SDL2 Course Day-5
Collision Detection and Resolution

Collision Detection (Done in last class):
Refer to the following for Collision detection: http://kishimotostudios.com/articles/aabb_collision/
In SDL2, the above method is implemented under the function

```
SDL_HasIntersection(&rect1, &rect2)
```

Returns a Boolean value (True or False)

Collision Resolution (Method 1):
Right now we can only detect if two rectangles have collided or not. If they have collided and we detect it, then we need to reset the position of the player back to its original state.

In method 1, we use two variables old_x and old_y to store the position of the player before it moves. Once it moves, if it collides with anything, the player is reset to its old position.

Creating a variable old_x and old_y in player_object.h:

```
class Player_Object {
public:
    int old_x;
    int old_y;
```

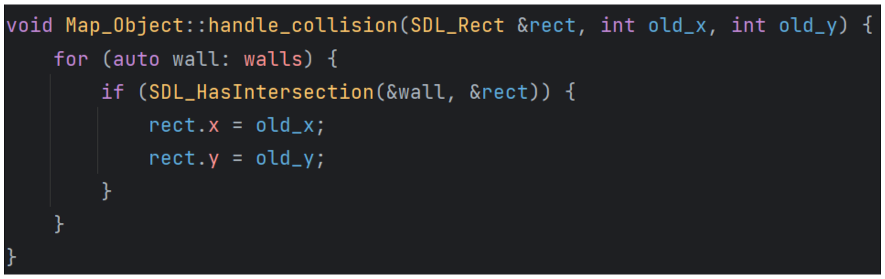
Changing our handle_input function to make it save the player's position before changing:

```
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) {
            rect.x += 10;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
            rect.x -= 10;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
            rect.y -= 10;
        }
        if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) {
            rect.y += 10;
        }
    }
}
```

Since the map owns the tile_map, we give the map object functionality to handle and resolve collsions:
Defining the function in map_object.h

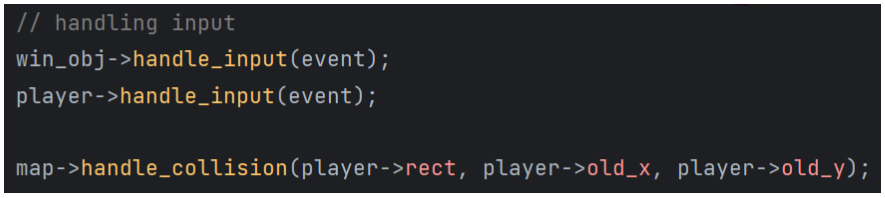
```
void handle_collision(SDL_Rect &rect, int old_x, int old_y);
```

Implementing the function:



```
void Map_Object::handle_collision(SDL_Rect &rect, int old_x, int old_y) {
    for (auto wall: walls) {
        if (SDL_HasIntersection(&wall, &rect)) {
            rect.x = old_x;
            rect.y = old_y;
        }
    }
}
```

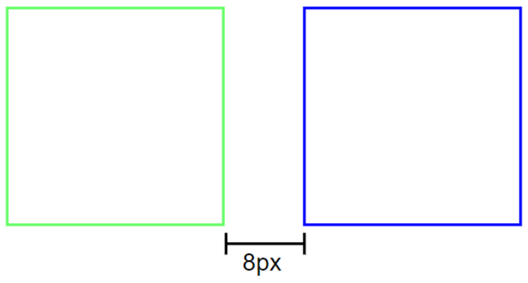
Calling the function in the main function:



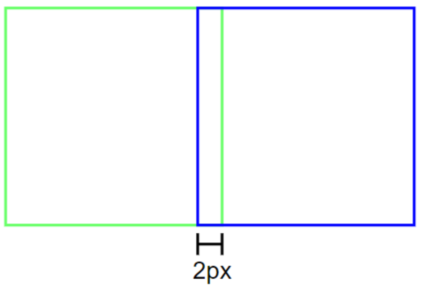
```
// handling input
win_obj->handle_input(event);
player->handle_input(event);

map->handle_collision(player->rect, player->old_x, player->old_y);
```

The problem with this implementation is that if the player's update speed is large enough, it will reset to a position which is a bit far away from the wall, hence creating a gap:
lets say the player (rectangle in green) has an update speed of 10 px and the wall is the rectangle in blue:

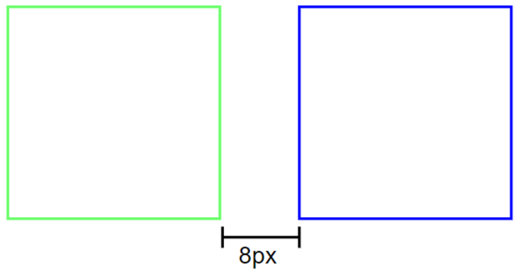
Before:



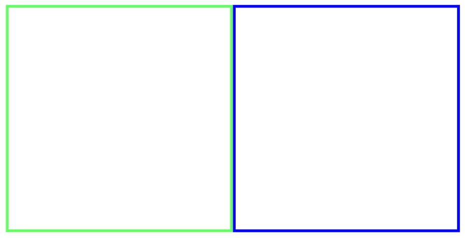
After key-press:



After collision resolution, we just go back to the old position which is 8 pixels away from the wall:



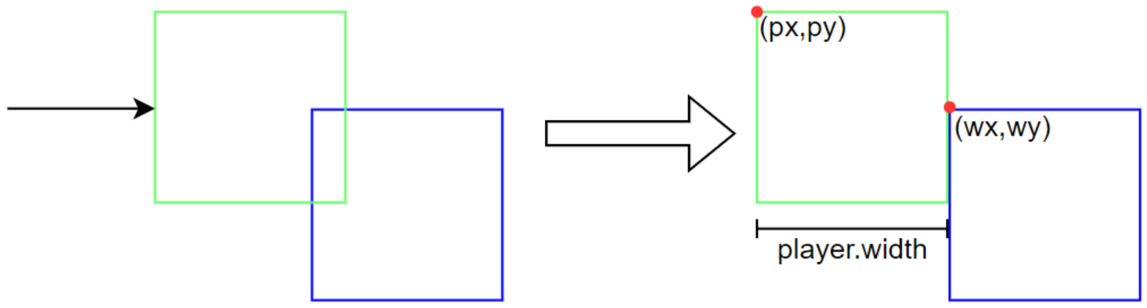
Hence, we will get an ugly gap whenever we try to go close to a wall. To fix this, instead of resetting to the player's old position, we reset to the position just next to wall like below:



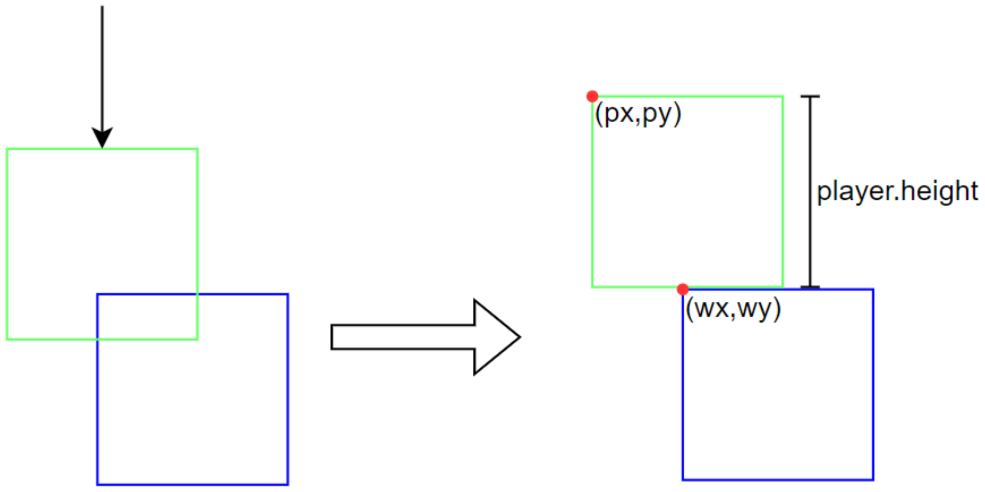
Collision Resolution (Method 2):
To implement the above method, we would need to know the direction in which player is hitting the wall as we can reset the player in 4 different different places (above, below, to the left of and to the right of the wall).

Below are some examples of the 4 cases (For each of the following cases, we need to calculate px and py and we have the knowledge of wx, wy, player's width and height, wall's width and height):

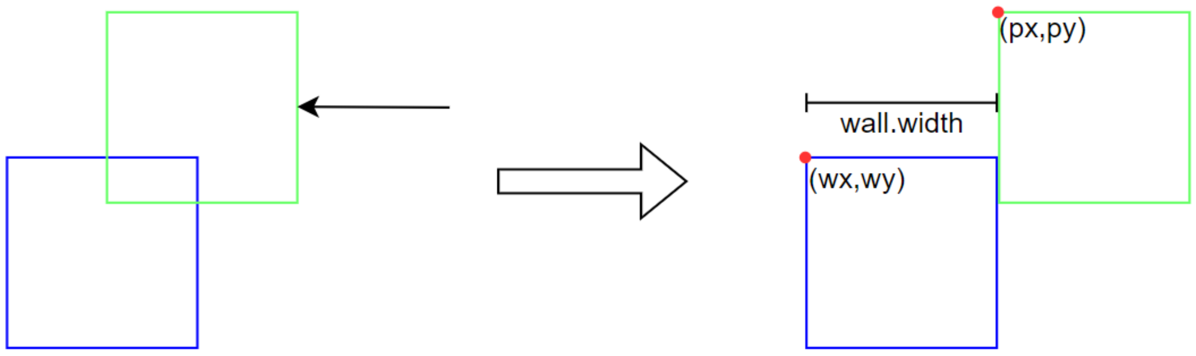
From the left, we can observe that the y co-ordinate does not change, for the x co-ordinate:
px = wx – player.width



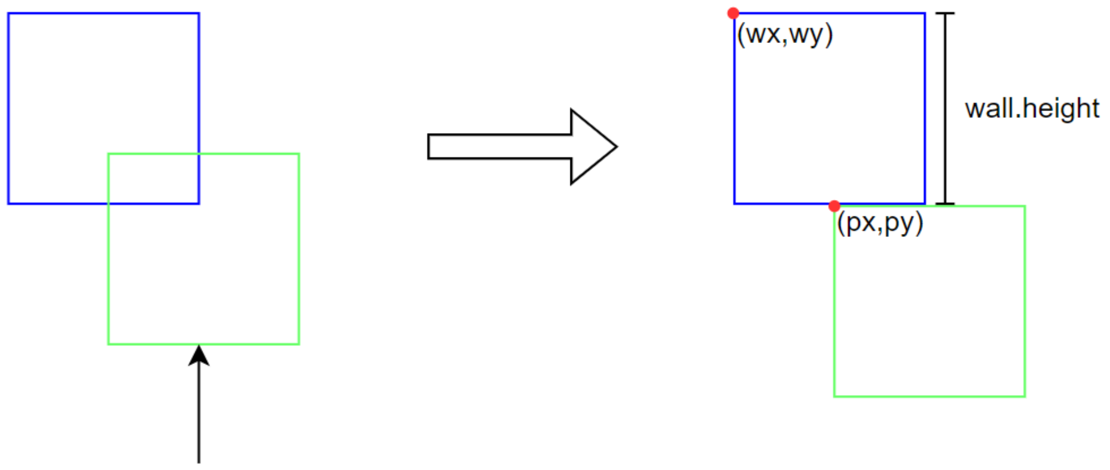
From the top, we can observe that the x co-ordinate does not change, for the y co-ordinate:
py = wy – player.height



From the right, we can observe that the y co-ordinate does not change, for the x co-ordinate:
px = wx + wall.width

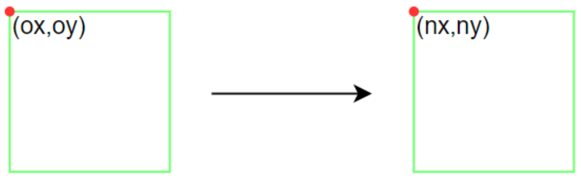


From the top, we can observe that the x co-ordinate does not change, for the y co-ordinate:
py = wy + wall.height

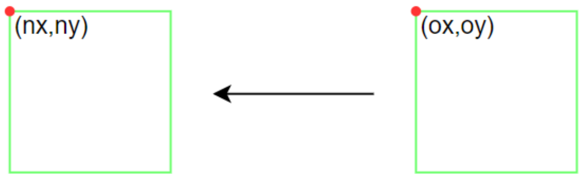


For each case, we would need to know the direction in which the player is hitting the wall; since we know the value of the old position of the player and the current position, we can use these two to calculate the current direction.

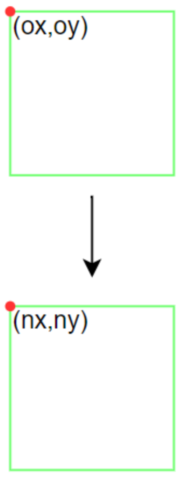
If **(nx > ox)** then -> player is moving right:



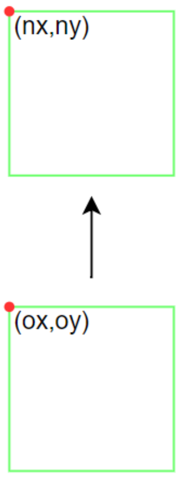
If **(nx < ox)** then -> player is moving left:

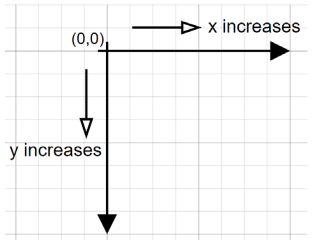


If **(ny > oy)** then -> player is moving down:



If **(ny < oy)** then -> player is moving up:





(the code for above is in github)