SDL2 Course Day-10
Pellets, Animation & TTF

Pellets Class:

We create a new class to store a vector of pellets. An object would take the vector provided my the map->get_pellets() function and store it as a member:

```
class Pellets_Object {
public:
    std::vector<SDL_Rect> pellets;
    SDL_Renderer *Renderer;

    Pellets_Object(std::vector<SDL_Rect> v, SDL_Renderer *Renderer);
```

```
Pellets_Object::Pellets_Object(std::vector<SDL_Rect> v, SDL_Renderer *Renderer)
        : Renderer(Renderer), pellets(v) {}
```

Including "Pellets_Object.h" in the main file:

```
#include "Pellets_Object.h"
```

We create pellets in the main file:

```
Pellets_Object *pellets = new Pellets_Object(map->get_pellets(), win_obj->Renderer);
```

To draw we create the draw function:

```
void Pellets_Object::draw() {
    SDL_SetRenderDrawColor(Renderer, 248, 176, 144, 255);
    for (auto rect: pellets) {
        SDL_RenderFillRect(Renderer, &rect);
    }
}
```

This is similar to drawing the rectangles of the walls in map:
We first set the color of the Renderer and then go through each rectangle in the pellets vector and fill that rectangle with the color.

Calling draw in the main function:

```
pellets->draw();
```

Deleting Pellets:

When the player collides with any pellet, we need to delete the pellet and increase the score.
"Deleting" a rectangle can happen in many ways.
If the rectangle is created using dynamic memory, we can call the delete function on it.
We can also change the opacity of the rectangle to 0 (rgb**a**).
We could also create a black rectangle or a rectangle as the same color as the background and overlap the pellet with this.
However, one of the easiest things to do, is to just change the x and y position of the pellet to -ve co-ordinates (off-screen). This is what we will follow.

We will check for collision in the update function of the pellets object. If we detect a collision, we change the pellet's co-ordinates to make it go off screen. Then we update score.

```
void Pellets_Object::update(SDL_Rect rect, int &score) {
    for (auto &pellet: pellets) {
        if (SDL_HasIntersection(&rect, &pellet)) {
            score += 10;
            pellet.x = -50;
            pellet.y = -50;
        }
    }
}
```

The score is passed as a parameter by reference.
In the main function we create a score variable outside the while loop:

```
int score = 0;
```

Then we call the function like so in the update part of the code:

```
// update
player->update(map);
map->handle_collision(player->rect, player->old_x, player->old_y);
pellets->update(player->rect, score);
```

Keeping count of pellets:

What happens if the player eats all the pellets?
The game is supposed to restart or go to the next level.

So we need to be able to know if our pellets are all eaten. But since the size of vector in the pellets object never changes, how would we get to know if all our pellets are eaten?

We do this by adding a count variable. Every time we delete a pellet, we increase the count by 1.

```
void Pellets_Object::update(SDL_Rect rect, int &score) {
    for (auto &pellet: pellets) {
        if (SDL_HasIntersection(&rect, &pellet)) {
            score += 10;
            pellet.x = -50;
            pellet.y = -50;
            count++;
        }
    }
}
```

We initialize count to 0 in the .h file:

```
int count = 0;
```

Now, we create an is_empty() function, this function will return true when all pellets are outside the screen and false otherwise:

```
bool Pellets_Object::is_empty() {
    return count == pellets.size();
}
```

Storing Player's Spritesheet:

We first create textures to store the pictures / spritesheets in the player_object:

```
SDL_Texture *Alive_Texture;
SDL_Texture *Dead_Texture;
```

In the constructor: (same method as creating a texture like we did in the map_object):

```
SDL_Surface *dummy = IMG_Load("pacman_alive.png");
Alive_Texture = SDL_CreateTextureFromSurface(Renderer, dummy);
SDL_FreeSurface(dummy);

dummy = IMG_Load("pacman_dead.png");
Dead_Texture = SDL_CreateTextureFromSurface(Renderer, dummy);
SDL_FreeSurface(dummy);
```

In the destructor:

```
Player_Object::~Player_Object() {
    SDL_DestroyTexture(Alive_Texture);
    SDL_DestroyTexture(Dead_Texture);
}
```

Animation:

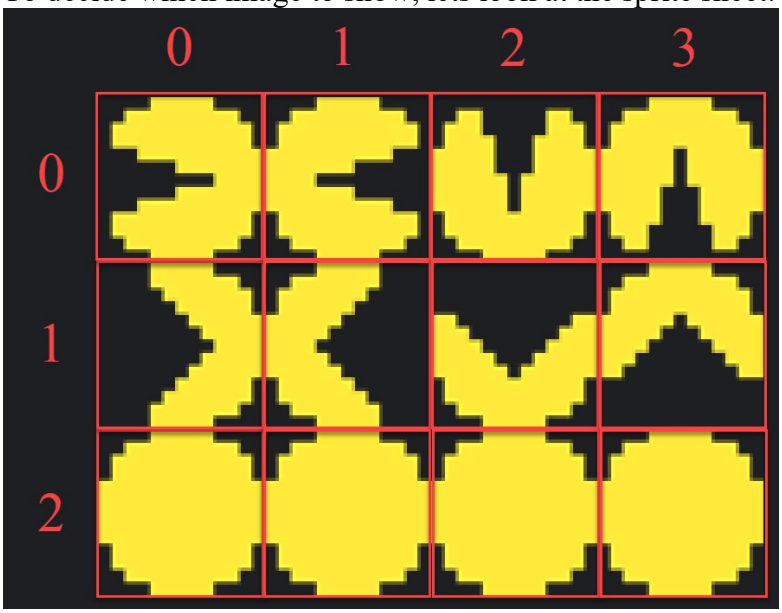
Animation is basically a sequence of images, each image being displayed after a certain time. To control how fast the animation repeats, we create the following structure:

```
struct frame {
    int current;
    int limit;
} Animation_Frame = {0, 7};
```

Each time the update function is called, we increment the current_frame value by 1.
When the value of current equals limit, we reset current and then change the image to the next image.

```
Animation_Frame.current++;
if (Animation_Frame.current == Animation_Frame.limit) {
    Animation_Frame.current = 0;
    // update to next image
}
```

To decide which image to show, lets look at the sprite sheet:



As we can see, the spritesheet can be split in a 3 x 4 grid.

Lets say we're currently displaying cell 0,0.
The next image in this sequence is 1,0.
The next image after that is 2,0.
And finally it goes back 0,0.

If the player is idle, the cell can be anywhere in the last row (row = 2).
If the player is moving left, its column is 0.
If its moving right, the column is 1.
If its moving up, the column is 2.
If its moving down, the column is 3.

To summarize:

To advance the image to the next image in the sequence, we just need to increase the row by 1. (row needs to be reset if it goes greater than 2).
To change direction, we change the column value.

```
Animation_Frame.current++;
if (Animation_Frame.current == Animation_Frame.limit) {
    Animation_Frame.current = 0;
    Cur_Frame.row++;
    if (Cur_Frame.row == 3)
        Cur_Frame.row = 0;
}

if (move_left) Cur_Frame.col = 0;
if (move_right) Cur_Frame.col = 1;
if (move_up) Cur_Frame.col = 2;
if (move_down) Cur_Frame.col = 3;
```

To check if the player is idle, we check if all the move variables are false, then we change row to 2, and make the current frame number, just below the limit. (This is done for smoothness).

```
if (!move_left && !move_right && !move_up && !move_down) {
    Cur_Frame.row = 2;
    Animation_Frame.current = Animation_Frame.limit - 1;
} else {
```

Restarting Level:
To restart, we just check if all pellets are eaten. If they are, we just reset the pellets:

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

if (pellets->is_empty()) {
    pellets->pellets = map->get_pellets();
}
```

If we wanted to quit, we would change the win_obj->quit to true. This would exit the while loop.

Displaying Text on Screen:
We use the TTF library to display text to the screen through SDL.
TTF stands for "Text To Font". For this, we include the ttf library:

```
#include <SDL2/SDL_ttf.h>
```

To initialize:

```
TTF_Init();
```

To Quit:

```
TTF_Quit();
```

We have all seen different fonts in MSWord or Google Docs. The fonts are stored in a .ttf file. These fonts can be found on Windows system in the following directory.

C:\Windows\Font

Go here and take any font of your choice, we take arial.ttf; copy the file and paste it in the same directory.
To Open a font:

```
TTF_Font *font = TTF_OpenFont("arial.ttf", 25);
```

The number decides the size of the final text.

To Close a font:

```
TTF_CloseFont(font);
```

This function should be called before quitting the application.

Now we have the font, we can choose what color it should be. For this SDL provides a structure called SDL_Color.

```
SDL_Color fontColor = {255, 255, 255, 255};
```

Next, we create a **surface** and use the renderer to create solid text on the **surface**.

```
SDL_Surface *text_on_surface = TTF_RenderText_Solid(font, "Testing!!!", fontColor);
```

Then, we convert the surface into a texture using the renderer:

```
SDL_Texture *texture = SDL_CreateTextureFromSurface(Renderer, text_on_surface);
```

Now we need to display this texture on the screen.
Usually when we display to part of the screen, we need to specify a dstRect rectangle which tells the renderer what part of the screen we want to draw to.
The problem is we don't know what is the width and height of the texture we just created.

So to get this we do the following:
Create two variables for storing width and height.
Pass the address of the two variables to a query function which will modify the variables and give the required width and height.

Use this width and height to create a dstRect and proceed normally.

The following is the code:

```
// creating two variables to store width and height
int texW = 0, texH = 0;
SDL_QueryTexture(texture, nullptr, nullptr, &texW, &texH);

// creating dstRect and displaying normally
SDL_Rect dstRect = {100, 100, texW, texH};
SDL_RenderCopy(Renderer, texture, nullptr, &dstRect);
```

Finally we destroy whatever we created:

```
SDL_DestroyTexture(texture);
SDL_FreeSurface(text_on_surface);
```

Now if run the code, we will see the text on the screen.

We, now wrap the above code in a function so that we can call a single function to display any message we want:

Note that since the SDL2 library was written in C, the TTF functions expect to receive a variable of type (char *) which is what C uses to store text.
In C++, we have strings.

To convert strings to (char *) we use the method:

```
msg.c_str()
```

Also when we need to display a number, we would need to convert it to a string:

```
int num = 5;
std::string message = std::to_string(num);
```

We now have the final function:

```
void draw_text(std::string msg, int x, int y, TTF_Font *font, SDL_Color fontColor, SDL_Renderer *Renderer) {
    SDL_Surface *text_on_surface = TTF_RenderText_Solid(font, msg.c_str(), fontColor);
    SDL_Texture *texture = SDL_CreateTextureFromSurface(Renderer, text_on_surface);

    int texW = 0, texH = 0;
    SDL_QueryTexture(texture, nullptr, nullptr, &texW, &texH);
    SDL_Rect dstRect = {100, 100, texW, texH};
    SDL_RenderCopy(Renderer, texture, nullptr, &dstRect);

    SDL_DestroyTexture(texture);
    SDL_FreeSurface(text_on_surface);
}
```

The function is called as so:

```
draw_text(std::to_string(30), 0, 0, font, fontColor, win_obj->Renderer);
```

Fin.