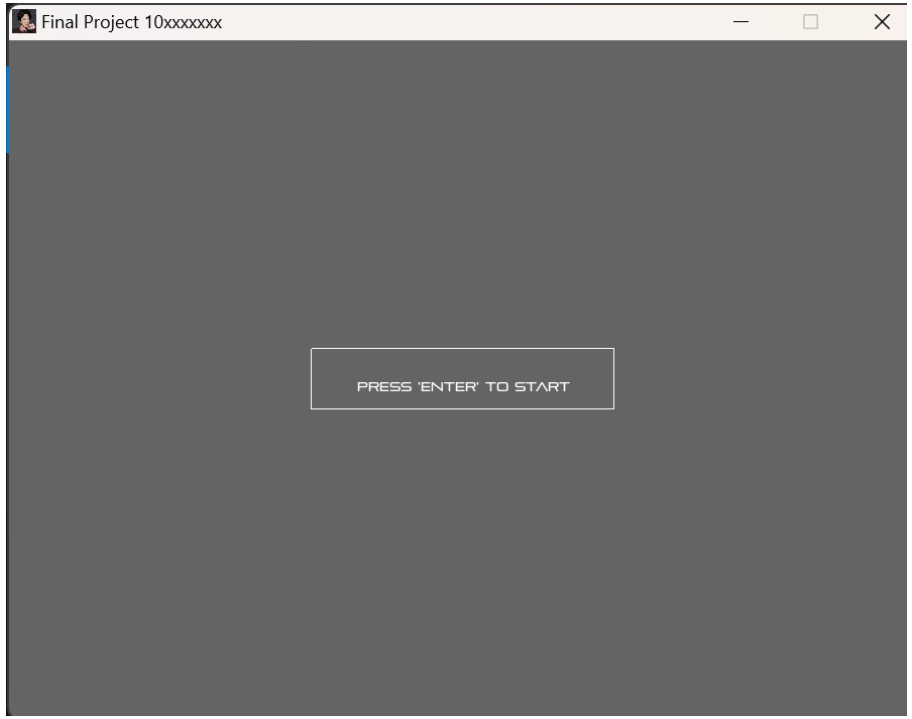
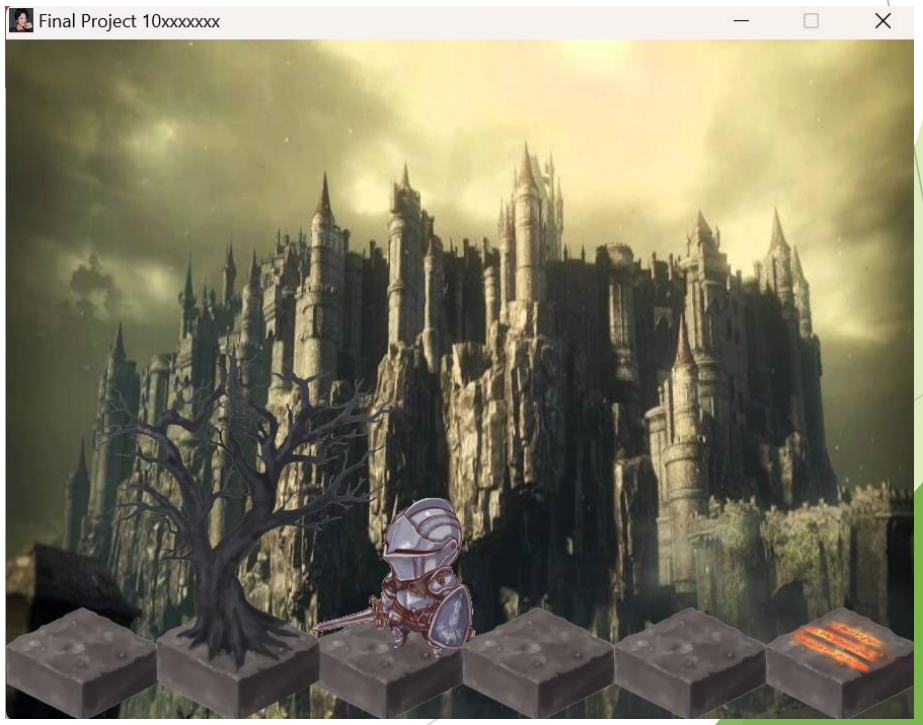
Template Trace

Overview

Design Game

- Want only control the scene
  - Control scene regardless the content





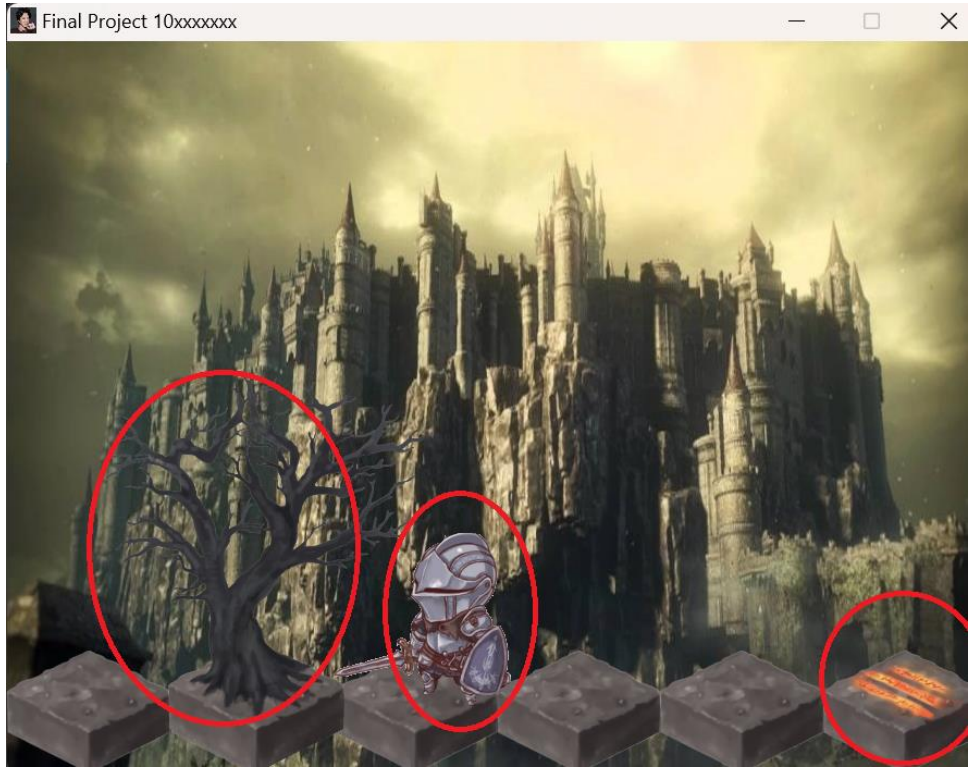
Design Game

- If we want to draw the scene just call the scene->Draw regardless whether it's menu or game scene

```
void game_draw(Game *game)
{
    // Flush the screen first.
    al_clear_to_color(al_map_rgb(100, 100, 100));
    scene->Draw(scene);
    al_flip_display();
}
```

Design Scene

- Want only control the element
  - Control element regardless the content



Design Scene

- If we want to draw the element just call the ele>Draw regardless which element it is.

```
void game_scene_draw(Scene *const pGameSceneObj)
{
    al_clear_to_color(al_map_rgb(0, 0, 0));
    GameScene *gs = ((GameScene *)(pGameSceneObj->pDerivedObj));
    al_draw_bitmap(gs->background, 0, 0, 0);
    ElementVec allEle = _Get_all_elements(pGameSceneObj);
    for (int i = 0; i < allEle.len; i++)
    {
        Elements *ele = allEle.arr[i];
        ele->Draw(ele);
    }
}
```

Problem need to solve

- We need struct share the same type but with different member
- We need function share the same definition but with different functionality

Problem need to solve

- We need struct share the same type but with different member
  - Use void pointer point to the sub-struct

```
struct _Scene
{
    int label;
    void *pDerivedObj;
    bool scene_end;
    int ele_num;
    EPNode *ele_list[MAX_ELEMENT];
    // interface for function
    fptrUpdate Update;
    fptrDraw Draw;
    fptrDestroy Destroy;
};
```

```
Scene *New_GameScene(int label)
{
    GameScene *pDerivedObj = (GameScene *)malloc(sizeof(GameScene));
    Scene *pObj = New_Scene(label);
    // setting derived object member
    pDerivedObj->background = al_load_bitmap("assets/image/stage.jpg");
    pObj->pDerivedObj = pDerivedObj;
    // register element
    _Register_elements(pObj, New_Floor(Floor_L));
    _Register_elements(pObj, New_Teleport(Teleport_L));
    _Register_elements(pObj, New_Tree(Tree_L));
    _Register_elements(pObj, New_Character(Character_L));
    // _Register_elements(pObj, New_Ball(Ball_L));
    // setting derived object function
    pObj->Update = game_scene_update;
    pObj->Draw = game_scene_draw;
    pObj->Destroy = game_scene_destroy;
    return pObj;
}
```

Problem need to solve

- We need function share the same definition but with different functionality
  - Use function pointer point to different function

```
struct _Scene
{
    int label;
    void *pDerivedObj;
    bool scene_end;
    int ele_num;
    EPNode *ele_list[MAX_ELEMENT];
    // interface for function
    fptrUpdate Update;
    fptrDraw Draw;
    fptrDestroy Destroy;
};
```

```
Scene *New_GameScene(int label)
{
    GameScene *pDerivedObj = (GameScene *)malloc(sizeof(GameScene));
    Scene *pObj = New_Scene(label);
    // setting derived object member
    pDerivedObj->background = al_load_bitmap("assets/image/stage.jpg");
    pObj->pDerivedObj = pDerivedObj;
    // register element
    _Register_elements(pObj, New_Floor(Floor_L));
    _Register_elements(pObj, New_Teleport(Teleport_L));
    _Register_elements(pObj, New_Tree(Tree_L));
    _Register_elements(pObj, New_Character(Character_L));
    // _Register_elements(pObj, New_Ball(Ball_L));
    // setting derived object function
    pObj->Update = game_scene_update;
    pObj->Draw = game_scene_draw;
    pObj->Destroy = game_scene_destroy;
    return pObj;
}
```

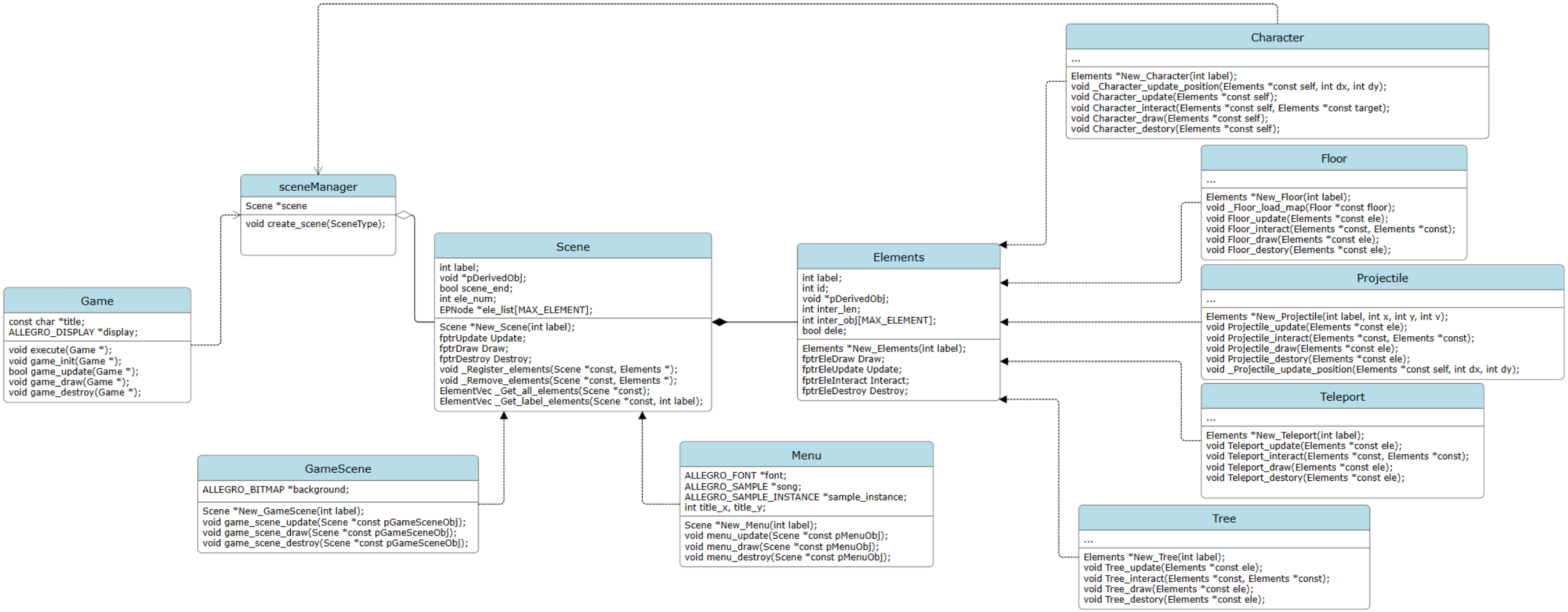
Void pointer and Function pointer

- Void pointer
  - Need to change the type before use it

```
void game_scene_draw(Scene *const pGameSceneObj)
{
    GameScene *gs = ((GameScene *)(pGameSceneObj->pDerivedObj));
```

- Function pointer

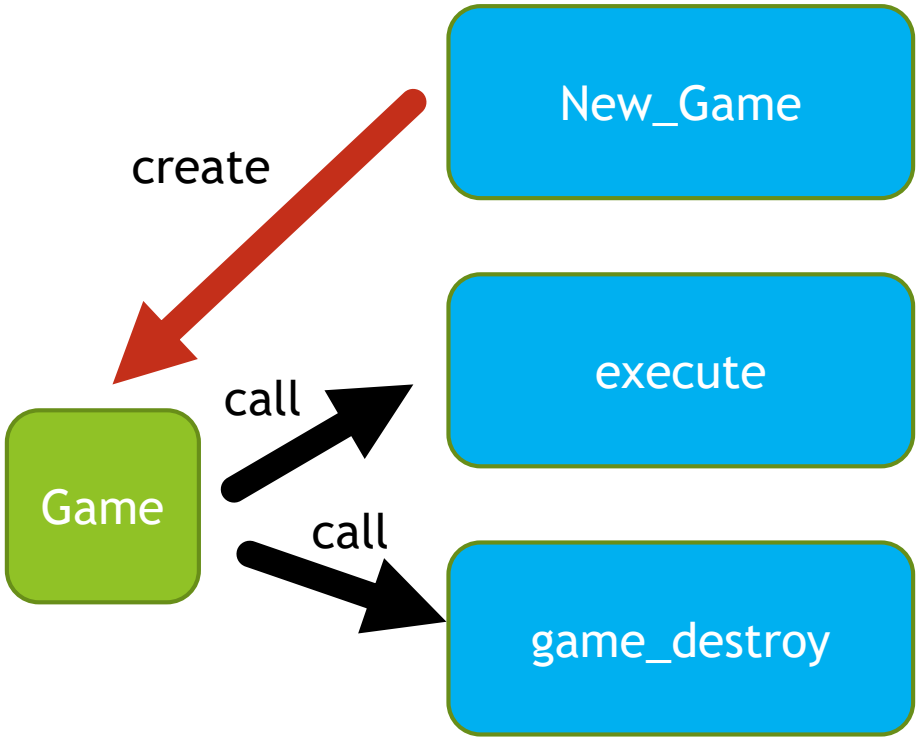
```
typedef void (*fptrUpdate)(Scene *const);
typedef void (*fptrDraw)(Scene *const);
typedef void (*fptrDestroy)(Scene *const);
```

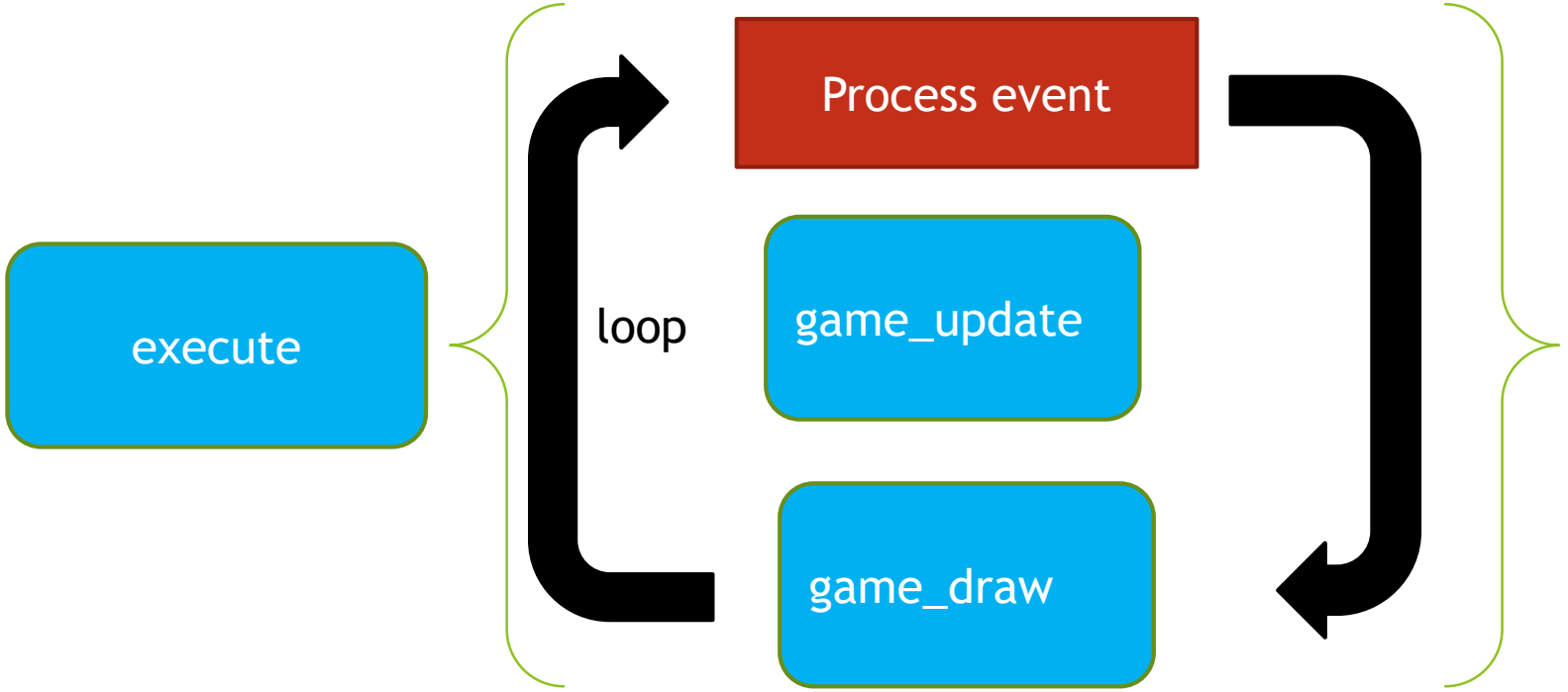

Character
...
Elements *New_Character(int label);
void _Character_update_position(Elements *const self, int dx, int dy);
void Character_update(Elements *const self);
void Character_interact(Elements *const self, Elements *const target);
void Character_draw(Elements *const self);
void Character_destory(Elements *const self);
Floor
...
Elements *New_Floor(int label);
void _Floor_load_map(Floor *const floor);
void Floor_update(Elements *const ele);
void Floor_interact(Elements *const, Elements *const);
void Floor_draw(Elements *const ele);
void Floor_destory(Elements *const ele);
sceneManager
Scene *scene
void create_scene(SceneType);
Scene
int label;
void *pDerivedObj;
bool scene_end;
int ele_num;
EPNode *ele_list[MAX_ELEMENT];
Scene *New_Scene(int label);
fptrUpdate Update;
fptrDraw Draw;
fptrDestroy Destroy;
void _Register_elements(Scene *const, Elements *);
void _Remove_elements(Scene *const, Elements *);
ElementVec _Get_all_elements(Scene *const);
ElementVec _Get_label_elements(Scene *const, int label);
Elements
int label;
int id;
void *pDerivedObj;
int inter_len;
int inter_obj[MAX_ELEMENT];
bool dele;
Elements *New_Elements(int label);
fptrEleDraw Draw;
fptrEleUpdate Update;
fptrEleInteract Interact;
fptrEleDestroy Destroy;
Projectile
...
Elements *New_Projectile(int label, int x, int y, int v);
void Projectile_update(Elements *const ele);
void Projectile_interact(Elements *const, Elements *const);
void Projectile_draw(Elements *const ele);
void Projectile_destory(Elements *const ele);
void _Projectile_update_position(Elements *const self, int dx, int dy);
Game
const char *title;
ALLEGRO_DISPLAY *display;
void execute(Game *);
void game_init(Game *);
bool game_update(Game *);
void game_draw(Game *);
void game_destroy(Game *);
Teleport
...
Elements *New_Teleport(int label);
void Teleport_update(Elements *const ele);
void Teleport_interact(Elements *const, Elements *const);
void Teleport_draw(Elements *const ele);
void Teleport_destory(Elements *const ele);
GameScene
ALLEGRO_BITMAP *background;
Scene *New_GameScene(int label);
void game_scene_update(Scene *const pGameSceneObj);
void game_scene_draw(Scene *const pGameSceneObj);
void game_scene_destroy(Scene *const pGameSceneObj);
Menu
ALLEGRO_FONT *font;
ALLEGRO_SAMPLE *song;
ALLEGRO_SAMPLE_INSTANCE *sample_instance;
int title_x, title_y;
Scene *New_Menu(int label);
void menu_update(Scene *const pMenuObj);
void menu_draw(Scene *const pMenuObj);
void menu_destroy(Scene *const pMenuObj);
Tree
...
Elements *New_Tree(int label);
void Tree_update(Elements *const ele);
void Tree_interact(Elements *const, Elements *const);
void Tree_draw(Elements *const ele);
void Tree_destory(Elements *const ele);

Game

Game

```
struct _GAME
{
    const char *title;
    // ALLEGRO Variables
    ALLEGRO_DISPLAY *display;
    fptrGameExecute execute;
    fptrGameInit game_init;
    fptrGameUpdate game_update;
    fptrGameDraw game_draw;
    fptrGameDestroy game_destroy;
};
Game *New_Game();
```


New_Game
create
execute
call
Game
call
game_destroy


execute
loop
Process event
game_update
game_draw

game_update

```
bool game_update(Game *game)
{
    scene->Update(scene);
    if (scene->scene_end)
    {
        scene->Destroy(scene);
        switch (window)
        {
        case 0:
            create_scene(Menu_L);
            break;
        case 1:
            create_scene(GameScene_L);
            break;
        case -1:
            return false;
        default:
            break;
        }
    }
    return true;
}
```

game_draw

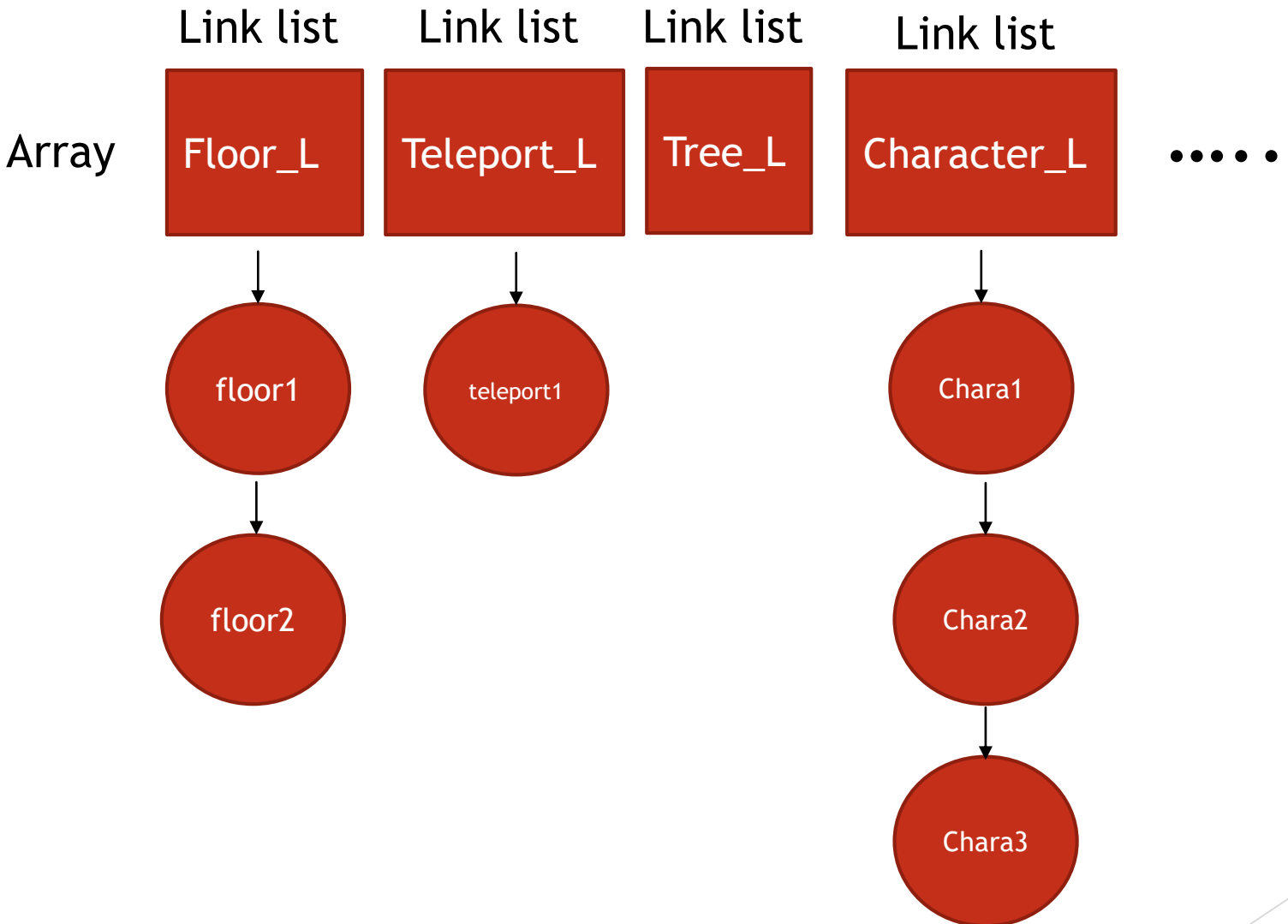
```
void game_draw(Game *game)
{
    // Flush the screen first.
    al_clear_to_color(al_map_rgb(100, 100, 100));
    scene->Draw(scene);
    al_flip_display();
}
```

Scene

scene

```
struct _Scene
{
    int label;
    void *pDerivedObj;
    bool scene_end;
    int ele_num;
    EPNode *ele_list[MAX_ELEMENT];
    // interface for function
    fptrUpdate Update;
    fptrDraw Draw;
    fptrDestroy Destroy;
};
Scene *New_Scene(int label);
```

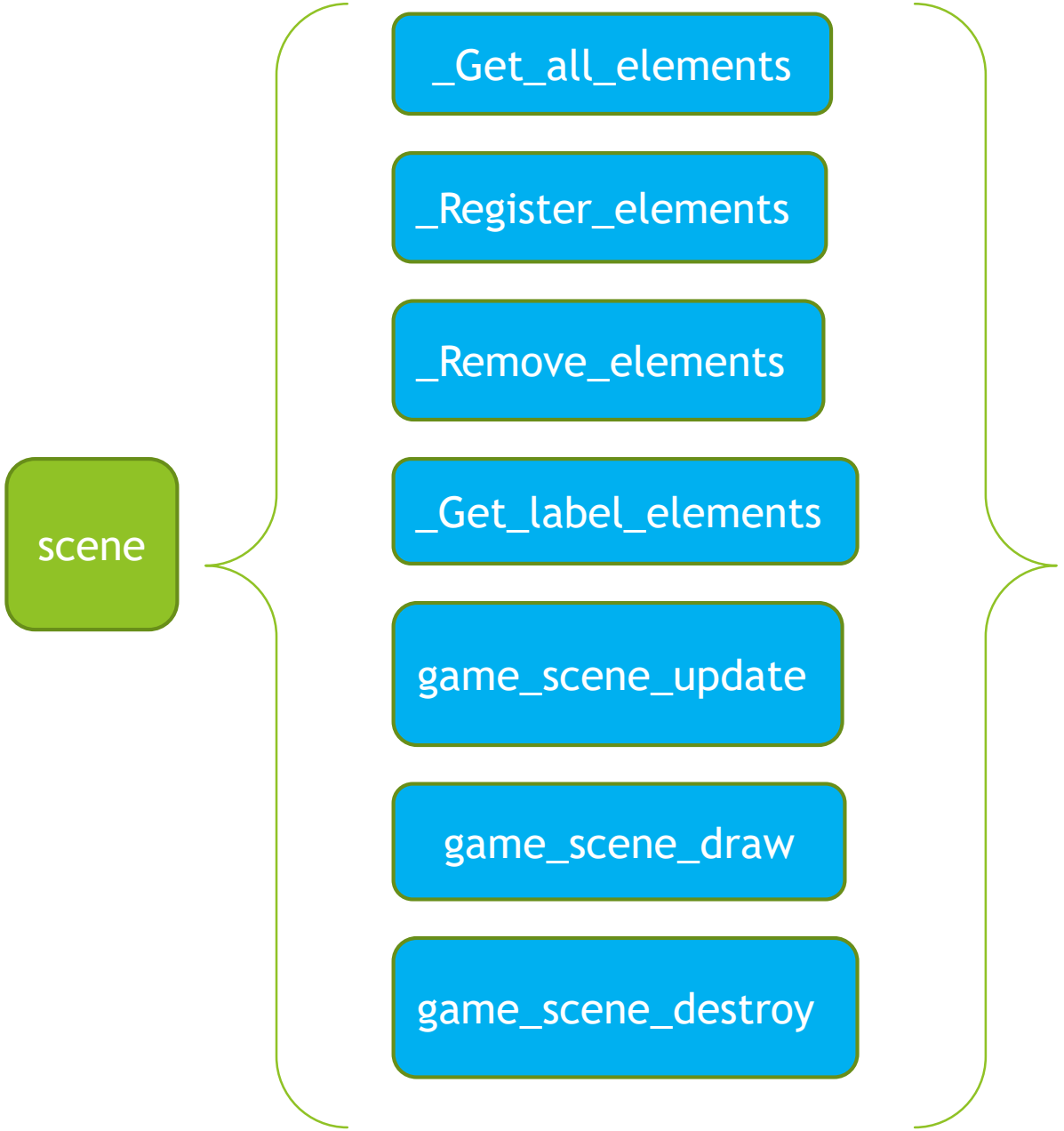
ele_list


Link list
Link list
Link list
Link list
Array
Floor_L
Teleport_L
Tree_L
Character_L
floor1
floor2
teleport1
Chara1
Chara2
Chara3

ele_list

```
EPNode *ele_list[MAX_ELEMENT];
```

```
typedef struct EPNode
{
    int id;
    Elements *ele;
    struct EPNode *next;
} EPNode;
```


scene
_Get_all_elements
_Register_elements
_Remove_elements
_Get_label_elements
game_scene_update
game_scene_draw
game_scene_destroy

Get element

- Return all/certain type element in the scence in a struct type ElementVec
- The ElementVec has two member
  - arr: array store the elements and
  - len: to indicate array length

```
typedef struct Element_vector
{
    Elements *arr[MAX_ELEMENT];
    int len;
} ElementVec;
```

_Get_all_elements

```
ElementVec _Get_all_elements(Scene *const scene)
{
    ElementVec res;
    int size = 0;
    for(int i = 0 ; i < MAX_ELEMENT ; i++){
        if(scene->ele_list[i] == NULL) continue;
        EPNode *ptr = scene->ele_list[i];
        while(ptr){
            res.arr[size++] = ptr->ele;
            ptr = ptr->next;
        }
        if(size == scene->ele_num) break;
    }
    res.len = scene->ele_num;
    return res;
}
```

_Get_label_elements

```
ElementVec _Get_label_elements(Scene *const scene, int label)
{
    EPNode *ptr = scene->ele_list[label];
    ElementVec res;
    int size = 0;
    while (ptr)
    {
        res.arr[size++] = ptr->ele;
        ptr = ptr->next;
    }
    res.len = size;
    return res;
}
```

Register and remove element

_Register_elements

```
void _Register_elements(Scene *const scene, Elements *ele)
{
    EPNode *ptr = scene->ele_list[ele->label];
    EPNode *new_node = (EPNode *)malloc(sizeof(EPNode));
    new_node->id = scene->ele_num++;
    new_node->ele = ele;
    new_node->next = NULL;
    ele->id = new_node->id;
    if (ptr == NULL)
    {
        scene->ele_list[ele->label] = new_node;
    }
    else
    {
        while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = new_node;
    }
}
```

_Remove_elements

```
void _Remove_elements(Scene *const scene, Elements *ele)
{
    EPNode *ptr = scene->ele_list[ele->label];
    EPNode *parent_ptr = NULL;
    while (ptr)
    {
        if (ptr->id == ele->id)
        {
            if (parent_ptr == NULL)
            {
                scene->ele_list[ele->label] = ptr->next;
                free(ptr);
                break;
            }
            else
            {
                parent_ptr->next = ptr->next;
                free(ptr);
                break;
            }
        }
        parent_ptr = ptr;
        ptr = ptr->next;
    }
    scene->ele_num--;
}
```

Register and remove element

- _Register_elements
  - Given element, insert it into link list
  - The element id will be set here
  - scene->ele_num update here
- _Remove_elements
  - Given element delete the element with same id
  - scene->ele_num update here

game_scene_update

For gamescene.c

```
void game_scene_update(Scene *const pGameSceneObj)
{
    // update every element
    ElementVec allEle = _Get_all_elements(pGameSceneObj);
    for (int i = 0; i < allEle.len; i++)
    {
        allEle.arr[i]->Update(allEle.arr[i]);
    }

    // run interact for every element
    for (int i = 0; i < allEle.len; i++)
    {
        Elements *ele = allEle.arr[i];
        // run every interact object
        for (int j = 0; j < ele->inter_len; j++)
        {
            int inter_label = ele->inter_obj[j];
            ElementVec labelEle = _Get_label_elements(pGameSceneObj, inter_label);
            for (int i = 0; i < labelEle.len; i++)
            {
                ele->Interact(ele, labelEle.arr[i]);
            }
        }
    }
    // remove element
    for (int i = 0; i < allEle.len; i++)
    {
        Elements *ele = allEle.arr[i];
        if (ele->dele)
            _Remove_elements(pGameSceneObj, ele);
    }
}
```

game_scene_draw

```
void game_scene_draw(Scene *const pGameSceneObj)
{
    al_clear_to_color(al_map_rgb(0, 0, 0));
    GameScene *gs = ((GameScene *)(pGameSceneObj->pDerivedObj));
    al_draw_bitmap(gs->background, 0, 0, 0);
    ElementVec allEle = _Get_all_elements(pGameSceneObj);
    for (int i = 0; i < allEle.len; i++)
    {
        Elements *ele = allEle.arr[i];
        ele->Draw(ele);
    }
}
```

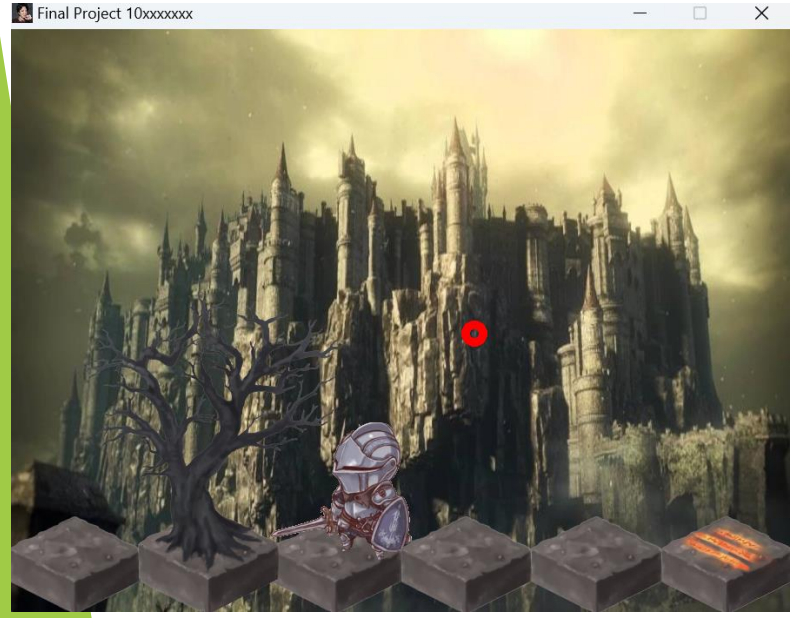
game_scene_destroy

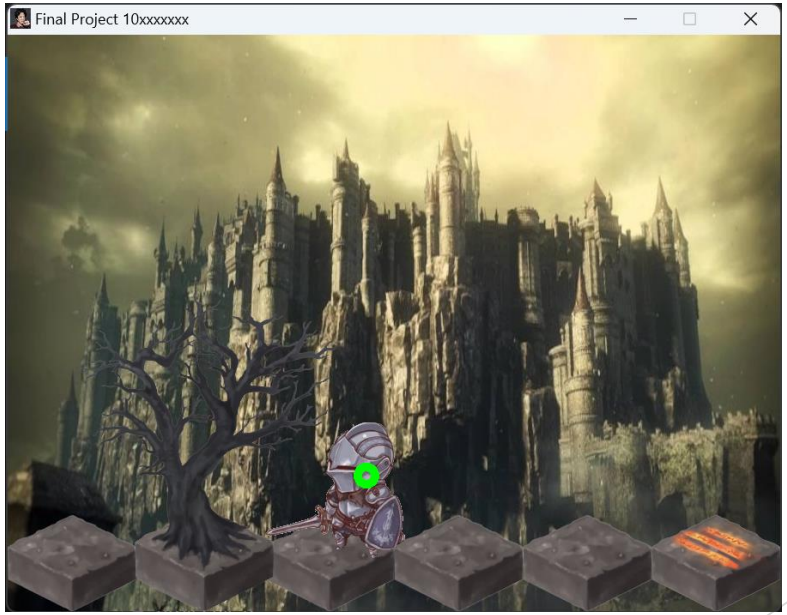
```
void game_scene_destroy(Scene *const pGameSceneObj)
{
    GameScene *Obj = ((GameScene *)(pGameSceneObj->pDerivedObj));
    ALLEGRO_BITMAP *background = Obj->background;
    al_destroy_bitmap(background);
    ElementVec allEle = _Get_all_elements(pGameSceneObj);
    for (int i = 0; i < allEle.len; i++)
    {
        Elements *ele = allEle.arr[i];
        ele->Destroy(ele);
    }
    free(Obj);
    free(pGameSceneObj);
}
```

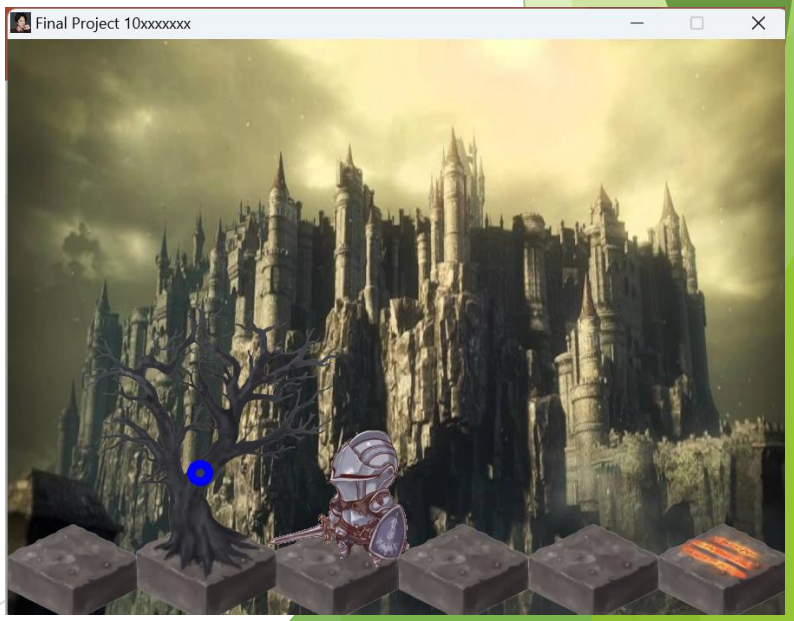
Practice

Target

- A red ball with the cursor
- If touch Character turn green
- If touch Tree turn blue







A red ball with the cursor

- Construct two files
  - Ball.h
  - Ball.c
- Build the struct of Ball → reference tree.h
- Variable:
  - Position: x, y
  - Interactive with Object: hitbox
  - Need color change: color

```c
typedef struct _Ball
{
    int x, y; // the position of image
    int r;    // the radius
    Shape *hitbox; // the hitbox of object
    ALLEGRO_COLOR color;
} Ball;

Elements *New_Ball(int label);
void Ball_update(Elements *self);
void Ball_interact(Elements *self, Elements *tar);
void Ball_draw(Elements *self);
void Ball_destory(Elements *self);
```

A red ball with the cursor

- Implement the function
- Elements *New_Ball(int label) → reference projectile.c
  - New element struct
  - Set the initial value for variable
  - Set the interact object
  - Set the function

```
Elements *New_Ball(int label)
{
    Ball *pDerivedObj = (Ball *)malloc(sizeof(Ball));
    Elements *pObj = New_Elements(label);
    pDerivedObj->x = mouse.x;
    pDerivedObj->y = mouse.y;
    pDerivedObj->r = 10;
    pDerivedObj->color = al_map_rgb(255, 0, 0);
    pDerivedObj->hitbox = New_Circle(pDerivedObj->x,
                                     pDerivedObj->y,
                                     pDerivedObj->r);
    // setting the interact object
    pObj->inter_obj[pObj->inter_len++] = Character_L;
    pObj->inter_obj[pObj->inter_len++] = Tree_L;

    // setting derived object function
    pObj->pDerivedObj = pDerivedObj;
    pObj->Draw = Ball_draw;
    pObj->Update = Ball_update;
    pObj->Interact = Ball_interact;
    pObj->Destroy = Ball_destory;
    return pObj;
}
```

A red circle with the cursor

- Implement the function
- void Ball_update(Elements *self) →reference projectile.c
- Update the x, y with mouse
- Update the hitbox

```c
void Ball_update(Elements *const self)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    Shape *hitbox = Obj->hitbox;
    hitbox->update_center_x(hitbox, mouse.x - Obj->x);
    hitbox->update_center_y(hitbox, mouse.y - Obj->y);
    Obj->x = mouse.x;
    Obj->y = mouse.y;
}
```

A red ball with the cursor

- Implement the function
- void Ball_draw(Elements *self) →reference tree.c
- void Ball_destory(Elements *self)→reference tree.c

```c
void Ball_draw(Elements *self)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    al_draw_circle(Obj->x, Obj->y, Obj->r, Obj->color, 10);
}
void Ball_destory(Elements *self)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    free(Obj->hitbox);
    free(Obj);
    free(self);
}
```

A red ball with the cursor

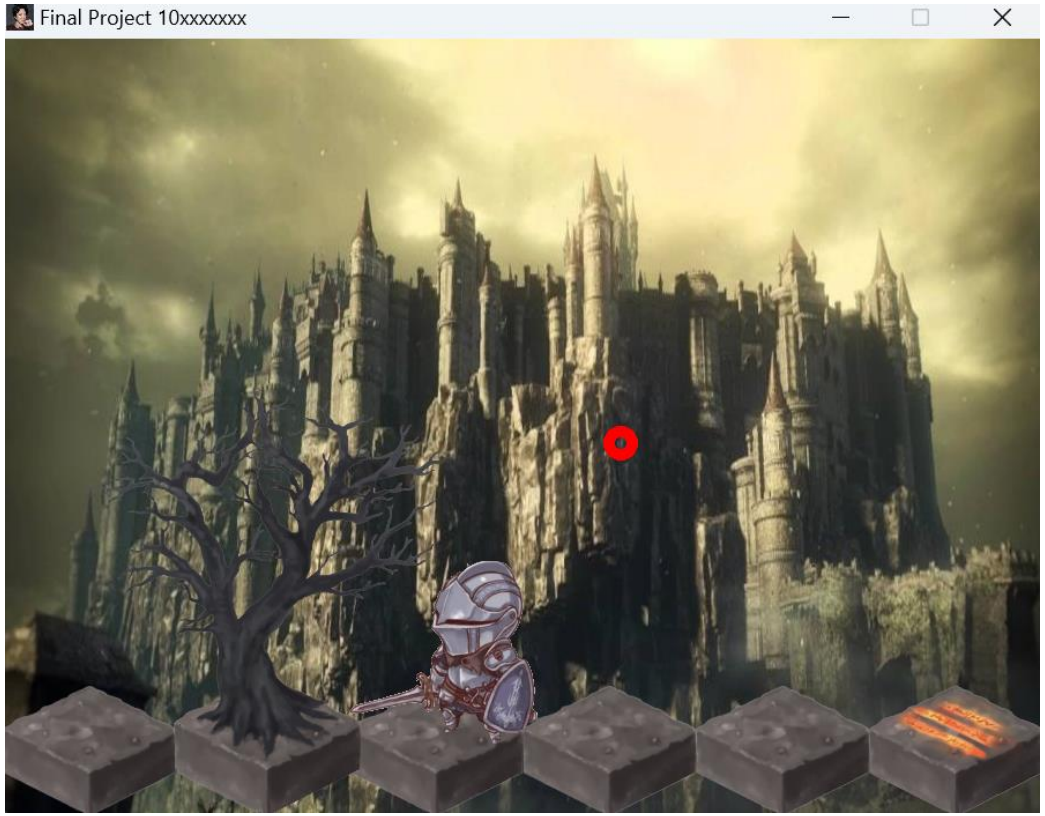
- Add the ball object into game scene
  - In gamescene.h
    - Add new BALL_L label to to enum EleType
  - In gamescene.c
    - In Scene *New_GameScene(int label) function
    - Register the object

```c
typedef enum EleType
{
    Floor_L,
    Teleport_L,
    Tree_L,
    Character_L,
    Projectile_L,
    Ball_L
} EleType;
```

```c
#include "../element/Ball.h"
/*
   [GameScene function]
*/
Scene *New_GameScene(int label)
{
    GameScene *pDerivedObj = (GameScene *)malloc(sizeof(GameScene));
    Scene *pObj = New_Scene(label);
    // setting derived object member
    pDerivedObj->background = al_load_bitmap("assets/image/stage.jpg");
    pObj->pDerivedObj = pDerivedObj;
    // register element
    _Register_elements(pObj, New_Floor(Floor_L));
    _Register_elements(pObj, New_Teleport(Teleport_L));
    _Register_elements(pObj, New_Tree(Tree_L));
    _Register_elements(pObj, New_Character(Character_L));
    _Register_elements(pObj, New_Ball(Ball_L));
    // setting derived object function
    pObj->Update = game_scene_update;
    pObj->Draw = game_scene_draw;
    pObj->Destroy = game_scene_destroy;
    return pObj;
}
```

A red ball with the cursor

- At this step you can already have a circle with your cursor



If touch Character turn green

- Implement the function
- void Ball_interact(Elements * self, Elements * tar) → reference projectile.c

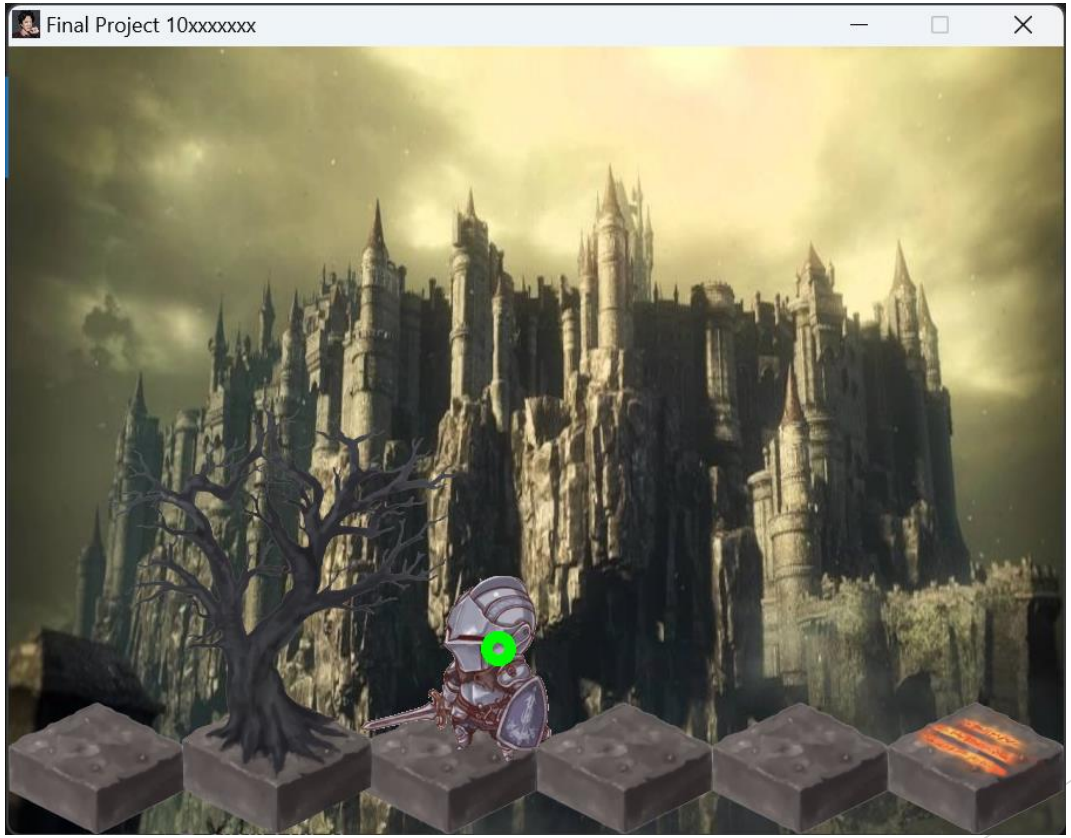
```
void Ball_interact(Elements *self, Elements *tar)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    if (tar->label == Character_L)
    {

        Character *Obj2 = ((Character *)(tar->pDerivedObj));

        if (Obj->hitbox->overlap(Obj->hitbox, Obj2->hitbox))
        {
            Obj->color = al_map_rgb(0, 255, 0);
            Obj->in = Character_L;
        }
        else
        {
            Obj->color = al_map_rgb(255, 0, 0);
            Obj->in = -1;
        }
    }
}
```

If touch Character turn green

- At this step you can change the color of your circle



If touch Tree turn blue

- Implement the function
- void Ball_interact(Elements * self, Elements * trar) → reference projectile.c
  - Add condition after the character

```
void Ball_interact(Elements *self, Elements *tar)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    if (tar->label == Character_L)
    {

        Character *Obj2 = ((Character *)(tar->pDerivedObj));

        if (Obj->hitbox->overlap(Obj->hitbox, Obj2->hitbox))
        {
            Obj->color = al_map_rgb(0, 255, 0);
        }
        else
        {
            Obj->color = al_map_rgb(255, 0, 0);
        }
    }
    else if (tar->label == Tree_L)
    {
        Tree *Obj2 = ((Tree *)(tar->pDerivedObj));
        if (Obj->hitbox->overlap(Obj->hitbox, Obj2->hitbox))
        {
            Obj->color = al_map_rgb(0, 0, 255);
        }
        else
        {
            Obj->color = al_map_rgb(255, 0, 0);
        }
    }
}
```

Bug?

- The ball can not turn green for the character
- Because the color was overwrite by the tree
  - We need to determined if we are in our target
  - Add another variable the struct to determine where the ball is in

```c
typedef struct _Ball
{
    int x, y; // the position of image
    int r;    // the radius
    int in;
    Shape *hitbox; // the hitbox of object
    ALLEGRO_COLOR color;
} Ball;
```

```c
Elements *New_Ball(int label)
{
    Ball *pDerivedObj = (Ball *)malloc(sizeof(Ball));
    Elements *pObj = New_Elements(label);
    pDerivedObj->x = mouse.x;
    pDerivedObj->y = mouse.y;
    pDerivedObj->r = 10;
    pDerivedObj->in = -1;
    pDerivedObj->color = al_map_rgb(255, 0, 0);
    pDerivedObj->hitbox = New_Circle(pDerivedObj->x,
                                     pDerivedObj->y,
                                     pDerivedObj->r);
    // setting the interact object
    pObj->inter_obj[pObj->inter_len++] = Character_L;
    pObj->inter_obj[pObj->inter_len++] = Tree_L;

    // setting derived object function
    pObj->pDerivedObj = pDerivedObj;
    pObj->Draw = Ball_draw;
    pObj->Update = Ball_update;
    pObj->Interact = Ball_interact;
    pObj->Destroy = Ball_destory;
    return pObj;
}
```

```c
void Ball_interact(Elements *self, Elements *tar)
{
    Ball *Obj = ((Ball *)(self->pDerivedObj));
    if (tar->label == Character_L)
    {

        Character *Obj2 = ((Character *)(tar->pDerivedObj));

        if (Obj->hitbox->overlap(Obj->hitbox, Obj2->hitbox))
        {
            Obj->color = al_map_rgb(0, 255, 0);
            Obj->in = Character_L;
        }
        else if (Obj->in == Character_L)
        {
            Obj->color = al_map_rgb(255, 0, 0);
            Obj->in = -1;
        }
    }
    else if (tar->label == Tree_L)
    {
        Tree *Obj2 = ((Tree *)(tar->pDerivedObj));
        if (Obj->hitbox->overlap(Obj->hitbox, Obj2->hitbox))
        {
            Obj->color = al_map_rgb(0, 0, 255);
            Obj->in = Tree_L;
        }
        else if (Obj->in == Tree_L)
        {
            Obj->color = al_map_rgb(255, 0, 0);
            Obj->in = -1;
        }
    }
}
```

The answer file

- Tutorial folder -> Practice_answer folder