华南理工大学

South China University of Technology

《High-level Language Programming Project》Report

Project Name: Celeste

School: 未来技术学院

Major: 数据科学与大数据技术专业

Student Name : 霍君安　龚思嘉　周宇哲

Teacher: 张怀东

Submission Date: 2024 年 1 月 17 日

Final report for “Celeste”

1. Requirement Analysis for System

1.1 The Background and Motivation of System

"Celeste" is renowned as the Best Independent Game of TGA 2018. *"Celeste is a surprise masterpiece,"* said IGN, awarding it a perfect score of 10.

In today's society, people face endless challenges and pressures. "Celeste" is about the inner journey of a young woman named Madeline, who decides to challenge herself by climbing Mount Celeste. Throughout her journey, she not only confronts external physical challenges but also grapples with inner fears and self-doubt.

In modern life, everyone has their own "Mount Celeste" to climb. Just like Madeline, as we ascend these "peaks," we face not only external obstacles but also the fears and doubts within us.

Inspired by this concept, our team decided to emulate the game "Celest," integrating elements of meditation and self-exploration. We hope players can not only experience the challenge of mountain climbing but also delve deep into their own psyche, confronting and overcoming their fears and challenges.

Through this game, we aspire for players to meditate and explore themselves in a tranquil and relaxing environment, while also experiencing the thrill and satisfaction of climbing "Mount Celeste." We believe that this game will not only entertain and relax players but also aid them in better confronting challenges and pressures in real life.

“This is it. Just breathe. You can do it.”

1.2 System Objectives

1.2.1 Architecture

We split the game into three parts: UI interface, game scenes, players.

- The structure of UI interface is: background, buttons, music and sound.
- The structure of the game scene is: physical world building, trap, listening (player collision listening, trap listening).
- The structure of player class is: keyboard and mouse listening, action (jumping, walking), character skills.

The parts are connected to each other by scene switching and buttons, etc.

1.2.2 Functions

- Game start interface, background music, and exit function
- Keyboard for character walking and jumping, mouse for adjust settings and enter levels
- Effects of skills and traps
- Pause and save function
- Movement display

2. Program Analysis

2.1 Key Issues for System

By studying the cocos2dx code of other games, we understand in general how to implement a game. Here are some of the key issues we have found through learning and practice.

a) Achieve continuous character movement

Technical Difficulties: When the character moves the movement needs to keep pressing the button in order to move continuously.

Preliminary solution: Use "*updata*" function and a keyboard listening to solve. And design a function for animation playback control.

b) Realize smooth character jump

Technical Difficulties: The collision between the character and the platform makes the movement of the character in the game become abnormal.

Preliminary solution: Using impulse instead of directly setting the velocity. Using raycasting to determine the validity of movement. Define the collision mask for players and obstacles, as well as their physical properties, such as whether they can rotate or move, the coefficient of friction, whether they are rigid bodies, the coefficient of restitution, etc.

c) Complex character state transitions and animation playback

Technical Difficulties: In platformer games, there are a considerable number of character states that need to be coordinated. Smoothly transitioning between them and ensuring the correct animation playback is a challenge.

Preliminary solution: Using finite state machines and a "*changeState*" function for control and state transitions. Define the collision mask for players and obstacles, as well as their physical properties, such as whether they can rotate or move, the coefficient of friction, whether they are rigid bodies, the

coefficient of restitution, etc.

d) Ground detection and wall detection

Technical Difficulties: Ground detection and wall detection are crucial for the transition between a character's jumping logic and wall-climbing logic. But the existing settings in the physics engine cannot perfectly address the issues.

Preliminary solution: Using the method of multiple raycasting. For ground detection, we cast a 1-pixel long ray downwards from both the leftmost and rightmost parts of the character, then check if the ray collides with an object named "ground". If it does, we set the *isOnGround* variable to 1 and refresh the limitations for the dash action. The wall detection is similar; we cast a 1-pixel long ray in the direction the character is facing and check for collisions with objects named "ground". If there's a collision, we set the *canClimb* variable to 1.

e) Complex music switching and control

Technical Difficulties: Outstanding music design is another advantage of the game 'Celeste', but it also ensures the complexity of its design.

Preliminary solution: Background music, character skill sound effects, and trap effects are managed in a layered manner to ensure the correctness of music playback logic.

f) The game's freedom and the complexity of its levels

Technical Difficulties: Due to the high degree of player freedom in this platformer game, unexpected situations may arise that cause the program to run incorrectly.

Preliminary solution: Repeated debugging. While ensuring the richness and fun of the levels, we will reduce some similar stages to ensure our program runs correctly at all times.

g) Control responsiveness and the coordination with visuals and music

Technical Difficulties: Combining visuals with gameplay controls inherently presents some challenges, and for platformer games, the responsiveness of controls is especially crucial. Perfectly integrating all three elements is a significant challenge.

Preliminary solution: Try to find as many people as possible to experience the game and collect their feedback on the game's controls. Also, appropriately reference the original game's numerical settings and strive to emulate the perfect rhythm of the original work.

h) Transition between levels

Technical Difficulties: General level transitions involve a transition area,

namely the check point area. And in the *update* function for *level* class, it checks whether the player collides or overlaps with that area. However, we have always failed to successfully use the engine's built-in collision detection

Preliminary solution: We abandoned the method of setting collision areas and instead adopted the ray detection method, which we are more familiar with in this project, to solve the problem.

i) Storing the data of the level where the player is located

Technical Difficulties: The function used for storing data is unable to discern which level the player is in, making it difficult to save the correct data.

Preliminary solution: Add a static public variable '*currentLevel*' to the Player class, and assign an appropriate value to this variable when adding the Player class in different levels to indicate the current level. This allows the functions in the *PauseMenu* to obtain the correct value.

2.2 Duty Assignments

Name	Student Number	Work
霍君安	202230051085	Overall framework writing, Designing the main interface ,Player Class, Trap Class, Level Base Class, Special effect, Material organization, Animation management, Music & Sound, Debug
龚思嘉	202264701109	UI Design, Setting, Staff, Database, Debug
周宇哲	202230054123	Level Class, Level Base, Physical collision, Debug

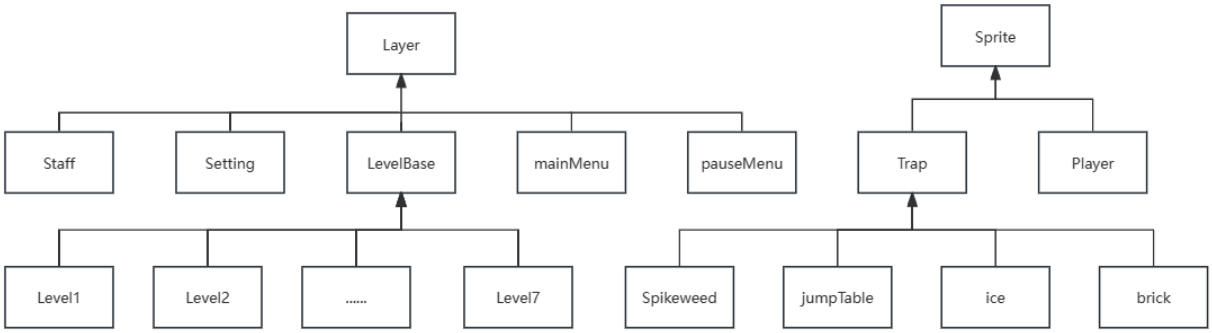
3. Technical Routine

3.1 Runtime Environment

environment	Version
Visual Studio	VS2022
Cocos-2dx	4.0
Windows	Windows 10, Windows 11

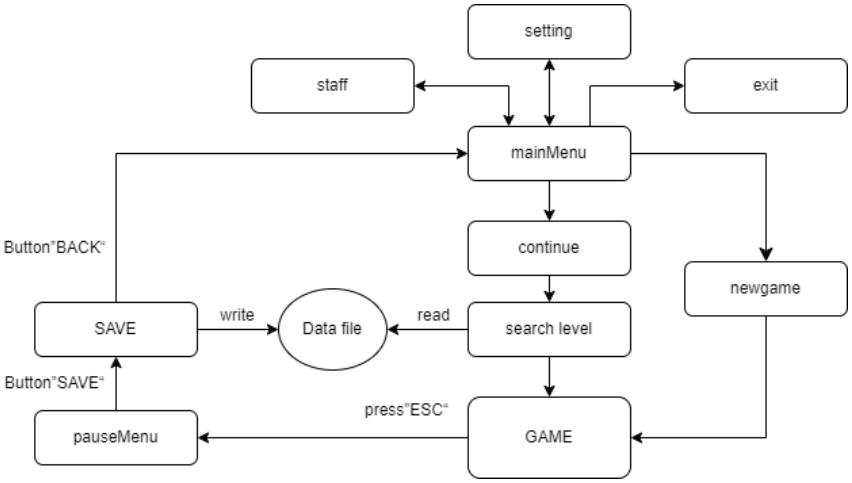
3.2 General Design

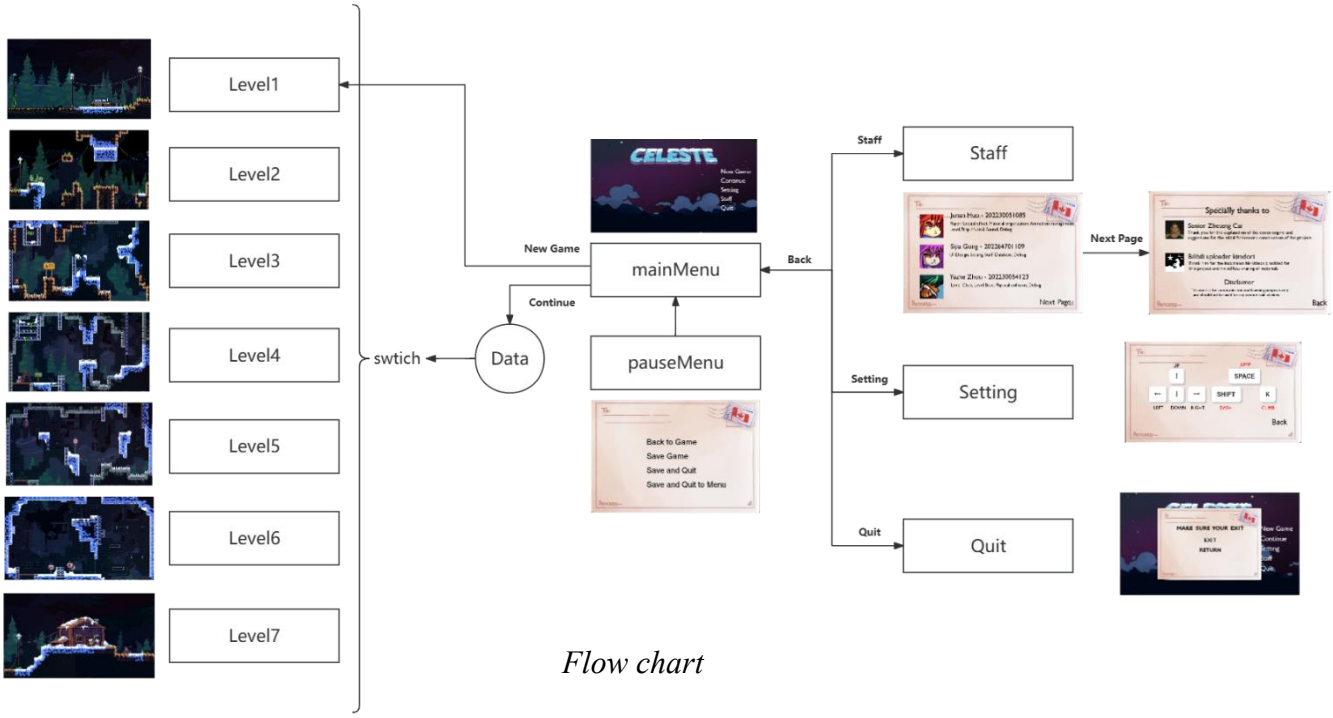
3.2.1 Class diagram UML



Inheritance Relationship Diagram

3.2.2 Game flowchart



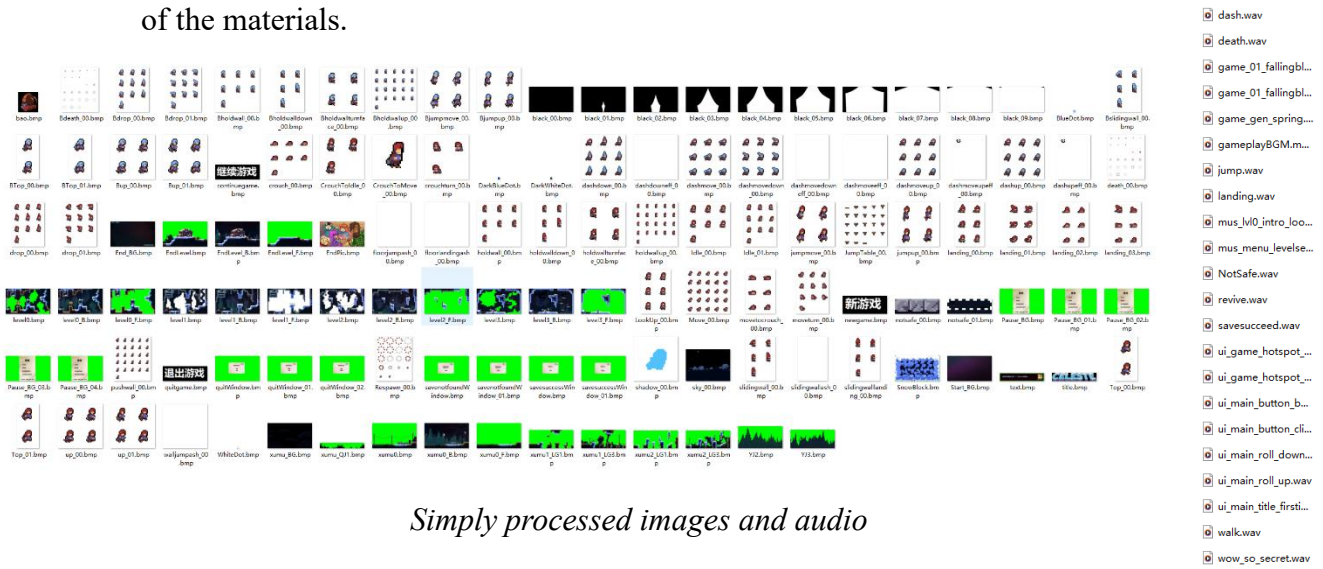


Flow chart

3.3 Detailed Design

3.3.1 Material Collection, Processing and Loading

The game materials originate from the selfless sharing of Bilibili UP *Kimdor,* who obtained them from the game Celeste using reverse engineering unpacking methods. After unpacking, he also did some simple processing and categorization of the materials.



Simply processed images and audio

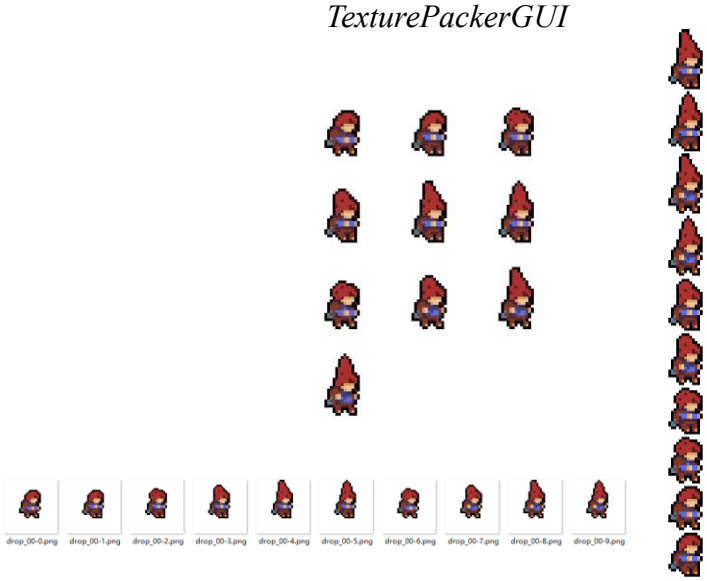
Most of the simply processed image materials are in BMP format, which preserves the integrity of the images to the greatest extent. However, since the Cocos engine does not support importing images in this format, and BMP format images do not have a transparent layer for layering, I used *Photoshop* to convert BMP format images into PNG format images with a transparent channel.

BMP vs PNG format comparison

Adobe Photoshop

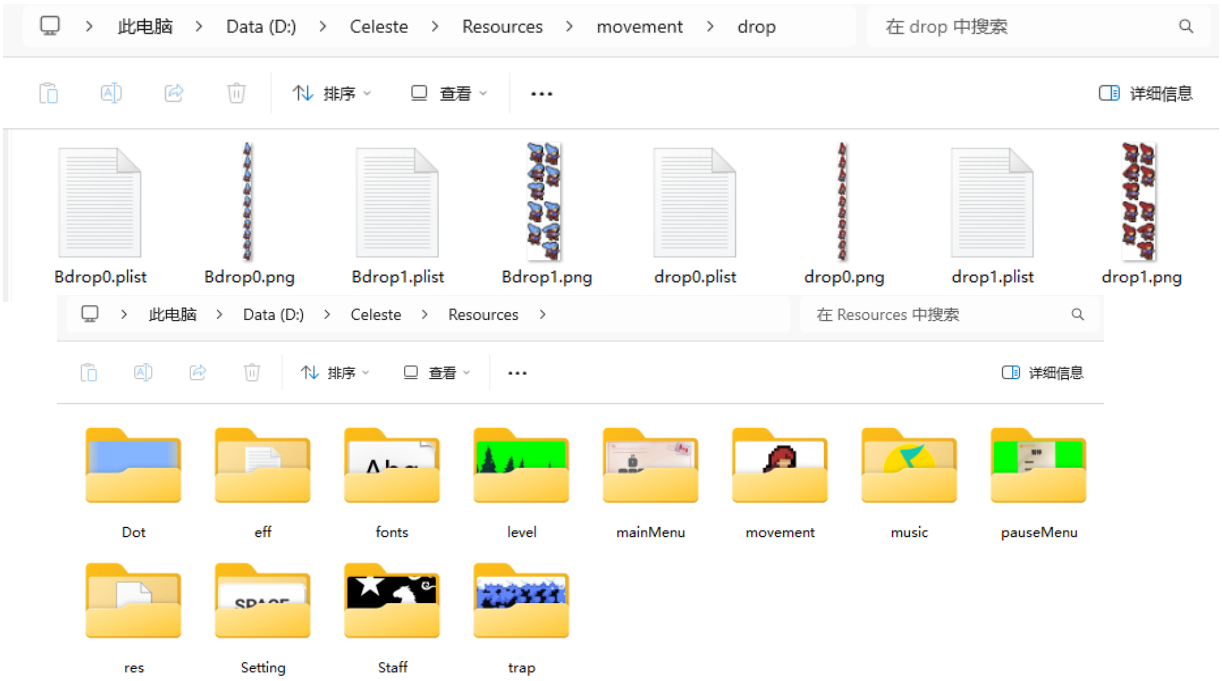
For the smoothness of the game, an animation system was adopted, along with the use of texture packing technology. This is a common technique in image processing and game development, used to optimize and manage the storage of multiple small images or textures. By combining many small images or textures into one large texture atlas, rendering efficiency can be increased, memory usage reduced, and the number of draw calls to the Graphics Processing Unit (GPU) decreased

TexturePackerGUI



Comparison of images before and after processing with TP

Taking the drop animation of the player as an example, initially we received a larger sprite sheet like the one in the upper left corner. It was necessary to first use TP to split this large sprite sheet into several smaller sprite images (shown in the lower left corner). Finally, these smaller images were imported into TP to generate a packed image as shown in the right image, along with a plist file. It can be visually observed from the images that the TP-packed image is more compact and occupies less space.



Final storage result in Resources

The plist file records the positions of the original sequence of images in the combined image. For repeated images, TP does not save them repeatedly but instead records them repeatedly in the plist file to achieve a reuse effect.

For a game like Celeste, which relies heavily on animation, the animation frames processed by TP make the import and utilization of materials in Cocos more convenient. During the import, it is only necessary to input the generated .plist file and the combined large image .png, as shown in the following line of code and the image below.

SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/crouch/crouch_00.plist", "movement/crouch/crouch_00.png");//蹲姿加载

```
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/crouch/crouch_00.plist", "movement/crouch/crouch_00.png");//蹲姿加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/crouch/CrouchToIdle.plist", "movement/crouch/CrouchToIdle.png");//蹲姿到静止加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/crouch/CrouchToMove.plist", "movement/crouch/CrouchToMove.png");//蹲姿到移动加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/dash/dashdown.plist", "movement/dash/dashdown.png");//向下冲刺加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/dash/dashmove.plist", "movement/dash/dashmove.png");//移动冲刺加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/dash/dashup.plist", "movement/dash/dashup.png");//向上冲刺加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/death/Bdeath.plist", "movement/death/Bdeath.png");//B死亡
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/death/death.plist", "movement/death/death.png");//死亡
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/death/Respawn.plist", "movement/death/Respawn.png");//重生
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/drop/drop0.plist", "movement/drop/drop0.png");//向下坠落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/drop/drop1.plist", "movement/drop/drop1.png");//移动坠落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/drop/Bdrop0.plist", "movement/drop/Bdrop0.png");//B向下坠落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/drop/Bdrop1.plist", "movement/drop/Bdrop1.png");//B移动坠落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/holdwall.plist", "movement/holdwall/holdwall.png");//爬墙加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/Bholdwall.plist", "movement/holdwall/Bholdwall.png");//B爬墙加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/holdwallup.plist", "movement/holdwall/holdwallup.png");//爬墙向上加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/Bholdwallup.plist", "movement/holdwall/Bholdwallup.png");//B爬墙向上加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/holdwalldown.plist", "movement/holdwall/holdwalldown.png");//爬墙向下加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/holdwall/Bholdwalldown.plist", "movement/holdwall/Bholdwalldown.png");//B爬墙向下加载    //可加爬墙turnface
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/idle/idle_0.plist", "movement/idle/idle_0.png");//站立1加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/idle/Idle1.plist", "movement/idle/Idle1.png");//站立2加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/jump/jumpup.plist", "movement/jump/jumpup.png");//跳跃加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/jump/jumpmove.plist", "movement/jump/jumpmove.png");//跳跃移动加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/top/Top0.plist", "movement/top/Top0.png");//跳跃补全
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/jump/Bjumpup.plist", "movement/jump/Bjumpup.png");//B跳跃加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/jump/Bjumpmove.plist", "movement/jump/Bjumpmove.png");//B跳跃移动加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/landing/landing0.plist", "movement/landing/landing0.png");//落地加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/landing/landing1.plist", "movement/landing/landing1.png");//移动落地加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/lookup/LookUp.plist", "movement/lookup/LookUp.png");//向上看加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/move/Move.plist", "movement/move/Move.png");//移动加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/move/MoveToCrouch.plist", "movement/move/MoveToCrouch.png");//移动到蹲加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/move/MoveTurn.plist", "movement/move/MoveTurn.png");//移动转向加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/pushwall/pushwall.plist", "movement/pushwall/pushwall.png");//推墙加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/shadow/shadow.plist", "movement/shadow/shadow.png");//残影加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/slidingwall/slidingwall.plist", "movement/slidingwall/slidingwall.png");//滑落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/slidingwall/Bslidingwall.plist", "movement/slidingwall/Bslidingwall.png");//滑落加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/black.plist", "eff/black.png");//黑幕加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/floorlandingash_00.plist", "eff/floorlandingash_00.png");//落地烟雾加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/floorjumpash_00.plist", "eff/floorjumpash_00.png");//跳跃烟雾加载
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/walljumpash_00.plist", "eff/walljumpash_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/slidingwallash_00.plist", "eff/slidingwallash_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/dashupeff_00.plist", "eff/dashupeff_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/dashmoveupeff_00.plist", "eff/dashmoveupeff_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/dashmoveeff_00.plist", "eff/dashmoveeff_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/dashmovedowneff_00.plist", "eff/dashmovedowneff_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("eff/dashdowneff_00.plist", "eff/dashdowneff_00.png");//
SpriteFrameCache::getInstance()->addSpriteFramesWithFile("movement/jumptable/jumpTable.plist", "movement/jumptable/jumpTable.png");//
```

AppDelegate.cpp loading plist

In the actual program, all animation materials (including the player's animations, trap animations, and special effect animations) are imported in AppDelegate.cpp (the main entry point of the program). With TP's processing, the import becomes simpler and more standardized.

```
Vector<SpriteFrame*> idleFrames;
auto cache = SpriteFrameCache::getInstance();
for (int i = 0; i <= 22; i++) {
    std::string frameName = StringUtils::format("JumpTable_00-%d.png", i);
    auto frame = cache->getSpriteFrameByName(frameName);
    if (frame) {
        idleFrames.pushBack(frame);
        CCLOG("Loaded frame: %s", frameName.c_str());
    }
    else {
        CCLOG("Error: Cannot find frame: %s", frameName.c_str());
    }
}
```

Plist test

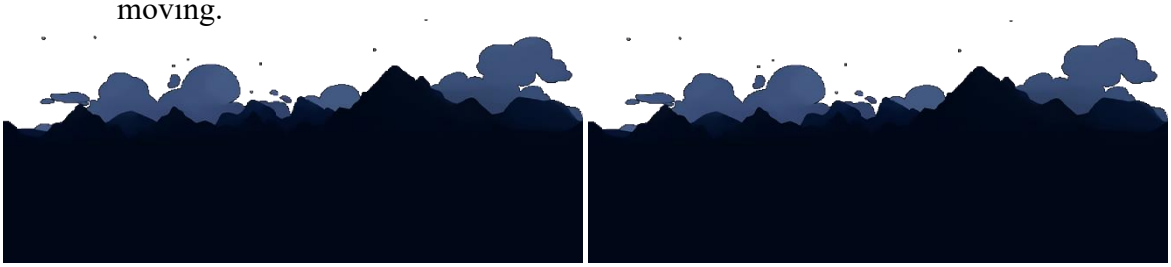
After importing, the code shown in the image above can also be used for debugging.

3.3.2 Main Menu Scene and Other Scene

3.3.2.1 Main Menu Scene

Main scene

In the design of the main interface's background, we used a special graphic that can be seamlessly connected on both sides. By continuously looping this image, we gave the impression that the mountains in the main interface are moving.



background graphic

```
void MainMenuScene::updateBackground(float dt)
{
    // 持续地移动两个背景图片
    bg1->setPositionX(bg1->getPositionX() - 1);
    bg2->setPositionX(bg2->getPositionX() - 1);

    // 如果bg1移出屏幕左侧，则重置它的位置到bg2的右侧
    if (bg1->getPositionX() + bg1->getContentSize().width * bg1->getScaleX() < 0)
    {
        bg1->setPositionX(bg2->getPositionX() + bg2->getContentSize().width * bg2->getScaleX());
    }

    // 如果bg2移出屏幕左侧，则重置它的位置到bg1的右侧
    if (bg2->getPositionX() + bg2->getContentSize().width * bg2->getScaleX() < 0)
    {
        bg2->setPositionX(bg1->getPositionX() + bg1->getContentSize().width * bg1->getScaleX());
    }
}
```

In the main interface, there are a total of five buttons. "New" is for starting a new game, which means beginning from the first level. "Continue" allows the player to resume from the last saved level. "Setting" and "Staff" display the key bindings and the team members, respectively. "Quit" is for exiting to the desktop.

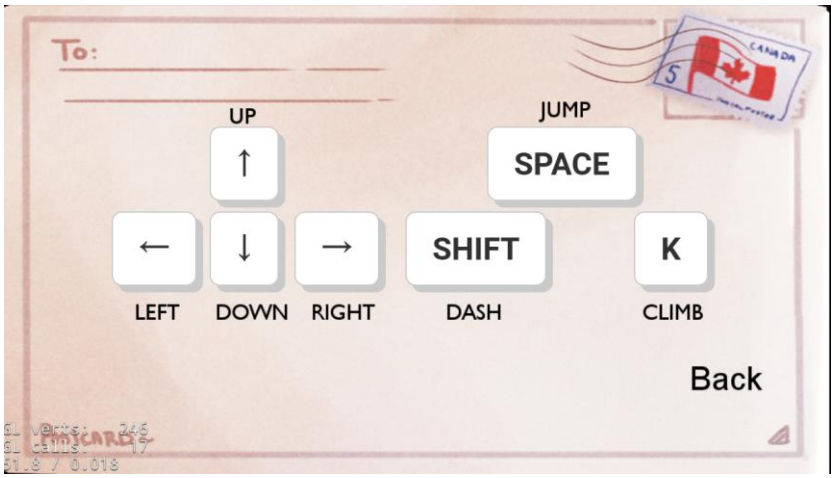
When the mouse hovers over each button, a small snow mountain icon appears in front of the button to indicate that the player has currently selected that button:

snow mountain icon

```
bool isOverAnyButton = false;

for (auto& pair : buttons) {
    cocos2d::ui::Button* button = pair.first;
    Vec2 iconPos = pair.second;

    if (button && getGlobalBoundingBox(button).containsPoint(Vec2(e->getCursorX(), e->getCursorY()))) {
        mountainSprite->setVisible(true);
        mountainSprite->setPosition(iconPos);
        mountainSprite->setLocalZOrder(button->getLocalZOrder() + 1); // 确保小图标出现在相应按钮的前面
        isOverAnyButton = true;
        break;  // 如果鼠标在某个按钮上，我们可以跳出循环
    }
}

if (!isOverAnyButton) {
    mountainSprite->setVisible(false);
}
};
```

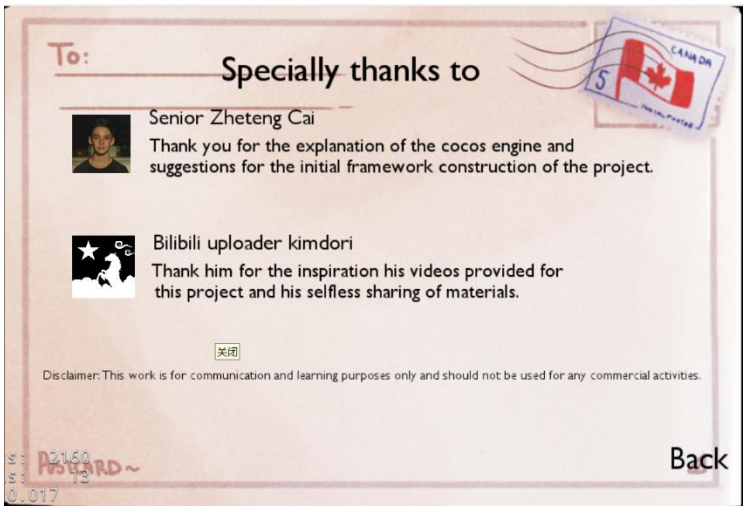
Code for snow mountain icon

3.3.2.2 Setting Scene



setting

3.3.2.3 Staff Scene



staff

In the main interface, when the "Setting" and "Staff" buttons are pressed, they respectively open the key binding settings and the team member display interface. These interfaces mainly use images and text for presentation. The "Staff" interface has two pages, showing the team members and special acknowledgments, connected by a "NextPage" button.

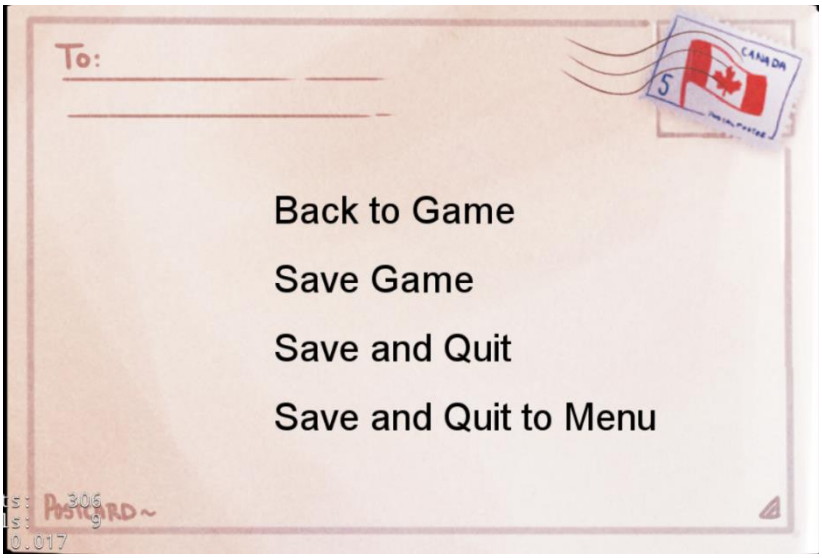
3.3.2.4 Quit Scene

quit

When the "Quit" button on the main interface is pressed, a dialogue pops up to confirm whether the player wants to exit to the desktop. The design for the Quit button is directly integrated into the initialization of the main interface.

```
exitDialog->setVisible(false); // 初始时隐藏它
quit->addTouchEventListener([=](Ref* sender, cocos2d::ui::Widget::TouchEventType type) {
    if (type == cocos2d::ui::Widget::TouchEventType::ENDED) {
        exitDialog->setVisible(true);
    }
```

3.3.2.5 Pause Menu Scene



Pause menu

The pause interface has four buttons: Return to Game, Save Game, Save and Quit, and Save and Exit to Desktop.

When the player presses the ESC key during a level, the pause menu appears:

```
void Level1Scene::onKeyPressed(EventKeyboard::KeyCode keycode, Event* event) {
    if (keycode == EventKeyboard::KeyCode::KEY_ESCAPE) {
        // ESC 键按下，切换到另一个场景
        auto pauseLayer = PauseMenu::create();
        Director::getInstance()->getRunningScene()->pause();
        Director::getInstance()->pushScene(pauseLayer);
    }
}
```

Pressing "Return to Game" in the pause interface ends the pause and returns to the game.

```
backGame->addTouchEventListener([](Ref* sender, cocos2d::ui::Widget::TouchEventType type) {
    if (type == cocos2d::ui::Widget::TouchEventType::ENDED) {
        Director::getInstance()->getRunningScene()->resume();
        Director::getInstance()->popScene();
    }
    });
```

Pressing "Save Game" in the pause interface saves the current level of the character into the save file.

```
saveGame->addTouchEventListener([this, saveEffectFile](Ref* sender, cocos2d::ui::Widget::TouchEventType type) { // 注意这里的[this]
    if (type == cocos2d::ui::Widget::TouchEventType::ENDED) {
        // 播放保存按钮的音效
        cocos2d::AudioEngine::play2d(saveEffectFile);

        this->SaveFile(Player::currentLevel); // 使用this来调用SaveFile
        Director::getInstance()->pause();
        auto scene = PauseOverlay::create();
        Director::getInstance()->pushScene(scene);
    }
    });
```

Pressing "Save and Quit" in the pause interface saves the current level of the character into the save file and immediately exits to the desktop.

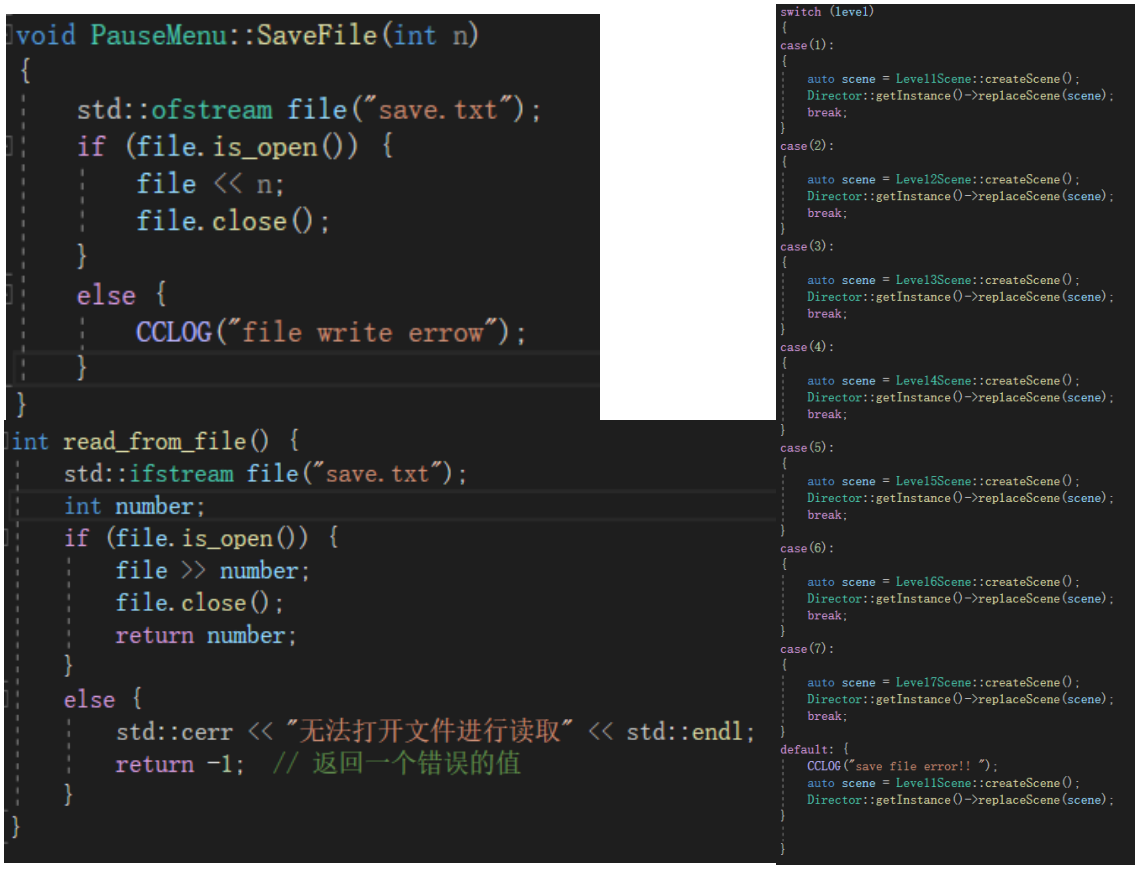
```
saveTable->addTouchEventListener([this](Ref* sender, cocos2d::ui::Widget::TouchEventType type) {
    if (type == cocos2d::ui::Widget::TouchEventType::ENDED) {
        this->SaveFile(Player::currentLevel); // 使用this来调用SaveFile
        auto scene = PauseOverlay::create();
        Director director;
        director.runWithScene(scene);
        cocos2d::Director::getInstance()->end();
    }
    });
```

Pressing "Save and Return to Menu" in the pause interface saves the current level of the character into the save file and regenerates the main menu scene.

```
saveMenu->addTouchEventListener([this](Ref* sender, cocos2d::ui::Widget::TouchEventType type) {
    if (type == cocos2d::ui::Widget::TouchEventType::ENDED) {
        this->SaveFile(Player::currentLevel); // 使用this来调用SaveFile
        auto scene = MainMenuScene::create();
        Director::getInstance()->replaceScene(TransitionFade::create(1.0, scene));
    }
    });
```

After pressing the second, third, or fourth button, a small window pops up indicating a successful save.

Below is the implementation of the function that saves the player's current level into the save file. When the player presses the 2nd, 3rd, or 4th button in the pause menu, the current level of the player is saved into the save file. When the "Continue" button is pressed on the main interface, the latest saved level from the save file is loaded. Through a switch statement, the game enters the correct level.



```
void PauseMenu::SaveFile(int n)
{
    std::ofstream file("save.txt");
    if (file.is_open()) {
        file << n;
        file.close();
    }
    else {
        CCLOG("file write errow");
    }
}
int read_from_file() {
    std::ifstream file("save.txt");
    int number;
    if (file.is_open()) {
        file >> number;
        file.close();
        return number;
    }
    else {
        std::cerr << "无法打开文件进行读取" << std::endl;
        return -1;  // 返回一个错误的值
    }
}
```

```
switch (level)
{
case(1):
{
    auto scene = Level1Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(2):
{
    auto scene = Level2Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(3):
{
    auto scene = Level3Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(4):
{
    auto scene = Level4Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(5):
{
    auto scene = Level5Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(6):
{
    auto scene = Level6Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
case(7):
{
    auto scene = Level7Scene::createScene();
    Director::getInstance()->replaceScene(scene);
    break;
}
default: {
    CCLOG("save file error!! ");
    auto scene = Level1Scene::createScene();
    Director::getInstance()->replaceScene(scene);
}
}
```

Save, read and load

3.3.3 Level

3.3.3.1 LevelBase Class

When *LevelScene* inherits from *LevelBase*, they inherit all of these generic features. Each level-specific scene can then add or modify specific features and elements based on this foundation.

Specifically, for scene creation and initialization, *LevelScene* inherits from *LevelBase* and may modify the background, platform layout, or physical properties. For level loading, *LevelBase* provides the basic framework for the *loadLevel* method, while *LevelScene* loads specific elements and settings as needed for each level.

```
class LevelBase : public cocos2d::Layer {
public:
    int _backgroundMusicID = -1; // 背景音乐ID

    // 使用静态create函数来创建层
    static LevelBase* create();

    // 初始化函数
    virtual bool init() override;

    // 加载关卡特有内容的函数（抽象函数）
    virtual void loadLevel() = 0;

    // 开始、结束、暂停游戏等通用接口（如果需要）
    virtual void startGame() = 0;
    virtual void endGame() = 0;
    virtual void pauseGame() = 0;

    // 通用的物理世界初始化设置
    virtual cocos2d::Scene* createScene();
```

By inheriting *LevelBase*, *LevelScene* inherits a set of basic game scenario functions. This inheritance mechanism allows each level to reuse common code and focus on implementing its own unique features and behaviors. This design reduces code duplication and makes game development more efficient, while keeping the code organized and maintainable.

3.3.3.2 Level Class

3.3.3.2.1 Create a Scene

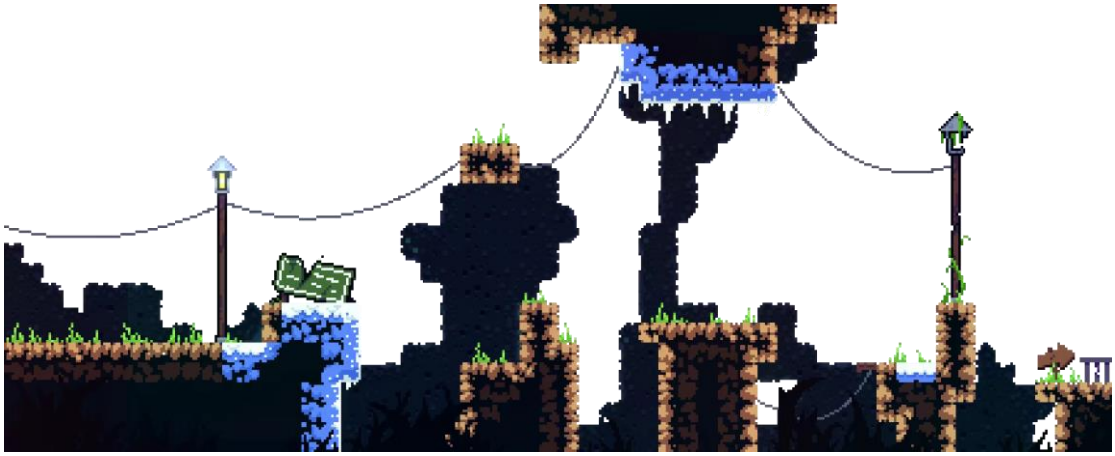
The *createScene* method creates a scene containing the physical world. This is accomplished with *Scene::createWithPhysics()*, which creates an instance of the scene and attaches a physics world. Physics worlds are used to simulate real-world physical effects such as gravity and collisions. In this scene, gravity is set Vec2(0, -1200) for downward gravity, debug drawing is turned on (to make it easier to view physics boundaries and shapes during development), and the physics world's substeps are set to 60, which helps to simulate the physics behavior more finely.

```cpp
cocos2d::Scene* Level3Scene::createScene() {
    auto scene = Scene::createWithPhysics();  // 创建一个带有物理世界的场景
    scene->getPhysicsWorld()->setDebugDrawMask(PhysicsWorld::DEBUGDRAW_ALL);

    scene->getPhysicsWorld()->setGravity(Vec2(0, -1200));//重力设置
    scene->getPhysicsWorld()->setSubsteps(60);  // 增加迭代次数

    auto layer = Level3Scene::create();
    scene->addChild(layer);

    return scene;
}
```

3.3.3.2.2 Create Background Layer

A new sprite is created with
Sprite::create("level/xumu/L1/xumu1_LG5.png"),
which loads an image to be used as the background of the scene. Then background->setAnchorPoint(Vec2(0.5, 0.5)); sets the anchor point of the background sprite to its center. The use of different z-orders ensures that the layers are rendered in the correct order, contributing to a more immersive and visually appealing game environment.

```cpp
// 创建背景层
    auto background = Sprite::create("level/xumu/L1/xumu1_LG5.png");  // 更远的背景
    background->setAnchorPoint(Vec2(0.5, 0.5));
    background->setScale(0.8);
    background->setPosition(Vec2(visibleSize.width / 2, visibleSize.height / 2));
    this->addChild(background, -3);

    auto midground = Sprite::create("level/xumu/L1/xumu1_LG4.png");  // 中间层背景
    midground->setAnchorPoint(Vec2(0.5, 0.5));
    midground->setScale(0.8);
    midground->setPosition(Vec2(visibleSize.width / 2, visibleSize.height / 2));
    this->addChild(midground, -2);
```



background image

3.3.3.2.3 Add characters to the Scene

The create method is used to instantiate a new player character. This method defines the initial state of the player, such as the initial appearance of the character, which may be a sprite frame or animation representing the player in an idle state. The position of the player is then set using

setPosition(Vec2(70, 200))

This will place the player at a specific location in the game world, the specified coordinates (70, 200) define the initial position of the player on the screen or in the game scene.

The player is then added to the scene as a child node using

this->addChild(player, 1).

The second parameter "1" determines the rendering order of the player relative to other elements in the scene.

Finally use

player->getPhysicsBody()->getFirstShape()->setFriction(0.5f)

to set the friction of the player, thus modifying the player's physics body.

This will adjust the way the player character interacts with other physical objects in the game, such as platforms or obstacles, to make movement and collisions more realistic.

```
// 创建玩家
auto player = Player::create(3, "movement/idle/Idle_00/Idle_00-0.png");
player->setPosition(Vec2(70, 200));
this->addChild(player, 1);
player->getPhysicsBody()->getFirstShape()->setFriction(0.5f);
loadLevel();
```

Characters image

3.3.3.2.4 Load the level

The *loadLevel* method is responsible for loading level-specific elements, such as backgrounds, platforms, borders, etc. These elements are usually represented as sprites. These elements are usually added to the scene as sprites, and some of them have a *PhysicsBody* attached to them so that they can interact in the physical world. In this way, the game scene becomes rich and interactive.

```
//创建ice
auto ice = Ice::create(Vec2(880, 498)); // 假设位置
this->addChild(ice); // 添加到场景或其他父节点中
```

Ice image

```
1  //创建spikeweed陷阱
2     auto spikeweed5 = Spikeweed::create(Vec2(1080, 365), Size(10, 60));
3     this->addChild(spikeweed5);
```

Spikeweed image

3.3.3.2.5 Polygon physics shapes

A polygonal physics shape is created and added to a physics body using an array *polygonPoints*, which defines a quadrilateral shape with vertices. This shape is then instantiated with *PhysicsShapePolygon::create* and its restitution is set to zero

polygonShape10->setRestitution(0.0f)

indicating the shape will not bounce upon collision, mimicking a more realistic interaction, like a wall or ground surface. Finally, this non-bouncy polygon shape is added to the physics body

addShape(polygonShape)

ready for use in collision detection and physical interactions within the game's physics world.

```
//// 创建物理体
    auto physicsBody = PhysicsBody::create();

    // 定义多边形顶点并创建多边形形状
    Vec2 polygonPoints1[] = {
Vec2(-622, 143),
Vec2(-480, 143),
Vec2(-480, 8),
Vec2(-622, 8)
    };

    // 创建多边形形状
    auto polygonShape1 = PhysicsShapePolygon::create(polygonPoints1, sizeof(polygonPoints1) / sizeof(polygonPoints1[0])); // 使用顶点数组创建形状
    polygonShape1->setRestitution(0.0f); // 设置反弹系数为0
    physicsBody->addShape(polygonShape1); // 将形状添加到物理体
```

Scene with polygon physics shape

3.3.3.2.6 Game music management

To make the game more complete, we have also added background music to the game.

The *onEnter* and *onExit* methods handle the life cycle of the scene. *onEnter* is called at the beginning of the scene to play the background music. *onExit* is called at the end of the scene to stop the music from playing. These methods ensure that the scene is able to manage resources, such as loading and unloading music, at the appropriate times.

```
void Level3Scene::onEnter() {
    cocos2d::Layer::onEnter();

    // Play background music
    _backgroundMusicID = cocos2d::AudioEngine::play2d("music/gameplayBGM.mp3", true, 1.0f);  // Remember to replace the path with your actual music file path
}

void Level3Scene::onExit() {
    // Stop background music
    if (cocos2d::AudioEngine::getState(_backgroundMusicID) == cocos2d::AudioEngine::AudioState::PLAYING) {
        cocos2d::AudioEngine::stop(_backgroundMusicID);
    }

    cocos2d::Layer::onExit();
}
```

3.3.3.2.7 Level Transition

For example, in the Level6Scene class, the *update()* function is designed to continuously check, every frame, whether the conditions for transitioning to Level7Scene are met. If these conditions are satisfied, as determined by the *checkForLevelTransition()* method, the game immediately transitions to the next level by replacing the current scene with Level7Scene without any visual transition effects, ensuring a smooth and uninterrupted gameplay progression from Level 6 to Level 7.

```
void Level6Scene::update(float dt) {
    if (checkForLevelTransition()) {
        // 切换到 Level7Scene，不使用渐变过渡
        auto scene = Level7Scene::createScene();
        Director::getInstance()->replaceScene(scene);
    }
}
```

3.3.3.2.8 Check for Level Transition

For example, the *checkForLevelTransition* method in Level6Scene utilizes raycasting to determine if a level transition is needed by detecting the player's presence at a specific point. It sets up a ray from a start point to an end point and by calling
Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(...)
which applies the raycast in the current scene's physics world. that checks if the ray intersects with the player character. If the player is detected within this specified area, indicated by *playerDetected* becoming true, the method signals that a level transition, such as moving to Level 7, should occur. This approach allows for a condition-based and seamless progression to the next stage of the game.

```
bool Level6Scene::checkForLevelTransition() {
    // 设置射线的起始点和终点
    Vec2 rayStart = Vec2(800, 750);
    Vec2 rayEnd = Vec2(800, 690); // 这里需要你设置好转换点
    bool playerDetected = false;  // 用于记录是否检测到player

    auto rayCallback = [&playerDetected](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data)->bool {
        auto node = info.shape->getBody()->getNode();
        if (node && node->getName() == "player") {
            // 如果射线检测到Player
            playerDetected = true;  // 记录检测到player
            return false; // 停止射线检测
        }
        return true; // 继续射线检测
        };

    Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, rayStart, rayEnd, nullptr);
    return playerDetected;  // 返回是否检测到player
}
```

3.3.3.3 EndLayer Class

It creates a special layer for the end of a game level or session, featuring a slideshow of images and background music. elegantly handles the end-of-game or level scenario by displaying a sequence of images with background music, followed by a smooth transition back to the main menu.

The background music is played using

cocos2d::AudioEngine::play2d

which plays the specified music file without looping.

The slideshow starts with the first image in the predefined list (*imageFiles*). This image is displayed as a sprite positioned in the center of the *screen.updateSlideShow* is called every 3 seconds to update the slideshow. It increments *currentSlideIndex* and updates the texture of the *slideShow* sprite to the next image in the list, creating a sequence of *images.endSlideShow* is triggered after 15 seconds to end the slideshow.

It transitions to the main menu scene using a fade transition effect. This provides a smooth ending to the slideshow and a transition back to the main game menu.

```
bool EndLayer::init() {
    if (!Layer::init()) {
        return false;
    }

    // 每3秒更新一次图片
    this->schedule([this](float dt) {
        this->updateSlideShow(dt);
        }, 3.0f, "slideShowScheduler");

    // 15秒后结束展示
    this->scheduleOnce([this](float dt) {
        this->endSlideShow(dt);
        }, 15.0f, "endShowScheduler");

    // 播放音乐
    cocos2d::AudioEngine::play2d(musicFile, false);

    // 初始化第一张图片
    currentSlideIndex = 0;
    slideShow = Sprite::create(imageFiles[currentSlideIndex]);
    slideShow->setPosition(Director::getInstance()->getVisibleSize() / 2);
    // 设置图片缩放为0.55
    slideShow->setScale(0.55f);
    this->addChild(slideShow);

    return true;
}
```

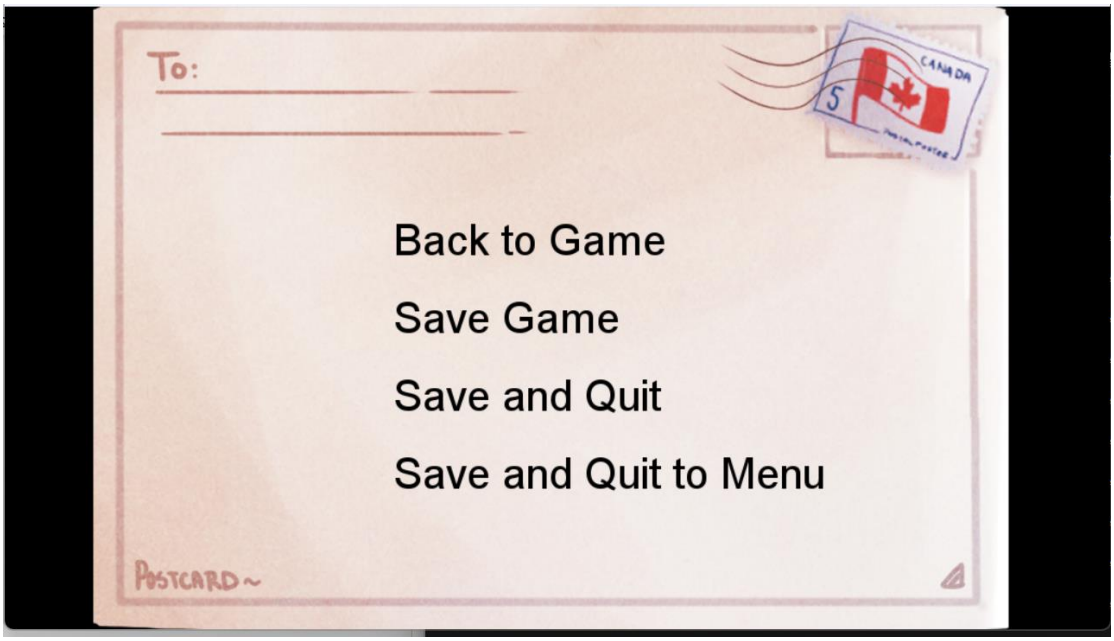
End image

3.3.3.4 Pause Menu Scene in Level

The *initKeyboardListener* method sets up a keyboard event listener using *EventListenerKeyboard*.

This listener is configured to call the *onKeyPressedL6* method when a key is released. Specifically, if the Escape (ESC) key is pressed, *onKeyPressedL6* pauses the current game scene and displays a *PauseMenu* scene. This functionality allows players to pause the game and interact with additional in-game options, effectively enhancing the user experience by providing an accessible and responsive game control mechanism.

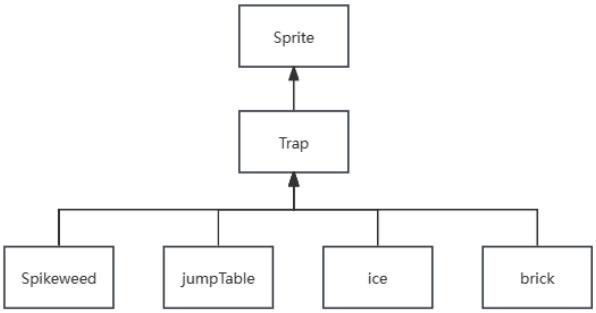
```cpp
void Level6Scene::initKeyboardListener() {
    EventListenerKeyboard* listenerkeyPad = EventListenerKeyboard::create();
    listenerkeyPad->onKeyReleased = CC_CALLBACK_2(Level6Scene::onKeyPressedL6, this);
    _eventDispatcher->addEventListenerWithSceneGraphPriority(listenerkeyPad, this);
}
void Level6Scene::onKeyPressedL6(cocos2d::EventKeyboard::KeyCode keycode, cocos2d::Event* event) {
    if (keycode == EventKeyboard::KeyCode::KEY_ESCAPE) {
        //ESC键
        auto pauseLayer = PauseMenu::create();
        Director::getInstance()->getRunningScene()->pause();
        Director::getInstance()->pushScene(pauseLayer);
    }
}
```



Pause menu

3.3.4 Trap Class (OOP)

3.3.3.1 Trap Class



Inheritance Relationship Diagram in Trap

```h
#pragma once
#ifndef __TRAP_H__
#define __TRAP_H__

#include "cocos2d.h"

class Trap : public cocos2d::Sprite {
public:
    // 使用静态create函数来创建陷阱
    static Trap* create(const std::string& filename);

    // 初始化函数
    virtual bool init() override;

    // 激活或触发陷阱的功能
    virtual void activate() = 0;

    // 检查陷阱是否被激活
    virtual bool isActivated() const;

    // 析构函数
    virtual ~Trap() {}

protected:
    bool _activated; // 表示陷阱是否已被激活
};

#endif // __TRAP_H__
```

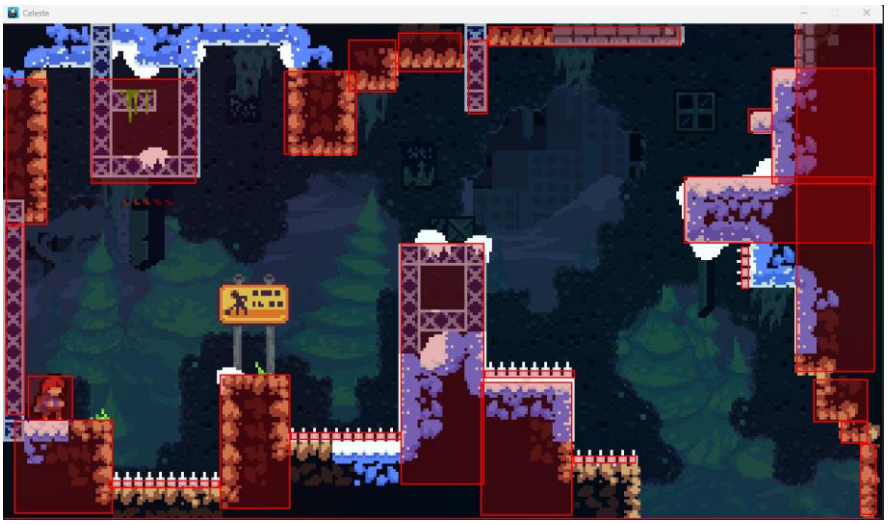
```cpp
#include "Trap.h"

USING_NS_CC;

bool Trap::init() {
    // 在这里执行初始化操作
    if (!Sprite::init()) {
        return false;
    }

    // 设置陷阱的默认属性，如未被激活状态
    _activated = false;

    // 初始化代码，例如设置陷阱的默认图像、动画、物理属性等

    return true;
}

bool Trap::isActivated() const {
    return _activated;
}

// 其他可能的通用陷阱方法...
```

Trap.h and Trap.cpp

The Trap class is a specialized abstract class used to represent traps in the game, publicly inheriting from cocos2d::Sprite. This means that the Trap class not only possesses the basic characteristics of the Sprite class, such as position, scaling, and rotation, but also extends specific functionalities unique to traps.

One of the core functionalities is the _activated private member variable, a boolean type used to indicate whether the trap is activated or not. The Trap class also includes a static creation method '*create*', which takes a file name representing the trap image as an argument and returns an instance of the Trap class. Furthermore, it overrides the *init* method of the base class Sprite for trap-specific initialization settings, such as setting its default state to not activated. The Trap class also contains an abstract method 'activate' to define the behavior when the trap is triggered, and an *isActivated* method to check if the trap is currently activated. Such a design makes the Trap class a flexible and expandable component, capable of implementing various complex and diverse trap functionalities in the game.

3.3.3.2 Spikeweed Class



Spikeweed in scene

Spikeweed class is publicly inheriting from Trap class. In a sense, Spikeweed is not a real trap; it essentially just creates a physical volume named 'Spikeweed'. It takes two parameters in total, one Vec parameter to determine its position in the scene, and another size parameter to determine the size of the Spikeweed. It's worth noting that Spikeweed has no texture; it is drawn according to the position of traps on the background image.

Raycasting Diagram

How does Spikeweed work? Essentially, the Player class is constantly checking using a ray casting method. The Player class has a bool type function *checkForSpikeweedCollision()*, which emits a ray downwards from the player's position.

If it detects a physical body named 'Spikeweed', it calls the function *changeState(PlayerState::DYING)* to change the character's state to dying and returns true; otherwise, it returns false. The *checkForSpikeweedCollision()* function is continuously called in the Player's update function.

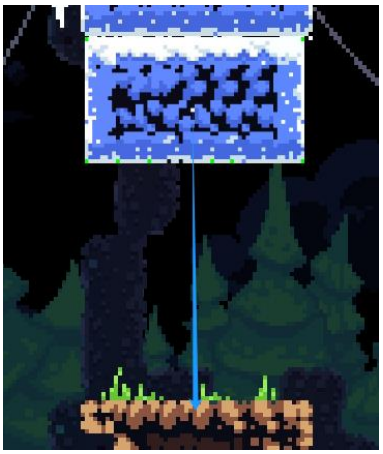
```cpp
bool Player::checkForSpikeweedCollision() {
    float rayLength = 20.0f;  // The same ray length as in adjustMovePosition
    std::vector<cocos2d::Vec2> directions = {
        cocos2d::Vec2(-1, 0),   // Left
        cocos2d::Vec2(1, 0),    // Right
        cocos2d::Vec2(0, -1)    // Bottom
    };

    cocos2d::Vec2 centerPoint = this->getPosition();
    cocos2d::Size playerSize = this->getContentSize();
    cocos2d::Vec2 bottomCenterPoint = centerPoint - cocos2d::Vec2(0, playerSize.height * 0.5f) + cocos2d::Vec2(0, 49.0f);
    cocos2d::Vec2 topCenterPoint = centerPoint + cocos2d::Vec2(0, playerSize.height * 0.5f) - cocos2d::Vec2(0, 49.0f);
    std::vector<cocos2d::Vec2> startPoints = { bottomCenterPoint, centerPoint, topCenterPoint };

    for (const auto& dir : directions) {
        for (const auto& startPoint : startPoints) {
            if (dir.x != 0 && startPoint == centerPoint) continue; // Skip vertical rays for the center point

            cocos2d::Vec2 endPoint = startPoint + dir * rayLength;
            bool collisionDetected = false;

            auto rayCallback = [this, &collisionDetected](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data) -> bool {
                if (info.shape->getBody()->getNode()->getName() == "Spikeweed") {
                    collisionDetected = true;
                    return false;  // Stop detecting further as we found Spikeweed
                }
                return true;
                };

            Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, startPoint, endPoint, nullptr);

            if (collisionDetected) {
                this->changeState(PlayerState::DYING);  // Change state to DYING if Spikeweed is detected
                return true;  // Return true if collision with Spikeweed is detected
            }
        }
    }
    return false;  // Return false if no Spikeweed is detected
}
```

checkForSpikeweedCollision()

3.3.3.3 ice Class

ice in scene

The Ice class publicly inherits from the Trap class and has a trigger function as well as a raycasting detection function. Ice continuously checks whether the ray has touched the player to determine if it is triggered



Raycasting Diagram

Initially, the physical body of ice is set to be dynamic but not affected by gravity, and its mask is set to collide with the player, with the name set as 'ground'. The update function of ice continuously executes *checkForPlayer*(), and when the player is detected, it calls the activate() function. The *activate*() function first assigns true to the ice block's *_activated*, and sets *canActive* to false to prevent the ice block from being activated again. It then plays the sound of the ice block about to fall, and after a delay of 0.5 seconds, changes the name to 'Spikeweed' and sets the ice to be affected by gravity. After the ice

block lands, its name is changed back to 'ground'.

```
void Ice::checkForPlayer() {
    // 这里你需要定义射线的起始点和方向
    Vec2 rayStart = this->getPosition(); // 例如中心点
    Vec2 rayEnd = rayStart - Vec2(0, 1000); // 向下延伸，长度可以调整

    auto rayCallback = [this](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data)->bool {
        auto node = info.shape->getBody()->getNode();
        if (node && node->getName() == "player") {
            // 如果射线检测到Player，激活冰块
            this->activate();
            return false; // 停止射线检测
        }
        return true; // 继续射线检测
        };

    Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, rayStart, rayEnd, nullptr);

}
```

checkForPlayer()

```
void Ice::activate() {
    if (!canActive) return; // 如果已经激活，直接返回

    // 播放摇晃声音
    cocos2d::AudioEngine::play2d("music/game_01_fallingblock_ice_shake_01.mp3", false);
    CCLOG("Ice shake music have been played");

    // 更改状态
    _activated = true;
    canActive = false; // 防止再次激活

    // 设置0.5秒后开始下落
    auto delayForGravity = DelayTime::create(0.7f);
    auto enableGravity = CallFunc::create([this]() {
        auto physicsBody = this->getPhysicsBody();
        if (physicsBody) {
            physicsBody->setGravityEnable(true);
            this->setName("Spikeweed");  // 修改名称以反映新状态
            // 播放冰块碰撞声音
            cocos2d::AudioEngine::play2d("music/game_01_fallingblock_ice_impact_03.mp3", false);
            CCLOG("Ice impact music have been played");
        }
        });

    // 设置2秒后更改名字为ground
    auto delayForChangeName = DelayTime::create(0.6189f);
    auto changeNameToGround = CallFunc::create([this]() {
        this->setName("ground");
        auto physicsBody = this->getPhysicsBody();
        if (physicsBody) {
            physicsBody->setGravityEnable(false);
            physicsBody->setDynamic(false);
            physicsBody->setCategoryBitmask(0x01);
            physicsBody->setCollisionBitmask(0x02); // 可以与分类为0x02的物体发生碰撞
            physicsBody->setContactTestBitmask(0xFFFFFFFF);
        }
        // 如果还需要停止下落或者其他逻辑，可以在这里添加
        });

    this->runAction(Sequence::create(delayForGravity, enableGravity, delayForChangeName, changeNameToGround, nullptr));
}
```

activate()

3.3.3.4 jumpTable Class

jumpTable in scene

JumpTable also publicly inherits from the Trap class. Since JumpTable has animations, the first frame of the animation is used to represent it in the initial texture image. Like Spikeweed, the effects on and the methods of detection for JumpTable(*checkForJumpTableInteraction()*) are also handled by functions within the player. JumpTable is simply set up as a physical body named 'JumpTable' based on the input position and size.

```
bool Player::checkForJumpTableInteraction() {
    float rayLength = 40.0f; // 假设的射线长度，可以根据需要进行调整

    // 只在向下方向检测
    cocos2d::Vec2 direction = cocos2d::Vec2(0, -1); // Down

    // 获取角色的中心点和底部中心点
    cocos2d::Vec2 centerPoint = this->getPosition();
    cocos2d::Size playerSize = this->getContentSize();
    cocos2d::Vec2 bottomCenterPoint = centerPoint - cocos2d::Vec2(0, playerSize.height * 0.5f);

    // 射线的结束点
    cocos2d::Vec2 endPoint = bottomCenterPoint + direction * rayLength;
    bool collisionDetected = false;

    auto rayCallback = [this, &collisionDetected](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data) -> bool {
        if (info.shape->getBody()->getNode()->getName() == "JumpTable") {
            collisionDetected = true;
            // 获取 JumpTable 对象并调用它的播放动画方法
            auto jumpTable = dynamic_cast<JumpTable*>(info.shape->getBody()->getNode());
            if (jumpTable) {
                if(!jumpTable->_canBeActivated){ return false; }
                jumpTable->playAnimation(); // 调用 JumpTable 的 playAnimation 方法
                jumpTable->deactivateTemporarily();//设置时间间隔
                CCLOG("弹簧动画播放");
            }
            return false;  // 停止进一步检测
        }
        return true; // 继续检测
        };

    Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, bottomCenterPoint, endPoint, nullptr);

    return collisionDetected; // 返回是否检测到 JumpTable
}
```

checkForJumpTableInteraction()

```
//弹簧

if (checkForJumpTableInteraction()) {
    this->getPhysicsBody()->setGravityEnable(true);
    //velocity.y = 0;
    this->getPhysicsBody()->applyImpulse(Vec2(0, 100));//使用冲量
}
```

The function in the update function used to achieve the impulse effect

It's worth noting that since the spring gives the player an impulse rather than a force, there is a displacement in the y-direction between the texture position of the spring and its collision volume. This allows for a more realistic depiction of the effect of the spring being compressed.

3.3.3.5 brick Class

brick in scene

The Brick class also publicly inherits from the Trap class. Initially, Brick sets its position through input parameters and sets itself as a texture. In the *toggleVisibility()* function, it hides its own collision volume and texture in turn to achieve the effect of the brick disappearing. The *toggleVisibility()* function finally uses a sequence to call a two-second delay and then *resetBrick()*, making the collision volume and texture reappear.

```
void Brick::toggleVisibility() {
    if (_isNormal) {
        // 如果当前是正常状态，则在2秒后隐藏碰撞体积和纹理
        auto delay = DelayTime::create(2.0f);
        auto hideBrick = CallFunc::create([this]() {
            this->setVisible(false);
            if (this->getPhysicsBody()) {
                this->getPhysicsBody()->removeFromWorld();
            }
            _isNormal = false;
        });

        auto resetDelay = DelayTime::create(2.0f);  // 等待2秒以重置砖块
        auto resetBrick = CallFunc::create([this]() {
            this->resetBrick();  // 重置砖块为可见和有物理体
        });

        // 运行动作序列
        this->runAction(Sequence::create(delay, hideBrick, resetDelay, resetBrick, nullptr));
    }
}
```

toggleVisibility()

```cpp
void Brick::resetBrick() {
    // 重新设置brick为可见，并恢复其物理属性
    this->setVisible(true);
    if (this->getPhysicsBody()) {
        // 假设有一种方式来重新添加物理体或恢复它的状态
        // 可能需要保存并恢复原始的物理设置
        auto physicsBody = PhysicsBody::createBox(Size(120, 35));
        physicsBody->setDynamic(false);
        // ... 其他物理属性设置
        this->setPhysicsBody(physicsBody);
    }
    _isNormal = true;
}
```

resetBrick()

Similarly, the detection of bricks and the calling of the *toggleVisibility()* function are also handled by the player's *checkForBrickInteraction()* function.

```cpp
bool Player::checkForBrickInteraction() {
    float rayLength = 15.0f; // 假设的射线长度，可以根据需要进行调整

    // 只在向下方向检测
    cocos2d::Vec2 direction = cocos2d::Vec2(0, -1); // Down

    // 获取角色的中心点和底部中心点
    cocos2d::Vec2 centerPoint = this->getPosition();
    cocos2d::Size playerSize = this->getContentSize();
    cocos2d::Vec2 bottomCenterPoint = centerPoint - cocos2d::Vec2(0, playerSize.height * 0.5f);

    // 射线的结束点
    cocos2d::Vec2 endPoint = bottomCenterPoint + direction * rayLength;
    bool collisionDetected = false;

    auto rayCallback = [this, &collisionDetected](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data) -> bool {
        if (info.shape->getBody()->getNode()->getName() == "brick") {
            collisionDetected = true;
            // 获取 brick 对象并调用它的播放动画方法
            auto brick = dynamic_cast<Brick*>(info.shape->getBody()->getNode());
            if (brick) {
                if (!brick->_isNormal) { return false; }
                brick->toggleVisibility();
                CCLOG("brick can not toach");
            }
            return false;  // 停止进一步检测
        }
        return true; // 继续检测
        };

    Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, bottomCenterPoint, endPoint, nullptr);

    return collisionDetected; // 返回是否检测到 JumpTable
}
```

checkForBrickInteraction()

3.3.3.6 OOP

Using different Trap classes to implement Object-Oriented Programming (OOP) offers several benefits:

Modularity: Each Trap class represents a specific trap, making the code more modular, easier to understand, and maintain. Each class is responsible for its own behavior and properties, reducing code complexity.

Extensibility: Through inheritance and polymorphism, it's easy to add new types of traps without modifying existing code. This enhances the game's ease of expansion and updates.

Reusability: Different types of traps can be reused in various levels or scenarios, saving development time and resources.

Maintainability: Each Trap class has its own functionality and properties. If adjustments or fixes are needed for a specific type of trap, they can be made within that class without affecting other parts of the code.

Clarity: Using different classes to represent various trap types makes the code clearer, more readable, and easier to comprehend. Each class serves a unique purpose and behavior.

In summary, employing different Trap classes to implement OOP enhances code maintainability, extensibility, and readability, making game development more efficient and flexible.

```
//陷阱测试
//创建spikeweed陷阱
auto spikeweed = Spikeweed::create(Vec2(400, 120), Size(100, 10));
this->addChild(spikeweed);

//创建jumpTable

// 获取第一帧的SpriteFrame
auto frame = SpriteFrameCache::getInstance()->getSpriteFrameByName("JumpTable_00-0.png");

// 检查是否成功获取到
if (frame) {
    // 设置JumpTable的纹理为获取到的帧

    //不能自己设置分割的物理体积的位置

    auto jumpTableSprite = JumpTable::create(Vec2(700, 140)); // 假设你有这样一个方法来创建JumpTable实例
    jumpTableSprite->setSpriteFrame(frame); // 使用第一帧作为纹理
    auto physicsBody = PhysicsBody::createBox(Size(100,10));// 设置物理形状
    physicsBody->setPositionOffset(Vec2(0, -100));// 设置物理体的位置偏移
    physicsBody->setDynamic(false);

    jumpTableSprite->setPhysicsBody(physicsBody);

    this->addChild(jumpTableSprite);
}
else {
    // 如果没有找到帧，可能需要输出错误或采取其他行动
    CCLOG("Error: Cannot find the first frame of JumpTable in SpriteFrameCache");
}

//创建brick
auto brick = Brick::create(Vec2(100, 200));
this->addChild(brick); // 将brick添加到场景
//创建ice
auto ice = Ice::create(Vec2(700, 400)); // 假设位置
this->addChild(ice); // 添加到场景或其他父节点中
```

Trap test in Level1

For example, in the early stages of game development, I can test each trap in my first level. Later on, I only need to change the parameters passed to different types of traps to complete the design of subsequent levels

```
//创建brick1
auto brick1 = Brick::create(Vec2(975+1, 103));
this->addChild(brick1); // 将brick添加到场景
//创建brick2
auto brick2 = Brick::create(Vec2(1137-1, 193));
this->addChild(brick2); // 将brick添加到场景
//创建brick3
auto brick3 = Brick::create(Vec2(945-1, 290));
this->addChild(brick3); // 将brick添加到场景
//创建brick4
auto brick4 = Brick::create(Vec2(1008, 452));
this->addChild(brick4); // 将brick添加到场景
```

The addition of four bricks at different positions in the Level6

For traps of the same type but different positions and sizes, I only need to change the parameters passed in, greatly enhancing code reusability.

3.3.5 Player Class

3.3.5.1 Keyboard Monitoring

First, I defined an enumeration class named *PlayerKey*, which contains all the key names I need in my game (the talk key is not used in the final game). Then, I used Cocos2d-x's *EventListenerKeyboard* to listen for keyboard input. Whenever a key is pressed or released, the corresponding callback functions *onKeyPressed* and *onKeyReleased* are triggered. These functions update the *keyStates* dictionary, which maintains the current state (pressed or not pressed) of each key (such as left, right, jump, etc.).

```
enum class PlayerKey {
    LEFT,
    RIGHT,
    UP,
    DOWN,
    JUMP,
    TALK,
    DASH,
    CLIMB
};
```

```
std::map<PlayerKey, bool> keyStates;//检测用户输入

bool Player::checkPlayerInput();//检测用户是否有按键输入

cocos2d::Vec2 Player::adjustMovePosition(const cocos2d::Vec2& desiredPosition);//完善位置检测

vate:
cocos2d::DrawNode* _debugDrawNode;//测试用射线
bool isOnSolidGround(); // 检查玩家是否在坚实的地面上

void changeState(PlayerState newState);//状态转换

cocos2d::EventListenerKeyboard* keyboardListener; // 键盘监听器

// 键盘事件回调函数
void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event* event);
void onKeyReleased(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event* event);
```

PlayerKey defined, onKeyPressed, onKeyReleased, EventListenerKeyboard function

In the Player.cpp file, the key states are initialized to false in the player's constructor, and a keyboard listener is added in the constructor. When the keyboard is pressed, the *onKeyPressed* and *onKeyReleased* functions change the state of the corresponding keys to true and false respectively, using a switch statement.

```
Player::Player()
    : keyStates{
        {PlayerKey::LEFT, false},
        {PlayerKey::RIGHT, false},
        {PlayerKey::UP, false},
        {PlayerKey::DOWN, false},
        {PlayerKey::JUMP, false},
        {PlayerKey::TALK, false},
        {PlayerKey::DASH, false},
        {PlayerKey::CLIMB, false}
    }
{

    auto listener = EventListenerKeyboard::create();
    listener->onKeyPressed = CC_CALLBACK_2(Player::onKeyPressed, this);
    listener->onKeyReleased = CC_CALLBACK_2(Player::onKeyReleased, this);
    _eventDispatcher->addEventListenerWithSceneGraphPriority(listener, this);


}
```

Player's constructor

```cpp
void Player::onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event* event) {
    switch (keyCode) {
    case EventKeyboard::KeyCode::KEY_LEFT_ARROW:
        keyStates[PlayerKey::LEFT] = true;
        break;
    case EventKeyboard::KeyCode::KEY_RIGHT_ARROW:
        keyStates[PlayerKey::RIGHT] = true;
        break;
    case EventKeyboard::KeyCode::KEY_UP_ARROW:
        keyStates[PlayerKey::UP] = true;
        break;
    case EventKeyboard::KeyCode::KEY_DOWN_ARROW:
        keyStates[PlayerKey::DOWN] = true;
        break;
        // 假设使用空格键代表跳跃
    case EventKeyboard::KeyCode::KEY_SPACE:
        //CCLOG("press space");
        keyStates[PlayerKey::JUMP] = true;
        break;
        // 假设使用J键代表交谈
    case EventKeyboard::KeyCode::KEY_J:
        keyStates[PlayerKey::TALK] = true;
        break;
        // 假设使用SHIFT键代表冲刺
    case EventKeyboard::KeyCode::KEY_SHIFT:
        keyStates[PlayerKey::DASH] = true;
        break;
        // 假设使用K键代表攀爬
    case EventKeyboard::KeyCode::KEY_K:
        keyStates[PlayerKey::CLIMB] = true;
        break;
    default:
        break;
    }
}
```

onKeyPressed

```cpp
void Player::onKeyReleased(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event* event) {
    switch (keyCode) {
    case EventKeyboard::KeyCode::KEY_LEFT_ARROW:
        keyStates[PlayerKey::LEFT] = false;
        break;
    case EventKeyboard::KeyCode::KEY_RIGHT_ARROW:
        keyStates[PlayerKey::RIGHT] = false;
        break;
    case EventKeyboard::KeyCode::KEY_UP_ARROW:
        keyStates[PlayerKey::UP] = false;
        break;
    case EventKeyboard::KeyCode::KEY_DOWN_ARROW:
        keyStates[PlayerKey::DOWN] = false;
        break;
    case EventKeyboard::KeyCode::KEY_SPACE:
        //CCLOG("release space");
        keyStates[PlayerKey::JUMP] = false;
        isJumping = false; // 当按键释放时，允许下一次跳跃
        break;
    case EventKeyboard::KeyCode::KEY_J:
        keyStates[PlayerKey::TALK] = false;
        break;
    case EventKeyboard::KeyCode::KEY_SHIFT:
        keyStates[PlayerKey::DASH] = false;
        break;
    case EventKeyboard::KeyCode::KEY_K:
        keyStates[PlayerKey::CLIMB] = false;
        CCLOG("KEY_K RELESE");
        break;
    default:
        break;
    }
}
```

onkeyReleased

3.3.5.2 State Machine

To control different animations for the player's complex actions, I implemented a simple state machine for the transitions between different states of the player. First, I defined an enumeration class named *PlayerState*, which includes the possible states of the player, such as IDLE, MOVING_LEFT, JUMPING, etc. The core of the state machine is the *changeState* method, which updates the current state based on the player's actions and environment (such as whether they are on the ground), and triggers the corresponding animations and sound effects. Since the specific transitions are too tedious and complex, a deeper understanding can be achieved by reading the *update()* code.

```
enum class PlayerState {
    IDLE,               // 站立
    MOVING_LEFT,        // 左移
    MOVING_RIGHT,       // 右移
    MOVING_TURN_RL,     // 从右向左转向
    MOVING_TURN_LR,     // 从左向右转向
    CROUCH,             // 下蹲
    LOOKUP,             // 向上看
    JUMPING,            // 跳跃
    LANDING,            // 落地
    DROP,               // 坠落
    PUSHWALL,           // 推墙
    HOLDWALL,           // 爬墙
    HOLDWALLUP,         // 爬墙向上
    HOLDWALLDOWN,       // 爬墙向下
    HOLDWALLJUMP,      // 爬墙跳跃
    DASH,               // 冲刺
    DYING,              // 死亡

};
```

PlayerState

```
void Player::changeState(PlayerState newState) {
    if (currentState == newState) return;
    previousState = currentState;
    currentState = newState;
    //音频控制
    if (previousState== PlayerState::MOVING_RIGHT|| previousState == PlayerState::MOVING_LEFT) {
        cocos2d::AudioEngine::stop(_walkMusicId);
    }
    //状态控制
    switch (currentState) {
    case PlayerState::CROUCH:
        playCrouchAnimation();
        break;
    case PlayerState::LOOKUP:
        playLookUpAnimation();
        break;
    case PlayerState::IDLE:
        if (previousState == PlayerState::CROUCH)
        {
            playCrouchToIdleAnimation();//过度动画（在动画调用完成后更新状态）
        }
        else {
            playIdleAnimation_1();//默认状态
        }
        break;
    case PlayerState::MOVING_LEFT:
        playMoveAnimation();
        this->setScaleX(-1.0f); // 镜像动画以表示左走
        facingDirection = -1;//面向左边
        break;
    case PlayerState::MOVING_RIGHT:
        playMoveAnimation();
        this->setScaleX(1.0f);  // 正常动画表示右走
        facingDirection = 1;//面向右边
        break;
    case PlayerState::MOVING_TURN_RL:
        playMoveTurnAnimation();
        this->setScaleX(-1.0f); // 镜像动画以表示左向右走
        break;
    case PlayerState::MOVING_TURN_LR:
        playMoveTurnAnimation();
        this->setScaleX(1.0f);  // 正常动画表示从右向左走
        break;
    case PlayerState::JUMPING:
        if (canDash) {
            playJumpUpAnimation();
        }
        else {
            playBJumpUpAnimation();
        }
        break;
    case PlayerState::DROP:
        if (canDash) {
            playDropAnimation();
        }
        else {
            playBDropAnimation();
        }
        break;
    case PlayerState::PUSHWALL:
        playPushWallAnimation();
        break;
    case PlayerState::LANDING:
        playLandingAnimation();
        break;
    case PlayerState::HOLDWALL:
        if (canDash) {
            playHoldWallAnimation();
        }
        else {
            playBHoldWallAnimation();
        }
        break;
    case PlayerState::HOLDWALLUP:
        if (canDash) {
            playHoldWallUpAnimation();
        }
        else {
            playBHoldWallUpAnimation();
        }
        break;
    case PlayerState::HOLDWALLDOWN:
        if (canDash) {
            playHoldWallDownAnimation();
        }
        else {
            playBHoldWallDownAnimation();
        }
        break;
    case PlayerState::HOLDWALLJUMP:
        if (canDash) {
            playHoldWallJumpAnimation();
        }
        else {
            playBHoldWallJumpAnimation();
        }
        break;
    case PlayerState::DASH:
        this->setScaleX(facingDirection);
        if (getDashDirection() == cocos2d::Vec2(0, 1) || getDashDirection() == cocos2d::Vec2(0,
        {
            playDashUpAndDownAnimation();
        }
        else {
            playDashAnimation();

        }
        break;
    case PlayerState::DYING:
        if (canDash) {
            playDeathAnimation();
        }
        else {
            playBDeathAnimation();
        }
        break;
        // 处理其他状态...
    }
```

changeState

3.3.5.3 Animation System

The animation system is primarily implemented through Animation and SpriteFrame. Each action (such as walking, jumping) has a corresponding animation composed of a series of frames. The play series functions, such as *playIdleAnimation_1* and *playJumpUpAnimation*, are responsible for initiating the animation of specific actions. These functions create an Animation object by loading a series of SpriteFrames, and then apply it to the player sprite.

```
void Player::playIdleAnimation_1() { // ...
    this->stopAllActions();
    //CCLOG("Starting IDLE animation");
    Vector<SpriteFrame*> idleFrames;
    auto cache = SpriteFrameCache::getInstance();

    for (int i = 0; i <= 6; i++) {
        std::string frameName = StringUtils::format("Idle_00-%d.png", i);
        auto frame = cache->getSpriteFrameByName(frameName);
        if (frame) {
            idleFrames.pushBack(frame);
        }
    }

    auto animation = Animation::createWithSpriteFrames(idleFrames, 0.2f);
    auto animate = Animate::create(animation);
    // 使用 RepeatForever 动作使动画无限循环播放
    auto repeatForever = RepeatForever::create(animate);

    this->runAction(repeatForever); // 使用 repeatForever 运行动画
   // CCLOG("Finished setting up IDLE animation");
}
```

playIdleAnimation_1()

The animation system can be further subdivided into action animations (including normal animations and animations after the character enters a dash state), transition animations (used to connect two animations such as turning and landing), transition effects (like a blackout transition effect), and special effects (such as the smoke effects when landing and jumping).

```
// 动画播放
void playIdleAnimation_1(); // 播放站立动画1
void playIdleAnimation_2(); // 播放站立动画2
void playMoveAnimation();//移动动画
void playCrouchAnimation();//蹲姿动画
void playLookUpAnimation();//向上看动画
void playJumpUpAnimation();//跳跃动画
void playJumpMoveAnimation();//跳跃移动动画
void playDropAnimation();//坠落动画
void playPushWallAnimation();//推墙动画
void playHoldWallAnimation();//爬墙
void playHoldWallUpAnimation();//爬墙向上
void playHoldWallDownAnimation();//爬墙向下
void playHoldWallJumpAnimation();//爬墙跳跃
void playDashAnimation();//冲刺动画
void playDashUpAndDownAnimation();//冲刺向上向下的特殊动画
void playDeathAnimation();//死亡
void playRespawnAnimation();//重生
//B动作
void playBDeathAnimation();//重生
void playBDropAnimation();//坠落动画
void playBHoldWallAnimation();//爬墙
void playBHoldWallUpAnimation();//爬墙向上
void playBHoldWallDownAnimation();//爬墙向下
void playBHoldWallJumpAnimation();//爬墙跳跃
void playBJumpUpAnimation();//跳跃动画
void playBJumpMoveAnimation();//跳跃移动动画
//过渡动画
void playMoveTurnAnimation();//转向动画
void playCrouchToIdleAnimation();//蹲姿到静止
void playLandingAnimation();//落地动画
//转场动画
void playBlackAnimation();//黑幕过渡
//特效
void playFloorLandingAshAnimation();//落地烟雾
void playFloorJumpAshAnimation();//跳跃烟雾
void playFloorWallJumpAshAnimation();//墙壁跳跃烟雾
void playFloorSlidingWallAshAnimation();//滑墙烟雾

void playDashUpEffAnimation();//
void playDashMoveUpEffAshAnimation();//
void playDashMoveEffAnimation();//
void playDashMoveDownEffAnimation();//
void playDashDownEffAnimation();//
```

Animation function in Palyer.h

3.3.5.4 Audio System

```
//音频
int _dashMusicId;
cocos2d::AudioEngine::AudioState _dashMusicState;
int _walkMusicId;
cocos2d::AudioEngine::AudioState _walkMusicState;
int _jumpMusicId;
cocos2d::AudioEngine::AudioState _jumpMusicState;
int _deathMusicId;
cocos2d::AudioEngine::AudioState _deathMusicState;
int _landingMusicId;
cocos2d::AudioEngine::AudioState _landingMusicState;
int _reviveMusicId;
cocos2d::AudioEngine::AudioState _reviveMusicState;
```

Audio define in Palyer.h

The audio system in the Player class uses Cocos2d-x's AudioEngine to handle sound effect playback. Corresponding audio files have been defined for various actions (such as jumping, dashing, landing). Audio playback is typically triggered when a state changes or a specific action occurs, for example, playing the jump sound in the *playJumpUpAnimation* method. Audio IDs and states (like *_jumpMusicId* and *_jumpMusicState*) are used to manage audio playback and status checks.

```
void Player::playJumpUpAnimation() {//跳跃

    // 检查音乐的状态
    _jumpMusicState = cocos2d::AudioEngine::getState(_jumpMusicId);

    // 如果音乐没有播放或者播放已经完成，那么开始播放音乐(mp3格式)
    if (_jumpMusicState != cocos2d::AudioEngine::AudioState::PLAYING) {
        _jumpMusicId = cocos2d::AudioEngine::play2d("music/jump.mp3", false);
    }
    // 停止所有正在运行的动画（确保不会与其他动画冲突）

    CCLOG("Starting JumpUp animation");
    this->stopAllActions();
    Vector<SpriteFrame*> idleFrames;
    auto cache = SpriteFrameCache::getInstance();

    for (int i = 0; i <= 3; i++) {
        std::string frameName = StringUtils::format("jumpup_00-%d.png", i);
        auto frame = cache->getSpriteFrameByName(frameName);
        if (frame) {
            idleFrames.pushBack(frame);
        }
    }

    for (int i = 0; i <= 1; i++) {
        std::string frameName = StringUtils::format("Top_00-%d.png", i);
        auto frame = cache->getSpriteFrameByName(frameName);
        if (frame) {
            idleFrames.pushBack(frame);
        }
    }
    playFloorJumpAshAnimation();
    auto animation = Animation::createWithSpriteFrames(idleFrames, 0.1f);
    auto animate = Animate::create(animation);
    this->runAction(animate);
     CCLOG("Finished setting up JumpUp animation");

}
```

playJumpUpAnimation()

3.3.5.5. Implementation of Key Functions

3.3.5.5.1 isOnSolidGround()

```
bool Player::isOnSolidGround() {//优化后判定
    float rayLength = 1.0f; // 射线长度固定为1像素
    Vec2 centerPoint = this->getPosition();

    // 获取角色的大小
    cocos2d::Size playerSize = this->getContentSize();

    // 定义射线的起始点
    Vec2 leftStartPoint = centerPoint - Vec2(playerSize.width * 0.5f - 42, playerSize.height * 0.5f - 47);
    Vec2 middleStartPoint = centerPoint - Vec2(0, playerSize.height * 0.5f - 47);
    Vec2 rightStartPoint = centerPoint + Vec2(playerSize.width * 0.5f - 42, -playerSize.height * 0.5f + 47);

    std::vector<cocos2d::Vec2> startPoints = { leftStartPoint, middleStartPoint, rightStartPoint };

    bool onSolidGround = false; // 默认情况下假设玩家不在地面上

    for (const auto& startPoint : startPoints) {
        cocos2d::Vec2 endPoint = startPoint - cocos2d::Vec2(0, rayLength); // 从startPoint向下延长1像素

        auto func = [&onSolidGround](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data) -> bool {
            if (info.shape->getBody()->getNode()->getName() == "ground"|| info.shape->getBody()->getNode()->getName() == "brick") {
                onSolidGround = true;
                return false; // 停止射线检测
            }
            return true; // 继续射线检测
            };

        Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(func, startPoint, endPoint, nullptr);

        if (onSolidGround) {
            break; // 如果检测到玩家在地面上，跳出循环
        }
    }

    return onSolidGround;
}
```

isOnSolidGround()

The *isOnSolidGround* function is responsible for detecting whether the player character is standing on solid ground. This function operates by creating three points at the bottom of the player character (left, center, right) and then shooting rays downward from these points for detection. Each ray is only 1 pixel long to check the surface directly below the player.

These rays check for the first collision object they encounter on their path. If a ray collides with an object named “ground” or “brick”, it is considered that the player is standing on solid ground. The function uses a loop to traverse these three rays, and as soon as any one of the rays detects solid ground, it immediately returns a true value, indicating that the player is indeed on the ground. This method effectively handles situations where the player is partially suspended or fully standing on a platform, ensuring the realism and accuracy of the game's physics.

Raycasting Diagram

3.3.5.5.2 adjustMovePosition()

```
cocos2d::Vec2 Player::adjustMovePosition(const cocos2d::Vec2& desiredPosition) {//此函数用于优化位置设置防止穿模，也用于判定canClimb
    cocos2d::Vec2 adjustedPosition = desiredPosition;                          //相比于之前的单射线（中心位置），优化成双射线（最上，最下）
    float rayLength = 40.0f;

    std::vector<cocos2d::Vec2> directions = {
        cocos2d::Vec2(-1, 0),
        cocos2d::Vec2(1, 0),
        cocos2d::Vec2(0, 1)
    };
    // 中心点
    cocos2d::Vec2 centerPoint = this->getPosition();
    // 获取角色的大小
    cocos2d::Size playerSize = this->getContentSize();
    // 定义射线的起点
    cocos2d::Vec2 bottomCenterPoint = centerPoint - cocos2d::Vec2(0, playerSize.height * 0.5f) + cocos2d::Vec2(0, 49.0f);
    cocos2d::Vec2 topCenterPoint = centerPoint + cocos2d::Vec2(0, playerSize.height * 0.5f) - cocos2d::Vec2(0, 49.0f);
    // 将三个起始点添加到一个列表中
    std::vector<cocos2d::Vec2> startPoints = { bottomCenterPoint, centerPoint, topCenterPoint };
    for (const auto& dir : directions) {
        bool collisionDetected = false;
        std::string collidedObjectName = "";

        for (const auto& startPoint : startPoints) {
            if (dir.x != 0 && startPoint == centerPoint) continue; // 为中心点跳过垂直方向的射线

            cocos2d::Vec2 endPoint = startPoint + dir * rayLength;

            auto rayCallback = [&collisionDetected, &collidedObjectName](PhysicsWorld& world, const PhysicsRayCastInfo& info, void* data) -> bool {
                if (info.shape->getBody()->getNode()->getName() != "player") {
                    collisionDetected = true;
                    collidedObjectName = info.shape->getBody()->getNode()->getName();
                    return false;
                }
                return true;
                };
            Director::getInstance()->getRunningScene()->getPhysicsWorld()->rayCast(rayCallback, startPoint, endPoint, nullptr);
            // 如果检测到碰撞，跳出内部循环
            if (collisionDetected) break;
        }
        if (collisionDetected) {
            if (dir == cocos2d::Vec2(-1, 0) && desiredPosition.x < this->getPositionX()) {
                adjustedPosition.x = this->getPositionX();
                if (facingDirection == -1 && collidedObjectName == "ground") {
                    canClimb = 1;
                }
            }
            else if (dir == cocos2d::Vec2(1, 0) && desiredPosition.x > this->getPositionX()) {
                adjustedPosition.x = this->getPositionX();
                if (facingDirection == 1 && collidedObjectName == "ground") {
                    canClimb = 1;
                }
            }
            else if (dir == cocos2d::Vec2(0, 1)) {
                adjustedPosition.y = this->getPositionY();
            }
        }
        else {
            if ((dir == cocos2d::Vec2(-1, 0) && facingDirection == -1) || (dir == cocos2d::Vec2(1, 0) && facingDirection == 1)) {
                canClimb = 0;
            }
        }
    }
    //CCLOG("canClimb: %d", canClimb);
    return adjustedPosition;
}
```

adjustMovePosition ()

The main purpose of the *adjustMovePosition* function is to adjust the character's movement position to avoid clipping through models (a phenomenon known as "clipping"), and it is also used to determine whether the character can climb (*canClimb*). This function receives a desired position as input and returns an adjusted position.

In its specific implementation, the function first creates three ray start points, located at the character's bottom center, top center, and the center of the character. Then, for horizontal (left, right) and vertical (upward) directions, it emits rays to detect collisions on the path. For each direction, if a collision with a non-player object is detected, the function prevents the character from entering the space occupied by that object and adjusts the character's position accordingly. Specifically, if the horizontal collision object facing the character is identified as “ground”, the character is marked

as able to climb (*canClimb* = 1). If no collision is detected or the collision object is not "ground", *canClimb* is set to 0, indicating that the character cannot climb.

This method allows the character to move and climb smoothly while maintaining physical realism, enhancing the playability and realism of the game.

3.3.5.5.3 update()

```
void Player::update(float dt) {
    if (!isAlive) { return; }//角色死亡直接返回
    if (isAlive) { if (checkForSpikeweedCollision()) {} }
    float deathThreshold = 35; // 你想要的死亡阈值
    if (this->getPositionY() < deathThreshold) {//检查角色高度
        // 进入死亡状态
        isAlive = 0;
        changeState(PlayerState::DYING);
    }
    if (isDashing) {
        dashTimer += dt;
        if (dashTimer >= DASH_DURATION) {
            isDashing = false;
            dashTimer = 0.0f;
        }
        return;  // 如果玩家正在冲刺，跳过所有其他的状态更新
    }
    if (wallJumpCooldown > 0) {
        wallJumpCooldown -= dt;
    }

   // CCLOG("canDash:%d",canDash);
    if ( previousState == PlayerState::DASH) {
        this->getPhysicsBody()->setGravityEnable(true);
    }
    if (canClimb == 0 && (currentState == PlayerState::HOLDWALL || currentState == PlayerState::HOLDWALLUP || currentState == PlayerState::
        this->getPhysicsBody()->setGravityEnable(true);
        velocity.y = -1;
        changeState(PlayerState::DROP);
    }

    if((previousState==PlayerState::HOLDWALL ||previousState == PlayerState::HOLDWALLUP || previousState == PlayerState::HOLDWALLDOWN)&&(cu
    {
        velocity.y = -1;
        this->getPhysicsBody()->setGravityEnable(true);
    }

    float verticalVelocity = this->getPhysicsBody()->getVelocity().y;//物理引擎中的vy
    float horizontalVelocity = this->getPhysicsBody()->getVelocity().x;//物理引擎中的vx


    //新方法
    Vec2 desiredPosition = this->getPosition() + velocity * dt;
    Vec2 adjustedPosition = adjustMovePosition(desiredPosition);

    this->setPosition(adjustedPosition);
```

```
//检查玩家是否在坚实的地面上
bool onGround = isOnSolidGround();
setOnGround(onGround);
if (onGround) {
    canDash = 1;
}


/* ... */
// ...


//蹲姿
if (keyStates[PlayerKey::DOWN] && isOnGround&&!keyStates[PlayerKey::DASH] && !keyStates[PlayerKey::CLIMB]&&velocity.x==0){
    changeState(PlayerState::CROUCH);
    CCLOG("Player state changed to CROUCH");
    return;
}
//向上看
if (keyStates[PlayerKey::UP] && isOnGround && !keyStates[PlayerKey::DASH]&&!keyStates[PlayerKey::DOWN]&&!keyStates[PlayerKey::CLIMB] && velocity.x == 0) {
    changeState(PlayerState::LOOKUP);
    CCLOG("Player state changed to LOOKUP");
    return;
}

//冲刺(默认)
if (keyStates[PlayerKey::DASH] && canDash  ) {
    this->getPhysicsBody()->setGravityEnable(false);
    canDash = 0;
    velocity.x = 0; velocity.y = 0;
    this->getPhysicsBody()->setVelocity(Vec2(0, 0));
    changeState(PlayerState::DASH);
    CCLOG("Player state changed to DASH");
    return;
}


//落地
if ((previousState == PlayerState::DROP && isOnGround)||currentState== PlayerState::LANDING) {
    changeState(PlayerState::LANDING);
    //CCLOG("Player state changed to LANDING");
    return;
}
//推墙
if (((keyStates[PlayerKey::RIGHT] && velocity.x > 0) || (keyStates[PlayerKey::LEFT] && velocity.x < 0)) && isOnGround && canClimb &&!keyStates[PlayerKey::CLIMB]) {
    changeState(PlayerState::PUSHWALL);
    CCLOG("Player state changed to PUSHWALL");
    return;
}
//下落

if (!isOnGround && verticalVelocity < 0) {
    changeState(PlayerState::DROP);
    return;
}
```

```
    //状态转换
    if (isOnGround) {
        if (velocity.x == 0) {
            changeState(PlayerState::IDLE);
            //CCLOG("Player state changed to IDLE");
            return;
        }
        else if (velocity.x > 0) {
            if (keyStates[PlayerKey::RIGHT]) {
                changeState(PlayerState::MOVING_RIGHT);
                facingDirection = 1;
                return;
            }
            else if(keyStates[PlayerKey::LEFT]){
                changeState(PlayerState::MOVING_TURN_RL);
                facingDirection = -1;
                return;
            }
        }
        else if (velocity.x < 0) {
            if (keyStates[PlayerKey::LEFT]) {
                changeState(PlayerState::MOVING_LEFT);
                facingDirection = -1;
                return;
            }
            else if (keyStates[PlayerKey::RIGHT]) {
                changeState(PlayerState::MOVING_TURN_LR);
                facingDirection = 1;
                return;
            }
        }
    }



    // ... 其他代码 ...
}
```

update ()

The *update* function plays a crucial role, integrating the core logic of character control and environmental interaction.

(1)Survival State Check

The function first determines if the player character is alive. If the player has died, the function immediately returns and performs no further operations.

Moreover, the function checks if the player's position is below a set death threshold. If so, the character's status is updated to dead.

(2)Action and State Management

In handling the character's actions, the function checks if the character is performing specific actions, such as dashing or wall jumping, which affect the character's physical state and behavior.

For the character's regular movement, the function calculates the expected direction and speed of movement based on player input and adjusts the position through the adjustMovePosition function to avoid clipping and maintain physical accuracy.

(3)State Machine Logic:

The update function is essentially a complex state machine that changes the

character's state based on the current environment and player input, such as from standing to jumping, from moving to crouching, etc.

This state machine also manages interactions between the character and the environment, such as collisions with traps and interactions with springs.

(4)Physics and Environmental Interaction:

The function manages interactions between the character and the environment through a physics engine, ensuring that the character's movements comply with physical laws.

This includes handling collision detection between the character and the ground, implementing wall jumps, and other physics-based actions. The update function is central to character control and environmental interaction in "Celeste". It not only handles basic actions of the character, such as moving and jumping, but also manages complex state transitions and physical interactions, ensuring smooth and realistic gameplay. Through this function, the game provides players with a responsive and challenging environment.

4. Programming Progress

Phases of Mission	Period	Planned Completion	Actual Completion
Confirm the theme and prepare materials.	2023.9.1-2023.10.1	Confirm the theme and prepare materials.	Confirm “celeste”, get the materials
Decide on the game engine to use and write a proposal.	2023.10.2-2023.10.7	Cocos2dx 4.0	Cocos2dx 4.0
Start building the framework, create the main interface, and develop the player class.	2023.10.8-2023.10.14	Player	Player
Refine the player class and create test scenarios.	2023.10.15-2023.10.21	Level1	Level1
Complete other options in the main interface and design the remaining levels.	2023.10.22-2023.11.1	Main scene	Main scene
Introduce trap designs and improve interactions between the character and traps.	2023.11.1-2023.11.7	Trap	Trap
Enhance the remaining buttons on the main interface and add an ending scene.	2023.11.8-2023.11.14	Staff, quit, setting	Staff, quit, setting
Incorporate data reading and storage, link all levels together and start the final debugging process.	2023.11.14-2023.12.1	data	data
Complete debugging, package the materials, and create an executable .exe program.	2023.12.1-2024.1.1	Debug, exe	Debug, exe
Write the final report, prepare a presentation PowerPoint, and make a demonstration video.	2024.1.1-2024.1.17	Final report, ppt, video	Final report, ppt, video

5. Testing Report

5.1 Function testing

Problem 1:

There are issues with data writing and reading, resulting in incorrect game level loading

Solution: Change the approach by rewriting the file reading and writing sections to make them more concise.

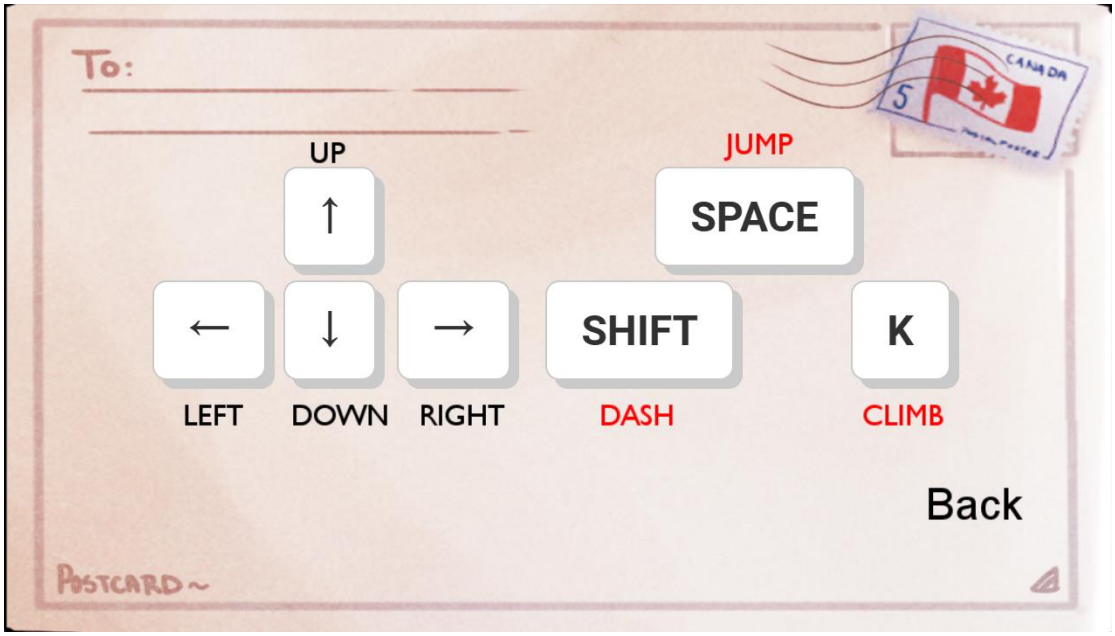
Problem 2:

The requirements for compiling debug and release versions are inconsistent when exporting

Solution: Modify the paths of some plist resources, using relative paths for all to ensure portability.

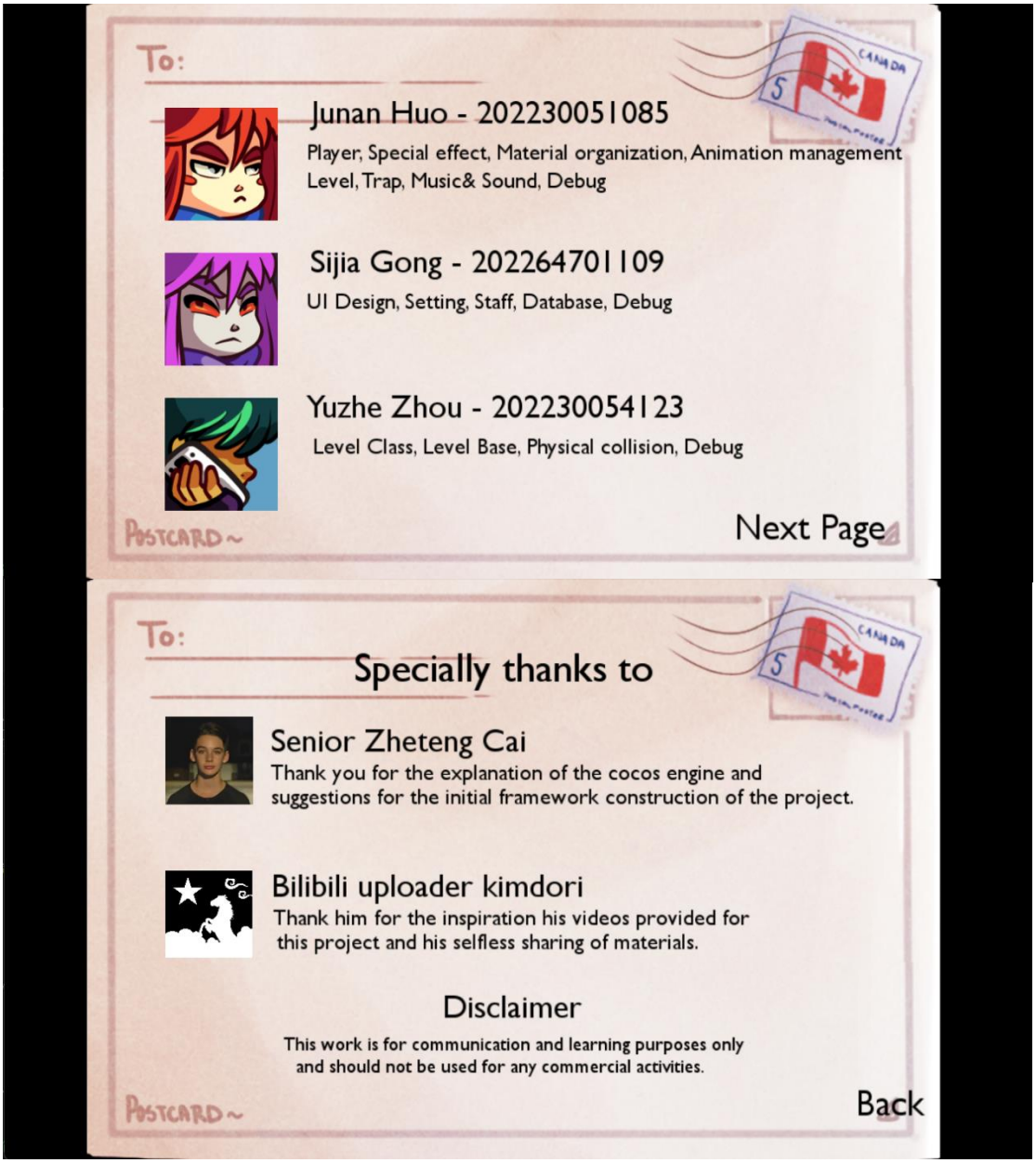
5.2 System testing

Main scene, music initialization, slider effect

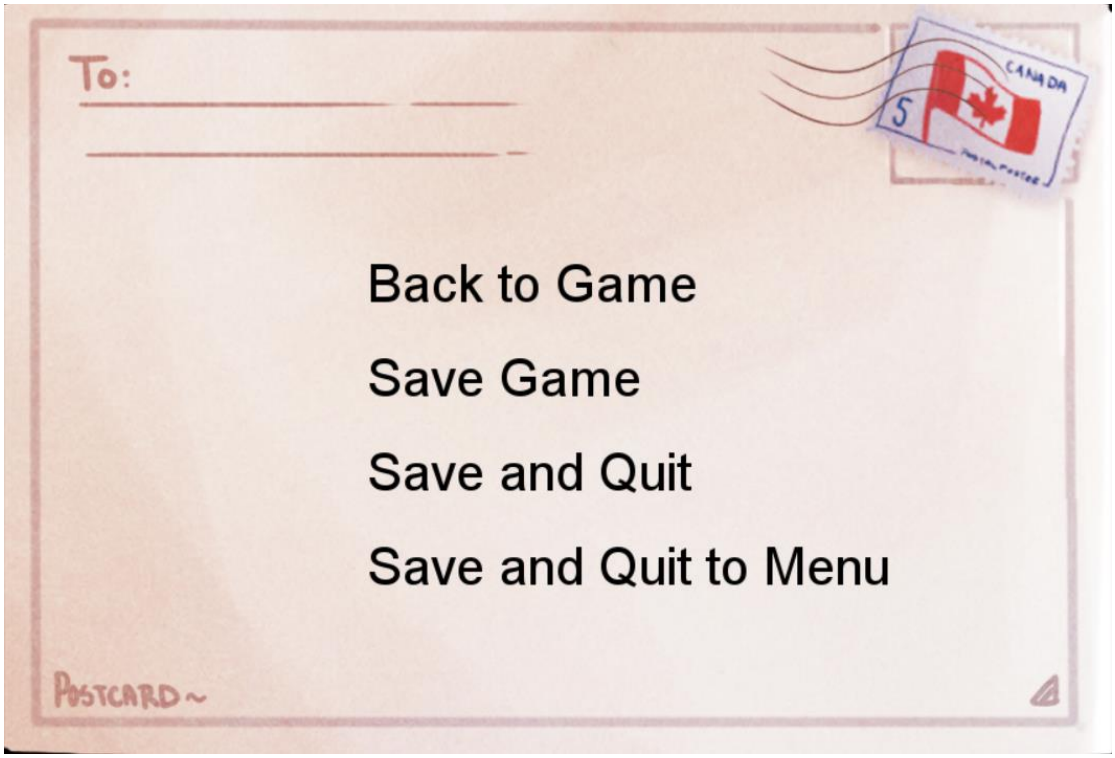


Setting

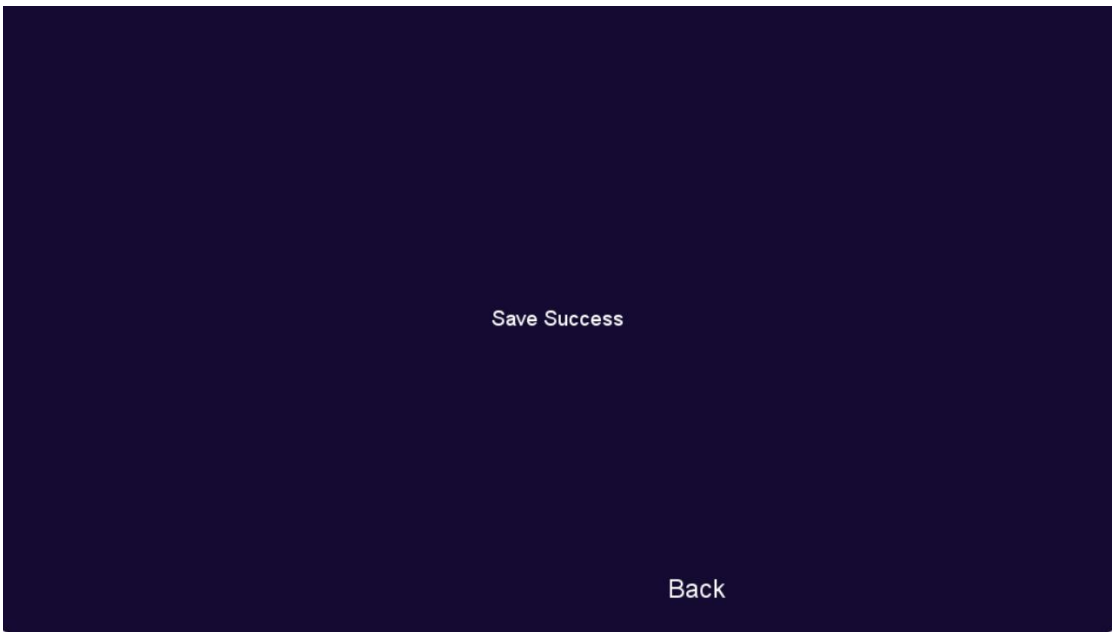
Exit



Staff, thanks, link with two pages



Pause menu



Save success

game

End image, music

6. Personal Summary

霍君安：

As the team leader, I handled almost all aspects of the project. This project made me realize that while programming skills are important for successfully completing a project, coordination among team members, collection of materials, and learning other skills also play a crucial role in the progress of the entire project. In this project, the selection of direction, acquisition and processing of materials, communication among team members, task allocation, and many methods are things that cannot be learned in a C++ classroom. Throughout the project, I also experienced the help of an unknown UP online, and the rationality of framework building under the experienced guidance of senior Cai, as well as the sense of achievement from exploring some unknown methods and finally succeeding. Although the final product may not be perfect in terms of animation, collision volume, or feel, I highly recognize this work. This challenging work has given me a great sense of satisfaction. Doing something different from others, exploring the unknown, and improving oneself is undoubtedly hard, but I believe this is the true meaning of this advanced program design project.

Some complaints and suggestions:

The Cocos engine indeed feels quite outdated, and it's not particularly user-friendly in practical use. For instance, I still haven't completely figured out how to utilize the built-in collision detection of the engine. The most problematic aspect is its lack of a visual programming interface, which makes adjusting the collision volumes in the scene extremely time-consuming. If you prioritize efficiency and prefer not to work solely with code, engines that come with a visual interface might be more suitable.

龚思嘉：

When I first received the task of designing Celeste games, I was very worried because I had no previous gaming experience. My main task was UI design, which was also an area I had never ventured into. I didn't spend too much time on this project before the final exam because I learned a lot in other classes, so I did it relatively hastily. Although the final result was not as perfect as we initially expected, we also created a decent game project under the leadership of our team leader and the concerted efforts of our team members. I also learned a lot about UI design and the knowledge and methods of the Cocos2dx engine in it. This experience has deepened my passion for game development and given me a deeper understanding of teamwork. This experience not only provided me with valuable practical experience, but also made me look forward to participating in more interesting and challenging game projects in the future.

周宇哲：

Over the past two months, my experience in developing a level in a Cocos2d-x game has been a deep dive into the complexities of game development, with a focus on scene creation, event handling, and the implementation of game mechanics. The process involved setting up physics-enabled scenes, implementing keyboard listeners for player interactions, and managing dynamic scene transitions, which enhanced my game programming skills. Key achievements include using raycasting for level transitions and creating custom physics shapes for game objects. Our team's use of GitHub for efficient code management, with separate branches for different aspects like AI and physics, was crucial in avoiding overlapping efforts and emphasized the importance of good version control in collaborative projects. Rapidly adapting to the cocos2d-x engine and applying its functionalities to our game demonstrated the importance of teamwork and adaptability in game development.

7. Special Thanks and Disclaimer

Special Thanks:

Senior Cai Zhengte, for his explanation of the Cocos engine and advice on the initial framework setup of the project.

Bilibili UP, kimdor, for the inspiration his videos provided for this project and his selfless sharing of materials.

Disclaimer:

This work is for exchange and learning purposes only and should not be used for any commercial activities.

8. Reference

1) Cocos2dx's API：Cocos2d-x: cocos2d-x

2) Cocos2dx's official Documents：了解引擎 · GitBook (cocos.com)

3) Cocos-Resource：fusijie/Cocos-Resource: :books:Cocos 资料大全（全版本） (github.com)

4）Inspiration Sources:个人作品 C++游戏项目简介_哔哩哔哩_bilibili