```cpp
#include "../hdr/Window.h"
#include "../hdr/GameWorld.h"

void update(Window* window, GameWorld* world) {
    SDL_Event* event = new SDL_Event();

    if (SDL_PollEvent(event)) {
        world->update(event);
        window->update(event); }

    delete event;
    event = nullptr;
}

int main(int argc, char* argv[]) {
    Window* window = new Window("Test window", 800, 600);
    GameWorld* world = new GameWorld(window);

    while (window->isRunning()) {
        update(window, world);
        world->render();
        window->clear(); }


    if (window) { delete window; }
    if (world) { delete world; }

    window = nullptr; world = nullptr;
    return 0;
}




```

```
#pragma once
#include <string>
#include <SDL.h>

class Window {
private:
    bool _is_running = true;
    SDL_Window* _window = nullptr;
    SDL_Renderer* _renderer = nullptr;

    std::string _title;
    int _width, _height;

public:
    Window(const std::string& title, int width, int height);
    ~Window();

    inline bool isRunning() const { return _is_running; };
    inline SDL_Renderer* getRenderer() const { return _renderer; }

    void update(SDL_Event* event);
    void clear() const;

private:
    bool init();
};
```

```cpp
#include <iostream>
#include "../hdr/Window.h"

Window::Window(const std::string& title, int width, int height)
    : _title(title), _width(width), _height(height) {
    _is_running = init(); }

Window::~Window() {
    SDL_DestroyRenderer(_renderer);
    SDL_DestroyWindow(_window);
    SDL_Quit(); }



void Window::update(SDL_Event* event) {
    if (event->type == SDL_QUIT) { _is_running = false; }
    else if (event->type == SDL_KEYDOWN) {
        if (event->key.keysym.sym == SDLK_ESCAPE) { _is_running = false; }
    }
}

// Draws a rectangle
void Window::clear() const {
    SDL_RenderPresent(_renderer);   // Show changes
    SDL_SetRenderDrawColor(_renderer, 0x00, 0x00, 0xFF, 0xFF);
    SDL_RenderClear(_renderer);     // Apply background changes
}



bool Window::init() {
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        std::cerr <<"Bad SDL_Video init.\n";
        return 0;}

    _window = SDL_CreateWindow(_title.c_str(), SDL_WINDOWPOS_CENTERED,
        SDL_WINDOWPOS_CENTERED, _width, _height, 0);
    if (!_window) {
        std::cerr << "Bad window instantiation.\n";
        return 0; }

    _renderer = SDL_CreateRenderer(_window, -1, SDL_RENDERER_ACCELERATED);
    if (!_renderer) {
        std::cerr << "Bad renderer instantiation.\n";
        return 0; }

    return true;
}

```

```cpp
#pragma once
#include <vector>
#include "Window.h"
#include "Player.h"

class GameWorld {
private:
    SDL_Renderer* _renderer = nullptr;
    Window* _window = nullptr;

    std::vector<Shape::Shape*> _shapes;
    Player* _player = nullptr;

public:
    GameWorld(Window* window);
    ~GameWorld();

    void update(SDL_Event* event);
    void render();

private:
    inline void addShape(Shape::Shape* shape) { _shapes.emplace_back
      (shape); }
    std::pair<int, int> handleInput(SDL_Event* event);

    bool checkCollision(Shape::Shape* shape1, Shape::Shape* shape2);
    bool checkRectCollision(Shape::Rect* rect1, Shape::Rect* rect2);
    bool checkCircleCollision(Shape::Circle* circle1, Shape::Circle*
      circle2);


    void handleCollisions(std::pair<int, int> prev_entity_pos);
    double distanceSquared(SDL_Point p1, SDL_Point p2);
};



```

```
#include <iostream>
#include "../hdr/GameWorld.h"

GameWorld::GameWorld(Window* window) {
    _window = window;

    if (_window) {
        Shape::ColourRGBA test_colour;
        Shape::ColourRGBA player_colour;
        player_colour.b = 0;
        player_colour.r = 150;

        Shape::ColourRGBA highlight_colour;
        highlight_colour.g = 120;
        highlight_colour.r = 0;

        SDL_Point origin_1;
        origin_1.x = 400;
        origin_1.y = 250;


        SDL_Point origin_2;
        origin_2.x = 100;
        origin_2.y = 250;

        _renderer = _window->getRenderer();
        Shape::Shape* player_rect = new Shape::Rect(_window, 120, 120,    ⤾
          100, 100,
            Shape::ShapeType::RECT, player_colour);
        Shape::Shape* player_circle = new Shape::Circle(_window, origin_2, ⤾
           40,
            Shape::ShapeType::CIRCLE, player_colour);
        Shape::Shape* rect_test = new Shape::Rect(_window, 50, 300, 600,   ⤾
          50,
            Shape::ShapeType::RECT, test_colour);
        Shape::Shape* circle_test = new Shape::Circle(_window, origin_1,   ⤾
          20,
            Shape::ShapeType::CIRCLE, test_colour);


        bool is_player_rect = false;
        if (is_player_rect) {
            _player = new Player(window, 120, 120, 120, 100,               ⤾
              player_colour,
                highlight_colour, player_rect);
            addShape(player_rect);
        }
        else {
            _player = new Player(window, 120, 120, origin_2.x - 20,       ⤾
                origin_2.y - 20, player_colour,
                  highlight_colour, player_circle);
            addShape(player_circle);
        }

        addShape(rect_test);
        addShape(circle_test);

    }
}

GameWorld::~GameWorld() {
    if (_player) { delete _player; _player = nullptr; }
    for (auto shape : _shapes) { delete shape; shape = nullptr; }
}

void GameWorld::update(SDL_Event* event) {
    std::pair<int, int> init_player_pos = _player->getPos();

    std::pair<int, int> move_data = handleInput(event);
    _player->update(move_data);

    std::vector<Shape::Shape*>::iterator shapes_it = _shapes.begin() + 1;
    for (shapes_it; shapes_it != _shapes.end(); ++shapes_it) {
        if (checkCollision(_player->getCollider(), *shapes_it)) {
            handleCollisions(init_player_pos);
        }
    }
}

void GameWorld::render() {
    _player->render();

    if (_shapes.size() > 1) {
        std::vector<Shape::Shape*>::iterator shapes_it = _shapes.begin() + 
           1;

        for (shapes_it; shapes_it != _shapes.end(); ++shapes_it) {
            (*shapes_it)->render(); }
    }
}


// first int for axis (0 for x, 1 for y) second for direction (0 for -, 1 
  for +).
std::pair<int, int> GameWorld::handleInput(SDL_Event* event) {
    if (event->type == SDL_KEYDOWN) {
        std::pair<int, int> input_data;

        switch (event->key.keysym.sym) {
        case SDLK_LEFT:
            input_data = { 0, 0 };
            break;
        case SDLK_RIGHT:
            input_data = { 0, 1 };
            break;
        case SDLK_UP:
            input_data = { 1, 0 };
            break;
        case SDLK_DOWN:
            input_data = { 1, 1 };
            break;
        default:
            input_data = { -1, -1 };
            break;
        }

        return input_data;
    }
}

bool GameWorld::checkCollision(Shape::Shape* shape1, Shape::Shape* shape2) 
   {
    auto shape_type_1 = shape1->getType();
    auto shape_type_2 = shape2->getType();

    switch (shape_type_1) {
    case Shape::ShapeType::RECT:
        switch (shape_type_2) {
        case Shape::ShapeType::RECT:
            return checkRectCollision((Shape::Rect*)shape1, (Shape::Rect*) 
              shape2);
        case::Shape::ShapeType::CIRCLE:
            return false;
        default: return false;
        }       //  Not implemented

    case Shape::ShapeType::CIRCLE: {
        switch (shape_type_2) {
        case Shape::ShapeType::RECT:
            return false;
        case Shape::ShapeType::CIRCLE:
            return checkCircleCollision((Shape::Circle*)shape1, 
              (Shape::Circle*)shape2);
        default: return false; }
        } default: return false;
    }

    return false;
}

bool GameWorld::checkRectCollision(Shape::Rect* rect1, Shape::Rect* rect2) 
   {
    auto[x1, y1] = rect1->getPos();
    auto[x2, y2] = rect2->getPos();
    auto[w1, h1] = rect1->getDimensions();
    auto[w2, h2] = rect2->getDimensions();


    if ( x1 < x2 + w2 &&
         x1 + w1 > x2 &&
         y1 < y2 + h2 &&
         y1 + h1 > y2) {
        return true; }

    return false;
}

bool GameWorld::checkCircleCollision(Shape::Circle* circle1, 
  Shape::Circle* circle2) {
    int r1 = circle1->getRadius();
    int r2 = circle2->getRadius();
    int total_r_squared = (r1 + r2) * (r1 + r2);
    double dist_squared = distanceSquared(circle1->getOrigin(), circle2-
      >getOrigin());

    return dist_squared < total_r_squared ; }

void GameWorld::handleCollisions(std::pair<int, int> prev_entity_pos) {
    _player->displaying_highlighted = !_player->displaying_highlighted;
    _player->setPos(prev_entity_pos); }

double GameWorld::distanceSquared(SDL_Point p1, SDL_Point p2) {
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;

    return dx*dx + dy*dy; }

```

```cpp
#pragma once
#include "Rect.h"
#include "Circle.h"

class Player {
private:
    // Cleanup handled by the GameWorld
    Shape::Shape* _collider = nullptr;
    Shape::ColourRGBA _colour_a, _colour_b;

    int _x, _y;
    int _velocity = 5;

public:
    bool displaying_highlighted = false;

public:
    Player(Window* window, int w, int h, int x, int y, Shape::ColourRGBA colour,
        Shape::ColourRGBA collision_colour, Shape::Shape* collider = nullptr);

    void setPos(std::pair<int, int> pos);

    inline std::pair<int, int> getPos() { return {_x, _y}; }
    inline Shape::Shape* getCollider() { return _collider; }
    inline Shape::ColourRGBA getColour() { return _colour_a; }
    inline Shape::ColourRGBA getHighlightedColour() { return _colour_b; }

    void update(std::pair<int, int> move_data);
    void render();
};
```

```cpp
#include "../hdr/Player.h"

Player::Player(Window* window, int w, int h, int x, int y,
    Shape::ColourRGBA colour, Shape::ColourRGBA collision_colour,
    Shape::Shape* collider) :
    _x(x), _y(y), _colour_a(colour), _colour_b(collision_colour) {

    if (collider == nullptr) {
        _collider = new Shape::Rect(window, w, h, x, y,
          Shape::ShapeType::RECT, colour); }
    else {
        _collider = collider;
        _colour_a = _collider->getColour(); }
}

void Player::setPos(std::pair<int, int> pos) {
    auto[x, y] = pos;
    _x = x;
    _y = y;

    _collider->setPos({_x, _y}); }


void Player::update(std::pair<int, int> move_data) {
    auto [axis, direction] = move_data;

    if (!axis) {    // Horizontal movement
        if (!direction) {   // move left
            setPos({_x - _velocity, _y}); }
        else if (direction == 1){ setPos({_x + _velocity, _y}); }
    } else {    // Vertical movement
        if (!direction) {   // move up
            setPos({_x, _y - _velocity});
        } else if (direction == 1) { setPos({_x, _y + _velocity}); }
    }

    if (displaying_highlighted) {
        _collider->setColour(_colour_b); }
    else { _collider->setColour(_colour_a); }
}

void Player::render() {
    if (_collider) _collider->render(); }

```

```cpp
#pragma once
#include <utility>
#include <SDL.h>

namespace Shape {
    enum class ShapeType {
        INVALID,
        RECT,
        CIRCLE,
        LINE,
    };

    struct ColourRGBA { int r=255, g=255, b=255, a=255; };

    class Shape {
    protected:
        SDL_Renderer* _renderer = nullptr;

        int _x=0, _y=0;
        ShapeType _type = ShapeType::INVALID;
        ColourRGBA _colour;
        SDL_Rect* _bounds = nullptr;

    public:
        inline std::pair<int, int> getPos() { return { _x, _y }; }
        inline ShapeType getType() { return _type; }
        inline ColourRGBA getColour() { return _colour; }

        virtual void setPos(std::pair<int, int> pos) = 0;
        inline void setType(ShapeType type) { _type = type; }
        inline void setColour(ColourRGBA colour) { _colour = colour; }

        virtual void render() = 0;

    protected:
        virtual void findBounds() = 0;
    };
}


```

```cpp
#pragma once
#include "Window.h"
#include "Shape.h"

/*  For now im reluctant to call this rectangle class a 2d renderer or
*   collider mesh or anything as given the trivial nature of the software
*   it's ok for the rect to do both. Therefore, in this case it does both
*   paint to renderer and handle collisions                             */

namespace Shape {
    class Rect : public Shape {
    private:
        int _w, _h;

    public:
        //  Could just move that data into structs if i wanted
        Rect();
        Rect(Window* window, int w, int h, int x, int y, ShapeType type,
          ColourRGBA colour);

        void setPos(std::pair<int, int> pos) override;
        inline std::pair<int, int> getDimensions() { return {_w, _h}; }
        inline void resize(int new_w, int new_h) { _w=new_w; _h=new_h; }

        void render() override;

    private:
        inline void setDimensions(int w, int h, int x, int y) { _w=w; _h=h;
            _x=x; _y = y; }
        void findBounds() override;
    };
}


```

```cpp
#include "../hdr/Rect.h"
#include <iostream>

Shape::Rect::Rect() {
    _x = 0;
    _y = 0;
    _w = 0;
    _h = 0;
    _type = ShapeType::INVALID;
}

Shape::Rect::Rect(Window* window, int w, int h, int x, int y,
    ShapeType type, ColourRGBA colour) : _w(w), _h(h) {
    _x = x; _y = y;
    _type = type;
    _colour = colour;
    findBounds();
    _renderer = window->getRenderer();

    if (_renderer == nullptr) {
        std::cerr << "Rect entity couldn't instance renderer.\n"; }
}

void Shape::Rect::setPos(std::pair<int, int> pos) {
    auto [x, y] = pos;

    _x = x;
    _y = y;
    findBounds(); }

void Shape::Rect::render() {
    SDL_SetRenderDrawColor(_renderer, _colour.r, _colour.g, _colour.b, 
      _colour.a);
    SDL_RenderFillRect(_renderer, _bounds); }

void Shape::Rect::findBounds() {
    if (_bounds) { delete _bounds; _bounds = nullptr; }

    _bounds = new SDL_Rect();
    _bounds->w = _w;
    _bounds->h = _h;
    _bounds->x = _x;
    _bounds->y = _y; }

```

```

#pragma once
#include "Window.h"
#include "Shape.h"

namespace Shape {
    class Circle : public Shape {
    private:
        SDL_Point _origin;
        int _radius;

    public:
        Circle(Window* window, SDL_Point origin, int radius,
            ShapeType type, ColourRGBA colour);

        inline SDL_Point getOrigin() { return _origin; }
        inline int getRadius() { return _radius; }

        void setPos(std::pair<int, int> pos);
        inline void setOrigin(SDL_Point new_origin) { _origin =
          new_origin; }
        inline void setRadius(int new_radius) { _radius = new_radius; }

        void render() override;

    private:
        void findBounds() override;
    };
}


```

```cpp
#include <iostream>
#include <algorithm>
#include "../hdr/Circle.h"

Shape::Circle::Circle(Window* window, SDL_Point origin, int radius,
    ShapeType type, ColourRGBA colour) : _origin(origin), _radius(radius) {
    _type = type;
    _colour = colour;
    findBounds();
    _renderer = window->getRenderer();

    if (_renderer == nullptr) {
        std::cerr << "Circle entity couldn't instance renderer.\n"; }
}


void Shape::Circle::setPos(std::pair<int, int> pos) {
    auto [x, y] = pos;
    float half_rad = _radius / 2;

    SDL_Point new_origin;
    new_origin.x = x + half_rad;
    new_origin.y = y + half_rad;

    setOrigin(new_origin);
    findBounds();
}

void Shape::Circle::render() {
    if (_renderer) {
        int renderer_width, renderer_height;    // Clamp values
        SDL_GetRendererOutputSize(_renderer, &renderer_width, &renderer_height);
        SDL_SetRenderDrawColor(_renderer, _colour.r, _colour.g, _colour.b, _colour.a);

        int x0 = _origin.x;                     // x origin
        int y0 = _origin.y;                     // y origin
        int x1 = _bounds->x;                        // x initial
        int x2 = _bounds->x + _bounds->w;           // x final
        int y1 = _bounds->y;                    // y initial
        int y2 = _bounds->y + _bounds->h;           // y final

        x1 = std::clamp(x1, 0, renderer_width);
        x2 = std::clamp(x2, 0, renderer_width);
        y1 = std::clamp(y1, 0, renderer_width);
        y2 = std::clamp(y2, 0, renderer_width);

        for (int x = x1; x < x2; ++x) {
            for (int y = y1; y < y2; ++y) {
                // If >= 0 we have a renderable pixel (filled circles if outline check == 0)
                int render_value = (x-x0)*(x-x0) + (y-y0)*(y-y0) - _radius*_radius;
                if (render_value <= 0) { SDL_RenderDrawPoint(_renderer, x, y); }
            }
        }
    }
}

void Shape::Circle::findBounds() {
    int x1 = _origin.x - _radius;
    int y1 = _origin.y - _radius;
    int diameter = 2 * _radius;

    if (_bounds) { delete _bounds; _bounds = nullptr; }
    _bounds = new SDL_Rect();

    _bounds->w = diameter;
    _bounds->h = diameter;
    _bounds->x = x1;
    _bounds->y = y1;
}

```