SDL2 Course Day-9
Timers, Frame Rate & State Machines

Timers

Timers are objects provided by SDL2 to get the current time since the start of the program in milliseconds.

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);
```

The function that gives this value is:

```
SDL_GetTicks();
```

Rate Limiting

The problem we currently face is that the same code on different hardware will run differently to the user. As on a faster system, the game loop executes faster, and the players moves more, compared to in a slow system, where the game loop executes slower, and the players moves less. This causes different positions of players on different systems. To fix this, we use timers to limit how fast our game executes in each loop.

A Normal Game Loop:

```
int main() {
    initialize_stuff();

    while (true) {
        take_input();
        update();
        draw();
    }

    destroy_stuff();
}
```

Method 1

```
int main() {
    initialize_stuff();

    while (true) {
        start_time = SDL_GetTicks();

        take_input();
        update();
        draw();

        end_time = SDL_GetTicks();
        dt = end_time - start_time;
        SDL_Delay(time_per_frame - dt);
    }

    destroy_stuff();
}
```

In this method, if the system is too fast and completes the loop before some time, then we delay for the remaining time.
This fixes the problem on fast systems, but on slow systems, it still updates slower.
This problem when the computer executes the loop slower than expected is called as lag.

Method 2

```
int main() {
    initialize_stuff();

    start_time = SDL_GetTicks();
    while (true) {
        take_input();

        end_time = SDL_GetTicks();
        dt = end_time - start_time;
        update(dt);
        start_time = SDL_GetTicks();

        draw();
    }

    destroy_stuff();
}
```

Here we update the players rectangle by a factor of time instead of just updating it normally.
The update function would normally be like so:

```
void update(){
    rect.x += 5;
}
```

However, now we change it by multiplying the distance with time:

```
void update(int dt){
    rect.x += 5 * dt;
}
```

The basic idea is that,
for a slow system, we update lesser but by a larger distance.
for a fast system, we update more frequently but by a smaller distance.
This would ensure that over some time period, both the systems would update by the same distance.

However, a problem with this method is that, if for a very slow system, the final distance calculated is very big, then there's a chance that it can phase through walls.

Method 3

```
int main() {
    initialize_stuff();

    start_time = SDL_GetTicks();
    lag = 0;
    while (true) {
        take_input();

        end_time = SDL_GetTicks();
        dt = end_time - start_time;
        lag += dt;
        while (lag >= time_per_frame) {
            update();
            lag -= time_per_frame;
        }
        start_time = SDL_GetTicks();

        draw();
    }

    destroy_stuff();
}
```

This is the most complicated method, however it guarantees that on a slower system, if we lag a lot, we update as much as needed to cover up for how much ever we lagged.

This would fix the previous problem we had with collision physics. However, the problem with this method is that if we lag behind by a lot, we update as much and then draw only once. This would appear to the user as if the player just teleported. However, this is the best we can do with slow systems. Lag :)

Changing side-scroll back to top-down

We now remove the jump mechanics and transition back to the top-down view. We remove the jump and related variables, and add two boolean variables, (move_up and move_down).

```
bool move_left{};
bool move_up{};
bool move_down{};
bool move_right{};
```

We now change our handle_input and update function accordingly.

Handle_input function:

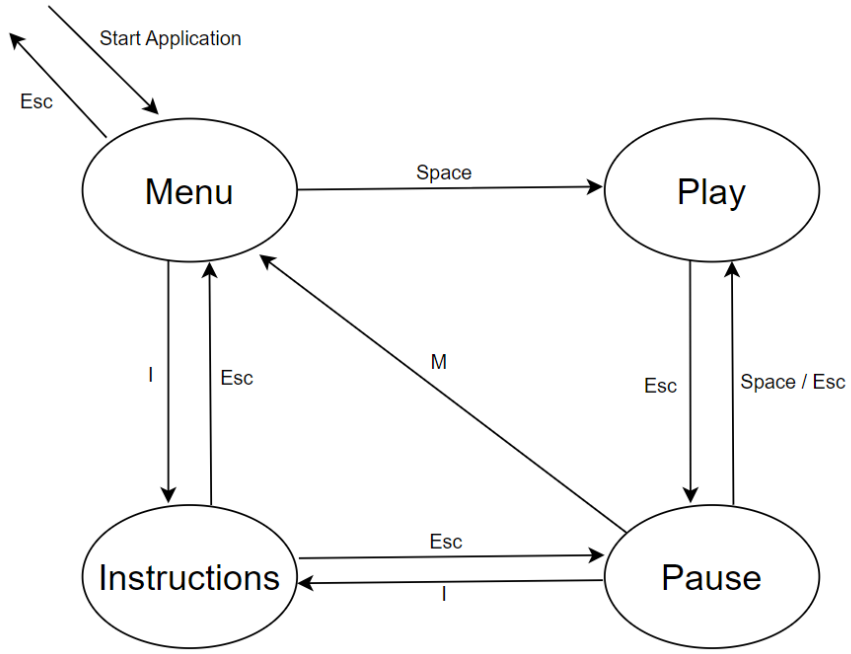
```
if (event.type == SDL_KEYDOWN) {
    if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
        move_right = true;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
        move_left = true;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
        move_up = true;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) {
        move_down = 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) {
        move_up = false;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) {
        move_down = false;
    }
}
```

The update function:

```
// movement
if (move_right) {
    rect.x += speed;
}
if (move_left) {
    rect.x -= speed;
}
if (move_up) {
    rect.y -= speed;
}
if (move_down) {
    rect.y += speed;
}
```

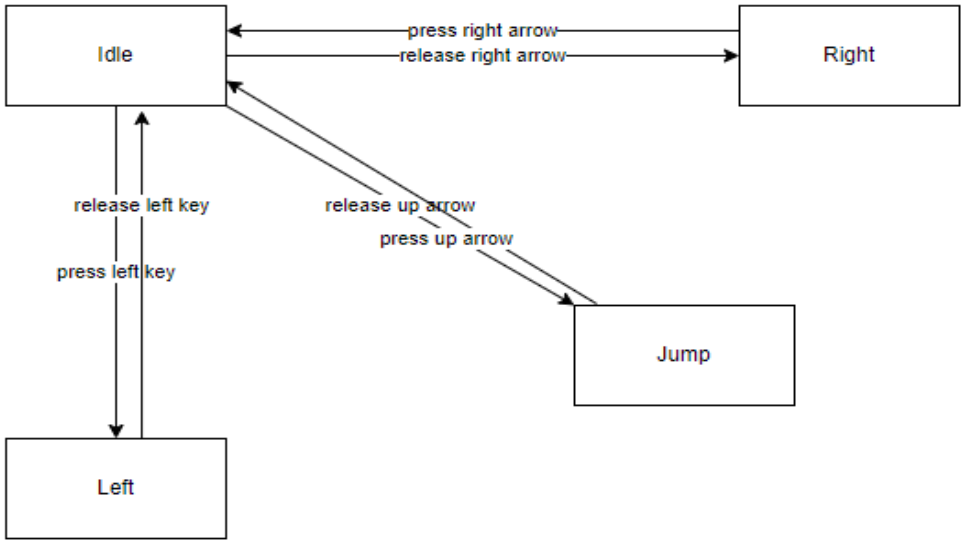
State Machines

Finite state machines are used to model complex behaviour in games. It's like a graph, where each node is a **state**, and each **edge** is a transition between different states.



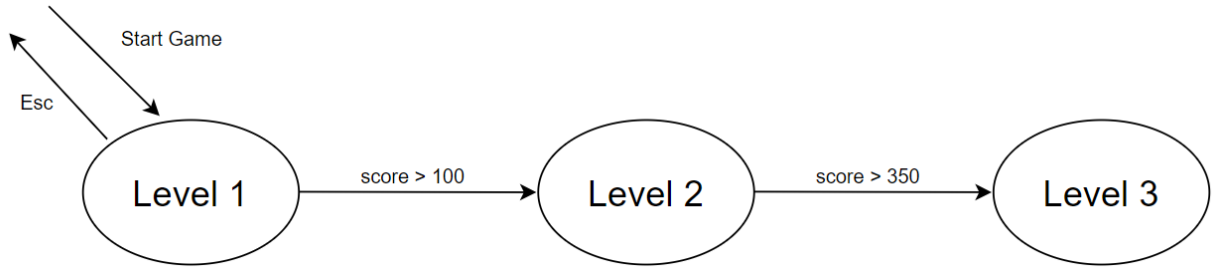
In the above example, we are referring to the change of the game's state. On entering the game, we start on the menu screen. We can transition to the playing phase, from where we can choose to pause and resume. We could also want to look at the instructions to play the game and want to exit the game too.

All these are modelled as game states represented in the circles. From each state you may take one or multiple actions by pressing a key and transitioning to a different state.



In this example, the player has 4 states: idle, move left, right and jump. At any point of time a player can be in only one of the states. To do this we use enums.

State Machines for Levels



An application of FSMs could be the implementation of levels in your game. When the score is greater than 100, move to level 2. When the score is greater than 350, move to level 3.

In this case we would initialise 3 map objects to depict the different levels and maintain state as an enum in the main file.

```
Map_Object *L1 = new Map_Object(25, 25, win_obj->Renderer);
Map_Object *L2 = new Map_Object(25, 25, win_obj->Renderer);
Map_Object *L3 = new Map_Object(25, 25, win_obj->Renderer);

enum States{
    Level_1, Level_2, Level_3
}State = Level_1;
```

Initially we would start at L1 and maintain a score.
When the score increases, we change the state.

```
if (score > 100) {
    State = Level_2;
}
if (score > 350) {
    State = Level_3;
}
```

Any function which used the map in the previous code will now change based on which level we are in. Using a switch case on State we use the respective level's map function to perform the correct operation.

Previous code:

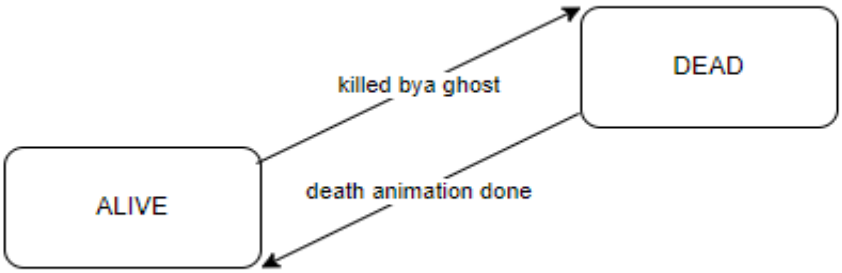
```
// drawing
win_obj->clear_background();  // clears background
map->draw();                  // draws map
player->draw();               // draws player
win_obj->swap_buffer();       // swaps buffers
```

Now:

```
// drawing
win_obj->clear_background();  // clears background
// draws map
switch (State){
    case Level_1:
        L1->draw();
        break;
    case Level_2:
        L2->draw();
        break;
    case Level_3:
        L3->draw();
        break;
}
player->draw();               // draws player
win_obj->swap_buffer();       // swaps buffers
```

State Machines for Player

For the case of Pacman, it might be in any of the two states- ALIVE or DEAD.
In the code we have written so far, we are assuming that Pacman is always alive.
So the current code in the input_handle and update functions are now put under the ALIVE case of the switch statement. When Pacman dies, a different animation plays which we will see in the upcoming lesson.



So let's create 2 states for Pacman in the Player_Object.h file: (The default state is ALIVE)

```
enum States{
    ALIVE, DEAD
}State = ALIVE;
```

```
switch(State){
    case ALIVE:
        // code to execute if State = Alive
        break;
    case DEAD:
        // code to execute if State = Dead
        break;
}
```

Now we modify the update function to do different actions in case of current state being ALIVE or DEAD.

```
void Player_Object::update(Map_Object *map) {
    // code for movement
    switch (State) {
        case ALIVE:
            // move
            if (move_right) {...}
            if (move_left) {...}
            if (move_up) {...}
            if (move_down) {...}
            break;
        case DEAD:
            // do nothing
            break;
    }
}
```

Getting Pellets out of the tile-map

Since we made our tile-map in such a way that if the element is 2 if it is a pellet, we can use this. But our Pellets are separate objects from our Map object and hence for good coding & design purposes, the Pellets vector cannot be a part of our Map object.
Hence, we create a function to get a vector of all the initial pellets.

In our Map_Object.h, we create:

```
std::vector<SDL_Rect> get_pellets();
```

In our Map_Object.cpp, we create:

```cpp
std::vector<SDL_Rect> Map_Object::get_pellets() {
    std::vector<SDL_Rect> pellets;
    int size = 5;
    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < cols; col++) {
            if (tile_map[row][col] == 2) {
                SDL_Rect rect = {
                        col * cell_width + cell_width / 2 - size / 2,
                        row * cell_height + cell_height / 2 - size / 2,
                        size, size
                };
                pellets.push_back(rect);
            }
        }
    }
    return pellets;
}
```

This concludes Day-9 :)