# Mini Project 2

CSI2P(II) Chen

# Contents

# Contents

# Tower Games

# Project Goal

- Complete and implement functions of “Tower Game” by using OOP concept and C++ programming skills learned from class
- Get more familiar with Allegro, and develop your own final project
- Mini project II will teach you how to use concept of class to construct a strategy game

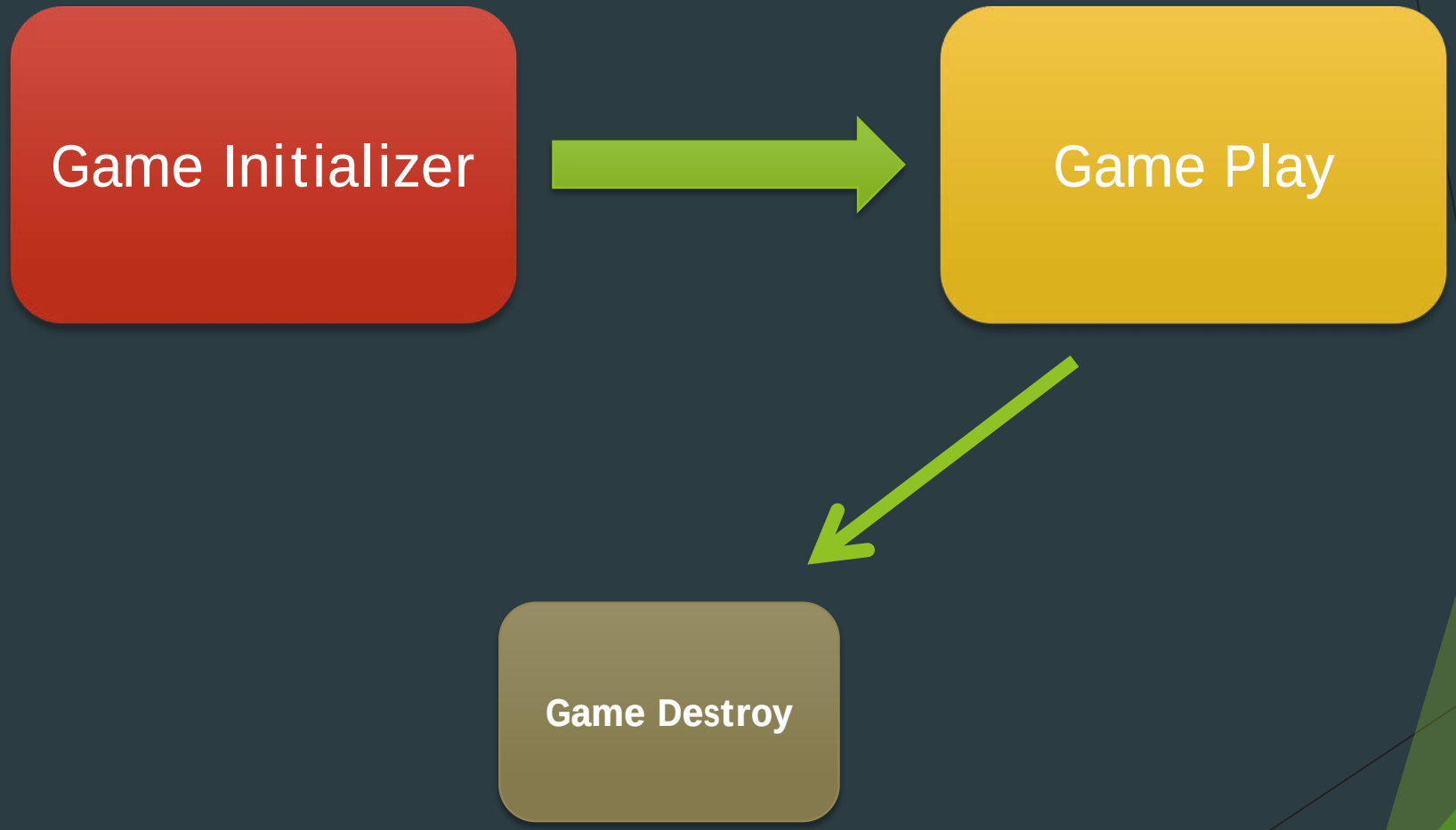
# Contents

# Tower Game Tutorial

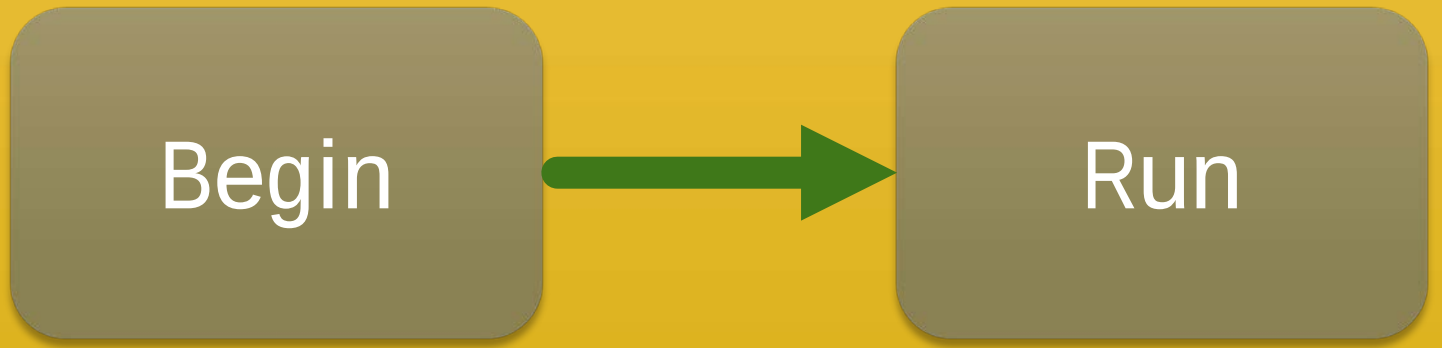
- Flow chart of Tower Game
- Classes and methods introduction

# Tower Game Tutorial

- Flow chart of Tower Game
- Classes and methods introduction

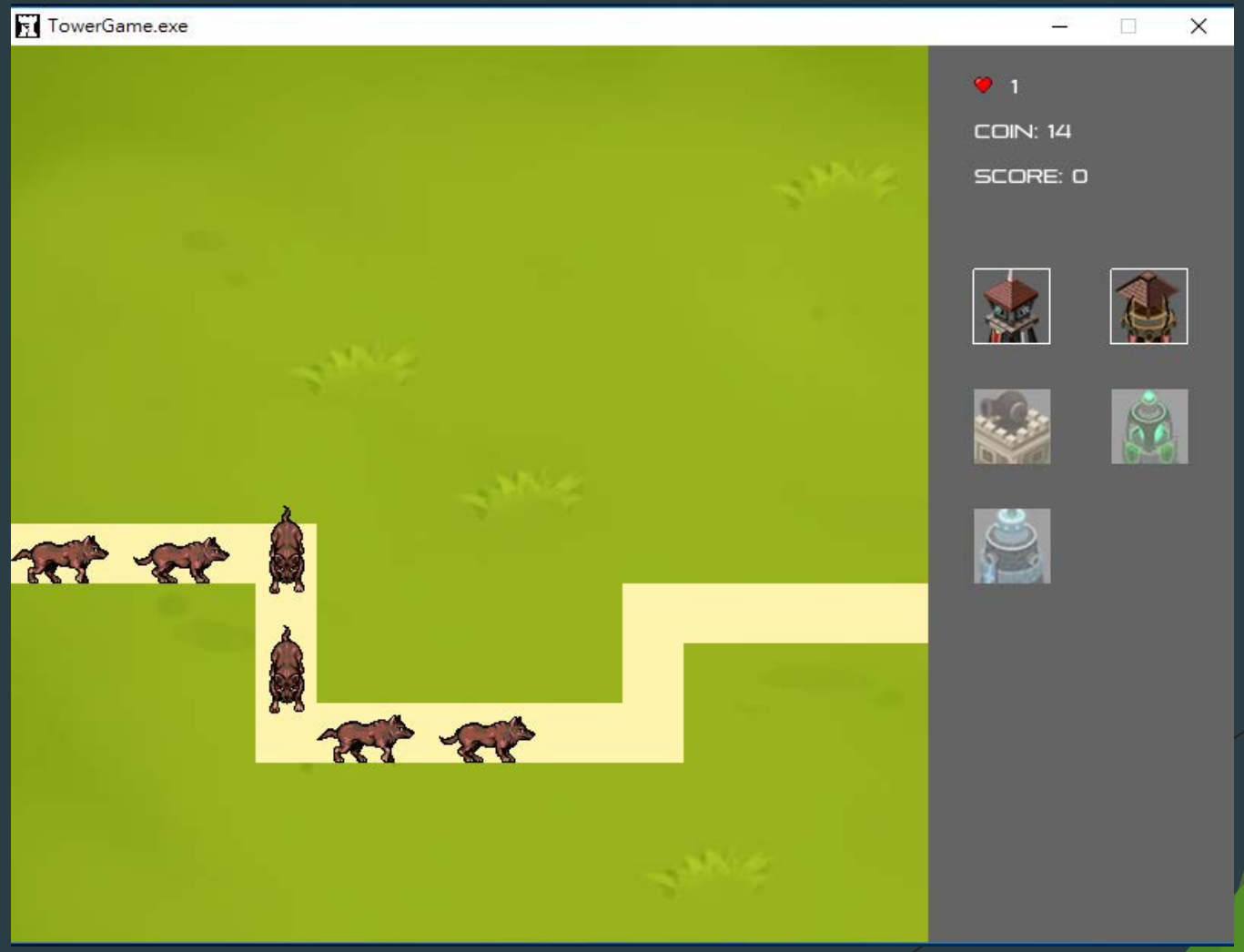

Game Initializer
Game Play
Game Destroy

# Game Play



# RUNNING

## Void GameWindow::game_run()



# Tower Game Tutorial

- Flow chart of Tower Game
- Classes and methods introduction

# GameWindow.h

- Define entire structure of tower game
- A series of functions control the procedure of the game, ex: game_play(), game_destroy()…etc.
- You can first trace game_play() to know how the process of this game is and add your code.

## Game Functions

```
class GameWindow
{
public:
    // constructor
    GameWindow();
    // each process of scene
    void game_init();
    void game_reset();
    void game_play();
    void game_begin();

    int game_run();
    int game_update();

    void show_err_msg(int msg);
    void game_destroy();

    // each drawing scene function
    void draw_running_map();

    // process of updated event
    int process_event();
```

## Variables

```
LEVEL *level = NULL;
Menu *menu = NULL;

std::vector<Monster*> monsterSet;
std::list<Tower*> towerSet;

int Monster_Pro_Count = 0;
int Coin_Inc_Count = 0;
int mouse_x, mouse_y;
int selectedTower = -1, lastClicked = -1;

bool redraw = false;
bool mute = false;
```

# void game_init()

- **Description**:

  Initialize all needed elements (bitmaps, sounds, ……)

  Besides, after sound samples are built, you also need to attach them to mixer(current use default mixer).

# int game_update()

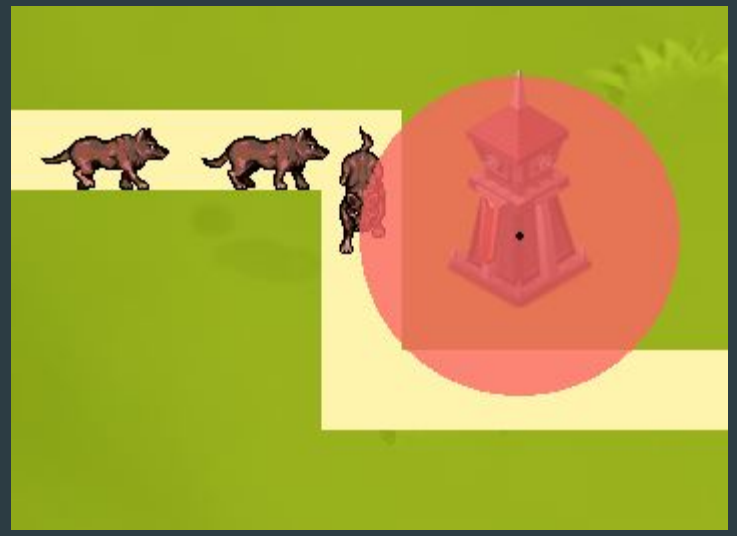
- **Description**:
  - The most important core for whole game routine
  - Return the next game status
  - Control the actions of each objects in game, including tower attack trigger, monster moving action, updating attack set…etc.

# global.h

- Define some global variable for some function to use.
- Don’t change the value of any variable in other files if you don’t understand whether the changes will cause any error or not.

# Circle.h

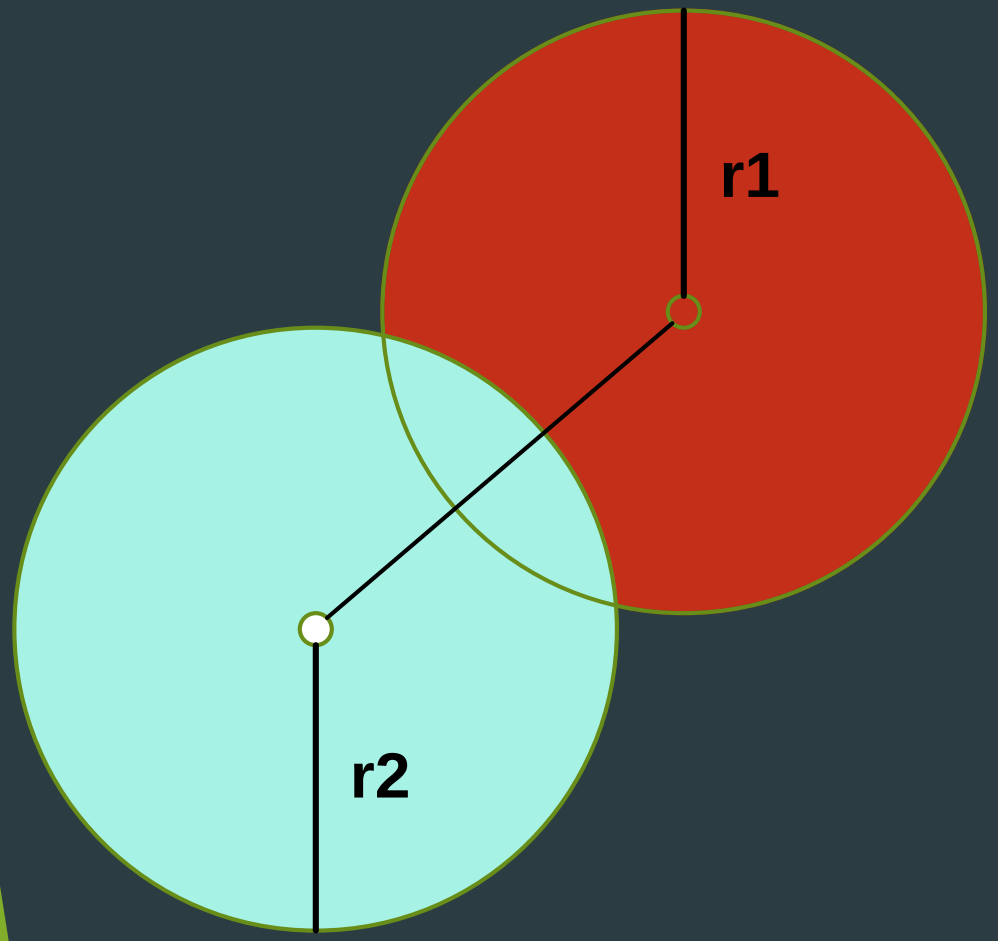
- Define a class named “Circle”
- This class is used to set the valid range of tower, monster and attack.



## Class View

```
class Circle{
public:
    Circle() {}

    Circle(int x, int y, int r)
    {
        this->x = x;
        this->y = y;
        this->r = r;
    }

    static bool isOverlap(Circle*, Circle*);

    int r;
    int x, y;
};
```

# static bool isOverlap (Circle*, Circle*)




- **Description**:
  - Check if is the distance between centers of both circles less than r1 + r2 or not.

# Level.h

- Define LEVEL class to store the information of a level.
- One container to record the path that monster follows. The path arguments all are stored in a text file
  - you can find in main folder
- You can also see that it already define the speed of producing a monster and the maximum number of monsters in a level.

# void LEVEL::setLevel(const int level)

- **Description**:
  - set up level content read from .txt in game folder
  - including total number of monsters, the path for monsters to follow

# Menu.h

- Define a class to record and show information of current level

- There are two important methods:
  - **int MouseIn**
  - **static bool isInRange**

# int MouseIn(int, int)

- Description:
  - Detect if mouse hovers over any tower image on menu.
  - Return the image index if coin is enough to pay
    - otherwise return -1

# static bool Menu::isInRange (int point, int startPos, int length)

- Description:
  - Input
    - point: the testing point
    - startPos: the start position of a line
    - length: the length of the line
  - If the testing point is on the line or not
  - For MouseIn, it can simplify the statement

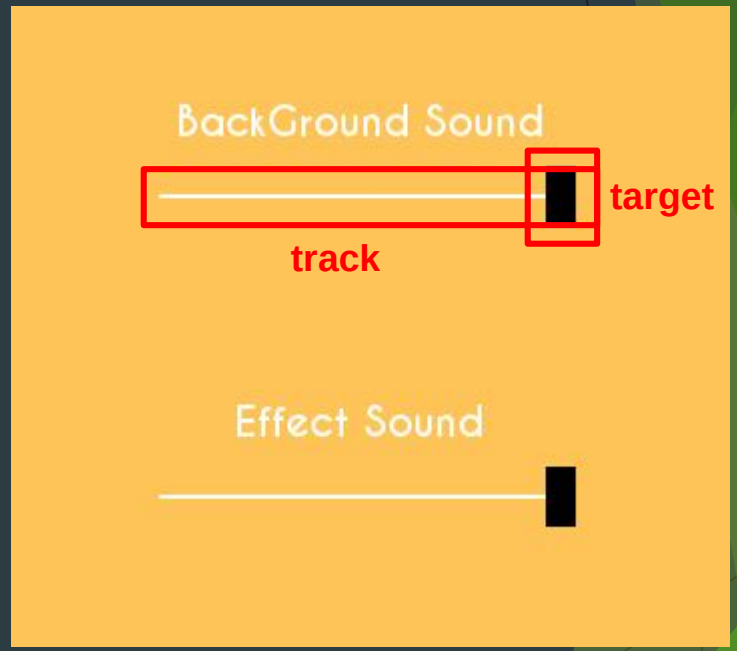
```
public:
    Menu();
    ~Menu();

    void Reset();
    void Draw();

    // Detect if cursor hovers over any
    // If so, return its type
    // Otherwise, return -1
    int MouseIn(int, int);

    // static function that detect if or
    // This function is just used to sim
    static bool isInRange(int, int, int)

    // Check if current coin is not less than needed coin
    bool Enough_Coin(int);
    void Change_Coin(int change) { Coin += change; }
    bool Subtract_HP(int escapeNum = 1);
    void Gain_Score(int);

    int getTowerCoin(int type) { return need_coin[type]; }
    int getScore() { return Score; }
    int getCoin() { return Coin; }
    int getKilled() { return killedMonster; }
```

```
int
Menu::MouseIn(int mouse_x, int mouse_y)
{
    bool enoughCoin;

    selectedTower = -1;

    for(int i=0; i < Num_TowerType; i++)
    {
        int pos_x = offsetX + (ThumbWidth + gapX) * (i % 2);
        int pos_y = offsetY + (ThumbHeight + gapY) * (i / 2);

        if(isInRange(mouse_x, pos_x, ThumbWidth) && isInRange(mouse_y, pos_y, ThumbHeight))
        {
            if(Enough_Coin(i))
            {
                selectedTower = i;
                break;
            }
        }
    }
    return selectedTower;
}
```

# Slider.h

- Define the functional class of slider.
- In this game, it is used to create two sliders to adjust the volume of background and effect sound.
- For mini-project2, you should use it to control volume.

```
class Slider : public Object
{
public:
    Slider(int, int);
    ~Slider();

    void Draw();
    void set_color(ALLEGRO_COLOR);
    void set_label_content(const char*);
    void toggleDrag() { dragged = !dragged; }

    float Drag(int, int);
    float getDegree() { return degree; }

    int getLength() { return lengthOftrack; }

    bool isClicked(int, int);
    bool isDragged() { return dragged; }

private:
    int target_x, target_y;
    int track_x, track_y;
    int lengthOftrack = 200;
    float degree = 1.0;
    bool dragged = false;
    char label[20];
    ALLEGRO_COLOR target_color;
    ALLEGRO_FONT *font;
};
```



# Monster.h

- Define class for monster and this will be inherited by four subclasses.

- Move(): determine animation image and direction of each step on map.
- Load_Move(): according to class name of a monster, it will load its animation images of four directions.




```
class Monster: public Object {
public:
    Monster(std::vector<int> path);
    ~Monster();

    // Draw image per frame
    // override virtual function "Obj
    void Draw();
    // Load bitmaps of animation ima
    void Load_Move();

    // Update monster position per frame
    // And detect if it reaches end point but not destroyed
    bool Move();

    // functions that return informations of monster
    int getDir() { return direction; }
    int getWorth() { return worth; }
    int getScore() { return score; }

    bool Subtract_HP(int);
```

```
void
Monster::Load_Move()
{
    char buffer[50];

    for(int i=0; i < 4; i++)
    {
        for(int j=0; j<direction_count[i]; j++)
        {
            ALLEGRO_BITMAP *img;
            sprintf(buffer, "./%s/%s_%d.png", class_name, direction_name[i], j);

            img = al_load_bitmap(buffer);
            if(img)
                moveImg.push_back(img);
        }
    }
}
```

# Tower.h

- Define the class for tower and this will be inherited by other five subclasses.

- UpdateAttack(): update attack set of the tower.
- DetectAttack(): make sure if the tower needs to trigger attack
- TriggerAttack(): Go through whole attack set to test if a monster will be destroyed by the attack of tower.

## Each Time Frame

# Attack.h

- Define the class for describing the form of an attack.
- You can see that attack only follows original straight line from the tower's pivot toward the position where monster collided with the tower.