

TRƯỜNG ĐẠI HỌC SÀI GÒN
KHOA CÔNG NGHỆ THÔNG TIN



PHÁT TRIỂN PHẦN MỀM MÃ NGUỒN MỞ

Xây dựng ứng dụng sử dụng Django

Mạng xã hội Open World

GVHD: Từ Lăng Phiêu

SV:

Đỗ Minh Quân - 2003

Lê Ngọc Giàu - 3121410169

Lê Tấn Minh Toàn - 3121560092

Tăng Quốc Tuấn - 2003

TP. HỒ CHÍ MINH, THÁNG 2/2024

Mục lục

1	Phần Mở Đầu	2
2	Tổng Quan Đề Tài	2
2.1	Lý do chọn đề tài:	2
2.2	Đối Tượng Ứng Dụng Cần Hướng Tới:	2
3	Chức năng người dùng	4
3.1	Đăng ký và đăng nhập	4
3.2	Quản lý hồ sơ cá nhân	7
3.3	Gửi và nhận yêu cầu kết bạn	12
3.4	Tìm kiếm người dùng khác	13
4	Nhắn tin	15
4.1	Tạo cuộc trò chuyện mới	15
4.2	Gửi và nhận tin nhắn	16
4.3	Gửi hình ảnh và tệp đính kèm	17
4.4	Gọi video call	18
5	Bài đăng và hoạt động mạng xã hội	20
5.1	Xem và tương tác với bài đăng của người dùng khác (like, bình luận)	20
5.2	Chia sẻ bài đăng	21
5.3	Chỉnh sửa nội dung hoặc xóa bài đăng	22
5.4	Theo dõi người dùng khác	23
6	Game	25
6.1	Stack Brick	25
6.2	Caro(Tic Tac Toe)	36
6.3	Game xếp số 2048	45

1 Phần Mở Đầu

Trong thời đại số hóa ngày nay, mạng xã hội không chỉ là nơi để kết nối cá nhân mà còn trở thành một phần không thể thiếu của cuộc sống công việc và xã hội. Tận dụng sức mạnh của công nghệ thông tin và sự phát triển của trí tuệ nhân tạo, chúng ta đã chứng kiến sự xuất hiện của các ứng dụng mạng xã hội tiên tiến, không chỉ giúp người dùng kết nối mà còn mang lại những trải nghiệm độc đáo và hữu ích. Những nền tảng này không chỉ là điểm hẹn của giới trẻ mà còn là công cụ quan trọng cho các tổ chức, doanh nghiệp trong việc tiếp cận khách hàng và xây dựng thương hiệu. Với những tiềm năng không ngừng được khai phá, việc xây dựng phần mềm ứng dụng mạng xã hội không chỉ là một cơ hội mà còn là một nhiệm vụ quan trọng, để mang lại giá trị và tiện ích cho cộng đồng mạng và xã hội.

2 Tổng Quan Đề Tài

2.1 Lý do chọn đề tài:

Mạng xã hội đã trở thành một phần không thể thiếu trong cuộc sống hiện đại, là nơi kết nối con người, chia sẻ thông tin và giải trí. Nhu cầu sử dụng mạng xã hội ngày càng tăng cao, thúc đẩy sự phát triển của các ứng dụng mới. Việc xây dựng phần mềm ứng dụng mạng xã hội tương tự Facebook là một đề tài đầy tiềm năng với nhiều lý do sau:

Thứ nhất, mạng xã hội đáp ứng nhu cầu kết nối và chia sẻ thông tin của con người. Facebook đang thống trị thị trường, nhưng vẫn còn nhiều tiềm năng cho các ứng dụng mới với những tính năng độc đáo, thu hút lượng người dùng lớn.

Thứ hai, sự phát triển của AI và nhận dạng giọng nói mở ra những trải nghiệm mới mẻ cho người dùng mạng xã hội. Ví dụ, người dùng có thể sử dụng giọng nói để điều khiển ứng dụng, tìm kiếm thông tin, tương tác với bạn bè.

Thứ ba, mạng xã hội là ngành công nghiệp có tiềm năng sinh lời cao thông qua quảng cáo, bán dữ liệu người dùng và cung cấp dịch vụ trả phí.

Thứ tư, xây dựng ứng dụng mạng xã hội là cơ hội để học hỏi và trau dồi các kỹ năng lập trình, thiết kế, quản lý dự án và kinh doanh.

Hơn nữa, mạng xã hội có thể đóng góp tích cực cho cộng đồng bằng cách kết nối mọi người, thúc đẩy chia sẻ thông tin và ý tưởng, hỗ trợ các hoạt động chung.

Nhìn chung, xây dựng phần mềm ứng dụng mạng xã hội tương tự Facebook là đề tài hấp dẫn, đầy tiềm năng, hứa hẹn mang lại nhiều lợi ích cho cá nhân và cộng đồng.

2.2 Đối Tượng Ứng Dụng Cần Hướng Tới:

1. Cá nhân:

- **Người dùng trẻ:** Nhóm này ưa chuộng kết nối bạn bè, chia sẻ thông tin, giải trí và thể hiện bản thân.



- **Người dùng trung niên:** Nhóm này quan tâm cập nhật tin tức, kết nối gia đình, bạn bè và tham gia hoạt động cộng đồng.
- **Người dùng lớn tuổi:** Nhóm này muốn kết nối con cháu, cập nhật thông tin sức khỏe và tham gia giải trí phù hợp.

2. Doanh nghiệp:

- **Doanh nghiệp nhỏ và vừa:** Nền tảng quảng bá sản phẩm, dịch vụ, kết nối khách hàng và xây dựng thương hiệu.
- **Doanh nghiệp lớn:** Kênh tiếp thị hiệu quả, thu thập dữ liệu khách hàng và quản lý uy tín thương hiệu.

3. Tổ chức:

- **Tổ chức phi lợi nhuận:** Nền tảng huy động vốn, kết nối nhà tài trợ và quảng bá hoạt động.
- **Cơ quan chính phủ:** Kênh giao tiếp với người dân, cung cấp thông tin chính sách và tương tác cử tri.

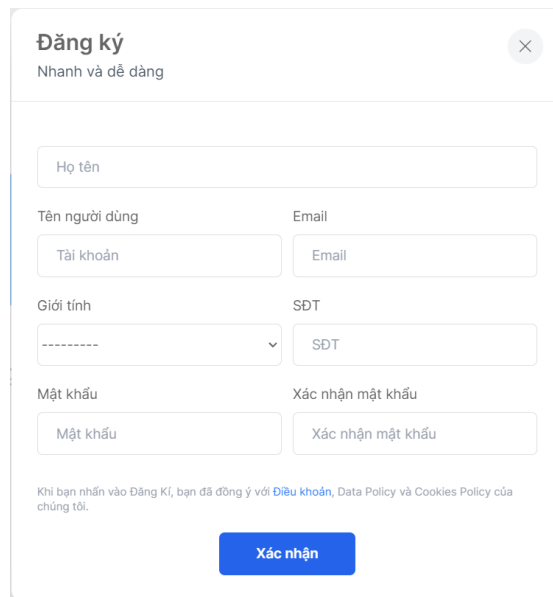
Bằng cách đáp ứng nhu cầu đa dạng của các đối tượng mục tiêu, ứng dụng mạng xã hội có thể thu hút lượng người dùng lớn, tạo dựng cộng đồng sôi động và mang lại lợi ích cho cả người dùng và doanh nghiệp.

3 Chức năng người dùng

3.1 Đăng ký và đăng nhập

Đăng ký

Để người dùng có thể đăng ký vào mạng xã hội của chúng em xây dựng thì người dùng trước hết cần tạo một tài khoản cho riêng mình, do đó chúng em có tạo một form đăng ký (signup pop-up) và xử lý các thông tin người dùng nhập vào để tạo một tài khoản mới. Dưới đây là các bước để triển khai chức năng đăng ký: Trên server, tạo một view để xử lý yêu cầu đăng ký. Trong



Hình 1: Modal để người dùng đăng kí

view này, kiểm tra thông tin được gửi từ form, và tạo một tài khoản người dùng mới trong cơ sở dữ liệu nếu thông tin hợp lệ, khi tạo tài khoản thành công sẽ tự động chuyển hướng người dùng dùng đến trang chính của ứng dụng

```
class UserRegistrationForm(UserCreationForm):
    full_name = forms.CharField(widget=forms.TextInput(attrs={'class': '', 'id': '', 'placeholder': 'Họ tên'}),
                                max_length=100, required=True, error_messages={'required': 'Vui lòng nhập họ tên'})
    username = forms.CharField(widget=forms.TextInput(attrs={'class': '', 'id': '', 'placeholder': 'Tên người dùng'}),
                                max_length=20, required=True, error_messages={'required': 'Vui lòng nhập tên người dùng'})
    phone = forms.CharField(widget=forms.TextInput(attrs={'class': '', 'id': '', 'placeholder': 'Số điện thoại'}),
                            max_length=10, required=True, error_messages={'required': 'Vui lòng nhập số điện thoại'})
    email = forms.EmailField(widget=forms.TextInput(attrs={'class': '', 'id': '', 'placeholder': 'Email'}),
                             required=True, error_messages={'required': 'Vui lòng nhập email'})
    password1 = forms.CharField(widget=forms.PasswordInput(attrs={'id': '', 'placeholder': 'Mật khẩu'}),
                                required=True, error_messages={'required': 'Vui lòng nhập mật khẩu'})
    password2 = forms.CharField(widget=forms.PasswordInput(attrs={'id': '', 'placeholder': 'Xác nhận mật khẩu'}),
                                required=True, error_messages={'required': 'Vui lòng nhập xác nhận mật khẩu'})
    gender = forms.CharField(widget=forms.TextInput(attrs={'class': 'with-border', 'id': '', 'placeholder': 'Giới tính'}))

    class Meta:
        model = User
        fields = ('full_name', 'username', 'email', 'password1', 'password2', 'phone', 'gender')

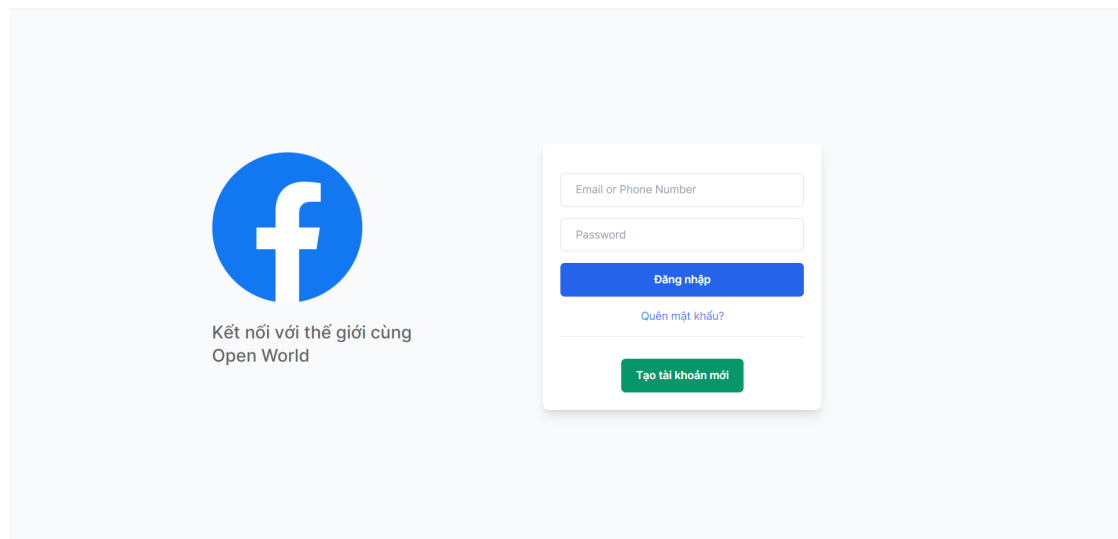
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for visible in self.visible_fields():
            visible.field.widget.attrs['class'] = 'with-border'
            visible.field.widget.attrs['placeholder'] = visible.field.label
```

Hình 2: Form để người dùng đăng kí



Đăng nhập

Sau khi người dùng đã có tài khoản, họ cần đăng nhập để truy cập vào mạng xã hội của chúng. Chúng em đã tạo trang chủ chứa một form đăng nhập (login form) để người dùng nhập email và mật khẩu, sau đó sẽ gửi yêu cầu đăng nhập (login request) từ form đến server.

A mockup of a Facebook login interface. On the left, there is a large blue Facebook 'f' logo. Below it, the text 'Kết nối với thế giới cùng Open World' is displayed. To the right of the logo is a white login form with a subtle shadow. The form contains two input fields: 'Email or Phone Number' and 'Password'. Below these fields is a blue button labeled 'Đăng nhập'. Underneath the button is a link that says 'Quên mật khẩu?'. At the bottom of the form is a green button labeled 'Tạo tài khoản mới'.

Hình 3: Giao diện đăng nhập

Xử lý phía server

- Trên server, tạo một view để xử lý yêu cầu đăng nhập.
- Trong view này, kiểm tra thông tin đăng nhập với thông tin trong cơ sở dữ liệu. Nếu thông tin chính xác, đánh dấu người dùng là đã đăng nhập và chuyển hướng họ đến trang chính của mạng xã hội. Nếu không, thông báo lỗi và yêu cầu nhập lại.

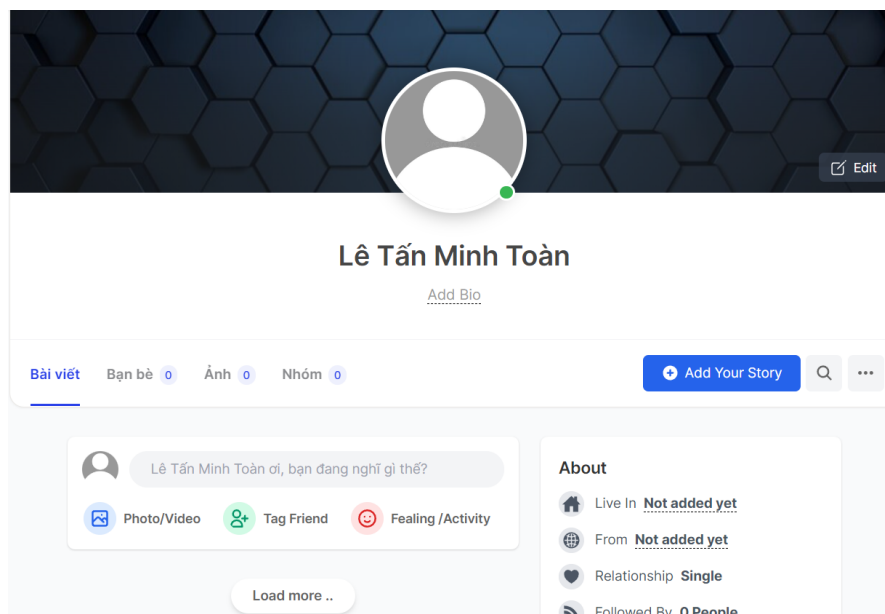
```
def LoginView(request):  
    # if request.user.is_authenticated:  
    #     return redirect('core:feed')  
  
    if request.method == 'POST':  
        email = request.POST.get('email')  
        password = request.POST.get('password')  
        try:  
            user = User.objects.get(email=email)  
            user = authenticate(request, email=email, password=password)  
            if user is not None:  
                login(request, user)  
                messages.success(request, "You are Logged In")  
                return redirect('core:feed')  
            else:  
                messages.error(request, 'Username or password does not exist.')  
        except:  
            messages.error(request, 'User does not exist')  
    return HttpResponseRedirect("/")
```

Hình 4: Code xử lý phía server

3.2 Quản lý hồ sơ cá nhân

Trang cá nhân

Trang cá nhân hiển thị thông tin cá nhân của người dùng, bao gồm avatar, tên, tiểu sử, danh sách bạn bè, bài đăng và các hoạt động khác. Thông tin này thường được lấy từ cơ sở dữ liệu và hiển thị theo giao diện người dùng. Trang cá nhân sẽ hiển thị tất cả các thông tin cơ bản của người dùng, như bài viết mà người dùng đăng lên, bạn bè của người dùng các nhóm mà người dùng đã tham gia



Hình 5: Giao diện trang cá nhân

```
@login_required
def my_profile(request):
    profile = request.user.profile
    posts = Post.objects.filter(active=True, user=request.user)
    groups = Group.objects.filter(active=True, user=request.user)

    context = {
        "posts": posts,
        "groups": groups,
        "profile": profile,
    }
    return render(request, "userauths/my-profile.html", context)
```

Hình 6: Code xử lý phía server

Xử lý phía server

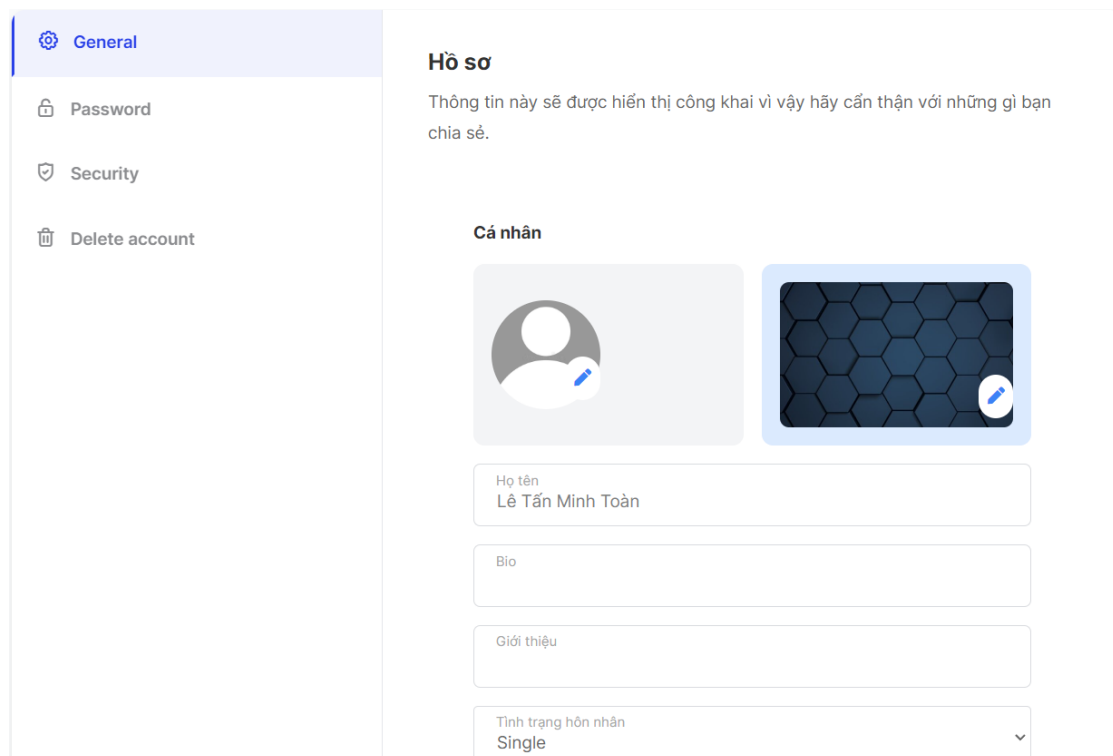
- Em sử dụng decorator `@login_required` được sử dụng để bảo vệ các view chỉ dành cho người dùng đã đăng nhập. Nó đảm bảo rằng chỉ những người dùng đã xác thực mới có thể truy cập vào các trang hoặc chức năng được bảo vệ
- Khi một người dùng cố gắng truy cập vào `my_profile`, Django sẽ kiểm tra xem họ đã đăng nhập hay chưa. Nếu họ đã đăng nhập, họ sẽ được chuyển đến trang được yêu cầu. Ngược lại, nếu họ chưa đăng nhập, họ sẽ được chuyển đến trang đăng nhập mặc định của Django. Sau khi người dùng đăng nhập thành công, họ sẽ được chuyển hướng trở lại trang đã yêu cầu.

Nếu người dùng đã đăng nhập, server sẽ truy xuất cơ sở dữ liệu để lấy tất cả các bài viết, nhóm, thông tin cơ bản của người dùng để hiển thị ra

Cập nhật thông tin cá nhân

Chức năng "Cập nhật thông tin cá nhân" trong Django Python là một phần quan trọng của bất kỳ ứng dụng web nào có tính đăng nhập và tài khoản người dùng. Chức năng này cho phép người dùng cập nhật thông tin cá nhân của họ như tên, địa chỉ email, mật khẩu, ảnh đại diện và các thông tin khác một cách dễ dàng và linh hoạt.

Trang cập nhật thông tin cá nhân cần cung cấp một form cho phép người dùng cập nhật thông tin cá nhân của họ. Form này thường bao gồm các trường như tên người dùng, email, ảnh đại diện và các trường khác tùy thuộc vào yêu cầu cụ thể của ứng dụng



Hình 7: Giao diện trang Quản lý hồ sơ cá nhân (1)



<div>Security</div> <div>Delete account</div>	<div>Liên lạc</div> <div>Số điện thoại 0923326749</div> <div>Email letanminhtoan2505@gmail.com</div> <div>Địa chỉ</div> <div>Địa chỉ</div> <div>Thành phố</div> <div>State</div> <div>Đất nước</div>
---	--

Hình 8: Giao diện trang Quản lý hồ sơ cá nhân (2)

	<div>Socila URLs</div> <div>Instagram https://instagram.com/</div> <div>WhatsApp +123 (456) 789</div> <div>Save Changes</div>
--	---

Hình 9: Giao diện trang Quản lý hồ sơ cá nhân (3)

Phần "General", cho phép người dùng xem và chỉnh sửa thông tin chung của hồ sơ cá nhân như tên, địa chỉ email, avatar, v.v.

- Hiển thị thông tin hiện tại của người dùng.
- Cho phép chỉnh sửa thông tin và lưu thay đổi.
- Cung cấp giao diện để tải lên hoặc cập nhật hình đại diện.

```
@login_required
def profile_update(request):
    if request.method == "POST":
        p_form = ProfileUpdateForm(request.POST, request.FILES, instance=request.user.profile)
        u_form = UserUpdateForm(request.POST, instance=request.user)

        if p_form.is_valid() and u_form.is_valid():
            p_form.save()
            u_form.save()
            messages.success(request, "Hồ sơ đã được cập nhật thành công")
            return redirect('userauths:profile-update')
    else:
        p_form = ProfileUpdateForm(instance=request.user.profile)
        u_form = UserUpdateForm(instance=request.user)

    context = {
        'p_form': p_form,
        'u_form': u_form,
    }
    return render(request, 'userauths/profile-update.html', context)
```

Hình 10: Code xử lý server cho phần cập nhật mục general

Đoạn mã này là một view function trong Django, được sử dụng để xử lý yêu cầu cập nhật hồ sơ cá nhân của người dùng. Khi người dùng gửi một yêu cầu cập nhật thông qua phương thức POST, hàm này sẽ xử lý dữ liệu được gửi từ biểu mẫu và lưu các thay đổi vào cơ sở dữ liệu.

- Đầu tiên, hàm kiểm tra xem yêu cầu được gửi từ người dùng là phương thức POST hay không.
- Nếu là POST, hàm sẽ tạo hai biểu mẫu: một để cập nhật thông tin hồ sơ cá nhân và một để cập nhật thông tin người dùng.
- Sau đó, nó kiểm tra tính hợp lệ của cả hai biểu mẫu. Nếu hợp lệ, dữ liệu sẽ được lưu và người dùng sẽ nhận được một thông báo xác nhận.
- Nếu không phải là POST, tức là người dùng đang truy cập trang cập nhật hồ sơ cá nhân, hàm sẽ tạo các biểu mẫu với dữ liệu hiện tại của người dùng để hiển thị trên trang.
- Sau đó, nó kiểm tra tính hợp lệ của cả hai biểu mẫu. Nếu hợp lệ, dữ liệu sẽ được lưu và người dùng sẽ nhận được một thông báo xác nhận.
- Cuối cùng, hàm render một template với các biểu mẫu tương ứng để người dùng có thể cập nhật thông tin của mình.

```
class ProfileUpdateForm(forms.ModelForm):
    image = ImageField(widget=FileInput)

    class Meta:
        model = Profile
        fields = [
            'cover_image',
            'image',
            'full_name',
            'bio',
            'about_me',
            'phone',
            'gender',
            'relationship',
            'friends_visibility',
            'country',
            'city',
            'state',
            'address',
            'working_at',
            'instagram',
            'whatsApp',
        ]

class UserUpdateForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ['username', 'email']
```

Hình 11: Tạo form cho người dùng cập nhật hồ sơ

ProfileUpdateForm

- Biểu mẫu này được sử dụng để cập nhật thông tin hồ sơ cá nhân của người dùng.
- Trong biểu mẫu này, có các trường như hình ảnh đại diện, tên đầy đủ, tiểu sử, số điện thoại, giới tính, mối quan hệ, v.v.
- Người dùng có thể tải lên hình ảnh đại diện mới hoặc cập nhật thông tin cá nhân của họ thông qua biểu mẫu này.

UserUpdateForm

- Biểu mẫu này được sử dụng để cập nhật thông tin người dùng, như tên người dùng và địa chỉ email.
- Chỉ có hai trường được hiển thị trong biểu mẫu này, là 'username' và 'email', để người dùng có thể cập nhật thông tin này một cách dễ dàng.



3.3 Gửi và nhận yêu cầu kết bạn



3.4 Tìm kiếm người dùng khác



- [Phần code R](#)



4 Nhấn tin

4.1 Tạo cuộc trò chuyện mới



4.2 Gửi và nhận tin nhắn



4.3 Gửi hình ảnh và tệp đính kèm



4.4 Gọi video call



- [Phần code R](#)



5 Bài đăng và hoạt động mạng xã hội

5.1 Xem và tương tác với bài đăng của người dùng khác (like, bình luận)



5.2 Chia sẻ bài đăng



5.3 Chỉnh sửa nội dung hoặc xóa bài đăng



5.4 Theo dõi người dùng khác



- [Phần code R](#)

6 Game

6.1 Stack Brick

Giới Thiệu Ý Tưởng:

Trò chơi này là một trò chơi xếp hộp, bạn phải cố gắng xây dựng một cấu trúc hình tháp cao bằng cách đặt các hộp lên nhau một cách chính xác. Mỗi lớp của hộp mới được thêm vào cấu trúc cần phải xếp chồng lên lớp trước đó một cách hoàn hảo. Nếu không, phần vượt ra ngoài sẽ bị cắt bớt và rơi xuống.

Người chơi có thể chọn chơi thủ công hoặc tự động. Trong chế độ tự động, một "máy giả lập" được lập trình sẵn sẽ đặt hộp với độ chính xác cố định. Người chơi cũng có thể điều khiển bằng cách nhấn chuột hoặc phím cách để đặt hộp.

Mục tiêu của trò chơi là xây dựng cấu trúc hình tháp càng cao càng tốt mà không làm rơi hộp ra ngoài cấu trúc. Nếu một hộp bị rơi, trò chơi kết thúc và điểm số của bạn được hiển thị lên màn hình.

Các công nghệ được sử dụng:

HTML, CSS, JavaScript (thư viện Three.js và Cannon.js), Python. Để tạo ra môi trường 3D và vật lý. Three.js dùng để render đồ họa còn Cannon.js dùng để mô phỏng vật lý.

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="https://cdn.jsdelivr.net/npm/three@0.124.0/build/three.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/cannon@0.6.2/build/cannon.min.js"></script>
  <style>
    @import url("https://fonts.googleapis.com/css2?family=Montserrat:wght@900&display=swap");
```

Hình 12: Mô tả ảnh

1. Function setRobotPrecision():

Mục đích: Thiết lập độ chính xác của robot(máy giả lập) trong chế độ tự động

Giải thích: Hàm này sẽ thiết lập giá trị robotPrecision là một số ngẫu nhiên nằm trong khoảng từ -0.5 đến 0.5. Giá trị này được sử dụng để xác định độ chính xác của robot khi tự động đặt các hộp trong trò chơi

```
// Determines how precise the game is on autopilot
function setRobotPrecision() {
  robotPrecision = Math.random() * 1 - 0.5;
}
```

Hình 13: Mô tả ảnh

2. Function `addLayer(x,z,width,depth,direction)`:

Mục đích: Thêm một hộp mới vào cấu trúc và xếp chồng lên nhau.

Giải thích: Hàm này tạo ra một lớp hộp mới và thêm vào cấu trúc ở vị trí và hướng chỉ định cao hơn, với kích thước được xác định bởi `width` và `depth`.

Ý Tưởng:

- Tính toán vị trí `y` của lớp mới dựa trên độ cao của mỗi hộp và số lượng lớp đã có trong stack. Lớp mới sẽ được thêm vào ở một tầng cao hơn so với các lớp trước đó.
- Gọi hàm `generateBox` để tạo ra hộp mới với các thông số `x`, `y`, `z`, `width`, `depth` cung cấp.
- Thiết lập thuộc tính "direction" cho lớp mới để xác định hướng của nó (có thể là hướng di chuyển hoặc hướng của các thành phần khác).
- Thêm lớp mới vào stack.

```
function addLayer(x, z, width, depth, direction) {  
  const y = boxHeight * stack.length; // Add the new box one layer higher  
  const layer = generateBox(x, y, z, width, depth, false);  
  layer.direction = direction;  
  stack.push(layer);  
}
```

Hình 14: Mô tả ảnh

3. `addOverhang(x, z, width, depth)`:

Mục đích: Tạo ra các phần nhô ra cho các hộp lên trên lớp hiện tại trong stack.

Giải thích: Hàm này tạo ra một phần rơi mới và thêm vào lớp hiện tại của cấu trúc, với vị trí và kích thước xác định bởi `x`, `z`, `width` và `depth`

Ý Tưởng:

- Tính toán vị trí `y` của phần "overhang" mới dựa trên độ cao của mỗi hộp và số lượng lớp đã có trong stack. "Overhang" mới sẽ được thêm vào ở cùng một tầng với lớp cuối cùng trong stack.
- Gọi hàm `generateBox` để tạo ra "overhang" mới với các thông số `x`, `y`, `z`, `width`, `depth` cung cấp.
- Đặt cờ "true" để xác định rằng "overhang" này sẽ được tạo ra như là một phần của lớp hiện tại.
- Thêm "overhang" mới vào danh sách "overhangs".

```
function addOverhang(x, z, width, depth) {  
  const y = boxHeight * (stack.length - 1); // Add the new box one the same layer  
  const overhang = generateBox(x, y, z, width, depth, true);  
  overhangs.push(overhang);  
}
```

Hình 15: Mô tả ảnh

4. generateBox(x, y, z, width, depth, falls):

```
function generateBox(x, y, z, width, depth, falls) {  
  // ThreeJS  
  const geometry = new THREE.BoxGeometry(width, boxHeight, depth);  
  const color = new THREE.Color(`hsl(${30 + stack.length * 4}, 100%, 50%)`);  
  const material = new THREE.MeshLambertMaterial({ color });  
  const mesh = new THREE.Mesh(geometry, material);  
  mesh.position.set(x, y, z);  
  scene.add(mesh);  
  
  // CannonJS  
  const shape = new CANNON.Box(  
    new CANNON.Vec3(width / 2, boxHeight / 2, depth / 2)  
  );  
  let mass = falls ? 5 : 0; |  
  mass *= width / originalBoxSize; // Reduce mass proportionately by size  
  mass *= depth / originalBoxSize; // Reduce mass proportionately by size  
  const body = new CANNON.Body({ mass, shape });  
  body.position.set(x, y, z);  
  world.addBody(body);  
  
  return {  
    threejs: mesh,  
    cannonjs: body,  
    width,  
    depth  
  };  
}
```

Hình 16: Mô tả ảnh

Mục đích: Tạo ra một hộp mới trong cả hai môi trường ThreeJS và CannonJS.

Giải thích: Hàm này tạo một hộp với kích thước và vị trí xác định bởi các tham số truyền vào. Nó tạo ra một đối tượng hộp trong môi trường đồ họa ThreeJS với màu sắc được tính dựa trên số lớp hiện tại của cấu trúc. Đồng thời, nó cũng tạo ra một đối tượng hộp tương ứng trong môi trường vật lý CannonJS, với khối lượng được xác định dựa trên suy đoán của hộp có rơi hay không và kích thước của nó.

Ý tưởng:

1. Tạo hộp trong ThreeJS:

- Sử dụng `THREE.BoxGeometry` để tạo hình hộp với kích thước được xác định bởi `width`, `boxHeight`, và `depth`.
- Tạo một vật liệu (material) cho hộp với màu sắc được tính toán dựa trên số lượng các lớp đã được thêm vào stack trước đó.
- Tạo một mesh (đối tượng mạng lưới) bằng cách kết hợp hình hộp và vật liệu.



- Đặt vị trí của mesh dựa trên các thông số **x**, **y**, và **z**.
- Thêm mesh vào scene để hiển thị trong không gian 3D.

2. Tạo hộp trong CannonJS:

- Sử dụng `CANNON.Box` để tạo hình hộp cho vật lý.
- Xác định khối lượng (**mass**) của hộp, với khả năng rơi (**falls**) được xác định bằng cách đặt **mass** là 5 nếu rơi, và 0 nếu không rơi (giữ nguyên tại chỗ).
- Khối lượng của hộp cũng được điều chỉnh dựa trên kích thước của nó (**width** và **depth**) so với kích thước ban đầu (**originalBoxSize**).
- Tạo một thể hiện của hộp(**body**) với khối lượng và hình dạng đã xác định.
- Đặt vị trí của cơ thể dựa trên các thông số **x**, **y**, và **z**.
- Thêm đối tượng đó vào thế giới vật lý (**world**).

3. Trả về thông tin về hộp:

- Trả về một đối tượng chứa tham chiếu đến mesh (đối tượng trong ThreeJS) và đối tượng trong thế giới vật lý (trong CannonJS).
- Bao gồm các thông số **width** và **depth** của hộp.

5. Function `startGame()`

```
function startGame() {
  autopilot = false;
  gameEnded = false;
  lastTime = 0;
  stack = [];
  overhangs = [];

  if (instructionsElement) instructionsElement.style.display = "none";
  if (resultsElement) resultsElement.style.display = "none";
  if (scoreElement) scoreElement.innerText = 0;
  if (world) {
    // Remove every object from world
    while (world.bodies.length > 0) {
      world.remove(world.bodies[0]);
    }
  }
  if (scene) {
    // Remove every Mesh from the scene
    while (scene.children.find((c) => c.type == "Mesh")) {
      const mesh = scene.children.find((c) => c.type == "Mesh");
      scene.remove(mesh);
    }
    // Foundation
    addLayer(0, 0, originalBoxSize, originalBoxSize);
    // First layer
    addLayer(-10, 0, originalBoxSize, originalBoxSize, "x");
  }
  if (camera) {
    // Reset camera positions
    camera.position.set(4, 4, 4);
    camera.lookAt(0, 0, 0);
  }
}
```

Hình 17: Mô tả ảnh

Mục đích: Khởi động trò chơi bằng cách thiết lập các biến và xóa các đối tượng đã tồn tại.
Giải thích:

- Đặt các biến autopilot, gameEnded, lastTime, stack, và overhangs về giá trị khởi đầu.
- Nếu tồn tại các phần tử DOM của hướng dẫn, kết quả hoặc điểm số, ẩn chúng đi.
- Nếu tồn tại thế giới CannonJS (world), loại bỏ mọi đối tượng từ thế giới bằng cách lặp qua tất cả các thể chất trong world và xóa chúng.
- Nếu tồn tại scene ThreeJS (scene), loại bỏ tất cả các Mesh từ scene bằng cách lặp qua tất cả các phần tử trong scene và loại bỏ các Mesh.

- Tạo lại cơ sở và lớp đầu tiên của trò chơi bằng cách gọi hàm addLayer với các tham số tương ứng.
- Đặt lại vị trí của camera về vị trí ban đầu.

6. Function cutBox(topLayer, overlap, size, delta):

```
function cutBox(topLayer, overlap, size, delta) {  
    const direction = topLayer.direction;  
    const newWidth = direction == "x" ? overlap : topLayer.width;  
    const newDepth = direction == "z" ? overlap : topLayer.depth;  
  
    // Update metadata  
    topLayer.width = newWidth;  
    topLayer.depth = newDepth;  
  
    // Update ThreeJS model  
    topLayer.threejs.scale[direction] = overlap / size;  
    topLayer.threejs.position[direction] -= delta / 2;  
  
    // Update CannonJS model  
    topLayer.cannonjs.position[direction] -= delta / 2;  
  
    // Replace shape to a smaller one (in CannonJS you can't simply just scale a shape)  
    const shape = new CANNON.Box(  
        new CANNON.Vec3(newWidth / 2, boxHeight / 2, newDepth / 2)  
    );  
    topLayer.cannonjs.shapes = [];  
    topLayer.cannonjs.addShape(shape);  
}
```

Hình 18: Mô tả ảnh

Mục đích: thực hiện việc cắt một hộp ở lớp trên cùng của cấu trúc, nếu nó tràn lên trên lớp dưới.

Giải Thích:

- Xác định hướng cắt dựa trên hướng của lớp trên cùng.
- Cập nhật kích thước mới của hộp dựa trên sự chồng lấn (overlap) giữa các khối gần nhất và hướng cắt.
- Cập nhật mô hình ThreeJS để phản ánh sự thay đổi trong kích thước và vị trí của hộp.
- Cập nhật mô hình CannonJS để phản ánh sự thay đổi trong vị trí của hộp.
- Thay thế hình dạng của hộp bằng hình dạng mới, với kích thước mới.

7. Các sự kiện bàn phím và chuột:

```
window.addEventListener("mousedown", eventHandler);
window.addEventListener("touchstart", eventHandler);
window.addEventListener("keydown", function (event) {
    if (event.key == " ") {
        event.preventDefault();
        eventHandler();
        return;
    }
    if (event.key == "R" || event.key == "r") {
        event.preventDefault();
        startGame();
        return;
    }
});
```

Hình 19: Mô tả ảnh

Mục đích: xử lý sự kiện khi người chơi nhấn chuột hoặc phím, chạm vào màn hình để cắt các khối hộp trong trò chơi và khi người chơi nhấn phím "R" hoặc "r" để bắt đầu lại trò chơi

8. eventHandler():

```
function eventHandler() {
    if (autopilot) startGame();
    else splitBlockAndAddNextOneIfOverlaps();
}
```

Hình 20: Mô tả ảnh

Mục đích: Xử lý sự kiện khi người chơi thực hiện hành động cắt hộp hoặc bắt đầu lại trò chơi.

Giải Thích:

Nếu đang ở chế độ tự động (autopilot), hàm này sẽ bắt đầu lại trò chơi. Nếu không, hàm splitBlockAndAddNextOneIfOverlaps() được gọi để kiểm tra xem hộp trên cần được cắt không.

9. function missedTheSpot():

```
function missedTheSpot() {  
    const topLayer = stack[stack.length - 1];  
  
    // Turn to top layer into an overhang and let it fall down  
    addOverhang(  
        topLayer.threejs.position.x,  
        topLayer.threejs.position.z,  
        topLayer.width,  
        topLayer.depth  
    );  
    world.remove(topLayer.cannonjs);  
    scene.remove(topLayer.threejs);  
  
    gameEnded = true;  
    if (resultsElement && !autopilot) resultsElement.style.display = "flex";  
}
```

Hình 21: Mô tả ảnh

Mục đích: Xử lý khi người chơi không cất hộp đúng vị trí, khiến cho hộp trên sẽ trở thành một phần đè lên và rơi xuống.

Giải Thích:

- Lấy thông tin về hộp trên cùng trong ngăn xếp.
- Chuyển hộp đó thành một phần đè lên và để nó rơi xuống, tạo thành một overhang.
- Loại bỏ hộp trên ra khỏi thế giới vật lý (CannonJS) và khỏi cảnh (ThreeJS).
- Đánh dấu trò chơi đã kết thúc.
- Nếu không ở chế độ tự động, hiển thị kết quả nếu có.

10. splitBlockAndAddNextOneIfOverlaps():

```
function splitBlockAndAddNextOneIfOverlaps() {
  if (gameEnded) return;
  const topLayer = stack[stack.length - 1];
  const previousLayer = stack[stack.length - 2];
  const direction = topLayer.direction;
  const size = direction == "x" ? topLayer.width : topLayer.depth;
  const delta =
    topLayer.threejs.position[direction] -
    previousLayer.threejs.position[direction];
  const overhangSize = Math.abs(delta);
  const overlap = size - overhangSize;
  if (overlap > 0) {
    cutBox(topLayer, overlap, size, delta);

    // Overhang
    const overhangShift = (overlap / 2 + overhangSize / 2) * Math.sign(delta);
    const overhangX =
      direction == "x"
        ? topLayer.threejs.position.x + overhangShift
        : topLayer.threejs.position.x;
    const overhangZ =
      direction == "z"
        ? topLayer.threejs.position.z + overhangShift
        : topLayer.threejs.position.z;
    const overhangWidth = direction == "x" ? overhangSize : topLayer.width;
    const overhangDepth = direction == "z" ? overhangSize : topLayer.depth;

    addOverhang(overhangX, overhangZ, overhangWidth, overhangDepth);

    // Next layer
    const nextX = direction == "x" ? topLayer.threejs.position.x : -10;
    const nextZ = direction == "z" ? topLayer.threejs.position.z : -10;
    const newWidth = topLayer.width; // New layer has the same size as the cut top layer
    const newDepth = topLayer.depth; // New layer has the same size as the cut top layer
    const nextDirection = direction == "x" ? "z" : "x";

    if (scoreElement) scoreElement.innerText = stack.length - 1;
    addLayer(nextX, nextZ, newWidth, newDepth, nextDirection);
  } else {
    missedTheSpot();
  }
}
```

Hình 22: Mô tả ảnh

- **Mục đích:** Kiểm tra xem hộp trên có chồng lên hộp dưới không, nếu có thì cắt hộp và thêm hộp mới vào trò chơi.
- **Giải Thích:**

- Kiểm tra xem trò chơi đã kết thúc (*gameEnded*) chưa. Nếu kết thúc, hàm thoát ra.
- Lấy ra lớp trên cùng (*topLayer*) và lớp trước đó (*previousLayer*) từ ngăn xếp khối (*stack*).
- Xác định hướng (*direction*) của lớp trên cùng.
- Tính toán kích thước (*size*) của lớp trên cùng dựa trên hướng.
- Tính toán độ lệch (*delta*) giữa vị trí của lớp trên cùng và lớp trước đó theo hướng tương ứng.
- Tính toán kích thước nhô ra (*overhangSize*) bằng giá trị tuyệt đối của độ lệch.
- Tính toán phần chồng chéo (*overlap*) giữa lớp trên cùng và lớp trước đó (bằng kích thước trừ đi kích thước nhô ra).
 - * Nếu có phần chồng chéo (*overlap* lớn hơn 0):
 - Gọi hàm `cutBox` để cắt lớp trên cùng theo độ chồng chéo, cập nhật dữ liệu và mô hình (cả hình ảnh và vật lý) của lớp đó.
 - Tính toán độ dịch chuyển (*overhangShift*) cho phần nhô ra dựa trên độ chồng chéo và kích thước nhô ra.
 - Tính toán vị trí theo trục X (*overhangX*) và Z (*overhangZ*) cho phần nhô ra dựa trên vị trí của lớp trên cùng, độ dịch chuyển và hướng.
 - Tính toán chiều rộng (*overhangWidth*) và sâu (*overhangDepth*) cho phần nhô ra dựa trên kích thước nhô ra và hướng.
 - Gọi hàm `addOverhang` để tạo một đối tượng mới đại diện cho phần nhô ra với vị trí và kích thước đã tính toán.
 - Tính toán vị trí theo trục X (*nextX*) và Z (*nextZ*) cho lớp tiếp theo dựa trên vị trí của lớp trên cùng và hướng.
 - Tính toán chiều rộng (*newWidth*) và sâu (*newDepth*) cho lớp tiếp theo, sao cho bằng với kích thước của phần còn lại của lớp trên cùng sau khi cắt.
 - Xác định hướng (*nextDirection*) cho lớp tiếp theo, vuông góc với hướng của lớp trên cùng.
 - Cập nhật điểm số (*scoreElement*) nếu tồn tại.
 - Gọi hàm `addLayer` để tạo và xếp lớp tiếp theo lên trên ngăn xếp với vị trí và kích thước đã tính toán, cùng với hướng mới.
 - * Nếu không có phần chồng chéo (*overlap* bằng 0 hoặc nhỏ hơn):
 - Gọi hàm `missedTheSpot` để xử lý trường hợp trượt khỏi ngăn xếp.

11. `updatePhysics(timePassed)`:

```
function updatePhysics(timePassed) {  
  world.step(timePassed / 1000); // Step the physics world  
  
  // Copy coordinates from Cannon.js to Three.js  
  overhangs.forEach((element) => {  
    element.threejs.position.copy(element.cannonjs.position);  
    element.threejs.quaternion.copy(element.cannonjs.quaternion);  
  });  
}
```

Hình 23: Mô tả ảnh

- **Mục đích:** Cập nhật vị trí và góc quay của các đối tượng vật lý dựa trên thời gian trôi qua.
- **Ý tưởng:**
 - Dùng `world.step()` trong Cannon.js để tính toán vị trí mới cho các đối tượng dựa trên thời gian trôi qua.
 - Lặp qua danh sách các đối tượng nhô ra (overhangs), sao chép các tọa độ từ Cannon.js sang Three.js để cập nhật vị trí và góc quay của các đối tượng hiển thị trên màn hình.

12. function animation(time):

```
function animation(time) {  
  if (lastTime) {  
    const timePassed = time - lastTime;  
    const speed = 0.008;  
    const topLayer = stack[stack.length - 1];  
    const previousLayer = stack[stack.length - 2];  
    const boxShouldMove =  
      !gameEnded &&  
      (!autopilot ||  
       (autopilot &&  
        topLayer.threejs.position[topLayer.direction] <  
         previousLayer.threejs.position[topLayer.direction] +  
         robotPrecision));  
    if (boxShouldMove) {  
      topLayer.threejs.position[topLayer.direction] += speed * timePassed;  
      topLayer.cannonjs.position[topLayer.direction] += speed * timePassed;  
      // If the box went beyond the stack then show up the fail screen  
      if (topLayer.threejs.position[topLayer.direction] > 10) {  
        missedTheSpot();  
      }  
    } else {  
      // If it shouldn't move then is it because the autopilot reached the correct position?  
      // Because if so then next level is coming  
      if (autopilot) {  
        splitBlockAndAddNextOneIfOverlaps();  
        setRobotPrecision();  
      }  
    }  
    // 4 is the initial camera height  
    if (camera.position.y < boxHeight * (stack.length - 2) + 4) {  
      camera.position.y += speed * timePassed;  
    }  
    updatePhysics(timePassed);  
    renderer.render(scene, camera);  
  }  
  lastTime = time;  
}
```

Hình 24: Mô tả ảnh

Mục đích: Thiết kế để điều khiển hoạt động di chuyển của khối trong trò chơi và cập nhật các thành phần như camera và vật lý mô phỏng

Ý Tưởng:

- **Kiểm tra thời gian:** Lấy thời gian hiện tại (*time*) và so sánh với thời gian từ frame trước (*lastTime*). Tính toán thời gian đã trôi qua (*timePassed*).
- **Xác định di chuyển:**
 - Lấy ra khối trên cùng (*topLayer*) và khối bên dưới (*previousLayer*) từ ngăn xếp (*stack*).
 - Xác định xem khối trên cùng có nên di chuyển (*boxShouldMove*) dựa trên:
 - * Trò chơi chưa kết thúc (*!gameEnded*).
 - * Chế độ tự động (*autopilot*) tắt hoặc bật:
 - Nếu bật (*autopilot*), kiểm tra vị trí hiện tại của khối (*topLayer.threejs.position[topLayer.direction]*) so với vị trí mục tiêu (vị trí của *previousLayer* cộng thêm *robotPrecision*).
 - Di chuyển khối (nếu cần thiết):
 - * Cập nhật vị trí của khối trên cùng trong cả mô hình trực quan (*threejs*) và mô phỏng vật lý (*cannonjs*) theo thời gian đã trôi qua (*timePassed*) và tốc độ (*speed*).
 - * Nếu vị trí mới vượt quá giới hạn (*topLayer.threejs.position[topLayer.direction] > 10*), gọi *missedTheSpot* để xử lý lỗi.
 - Xử lý chế độ tự động (nếu cần thiết):
 - * Nếu khối không di chuyển (*!boxShouldMove*) và chế độ tự động bật (*autopilot*):
 - Gọi *splitBlockAndAddNextOneIfOverlaps* để đặt khối hiện tại và thêm khối mới.
 - Gọi *setRobotPrecision* để điều chỉnh độ chính xác cho vị trí khối tiếp theo.
- **Điều chỉnh camera:** Cập nhật vị trí camera (*camera.position.y*) nếu thấp hơn vị trí mong muốn dựa trên số lượng khối (*stack.length - 2*), độ cao khối (*boxHeight*) và độ cao cơ bản (+ 4).
- **Cập nhật vật lý và hiển thị:**
 - Gọi *updatePhysics(timePassed)* để cập nhật mô phỏng vật lý.
 - Hiển thị cảnh được cập nhật (*scene*) với camera (*camera*) bằng công cụ render (*renderer*).
- **Lưu trữ thời gian:** Cập nhật *lastTime* với thời gian hiện tại (*time*) để sử dụng cho frame tiếp theo.

6.2 Caro(Tic Tac Toe)

Giới Thiệu Ý Tưởng:

Trò chơi Tic Tac Toe là một trò chơi cổ điển với mục tiêu đơn giản: tạo ra một chuỗi liên tiếp của các ký hiệu "X" hoặc "O" trên bảng. Trò chơi này kết hợp giữa chiến thuật và sự chọn lựa nhanh nhạy. Bằng cách lần lượt đặt ký hiệu của mình trên bảng, người chơi cần phải tìm ra chiến lược phù hợp để ngăn chặn đối thủ hoặc tạo ra cơ hội chiến thắng cho chính mình.

Trò chơi không chỉ là cuộc đối đầu giữa hai người chơi mà còn là một thử thách với máy tính. Thuật toán Minimax được sử dụng để máy tính xác định nước đi tốt nhất trong mỗi tình huống, tạo ra một đối thủ không dễ dàng để vượt qua.

Với cách tiếp cận tương tác và logic, trò chơi Tic Tac Toe không chỉ là một trò giải trí đơn giản mà còn là một cơ hội để thể hiện sự sắc bén trong suy luận và chiến thuật.

```
let gameActive = true;
let currentPlayer = "X";
let gameState = ["", "", "", "", "", "", "", "", ""];

const winningMessage = () => `Player ${currentPlayer} has won!`;
const drawMessage = () => `Game ended in a draw!`;
const currentPlayerTurn = () => `It's ${currentPlayer}'s turn`;

statusDisplay.innerHTML = currentPlayerTurn();
```

Hình 25: Mô tả ảnh

1. `winningMessage`:

- **Mục đích:** Hàm này tạo thông báo ngắn gọn thông báo người chiến thắng trong trò chơi.
- **Ý tưởng:** Sử dụng ký tự đặc biệt để nhúng biến `currentPlayer` (người chơi hiện tại) vào chuỗi, tạo ra thông báo như "Người chơi X đã chiến thắng!" hoặc "Người chơi O đã chiến thắng!".

2. `drawMessage`:

- **Mục đích:** Hàm này định nghĩa thông báo hiển thị khi trò chơi kết thúc hòa (không có người chiến thắng).
- **Ý tưởng:** Đơn giản trả về chuỗi "Trò chơi kết thúc hòa!" để thông báo cho người chơi.

3. `currentPlayerTurn`:

- **Mục đích:** Hàm này tạo thông báo cho biết lượt chơi của ai.
- **Ý tưởng:** Sử dụng ký tự đặc biệt để nhúng biến `currentPlayer` vào chuỗi, hiển thị hoạt động "Lượt của X" hoặc "Lượt của O" tùy thuộc vào người chơi hiện tại.

4. **winningConditions:** Là một mảng chứa các điều kiện chiến thắng trong trò chơi, mỗi phần tử là một mảng chứa các chỉ số của các ô trên bàn cờ cần kiểm tra để xác định chiến thắng. Ví dụ, `[0, 1, 2]` đại diện cho việc người chơi thắng nếu đánh dấu X hoặc O vào ô 0, 1, và 2.

```
const winningConditions = [  
  [0, 1, 2],  
  [3, 4, 5],  
  [6, 7, 8],  
  [0, 3, 6],  
  [1, 4, 7],  
  [2, 5, 8],  
  [0, 4, 8],  
  [2, 4, 6]  
];
```

Hình 26: Mô tả ảnh

5. `handleCellPlayed(clickedCell, clickedCellIndex)`:

Mục đích: Hàm này xử lý việc người chơi đánh dấu vào một ô trong trò chơi.

Giải thích: Hàm này được gọi khi một ô trên bàn cờ được chơi (click vào). Nó cập nhật trạng thái của trò chơi trong biến `gameState`, đánh dấu ô đã được chơi bởi người chơi hiện tại (`currentPlayer`), và cập nhật nội dung của ô đó trên giao diện người dùng.

Cụ thể:

```
function handleCellPlayed(clickedCell, clickedCellIndex) {  
  gameState[clickedCellIndex] = currentPlayer;  
  clickedCell.innerHTML = currentPlayer;  
}
```

Hình 27: Mô tả ảnh

- `clickedCell`: Tham số này chứa phần tử HTML của ô được người chơi đánh dấu.
- `clickedCellIndex`: Tham số này chứa vị trí (index) của ô được đánh dấu trong mảng `gameState`.
- Hàm cập nhật giá trị của phần tử `gameState` tại vị trí `clickedCellIndex` bằng ký hiệu của người chơi hiện tại (`currentPlayer`).
- Cuối cùng, hàm cập nhật nội dung hiển thị của ô `clickedCell` bằng ký hiệu của người chơi hiện tại.

6. `handlePlayerChange()`:

Mục đích: xử lý việc chuyển lượt giữa hai người chơi.

Giải thích: Hàm này được gọi sau khi một người chơi đã chơi một nước đi. Nó chuyển đổi người chơi hiện tại giữa "X" và "O", và cập nhật giao diện người dùng thông qua việc cập nhật trạng thái hiển thị trạng thái lượt chơi của người chơi tiếp theo.

Cụ thể:

```
function handlePlayerChange() {  
    currentPlayer = currentPlayer === "X" ? "O" : "X";  
    statusDisplay.innerHTML = currentPlayerTurn();  
}
```

Hình 28: Mô tả ảnh

- Hàm kiểm tra xem người chơi hiện tại là "X" hay "O".
- Nếu là "X", hàm đổi sang người chơi "O" và ngược lại.
- Cuối cùng, hàm cập nhật nội dung hiển thị của statusDisplay bằng hàm currentPlayerTurn() (giả sử hàm này trả về chuỗi thông báo về lượt của người chơi hiện tại).

7. handleResultValidation():

```
function handleResultValidation() {  
    let roundWon = false;  
    for (let i = 0; i < winningConditions.length; i++) {  
        const winCondition = winningConditions[i];  
        let a = gameState[winCondition[0]];  
        let b = gameState[winCondition[1]];  
        let c = gameState[winCondition[2]];  
        if (a === '' || b === '' || c === '') {  
            continue;  
        }  
        if (a === b && b === c) {  
            roundWon = true;  
            break;  
        }  
    }  
  
    if (roundWon) {  
        statusDisplay.innerHTML = winningMessage();  
        gameActive = false;  
        return;  
    }  
  
    let roundDraw = !gameState.includes("");  
    if (roundDraw) {  
        statusDisplay.innerHTML = drawMessage();  
        gameActive = false;  
        return;  
    }  
  
    handlePlayerChange();  
}
```

Hình 29: Mô tả ảnh

Mục đích: Hàm này được sử dụng để kiểm tra kết quả của trò chơi Tic Tac Toe sau mỗi lượt đánh của người chơi và cập nhật trạng thái trò chơi tương ứng.

Giải thích:

- Hàm duyệt qua tất cả các điều kiện chiến thắng trong mảng `winningConditions`.
- Đối với mỗi điều kiện chiến thắng, nó kiểm tra xem có ba ô liên tiếp giống nhau và không rỗng hay không. Nếu có, biến `roundWon` được đặt thành `true`, và vòng lặp dừng lại.
- Nếu có người chiến thắng, hàm cập nhật nội dung của `statusDisplay` thành thông báo chiến thắng sử dụng hàm `winningMessage()`, đặt `gameActive` thành `false`, và kết thúc hàm.
- Nếu không có người chiến thắng, hàm kiểm tra xem trò chơi có hòa không bằng cách kiểm tra xem mỗi ô trên bàn cờ đã được đánh hay chưa. Nếu không còn ô trống nào, biến `roundDraw` được đặt thành `true`.
- Nếu trò chơi hòa, hàm cập nhật nội dung của `statusDisplay` thành thông báo hòa sử dụng hàm `drawMessage()`, đặt `gameActive` thành `false`, và kết thúc hàm.
- Nếu không có người chiến thắng và trò chơi vẫn tiếp tục, hàm gọi `handlePlayerChange()` để chuyển lượt đến người chơi tiếp theo.

8. `handleCellClick(clickedCellEvent)`:

```
function handleCellClick(clickedCellEvent) {
    const clickedCell = clickedCellEvent.target;
    const clickedCellIndex = parseInt(clickedCell.getAttribute('data-cell-index'));

    if (gameState[clickedCellIndex] !== "" || !gameActive) {
        return;
    }

    handleCellPlayed(clickedCell, clickedCellIndex);
    handleResultValidation();

    if (gameActive && currentPlayer === "0") {
        // Máy tính chọn nước đi
        const bestMove = getBestMove();
        handleCellPlayed(document.querySelector(`[data-cell-index="${bestMove}"]`), bestMove);
        handleResultValidation();
    }
}
```

Hình 30: Mô tả ảnh

Mục đích: Xử lý sự kiện khi người chơi click vào một ô trên bàn cờ và thực hiện các hành động tương ứng.

Giải thích:

- Khi một ô trên bàn cờ được nhấp vào, hàm `handleCellClick` được gọi với thông tin về ô được nhấp vào được truyền qua tham số `clickedCellEvent`.

- Hàm này bắt đầu bằng cách lấy ra ô được nhấp vào từ `clickedCellEvent.target`, sau đó xác định chỉ số của ô đó trong mảng `gameState` bằng cách chuyển đổi thuộc tính `data-cell-index` của ô đó thành một số nguyên.
- Tiếp theo, hàm kiểm tra xem ô được nhấp vào đã được đánh dấu hay không bằng cách kiểm tra giá trị tương ứng trong mảng `gameState` và xem trò chơi có đang hoạt động hay không (`gameActive`). Nếu ô đã được đánh dấu hoặc trò chơi không hoạt động, hàm kết thúc ngay lập tức.
- Nếu ô được nhấp vào chưa được đánh dấu và trò chơi đang hoạt động, hàm gọi hai hàm quan trọng khác:
 - * `handleCellPlayed`: Để xử lý việc đánh dấu ô được nhấp vào với ký hiệu của người chơi hiện tại và cập nhật trạng thái trò chơi.
 - * `handleResultValidation`: Để kiểm tra kết quả của trò chơi sau mỗi lượt đánh và cập nhật trạng thái trò chơi tương ứng.
- Nếu trò chơi vẫn đang hoạt động và lượt hiện tại là của người chơi "O", điều khiển sẽ chuyển sang máy tính để thực hiện nước đi. Điều này được thực hiện bằng cách gọi hàm `getBestMove()` để lấy nước đi tốt nhất từ máy tính, sau đó sử dụng hàm `handleCellPlayed` để đánh dấu ô tương ứng và `handleResultValidation` để kiểm tra kết quả của trò chơi.

9. `getBestMove()`:

```
function getBestMove() {  
    // Lập trình thuật toán Minimax ở đây để tìm nước đi tốt nhất cho máy  
    let bestScore = -Infinity;  
    let bestMove;  
    for (let i = 0; i < gameState.length; i++) {  
        if (gameState[i] === "") {  
            gameState[i] = "O";  
            let score = minimax(gameState, 0, false);  
            gameState[i] = "";  
            if (score > bestScore) {  
                bestScore = score;  
                bestMove = i;  
            }  
        }  
    }  
    return bestMove;  
}
```

Hình 31: Mô tả ảnh

Mục đích: là sử dụng thuật toán Minimax để máy tính tìm nước đi tối ưu nhất trong trò chơi Tic Tac Toe.

Ý tưởng:

1. Khởi Tạo các biến lưu trữ:

- Biến `bestScore` được khởi tạo với giá trị âm vô cực ($-\text{Infinity}$), đại diện cho điểm số thấp nhất có thể.
- Biến `bestMove` được khởi tạo để lưu trữ vị trí (index) của nước đi tốt nhất.

2. Duyệt qua các ô trống:

- Dùng vòng lặp để duyệt qua tất cả các ô trống trên bàn cờ Tic Tac Toe.
- Đối với mỗi ô trống, máy tính sẽ thử đánh dấu ô đó với ký hiệu của mình ("O").
- Gọi thuật toán Minimax để đánh giá trạng thái hiện tại từ góc nhìn của máy tính, xem xét tất cả các nước đi tiềm năng của cả hai người chơi.
- Sau khi đánh giá, nước đi đó được hoàn tác để khôi phục trạng thái trò chơi ban đầu.
- Nếu điểm số thu được lớn hơn `bestScore`, `bestScore` và `bestMove` sẽ được cập nhật để lưu trữ nước đi tốt nhất.

3. Trả về nước đi tốt nhất:

- Sau khi duyệt qua tất cả các ô trống và đánh giá các nước đi tiềm năng, hàm trả về `bestMove`, đại diện cho vị trí của ô mà máy tính nên đánh dấu để tối đa hóa cơ hội chiến thắng hoặc hòa.

10. `minimax(state, depth, isMaximizing)`:

```
function minimax(state, depth, isMaximizing) {
  if (checkWinner(state, "X")) {
    return -10 + depth;
  } else if (checkWinner(state, "O")) {
    return 10 - depth;
  } else if (state.every(cell => cell !== "")) {
    return 0;
  }

  if (isMaximizing) {
    let bestScore = -Infinity;
    for (let i = 0; i < state.length; i++) {
      if (state[i] === "") {
        state[i] = "O";
        let score = minimax(state, depth + 1, false);
        state[i] = "";
        bestScore = Math.max(score, bestScore);
      }
    }
    return bestScore;
  } else {
    let bestScore = Infinity;
    for (let i = 0; i < state.length; i++) {
      if (state[i] === "") {
        state[i] = "X";
        let score = minimax(state, depth + 1, true);
        state[i] = "";
        bestScore = Math.min(score, bestScore);
      }
    }
    return bestScore;
  }
}
```

Hình 32: Mô tả ảnh

Mục đích: Hàm `minimax` sử dụng tính đệ quy để mô phỏng nhiều trạng thái trò chơi tiềm năng, tính toán điểm số cho mỗi trạng thái dựa trên kết quả thắng, thua, hòa và độ sâu của mô phỏng. Bằng cách này, hàm `minimax` cung cấp thông tin cho hàm `getBestMove` để lựa chọn nước đi dẫn đến trạng thái có điểm số cao nhất.

Ý Tưởng:

1. Kiểm tra trạng thái trò chơi:
 - Nếu trạng thái trò chơi hiện tại có người chiến thắng là "X", hàm trả về -10 cộng với độ sâu của nút trong cây trò chơi.
 - Nếu trạng thái trò chơi hiện tại có người chiến thắng là "O", hàm trả về 10 trừ đi độ sâu của nút trong cây trò chơi.
 - Nếu trạng thái trò chơi hiện tại không có người chiến thắng và không còn ô trống nào, hàm trả về 0, đại diện cho trạng thái hòa.
2. Nếu đang là lượt của máy tính (`isMaximizing = true`):
 - Duyệt qua tất cả các ô trống trong trạng thái trò chơi.
 - Đối với mỗi ô trống, máy tính thử đánh dấu ô đó là "O" và gọi đệ quy để đánh giá trạng thái tiếp theo.
 - Sau khi đánh giá, nước đi được hoàn tác và máy tính lựa chọn điểm số lớn nhất từ tất cả các nước đi tiềm năng.
3. Nếu đang là lượt của người chơi (`isMaximizing = false`):
 - Tương tự như bước 2, nhưng lần này người chơi đóng vai trò của "X" và lựa chọn điểm số nhỏ nhất từ tất cả các nước đi tiềm năng.
4. Cuối cùng, hàm trả về điểm số tốt nhất dựa trên vai trò của người chơi hiện tại (máy tính hoặc người chơi).

11. `checkWinner(state, player)`:

```
function checkWinner(state, player) {  
  for (let i = 0; i < winningConditions.length; i++) {  
    const [a, b, c] = winningConditions[i];  
    if (state[a] === player && state[b] === player && state[c] === player) {  
      return true;  
    }  
  }  
  return false;  
}
```

Hình 33: Mô tả ảnh

Mục đích: Để kiểm tra xem một người chơi có chiến thắng trong trò chơi Tic Tac Toe không.

Ý Tưởng:

`checkWinner(state, player)` nhận vào 2 tham số

- `state`: Một mảng biểu diễn trạng thái hiện tại của bàn cờ.

- *player*: Ký hiệu của người chơi cần kiểm tra chiến thắng ("X" hoặc "O").
- Hàm này duyệt qua danh sách các điều kiện chiến thắng để kiểm tra xem người chơi đã chiến thắng hay chưa dựa trên trạng thái hiện tại của bàn cờ và người chơi đó.
- Với mỗi điều kiện chiến thắng, hàm kiểm tra xem ba ô tương ứng có giống với người chơi hay không.
- Nếu tất cả ba ô có giá trị giống với người chơi, hàm trả về true, đồng nghĩa với việc người chơi đã chiến thắng theo điều kiện đó.
- Nếu không có điều kiện chiến thắng nào được thỏa mãn, hàm trả về false, cho biết rằng người chơi không chiến thắng theo bất kỳ điều kiện nào.

12. `handleRestartGame()`:

```
function handleRestartGame() {  
    gameActive = true;  
    currentPlayer = "X";  
    gameState = ["", "", "", "", "", "", "", "", ""];  
    statusDisplay.innerHTML = currentPlayerTurn();  
    document.querySelectorAll('.cell').forEach(cell => cell.innerHTML = "");  
}
```

Hình 34: Mô tả ảnh

Mục đích: Thiết lập lại trạng thái của trò chơi khi người dùng chọn khởi động lại.

Ý tưởng:

- Gán giá trị *true* cho biến *gameActive*, cho biết rằng trò chơi đang hoạt động.
- Đặt người chơi hiện tại là "X" bằng cách gán giá trị "X" cho biến *currentPlayer*.
- Đặt lại trạng thái của bàn cờ bằng cách gán một mảng chứa 9 phần tử rỗng cho biến *gameState*, đại diện cho trạng thái ban đầu của bàn cờ.
- Cập nhật nội dung hiển thị của *statusDisplay* bằng cách gọi hàm *currentPlayerTurn()*.
- Duyệt qua tất cả các ô trên bàn cờ và gán nội dung rỗng cho mỗi ô, đảm bảo rằng bàn cờ được làm sạch khi trò chơi bắt đầu lại.

13. Đăng ký sự kiện cho các ô trên bàn cờ và nút khởi động lại:

```
document.querySelectorAll('.cell').forEach(cell => cell.addEventListener('click', handleCellClick));  
document.querySelector('.game--restart').addEventListener('click', handleRestartGame);
```

Hình 35: Mô tả ảnh

Mục đích: Khi người dùng click vào một ô trên bàn cờ, hàm *handleCellClick* được gọi để xử lý việc đánh dấu vào ô đó. Khi người dùng click vào nút khởi động lại, trò chơi sẽ bắt đầu lại từ đầu.

Ý tưởng:

- Sử dụng `document.querySelectorAll('.cell').forEach(cell => cell.addEventListener('click', handleCellClick))` để lặp qua tất cả các ô trên bàn cờ và gán sự kiện click cho mỗi ô, khi người dùng click vào một ô, hàm `handleCellClick` sẽ được gọi để xử lý việc đánh dấu vào ô đó.
- Sử dụng `document.querySelector('.game-restart').addEventListener('click', handleRestartGame)` để gán sự kiện click cho nút khởi động lại. Khi người dùng click vào nút này, hàm `handleRestartGame` sẽ được gọi để bắt đầu lại trò chơi từ đầu.

6.3 Game xếp số 2048

1. randomAddItem():

```
randomAddItem() {  
    if (this.isFull()) return;  
    while (true) {  
        var index = random(0, data.cell.length - 1);  
        var exist = data.cell[index].val !== 0;  
        if (!exist) {  
            this.addItem(index, 2);  
            break;  
        }  
    }  
}
```

Hình 36: Mô tả ảnh

Mục đích: Hàm này thêm một ô số mới vào một vị trí trống ngẫu nhiên trên bảng.

Ý tưởng:

- Hàm đầu tiên kiểm tra xem bảng có đầy hay không bằng hàm `isFull()`. Nếu đầy, hàm thoát ra mà không thêm bất kỳ thứ gì.
- Nếu còn chỗ trống, nó đi vào vòng lặp và tiếp tục cho đến khi tìm thấy một vị trí trống:
 - * Tạo một chỉ số ngẫu nhiên trong phạm vi kích thước của bảng (sử dụng `random(0, data.cell.length - 1)`).
 - * Kiểm tra xem giá trị tại chỉ mục đó (`data.cell[index].val`) có khác 0 hay không (nghĩa là không trống).
 - * Nếu vị trí trống (`!exist`), gọi hàm `addItem` để đặt ô số mới (có giá trị 2 trong trường hợp này) tại khối hộp đó.

* Nếu vị trí không trống, vòng lặp tiếp tục tìm kiếm một vị trí trống ngẫu nhiên khác.

2. addItem(index, val):

```
addItem(index, val) {  
    data.cell[index] = {  
        val: val,  
        index: index  
    };  
    this.view.appear(index);  
}
```

Hình 37: Mô tả ảnh

Mục đích: thêm một phần tử mới vào cấu trúc dữ liệu ở vị trí được chỉ định và sau đó hiển thị phần tử đó trên giao diện người dùng.

Ý tưởng: tạo một đối tượng mới chứa giá trị và index, sau đó gán đối tượng này vào mảng cell trong đối tượng data tại vị trí index. Tiếp theo, gọi phương thức appear() từ đối tượng view để hiển thị phần tử mới được thêm vào.

3. isFull():

```
isFull() {  
    var full = cell.filter(function (el) {  
        return el.val === 0;  
    });  
    return full.length === 0;  
}
```

Hình 38: Mô tả ảnh

Mục đích: là kiểm tra xem cấu trúc có đầy đủ các ô số hay chưa.

Ý tưởng: sử dụng phương thức filter() để lọc ra các phần tử có giá trị bằng 0 từ mảng cell. Sau đó, kiểm tra độ dài của mảng được lọc. Nếu độ dài bằng 0, tức là không có phần tử nào có giá trị bằng 0, thì cấu trúc dữ liệu được coi là đầy đủ và hàm trả về true, ngược lại trả về false.

4. start():

```
start() {  
    for (var i = 0; i < 2; i++) {  
        this.randomAddItem();  
    }  
}
```

Hình 39: Mô tả ảnh

Mục đích: khởi động quá trình bằng cách thêm một số phần tử ngẫu nhiên vào cấu trúc dữ liệu.

Ý tưởng: sử dụng một vòng lặp để thêm một số phần tử ngẫu nhiên vào cấu trúc dữ liệu. Trong trường hợp này, vòng lặp chạy hai lần, và mỗi lần vòng lặp chạy, phương thức randomAddItem() được gọi để thêm một phần tử ngẫu nhiên vào cấu trúc dữ liệu. Điều này giúp bắt đầu quá trình với một số dữ liệu ngẫu nhiên ban đầu.

5. restart():

```
restart() {  
    var _this = this;  
    over = false;  
    this.initCell();  
    this.view.restart();  
    this.start();  
    data.score = 0;  
    this.save();  
    setTimeout(function () {  
        _this.view.setup();  
    });  
}
```

Hình 40: Mô tả ảnh

Mục đích: sử dụng để khởi động lại trò chơi.

Ý tưởng:

- Gán giá trị false cho biến over để đánh dấu rằng trò chơi chưa kết thúc.
- Khởi tạo lại cấu trúc dữ liệu cell bằng cách gọi phương thức initCell().
- Khởi động lại giao diện người dùng bằng cách gọi phương thức restart() từ đối tượng view.

- Bắt đầu trò chơi bằng cách gọi phương thức `start()`.
- Đặt điểm số về 0.
- Lưu trạng thái trò chơi.
- Thiết lập lại giao diện người dùng bằng cách gọi phương thức `setup()` từ đối tượng `view` sau một khoảng thời gian nhất định.

6. `save()`:

```
restart() {  
    var _this = this;  
    over = false;  
    this.initCell();  
    this.view.restart();  
    this.start();  
    data.score = 0;  
    this.save();  
    setTimeout(function () {  
        _this.view.setup();  
    });  
}
```

Hình 41: Mô tả ảnh

Mục đích: để lưu trạng thái của trò chơi, bao gồm cấu trúc dữ liệu cell và điểm số, vào bộ nhớ cục bộ (local storage) của trình duyệt web.

Ý tưởng:

- Lưu điểm số tốt nhất (`bestScore`) và trạng thái của trò chơi (`gameState`) vào local storage.
- `bestScore` được lưu trực tiếp vào local storage.
- Trạng thái của trò chơi được lưu dưới dạng một đối tượng JSON, bao gồm cấu trúc dữ liệu cell và điểm số. Trước khi lưu vào local storage, đối tượng JSON này được chuyển đổi thành một chuỗi bằng phương thức `JSON.stringify()`.

7. winning():

```
winning() {  
    over = true;  
    localStorage.gameState = '';  
    this.view.winning();  
}
```

Hình 42: Mô tả ảnh

Mục đích: để xử lý trạng thái khi người chơi chiến thắng trong trò chơi.

Ý tưởng:

- Thiết lập biến `over` thành `true` để đánh dấu rằng trò chơi đã kết thúc.
- Xóa trạng thái của trò chơi (`gameState`) khỏi local storage bằng cách gán giá trị rỗng cho `localStorage.gameState`. Điều này giúp đặt lại trò chơi để bắt đầu một ván mới.
- Gọi phương thức `winning()` từ đối tượng `view` để hiển thị trạng thái chiến thắng trên giao diện người dùng.

8. checkWinning():

```
checkWinning() {  
    var isWinning = cell.find(function (el) {  
        return el.val === config.max;  
    });  
    if (isWinning) {  
        this.winning();  
    }  
}
```

Hình 43: Mô tả ảnh

Mục đích: kiểm tra xem người chơi đã chiến thắng trong trò chơi hay chưa.

Ý tưởng:

- Sử dụng phương thức `find()` để tìm kiếm một phần tử trong mảng `cell` có giá trị bằng giá trị tối đa (`config.max`).
- Nếu tìm thấy phần tử có giá trị tối đa, tức là người chơi đã đạt được điểm số cao nhất có thể, thì gọi phương thức `winning()` để xử lý trạng thái chiến thắng.

9. failure():

```
failure() {  
    over = true;  
    localStorage.gameState = '';  
    this.view.failure();  
}
```

Hình 44: Mô tả ảnh

Mục đích: xử lý trạng thái khi người chơi thất bại trong trò chơi.

Ý tưởng:

- Sử dụng phương thức `find()` để tìm kiếm một phần tử trong mảng `cell` có giá trị bằng giá trị tối đa (`config.max`).
- Nếu tìm thấy phần tử có giá trị tối đa, tức là người chơi đã đạt được điểm số cao nhất có thể, thì gọi phương thức `winning()` để xử lý trạng thái chiến thắng.

10. checkSame(arr, index):

```
checkSame(arr, index) {  
    same = arr.some(function (el, index, arr) {  
        if (index === arr.length - 1) return;  
        return el.val === arr[index + 1].val;  
        return true;  
    }));  
    return same;  
}
```

Hình 45: Mô tả ảnh

Mục đích: kiểm tra xem trong một mảng các phần tử có giống nhau không.

Ý tưởng:

- Sử dụng phương thức `some()` để kiểm tra xem có phần tử nào trong mảng `arr` thỏa mãn điều kiện được đưa ra hay không.
- Trong hàm gọi của `some()`, kiểm tra xem giá trị của phần tử hiện tại (`el`) có bằng giá trị của phần tử tiếp theo trong mảng hay không. Nếu có, trả về `true`; nếu không, trả về `false`.
- Hàm sẽ trả về kết quả của việc kiểm tra, tức là `true` nếu có ít nhất một cặp phần tử giống nhau trong mảng, và `false` nếu không có cặp nào giống nhau.

10. `checkfailure()`:

```
checkfailure() {  
    var _this = this;  
    var same = false;  
    var called = function (arr, str) {  
        if (same) return;  
        same = arr.some(function (el) {  
            return _this.checkSame(el);  
        });  
    };  
    called(this.chunkX(), 'x');  
    called(this.chunkY(), 'y');  
    setTimeout(function () {  
        if (!same) {  
            _this.failure();  
        }  
    });  
}
```

Hình 46: Mô tả ảnh

Mục đích: Để kiểm tra xem trò chơi đã kết thúc chưa bằng cách xác định xem có cặp phần tử giống nhau trong các hàng hoặc cột của trò chơi hay không.

Ý tưởng:

- Tạo một biến `same` để lưu trạng thái của việc tìm thấy cặp phần tử giống nhau.
- Tạo một hàm `called` để kiểm tra xem trong mỗi hàng hoặc cột có cặp phần tử giống nhau không.
- Trong hàm `called`, sử dụng phương thức `some()` để kiểm tra xem có phần tử nào trong hàng hoặc cột đó giống nhau không bằng cách gọi hàm `checkSame()`.

- Nếu tìm thấy cặp phần tử giống nhau, đặt biến `same` thành `true`.
- Sau khi kiểm tra xong các hàng và cột, sử dụng `setTimeout()` để thực hiện kiểm tra kết quả.
- Nếu không tìm thấy cặp phần tử giống nhau (`same` vẫn là `false`), gọi phương thức `failure()` để xử lý trạng thái thất bại trong trò chơi.

11. `setBest()`:

```
setBest() {  
    var best = localStorage('bestScore');  
    data.best = best || 0;  
}
```

Hình 47: Mô tả ảnh

Mục đích: cập nhật điểm số cao nhất (`bestScore`) từ local storage và lưu vào biến dữ liệu của trò chơi.

Ý tưởng:

- Sử dụng hàm `localStorage()` để lấy giá trị của `bestScore` từ local storage.
- Gán giá trị lấy được từ local storage cho biến `best`. Nếu không có giá trị được lấy từ local storage (do chưa có điểm số cao nhất được lưu trữ trước đó), gán giá trị mặc định là 0 cho `best`.
- Lưu giá trị `best` vào biến dữ liệu `data.best` của trò chơi.

12. `getHistory()`:

```
getHistory() {  
    var gameState = localStorage('gameState');  
    if (gameState && gameState.socre && gameState.cell) {  
        return gameState;  
    }  
}
```

Hình 48: Mô tả ảnh

Mục đích: Lấy trạng thái trò chơi trước đó từ local storage.

Ý tưởng:

- Sử dụng hàm `localStorage()` để lấy giá trị của trạng thái trò chơi từ local storage và gán vào biến `gameState`.
- Kiểm tra xem `gameState` có tồn tại không và có chứa thông tin về điểm số (`socre`) và cấu trúc dữ liệu ô (`cell`) không.

- Nếu `gameState` tồn tại và chứa thông tin đầy đủ về điểm số và cấu trúc dữ liệu ô, thì trả về `gameState`.

13. `restoreHistory(history)`:

```
restoreHistory(history) {  
    data.cell = history.cell;  
    data.score = history.score;  
    cell = data.cell;  
    this.view.restoreTile();  
}
```

Hình 49: Mô tả ảnh

Mục đích: khôi phục trạng thái trò chơi từ lịch sử trước đó.

Ý tưởng:

- Nhận đối số `history`, đại diện cho thông tin trạng thái trò chơi trước đó.
- Gán giá trị của cấu trúc dữ liệu ô (`cell`) từ `history.cell` cho `data.cell`.
- Gán giá trị điểm số (`score`) từ `history.score` cho `data.score`.
- Gán giá trị của `data.cell` cho biến `cell`.
- Gọi phương thức `restoreTile()` từ đối tượng `view` để khôi phục các ô trên giao diện người dùng.

14. `initCell()`:

```
initCell() {  
    for (var i = 0; i < 16; i++) {  
        cell[i] = {  
            val: 0,  
            index: i,  
        };  
    }  
}
```

Hình 50: Mô tả ảnh

Mục đích: khởi tạo cấu trúc dữ liệu ô trong trò chơi.

Ý tưởng:

- Sử dụng vòng lặp để duyệt qua các vị trí trong cấu trúc dữ liệu ô (`cell`).
- Tại mỗi vị trí, gán một đối tượng mới cho `cell[i]` với hai thuộc tính:
 - * `val`: Giá trị của ô, ban đầu được thiết lập là 0.
 - * `index`: Chỉ mục của ô trong cấu trúc dữ liệu, từ 0 đến 15.

15. `addScore(score)`:

```
addScore(score) {  
    data.score += score;  
    if (data.best < data.score) {  
        data.best = data.score;  
        this.view.updateBest();  
    }  
    this.view.updateScore(score);  
}
```

Hình 51: Mô tả ảnh

Mục đích: cập nhật điểm số của trò chơi và điểm số tốt nhất, sau đó cập nhật giao diện người dùng để hiển thị các thay đổi.

Ý tưởng:

- Tăng điểm số hiện tại của trò chơi (`data.score`) bằng giá trị `score` được truyền vào.
- So sánh điểm số hiện tại với điểm số tốt nhất (`data.best`). Nếu điểm số hiện tại lớn hơn điểm số tốt nhất, thì cập nhật điểm số tốt nhất thành điểm số hiện tại.
- Gọi phương thức `updateBest()` từ đối tượng `view` để cập nhật giao diện hiển thị điểm số tốt nhất mới.
- Gọi phương thức `updateScore(score)` từ đối tượng `view` để cập nhật giao diện hiển thị điểm số mới (được tính từ `score` được truyền vào).

16. `chunkX()`:

```
chunkX() {  
    var new_cell = [];  
    for (var i = 0; i < cell.length; i += 4) {  
        new_cell.push(cell.slice(i, i + 4));  
    }  
    return new_cell;  
}
```

Hình 52: Mô tả ảnh

Mục đích: chia cấu trúc dữ liệu `cell` thành các hàng (chunks) có chiều dài bằng 4, tức là tạo ra một mảng hai chiều của các ô trong trò chơi.

Ý tưởng:

- Tạo một mảng mới có tên là `new_cell` để lưu trữ các hàng của cấu trúc dữ liệu `cell` sau khi chia nhỏ.
- Sử dụng một vòng lặp để duyệt qua từng phần tử của cấu trúc dữ liệu `cell`.
- Trong mỗi lần lặp, cắt một phần của mảng `cell` có chiều dài bằng 4 bắt đầu từ vị trí hiện tại và thêm vào mảng `new_cell`.

- Trả về mảng `new_cell`, chứa các hàng của cấu trúc dữ liệu cell đã được chia nhỏ.

17. `chunkY()`:

```
chunkY() {  
    var arr = this.chunkX();  
    var new_cell = [  
        [],  
        [],  
        [],  
        []  
    ];  
    for (var i = 0; i < arr.length; i++) {  
        for (var j = 0; j < arr[i].length; j++) {  
            new_cell[j][i] = arr[i][j];  
        }  
    }  
    return new_cell;  
}
```

Hình 53: Mô tả ảnh

Mục đích: chia cấu trúc dữ liệu cell thành các cột (chunks) có chiều rộng bằng 4, tức là tạo ra một mảng hai chiều của các ô trong trò chơi, nhưng dữ liệu trong mỗi hàng được chuyển thành cột và ngược lại.

Ý tưởng:

- Sử dụng hàm `chunkX()` để chia cấu trúc dữ liệu cell thành các hàng.
- Khởi tạo một mảng hai chiều mới có tên là `new_cell`, với 4 hàng và số cột tương ứng với số hàng trong mỗi hàng của cấu trúc dữ liệu cell.
- Sử dụng hai vòng lặp lồng nhau để duyệt qua từng phần tử của mảng đã chia thành các hàng.
- Trong mỗi lần lặp, gán giá trị của các phần tử tương ứng trong các hàng của mảng đã chia thành các cột trong mảng `new_cell`.
- Trả về mảng `new_cell`, chứa các cột của cấu trúc dữ liệu cell đã được chia nhỏ.

18. `arrayInnerReverse(arr)`:

```
arrayInnerReverse(arr) {  
    arr.forEach(function (el, index) {  
        arr[index] = el.reverse();  
    });  
    return arr;  
}
```

Hình 54: Mô tả ảnh

Mục đích: để đảo ngược các phần tử trong mỗi mảng con của một mảng hai chiều.

Ý tưởng:

- Sử dụng phương thức `forEach()` để lặp qua từng phần tử của mảng `arr`.
- Trong mỗi lần lặp, sử dụng phương thức `reverse()` để đảo ngược thứ tự các phần tử trong mảng con `el`.
- Gán lại mảng con đã được đảo ngược vào vị trí tương ứng trong mảng `arr`.
- Trả về mảng `arr` đã có các mảng con bên trong được đảo ngược thứ tự các phần tử.

19. updatePos(old_index, index):

```
updatePos(old_index, index) {  
    cell[index].val = cell[old_index].val;  
    cell[old_index].val = 0;  
    move = true;  
    return old_index;  
}
```

Hình 55: Mô tả ảnh

Mục đích: cập nhật vị trí của một phần tử trong cấu trúc dữ liệu khi di chuyển từ vị trí cũ (`old_index`) đến vị trí mới (`index`).

Ý tưởng:

- Gán giá trị của phần tử ở vị trí mới (`index`) bằng giá trị của phần tử ở vị trí cũ (`old_index`), để di chuyển phần tử đến vị trí mới.
- Gán giá trị của phần tử ở vị trí cũ (`old_index`) bằng 0, để xóa phần tử khỏi vị trí cũ.
- Thiết lập biến `move` thành `true` để đánh dấu rằng đã có sự di chuyển xảy ra.
- Trả về giá trị của `old_index`, để sử dụng cho mục đích kiểm tra sau này.

20. updateVal(index, val):

```
updateVal(index, val) {  
    var _this = this;  
    cell[index].val = val;  
    setTimeout(function () {  
        _this.view.updateVal(index);  
    }, 0);  
}
```

Hình 56: Mô tả ảnh

Mục đích: Cập nhật giá trị của một phần tử trong cấu trúc dữ liệu và cập nhật giao diện người dùng để hiển thị thay đổi.

Ý tưởng: Gán giá trị mới (*val*) cho phần tử tại vị trí *index* trong mảng *cell*. Sử dụng *setTimeout* để lên lịch cập nhật giao diện người dùng, với thời gian trễ là 0, để thực hiện ngay lập tức sau khi hàm gọi xong và tránh sự chậm trễ trong việc cập nhật giao diện.

21. *updateItem(old_index, index):*

```
updateItem(old_index, index) {  
    if (cell[old_index] === cell[index]) return;  
    var old_index = this.updatePos(old_index, index);  
    this.view.move(old_index, index);  
}
```

Hình 57: Mô tả ảnh

Mục đích: Cập nhật vị trí của một phần tử trong cấu trúc dữ liệu và di chuyển phần tử đó trên giao diện người dùng nếu cần thiết.

Ý tưởng: Kiểm tra xem phần tử ở vị trí cũ (*old_index*) có giống phần tử ở vị trí mới (*index*) không. Nếu giống nhau, không cần thực hiện gì cả và kết thúc hàm. Gọi hàm *updatePos(old_index, index)* để cập nhật vị trí của phần tử từ *old_index* sang *index*. Gọi phương thức *move(old_index, index)* từ đối tượng *view* để di chuyển phần tử từ vị trí cũ sang vị trí mới trên giao diện người dùng.

22. *removeItem(index):*

```
removeItem(index) {  
    cell[index].val = 0;  
    this.view.remove(index);  
}
```

Hình 58: Mô tả ảnh

Mục đích: Xóa một phần tử khỏi cấu trúc dữ liệu và giao diện người dùng tại vị trí chỉ định.

Ý tưởng: Đặt giá trị của phần tử tại vị trí *index* trong mảng *cell* về 0 để đánh dấu việc xóa phần tử đó. Sau đó, gọi phương thức *remove()* từ đối tượng *view* để cập nhật giao diện, thông báo rằng phần tử đã được xóa.

23. `getSum(obj, i, j):`

```
getSum(obj, i, j) {  
    return obj[i].val + obj[j].val;  
}
```

Hình 59: Mô tả ảnh

Mục đích: Tính tổng của hai phần tử trong cấu trúc dữ liệu `obj` tại hai vị trí `i` và `j`.

Ý tưởng: Lấy giá trị của phần tử tại vị trí `i` và `j` từ cấu trúc dữ liệu `obj`, sau đó thực hiện phép cộng giữa chúng và trả về kết quả. Điều này giúp tính toán tổng của hai phần tử từ cấu trúc dữ liệu bất kỳ.

23. `move(dir):`

```
move(dir) {  
    if (over) return;  
    var _this = this;  
    var _score = 0;  
    var _move = false;  
    var new_cell = [];  
    if (dir === 0 || dir === 2) {  
        new_cell = this.chunkX();  
    } else if (dir === 1 || dir === 3) {  
        new_cell = this.chunkY();  
    }  
    if (dir === 2 || dir === 3) {  
        new_cell = this.arrayInnerReverse(new_cell);  
    }  
    new_cell.forEach(function (arr, index) {  
        var moveInfo = _this.moving(arr, indexs[dir][index]);  
        _score += moveInfo.score;  
    });  
    this.addScore(_score);  
    if (move) {  
        this.randomAddItem();  
        _move = true;  
        move = false;  
    }  
    this.save();  
    this.checkWinning();  
    if (this.isFull()) {  
        this.checkFailure();  
    }  
    return {  
        move: _move,  
    };  
}
```

Hình 60: Mô tả ảnh

Mục đích:

Để thực hiện di chuyển các phần tử trong trò chơi 2048 theo hướng được chỉ định.

Ý tưởng:

- * Kiểm tra xem trò chơi đã kết thúc chưa. Nếu đã kết thúc (biến `over` được thiết lập), hàm sẽ kết thúc và không thực hiện bất kỳ hành động nào.
- Khởi tạo biến `_score` để lưu tổng số điểm được ghi nhận từ các phép di chuyển.
- Khởi tạo biến `_move` để theo dõi xem có di chuyển nào được thực hiện hay không.
- Tạo một mảng mới `new_cell` để chứa các hàng hoặc cột của trò chơi, tùy thuộc vào hướng di chuyển (`dir`).
- Nếu hướng di chuyển là 2 hoặc 3 (di chuyển sang phải hoặc xuống dưới), thực hiện lật ngược các hàng hoặc cột trong mảng `new_cell`.
- Duyệt qua mỗi hàng hoặc cột trong `new_cell` và thực hiện di chuyển các phần tử bằng cách gọi phương thức `moving()`, trả về thông tin về phép di chuyển như điểm số.
- Tổng hợp điểm số từ tất cả các phép di chuyển và cập nhật điểm số của trò chơi bằng cách gọi phương thức `addScore()`.
- Nếu có di chuyển được thực hiện (biến `move` được thiết lập), thêm một phần tử mới vào trò chơi bằng cách gọi phương thức `randomAddItem()`.
- Lưu trạng thái của trò chơi bằng cách gọi phương thức `save()`.
- Kiểm tra xem người chơi đã chiến thắng hay không bằng cách gọi phương thức `checkWinning()`.
- Nếu không còn ô trống trong trò chơi, kiểm tra xem người chơi đã thất bại hay không bằng cách gọi phương thức `checkfailure()`.
- Trả về một đối tượng với thuộc tính `move`, cho biết liệu có di chuyển nào được thực hiện hay không.

23. `mergeMove(_cell, index, num1, num2, num3):`

```
mergeMove(_cell, index, num1, num2, num3) {  
    var sum = this.getSum(_cell, num1, num2);  
    this.removeItem(_cell[num1].index);  
    this.updateItem(_cell[num2].index, index[num3]);  
    this.updateVal(index[num3], sum);  
}
```

Hình 61: Mô tả ảnh

Mục đích:

Hợp nhất các ô trong trò chơi 2048 khi chúng có cùng giá trị.

Ý tưởng:

1. Tính tổng của hai ô có giá trị giống nhau.
2. Loại bỏ ô thứ nhất (`num1`) khỏi trò chơi bằng cách gọi phương thức `removeItem()`.
3. Cập nhật ô thứ hai (`num2`) với giá trị mới (`index[num3]`) bằng cách gọi phương thức `updateItem()`.
4. Cập nhật giá trị của ô thứ hai (`num2`) thành tổng đã tính được.

23. `normalMove(_cell, index):`

```
normalMove(_cell, index) {  
    var _this = this;  
    _cell.forEach(function (el, i) {  
        _this.updateItem(_cell[i].index, index[i]);  
    });  
}
```

Hình 62: Mô tả ảnh

Mục đích: Di chuyển các phần tử trong trò chơi 2048 từ vị trí hiện tại của chúng đến vị trí mới được chỉ định.

Ý Tưởng:

- Đầu tiên, một biến `this` được khởi tạo để trỏ đến đối tượng hiện tại.
- Dùng vòng lặp `forEach` để duyệt qua mỗi phần tử `el` trong mảng `cell`.
- Trong mỗi lần lặp, gọi phương thức `updateItem()` để cập nhật vị trí của phần tử từ vị trí hiện tại (`cell[i].index`) sang vị trí mới (`index[i]`).

24. `moving(arr, index):`

```
moving(arr, index) {
    var _this = this;
    var _score = 0;
    var _cell = arr.filter(function (el) {
        return el.val !== 0;
    });
    if (_cell.length === 0) {
        return {
            score: 0,
        };
    };
    var calls = [
        function () {
            _this.normalMove(_cell, index);
        },
        function () {
            if (_cell[0].val === _cell[1].val) {
                _this.mergeMove(_cell, index, 0, 1, 0);
                _score += config.bonus_point;
            } else {
                _this.normalMove(_cell, index);
            }
        },
        function () {
            if (_cell[0].val === _cell[1].val) {
                _this.mergeMove(_cell, index, 0, 1, 0);
                _this.updateItem(_cell[2].index, index[1]);
                _score += config.bonus_point;
            } else if (_cell[1].val === _cell[2].val) {
                _this.updateItem(_cell[0].index, index[0]);
                _this.mergeMove(_cell, index, 1, 2, 1);
                _score += config.bonus_point;
            } else {
                _this.normalMove(_cell, index);
            }
        },
        function () {
            if (_cell[0].val === _cell[1].val) {
                _this.mergeMove(_cell, index, 0, 1, 0);
                _score += config.bonus_point;
                if (_cell[2].val === _cell[3].val) {
                    _this.mergeMove(_cell, index, 2, 3, 1);
                    _score += config.bonus_point;
                } else {
                    _this.updateItem(_cell[2].index, index[1]);
                    _this.updateItem(_cell[3].index, index[2]);
                }
            } else if (_cell[1].val === _cell[2].val) {
                _this.mergeMove(_cell, index, 1, 2, 1);
                _this.updateItem(_cell[3].index, index[2]);
                _score += config.bonus_point;
            } else if (_cell[2].val === _cell[3].val) {
                _this.mergeMove(_cell, index, 2, 3, 2);
                _score += config.bonus_point;
            }
        }
    ];
}
```

Hình 63: Mô tả ảnh

Mục đích: Điều khiển các phần tử trong trò chơi 2048 và thực hiện các hành động cụ thể như hợp nhất, cập nhật điểm số.

Ý Tưởng:

- Đầu tiên, một biến `_this` được sử dụng để trỏ đến đối tượng hiện tại.
- Một biến `_score` được khởi tạo để tính tổng điểm số từ các hành động di chuyển.
- Mảng `_cell` được tạo ra bằng cách lọc các phần tử có giá trị không phải 0 từ mảng `arr`. Điều này giúp loại bỏ các ô trống và chỉ giữ lại các ô có giá trị.
- Kiểm tra nếu độ dài của mảng `_cell` là 0, tức là không có phần tử nào cần di chuyển, trả về điểm số 0 và kết thúc hàm.
- Tạo một mảng gọi `calls` chứa các hàm để thực hiện các hành động di chuyển khác nhau tùy thuộc vào số lượng các ô cần di chuyển.
- Dựa vào độ dài của mảng `_cell`, gọi hàm tương ứng từ mảng `calls` để thực hiện các hành động di chuyển.
- Trả về tổng điểm số được tính sau các hành động di chuyển.

24. data.js:

```
var data = {
  score: 0,
  best: 0,
  cell: [

  ]
}
var indexes = [
  // left
  [
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11],
    [12, 13, 14, 15],
  ],
  // top
  [
    [0, 4, 8, 12],
    [1, 5, 9, 13],
    [2, 6, 10, 14],
    [3, 7, 11, 15],
  ],
  // right
  [
    [3, 2, 1, 0],
    [7, 6, 5, 4],
    [11, 10, 9, 8],
    [15, 14, 13, 12],
  ],
  // bottom
  [
    [12, 8, 4, 0],
    [13, 9, 5, 1],
    [14, 10, 6, 2],
    [15, 11, 7, 3],
  ]
]
```

Hình 64: Mô tả ảnh



Biến đối tượng data chứa các thuộc tính:

- **score**: Điểm số, được khởi tạo ban đầu là 0.
- **best**: Điểm cao nhất, được khởi tạo ban đầu là 0.
- **cell**: Một mảng trống, có thể được sử dụng để lưu trữ dữ liệu về các ô trong trò chơi nếu đoạn mã liên quan đến một trò chơi hoặc ứng dụng nào đó.

Mảng `indexs` chứa các mảng con đại diện cho các hướng khác nhau trên một lưới các ô:

- Các mảng con trong **`indexs`** đại diện cho các hàng hoặc cột trên lưới.
- Các số trong các mảng con đó là chỉ số của các ô trong lưới, được sắp xếp theo hướng tương ứng (trái, phải, trên, dưới).