SDL2 Course Day-7
Jumping & Platformer Physics

Couple of small things before delving into the main topics:

In C++, a < b < c cannot be used as a condition. We must specify a < b && b < c to represent the same. Correct the "is_wall" function of your Map_Object.cpp file like so:

```
bool Map_Object::is_wall(int x_index, int y_index) {
    if (!(0 <= x_index && x_index <= cols))
        return false;
    if (!(0 <= y_index && y_index <= rows))
        return false;
    return (tile_map[y_index][x_index] == 1);
}
```

To set a value to default use '{}', for an int the default would be 0, for a bool it would be false, for a float 0.0 and so on.
For example:

```
bool move_left{};
```

<u>Changing Our View</u>:

To jump, we need to decrease the y value of the player rectangle by a big value to move the player up and then introduce gravity to make the player fall, like on Earth.

To do this, we need our player to have these three attributes.
Add these attributes to the Player_Object.h file:

```
int jump_acc = 15;
int grav_acc = 2;
int y_vel{};
```

In all the code we have written so far, we are looking at a top-down view of the game. For the purpose of jumping, lets now use the side-scrolling view.

In a top-down view, we can move- up, down, left or right and we used an enum as we can only move in only one direction at one.

In a side-scrolling view we can move- left, right and jump.
We can also move in multiple directions at once- so an enum is replaced by 3 boolean variables. move_left, move_right and jump.

```
bool move_left{};
bool jump{};
bool move_right{};
```

The handle input and update functions are modified accordingly. The input sets the values of these variables to true on key down and false on key up.

```
void Player_Object::handle_input(SDL_Event event) {
    old_x = rect.x;
    old_y = rect.y;
    if (event.type == SDL_KEYDOWN) {
        if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
            move_right = true;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
            move_left = true;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
            jump = true;
        }
    } else if (event.type == SDL_KEYUP) {
        if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
            move_right = false;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
            move_left = false;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
            jump = false;
        }
    }
```

The update function checks these variables to change the rectangle's co-ordinates. Note how we only have if statements, not an if-else-if ladder. This serves to allow movement in multiple directions at once.

```
void Player_Object::update(Map_Object *map) {
    y_vel += grav_acc;
    if (y_vel > map->cell_height){
        y_vel = map->cell_height - 1;
    }

    if (move_right){
        rect.x += speed;
    }
    if (move_left){
        rect.x -= speed;
    }
    if (jump){
        y_vel = -jump_acc;
    }

    rect.y += y_vel;
}
```

Before actually jumping, let's talk about GRAVITY.

Gravity:

Gravity in physics is a constant downward acceleration. Similarly, in our game, for each frame a constant downward (+ve) acceleration is added to the y_velocity of the rectangle.

However, if we keep increasing the y_velocity infinitely, we will end up phasing past some rectangles if the value becomes greater than that of the walls.

To prevent this, we add a check to stop increasing the y_velocity when a value of wall_height is reached, after which we keep it constant.

Initial gravity in the Player_Object.h file

```
int grav_acc = 2;
```

Modify the Player_Object.cpp as so:

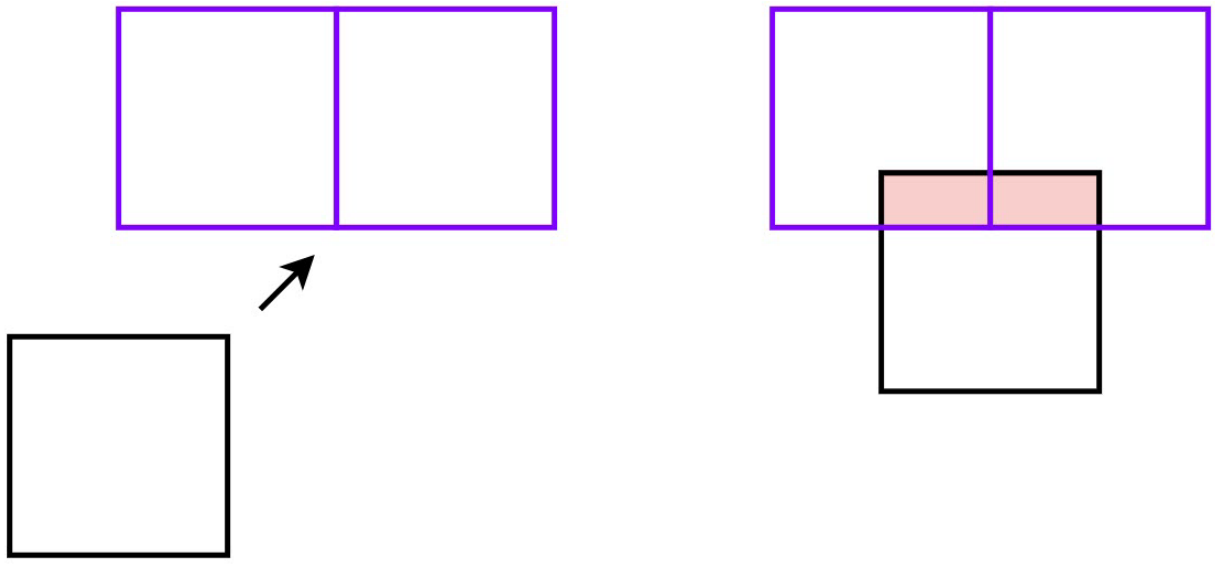
```
void Player_Object::update(Map_Object *map) {
    y_vel += grav_acc;
    if (y_vel > map->cell_height){
        y_vel = map->cell_height - 1;
    }

    if (move_right){
        rect.x += speed;
    }
    if (move_left){
        rect.x -= speed;
    }
    if (jump){
        y_vel = -jump_acc;
    }

    rect.y += y_vel;
}
```

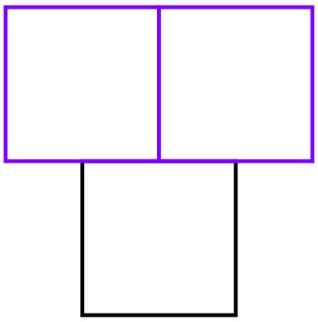
If you run the code at this point you will notice that gravity works fine but if you press the right arrow key, you move left, which is not the way its supposed to run :(.

Consider the scenario:

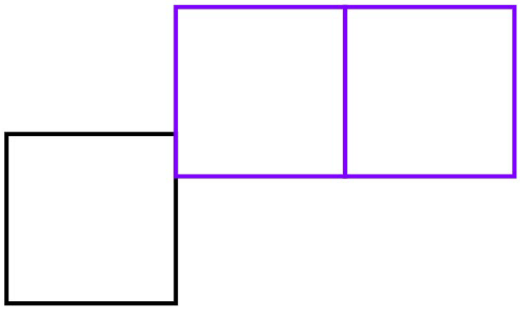


Lets say the player is moving diagonally and its new position is in between two walls.

What is supposed to happen:



What Actually Happens:



This is due to the fact that our current collision is checking only one direction at a time instead of multiple directions. So we need to change our collision logic.

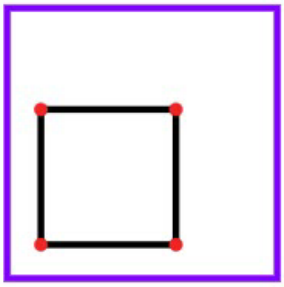
Collision Detection:

The first change we will make is change how we are detecting collision:

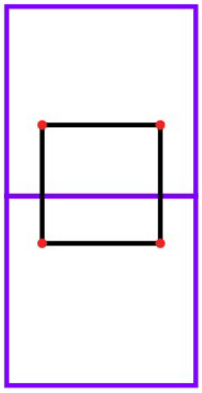
Since our map contains 1's and 0's where 1 is a wall, if we converted the rectangles position into map indexes then we can see if that index is a 1 or 0 and note accordingly whether the player is colliding or not.

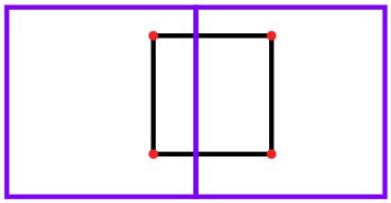
Look at it this way- any collision can have one of 3 outcomes:

1> The player rectangle fits exactly within the wall's rectangle.

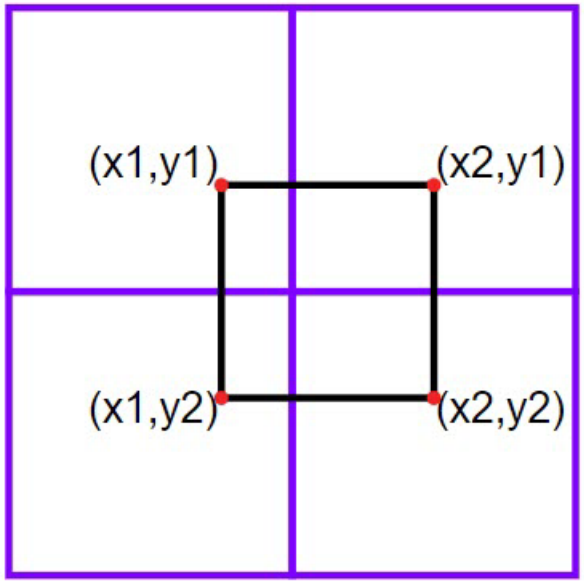


2> It lies in 2 adjacent boxes (adjacent side by side or top and down)



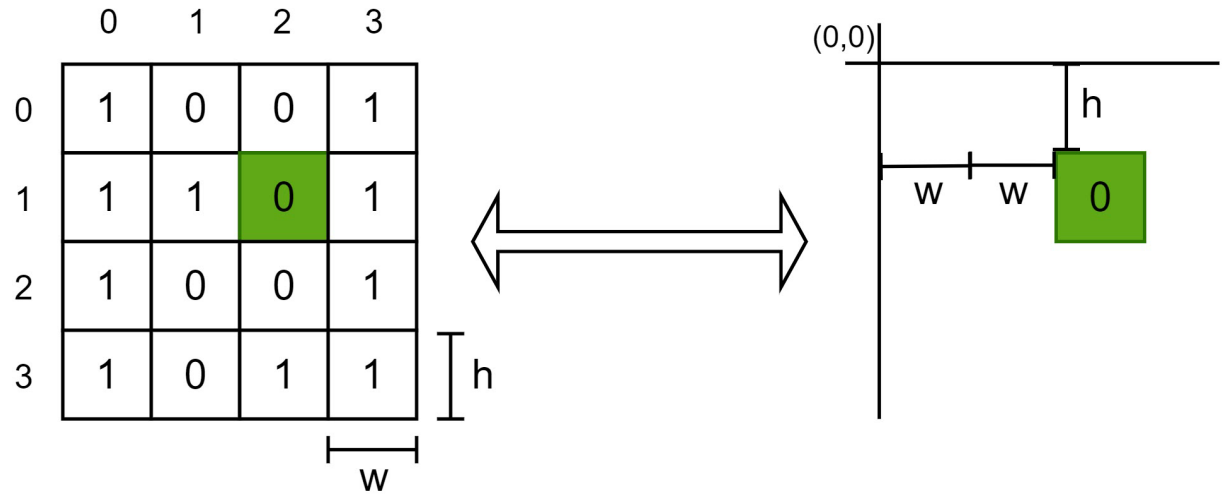


3> It lies in parts in 4 different cells.



To check for collision any of the 4 vertices of the rectangle must satisfy one of the above constraints.

The 4 points x1, x2, y1 and y2 are currently in terms of the window dimensions. We convert the same to find what position of the tile-map they would lie in.
To do so, divide the x co-ordinates by the width of the cells and the y-coordinates by the height of the cells.



To convert map indexes to position on the window, we used to multiply the index with the cell size. To convert positions on the window to map indexes we perform the opposite operation (division).

In Map_Object.cpp:

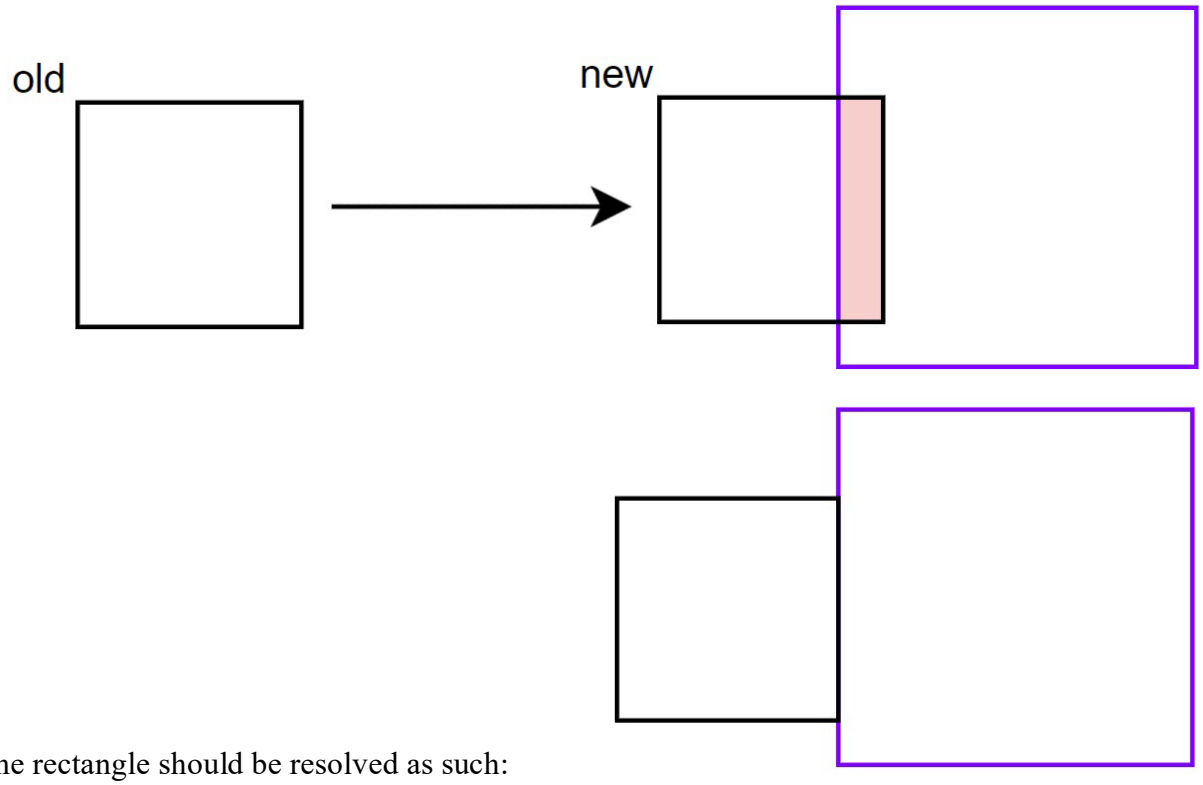
```
bool Map_Object::is_colliding(SDL_Rect &rect) {
    int x1 = (rect.x + 1) / cell_width;
    int x2 = (rect.x + rect.w - 1) / cell_width;
    int y1 = (rect.y + 1) / cell_height;
    int y2 = (rect.y + rect.h - 1) / cell_height;

    return is_wall(x1, y1) ||
           is_wall(x2, y1) ||
           is_wall(x1, y2) ||
           is_wall(x2, y2);

}
```

These random +1 and -1 are used to fix the case wherein the player's rectangle exactly lines up with the wall rectangle and the common edges are considered as colliding even if there is no intersection.

Collision Resolution:



The rectangle should be resolved as such:

Our collision resolution so far is only checking the direction of motion as coming from up, down, left or right and assuming that changing just the x or the y coordinate to the edge of the wall fixes the collision.

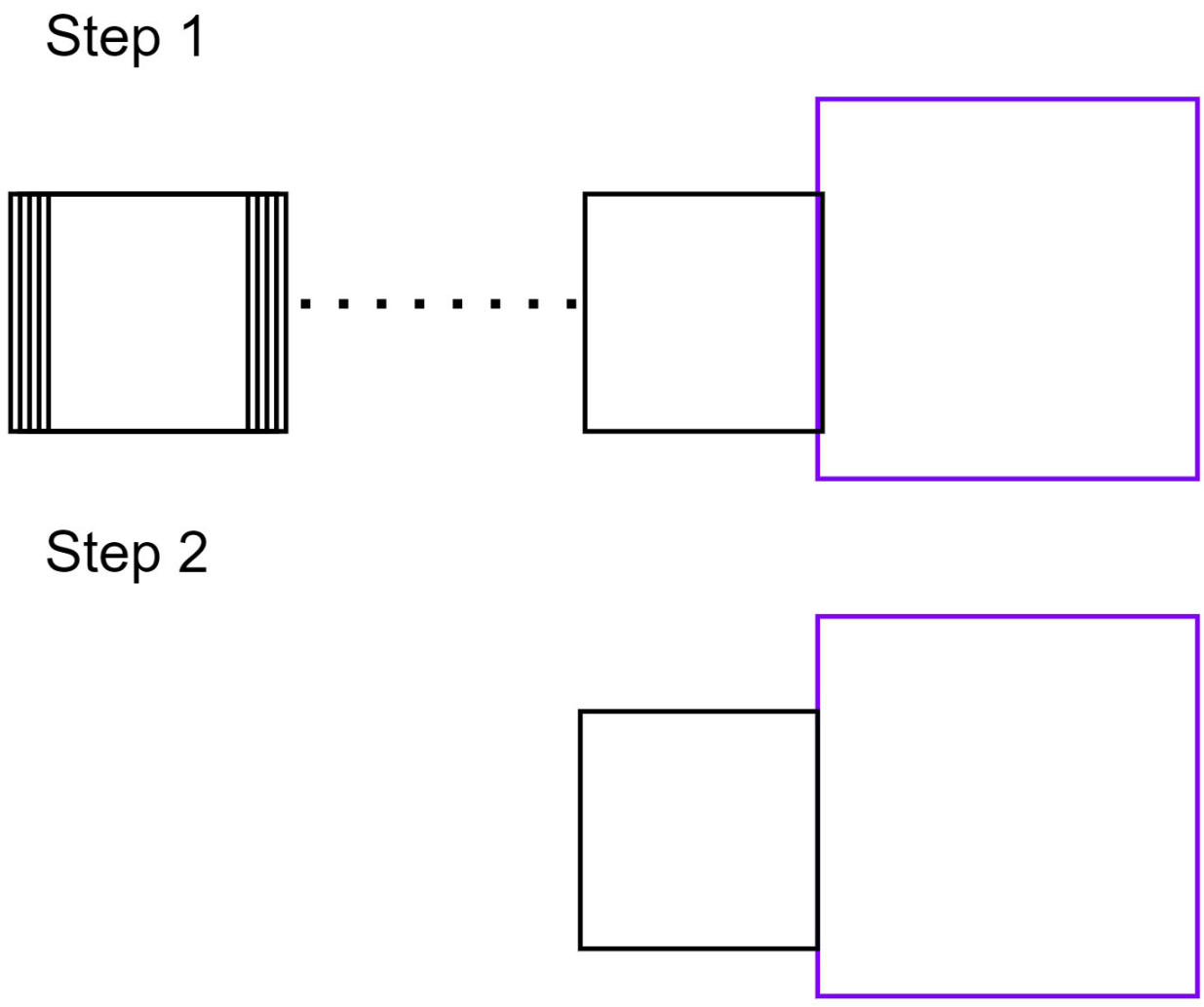
However, in our case there is a problem- if I am moving in up *and* right simultaneously leading to a collision, how do you resolve it?

The first step is to calculate the distance between the old position and the new position, so we know how much we should move in each direction.

Then we reset the rectangle to its old position and start to move by 1px and check whether the new position is colliding or not.

We repeat the above steps until a collision is detected.

We then reset to the position which was 1px behind this.



Since we have two directions to update, we would need to update only one direction at a time, 1px each time.

To prioritize gravity, we update the y direction fully first and then the x direction.

Calculating distance to cover in each direction and resetting to old position:

```
int x_dist = rect.x - old_x;
int y_dist = rect.y - old_y;

rect.x = old_x;
rect.y = old_y;
```

Updating y direction:

```
while (!is_colliding(rect)) {
    old_y = rect.y;

    if (y_dist > 0) {
        rect.y += 1;
        y_dist -= 1;
    } else if (y_dist < 0) {
        rect.y -= 1;
        y_dist += 1;
    } else if (y_dist == 0) {
        break;
    }
}
rect.y = old_y;
```

Then we update x entirely:

```
while (!is_colliding(rect)) {
    old_x = rect.x;

    if (x_dist > 0) {
        rect.x += 1;
        x_dist -= 1;
    } else if (x_dist < 0) {
        rect.x -= 1;
        x_dist += 1;
    } else if (x_dist == 0) {
        break;
    }
}
rect.x = old_x;
```

FINALLY WE CAN JUMP!!!

Define a jump value large enough and modify the Player_Object's update function to increase the y value on hitting the up arrow like so-

```
void Player_Object::update(Map_Object *map) {
    y_vel += grav_acc;
    if (y_vel > map->cell_height){
        y_vel = map->cell_height - 1;
    }

    if (move_right){
        rect.x += speed;
    }
    if (move_left){
        rect.x -= speed;
    }
    if (jump){
        y_vel = -jump_acc;
    }

    rect.y += y_vel;
}
```

TADA You can jump!

The problem with this is the player can now jump infinitely. To prevent this we need to not allow the player to jump when the player is already jumping.
For this we create a Boolean variable called is_jumping.

When the player is on a floor, we reset is_jumping to false and when the player pressed the jump key, then only if is_jumping is false, we allow the player to jump and set is_jumping as true.

```
if (map->is_on_floor(rect)) {
    is_jumping = false;
}
```

```
if (jump && !is_jumping){
    y_vel = -jump_acc;
    is_jumping = true;
}
```

For smoother gameplay, we assign the y_velocity to 0 when it is on the floor or when it touches the ceiling when it jumps to the ceiling.

```
if (map->is_on_floor(rect)) {
    y_vel = 0;
    is_jumping = false;
}
if (map->is_below_ceil(rect) && y_vel < 0) {
    y_vel = 0;
}
```

The implementations for is_on_floor and is_below_ceil is given in github under Day-7.