TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Nguyễn Quang Thông
Nguyễn Anh Vũ
Võ Nhật Phước
Hoàng Trung Nguyên

XÂY DỰNG GAME CỜ CARO

ĐỒ ÁN MÔN HỌC
KĨ THUẬT LẬP TRÌNH

Thành Phố Hồ Chí Minh, Tháng 04 năm 2023

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN MÔN HỌC
KĨ THUẬT LẬP TRÌNH
ĐỀ TÀI:
XÂY DỰNG GAME CỜ CARO

Nhóm 11

22127401 - Nguyễn Quang Thông

22127298 - Hoàng Trung Nguyên

22127339 - Võ Nhật Phước

22127463 - Nguyễn Anh Vũ

Giáo Viên Hướng Dẫn: Trương Toàn Thịnh

Thành Phố Hồ Chí Minh, Tháng 04 năm 2023

Lời cảm ơn

Mục lục

Danh sách hình

Chương 1.

Hướng dẫn sử dụng

1.1. Hiển thị hình ảnh

```
#figure(
    image("HCMUS_Logo.png", width: 40%),
    caption: [Test fig]
)
```

Nhớ bỏ caption vô không là cái mục lục hình bị lỗi

Hình 1: Test fig

1.2. Code có highlight (Paste code ít thôi, lấy cái nào quan trọng á)

```
    while (1) {
        auto tmp = InputHandle::Get();
        if (tmp == L"b" || tmp == L"B") {
            return NavHost.Back();
        }
        if (tmp == L"\r") {
            system(
                "rundll32 url.dll,FileProtocolHandler "
                "https://github.com/thng292/CaroGame"
            );
        }
    }
```

Chương 2.

Table

Thành viên	Đánh giá
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.	Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

Chương 3.

Tổng quan về trò chơi

3.1. Giới thiệu về trò chơi

3.1.1. Gomoku

Nguyên

3.1.2. Các yêu cầu về tính năng

- Có thể save, load trò chơi
- Nhận biết được thắng, thua, hòa
- Xử lí hiệu ứng thắng, thua, hòa
- Xử lí giao diện màn hình khi chơi
- Xử lí màn hình chính
- Game có nhiều ngôn ngữ, người dùng có thể thêm được ngôn ngữ mới
- Có thể load được các theme(chủ đề) bên ngoài
- Lưu được các thiết lập của người chơi
- Có thể lưu và phát lại các game đã hoàn thành
- Có nhiều chế độ chơi
- Có thể chơi với máy, máy có nhiều mức độ
- Game có thể phát nhạc nền, hiệu ứng. Có thể bật tắt được

3.1.3. Thông tin chung về trò chơi

Nguyên Link souce code, chạy trên nền tảng nào, ...

3.2. Mô tả về các tính năng của game

3.2.1. Màn hình chính

3.2.2. Save/Load game đang chơi, replay game đã chơi xong

3.2.3. Xử lí, hiệu ứng thắng, thua, hòa

3.2.4. Giao diện màn hình khi chơi

3.2.5. Đa ngôn ngữ

3.2.6. Thay đổi Theme(Chủ đề)

3.2.7. Lưu thiết lập của người chơi

3.2.8. Chế độ chơi Thường

3.2.9. Chế độ chơi Rush

3.2.10. Đánh với máy

3.2.11. Đánh với người

3.2.12. Các hỗ trợ trong lúc chơi game

3.2.12.1. Gợi ý

3.2.12.2. Nổi bật nước mới đi

3.2.12.3. Cảnh báo nước 4

3.2.12.4. Hoàn tác nước đi

3.2.12.5. Đi nháp

Chương 4.

Chi tiết các chức năng

4.1. Logic

4.1.1. Chơi hiệu ứng, nhạc nền

Âm thanh là một phần không thể thiếu trong các trò chơi điện tử, nó khiến cho trò chơi thêm sinh động và chân thực, nâng cao trải ngiệm thi chơi. Dưới đây là những phương pháp mà chúng em đã áp dụng để chơi âm thanh và những khó khăn mà chúng em đã gặp phải.

Các file âm thanh được đặt trong thư mục asset/audio và có thể truy cập bằng các `enum`. Các `enum` được map sang một mảng chứa tên các file âm thanh. Các hàm và class sau đây nằm trong `namespace Audio`, file `Audio.h`, `Audio.cpp`

```
enum class Sound : char {
    NoSound = 0,
    OnKey,
    Draw,
    Win,
    Lose,
    MenuBGM,
    MenuMove,
    MenuSelect,
    GameBGM,
    WinSound,
    GamePlace,
    GameStart,
    Pause,
    WarningSound
};
```

```
constexpr std::array SoundName{
    L"",
    L"Key.wav",
    L"Draw.mp3",
    L"Win.mp3",
    L"Lose.mp3",
    L"MenuBGM.mp3",
    L"MenuMove.wav",
    L"MenuSelect.wav",
    L"GameBGM.mp3",
    L"WinSound.mp3",
    L"GamePlaceMove.mp3",
    L"GameStart.wav",
    L"Pause.wav",
    L"Warning.mp3"
};
```

4.1.1.1. Giải pháp để chơi các âm thanh của giao diện

Để chơi các âm thanh của giao diện, giải pháp của chúng em là sử dụng hàm `PlaySound` [1]. Hàm này có thể chơi các tài nguyên âm thanh thông qua tên

file, con trỏ đến tài nguyên âm thanh trong bộ nhớ hoặc chơi âm thanh của một sự kiện hệ thống

Interface:

```
BOOL PlaySound(
    LPCTSTR pszSound,
    HMODULE hmod,
    DWORD   fdwSound
);
```

Parameter:

- `pszSound`: Con trỏ đến tên file âm thanh, tài nguyên âm thanh trong bộ nhớ hoặc tên sự kiện hệ thống
- `hmod`: Handle của chương trình chứa tài nguyên âm thanh, nếu chơi bằng tài nguyên âm thanh trong bộ nhớ
- `fdwSound`: Các flag để chỉ định cách chơi âm thanh

Return:

- `true` ⟹ phát âm thanh thành công
- `false` ⟹ phát âm thanh thất bại

Ví dụ sử dụng hàm `PlaySound`:

```
    // Chơi âm thanh từ file "recycle.wav"
    PlaySound(TEXT("recycle.wav"), NULL, SND_FILENAME);
    // Chơi âm thanh của 1 sự kiện hệ thống
    PlaySound(TEXT("SystemStart"), NULL, SND_ALIAS);
```

Ưu điểm:

- Khi gọi hàm sẽ tự load file vào memory, đọc và phát
- Sau khi gọi xong thì không cần quan tâm đến nữa nên có độ linh hoạt cao

Nhược điểm:

- Phải load cả file vào bộ nhớ khi chơi nên chỉ chơi những âm thanh ngắn, dung lượng nhỏ, vừa memory

- Khi chơi có độ delay cao (có thể được khắc phục bằng cách load trước file cần chơi)
- Chỉ có thể mở được file có định dạng `wav`.
- Không thể chơi cùng lúc nhiều âm thanh

Vì những âm thanh giao diện là những file ngắn, nhỏ, nên khắc phục được những điểm yếu của hàm và tận dụng tốt sự linh hoạt cao của hàm `PlaySound`. Nên chúng em đã chọn giải pháp này.

Để thuận tiện hơn trong việc sử dụng, chúng em đã viết hàm `PlayAndForget`.

Interface:

```
    bool PlayAndForget(Sound sound, bool wait)
```

Parameters:

- `Sound`: âm thanh cần chơi
- `wait`:
  - `true` ⟹ phát âm thanh một cách đồng bộ (synchronous)
  - `false` ⟹ phát âm thanh một cách bất đồng bộ (asynchronous)

Return:

- `true` ⟹ phát âm thanh thành công
- `false` ⟹ phát âm thanh thất bại

Usage:

```
    // Chơi âm thanh "MenuSelect.wav" bất đồng bộ
    Audio::PlayAndForget(Audio::Sound::MenuSelect);
```

4.1.1.2. Giải pháp để chơi nhạc nền

Với những nhược điểm trên thì hàm `PlaySound` không phù hợp để chơi những file dung lượng lớn như nhạc nền và những âm thanh yêu cầu độ trễ thấp. Vì vậy chúng em đã sử dụng một giải pháp khác là sử dụng `Media Control Interface` [2]. Media Control Interface là một chuẩn giao tiếp giữa các ứng dụng và các thiết bị âm thanh, video, hình ảnh, v.v... Nó cho phép

các ứng dụng giao tiếp với các thiết bị âm thanh, video, hình ảnh thông qua chuỗi lệnh đơn giản.

Để gửi lệnh đến thiết bị âm thanh, chúng em sử dụng hàm `mciSendString` [3]. Hàm này có thể gửi chuỗi lệnh đến thiết bị âm thanh và nhận kết quả trả về.

Interface:

```
    MCIERROR mciSendString(
        LPCTSTR lpszCommand,
        LPTSTR  lpszReturnString,
        UINT    cchReturn,
        HANDLE  hwndCallback
    );
```

Parameters:

- `lpszCommand`: Con trỏ đến chuỗi lệnh cần gửi đến thiết bị âm thanh
- `lpszReturnString`: Con trỏ đến mảng chứa chuỗi nhận thông tin trả về
- `cchReturn`: Độ dài của chuỗi trả về
- `hwndCallback`: Handle của cửa sổ sẽ nhận thông báo khi thiết bị âm thanh hoàn thành công việc

Return:

- Nếu gửi thành công sẽ trả về `0`, nếu lỗi trả về một giá trị biểu thị lỗi

Usage:

```
    // Mở file "song.mp3"
    mciSendString("open song.mp3 type mpegvideo alias song", NULL,
0, NULL);
    // Chơi file "song.mp3"
    mciSendString("play song from 0 repeat", NULL, 0, NULL);
    // Dừng file "song.mp3"
    mciSendString("stop song", NULL, 0, NULL);
    // Đóng file "song.mp3"
    mciSendString("close song", NULL, 0, NULL);
```

Ưu điểm:

- Chơi được nhiều định dạng âm thanh khác nhau

- Chơi được các file lớn
- Khi chơi ít bị delay do không cần load hết file vào memory
- Có thể chơi cùng lúc nhiều âm thanh

Nhược điểm:

- Phải sử dụng chuỗi để giao tiếp
- Phải tự quản lí các file âm thanh đã mở nên không có độ linh hoạt cao

Để khắc phục nhược điểm trên, chúng em đã tạo `class AudioPlayer` để việc sử dụng và quản lí tài nguyên thuận tiện, dễ dàng hơn

Interface:

```
class AudioPlayer {
    AudioPlayer();
    AudioPlayer(Sound song); // Khởi tạo và mở file

    // Mở file. Có thể dùng để đổi file cần chơi
    int Open(Sound song);
    Sound getCurrentSong() const;

    int Play(bool fromStart = 1, bool repeat = 0) const; // Chơi
    int Pause() const;      // Tạm dừng
    int Resume() const;     // Tiếp tục
    int Stop() const;       // Dừng chơi và trả con trỏ về đầu
    int Close();            // Đóng file đang mở

    ~AudioPlayer();         // Đóng file đang mở
}
```

Parameters:

- `song` : âm thanh cần chơi
- `fromStart` :
    - `true` ⟹ chơi từ đầu
    - `false` ⟹ chơi tiếp tại vị trí con trỏ
- `repeat` :
    - `true` ⟹ lặp lại khi kết thúc

Return:

- Các phương thức sẽ trả về `MCI code` của lệnh MCI tương ứng

Usage:

```
{
    Audio::AudioPlayer player(Audio::Sound::Draw);
    player.play(true, true);
}
```

`class AudioPlayer` có một nhược điểm lớn là tuổi thọ phụ thuộc vào thời gian sống của biến cục bộ và không thể được truy cập được từ các thành phần bên ngoài. Để khắc phục điểm yếu ấy chúng em đã tạo ra `class BackgroundAudioService` để tăng tuổi thọ của `class AudioPlayer`, đồng thời khiến cho nhạc nền có thể được điều khiển những nơi khác.

Interface:

```
class BackgroundAudioService {
    BackgroundAudioService() = delete;
    static Audio::Sound GetCurrentSong();

    static int ChangeSong(Audio::Sound song) // Đổi nhạc

    static int Play(bool fromStart = 0, bool repeat = 1); // Chơi
    static int Pause();                                   // Tạm dừng
    static int Resume();                                  // Tiếp tục
    // Dừng chơi, trả con trỏ về đầu
    static int Stop();
};
```

Parameters:

- `song`: âm thanh cần chơi
- `fromStart`:
    - `true` ⟹ chơi từ đầu
    - `false` ⟹ chơi tiếp tại vị trí con trỏ
- `repeat`:
    - `true` ⟹ lặp lại khi kết thúc

Usage:

```
{
    BackgroundAudioService::ChangeSong(Audio::Sound::MenuBGM);
    BackgroundAudioService::Play(true, true);
}
```

4.1.2. Điều hướng trong ứng dụng

Việc chuyển đổi giữa các màn hình khác nhau trong trò chơi là một thách thức lớn đối với chúng em, vì đây là lần đầu chúng em gặp phải vấn đề này. Để giải quyết vấn đề này, ban đầu chúng em gọi các hàm trực tiếp từ main, muốn chuyển tới màn hình nào thì gọi hàm của màn hình đó. Nhưng phương pháp này nhanh chóng để lộ nhiều điểm yếu:

- Cần phải biết chữ kí hàm của màn hình cần chuyển đến
- Khó quản lí các màn hình và các đích đến của chúng
- Có thể bị tràn stack khi chuyển màn hình nhiều lần
- Nếu muốn sửa lại code phải sửa ở nhiều nơi
- Khó mở rộng, dễ lỗi

Để khắc phục điểm yếu đó, chúng em đã tạo ra một hệ thống để quản lí các màn hình, lấy ý tưởng từ thư viện `navigation-compose` [4] (thư viện điều hướng của Jetpack [5]). Hệ thống này giúp chúng em có thể chuyển đổi giữa các màn hình một cách dễ dàng, linh hoạt và có thể mở rộng dễ dàng hơn.

Hệ thống này gồm 2 thanh phần:

- `class NavigationHost`: class trung tâm để quản lí các màn hình
- Các màn hình: là các hàm có chữ kí như sau:
  - `void ScreenName(NavigationHost& host)`

Mỗi màn hình sẽ được gán một nhãn độc nhất, nhãn này sẽ được sử dụng để chuyển đến màn hình đó. Các màn hình muốn chuyển đến màn hình khác sẽ gọi phương thức `Navigate("Nhãn")` để chuyển đến màn hình đó hoặc `NavigateStack("Nhãn")` để chuyển đến màn hình đó nhưng không xóa đi màn hình hiện tại. Ngoài ra, hệ thống còn có xử lí việc xóa màn hình trước khi

chuyển đến màn hình mới, lưu lịch sử di chuyển giữa các màn hình để có thể quay lại màn hình trước đó và cung cấp một số phương thức để hỗ trợ việc truyền dữ liệu giữa các màn hình.

Interface:

```
#define ViewFunc std::function<void(NavigationHost&)>
#define ViewFuncMap std::unordered_map<std::string, ViewFunc>

class NavigationHost {

    NavigationHost() = default;
    NavigationHost(
        const std::string& Start,
        const ViewFuncMap& links
    );

    // Các phương thức hỗ trợ truyền dữ liệu giữa các màn hình
    std::any& GetFromContext(const std::string& name);
    bool CheckContext(const std::string& name);
    void SetContext(
        const std::string& name,
        const std::any& data
    );
    void DeleteContext(const std::string& name);

    // Thêm màn hình khi đang chạy
    void Add(const std::string& path, const ViewFunc& view);

    // Các phương thức hỗ trợ điều hướng
    void NavigateStack(const std::string& path);
    void Navigate(const std::string& path);
    void Back();
    void BackToLastNotOverlay();
    void NavigateExit();

    ~NavigationHost();
};
```

Parameters:

- `Start`: nhãn của màn hình bắt đầu

- `links` : danh sách các màn hình có trong ứng dụng và nhãn của chúng
- `path` : nhãn của màn hình cần chuyển đến
- `view` : hàm của màn hình cần chuyển đến
- `name` : nhãn của dữ liệu cần truyền
- `data` : dữ liệu cần truyền

Usage:

```
#include <iostream>

void GameScreen(NavigationHost& NavHost) {
    // Lấy dữ liệu đã truyền
    int a = std::any_cast<int>(NavHost.GetFromContext("Context"));
    std::cout << a;
    // ...
    // Thoát chương trình
    return NavHost.NavigateExit();
}

void StartScreen(NavigationHost& NavHost) {
    // Truyền dữ liệu giữa các màn hình
    NavHost.SetContext("Context", 90);
    // ...
    return NavHost.Navigate("Game");
}

int main() {
    // Khởi tạo hệ thống điều hướng
    NavigationHost(
        "Start",
        {
            {"Start", StartScreen},
            {"Game", GameScreen},
        }
    );
    return 0;
}
```

Ưu điểm:

- Dễ mở rộng

- Không cần phải biết tên hàm để chuyển màn hình
- Khi thay đổi chỉ cần sửa ở một nơi

Nhược điểm:

- Khó chuyển dữ liệu giữa các màn hình
- Dễ đánh sai nhãn màn hình

4.1.3. Phương pháp lưu và tải game(save/load game)

Để lưu game, chúng em đã chọn lưu trạng thái của game vào một file văn bản thuần và lưu trong một thư mục riêng. Để tải game, chúng em sẽ đọc file văn bản đó và khởi tạo lại trạng thái của game. Khi người dùng muốn tải game, chúng em muốn hiển thị một danh sách các file lưu game để người dùng có thể chọn file cần tải. Có nhiều phương pháp để thực hiện việc này. Một trong những giải pháp mà chúng em đã cân nhắc là lưu tên file lưu game vào một file văn bản thuần, khi cần tải, chúng em sẽ đọc file đó và hiển thị cho người dùng. Tuy nhiên, chúng em đã quyết định không sử dụng phương pháp này do nó có nhiều khuyết điểm như:

- Tạo ra một file không cần thiết
- Người dùng không thể tải game nếu file đó bị xóa/lỗi
- Người dùng không thể load các file copy từ máy khác

Một cách tiếp cận khác là mỗi khi người dùng muốn tải game thì sẽ duyệt qua các file trong thư mục lưu game và hiển thị cho người dùng. Điều này có nhiều ưu điểm như:

- Không cần tạo ra file không cần thiết
- Người dùng có thể tải game từ máy khác

Để quét các file trong thư mục, chúng em đã sử dụng thư viện `filesystem` [6]. Đây là một thư viện mới xuất hiện trong phiên bản C++17. Nó cung cấp các tiện ích để thực hiện các thao tác trên hệ thống tập tin và các thành phần của chúng, chẳng hạn như đường dẫn, tập tin thông thường và thư mục. Để việc sử dụng thư viện thuận tiện hơn, chúng em đã viết hàm `GetAllTextFileInDir`.

Implementation:

```
namespace FileHandle {
struct FileDetail {
    std::filesystem::path               filePath;
    std::filesystem::file_time_type     lastModified;
};

std::vector<FileHandle::FileDetail>
GetAllTextFileInDir(
    const std::filesystem::path& Dir
)
{
    std::vector<FileDetail> res;
    Ensure(Dir);
    for (auto& file : std::filesystem::directory_iterator(Dir)) {
        if (file.is_regular_file()) {
            res.emplace_back(
                file.path(),
                file.last_write_time()
            );
        }
    }
    return res;
}
} // namespace FileHandle
```

Note: Chi tiết về hàm `Ensure` nằm ở Mục 4.1.3.1.2

Parameters:

- `Dir`: đường dẫn đến thư mục muốn tìm

Return:

- Trả về một `vector` chứa các thông tin của các file đã tìm được

Usage:

```
{
    // Tìm các file văn bản trong đường dẫn
    // tương đối "saves"
    auto files = FileHandle::GetAllTextFileInDir(
        "saves"
    );
    for (auto& file:files) {
        std::cout << file.filePath.filename() << '\n';
    }
}
```

4.1.3.1. Các hàm hỗ trợ khác

Các hàm sau nằm trong `namespace FileHandle`, file `FileHandle.h`, `FileHandle.cpp`

4.1.3.1.1. Các hàm hỗ trợ mở file

Hỗ trợ mở các file văn bản `utf-8`

Interface:

```
    typedef std::filesystem::path fsPath;
    std::wofstream OpenOutFile(const fsPath& filePath);
    std::wifstream OpenInFile (const fsPath& filePath);
```

Parameters:

- `filePath`: đường dẫn đến file cần mở

Usage:

```
#include <string>
{
    std::wstring str = L"Tiếng Việt";
    auto outFile = FileHandle::OpenOutFile("test.txt");
    outFile <<  str;
    outFile.close();
    auto inFile  = FileHandle::OpenInFile ("test.txt");
    infile  >> str;
}
```

4.1.3.1.2. Hàm Ensure

Dùng để đảm bảo đường dẫn đến file muốn mở có tồn tại, nếu không tồn tại, nếu không tồn tại thì tạo đường dẫn đó.

Interface:

```
    void Ensure(const std::filesystem::path& Dir);
```

Parameters:

- `Dir`: đường dẫn muốn kiểm tra/tạo

Return:

- Các fstream tương ứng với thao tác In/Out

Usage:

```
{
    // Đảm bảo đường dẫn tương đối "asset/language" tồn tại
    FileHandle::Ensure("asset/language");
}
```

4.1.3.1.3. Hàm Delete

Dùng để xóa file

Interface:

```
    bool Delete(const std::filesystem::path& target)
```

Parameters:

- target: đường dẫn tới file cần xóa

Return:

- Trả về `true` nếu xóa thành công, `false` khi lỗi

Usage:

```
{
    // Xóa file tmp.cpp
    bool res = FileHandle::Delete("tmp.cpp");
    if (res) {
        std::cout << "Success";
    } else {
        std::cout << "Failed";
    }
}
```

4.1.4. Đếm giờ trong khi chơi game

Việc đếm và hiển thị thời gian trực tiếp trong lúc chơi khá phức tạp vì luồng chính trong game luôn phải chờ đợi và xử lí đầu vào của người dùng, nên việc sử dụng luồng chính để nó phụ thuộc vào việc người chơi có thực hiện các thao tác trong game hay không. Nếu người chơi không thực hiện các thao tác trong game thì thời gian sẽ không được cập nhật lên màn hình. Để giải quyết vấn đề này, chúng em đã tạo thêm một luồng riêng để đếm thời gian và cập nhật lên màn hình. Sử dụng thư viện `thread` [7] để tạo luồng mới, chúng em đã có giải pháp để chạy một hàm sau một khoảng thời gian nhất định.

Implementation:

```cpp
using namespace std::chrono;

struct TimerInternalState {
    std::function<void(void)> callback;
    milliseconds interval;
    bool running = false;
    bool pause = false;
};

class Timer {
    std::thread _thread;
    std::shared_ptr<TimerInternalState> _state {
        new TimerInternalState };

   public:
    Timer(
        std::function<void(void)> callback,
        const long& interval = 1000
    ) {
        _state->callback = callback;
        _state->interval = milliseconds{interval};
    }

    inline void Start()
    {
        _state->running = true;
        auto state = _state;
        _thread = std::thread([state] {
            while (state->running) {
                auto nextInterval = steady_clock::now();
                nextInterval += state->interval;
                if (!state->pause) { state->callback(); }
                std::this_thread::sleep_until(nextInterval);
            }
        });
        _thread.detach();
    }

    inline void Pause() { _state->pause = true; }
    inline void Continue() { _state->pause = false; }
    inline void Stop() { _state->running = false; }
    inline ~Timer() { Stop(); }
};
```

Parameters:

- `callback`: hàm sẽ được gọi sau mỗi khoảng thời gian `interval`
- `interval`: khoảng thời gian giữa các lần gọi hàm `callback` tính bằng mili giây

Việc lập trình đa luồng trong C++ khá phức tạp, và cũng là phần dễ gây lỗi nhất trong trò chơi, do việc vẽ lên màn hình phải được đồng bộ giữa các luồng với nhau. Nếu không đồng bộ thì có thể dẫn đến việc các phần tử trên màn hình bị vẽ sai vị trí. Để việc đó không xảy ra, chúng em đã sử dụng một khóa `mutex` [8] chung để đồng bộ các luồng với nhau.

Đoạn code sử dụng `Timer` và `mutex` trích từ trò chơi:

```
void GameScreenView::GameScreenView(NavigationHost& NavHost) {
// ...
std::mutex lock;
// ...
Timer timerPlayerOne(
    [&] {
        if (!endGame) {
            curGameState.playerTimeOne += timeAddition;
            std::lock_guard guard(lock);
            auto currPos = View::GetCursorPos();
            // Vẽ thời gian lên màn hình
            gameScreen.timerContainerOne.DrawToContainer(
                Utils::SecondToMMSS(curGameState.playerTimeOne),
                Theme::GetColor(ThemeColor::PLAYER_ONE_COLOR)
            );
            View::Goto(currPos.X, currPos.Y);
            if (curGameState.playerTimeOne == 0 && !endGame) {
                // Xử lí khi hết thời gian
            }
        }
    }
);
}
```

4.1.5. Giải pháp cho đa ngôn ngữ

Các văn bản trong trò chơi thay vì được code cứng vào trò chơi thì sẽ được load từ một file riêng, điều này kiến cho phần ngôn ngữ trong game có thể được chỉnh sửa một cách dễ dàng và khiến cho việc thêm các ngôn ngữ mới dễ dàng hơn.

File ngôn ngữ là một file văn bản thuần chứa các nhãn và phần văn bản ngăn cách bởi dấu "=", các nhãn có nằm bên trong cặp ngoặc `[]` là `meta` được dùng để chứa thông tin về file ngôn ngữ. Sau khi load xong, các nhãn và văn bản sẽ được lưu vào một bảng để có thể dễ dàng truy xuất.

Ví dụ file ngôn ngữ:

```
[LANGUAGE]                  =    English
[LANG_SELECT]               =    Language
ABOUT_DESC                  =    LIST OF GROUP MEMBERS AND SOURCE
CODE LINK
ABOUT_SHORTCUT              =    A
ABOUT_TITLE                 =    About us
ABOUT_US_TITLE              =    About us
```

Các nhãn và văn bản trong trò chơi được quản lí, truy xuất và load thông qua class `Language` nằm trong file `Language.h` và `Language.cpp`

Interface:

```
typedef std::unordered_map<std::wstring, std::wstring> Dict;
typedef std::filesystem::path fsPath;

struct LanguageOption {
  Dict meta;
  fsPath path;
};

class Language {
  static Dict languageDict;

public:
  Language() = delete;

  // Chỉ đọc các phần thông tin về ngôn ngữ
  static Dict ExtractMetaFromFile(const fsPath& filePath);

  // Load file ngôn ngữ
  static void LoadLanguageFromFile(const fsPath& filePath);

  // Tìm các file ngôn ngữ
  static std::vector<LanguageOption>
  DiscoverLanguageFile(const fsPath& dirPath);

  // Truy xuất văn bản bằng nhãn
  static const std::wstring&
  GetString(const std::wstring& Label);

  static const std::wstring&
  GetMeta(const std::wstring& Label);
};
```

Parameters:

- `filePath`: đường dẫn tới file cần mở
- `dirPath`: đường dẫn tới thư mục cần tìm
- `Label`: nhãn của văn bản cần lấy

Usage:

```
{
    Language::LoadLanguageFromFile("asset/language/en.txt");
    std::cout << Language::GetMeta(L"[LANGUAGE]");
    std::cout << Language::GetMeta(L"ABOUT_TITLE");
}
```

4.1.6. Cài đặt

Thông

4.1.7. Chủ đề

Thông

4.1.8. Hàm trung gian hỗ trợ vẽ giao diện

Thông

4.1.9. Nhận biết thắng thua

Việc nhận biết kết quả thắng, thua, và hòa của một ván đấu được thực hiện trong `namespace Logic` của chương trình. Các kết quả này là điều kiện để chương trình quyết định kết thúc ván đấu. Ngoài ra, việc biết được kết quả thắng, thua, và hòa sẽ giúp AI của trò chơi đưa ra đánh giá về trạng thái bàn cờ một cách đúng đắn.

4.1.9.1. Hàm GetGameState

Hàm `GetGameState` có vai trò đánh giá hiện trạng của ván đấu sau nước đi mới nhất. Cụ thể hơn, hàm xem xét nước đi mới nhất có dẫn đến một **kết quả thắng** hay **kết quả hòa**. Một nước đi sẽ dẫn đến kết quả thắng nếu nước đi đó tạo nên một chuỗi 5 nước đi liên tiếp đồng chất, và một nước đi sẽ dẫn đến kết quả hòa nếu nước đi đó không phải là nước đi thắng, đồng thời là nước đi hợp lệ cuối cùng của bàn đấu.

Interface

```
typedef std::vector<std::vector<short>> Board;
struct Point {
    int row, col;
}

short GetGameState(
    const GameAction::Board& board,
    const short& moveCount,
    const GameAction::Point& move,
    const short& playerValue,
    GameAction::Point& winPoint = temp,
    bool getWinPoint
);
```

Parameters

- `board`: Bàn đấu hiện tại.
- `moveCount`: Số nước đi đã thực hiện.
- `move`: Nước đi mới nhất.
- `playerValue`: Người chơi thực hiện nước đi.
- `winPoint`: Đầu mút của chuỗi thắng (nếu có).
- `getWinPoint`:
    - `true` => lấy đầu mút của chuỗi thắng (nếu có).
    - `false` => không lấy đầu mút của chuỗi thắng.

Return

- `Logic::WIN_VALUE`: Giá trị tượng trưng kết quả thắng (người thắng là playerValue).
- `Logic::DRAW_VALUE`: Giá trị tượng trưng kết quả hòa.
- `Logic::NULL_VALUE`: Giá trị tượng trưng kết quả vô định.

Usage

```
{
    short gameState = Logic::GetGameState(
        gameBoard,
        moveCount,
        latestMove,
        currentPlayer,
        winPoint,
        true);
    if (gameState == Logic::WIN_VALUE) {
        std::cout << currentPlayer << " wins";
    }
    else if (gameState == Logic::DRAW_VALUE) {
        std::cout << "Game draw";
    }
}
```

4.1.9.2. Cách phát hiện nước đi thắng

Nước đi thắng là một nước đi dẫn đến một chuỗi 5 quân cờ liền kề đồng chất, dựa vào đặc điểm này, ta viết nên các hàm xử lý đáp ứng việc kiểm tra ấy. Cụ thể, bên trong hàm `GetGameState`, việc kiểm tra nước thắng sẽ được thực hiện qua bốn hàm `CheckVerticalWin`, `CheckHorizontalWin`, `CheckLeftDiagonalWin`, và `CheckRightDiagonalWin`. Các hàm ấy sẽ kiểm tra nước thắng cho 4 hướng tương ứng là: dọc, ngang, chéo từ trái sang phải, chéo từ phải sang trái. Tuy khác về hướng kiểm tra, nhưng bốn hàm đều có chung một phương pháp kiểm tra. Ví dụ, đối với hàm `CheckVerticalWin`, việc kiểm tra sẽ diễn ra như sau:

- Từ vị trí của nước đi mới nhất, ta cho kiểm tra nước ngay cạnh, theo hướng của chiều dọc, có **đồng chất** hay không:
    - **Nếu có** => tiếp tục kiểm tra nước ngay cạnh nước ấy.
    - **Nếu không** => ngưng kiểm tra.
- Nếu số nước đồng chất tính được bằng số nước đi thắng (trong trường hợp này là 5) thì trả về **true**, ngược lại trả về **false**.

Implementation

```
bool CheckVerticalWin(
    const GameAction::Board& board,
    const GameAction::Point& move,
    const short& playerValue
)
{
    // Biến đếm số quân cờ đồng chất
    short sameValueCount = 1;

    // Kiểm tra theo chiều dọc trên
    for (short row = move.row + 1; row < Constants::BOARD_SIZE; ++row) {
        if (board[row][move.col] == playerValue) {
            // Tăng số quân đồng chất tìm được
            sameValueCount++;
            // Nếu số quân bằng 5, trả về true
            if (sameValueCount == Constants::WIN_VALUE_COUNT) {
                return true;
            }
        } else {
            break;
        }
    }

    // Kiểm tra theo chiều dọc dưới
    for (short row = move.row - 1; row >= 0; --row) {
        if (board[row][move.col] == playerValue) {
            sameValueCount++;
            if (sameValueCount == Constants::WIN_VALUE_COUNT) {
                return true;
            }
        } else {
            break;
        }
    }

    return false;
}
```

Usage

```
{
    /* Nếu một trong các phương kiểm tra trả về`true
    thì nước đang xét là nước đi thắng*/
    if (CheckVerticalWin(board, move, playerValue)
    || CheckHorizontalWin(board, move, playerValue))
    || CheckLeftDiagonalWin(board, move, playerValue))
    || CheckRightDiagonalWin(board, move, playerValue))
    {
        return Logic::WIN_VALUE;
    }
}
```

4.1.9.3. Cách phát hiện nước đi hòa

Trận đấu sẽ đạt kết quả hòa nếu:

- Không có người chơi nào thắng.
- Không còn vị trí trống để thực hiện nước đi kế tiếp.

Từ đó, ngay sau việc kiểm tra nước thắng, nếu không có người thắng, ta sẽ có gọi hàm kiểm tra nước hòa thỏa mãn các điều kiện trên.

Implementation

```
bool CheckDraw(const short& moveCount) {
    return (moveCount == BOARD_SIZE * BOARD_SIZE);
}
```

Usage

```
{
    if(CheckDraw(moveCount) == true) {
        return Logic::DRAW_VALUE;
    }
}
```

4.1.10. Các tương tác với bàn cờ

Trạng thái bàn cờ có thể được thay đổi qua hai hình thức: **thực hiện nước đi** và **xóa nước đi**. Hàm `MakeMove` và `UndoMove` đảm nhiệm việc thực hiện hai chức năng ấy. Hai hàm này tuy đơn giản nhưng nắm vai trò quan trọng xuyên suốt

quá trình chơi, vì các thao tác người chơi chỉ thực sự được ghi lại trên bàn cờ khi hai hàm này được gọi đến. Việc xóa nước đi là để phục vụ cho chức năng **Hoàn tác** của trò chơi.

Interface

```
 // Thực hiện nước đi lên bàn cờ
 void MakeMove(
    Board& board,
    short& moveCount,
    const Point& move,
    const short& playerValue
);

// Xóa nước đi khỏi bàn cờ
void UndoMove(
    Board& board,
    short& moveCount,
    const Point& move
);
```

Parameters

- `board`: Bàn đấu hiện tại.
- `moveCount`: Số nước đi đã thực hiện.
- `move`: Nước đi được thực hiện.
- `playerValue`: Người chơi thực hiện nước đi.

Usage

```
{
    //...
    /* Gán vị trí hiện tại của con trỏ là nước đi mới nhất.*/
    latestMove = {row, col};
    GameAction::MakeMove(
        board,
        moveCount,
        latestMove,
        currentPlayer);
}
```

4.1.11. Chế độ đánh với máy

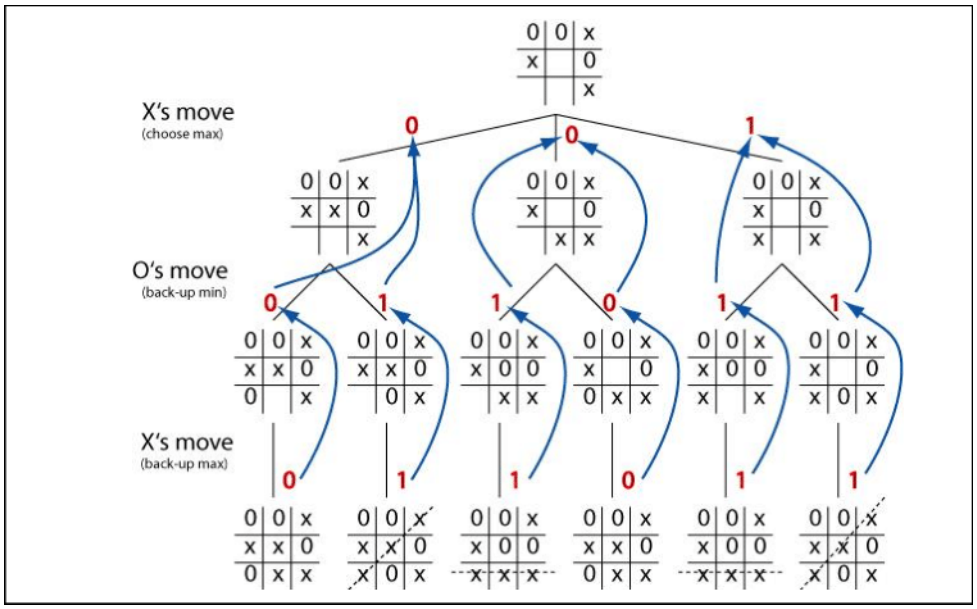
Việc thiết kết chương trình cho chế độ **"Đánh với máy"** là một trong những thách thức lớn nhất của đồ án. Khác với những tính năng khác của chương trình, tính năng này đòi hỏi những mảng kiến thức chuyên biệt về các thuật toán, kĩ thuật cụ thể. Ngoài ra, việc đánh giá độ đúng/sai của chương trình, hay nói cách khác là nước đi máy tính tìm được là tốt hay xấu, sẽ phần lớn phụ thuộc vào cảm tính và sự hiểu biết của người viết. Chính vì vậy, chương trình có thể đánh hay đối với người này, nhưng đánh không tốt đối với người khác. Phần tiếp theo sẽ trình bày những kĩ thuật mà nhóm đã sử dụng cho chức năng này.

4.1.11.1. Thuật toán Minimax

Đối với những trò chơi đối kháng hai người như cờ vua, cờ caro,... Để có thể tìm được một nước tốt qua code, ta không thể lập trình cứng để xét thế nào là nước đi tốt, thế nào là nước đi xấu được. Đơn thuần là vì có quá nhiều trường hợp để xét, và làm như thế chưa thể đảm bảo có thực sự đạt hiệu quả hay không. Thay vào đó, ta cần phải dựa vào thế mạnh của máy tính. Tuy không thể tư duy như con người, nhưng máy tính có khả năng thực hiện hàng chục nghìn phép tính trong một khoảng thời gian rất ngắn. Dựa vào đặc tính ấy, ta phát triển được phương pháp tìm một nước đi tốt cho cờ Caro.

Thuật toán Minimax[9] là một thuật toán phổ biến được áp dụng trong việc tìm kiếm một nước đi tốt trong các trò chơi đối kháng giữa hai người. Chính vì vậy, nhóm đã quyết định sử dụng thuật toán này để viết nên chương trình "AI" cho trò chơi. Giải thích một cách đơn giản, thuật toán sẽ tìm nước đi tốt nhất thông qua việc đánh giá tất cả các nước đi có thể trong mỗi lượt đi. Ví dụ, đối với cờ Caro, nếu hiện tại là lượt của người chơi O, thuật toán sẽ tìm mọi nước đi có thể của người chơi O. Sau khi đã thực hiện lượt chơi của O, thuật toán sẽ tìm mọi nước đi có thể của người chơi X. Quá trình này sẽ lặp lại đến một độ sâu nhất định, và khi đã đến độ sâu cuối cùng, một phép đánh giá tương đối sẽ được thực hiện để đánh giá "điểm" của bàn cờ. Người chơi **"tối đa hóa"** sẽ cố gắng

đạt được bàn cờ có điểm số cao nhất, ngược lại, người chơi **"tối thiểu hóa"** sẽ cố gắng đạt được bàn cờ có điểm số thấp nhất.



Hình 2: Sơ đồ tìm kiếm Minimax đối với trò chơi Tic-Tac-Toe

Ta sẽ áp dụng thuật toán này vào việc thiết kế chương trình AI của trò chơi. Mỗi khi người chơi hoàn thành thực hiện lượt chơi của mình, hàm `GetBestMove` sẽ được gọi để tìm nước đi tốt nhất cho lượt chơi của AI. Bên trong hàm `GetBestMove`, ta thực hiện việt tìm kiếm nước đi tốt nhất thông qua thuật toán Minimax.

Interface

```
GameAction::Point GetBestMove(
    GameAction::Board& board,
    short& moveCount
);
```

Parameters

- `board`: bàn cờ hiện tại
- `moveCount`: số nước đi đã thực hiện

Return

- Nước đi tốt nhất AI tìm được cho lượt đánh hiện tại

Usage

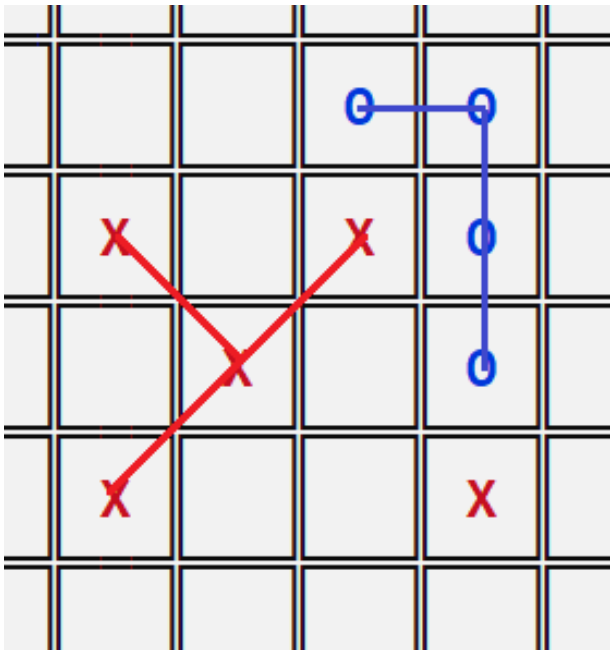
```
{
    // Tìm nước đi tốt nhất của AI
    GameAction::Point aiMove = AI::GetBestMove(board, moveCount);
    // Thực hiện nước đi AI
    GameAction::MakeMove(board, moveCount, aiMove);
}
```

4.1.11.2. Đánh giá bàn cờ

Thành phần quan trọng nhất trong thuật toán Minimax là **hàm đánh giá trạng thái**. Cần phải biết được trong một bàn cờ nhất định, lợi thế đang thuộc về người chơi nào. Trong cờ Caro, ta thấy rằng mục tiêu của mỗi nước đánh đều sẽ cố đạt được chuỗi 5 nước đồng chất liên tiếp. Ta gọi đó là chuỗi ấy là chuỗi **5 combo**. Để có thể đạt được 5 combo, ta phải có được chuỗi 4 nước đồng chất liên tiếp, gọi là chuỗi 4 combo. Và tương tự, muốn được 4 combo ta phải có 3 combo, muốn có 3 combo ta phải có nước 2 combo,... Nhìn chung, có thể thấy người chơi có được combo với độ dài càng gần với 5, họ sẽ có tỉ lệ thắng cao hơn. Ngoài ra, giả sử hai người chơi có **số lượng combo** 1 và combo 2 như nhau, thì người có số lượng combo 3 hay combo 4 lớn hơn sẽ có lợi thế cao hơn. Cuối cùng, một combo **bị chặn** (có một quân cờ của đối phương ở một hoặc cả hai đầu mút của combo) càng ít sẽ cho lợi thế càng cao. Vậy, dựa vào những tính chất ấy, ta xây dựng được thuật toán đánh giá như sau:

- Với một bàn cờ cho trước, ta duyệt qua tất cả những nước đi đã được thực hiện.
- Tại vị trí của mỗi nước đi, ta tính số điểm combo nước đó mang lại:
    - Ta gán cho mỗi combo một số điểm, combo có độ dài càng cao sẽ cho số điểm càng lớn.
        - Trong chương trình, combo 1, 2, 3, 4 không bị chặn có số điểm tương ứng là 1, 10, 50, 200.
    - Combo bị chặn một đầu sẽ bị giảm một nửa số điểm, bị chặn hai đầu sẽ không có điểm.

    - Combo sẽ được tính theo ba hướng: dọc, ngang và chéo. Ta lấy tổng số điểm của các hướng này trả lại.
- Nếu nước đi đang xét có cùng giá trị với người chơi đang xét, ta cộng số điểm combo tính được vào tổng điểm đánh giá. Ngược lại, nếu nước đi đang xét khác với người chơi đang xét, ta trừ giá trị điểm tính được khỏi tổng điểm đánh giá.
- Mỗi combo chỉ được xét một lần, tức nếu một nước thuộc một combo đã được duyệt từ trước, thì ta sẽ không xét combo cùng hướng tính từ nước này.



Hình 3: Minh họa combo mỗi người chơi qua đường nối liền

Implementation

```
short Evaluation::GetComboEval(
    const GameAction::Board& board, const short& playerValue
)
{
    // Bảng đánh dấú combo
    GameAction::Board comboCheckBoard(
        Constants::BOARD_SIZE,
        std::vector<short>(Constants::BOARD_SIZE, 210)
    );

    short evalResult = 0;
    for (short row = 0; row < Constants::BOARD_SIZE; ++row) {
        for (short col = 0; col < Constants::BOARD_SIZE; ++col) {
            if (board[row][col]) {
                short evalValue = 0;
                // Combo chưa được đánh dấú sẽ được kiểm tra
                if (comboCheckBoard[row][col] % 2 == 0) {
                // Lấý sô´điểm có được từ combo theo phương ngang
                    short eval = GetHorizontalComboEval(
                        board,
                        comboCheckBoard,
                        {row, col},
                         board[row][col]
                    );
                    // Cộng vào tổng điểm của nước đi này
                    evalValue += eval;
                }
```

```
                if (comboCheckBoard[row][col] % 3 == 0) {
                // Lấý sô´điểm có được từ combo theo phương dọc
                    short eval = GetVerticalComboEval(
                        board,
                        comboCheckBoard,
                        {row, col},
                        board[row][col]
                    );
                    evalValue += eval;
                }
```

```
                if (comboCheckBoard[row][col] % 5 == 0) {
                /*Lấy số điểm có được từ combo theo phương chéo
                phải sang trái*/
                    short eval = GetDiagonalRightComboEval(
                        board,
                        comboCheckBoard,
                        {row, col},
                        board[row][col]
                    );
                    evalValue += eval;
                }


                if (comboCheckBoard[row][col] % 7 == 0) {
                /*Lấy số điểm có được từ combo theo phương chéo
                trái sang phải*/
                    short eval = GetDiagonalLeftComboEval(
                        board,
                        comboCheckBoard,
                        {row, col},
                        board[row][col]
                    );
                    evalValue += eval;
                }

                if (board[row][col] == playerValue)
                    // Cộng vào tổng điểm đánh giá của bàn cờ
                    evalResult += evalValue;
                else
                    // Trừ khỏi tổng điểm đánh giá của bàn cờ
                    evalResult -= evalValue;
            }
        }
    }
    return evalResult;
}
```

Usage

```
{
    // Kết thúc một lần tìm kiếm của Minimax
    if (depth == 0) {
        return GetComboEval(board, playerValue);
    }
}
```

Khi áp dụng hàm đánh giá ấy vào thuật toán Minimax, với độ sâu tìm kiếm bằng 2, các nước đi chương trình tìm được đã thỏa yêu cầu ta đặt ra: các nước đi đều hướng đến việc tạo combo với độ dài càng cao càng tốt, và với số lượng càng nhiều càng tốt. Đồng thời, trong hàm đánh giá, ta cũng xét đến những combo mà đối thủ tạo nên, và trừ những số điểm đó đi khỏi kết quả đánh giá cuối cùng. Điều này giúp chương trình ngoài việc tìm những nước đi tạo combo mang lại lợi thế cao nhất, việc chặn combo của đối phương cũng được xem xét là những lựa chọn tốt.

4.1.11.3. Xử lí nước đi đầu tiên

Đối với trường hợp AI thực hiện nước đi đầu tiên, vì hiện không có bất kì nước đi nào trên bàn cờ để dựa trên mà đánh giá, ta sẽ cài đặt hàm `GetFirstMove` để xử lý việc này. Vì phần lớn các trận đấu Caro đều có nước đi nằm ở khoảng giữa của bàn cờ, vì khu vực này có nhiều khoảng không nhất, nên hàm `GetFirstMove` sẽ trả về một nước đi trong phạm vi đó.

Implementation

```
inline GameAction::Point GetFirstMove()
{
    srand(time(NULL));
    // Tạo nước đi nằm ở khoảng giữa bàn cờ
    short row = Constants::BOARD_SIZE / 2 - 2 + (rand() % 3);
    short col = Constants::BOARD_SIZE / 2 - 2 + (rand() % 3);
    return {row, col};
}
```

Usage

```
{
    // Lượt đầu tiên là của AI
    if (isAIFirst) {
        aiMove = GetFirstMove();
        /*...*/
    }
}
```

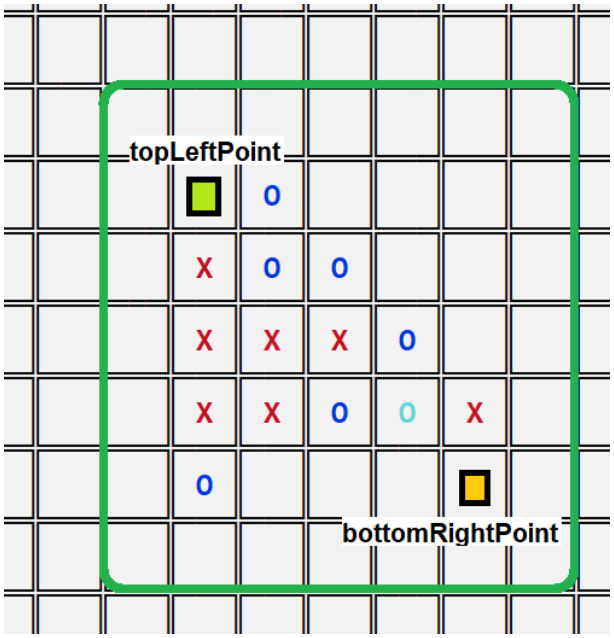
4.1.11.4. Cải thiện tốc độ

Một vấn đề có thể thấy rõ với AI hiện tại là thời gian tìm kiếm còn dài. Với độ sâu tìm kiếm bằng 3, trung bình mỗi lần tìm kiếm của AI mất khoảng 30 giây, với số lượng bàn cờ truy xét là hơn 100000 bàn cờ. Nếu giữ nguyên chương trình như vậy, người chơi sẽ dễ dàng cảm thấy chán nản khi phải đợi lượt đánh của AI. Ta cần phải có những biện pháp cải thiện tốc độ xử lý.

4.1.11.4.1. Giới hạn phạm vi tìm kiếm

Hiện giờ, thuật toán Minimax đang xét tất cả nước đi có thể thực hiện của toàn bộ bàn cờ, nhưng việc làm này rất tốn kém và mất thời gian. Trong cờ Caro, các nước đi thường sẽ nằm liền kề nhau, tạo nên một phạm vi mà phần lớn các quân cờ đều nằm bên trong. Lí do là vì những nước đi tách biệt quá xa khỏi phạm vi ấy thường là những nước đi không tốt, không mang lại lợi thế cho người chơi. Dựa vào việc này, ta sẽ giới hạn phạm vi tìm kiếm của thuật toán Minimax để có thể giảm thời gian xử lí. Gọi `topLeftPoint` là vị trí có row bằng row của quân cờ cao nhất, col bằng col của quân cờ trái cùng nhất, `bottomRightPoint` là vị

trí có row bằng row của quân cờ thấp nhất, col bằng col của quân cờ phải cùng nhất. Khi ấy, phạm vi tìm kiếm của chúng ta sẽ là một hình chữ nhật như **hình 4**.

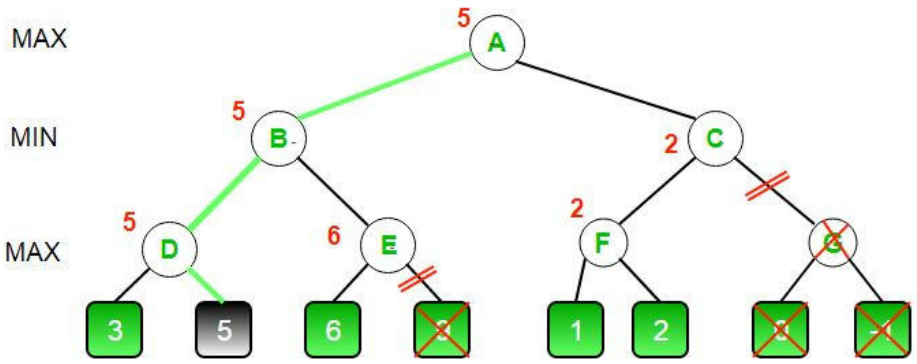


Hình 4: Giới hạn tìm kiếm minh họa qua khung màu xanh

Với sự cải thiện này, trong một trận đấu mà các quân cờ nằm gần nhau, thời gian xử lý sẽ được rút ngắn đi. Tuy nhiên, sự rút ngắn ấy không đáng kể, chưa kể người chơi có thể thực hiện hai nước đi ở hai góc đối của bàn cờ, từ đó khiến cho việc giới hạn phạm vi trở nên vô ích.

4.1.11.4.2. Alpha-Beta pruning

Một phương pháp hiệu quả để tăng tốc thuật toán Minimax là kĩ thuật **Alpha-Beta pruning**[10]. Ý tưởng là nếu nước đi đang xét có thể được chứng minh là tệ hơn một nước đi đã tìm được trước đó, thì ta sẽ ngừng truy xét nước đi này. Hay nói cách khác, ta sẽ "tỉa" những đoạn kiểm tra không cần thiết khỏi quá trình truy xét. Việc này giúp giảm số lượng nước đi phải kiểm tra, từ đó tăng tốc độ xử lý của thuật toán.



Hình 5: Quá trình cắt tỉa thông qua Alpha-Beta pruning

4.1.11.4.3. Sắp xếp nước đi tìm kiếm

Phương pháp Alpha-Beta pruning chỉ thực sự phát huy hiệu quả khi ta kiểm tra những nước đi tốt trước[11]. Hiện giờ, thuật toán chỉ kiểm tra từng nước đi theo thứ tự tuần tự trên bàn cờ (từ trái sang phải, từ trên xuống dưới). Ta cần truy xét những nước đi theo một trật tự sao cho nước đi tốt được xét trước và nước đi xấu được xét sau. Thế nhưng, làm thế nào để biết một nước đi là nước đi tốt? Không phải chúng ta thực hiện thuật toán Minimax cũng là để tìm nước đi đó hay sao? Trong cờ vua, những nước đi như cho một quân có giá trị thấp ăn một quân có giá trị cao hơn có thể nói là một nước đi tốt, mặc dù ta không biết nó có ảnh hưởng lâu dài đến lúc sau hay không. Dựa vào ý tưởng đó, ta có thể viết một hàm phỏng đoán một nước đi có phải là nước đi tốt hay không. Sau đó, ta sắp xếp các nước đi có thể thực hiện vào một danh sách theo thứ tự **nước đi tốt nhất đến nước đi xấu nhất**, và cho thuật toán Minimax truy xét danh sách ấy. Hàm đảm nhiệm việc lập nên danh sách nước đi thỏa yêu cầu trên là hàm `GetMoveList`. Ý tưởng đánh giá một nước đi tốt trong cờ Caro có thể được miêu tả như sau:

- Xét vị trí của một nước đi, ta kiểm tra xem trên một phương nhất định, còn thiếu bao nhiêu quân cờ để tạo nên một chuỗi 5 nước đồng chất.
- Nếu số quân cờ thiếu càng ít, thì số điểm gán cho nước đi đang xét sẽ càng cao, và ngược lại.

- Ta thực hiện việc kiểm tra đối với cả 8 phía xung quanh nước đi ấy, và lấy tổng của từng kết quả.
- Tiếp đến, ta kiểm tra trên một phía nhất định có bao nhiêu quân cờ của đối phương.
- Nếu số quân cờ của đối phương càng nhiều, số thì điểm gán cho nước đang xét sẽ càng cao, và ngược lại.
- Ta thực hiện việc kiểm tra đối với cả 8 phía xung quanh nước đi ấy, và lấy tổng của từng kết quả.
- Kết quả cuối cùng gán cho nước đi sẽ là tổng của việc xét quân cờ đồng chất và xét quân cờ đối phương.

Lí do tại sao không đánh giá điểm của nước đi theo cách đánh giá bàn cờ trong hàm `GetComboEval` là vì một nước đi tốt không nhất thiết phải là nước tạo nên chuỗi quân cờ đồng chất dài nhất. Nếu đối phương có 4 quân cờ liền kề nhau bị chặn một đầu, thì việc ta chặn đầu còn lại của chuỗi quân cờ ấy là một nước đi rất tốt.

Interface

```
MoveQueue GetMoveList(
    short rowLowerLimit,
    short rowUpperLimit,
    short colLowerLimit,
    short colUpperLimit,
    short moveCount,
    GameAction::Board& board,
    short playerValue
)
```

Parameters

- rowLowerLimit: giới hạn tìm kiếm dưới của dòng
- rowUpperLimit: giới hạn tìm kiếm trên của dòng
- colLowerLimit: giới hạn tìm kiếm trái của cột
- colUpperLimit: giới hạn tìm kiếm phải của cột
- moveCount: số nước đi đã thực hiện

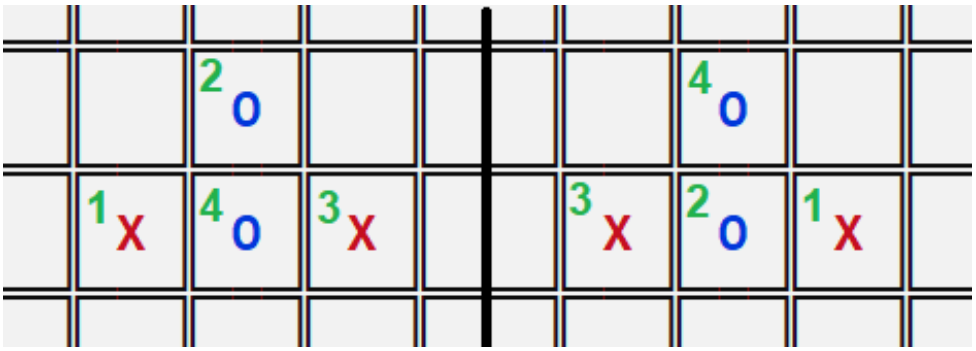
- board: bàn cờ hiện tại
- playerValue: giá trị người chơi đang Xét

Usage

```
{
    MoveQueue moveList = GetMoveList(
        rowLowerLimit,
        rowUpperLimit,
        colLowerLimit,
        colUpperLimit,
        moveCount,
        board,
        currentPlayer);
    while (!moveList.empty()) {
        GameAction::Point move = moveList.top();
        /* Thực hiện Minimax với move...*/
    }
}
```

4.1.11.4.4. Transposition table (Bảng hoán vị)

Trong quá trình tìm kiếm của thuật toán Minimax, sẽ có nhiều trường hợp một trạng thái bàn cờ bị lặp lại theo thứ tự nước đi khác nhau. Việc này sẽ gây lãng phí thời gian truy xét, vì ta đang thực hiện lại phép tính đã có kết quả từ trước.



Hình 6: Hai thứ tự nước đi khác nhau cùng đạt một bàn cờ

Để khắc phục được việc này, ta cần một bảng lưu trữ những kết quả đánh giá có được của các bàn cờ đã xét, để khi gặp lại những bàn cờ ấy, ta trả về giá trị lưu trong bảng, từ đó tránh việc phải lặp lại phép tính. Một bảng lưu trữ như vậy được gọi là **bảng hoán vị** (Transposition table)[12]. Để thực hiện yêu

cầu ấy, ta sử dụng cấu trúc dữ liệu `unordered_map`[13] trong C++. Cấu trúc dữ liệu này lưu trữ dữ liệu theo hình thức **key-value pair**, với mỗi key là một giá trị đơn nhất. Dựa vào tính chất ấy, giả sử key trong trường hợp này là một bàn cờ nhất định, thì value lúc này sẽ là kết quả đánh giá của bàn cờ ấy. Tuy nhiên, bàn cờ trong chương trình được lưu dưới dạng một mảng hai chiều, `vector<vector<short>>`, `unordered_map` không hỗ trợ key với cấu trúc dữ liệu ấy. Vì vậy, ta cần mã hóa bàn cờ thành một giá trị mà có thể sử dụng để làm key. Việc này có thể được thực hiện thông qua kĩ thuật **Zobrist Hashing**[14]. Kĩ thuật này giúp chúng ta chuyển hóa một bàn cờ hai chiều thành một con số đơn nhất, từ đó có thể sử dụng con số ấy làm key cho bảng lưu. Những con bot cờ vua, cờ vây cũng sử dụng kĩ thuật này để mã hóa bàn cờ[15].

4.1.11.4.5. So sánh tốc độ

4.1.11.5. Phân độ khó

Sau khi áp dụng những kĩ thuật để tối ưu hóa thuật toán, độ sâu tối đa mà chương trình có thể thực hiện trong một khoảng thời gian hợp lý là 3. Vì vậy, ta có thể phân độ khó của chế độ đánh với máy với những độ khó:

- **Dễ**: chiều sâu 1
- **Trung bình**: chiều sâu 2
- **Khó**: chiều sâu 3

Với độ sâu bằng 1, thuật toán chỉ kiểm tra tất cả nước đi của 1 lượt duy nhất, vì vậy, tuy các nước đi vẫn sẽ cố gắng tạo nên các combo dẫn đến chuỗi thắng, máy không thể nhìn trước được những đòn tấn công do đối phương gây nên. Với độ sâu bằng 2, thuật toán sẽ có thể kiểm tra thêm những nước đi của đối phương, vì vậy, máy có thể ngăn chặn những mối nguy do đối phương gây nên, như là những nước tạo nên combo 3, combo 4. Với độ sâu bằng 3, không chỉ gây khó dễ cho đối phương qua việc phòng thủ, vì 2 trong 3 lượt kiểm trà là lượt của máy, nên thuật toán sẽ tìm được nhiều nước đi tấn công hơn, và trong cờ Caro, người chơi có thế chủ động thường sẽ có lợi thế cao hơn.

4.1.11.6. Chức năng "Gợi ý"

Ngoài chế độ đánh với máy, ta có thể tận dụng tốc độ xử lý nhanh của AI vào một chức năng khác của trò chơi, đó là chức năng **"Gợi ý"**. Thay vì cố định giá trị người chơi như trong chế độ đánh với máy (người là người chơi X, máy là người chơi O), mỗi khi đến lượt đánh của một người chơi nào đó, khi sử dụng chức năng "Gợi ý", ta sẽ gán giá trị người chơi AI là người chơi hiện tại, từ đó, tìm ra một nước đi tốt cho người chơi ấy.

Implementation

```
GameAction::Point GameScreenAction::GetHintMove(
    GameAction::Board& board,
    short moveCount,
    bool isPlayerOneTurn,
    AI ai
)
{
    if (isPlayerOneTurn) {
        ai.PLAYER_AI = Constants::PLAYER_ONE.value;
        ai.PLAYER_HUMAN = Constants::PLAYER_TWO.value;
    } else {
        ai.PLAYER_AI = Constants::PLAYER_TWO.value;
        ai.PLAYER_HUMAN = Constants::PLAYER_ONE.value;
    }
    ai.SetDifficulty(AI::AI_DIFFICULTY_HARD);
    if (moveCount == 0) return ai.GetFirstMove();
    return ai.GetBestMove(board, moveCount);
}
```

Usage

```
{
    // Thực hiện chức năng Gợi ý
    if (keyPressed == "H") {
        hintMove = GameScreenAction::GetHintMove(
            board,
            moveCount,
            isPlayerOneTurn,
            ai
        );
        /*...*/
    }
}
```

4.1.11.7. Những mặt cần cải thiện

4.2. Giao diện

4.2.1. Màn hình chính

4.2.2. Cài đặt

Thông

4.2.3. Các màn hình lưu, tải game và replay

Thông

4.2.4. Màn hình trò chơi chính

Vũ

4.2.5. Các màn hình khác

Chương 5.

Đánh giá thành viên

Chương 6.

Kết luận

6.1. Kết quả đạt được

6.1.1. Ưu điểm của trò chơi

- Có thể thêm nhiều ngôn ngữ và theme vào trò chơi
- Có nhạc hay, hiệu ứng sống động
- Có chế độ tính thời gian
- AI chạy tương đối tốt, đánh nhanh
- Lối chơi đa dạng
- Có nhiều nhân vật ngộ nghĩnh
- Có nhiều tính năng hỗ trợ khi chơi game
- Có thể xem lại trận đấu đã chơi
- Màn hình save/load có khả năng tương tác tốt
- Hướng dẫn dễ hiểu, có gợi ý ở mỗi màn hình

6.1.2. Khuyết điểm của trò chơi

-

6.2. Các khó khăn gặp phải

- Một vài thành viên không có kinh nghiệm sử dụng git và GitHub
- Khó khăn khi lập trình đa luồng do không có kinh nghiệm
- Màn hình terminal vẽ các kí tự chậm
- Khó khăn trong việc thiết kế AI do có nhiều kiến thức mới, lạ
- Chưa có kinh nghiệm trong việc thiết kế giao diện, viết ứng dụng có giao diện

6.3. Những gì đã học được

- Cách làm việc nhóm với git và GitHub

- Cách sử dụng các tính năng mới của C++ 20
- Cách sử dụng các tính năng liên quan tới đo hiệu năng, format code và debug trong Visual Studio
- Cách làm việc nhóm hiệu quả
- Cách lên kế hoạch, phân chia công việc
- Cách lập trình hướng đối tượng, lập trình đa luồng cơ bản

6.4. Các kinh nghiệm rút ra

- Khi code, nên tách nhỏ các hàm ra, không nên viết hàm dài
- Code xong một hàm phải kiểm tra kĩ, tránh phát sinh lỗi về sau
- Không nên viết code mà không thiết kế trước
- Nên viết code theo một quy chuẩn nhất định và đồng bộ trong 1 dự án
- Nên viết code có tính tái sử dụng cao
- Cần có kế hoạch và phân chia công việc rõ ràng
- Code xong phải có người review lại

6.5. Lí do hoàn thành mục tiêu

Nguyên

6.6. Hướng phát triển ứng dụng

- Có thể chơi 2 người qua mạng lan
- Thêm nhiều ngôn ngữ mới
- Thêm nhiều chủ đề hơn
- Hiện lợi thế của 2 bên
- Đưa game lên nhiều nền tảng khác

Tài liệu tham khảo

[1] "Playsound function." https://learn.microsoft.com/en-us/previous-versions/dd743680(v=vs.85)

[2] "Mci." https://learn.microsoft.com/vi-vn/windows/win32/multimedia/mci

[3] "Mcisendstring function." https://learn.microsoft.com/en-us/previous-versions//dd757161(v=vs.85)

[4] “Navigating with compose.” https://developer.android.com/jetpack/compose/navigation

[5] “Android jetpack.” https://developer.android.com/jetpack

[6] “Filesystem library.” https://en.cppreference.com/w/cpp/filesystem

[7] “Std::thread.” https://en.cppreference.com/w/cpp/thread/thread

[8] “Std::mutex.” https://en.cppreference.com/w/cpp/thread/mutex

[9] “Minimax.” https://vi.wikipedia.org/wiki/Minimax

[10] “Alpha-beta pruning.” https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[11] “Move ordering.” https://www.chessprogramming.org/Move_Ordering

[12] “Tranposition table.” https://en.wikipedia.org/wiki/Transposition_table

[13] “Unordered map.” https://en.cppreference.com/w/cpp/container/unordered_map

[14] “Zobrist hashing.” https://en.wikipedia.org/wiki/Zobrist_hashing

[15] “Zobrist hashing in chess.” https://www.chessprogramming.org/Zobrist_Hashing