Monigote && ./Monigote

Tecla útiles:

* A / flecha izquierda: mover izquierda
* D / flecha derecha: mover derecha
* W / flecha arriba: salto

Tarea 3: Implementar Applicative y usar fmap Refactoriza los tipos de datos de tu juego HAUS para que implementen la clase de tipos Applicative (y por tanto también Functor). Identifica en tu código todas las oportunidades donde se puedan usar las funciones fmap (o su operador <$>), <\*>, y pure para hacer el código más declarativo y funcional. Esta tarea se centra en aplicar los conceptos de functores aplicativos que se impartirán en la próxima clase. Debes modificar las funciones existentes para aprovechar estas abstracciones cuando sea posible, por ejemplo al actualizar el estado del juego, transformar coordenadas, o aplicar funciones a valores dentro de contextos. La implementación debe respetar las leyes de Functor y Applicative. Nota Esta tarea requiere comprender los conceptos que se enseñarán en la próxima clase sobre Functores y Applicatives. Se recomienda revisar material previo sobre estas type classes si deseas adelantar trabajo.

Las funciones sobre operaciones matemáticas o creación de los diferentes tipos no se han modificado, solo se han cambiado aquellas funciones que devuelven un Maybe x o IO x.

En Bot.hs:

* botEjemplo:

Devuelve Just (DispararA id) si el carro ve algún enemigo (toma el primero), o Nothing si no ve a nadie.

* botCombinado:

Combina las acciones de atacar con moverse. Si no hay acción de atacar devuelve Nothing, pero si hay, se obtiene una lista combinada de Just [DispararA id, Mover (1,0)].

En Main.hs:

* ataqueInstantaneo:

Como parámetros damos el id de un proyectil, un carro atacante y otro que va a ser atacado. Como salida, devuelve un par: el carro atacante después de realizar el disparo (normalmente lo único que se va a ver alterado va a ser su lista de municiones, que va a tener una menos) y un Maybe con el carro atacado / objetivo (Nothing si no se consiguió atacarle, Just CarroCombate si se le aplica daño y posibilidad de morir un tripulante).

* procesarDisparo:

Genera un proyectil, actualiza al atacante, aplica el impacto directo al objetivo y calcula el daño realizado. Devuelve un Maybe por si alguna etapa falla y no se realiza el disparo finalmente. El uso del **do** en esta función sirve para que, si alguna de las líneas de la función devuelve Nothing, toda la función devolverá Nothing. En caso contrario, si todas son Just x, extrae los valores y sigue.

En Unidad.hs:

* aplicarDanioConMuerteAleatoria:

Con esta función reducimos la vida de un CarroCombate, se decide aleatoriamente si muere algún tripulante por el impacto (lo hace la función matarTripulanteAleatorioSiDanio) y, en caso de que muera alguno, se modifican los atributos según cuál tripulante haya muerto.

* buscarMunicionPreferida:

Tiene como parámetros un tipo de munición, un CarroCombate y devuelve el índice, dentro de la lista municiones del CarroCombate del tipo de munición especificado, que tiene mayor calibre. Si no hay ninguna de ese tipo, devuelve Nothing.

* elegirMunicionPara:

Elige qué munición usa el atacante contra el objetivo. En un primer caso se busca la mejor munición de tipo AP que tiene el atacante. Por el uso del **do**, si no se obtiene ninguna munición del tipo AP, se devuelve Nothing y la función termina. En caso de que sí haya encontrado una munición del tipo AP, calcula si la penetración de esa munición será mayor al blindaje del objetivo: en caso positivo, devuelve el índice de la munición; en caso negativo, busca el índice de la mejor munición de tipo AE (también puede devolver el índice o Nothing si no hay de este tipo).

* dispararA:

Esta función crea un proyectil y actualiza el estado del carro atacante (quitándole la munición usada). Igual que en las otras funciones, el **do** evita que sí una línea produce Nothing, la función termine y devuelva ese Nothing. Por lo que si idx es Nothing, la función termina ahí, evitando la creación del proyectil, que daría error.

* aplicarImpactoDirecto:

Aplica todo el daño de una munición a un CarroCombate. Primero valida que el objetivo tiene una energía mayor a 0, si no es así, la función termina devolviendo un Nothing. Si tiene más energía que 0, se aplica calcularNuevaEnergia.

* matarTripulanteAleatorioSiDanio:

Se le da como parámetros un daño y una tripulación. Si no hay tripulación que esté viva, devuelve Nothing. Si el daño es menor o igual que 0 devuelve la tripulación como estaba. Si hay tripulación viva, se elige un índice aleatorio entre esos tripulantes vivos y se elimina al tripulante del índice seleccionado.

* mostrarVisionDe:

Imprime por pantalla para cada carro del mundo una línea indicando los carros que pueden ver. Se usa mapM\_ para que se aplique la función printVision a todos los carros del mundo. También se usa map para coger los tiposCarro de todos los elementos de la lista vistos.

En Collision.hs:

* detectRobotProjectileCollisions:

Su objetivo es identificar los impactos que ocurren en un momento dado de la simulación y devolver una lista de eventos [CollisionEvent] que describen esas colisiones. Para cada carro, calcula con qué proyectiles ha colisionado, y concatena todos los resultados en una lista. Se usa el mapMaybe para que devuelva un Just evento si colisiona un carro con un proyectil, o un Nothing si no lo hace. Los Nothing en el mapMaybe se descartan y no se añaden a la lista. Luego, concatMap concatena todas las listas obtenidas en una sola.

* detectRobotRobotCollisions:

Igual que la función anterior, pero para comprobar la colisión entre dos carros. Ahora concatMap aplica todo lo anterior a cada tupla dada, que cada tupla será un carro c1 y una lista de carros rest siguientes que no sean c1. Por ejemplo, en una lista carros = [c1, c2, c3], las tuplas que se obtendrán son: (c1, [c2, c3]), (c2, [c3]).