# Informe Técnico MakyBotcho Juego de aprendizaje de programación

Edilson Bonet Mamani Yucra, Wilson Isaac Mamani Casilla, Jose Alejandro Machaca Muñiz
Roxana Flores Quispe
*Escuela Profesional de CIencia de la Computación, Universidad Nacional de San Agustín*
*Arequipa, Perú*
emamaniyu@unsa.edu.pe
wmamani@unsa.edu.pe
jmachacamu@unsa.edu.pe

1. **Creación del Mapa:**
   - Se diseñó el mapa utilizando una matriz que representaba la disposición de los elementos del juego.
   - Los sprites fueron utilizados para dar una apariencia tridimensional al mapa.
   - Se añadieron bloques para obstaculizar el movimiento del robot, creando un entorno desafiante.
   - En la matriz descrita, el valor 0 representa un bloque por el cual el auto puede pasar libremente, sin obstáculos. Un valor de -1 indica un bloque de llegada, que probablemente señala el objetivo final o un punto de destino en el entorno del juego. Los valores mayores o iguales a 1 representan bloques de pared, cuya altura o tamaño aumenta proporcionalmente con el valor numérico; es decir, a mayor valor, mayor es el tamaño del bloque de pared.

```
const int mapas[5][8][8] = {
   {
      {0, 0, 0, 0, -1, 0, -1, 0},
      {0, 0, 0, 0, 0, 0, 5, 0},
      {1, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 3, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 6},
      {0, 0, 0, 0, 7, 0, 0, 0},
      {-1, -1, 0, 1, 1, 1, -1, -1}
   },
   // …
}
```

2. **Definición de la Meta:**
   - Se estableció una meta en el mapa, representada por un piso de color azul. El objetivo del robot es alcanzar esta meta para avanzar al siguiente nivel. Dentro de la matriz este esta definida con la clave -1
3. **Implementación de un Minimapa:**
   - Se creó un minimapa en 2D para que el usuario pueda visualizar la posición del personaje, los bloques y la meta. Esto es especialmente útil ya que los efectos 3D pueden ocultar al personaje principal.
4. **Simulación de Movimientos:**
   - Se implementaron botones de control para simular los movimientos del robot:
     - **Botón Izquierda y Derecha:** Permiten cambiar la dirección del robot.
     - **Botón Avanzar:** Hace que el robot avance en la dirección actual.
     - **Botón F1:** Hace un llamado a las instrucciones que están el bloque función.
     - **Bucle**: Hace un llamado a las instrucciones que están el bloque bucle junto con la cantidad de iteraciones.
5. **Lista de Instrucciones:**

Se creó una función que maneja una lista de instrucciones, similar al juego original de Lightbot. Esto permite programar una secuencia de acciones que el robot debe seguir para completar los niveles
La **estructura** `Button` define un botón en un contexto gráfico utilizando la biblioteca SFML. Cada botón tiene un objeto `sf::Sprite`, que representa su apariencia visual en pantalla, y un identificador `id` que puede usarse para distinguir entre diferentes botones o asignarles funciones específicas. Además, el botón cuenta con una variable booleana `isHovered` que indica si el cursor del ratón está sobre él.

```
struct Button
{
   sf::Sprite sprite;
   int id;
   bool isHovered = false;
};
```

En el fragmento de código proporcionado, se está creando un conjunto de botones para una interfaz gráfica, cada uno asociado con una textura y un identificador específico que representa diferentes instrucciones o acciones. Los botones se inicializan en un vector de tamaño 6, pero el índice de los botones en el vector no coincide directamente con los identificadores. Los identificadores

asignados a cada botón están vinculados a acciones específicas: 0 es para una opción vacía, **5** corresponde a la función de "**F1**", **3** está asociado con "**giro a la izquierda"**, **2** representa "**giro a la derecha"**, **1** es para "**avanzar**", **6** se usa para "**bucle**", **4** para "**foco**", y **7** para "current". Estos identificadores permiten que cada botón ejecute una acción o función particular en la interfaz según la textura y el id asignado.

```
std::vector<Button> buttons(6);
 buttons[0] = {sf::Sprite(buttonTexture[6]), 6};
 buttons[1] = {sf::Sprite(buttonTexture[5]), 5};
 buttons[2] = {sf::Sprite(buttonTexture[4]), 4};
 buttons[3] = {sf::Sprite(buttonTexture[3]), 3};
 buttons[4] = {sf::Sprite(buttonTexture[2]), 2};
 buttons[5] = {sf::Sprite(buttonTexture[1]), 1};
```

**main():** Función principal donde se cargan los recursos como texturas, sonidos, imágenes. También se llaman funciones que permiten la inicialización de los sprites para generar el mapa 2D y el isométrico.

- **while(window.isOpen()):** Loop principal que permite la ejecución constante del código, manteniendo la ventana abierta.
- **while(window.pollEvent(event)):** Loop donde se ejecutan los eventos por entrada de hardware (teclas,mouse,etc)

```
int main():
  // Inicializacion de parametros

  while(window.isOpen()){
    // Logica principal de la aplicación

    window.clean();
    window.draw();
    window.display();
  }
```

- **for (auto &button:buttons):** Loop que registra el click del mouse sobre los botones (instrucciones) del juego.

```
for (auto &button : buttons)
      {
         sf::FloatRect bounds =
button.sprite.getGlobalBounds();
         if
(bounds.contains(static_cast<sf::Vector2f>(mousePos)))
         {
            if (!button.isHovered)
            {
               button.sprite.setScale(BUTTON_SCALE,
BUTTON_SCALE);
               button.isHovered = true;
            }

        // …
    }
```

- **if(boolIniciar):** Estructura condicional donde se implementa la lógica que recupera las instrucciones de movimiento del móvil luego de presionar el botón run de la interfaz.
    - **Control de Movimiento**: El bot ejecuta movimientos en función del valor almacenado en el array mainbot. Los movimientos incluyen girar en diferentes direcciones o moverse en la dirección actual.
    - **Secuencias de Comandos**: Existen secuencias de movimientos especiales que se activan al encontrar ciertos valores en mainbot, que dirigen al bot a ejecutar movimientos definidos en otros arrays (f1bot, buclebot).
    - **Manejo de Estados**: Se utilizan varios indicadores (moving, girando, colisionando) para asegurarse de que el bot no realice movimientos simultáneos, previniendo colisiones y asegurando que se complete cada acción antes de pasar a la siguiente.
    - **Iteraciones y Bucles**: El código también maneja bucles de movimiento (buclebot) que se repiten un número específico de veces definido por counter.
    - **Finalización**: Cuando el bot llega al final del array de movimientos o cuando se cumplen todas las iteraciones de un bucle, el proceso se reinicia o se detiene dependiendo de los valores de control.

```
if (booliniciar) {
   if (contadorMovimientos < sizeof(mainbot) /
sizeof(mainbot[0]) && !moving && !girando &&
!colisionando) {
      movimiento = mainbot[contadorMovimientos];

      if (movimiento == 2 || movimiento == 3) {
         lastmov = movimiento;
         mainbot[contadorMovimientos] = 7;
```

```
            updateDirection(contador, movimiento);
            girando = true;
            contadorMovimientos++;
        } else if (movimiento == 1) {
            lastmov = movimiento;
            mainbot[contadorMovimientos] = 7;
            move2(targetPosition, moving, estado, xIso, yIso);
            contadorMovimientos++;
        } else if (movimiento == 5) {
            // Lógica específica para f1bot
        } else if (movimiento == 6 && currentIteraciones <
counter) {
            // Lógica específica para buclebot
        } else if (movimiento == 0) {
            contadorMovimientos = 0;
            booliniciar = false;
        } else if (currentIteraciones == counter) {
            contadorMovimientos++;
            currentIteraciones = 0;
        }
    } else if (!moving && !girando && !colisionando) {
        contadorMovimientos = 0;
        booliniciar = false;
    }
}
```

- **if (moving || colisionando) else if (girando):**Este código maneja el movimiento y las colisiones de un robot llamado makibot en un juego con vista isométrica. El robot puede moverse en diferentes direcciones dentro de un mapa de bloques y, al colisionar con ciertos bloques específicos o con los bordes del mapa, se detiene y cambia de mapa.

  El código también maneja la animación del robot según la dirección en la que se mueve. Si se detecta una colisión con un bloque especial, el juego resetea varios estados y procede a cambiar al siguiente mapa en la secuencia, reasignando los sprites y posiciones necesarias para continuar el juego.

  Además, el código incluye la lógica para manejar los giros del robot, ajustando la textura y la escala del sprite para reflejar la dirección en la que mira.

```
if (moving || colisionando) {
    Vector2f newPosition = targetPosition;
    Vector2i indices = calculateGridIndices(newPosition,
pos_origin);
    int posXIso = indices.x;
    int posYISo = indices.y;

    if (mapas[mapaActual][posXIso][posYISo] == -1 &&
posXIso != -1 && posYISo != -1) {
        resetGameState();
        mapaActual = (mapaActual + 1) % 5;
        resetSpritesAndPositions();
        cargarNuevoMapa();
    }

    if (!colisionando) {
        if (!isValidPosition(posXIso, posYISo) ||
!isWalkable(mapas[mapaActual], posXIso, posYISo)) {
            colisionando = true;
            moving = false;
            targetPosition = makibot.getPosition();
        } else {
            actualizarAnimacion();
            moveRobot(makibot, targetPosition, xIso, yIso);  t
            actualizarMapa2D(posXIso, posYISo);
    } else if (colisionando &&
animationClock.getElapsedTime().asSeconds() > 1.5) {
        colisionando = false;
        animationClock.restart();
    }

    actualizarEscala(miraNE, miraNO, miraSO, miraSE);
 else if (girando) {
    if (animationClock.getElapsedTime().asSeconds() >
1.5) {
        actualizarTextura(miraNE, miraNO, miraSO,
miraSE);
        girando = false;
        animationClock.restart();
    }
}
```

- **window.clean(), window.draw(), window.display():** :Este bloque de código se encarga de renderizar y mostrar todos los elementos gráficos del juego en la ventana principal. Los elementos incluyen el piso 2D, el

piso isométrico, los bloques, el personaje principal (makibot), y los botones de control.

El proceso se realiza en varios pasos: primero se limpia la ventana, luego se dibujan los diferentes elementos en el orden adecuado para asegurar que todo se visualice correctamente. También se aplican transformaciones isométricas a ciertos elementos para darles la apariencia tridimensional. Finalmente, se manejan interacciones como el escalado de botones cuando se pasa el cursor por encima.

- **void renderImagesBlocksWithControls():** Esta función maneja la visualización de imágenes y la interacción con botones en una ventana
    - **Estructura Condicional:** Revisa el estado de la ventana y maneja eventos.
    - **Renderizado de Imágenes:** Dibuja las imágenes en la ventana usando los índices proporcionados.
    - **Control de Botones:** Define y actualiza la apariencia y posición de los botones.
    - **Manejo de Eventos de Botones:** Controla la interacción del usuario con los botones.
    - **Dibujo de Botones en la Ventana:** Renderiza los botones en la ventana.

```
void renderImagesBlocksWithControls(sf::RenderWindow &window, const std::vector<sf::Texture> &texturas, int array[], int tamano, int xx, int yy, sf::Event event, bool &estado)
{
   // Tamaño de cada imagen (asumimos que todas tienen el mismo tamaño)
   sf::Vector2u imageSize = texturas[0].getSize();
   int imageWidth = imageSize.x;
   int imageHeight = imageSize.y;

   int offsetX = xx; // Desplazamiento en x
   int offsetY = yy; // Desplazamiento en y
   // …
}
```