Informe del Proyecto 1 de Robótica

Oscar Mellado Bustos

3 de mayo de 2025

1. Introducción

La simulación en robótica emerge como una herramienta indispensable para validar y profundizar en el entendimiento de algoritmos complejos, permitiendo trasladar conceptos teóricos a entornos prácticos y controlados. Este enfoque no solo facilita la visualización de comportamientos dinámicos, sino que también revela interacciones críticas entre modelos matemáticos y su implementación real. En este proyecto, se integraron disciplinas como la cinemática tridimensional, el álgebra matricial y el renderizado de gráficos 3D para desarrollar una simulación que recrea un escenario inspirado en el baloncesto, donde robots autónomos compiten para encestar una pelota en el aro rival.

El núcleo del sistema reside en la capacidad de los robots para calcular, en tiempo real, parámetros esenciales como la distancia euclidiana al objetivo, la velocidad de disparo y el ángulo de elevación óptimo. Gracias a la precisión de los algoritmos implementados basados en ecuaciones de lanzamiento parabólico y ajustes trigonométricos, los robots garantizan un éxito constante en los lanzamientos, demostrando la eficacia de utilización de modelos físicos rigurosos en ambientes simulados y controlados.

Para el desarrollo del proyecto se realizo utilizando la librería gráfica Raylib (Santamaria, 2025). Siendo una librería ligera y minimalista,

2. Funciones Matemáticas en `utils.c`

El archivo `utils.c` contiene funciones clave para los cálculos matemáticos necesarios para los ángulos de: disparo de la pelota, y orientación del robot, así como una función para determinar la velocidad de disparo de la pelota dependiendo de la distancia respecto del aro del equipo rival. A continuación, se explican en detalle las funciones `get_target_angle()` y `get_shot_angle()`.

2.1. `get_target_angle()`

Esta función calcula el ángulo entre la posición actual de un robot y el aro objetivo en el plano XZ. La fórmula utilizada considera las coordenadas relativas del objetivo respecto a la posición del robot.

```
float get_target_angle(Vector3 position, Vector3 target) {
    Vector3 relative_target = Vector3Subtract(target, position);

    float sum = 0.0f;
    if (relative_target.x < 0.0f && relative_target.z < 0.0f)
        sum = PI;
    else if (relative_target.x < 0.0f && relative_target.z > 0.0f)
        sum = 2 * PI;
    else if (relative_target.x > 0.0f && relative_target.z < 0.0f)
        sum = PI;

    float angle = atanf(relative_target.x / relative_target.z);

    return angle + sum;
}
```

Explicación:

- Se calcula el vector relativo entre el objetivo y la posición actual.
- Dependiendo del cuadrante relativo en el que se encuentra el aro objetivo, se ajusta el ángulo base (`sum`) para garantizar que el resultado esté en el rango $[0, 2\pi]$.
- Finalmente, se calcula el ángulo utilizando la función `atanf()` y se suma el ajuste correspondiente.

2.2. `get_shot_angle()`

Esta función calcula el ángulo de lanzamiento necesario para que un robot alcance el aro objetivo a una distancia dada, considerando la gravedad y la velocidad inicial.

```
float get_shot_angle(float distance, float velocity, float h) {
    float gravity = -GRAVITY;
    float solutions[2] = { 0 };

    float a = (gravity * distance) / (2 * velocity * velocity);

    float det = (h / (a * distance)) - 1 + (1 / (4 * a * a));

    if (det < 0) return -1.0f;

    float b = -1 / (2 * a);
    float c = sqrtf(det);
    solutions[0] = atanf(b + c);
    solutions[1] = atanf(b - c);

    if (solutions[0] < solutions[1]) return solutions[0];

    return solutions[1];
```

```
}
```

Explicación:

- Se define la gravedad como una constante negativa (`-GRAVITY`).
- Se calcula el coeficiente a que relaciona la distancia, la velocidad y la gravedad.
- Se determina el discriminante (`det`) para verificar si existen soluciones reales.
- Si el discriminante es negativo, no hay solución real, y la función devuelve un error.
- Si hay soluciones, se calculan dos posibles ángulos de lanzamiento (`solutions[0]` y `solutions[1]`), y se selecciona uno de ellos.

2.2.1. Desarrollo matemático del calculo del ángulo de disparo

Para calcular el ángulo de disparo tenemos lo siguientes datos: coordenadas del aro, coordenadas del robot, y la velocidad inicial que tendrá la pelota al momento de dispararse.

Primero calculamos la distancia euclidiana entre el robot y el aro en el plano XZ.

$$d(Robot, Aro) = \sqrt{(z_r - z_a)^2 + (x_r - x_a)^2} \tag{1}$$

Con la distancia calculada y la velocidad en el eje horizontal podemos determinar el tiempo de vuelo de la pelota.

$$\frac{d(Robot, Aro_{xz})}{(v_b \cdot \cos\alpha)} = t \tag{2}$$

Teniendo el tiempo de vuelo se reemplazara en la ecuación de posición del eje vertical.

$$h = (v_b \cdot \sin\alpha)\frac{d(Robot, Aro_{xz})}{v_b \cdot \cos\alpha} - \frac{1}{2}g\frac{d(Robot, Aro_{xz})^2}{(v_b \cdot \cos\alpha)^2} \tag{3}$$

Debido a identidades trigonométricas podemos sustituir algunas partes de la ecuación con términos que incluyan la tangente del ángulo de elevación.

$$h = \tan\alpha \cdot d(Robot, Aro_{xz}) - g\frac{d(Robot, Aro_{xz})^2}{2v_b^2}(1 + \tan^2\alpha) \tag{4}$$

Haremos un cambio de variable con la tangente del ángulo α por la variable x.

$$h = x \cdot d(Robot, Aro_{xz}) - g\frac{d(Robot, Aro_{xz})^2}{2 \cdot v_b^2}(1 - x^2) \tag{5}$$

Dividimos toda la expresión en la distancia para simplificar algunos términos.

$$\frac{h}{d(Robot, Aro_{xz})} = x - g\frac{d(Robot, Aro_{xz})}{2 \cdot v_b^2}(1 - x^2) \tag{6}$$

Con el fin de agrupar valores constantes se define la variable a donde se calcula la relación entre la aceleración de la gravedad y la distancia entre el doble del cuadrado de la velocidad inicial.

$$a = -g\frac{d(Robot, Aro_{xz})}{2 \cdot v_b^2} \tag{7}$$

Desde los procedimientos anteriores llegamos a la siguiente ecuación de segundo grado.

$$\frac{h}{d(Robot, Aro_{xz})} = x + a + ax^2 \tag{8}$$

Donde resolviendo por completación de cuadrados o la formula general llegamos a las siguientes soluciones:

$$\arctan\left(-\frac{1}{2a} + \sqrt{\frac{h}{a \cdot d(Robot, Aro_{xz})} - 1 + \frac{1}{4a^2}}\right) = \alpha_0$$

$$\arctan\left(-\frac{1}{2a} - \sqrt{\frac{h}{a \cdot d(Robot, Aro_{xz})} - 1 + \frac{1}{4a^2}}\right) = \alpha_1$$

3. Tipo `uniciclo_t`

El tipo `uniciclo_t` representa un robot uniciclo en la simulación. Está definido en el archivo `robot.h` y contiene los siguientes campos:

```
typedef struct RobotUniciclo {
    frame_t *obj;
    float w;
    float old_w;
    float *y_rotation;
    float y_rotation_expected;
    float old_y_rotation;
    float vl;
    float old_vl;
    float steps;
    uint8_t config;
    float time_to_shot;
    float wait_for_shot;
    ball_t *ball;
    Vector3 target;
    Color team;
} uniciclo_t;
```

Descripción de Campos:

- `obj`: Donde se guardara el puntero al frame que representará de forma gráfica el robot.
- `w`: Velocidad angular.
- `old_w`: Velocidad angular antigua.
- `y_rotation`: Puntero de la rotación actual en el eje Y que tiene el campo `obj`.

- `y_rotation_expected`: Es la rotación esperada mientras el robot ajusta su orientación.
- `old_y_rotation`: Es el ultimo ángulo que tuvo el robot.
- `vl`: Velocidad lineal del robot.
- `old_vl`: Ultima velocidad lineal del robot.
- `steps`: Tiempo en el que el robot tendrá una misma velocidad lineal y angular antes de reiniciarse.
- `config`: Configuración de los estados del robot, almacenada como un conjunto de bits.
- `time_to_shot`: Tiempo en que el robot esperará entre cada proceso de disparo.
- `wait_for_shot`: Tiempo de espera del robot una vez que apunta al objetivo.
- `ball`: Puntero a la pelota asociada al robot.
- `target`: Aro objetivo del robot.
- `team`: Color del equipo al que pertenece el robot.

Funciones relevantes:

- `create_robot()`: Crea e inicializa un robot.
- `destroy_robot()`: Libera los recursos asociados al robot.
- `update_robot()`: Actualiza el estado del robot, incluyendo movimiento, rotación, colisiones y disparos.
- `draw_robot()`: Grafica el robot.
- `restart_robot()`: Reinicia los valores de velocidad angular, lineal y el tiempo que se tiene que mantener con esos valores el robot.
- `configure_rotation_robot()`: Configura al robot en un estado de rotación a un ángulo de orientación deseado.
- `ready_rotation_robot()`: Revisa si el robot termino su proceso de rotación.
- `launch_ball()`: Configura la velocidad inicial, y el ángulo de lanzamiento de la pelota.
- `handle_shot()`: Rota el robot para apuntar al objetivo y al terminar devuelve al robot a su orientación original.
- `move_robot()`: Calcula el movimiento del robot en función de su velocidad y rotación.
- `rotate_robot()`: Ajusta la rotación del robot.

4. Tipo `ball_t`

El tipo `ball_t` representa una pelota en la simulación. Está definido en el archivo `ball.h` y contiene los siguientes campos:

```
typedef struct Ball {
    Vector3 *position;
    Vector3 velocity;
    float fly_time;
    float t;
    bool visible;
    sphere_t *sphere;
} ball_t;
```

Descripción de Campos:

- `position`: Puntero a la posición actual de la pelota.
- `velocity`: Vector de velocidad de la pelota.
- `fly_time`: Tiempo total de vuelo de la pelota.
- `t`: Tiempo transcurrido desde el lanzamiento.
- `visible`: Indica si la pelota es visible en la simulación.
- `sphere`: Representación gráfica de la pelota.

Funciones Relacionadas:

- `create_ball()`: Crea e inicializa una pelota.
- `destroy_ball()`: Libera los recursos asociados a la pelota.
- `update_ball()`: Actualiza la posición y velocidad de la pelota durante su vuelo.
- `draw_ball()`: Dibuja la pelota en la simulación.

5. Lógica del Campo `config` del Tipo `uniciclo_t`

El campo `config` es un entero de 8 bits (`uint8_t`) que utiliza bits individuales como banderas para representar diferentes estados del robot. Este diseño permite manejar múltiples estados de manera eficiente y compacta. Cada bit tiene un propósito específico, y se manipula mediante macros definidas en `robot.h` y `collision.h`.

5.1. Bits y su Significado

- **Bit 0 (`COLLISION`)**: Indica si el robot está en estado de colisión.
- **Bit 1 (`COLLISION_UP`)**: Si está activado, indica colisión hacia arriba; si no, indica colisión hacia abajo.
- **Bit 2 (`COLLISION_RIGHT`)**: Si está activado, indica colisión hacia la derecha; si no, indica colisión hacia la izquierda.

- **Bit 3 (`CHECK_COLLISION_X`)**: Indica si se debe verificar colisión en el eje X.
- **Bit 4 (`CHECK_COLLISION_Z`)**: Indica si se debe verificar colisión en el eje Z.
- **Bit 5 (`ROTATING`)**: Indica si el robot está rotando hacia un ángulo objetivo.
- **Bit 6 (`SHOOTING`)**: Indica si el robot está en proceso de disparar la pelota.
- **Bit 7 (`LAUNCHING`)**: Indica si el robot está lanzando la pelota.

5.2. Macros para Manipular los Bits

Las macros definidas en `robot.h` y `collision.h` permiten manipular los bits de manera sencilla:

- `SET_BIT(bit, pos)`: Activa un bit en la posición `pos`.
- `UNSET_BIT(bit, pos)`: Desactiva un bit en la posición `pos`.
- `CHECK_BIT(bit, pos)`: Verifica si un bit en la posición `pos` está activado.
- Ejemplos específicos:
  - `UNSET/SET_ROTATING`: Manipulan el bit `ROTATING`.
  - `UNSET/SET_SHOOTING`: Manipulan el bit `SHOOTING`.
  - `UNSET/SET_LAUNCHING`: Manipulan el bit `LAUNCHING`.

6. Lógica de las Funciones del Robot

6.1. `handle_shot(uniciclo_t *robot)`

Esta función gestiona el proceso completo de disparo del robot, desde la rotación hacia el objetivo hasta el lanzamiento de la pelota.

6.1.1. Lógica

1. **Inicio del disparo:**
   - Si el robot no está disparando (`SHOOTING` no está activado), se activa el bit `SHOOTING`.
   - Se guarda la velocidad angular y la rotación actual del robot.
   - Se calcula el ángulo hacia el objetivo usando `get_target_angle()` y se configura la rotación del robot hacia ese ángulo con `configure_robot_rotat`

```
if (!CHECK_BIT(robot->config, SHOOTING)) {
    SET_SHOOTING(robot->config);
    robot->old_w = robot->w;
```

```
    robot->old_y_rotation = *(robot->y_rotation);
    float rotation_target = get_target_angle(robot->obj->
   position, robot->target);
    configure_robot_rotation(robot, rotation_target);
}
```

2. **Rotación hacia el objetivo:**
   - Si el robot está rotando (`ROTATING` activado) pero no está lanzando (`LAUNCHING` desactivado), se verifica si la rotación está lista con `ready_robot_rotation()`.
   - Si la rotación está lista, se activa el bit `LAUNCHING`.

```
if (CHECK_BIT(robot->config, ROTATING) && !CHECK_BIT(robot->
   config, LAUNCHING)) {
    if (!ready_robot_rotation(robot)) return;
    SET_LAUNCHING(robot->config);
}
```

3. **Lanzamiento de la pelota:**
   - Si el robot no está rotando (`ROTATING` desactivado), se espera un tiempo antes de lanzar la pelota.
   - Se llama a `launch_ball()` para realizar el lanzamiento.
   - Después del lanzamiento, se configura la rotación del robot para volver a su ángulo original.

```
if (!CHECK_BIT(robot->config, ROTATING)) {
    robot->wait_for_shot -= GetFrameTime();
    if (robot->wait_for_shot > 0) return;
    robot->wait_for_shot = WAIT_TIME_FOR_SHOT;

    launch_ball(robot);

    configure_robot_rotation(robot, robot->old_y_rotation);
}
```

4. **Finalización del disparo:**
   - Una vez que el robot ha rotado de vuelta a su ángulo original, se desactivan los bits `LAUNCHING` y `SHOOTING`.
   - Se reinicia el tiempo para el próximo disparo.

```
if (!ready_robot_rotation(robot)) return;
robot->time_to_shot = get_random_float(5.0f, 10.0f);
robot->w = robot->old_w;

UNSET_LAUNCHING(robot->config);
UNSET_SHOOTING(robot->config);
```

6.2. `launch_ball(uniciclo_t *robot)`

Esta función calcula los parámetros necesarios para lanzar la pelota y actualiza su estado.

6.2.1. Lógica

1. **Cálculo de parámetros:**
   - Se calcula la distancia al objetivo usando `Vector3Distance()`.
   - Se calcula la velocidad inicial necesaria con `get_shot_velocity()`.
   - Se calcula el ángulo de lanzamiento con `get_shot_angle()`.

```
float distance= Vector3Distance(robot->obj->position,target);
float velocity = get_shot_velocity(distance);
float angle = get_shot_angle(distance, velocity, h);
```

2. **Cálculo del vector de velocidad:**
   - Se descompone la velocidad inicial en componentes horizontal y vertical.
   - La componente horizontal se divide en las direcciones X y Z según la rotación actual del robot.

```
Vector3 velocity_vector = { 0 };
float horizontal_velocity = velocity * cosf(angle);
velocity_vector.y = velocity * sinf(angle),
velocity_vector.x = horizontal_velocity * sinf(y_rotation);
velocity_vector.z = horizontal_velocity * cosf(y_rotation);
```

3. **Actualización del estado de la pelota:**
   - Se calcula el tiempo de vuelo de la pelota.
   - Se marca la pelota como visible.
   - Se actualizan la posición y la velocidad de la pelota.

```
robot->ball->fly_time = (distance / horizontal_velocity) +
   0.01f;
robot->ball->visible = true;
*(robot->ball->position) = robot->obj->position;
robot->ball->velocity = velocity_vector;
```

6.3. `update_robot(uniciclo_t *robot)`

Actualiza el estado del robot en cada iteración del bucle principal.

6.3.1. Lógica

1. **Verificación de disparo:**

- Si el robot está disparando (`SHOOTING` activado) o el tiempo para el próximo disparo ha expirado, se llama a `handle_shot()`.

```
if (robot->time_to_shot > 0)
    robot->time_to_shot -= GetFrameTime();
if (CHECK_BIT(robot->config,SHOOTING)||robot->time_to_shot<0)
    handle_shot(robot);
```

2. **Gestión de colisiones:**
   - Si el robot está en colisión (`COLLISION` activado), se llama la función `handle_robot_collision()`.

```
if (CHECK_BIT(robot->config, COLLISION))
    handle_robot_collision(robot);
```

3. **Actualización de rotación y movimiento:**
   - Se llama a `rotate_robot()` para actualizar la rotación del robot.
   - Se llama a `move_robot()` para actualizar la posición del robot.

```
rotate_robot(robot);
move_robot(robot);
```

4. **Actualización de la pelota:**
   - Si la pelota es visible, se llama a `update_ball()` para actualizar su posición y estado.

```
if (!robot->ball->visible) return;
update_ball(robot->ball);
```

7. Imágenes de la simulación

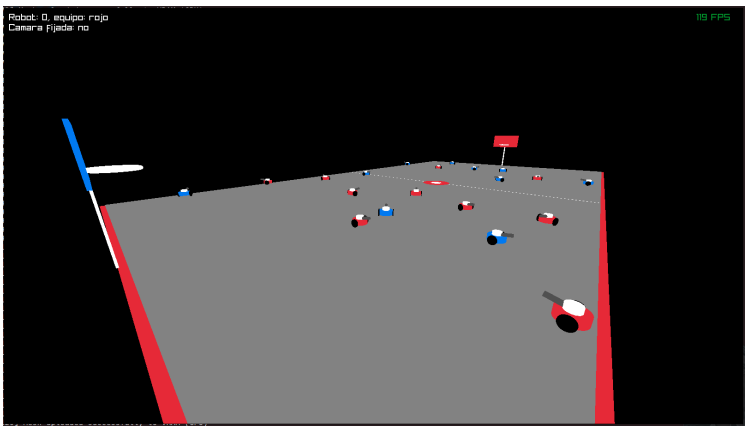

Figura 1: Imagen de la simulación.

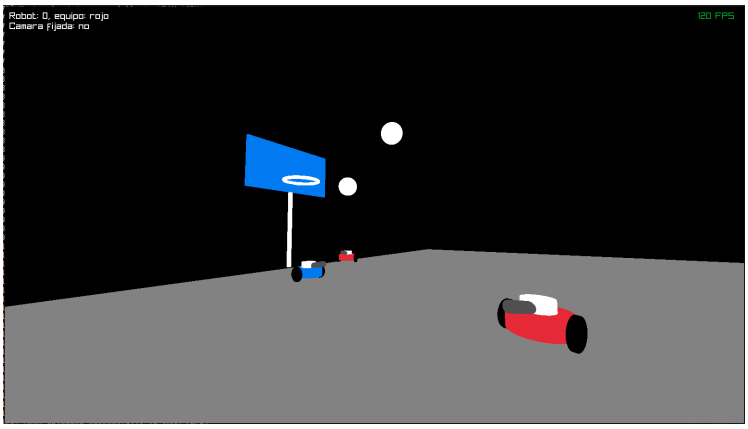

Figura 2: Imagen de robot disparando al aro.

8. Conclusiones

El proyecto logró con éxito su objetivo principal de simular robots capaces de navegar, apuntar y lanzar una pelota con precisión hacia el aro contrario. Esto fue posible gracias a la implementación de ecuaciones que modelan el movimiento parabólico de proyectiles, calculando dinámicamente el ángulo de elevación necesario en función de la velocidad inicial y la distancia al objetivo. Este enfoque matemático garantizó que cada lanzamiento fuera preciso y predecible.

Para gestionar los diversos estados de los robots, se implementó un sistema basado en un campo de 8 bits llamado `config`. Cada bit de este campo representa un estado específico del robot, incluyendo detección de colisiones en los ejes X y Z, procesos de rotación, y las fases de disparo y lanzamiento de la pelota. Este diseño permitió una gestión eficiente de múltiples estados simultáneamente.

La implementación mediante operaciones a nivel de bit demostró ser significativamente más eficiente que los métodos tradicionales basados en condicionales o variables booleanas. Las macros utilizadas para manipular estos bits se resuelven durante la compilación, lo que contribuye a un mejor rendimiento general del sistema. Además, esta solución optimiza el uso de memoria al requerir solamente un byte por robot para almacenar toda su información de estado.

Sin embargo, el proyecto reveló áreas que podrían mejorarse en futuras versiones. El sistema de colisiones actual podría ampliarse para incluir interacciones entre robots y entre múltiples pelotas, añadiendo mayor realismo a la simulación. También sería valioso implementar algoritmos de aprendizaje automático que permitan a los robots ajustar autónomamente sus parámetros de lanzamiento.

Finalmente, el trabajo destacó la importancia de optimizar los recursos gráficos en simulaciones robóticas. Si bien se utilizaron librerías minimalistas para la representación visual, esta experiencia mostró oportunidades para mejorar el rendimiento gráfico sin sacrificar la eficiencia computacional.

Este aspecto resulta particularmente relevante para simulaciones más complejas o con mayor número de agentes.

Referencias

Santamaria, R. (2025). raylib - cheatsheet. https://www.raylib.com/cheatsheet/cheatsheet.html

9. Anexos

9.1. Código fuente

9.1.1. `main.c`

```
#include <robot.h>
#include <random.h>
#include <config.h>
#include <collision.h>
#include <graphics.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <raymath.h>

void draw_world(Model *aro) {
    DrawCubeV((Vector3){ 0.0f, -0.05f, 0.0f }, (Vector3){
   CANCHA_HEIGHT + 0.2f, 0.03f, CANCHA_WIDTH + 0.2f}, RED);
    DrawCubeV((Vector3){ 0.0f, -0.025f, 0.0f }, (Vector3){
   CANCHA_HEIGHT, 0.05f, CANCHA_WIDTH}, GRAY);
    DrawPlane((Vector3){0.0f, 0.0001f, 0.0f}, (Vector2){0.03f, 9.8
   f}, WHITE);
    DrawCylinder((Vector3){0.0f, -0.099f, 0.0f}, 0.5f, 0.5f, 0.1f,
     32, RED);
    DrawCylinder((Vector3){0.0f, -0.098f, 0.0f}, 0.2f, 0.5f, 0.1f,
     32, WHITE);

    DrawCubeV((Vector3){ 9.0f, 1.5f, 0.0f }, (Vector3){0.04f, 0.7f
   , 1.4f}, RED);
    DrawCubeV((Vector3){ 9.0f, 0.6f, 0.0f }, (Vector3){0.03f, 1.2f
   , 0.06f}, WHITE);
    DrawModel(*aro, POS_RING_RED, 1.0f, WHITE);

    DrawCubeV((Vector3){ -9.0f, 1.5f, 0.0f }, (Vector3){0.04f, 0.7
   f, 1.4f}, BLUE);
    DrawCubeV((Vector3){ -9.0f, 0.6f, 0.0f }, (Vector3){0.03f, 1.2
   f, 0.06f}, WHITE);
    DrawModel(*aro, POS_RING_BLUE, 1.0f, WHITE);
}

int main() {
    srand((unsigned int)time(NULL));

    InitWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Dron");

    Mesh aro = GenMeshTorus(0.001f, 0.5f, 32, 32);
    Model aro_m = LoadModelFromMesh(aro);
    aro_m.transform = MatrixRotateX(90.0f * DEG2RAD);

    Camera3D camera = { 0 };
    camera.position = POS_RING_BLUE;
    camera.target = (Vector3){ 0.0f, 0.0f, 0.0f };
    camera.up = (Vector3){ 0.0f, 1.0f, 0.0f };
    camera.fovy = 80.0f;
    camera.projection = CAMERA_PERSPECTIVE;
```

```
    uniciclo_t **robots_red = (uniciclo_t **)malloc(sizeof(
    uniciclo_t *) * N_ROBOTS_PER_TEAM);
#if !ONCE_TEAM
    uniciclo_t **robots_blue = (uniciclo_t **)malloc(sizeof(
    uniciclo_t *) * N_ROBOTS_PER_TEAM);
#endif
    for (size_t i = 0; i < N_ROBOTS_PER_TEAM; i++) {
        robots_red[i] = create_robot((Vector3){0.0f, 0.0f, 0.0f},
    RED);
#if !ONCE_TEAM
        robots_blue[i] = create_robot((Vector3){0.0f, 0.0f, 0.0f},
     BLUE);
#endif
    }

    DisableCursor();

    bool fixed_target_camera = false;
    size_t robot_index = 0;
    char team = 'r';

    SetTargetFPS(120);
    while (!WindowShouldClose()) {

        if (IsKeyDown(KEY_SPACE)) camera.position.y += 2.0f *
    GetFrameTime();
        if (IsKeyDown(KEY_LEFT_SHIFT)) camera.position.y -= 2.0f *
     GetFrameTime();
        if (IsKeyPressed(KEY_F)) fixed_target_camera = !
    fixed_target_camera;
        if (IsKeyPressed(KEY_T)) team = (team == 'r') ? 'b' : 'r';
        if (IsKeyPressed(KEY_RIGHT)) {
            robot_index++;
            if (robot_index >= N_ROBOTS_PER_TEAM) robot_index = 0;
        }
        if (IsKeyPressed(KEY_LEFT)) {
            robot_index--;
            if (robot_index >= N_ROBOTS_PER_TEAM) robot_index =
    N_ROBOTS_PER_TEAM - 1;
        }

        if (fixed_target_camera) {
            if (team == 'r') camera.target = robots_red[
    robot_index]->obj->position;
            if (team == 'b') camera.target = robots_blue[
    robot_index]->obj->position;
        }

        check_camera_collision(&camera);

        for (size_t i = 0; i < N_ROBOTS_PER_TEAM; i++) {
            update_robot(robots_red[i]);
#if !ONCE_TEAM
            update_robot(robots_blue[i]);
#endif
        }

        UpdateCamera(&camera, CAMERA_FIRST_PERSON);
```

```
        BeginDrawing();

            ClearBackground(BLACK);

            BeginMode3D(camera);
            draw_world(&aro_m);

            for (size_t i = 0; i < N_ROBOTS_PER_TEAM; i++) {
                draw_robot(robots_red[i]);
#if !ONCE_TEAM
                draw_robot(robots_blue[i]);
#endif
            }

            EndMode3D();
            const char *info_robot = TextFormat("Robot: %zu,
   equipo: %s\nCamara fijada: %s",
                robot_index,
                team == 'r' ? "rojo" : "azul",
                fixed_target_camera ? "si" : "no");

            DrawText(info_robot, 10, 10, 20, WHITE);
            DrawFPS(SCREEN_WIDTH - 100, 10);
        EndDrawing();
    }

    for (size_t i = 0; i < N_ROBOTS_PER_TEAM; i++) {
        destroy_robot(robots_red[i]);
#if !ONCE_TEAM
        destroy_robot(robots_blue[i]);
#endif
    }

    free(robots_red);
#if !ONCE_TEAM
    free(robots_blue);
#endif
    UnloadModel(aro_m);
    CloseWindow();

    return 0;
}
```

9.1.2. `robot.c`

```
#include <robot.h>
#include <config.h>
#include <random.h>
#include <objects.h>
#include <collision.h>
#include <utils.h>

#include <raymath.h>
#include <stdlib.h>

void generate_frame(frame_t *frame, Color team) {
    if (frame == NULL) return;
```

```
    cylinder_t *base = create_cylinder(0.15f, 0.10f, (Vector3){0.0
    f, 0.04f, 0.0f}, (Vector3){0.0f, 0.0f, 0.0f}, team);
    add_entity(frame, base);
    cylinder_t *head = create_cylinder(0.08f, 0.06f, (Vector3){0.0
    f, 0.14f, 0.0f}, (Vector3){0.0f, 0.0f, 0.0f}, WHITE);
    add_entity(frame, head);

    cylinder_t *r_wheel = create_cylinder(0.08f, 0.02f, (Vector3)
    {0.15f, 0.08f, 0.0f}, (Vector3){0.0f, 0.0f, -90.0f * DEG2RAD},
    BLACK);
    add_entity(frame, r_wheel);
    cylinder_t *l_wheel = create_cylinder(0.08f, 0.02f, (Vector3)
    {-0.15f, 0.08f, 0.0f}, (Vector3){0.0f, 0.0f, 90.0f * DEG2RAD},
    BLACK);
    add_entity(frame, l_wheel);

    cylinder_t *cannon = create_cylinder(0.03f, 0.2f, (Vector3)
    {0.0f, 0.16f, 0.05f}, (Vector3){80.0f * DEG2RAD, 0.0f, 0.0f},
    DARKGRAY);
    add_entity(frame, cannon);
}

void restart_robot(uniciclo_t *robot, float start_angle, float
    end_angle) {
    float new_angle = get_random_float(start_angle * DEG2RAD,
    end_angle * DEG2RAD);
    float angle_diff = get_angle_diff(*(robot->y_rotation),
    new_angle);

    robot->steps = get_random_float(0.2f, 1.0f);
    robot->w = angle_diff / robot->steps;
    robot->vl = get_random_float(0.0f, 6.0f);
}


uniciclo_t* create_robot(Vector3 pos, Color color) {

    uniciclo_t* robot = (uniciclo_t*)malloc(sizeof(uniciclo_t));

    bool team = color.r == RED.r && color.g == RED.g && color.b ==
    RED.b;
    Vector3 target = { 0 };
    if (team) target = POS_RING_BLUE;
    else target = POS_RING_RED;
    frame_t *frame = create_frame(pos, (Vector3){0.0f, 0.0f, 0.0f
    });
    generate_frame(frame, color);

    ball_t *ball = create_ball();

    *robot = (uniciclo_t) {
        .obj = frame,
        .vl = 0.0f,
        .steps = 0,
        .team = color,
        .y_rotation = &frame->rotation.y,
        .y_rotation_expected = -1.0f,
```

```
        .time_to_shot = get_random_float(5.0f, 10.0f),
        .wait_for_shot = WAIT_TIME_FOR_SHOT,
        .target = target,
        .ball = ball,
        .config = 0,
        .w = 0.0f
    };

    restart_robot(robot, 0.0f, 360.0f);

    return robot;
}

void destroy_robot(uniciclo_t* robot) {
    destroy_frame(robot->obj);
    destroy_ball(robot->ball);
    free(robot);
}


void draw_robot(uniciclo_t *robot) {
    if (robot == NULL || robot->obj == NULL) return;

    draw_frame(robot->obj);
    draw_ball(robot->ball);
}

void rotate_robot(uniciclo_t *robot) {
    float y_rotation = *(robot->y_rotation);
    y_rotation += robot->w * GetFrameTime();

    if (y_rotation > 2 * PI) y_rotation -= 2 * PI;
    else if (y_rotation < 0.0f) y_rotation += 2 * PI;

    *(robot->y_rotation) = y_rotation;
}

void move_robot(uniciclo_t *robot) {
    if (CHECK_BIT(robot->config, SHOOTING)) return;
    if (CHECK_BIT(robot->config, COLLISION)) return;

    float y_rotation = *(robot->y_rotation);
    float *x = &robot->obj->position.x;
    float *z = &robot->obj->position.z;

    robot->steps -= GetFrameTime();
    if (robot->steps < 0) {
        restart_robot(robot, 0.0f, 360.0f);
        return;
    }

    float velocity = robot->vl * GetFrameTime();
    *x += velocity * sinf(y_rotation);
    *z += velocity * cosf(y_rotation);

    if (*x <= -8.7f) {
        *x = -8.7f;
        COLLISION_DOWN(robot->config);
```

```
    }
    if (*x >= 8.7f) {
        *x = 8.7f;
        COLLISION_UP(robot->config);
    }
    if (*z <= -4.7f) {
        *z = -4.7f;
        COLLISION_LEFT(robot->config);
    }
    if (*z >= 4.7f) {
        *z = 4.7f;
        COLLISION_RIGHT(robot->config);
    }
}

bool ready_robot_rotation(uniciclo_t *robot) {
    float y_rotation = *(robot->y_rotation);
    if (fabsf(robot->y_rotation_expected - y_rotation) > 0.1f)
        return false;

    *(robot->y_rotation) = robot->y_rotation_expected;
    robot->w = 0.0f;
    UNSET_ROTATING(robot->config);
    return true;
}

void configure_robot_rotation(uniciclo_t *robot, float
   rotation_target) {
    SET_ROTATING(robot->config);
    float y_rotation = *(robot->y_rotation);

    robot->w = 5.0f;

    robot->y_rotation_expected = rotation_target;
    float angle_diff = get_angle_diff(y_rotation, rotation_target)
   ;
    if (angle_diff < 0.0f) robot->w = -robot->w;
}

void launch_ball(uniciclo_t *robot) {

    Vector3 target = {
        robot->target.x,
        0.0f,
        robot->target.z
    };

    float h = robot->target.y;
    float distance = Vector3Distance(robot->obj->position, target)
   ;
    float velocity = get_shot_velocity(distance);
    float angle = get_shot_angle(distance, velocity, h);

    float y_rotation = *(robot->y_rotation);

    Vector3 velocity_vector = { 0 };
    float horizontal_velocity = velocity * cosf(angle);
    velocity_vector.y = velocity * sinf(angle),
```

```
    velocity_vector.x = horizontal_velocity * sinf(y_rotation);
    velocity_vector.z = horizontal_velocity * cosf(y_rotation);

    robot->ball->fly_time = (distance / horizontal_velocity) +
   0.01f;
    robot->ball->visible = true;
    *(robot->ball->position) = robot->obj->position;
    robot->ball->velocity = velocity_vector;
}

void handle_shot(uniciclo_t *robot) {
    if (!CHECK_BIT(robot->config, SHOOTING)) {
        SET_SHOOTING(robot->config);
        robot->old_w = robot->w;
        robot->old_y_rotation = *(robot->y_rotation);
        float rotation_target = get_target_angle(robot->obj->
   position, robot->target);
        configure_robot_rotation(robot, rotation_target);
    }

    if (CHECK_BIT(robot->config, ROTATING) && !CHECK_BIT(robot->
   config, LAUNCHING)) {
        if (!ready_robot_rotation(robot)) return;
        SET_LAUNCHING(robot->config);
    }

    if (!CHECK_BIT(robot->config, LAUNCHING)) return;

    if (!CHECK_BIT(robot->config, ROTATING)) {
        robot->wait_for_shot -= GetFrameTime();
        if (robot->wait_for_shot > 0) return;
        robot->wait_for_shot = WAIT_TIME_FOR_SHOT;

        launch_ball(robot);

        configure_robot_rotation(robot, robot->old_y_rotation);
    }

    if (!ready_robot_rotation(robot)) return;
    robot->time_to_shot = get_random_float(5.0f, 10.0f);
    robot->w = robot->old_w;

    UNSET_LAUNCHING(robot->config);
    UNSET_SHOOTING(robot->config);
}

void update_robot(uniciclo_t *robot) {
    if (robot == NULL) return;

    if (robot->time_to_shot > 0) robot->time_to_shot -=
   GetFrameTime();

    if (CHECK_BIT(robot->config, SHOOTING) || robot->time_to_shot
   < 0)
        handle_shot(robot);

    if (CHECK_BIT(robot->config, COLLISION))
        handle_robot_collision(robot);
```

```
    rotate_robot(robot);
    move_robot(robot);
    if (!robot->ball->visible) return;
    update_ball(robot->ball);
}
```

9.1.3. `ball.c`

```
#include <ball.h>
#include <config.h>

#include <stdlib.h>
ball_t* create_ball() {
    ball_t *ball = (ball_t *)malloc(sizeof(ball_t));
    if (ball == NULL)
        return NULL;

    ball->sphere = create_sphere(RADIUS_BALL, (Vector3){ 0.0f, 0.0
    f, 0.0f }, (Vector3){ 0.0f, 0.0f, 0.0f }, WHITE);
    ball->position = &(ball->sphere->object.position);
    ball->velocity = (Vector3){ 0.0f, 0.0f, 0.0f };
    ball->visible = false;
    ball->t = 0.0f;

    return ball;
}

void destroy_ball(ball_t *ball) {
    destroy_sphere(ball->sphere);
    free(ball);
}

void draw_ball(ball_t *ball) {
    if (!ball->visible) return;
    draw_sphere(*(ball->sphere));
}

void update_ball(ball_t *ball) {
    if (!ball->visible) return;

    float dt = GetFrameTime();
    ball->t += dt;

    if (ball->t > ball->fly_time) {
        ball->visible = false;
        ball->t = 0.0f;
        return;
    }

    Vector3 *position = ball->position;

    position->x += ball->velocity.x * dt;
    position->z += ball->velocity.z * dt;

    ball->velocity.y += -GRAVITY * dt;
    position->y += ball->velocity.y * dt;
```

```
}
```

9.1.4. collision.c

```
#include <collision.h>
#include <config.h>
#include <random.h>

#include <raymath.h>

void check_camera_collision(Camera *camera) {
    if (camera->position.y < 0.0f)
        camera->position.y = 0.0f;

    if (camera->position.y > WORLD_HEIGHT)
        camera->position.y = WORLD_HEIGHT;

    if (camera->position.x > WORLD_HEIGHT)
        camera->position.x = WORLD_HEIGHT;

    if (camera->position.x < -WORLD_HEIGHT)
        camera->position.x = -WORLD_HEIGHT;

    if (camera->position.z > WORLD_HEIGHT)
        camera->position.z = WORLD_HEIGHT;

    if (camera->position.z < -WORLD_HEIGHT)
        camera->position.z = -WORLD_HEIGHT;
}

void handle_robot_collision(uniciclo_t *robot) {
    if (CHECK_BIT(robot->config, SHOOTING)) return;

    if (CHECK_BIT(robot->config, ROTATING)) {

        if (!ready_robot_rotation(robot)) return;

        NO_COLLISION(robot->config);
        robot->vl = 3.0f;
        robot->steps = 0.5f;
        return;
    }

    float new_angle = 0.0f;

    if (CHECK_BIT(robot->config, CHECK_COLLISION_X)) {
        uint8_t sense = CHECK_BIT(robot->config, COLLISION_X);
        if (sense) new_angle = 270.0f;
        else       new_angle = 90.0f;
    }
    if (CHECK_BIT(robot->config, CHECK_COLLISION_Z)) {
        uint8_t sense = CHECK_BIT(robot->config, COLLISION_Z);
        if (sense) new_angle = 180.0f;
        else       new_angle = 0.0f;
    }
    new_angle *= DEG2RAD;

    configure_robot_rotation(robot, new_angle);
```

```
}
```

9.1.5. utils.c

```
#include <utils.h>
#include <stdio.h>

float get_angle_diff(float current_angle, float new_angle) {
    current_angle += ((2 * PI) * (float)(current_angle < 0)) - ((2
    * PI) * (float)(current_angle > 2 * PI));

    if (new_angle > PI) new_angle -= 2 * PI;

    float angle_diff = new_angle - current_angle;

    if (angle_diff > PI) angle_diff -= 2 * PI;
    else if (angle_diff < -PI) angle_diff += 2 * PI;

    return angle_diff;
}

float get_target_angle(Vector3 position, Vector3 target) {
    Vector3 relative_target = Vector3Subtract(target, position);

    float sum = 0.0f;
    if (relative_target.x < 0.0f && relative_target.z < 0.0f)
        sum = PI;
    else if (relative_target.x < 0.0f && relative_target.z > 0.0f)
        sum = 2 * PI;
    else if (relative_target.x > 0.0f && relative_target.z < 0.0f)
        sum = PI;

    float angle = atanf(relative_target.x / relative_target.z);

    return angle + sum;
}

float get_shot_velocity(float distance) {
    return distance + 5.0f + (distance * distance * 1e-10f);
}

float get_shot_angle(float distance, float velocity, float h) {

    float gravity = -GRAVITY;
    float solutions[2] = { 0 };

    float a = (gravity * distance) / (2 * velocity * velocity);

    float det = (h / (a * distance)) - 1 + (1 / (4 * a * a));

    if (det < 0) {
        return -1.0f;
    }
    float b = -1 / (2 * a);
    float c = sqrtf(det);
    solutions[0] = atanf(b + c);
    solutions[1] = atanf(b - c);
```

```
#if GREATER
    if (solutions[0] > solutions[1]) return solutions[0];
#else
    if (solutions[0] < solutions[1]) return solutions[0];
#endif

    return solutions[1];
}
```

9.1.6. `frame.c`

```
#include <frame.h>
#include <graphics.h>

#include <raymath.h>
#include <stdlib.h>

frame_t *create_frame(Vector3 position, Vector3 rotation) {
    frame_t *frame = (frame_t *)malloc(sizeof(frame_t));
    if (frame == NULL) {
        return NULL;
    }

    frame->position = position;
    frame->rotation = rotation;
    frame->entities = create_linked_list();

    return frame;
}

void draw_frame(frame_t *frame) {
    void **entities = get_array(frame->entities);

    float y_rotation = frame->rotation.y;

    for (size_t i = 0; i < frame->entities->size; i++) {
        cube_t entity = *(cube_t *)entities[i];

        entity.object.position = Vector3RotateByAxisAngle(entity.
   object.position, (Vector3){0.0f, 1.0f, 0.0f}, y_rotation);
        entity.object.position = Vector3Add(entity.object.position
   , frame->position);

        Matrix t_rotation = MatrixRotate((Vector3){0.0f, 1.0f, 0.0
   f}, y_rotation + entity.object.rotation.y);
        entity.object.model.transform = MatrixMultiply(t_rotation,
    entity.object.model.transform);

        draw_cube(entity);
    }

    free(entities);
}

void add_entity(frame_t *frame, void *entity) {

    if (frame == NULL || entity == NULL) {
        return;
```

```
    }

    append(frame->entities, entity);
}

void destroy_frame(frame_t *frame) {
    if (frame == NULL) {
        return;
    }
    destroy_linked_list(frame->entities);

    free(frame);
}
```

9.1.7. graphics.c

```
#include <graphics.h>
#include <config.h>

#include <raymath.h>

void draw_object(object_t object) {
    object.model.transform = MatrixMultiply(MatrixRotateXYZ(object
    .rotation), object.model.transform);
    DrawModel(object.model, object.position, 1.0f, object.color);
}

void draw_cube(cube_t cube) {
    draw_object(cube.object);
}

void draw_cylinder(cylinder_t cylinder) {
    draw_object(cylinder.object);
}

void draw_sphere(sphere_t sphere) {
    draw_object(sphere.object);
}
```

9.1.8. objects.c

```
#include <objects.h>
#include <stdlib.h>

object_t create_object(Vector3 position, Vector3 rotation, Color
    color) {
    object_t object = { 0 };

    object.position = position;
    object.rotation = rotation;
    object.color = color;

    return object;
}

void destroy_object(object_t *object) {
    UnloadModel(object->model);
}
```

```
cube_t *create_cube(Vector3 size, Vector3 position, Vector3
    rotation, Color color) {
    cube_t *cube = (cube_t *)malloc(sizeof(cube_t));

    cube->size = size;
    cube->object = create_object(position, rotation, color);

    cube->object.model = LoadModelFromMesh(GenMeshCube(size.x,
    size.y, size.z));

    return cube;
}

sphere_t *create_sphere(float radius, Vector3 position, Vector3
    rotation, Color color) {
    sphere_t *sphere = (sphere_t *)malloc(sizeof(sphere_t));

    sphere->radius = radius;
    sphere->object = create_object(position, rotation, color);

    sphere->object.model = LoadModelFromMesh(GenMeshSphere(radius,
     32, 32));

    return sphere;
}

cylinder_t *create_cylinder(float radius, float height, Vector3
    position, Vector3 rotation, Color color) {
    cylinder_t *cylinder = (cylinder_t *)malloc(sizeof(cylinder_t)
    );

    cylinder->radius = radius;
    cylinder->height = height;
    cylinder->object = create_object(position, rotation, color);

    cylinder->object.model = LoadModelFromMesh(GenMeshCylinder(
    radius, height, 32));

    return cylinder;
}

void destroy_cube(cube_t *cube) {
    destroy_object(&cube->object);
    free(cube);
}

void destroy_sphere(sphere_t *sphere) {
    destroy_object(&sphere->object);
    free(sphere);
}

void destroy_cylinder(cylinder_t *cylinder) {
    destroy_object(&cylinder->object);
    free(cylinder);
}
```

9.1.9. `random.c`

```c
#include <random.h>

#include <stdlib.h>

float get_random_float(float min, float max) {
    return ((float)rand() / (float)(RAND_MAX)) * (max - min) + min
    ;
}

size_t get_random_size_t(size_t min, size_t max) {
    return (size_t)(get_random_float((float)min, (float)max));
}
```

9.1.10. `linked_list.c`

```c
#include <linked_list.h>
#include <objects.h>

#include <stdio.h>
#include <stdlib.h>

linked_list_t* create_linked_list() {
    linked_list_t* list = (linked_list_t*)malloc(sizeof(
    linked_list_t));
    if (list == NULL) exit(1);
    list->head = NULL;
    list->tail = NULL;
    list->size = 0;
    return list;
}

node_t* create_node(void *data) {
    node_t* new_node = (node_t*)malloc(sizeof(node_t));
    if (new_node == NULL) exit(1);
    new_node->data = data;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}

void append(linked_list_t *l, void *data) {
    if (l == NULL) return;

    node_t* new_node = create_node(data);
    l->size++;
    if (l->head == NULL) {
        l->head = new_node;
        l->tail = new_node;
        return;
    }

    l->tail->next = new_node;
    new_node->prev = l->tail;
    l->tail = new_node;
}

void **get_array(linked_list_t *list) {
    void **array = (void **)malloc(sizeof(void *) * list->size);
```

```
    node_t *current = list->head;

    size_t i = 0;
    while (current != NULL) {
        array[i] = current->data;
        i++;
        current = current->next;
    }

    return array;
}

void destroy_node(node_t *node) {
    if (node == NULL) return;
    if (node->data != NULL) {
        destroy_cube((cube_t *)node->data);
    }

    free(node);
}

void delete_at(linked_list_t* l, size_t position) {
    if (l == NULL) return;
    if (position >= l->size) return;

    node_t *current = l->head;
    size_t i = 0;
    while (current != NULL || i != position) {
        i++;
        current = current->next;
    }

    if (current == NULL) return;

    node_t *del_node = current;
    node_t *prev_node = current->prev;
    node_t *next_node = current->next;

    prev_node->next = next_node;
    next_node->prev = prev_node;

    destroy_node(del_node);
}

void destroy_linked_list(linked_list_t *list) {
    if (list == NULL) return;

    node_t* current = list->head;
    while (current != NULL) {
        node_t* del_node = current;
        current = del_node->next;
        destroy_node(del_node);
    }

    free(list);
}
```

9.2. Cabeceras

9.2.1. config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#include <raylib.h>
#define SCREEN_WIDTH 1600
#define SCREEN_HEIGHT 900

#define GRAVITY 9.8f

#define WORLD_WIDTH 24
#define WORLD_HEIGHT 32
#define CANCHA_WIDTH 9.8f
#define CANCHA_HEIGHT 17.8f

#define POS_RING_RED  ((Vector3){  8.74f, 1.3f, 0.0f })
#define POS_RING_BLUE ((Vector3){ -8.74f, 1.3f, 0.0f })

#define N_ROBOTS_PER_TEAM 10
#define RADIUS_BALL 0.1f
#define GREATER 1

#define ONCE_TEAM 0

#define WAIT_TIME_FOR_SHOT 0.5f

#define BACKGROUND_COLOR WHITE

#endif
```

9.2.2. robot.h

```
#ifndef ROBOT_H
#define ROBOT_H

#include <frame.h>
#include <ball.h>

#include <raylib.h>
#include <stdio.h>
#include <stdint.h>

/*
bits del robot->config
      0: set if is collision
      1: if set collision up else collision down
      2: if set collision right else collision left
      3: check collision X
      4: check collision Z
      5: set if is rotating
      6: set if is shooting
      7: set if is launching
*/

#define ROTATING             (uint8_t)5
#define SHOOTING             (uint8_t)6
```

```
#define LAUNCHING            (uint8_t)7

#define    SET_BIT(bit, pos) (uint8_t)((bit) | (1 << (pos)))
#define  UNSET_BIT(bit, pos) (uint8_t)((bit) & (~(1 << (pos))))
#define  CHECK_BIT(bit, pos) (uint8_t)(((bit) & (1 << (pos))) >> (
    pos))

#define    SET_ROTATING(bit) ((bit) |= (uint8_t)(SET_BIT(bit,
    ROTATING)))
#define  UNSET_ROTATING(bit) ((bit) &= (uint8_t)(UNSET_BIT(bit,
    ROTATING)))
#define    SET_SHOOTING(bit) ((bit) |= (uint8_t)(SET_BIT(bit,
    SHOOTING)))
#define  UNSET_SHOOTING(bit) ((bit) &= (uint8_t)(UNSET_BIT(bit,
    SHOOTING)))
#define   SET_LAUNCHING(bit) ((bit) |= (uint8_t)(SET_BIT(bit,
    LAUNCHING)))
#define UNSET_LAUNCHING(bit) ((bit) &= (uint8_t)(UNSET_BIT(bit,
    LAUNCHING)))

typedef struct RobotUniciclo {
    frame_t *obj;
    float w;
    float old_w;
    float *y_rotation;
    float y_rotation_expected;
    float old_y_rotation;
    float vl;
    float old_vl;
    float steps;
    uint8_t config;
    float time_to_shot;
    float wait_for_shot;
    ball_t *ball;
    Vector3 target;
    Color team;
} uniciclo_t;

extern uniciclo_t* create_robot(Vector3 pos, Color team);
extern void destroy_robot(uniciclo_t* robot);
extern void update_robot(uniciclo_t* robot);
extern void rotate_robot(uniciclo_t* robot);
extern bool ready_robot_rotation(uniciclo_t *robot);
extern void configure_robot_rotation(uniciclo_t *robot, float
    rotation_target);
extern void move_robot(uniciclo_t* robot);
extern void draw_robot(uniciclo_t *robot);
extern float get_angle_diff(float current_angle, float new_angle);

#endif
```

9.2.3. ball.h

```
#ifndef BALL_H
#define BALL_H

#include <graphics.h>
```

```
typedef struct Ball {
    Vector3 *position;
    Vector3 velocity;
    float fly_time;
    float t;
    bool visible;
    sphere_t *sphere;
} ball_t;

extern ball_t* create_ball();
extern void destroy_ball(ball_t *ball);
extern void draw_ball(ball_t *ball);
extern void update_ball(ball_t *ball);
#endif
```

9.2.4. collision.h

```
#ifndef COLLISION_H
#define COLLISION_H

#include <robot.h>

#include <raylib.h>

#define COLLISION         (uint8_t)0
#define COLLISION_X       (uint8_t)1
#define COLLISION_Z       (uint8_t)2
#define CHECK_COLLISION_X (uint8_t)3
#define CHECK_COLLISION_Z (uint8_t)4

#define NO_COLLISION(bit)    ((bit) &= (uint8_t)(~0x1f))
#define COLLISION_UP(bit)    ((bit) |= (uint8_t)((SET_BIT(  bit,
   COLLISION_X)) | (SET_BIT(bit, COLLISION)) | (SET_BIT(bit,
   CHECK_COLLISION_X))))
#define COLLISION_DOWN(bit)  ((bit) |= (uint8_t)((UNSET_BIT(bit,
   COLLISION_X)) | (SET_BIT(bit, COLLISION)) | (SET_BIT(bit,
   CHECK_COLLISION_X))))
#define COLLISION_RIGHT(bit) ((bit) |= (uint8_t)((SET_BIT(  bit,
   COLLISION_Z)) | (SET_BIT(bit, COLLISION)) | (SET_BIT(bit,
   CHECK_COLLISION_Z))))
#define COLLISION_LEFT(bit)  ((bit) |= (uint8_t)((UNSET_BIT(bit,
   COLLISION_Z)) | (SET_BIT(bit, COLLISION)) | (SET_BIT(bit,
   CHECK_COLLISION_Z))))

extern void check_camera_collision(Camera3D *camera);
extern void handle_robot_collision(uniciclo_t *robot);

#endif
```

9.2.5. utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <config.h>

#include <raymath.h>

float get_angle_diff(float current_angle, float new_angle);
```

```
float get_target_angle(Vector3 position, Vector3 target);
float get_shot_velocity(float distance);
float get_shot_angle(float distance, float velocity, float h);

#endif
```

9.2.6. frame.h

```
#ifndef FRAME_H
#define FRAME_H

#include <linked_list.h>

#include <raylib.h>
#include <stdio.h>

typedef struct Frame {
    Vector3 position;
    Vector3 rotation;
    linked_list_t *entities;
} frame_t;

extern frame_t *create_frame(Vector3 position, Vector3 rotation);
extern void add_entity(frame_t *frame, void *entity);
extern void destroy_frame(frame_t *frame);
extern void draw_frame(frame_t *frame);

#endif
```

9.2.7. graphics.h

```
#ifndef GRAPHICS_H
#define GRAPHICS_H

#include <stdio.h>
#include <objects.h>
#include <raylib.h>

typedef struct _Rectangle {
    size_t width, height;
    Vector2 x;
    Vector2 *corners;
    float rotation;
} rectangle_t;

extern void draw_cube(cube_t cube);
extern void draw_cylinder(cylinder_t cylinder);
extern void draw_sphere(sphere_t sphere);

#endif
```

9.2.8. objects.h

```
#ifndef OBJECTS_H
#define OBJECTS_H

#include <raylib.h>

typedef struct _Object {
```

```
    Vector3 position;
    Vector3 rotation;
    Model model;
    Color color;
} object_t;

typedef struct _Cube {
    object_t object;
    Vector3 size;
} cube_t;

typedef struct _Sphere {
    object_t object;
    float radius;
    void *padding_;
} sphere_t;

typedef struct _Cylinder {
    object_t object;
    float radius;
    float height;
    void *padding_;
} cylinder_t;

extern cube_t *create_cube(Vector3 size, Vector3 position, Vector3
    rotation, Color color);
extern sphere_t *create_sphere(float radius, Vector3 position,
   Vector3 rotation, Color color);
extern cylinder_t *create_cylinder(float radius, float height,
   Vector3 position, Vector3 rotation, Color color);

extern void destroy_cube(cube_t *cube);
extern void destroy_sphere(sphere_t *sphere);
extern void destroy_cylinder(cylinder_t *cylinder);

#endif
```

9.2.9. random.h

```
#ifndef RANDOM_H
#define RANDOM_H

#include <stddef.h>

extern float get_random_float(float min, float max);
extern size_t get_random_size_t(size_t min, size_t max);

#endif
```

9.2.10. linked_list.h

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include <stdio.h>
#include <stdbool.h>

typedef struct Node {
    void *data;
```

```
    struct Node* next;
    struct Node* prev;
} node_t;


typedef struct LinkedList {
    node_t *head;
    node_t *tail;
    size_t size;
} linked_list_t;


extern linked_list_t* create_linked_list();
extern void append(linked_list_t *l, void *data);
extern void **get_array(linked_list_t *list);
extern void delete_at(linked_list_t* l, size_t position);
extern void destroy_linked_list(linked_list_t* l);

#endif
```