

# Katarenga-co

## Table des matières

Prérequis .....	5
Installation sur Windows.....	5
Installation sur <b>macOS</b> .....	6
Installation sur <b>linux (Ubuntu/Debian)</b> .....	6
Télécharger le code source .....	7
Windows.....	7
<b>macOS</b> .....	8
Linux (Ubuntu/Debian).....	8
Lancement du jeu .....	8
Windows.....	9
macOS.....	9
Linux (Ubuntu/Debian).....	9
Justification du choix du langage et de la librairie graphique.....	10
2.1 CHOIX DU LANGAGE PYTHON .....	10
2.2 CHOIX DE LA BIBLIOTHÈQUE PYGAME .....	11
DESCRIPTION PRÉCISE DES STRUCTURES DE DONNÉES .....	12
3.1 MODÉLISATION DU PLATEAU DE JEU .....	12
3.2 MODÉLISATION DES QUADRANTS.....	13
3.3 MODÉLISATION DES PIONS .....	15
3.4 MODÉLISATION DES CAMPS (KATARENGA) .....	17
3.5 MODÉLISATION DES JOUEURS ET ÉTATS.....	20
3.6 STRUCTURES RÉSEAU .....	22
DESCRIPTION DES COMPOSANTS GRAPHIQUES UTILISÉS.....	27
4.1 ARCHITECTURE GÉNÉRALE DE L'AFFICHAGE .....	27
4.2 SYSTÈME DE RENDU DU PLATEAU DE JEU.....	28
4.3 SYSTÈME D'ANIMATION DES PIONS .....	30
4.4 INDICATEURS VISUELS SPÉCIALISÉS.....	34
4.5 INTERFACE UTILISATEUR ET MENUS .....	37
4.6 GESTION AUDIO INTÉGRÉE.....	39
4.7 MESSAGES DE VICTOIRE ET DIALOGUES .....	41
PRÉSENTATION DES ALGORITHMES DE GESTION DU DÉPLACEMENT DES PIONS .....	42

5.1 ARCHITECTURE GÉNÉRALE DU SYSTÈME DE DÉPLACEMENT .....	43
5.2 ALGORITHMES DE DÉPLACEMENT PAR TYPE DE CASE .....	44
5.3 GESTION SPÉCIALE POUR KATARENGA - ACCÈS AUX CAMPS .....	47
5.4 ALGORITHME DE VALIDATION DES MOUVEMENTS .....	49
5.5 GESTION DES DIFFÉRENCES ENTRE MODES DE JEU .....	50
5.6 OPTIMISATIONS ET PERFORMANCES .....	51
5.7 INTÉGRATION AVEC L'INTERFACE GRAPHIQUE .....	52
PRÉSENTATION DES ALGORITHMES DE GESTION DE VICTOIRE .....	53
6.1 ALGORITHMES DE VICTOIRE - KATARENGA.....	53
6.1.1 VICTOIRE PAR OCCUPATION DES CAMPS.....	53
6.1.2 VICTOIRE PAR ÉLIMINATION STRATÉGIQUE.....	56
6.2 ALGORITHMES DE VICTOIRE - CONGRESS .....	58
6.2.1 PRINCIPE DE CONNEXION.....	58
6.2.2 ALGORITHME DE RECHERCHE DE CONNEXION .....	59
6.2.3 AFFICHAGE VISUEL DE LA VICTOIRE .....	60
6.3 ALGORITHMES DE VICTOIRE - ISOLATION .....	61
6.3.1 PRINCIPE DE BLOCAGE .....	61
6.3.2 CALCUL DES POSITIONS SÛRES.....	62
6.3.3 RÈGLES D'ATTAQUE PAR TYPE DE CASE .....	63
6.3.4 PLACEMENT ET VÉRIFICATION .....	65
6.3.5 IA POUR L'ISOLATION .....	66
EXPLICATION DU PROCÉDÉ DE COMMUNICATION RÉSEAU .....	67
7.1 ARCHITECTURE DE COMMUNICATION GÉNÉRALE.....	67
7.2 ÉTABLISSEMENT DE LA CONNEXION .....	67
7.2.1 CÔTÉ SERVEUR (HÉBERGEMENT) .....	67
7.2.2 CÔTÉ CLIENT (CONNEXION).....	69
7.2.3 INTERFACE UTILISATEUR DE CONNEXION (network_menu.py).....	70
7.3 ÉCHANGE DE MESSAGES EN TEMPS RÉEL .....	71
7.3.1 MÉCANISME D'ENVOI.....	71
7.3.2 MÉCANISME DE RÉCEPTION.....	72
7.3.3 TRAITEMENT DES MESSAGES REÇUS.....	74
7.4 SYNCHRONISATION DES CONFIGURATIONS.....	77
7.4.1 ÉCHANGE DES QUADRANTS .....	77
7.4.2 ASSEMBLAGE DU PLATEAU FINAL .....	78
7.5 GESTION DES TOURS ET SYNCHRONISATION .....	79
7.5.1 ATTRIBUTION DES RÔLES .....	79

7.5.2 GESTION AVANCÉE DES ÉTATS DE VICTOIRE .....	80
7.6 GESTION DES ERREURS ET DÉCONNEXIONS.....	81
7.6.1 DÉTECTION DE DÉCONNEXION.....	81
7.6.2 INTERFACE DE VICTOIRE AMÉLIORÉE .....	82
7.7 SPÉCIFICITÉS PAR MODE DE JEU.....	85
7.7.1 KATARENGA.....	85
7.7.2 CONGRESS .....	86
7.7.3 ISOLATION .....	86

# Prérequis

## Installation sur Windows

### 1. Installer Python :

- Rendez-vous sur [python.org/downloads](https://python.org/downloads).
- Téléchargez la version 3.12.6 de Python pour Windows.
- Exécutez le fichier d'installation et cochez la case **"Add Python to PATH"** avant de cliquer sur **"Install Now"**.
- Vérifiez l'installation en ouvrant l'Invite de commandes et en tapant :  

```
python --version
```

### 2. Installer Pygame :

- Ouvrez l'Invite de commandes et tapez :  

```
pip install pygame
```
- Vérifiez l'installation avec :  

```
python -m pygame --version
```

### 3. Si pip n'est pas installé

- Ouvrez l'Invite de commandes et tapez :  

```
python -m pip install pygame
```

### 4. Installer Git

- Rendez-vous sur [git-scm.com/download/win](https://git-scm.com/download/win).
- Exécutez l'installateur et cochez :  
"Add Git to PATH" (essentiel pour utiliser git dans CMD).  
"Use Git from the Windows Command Prompt".
- Vérifiez l'installation avec :

git --version

## Installation sur **macOs**

### 1. Installer Python :

- Rendez-vous sur [python.org/downloads/macos](https://python.org/downloads/macos).
- Téléchargez la version 3.12.6 de Python pour macOS.
- Exécutez le fichier d'installation et cochez la case **"Add Python to PATH"** avant de cliquer sur **"Install Now"**.
- Vérifiez l'installation en ouvrant le Terminal et en tapant :

```
python3 --version
```

### 2. Installer Pygame :

- Dans le Terminal, tapez :  

```
python3 -m pip install pygame
```
- Vérifiez l'installation avec :  

```
python3 -m pygame --version
```

### 3. Installer Git

- Rendez-vous sur [git-scm.com/download/mac](https://git-scm.com/download/mac).
- Exécutez l'installeur
- Vérifiez l'installation avec :  

```
git --version
```

## Installation sur **linux (Ubuntu/Debian)**

### 1. Vérifier si Python 3 est installé

- Ouvrez un terminal (**Ctrl + Alt + T**) et tapez :

```
python3 --version
```

- → Si **Python 3.12.6** s'affiche, passez à l'étape 2.
- → Sinon, installez la version 3.12.6 (recommandée) avec :

```
sudo apt install python3.12
```

### 2. Installer Pip (si absent)

- Vérifiez si pip3 est disponible :

```
pip3 --version
```

- Si ce n'est pas le cas, installez-le :

```
sudo apt install python3-pip
```

### 3. Installer Pygame

- Utilisez pip3 pour installer Pygame :

```
pip3 install pygame
```

- *(Si vous avez plusieurs versions de Python, utilisez python3.X -m pip install pygame en remplaçant X par votre version.)*

### 4. Vérifier l'installation

- Lancez la commande :

```
python3 -m pygame --version
```

### 5. Installer Git via APT :

- `sudo apt update && sudo apt install git -y`

- Vérification :

```
git --version
```

## Télécharger le code source

### Windows


#### 1. Ouvrez l'Invite de commandes (CMD) :

- tapez **cmd** dans la barre de recherche Windows.

#### 2. Naviguez vers le Bureau :

- cd %USERPROFILE%\Desktop
- 3. Clonez le dépôt Github :
  - git clone <https://github.com/Opollo52/KATARENGA-CO>
- 4. Accédez au dossier cloné :
  - cd KATARENGA-CO

## macOS

1. Ouvrez le Terminal :
  -  + Espace → Tapez Terminal.
2. Naviguez vers le Bureau :
  - cd ~/Desktop
3. Clonez le dépôt Github :
  - git clone <https://github.com/Opollo52/KATARENGA-CO>
4. Accédez au dossier cloné :
  - cd KATARENGA-CO

## Linux (Ubuntu/Debian)

1. Ouvrez le Terminal :
  - Ctrl + Alt + T.
2. Naviguez vers le Bureau :
  - cd ~/Desktop
3. Clonez le dépôt Github :
  - git clone <https://github.com/Opollo52/KATARENGA-CO>
4. Accédez au dossier cloné :
  - cd KATARENGA-CO

## Lancement du jeu

Maintenant que le dossier KATARENGA-CO a été téléchargé, vous devez lancer le fichier principal du jeu.



## Windows

1. Ouvrir l'invite de commandes (CMD)
  - Appuyez sur Windows + R.
  - Tapez cmd et appuyez sur Entrée.
2. Se Déplacer vers le Dossier du Projet
  - `cd %USERPROFILE%\Desktop\KATARENGA-CO`
3. Vérifier le Contenu du Dossier
  - `Dir`
  - → Vous devriez voir le fichier main.py dans la liste
4. Lancer le Programme
  - `python main.py`

## macOS

1. Ouvrir le Terminal
  - Pressez `⌘` + Espace pour ouvrir Spotlight
  - Tapez Terminal et appuyez sur Entrée
2. Se Déplacer vers le Dossier du Projet
  - `cd ~/Desktop/KATARENGA-CO`
3. Vérifier le Contenu du Dossier
  - `Ls`
  - → Vous devriez voir main.py dans la liste
4. Lancer le Programme
  - `python3 main.py`
  - Si vous avez plusieurs versions :  
`python3.12 main.py`

## Linux (Ubuntu/Debian)

1. Ouvrir le Terminal
  - `Ctrl + Alt + T`
  - Ou via le menu Applications > Accessoires > Terminal
2. Se Déplacer vers le Dossier du Projet
  - `cd ~/Desktop/KATARENGA-CO`
3. Vérifier le Contenu du Dossier
  - `ls -l`

→ Doit afficher main.py avec les permissions (-rwxr-xr-x)

4. Installer les Dépendances (si nécessaire)
  - `sudo apt update`

- `sudo apt install python3 python3-pip python3-pygame`
- 5. Lancer le Programme
  - `python3 main.py`
  - Si vous avez plusieurs versions :  
`python3.12 main.py`

## Justification du choix du langage et de la librairie graphique.

### 2.1 CHOIX DU LANGAGE PYTHON

Nous avons choisi Python comme langage principal pour développer notre projet. Ce choix repose avant tout sur notre bonne maîtrise de Python, acquise au cours de notre formation. Bien que le C soit un langage performant, Python nous permet de coder plus rapidement, avec une syntaxe claire et lisible, ce qui facilite la compréhension du code, sa maintenance et le travail en équipe.

De plus, Python offre une large palette de bibliothèques, ce qui est un avantage important pour un projet intégrant des aspects graphiques, audio, réseau et fichiers. Cette richesse de l'écosystème Python nous a permis d'intégrer facilement :

- Gestion réseau : sockets TCP pour le mode multijoueur
- Manipulation de fichiers : JSON pour la configuration et sauvegarde des quadrants
- Threading : pour la réception asynchrone des messages réseau

#### ALTERNATIVES CONSIDÉRÉES :

- C : Rejeté pour sa complexité de développement disproportionnée par rapport aux besoins
- JavaScript : Non retenu car moins adapté aux applications desktop

## 2.2 CHOIX DE LA BIBLIOTHÈQUE PYGAME

Concernant la partie graphique, nous avons opté pour la bibliothèque Pygame. Ce choix s'explique par les fonctionnalités complètes qu'elle propose pour le développement de jeux, notamment :

- Gestion des surfaces : affichage 2D optimisé pour les jeux de plateau
- Événements : gestion naturelle des entrées clavier/souris
- Animations fluides : système de déplacement des pions avec interpolation
- Audio intégré : sons d'interface et effets sonores
- Performance suffisante : adaptée parfaitement aux jeux de plateau

Contrairement à d'autres bibliothèques comme Tkinter, qui sont plutôt adaptées à la création d'interfaces classiques, Pygame est entièrement orientée jeu vidéo. Elle permet une gestion fine de l'affichage et des ressources multimédia, ce qui nous a permis d'intégrer facilement des éléments sonores, des animations fluides et des mécanismes de contrôle adaptés au gameplay.

### COMPARAISON AVEC LES ALTERNATIVES :

- Tkinter : Interface trop rigide pour un jeu, animations limitées
- PyQt/PySide : Surdimensionné pour nos besoins, courbe d'apprentissage plus élevée
- Kivy : Plus adapté au mobile qu'au desktop
- Arcade : Plus moderne mais moins documenté que Pygame

Enfin, la documentation fournie et la communauté active autour de Pygame ont rendu son utilisation à la fois pratique et efficace tout au long du développement. Le large choix d'exemples et tutoriels disponibles nous a permis de résoudre rapidement les défis techniques rencontrés.

# DESCRIPTION PRÉCISE DES STRUCTURES DE DONNÉES

## 3.1 MODÉLISATION DU PLATEAU DE JEU

Le plateau de jeu est organisé autour d'une grille principale de 10x10 cases, permettant d'accueillir les trois modes de jeu (Katarena, Congress, Isolation) dans une architecture unifiée.

STRUCTURE PRINCIPALE :

```
```python
def create_game_board(quadrant_grid_data):
    """Crée un plateau de jeu à partir des données de grille des 4 quadrants"""
    # Créer une grille 10x10 vide pour le plateau
    board_grid = [[0 for j in range(10)] for i in range(10)]

    if len(quadrant_grid_data) == 4:
        # Quadrant 1 (haut gauche)
        for i in range(4):
            for j in range(4):
                board_grid[i + 1][j + 1] = quadrant_grid_data[0][i][j]

        # Quadrant 2 (haut droite)
        for i in range(4):
            for j in range(4):
                board_grid[i + 1][j + 5] = quadrant_grid_data[1][i][j]

        # Quadrant 3 (bas gauche)
        for i in range(4):
```

```

    for j in range(4):
        board_grid[i + 5][j + 1] = quadrant_grid_data[2][i][j]

# Quadrant 4 (bas droite)
for i in range(4):
    for j in range(4):
        board_grid[i + 5][j + 5] = quadrant_grid_data[3][i][j]

return board_grid
...

```

#### ORGANISATION DE LA GRILLE :

- Grille 10x10 (indices 0 à 9) contenant les valeurs des cases
- Zone de jeu effective : 8x8 au centre (indices 1 à 8)
- Zones spéciales Katarenga : coins du plateau (0,0), (0,9), (9,0), (9,9)

#### Valeurs des cases :

- 0 : Case vide (bordures)
- 1 : Case jaune (mouvement de fou)
- 2 : Case verte (mouvement de cavalier)
- 3 : Case bleue (mouvement de roi)
- 4 : Case rouge (mouvement de tour)

## 3.2 MODÉLISATION DES QUADRANTS

Chaque quadrant est défini par une structure JSON contenant ses données de configuration et sa représentation graphique.

STRUCTURE JSON D'UN QUADRANT :

```
```json
{
  "quadrant_1": {
    "image_path": "quadrant/quadrant/quadrant_1.png",
    "grid": [
      [1, 3, 2, 4],
      [4, 2, 1, 1],
      [2, 4, 3, 3],
      [3, 1, 4, 2]
    ]
  }
}
```
```

SYSTÈME DE ROTATION :

```
```python
def apply_rotation_to_grid(grid, rotation):
    """Applique la rotation spécifiée (0, 90, 180, 270) à une grille"""
    rotated_grid = [row.copy() for row in grid]
    num_rotations = rotation // 90
    for i in range(num_rotations):
        rotated_grid = rotate_grid(rotated_grid)
    return rotated_grid

def rotate_grid(grid):
    """Tourne une grille 4x4 de 90 degrés dans le sens horaire"""
    rows = len(grid)
```

```

cols = len(grid[0])
rotated = [[0 for j in range(rows)] for i in range(cols)]

for i in range(rows):
    for j in range(cols):
        rotated[j][rows - 1 - i] = grid[i][j]

return rotated
'''

UTILISATION DANS LA CONFIGURATION :

'''python
def get_quadrant_grid(quadrant_id, rotation):
    """Obtient la grille du quadrant avec la rotation spécifiée"""
    if quadrant_id in quadrants:
        original_grid = quadrants[quadrant_id].get("grid", [])
        return apply_rotation_to_grid(original_grid, rotation)
    return []
'''

```

### 3.3 MODÉLISATION DES PIONS

Les pions sont représentés dans une grille séparée de même dimension que le plateau, permettant une gestion indépendante des cases et des pièces.

```

STRUCTURE DES PIONS :

'''python
# Grille 10x10 pour les pions

```

```
pawn_grid = [[0 for j in range(10)] for i in range(10)]
```

```
# 0 = case vide, 1 = pion rouge, 2 = pion bleu
```

```
...
```

INITIALISATION PAR MODE DE JEU :

```
```python
```

```
def initialize_pawns_for_game_mode(game_mode):
```

```
    """Initialise les pions selon le mode de jeu sélectionné"""
```

```
    pawn_grid = [[0 for j in range(10)] for i in range(10)]
```

```
    if game_mode == 0: # Katarenga
```

```
        # Pions rouges ligne 1 (joueur 1) - colonnes 1 à 8
```

```
        for col in range(1, 9):
```

```
            pawn_grid[1][col] = 1
```

```
        # Pions bleus ligne 8 (joueur 2) - colonnes 1 à 8
```

```
        for col in range(1, 9):
```

```
            pawn_grid[8][col] = 2
```

```
    elif game_mode == 1: # Congress
```

```
        # Positions des pions rouges (joueur 1)
```

```
        red_positions = [
```

```
            (1, 2), (1, 5), (2, 8), (4, 1),
```

```
            (5, 8), (7, 1), (8, 4), (8, 7)
```

```
        ]
```

```
        # Positions des pions bleus (joueur 2)
```

```
        blue_positions = [
```

```
            (1, 4), (1, 7), (2, 1), (4, 8),
```

```
            (5, 1), (7, 8), (8, 2), (8, 5)
```



```

    ]

    for row, col in red_positions:
        pawn_grid[row][col] = 1
    for row, col in blue_positions:
        pawn_grid[row][col] = 2

    elif game_mode == 2: # Isolation
        # Plateau vide au début
        pass

    return pawn_grid
'''

```

CLASSE PION (pour gestion avancée) :

```

'''python
class Pawn:
    def __init__(self, row, col, color):
        self.row = row
        self.col = col
        self.color = color # 1 (rouge) ou 2 (bleu)
        self.selected = False
'''

```

### 3.4 MODÉLISATION DES CAMPS (KATARENGA)

Les camps représentent les zones d'objectif spécifiques au mode Katarenga, situées dans les coins du plateau étendu.

#### STRUCTURE DES CAMPS :

```
```python
# Variables globales pour les camps (en dehors du plateau 8x8)
camps_player1 = {"camp1": [], "camp2": []} # Camps du joueur 1 (rouge)
camps_player2 = {"camp1": [], "camp2": []} # Camps du joueur 2 (bleu)

def reset_camps():
    """Réinitialise les camps au début d'une partie"""
    global camps_player1, camps_player2
    camps_player1 = {"camp1": [], "camp2": []}
    camps_player2 = {"camp1": [], "camp2": []}
...
```
```

#### GESTION DES POSITIONS :

```
```python
def get_camp_positions(player):
    """Retourne les positions des camps pour un joueur"""
    if player == 1: # Joueur rouge vise les camps rouges (en bas)
        return [(9, 0), (9, 9)] # Camps rouges en bas
    else: # Joueur bleu vise les camps bleus (en haut)
        return [(0, 0), (0, 9)] # Camps bleus en haut

def is_camp_occupied(row, col, player):
    """Vérifie si un camp est déjà occupé AVANT de tenter le mouvement"""
    camp_positions = get_camp_positions(player)

    if (row, col) not in camp_positions:

```

```

return False # Ce n'est pas un camp

if player == 1: # Rouge vérifie ses camps rouges
    if (row, col) == (9, 0):
        return len(camps_player1["camp1"]) > 0
    else: # (9, 9)
        return len(camps_player1["camp2"]) > 0
else: # Bleu vérifie ses camps bleus
    if (row, col) == (0, 0):
        return len(camps_player2["camp1"]) > 0
    else: # (0, 9)
        return len(camps_player2["camp2"]) > 0
...

```

PLACEMENT DANS LES CAMPS :

```

```python
def place_in_camp(row, col, pawn_grid, player):
    """Place un pion dans un camp et le retire du jeu - LIMITE À UN PION PAR CAMP"""
    global camps_player1, camps_player2

    camp_positions = get_camp_positions(player)

    if (row, col) in camp_positions:
        if player == 1: # Rouge va dans ses camps rouges
            if (row, col) == (9, 0):
                if len(camps_player1["camp1"]) == 0: # Camp vide
                    camps_player1["camp1"].append(player)
                return True

```

```

        else:
            return False # Camp déjà occupé
    else: # (9, 9)
        if len(camps_player1["camp2"]) == 0: # Camp vide
            camps_player1["camp2"].append(player)
            return True
        else:
            return False # Camp déjà occupé
# ... logique similaire pour le joueur bleu

return False
...

```

### 3.5 MODÉLISATION DES JOUEURS ET ÉTATS

Les informations des joueurs et l'état de la partie sont gérés par des variables globales et des structures simples.

GESTION DES MODES DE JEU :

```

```python
# Variables globales dans game_modes.py

GLOBAL_SELECTED_GAME = 0 # 0=Katarena, 1=Congress, 2=Isolation
GLOBAL_SELECTED_OPPONENT = 0 # 0=Ordinateur, 1=Local, 2=Réseau
FIRST_RUN = True

def reset_game_state():
    """Réinitialise l'état du jeu"""

    global GLOBAL_SELECTED_GAME, GLOBAL_SELECTED_OPPONENT

```

```

# Les variables gardent leur valeur pour permettre la sélection
pass
...

```

ÉTAT DE LA PARTIE :

```

```python
# Variables dans game_board.py

current_player = 1      # Joueur actuel (1=rouge, 2=bleu)
game_over = False      # État de fin de partie
winner = 0             # Joueur gagnant (0=aucun, 1=rouge, 2=bleu)
selected_pawn = None    # Pion actuellement sélectionné (row, col)
possible_moves = []     # Liste des mouvements possibles
connected_pawns = []    # Pour Congress (pions connectés)

# Phase de jeu
game_phase = "play"     # "play", "menu", "setup"
...

```

GESTION DES ANIMATIONS :

```

```python
class Animation:
    def __init__(self):
        self.moving_pawn = None    # Pion en mouvement (row, col, end_row, end_col)
        self.move_start_time = 0   # Timestamp début animation
        self.move_duration = 0.8   # Durée animation en secondes
        self.start_pos = None       # Position écran début
        self.end_pos = None         # Position écran fin
        self.moving_pawn_color = None # Couleur du pion animé

```

```

self.pending_move = None    # Mouvement en attente d'exécution

self.animation_finished = False

def start_move(self, start_row, start_col, end_row, end_col,
               board_x, board_y, cell_size, pawn_color):
    """Démarré une animation de déplacement"""
    self.moving_pawn = (start_row, start_col, end_row, end_col)
    self.moving_pawn_color = pawn_color
    self.move_start_time = time.time()
    # Calcul des positions d'écran...
    ...

```

## 3.6 STRUCTURES RÉSEAU

Pour le mode multijoueur, les données sont échangées via un protocole JSON structuré utilisant des sockets TCP.

CLASSE PRINCIPALE (network\_manager.py) :

```

```python
class NetworkManager:
    def __init__(self):
        self.socket = None
        self.is_server = False
        self.is_connected = False
        self.connection = None
        self.received_messages = []
        self.message_lock = threading.Lock()

```

```

def send_message(self, message_type, data):
    """Envoie un message à l'autre joueur"""
    if not self.is_connected:
        return False

    try:
        message = {
            'type': message_type,
            'data': data,
            'timestamp': time.time()
        }

        json_message = json.dumps(message)
        self.connection.send(json_message.encode('utf-8'))
        return True

    except Exception as e:
        print(f"Erreur lors de l'envoi: {e}")
        return False
...

```

TYPES DE MESSAGES RÉELS (network\_game.py) :

1. **\*\*Message de mouvement : \*\***

```

```python
# Envoi
network_manager.send_message("move", {
    "from": from_pos,

```

```

        "to": to_pos,
        "player": my_player
    })

```

# Réception

```

if msg['type'] == 'move':
    from_pos = msg['data']['from']
    to_pos = msg['data']['to']
    ...

```

2. **\*\*Message de placement (Isolation) :\*\***

```

```python
# Envoi
network_manager.send_message("placement", {
    "position": position,
    "player": my_player
})

```

# Réception

```

elif msg['type'] == 'placement':
    row, col = msg['data']['position']
    player = msg['data']['player']
    ...

```

3. **\*\*Message de mode de jeu :\*\***

```

```python
# Envoi (serveur)
network_manager.send_message("game_mode", {"mode": local_game_mode})

```



```
# Réception (client)

if msg['type'] == 'game_mode':
    current_game_mode = msg['data']['mode']
...

```

4. **\*\*Message de victoire : \*\***

```
```python
# Envoi

network_manager.send_message("victory", {
    "winner": winner,
    "game_mode": current_game_mode
})
...

```

CONFIGURATION DES QUADRANTS (network\_quadrant\_setup.py) :

```
```python
def send_my_quadrants():
    """Envoie la configuration de mes quadrants à l'adversaire"""
    my_config = []
    for i, quad_id in enumerate(selected_quadrants):
        grid = get_quadrant_grid(quad_id, quadrant_rotations[i])
        my_config.append(grid)

    config_data = {
        "player": "server" if is_server else "client",
        "quadrants": my_config,
        "quadrant_indices": quadrant_indices,
    }

```

```

    "selected_ids": selected_quadrants,
    "rotations": quadrant_rotations
}

```

```

network_manager.send_message("quadrant_config", config_data)
...

```

RÉCEPTION ET TRAITEMENT (network\_game.py) :

```

``python
def process_messages():
    messages = network_manager.get_messages()
    for msg in messages:
        if msg['type'] == 'move':
            from_pos = msg['data']['from']
            to_pos = msg['data']['to']

            # Exécuter le mouvement reçu
            pawn_grid[to_pos[0]][to_pos[1]] = pawn_grid[from_pos[0]][from_pos[1]]
            pawn_grid[from_pos[0]][from_pos[1]] = 0

        elif msg['type'] == 'quadrant_config':
            opponent_config = msg['data']
            opponent_config_received = True
    ...

```

SYNCHRONISATION CLIENT-SERVEUR :

```

``python
# Serveur attend confirmation

```

```

if network_manager.is_server:
    network_manager.send_message("game_start", {"ready": True})
    # Attente de réponse...
else:
    # Client répond
    network_manager.send_message("game_start_confirm", {"ready": True})
'''

```

## DESCRIPTION DES COMPOSANTS GRAPHIQUES UTILISÉS

### 4.1 ARCHITECTURE GÉNÉRALE DE L'AFFICHAGE

Notre système graphique repose sur une architecture modulaire utilisant Pygame, organisée autour de plusieurs composants spécialisés qui gèrent différents aspects de l'interface utilisateur.

STRUCTURE PRINCIPALE :

- Affichage en plein écran adaptatif (pygame.FULLSCREEN)
- Système de coordonnées centrées dynamiquement selon la résolution
- Gestion des ressources (images, polices, sons) centralisée
- Interface responsive s'adaptant aux différentes tailles d'écran

COMPOSANTS PRINCIPAUX :

```

```python
# Initialisation de l'affichage principal
screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)

```

```

# Récupération dynamique des dimensions
current_width, current_height = screen.get_size()

# Calcul automatique de la taille des éléments
cell_size = min(current_width, current_height) // 10
board_size = 10 * cell_size
board_x = (current_width - board_size) // 2
board_y = (current_height - board_size) // 2
...

```

## 4.2 SYSTÈME DE RENDU DU PLATEAU DE JEU

Le plateau de jeu constitue l'élément graphique central, avec un système de rendu adaptatif qui s'ajuste selon le mode de jeu sélectionné.

RENDU DU PLATEAU (game\_board.py) :

```

```python
# Chargement et mise à l'échelle des textures de cases
images = {}

images[1] = pygame.image.load(PATH / "assets" / "img" / "yellow.png") # Fou
images[2] = pygame.image.load(PATH / "assets" / "img" / "green.png") # Cavalier
images[3] = pygame.image.load(PATH / "assets" / "img" / "blue.png") # Roi
images[4] = pygame.image.load(PATH / "assets" / "img" / "red.png") # Tour

# Redimensionnement dynamique selon la taille d'écran
for key in images:
    images[key] = pygame.transform.scale(images[key], (cell_size, cell_size))

```

```

# Affichage selon le mode de jeu

if current_game_mode == 0: # Katarenga - plateau 10x10
    for row in range(10):
        for col in range(10):
            if 1 <= row <= 8 and 1 <= col <= 8:
                cell_value = board_grid[row][col]
                cell_rect = pygame.Rect(
                    board_x + col * cell_size,
                    board_y + row * cell_size,
                    cell_size, cell_size
                )
                if cell_value in images:
                    screen.blit(images[cell_value], cell_rect)
                    pygame.draw.rect(screen, BLACK, cell_rect, 2)

else: # Congress et Isolation - zone 8x8
    for row in range(1, 9):
        for col in range(1, 9):
            cell_value = board_grid[row][col]
            cell_rect = pygame.Rect(
                board_x + col * cell_size,
                board_y + row * cell_size,
                cell_size, cell_size
            )
            if cell_value in images:
                screen.blit(images[cell_value], cell_rect)
                pygame.draw.rect(screen, BLACK, cell_rect, 2)
...

```

CADRE DÉCORATIF :

```
```python
# Chargement et affichage du cadre autour du plateau

frame_image = pygame.image.load(PATH / "assets" / "img" / "frame.png")

if frame_image:
    scaled_frame = pygame.transform.scale(frame_image, (
        board_size + 2 * cell_size,
        board_size + 2 * cell_size
    ))
    screen.blit(scaled_frame, (board_x - cell_size, board_y - cell_size))
```
```

=====

=====

## 4.3 SYSTÈME D'ANIMATION DES PIONS

L'animation des pions utilise un système d'interpolation temporelle pour créer des mouvements fluides entre les cases.

CLASSE D'ANIMATION (game\_board.py) :

```
```python
class Animation:
    def __init__(self):
        self.moving_pawn = None      # Pion en cours d'animation
        self.move_start_time = 0     # Timestamp de début
        self.move_duration = 0.8     # Durée en secondes
```

```

self.start_pos = None          # Position d'écran de départ
self.end_pos = None           # Position d'écran d'arrivée
self.moving_pawn_color = None  # Couleur du pion animé
self.pending_move = None      # Mouvement en attente d'exécution

def start_move(self, start_row, start_col, end_row, end_col,
               board_x, board_y, cell_size, pawn_color):
    """Démarré une animation de déplacement avec effet sonore"""
    self.moving_pawn = (start_row, start_col, end_row, end_col)
    self.moving_pawn_color = pawn_color
    self.move_start_time = time.time()

    # Calcul des positions d'écran
    self.start_pos = (
        board_x + start_col * cell_size + cell_size // 2,
        board_y + start_row * cell_size + cell_size // 2
    )
    self.end_pos = (
        board_x + end_col * cell_size + cell_size // 2,
        board_y + end_row * cell_size + cell_size // 2
    )

    # Effet sonore
    audio_manager.play_sound('pawn_move')

def get_current_pos(self):
    """Calcule la position interpolée actuelle"""
    if not self.moving_pawn:

```

```

        return None

    elapsed = time.time() - self.move_start_time
    progress = min(elapsed / self.move_duration, 1.0)
    progress = 1 - (1 - progress) ** 3 # Fonction d'easing cubique

    current_x = self.start_pos[0] + (self.end_pos[0] - self.start_pos[0]) * progress
    current_y = self.start_pos[1] + (self.end_pos[1] - self.start_pos[1]) * progress

    return (int(current_x), int(current_y))
...

```

#### RENDU DES PIONS ANIMÉS :

```

```python
def draw_animated_pawns(screen, pawn_grid, board_x, board_y, cell_size,
                        selected_pawn, animation, current_game_mode):
    """Dessine les pions avec gestion des animations"""

    DARK_RED = Colors.DARK_RED
    DARK_BLUE = Colors.DARK_BLUE
    BLACK = Colors.BLACK

    moving_pos = animation.get_current_pos()
    moving_pawn_info = animation.moving_pawn

    # Définir la zone d'affichage selon le mode
    if current_game_mode == 0: # Katarenga
        grid_range = range(10)
        offset_row, offset_col = 0, 0

```



```

else: # Congress et Isolation

    grid_range = range(1, 9)

    offset_row, offset_col = 1, 1

# Dessiner les pions statiques
for row in grid_range:
    for col in grid_range:
        if pawn_grid[row][col] > 0:
            # Ignorer le pion en cours d'animation
            if (moving_pawn_info and
                moving_pawn_info[0] == row and moving_pawn_info[1] == col):
                continue

            pawn_color = DARK_RED if pawn_grid[row][col] == 1 else DARK_BLUE

            display_row = row - offset_row
            display_col = col - offset_col

            center = (
                board_x + display_col * cell_size + cell_size // 2,
                board_y + display_row * cell_size + cell_size // 2
            )

            radius = cell_size // 3

            # Surbrillance de sélection
            if selected_pawn and selected_pawn == (row, col):
                pygame.draw.circle(screen, (255, 255, 0), center, radius + 4, 3)

            # Pion principal

```

```

pygame.draw.circle(screen, pawn_color, center, radius)

pygame.draw.circle(screen, BLACK, center, radius, 2)

# Dessiner le pion en mouvement

if moving_pos and moving_pawn_info and animation.moving_pawn_color:
    pawn_color = DARK_RED if animation.moving_pawn_color == 1 else DARK_BLUE
    radius = cell_size // 3

    pygame.draw.circle(screen, pawn_color, moving_pos, radius)
    pygame.draw.circle(screen, BLACK, moving_pos, radius, 2)
...

```

## 4.4 INDICATEURS VISUELS SPÉCIALISÉS

Chaque mode de jeu dispose d'indicateurs visuels spécifiques pour améliorer l'expérience utilisateur.

MOUVEMENTS POSSIBLES (pawn.py) :

```

```python
def highlight_possible_moves(screen, possible_moves, board_x, board_y, cell_size):
    """Met en surbrillance les mouvements possibles avec des points noirs"""

    dot_radius = cell_size // 8

    for r, c in possible_moves:
        x = board_x + c * cell_size + cell_size // 2
        y = board_y + r * cell_size + cell_size // 2
        pygame.draw.circle(screen, (0, 0, 0), (x, y), dot_radius)
...

```

CROIX D'INTERDICTION (ISOLATION) :

```
```python
def show_invalid_positions_isolation(screen, pawn_grid, board_grid, board_x, board_y,
cell_size):

    """Affiche des croix noires sur les positions interdites en mode Isolation"""

    # Calcul des positions interdites selon les règles d'attaque

    forbidden_positions = set()

    # [Logique de calcul des positions interdites...]

    # Dessiner les croix

    for row, col in forbidden_positions:

        if pawn_grid[row][col] == 0: # Case vide

            x = board_x + col * cell_size

            y = board_y + row * cell_size

            margin = cell_size // 4

            # Croix noire épaisse

            pygame.draw.line(screen, (0, 0, 0),

                             (x + margin, y + margin),

                             (x + cell_size - margin, y + cell_size - margin), 6)

            pygame.draw.line(screen, (0, 0, 0),

                             (x + cell_size - margin, y + margin),

                             (x + margin, y + cell_size - margin), 6)

...
```
```

CAMPS KATARENGA :

```
```python
```

```

def draw_camps(screen, board_x, board_y, cell_size):
    """Dessine les camps dans les coins du plateau"""
    RED = (200, 50, 50)
    BLUE = (50, 50, 200)
    BLACK = (0, 0, 0)

    camp_positions = [
        (0, 0), (0, 9), # Camps bleus en haut
        (9, 0), (9, 9) # Camps rouges en bas
    ]

    for row, col in camp_positions:
        camp_rect = pygame.Rect(
            board_x + col * cell_size,
            board_y + row * cell_size,
            cell_size, cell_size
        )

        # Vérifier si le camp contient un pion
        has_pions = False
        pion_color = None

        # [Logique de vérification...]

        if has_pions:
            # Dessiner le pion dans le camp
            pygame.draw.circle(screen, pion_color, camp_rect.center, cell_size // 3)
            pygame.draw.circle(screen, BLACK, camp_rect.center, cell_size // 3, 3)

```

```
...
```

SURBRILLANCE CONGRESS :

```
```python
def highlight_connected_pawns(screen, connected_pawns, board_x, board_y, cell_size,
player_color):
    """Met en surbrillance les pions connectés pour la victoire"""
    highlight_color = (255, 255, 0) # Jaune
    highlight_thickness = 3

    for row, col in connected_pawns:
        x = board_x + col * cell_size + cell_size // 2
        y = board_y + row * cell_size + cell_size // 2
        radius = cell_size // 3 + 5
        pygame.draw.circle(screen, highlight_color, (x, y), radius, highlight_thickness)
...
```
```

## 4.5 INTERFACE UTILISATEUR ET MENUS

Le système d'interface utilise un design cohérent avec des éléments graphiques adaptatifs et des effets sonores.

SYSTÈME DE COULEURS (colors.py) :

```
```python
class Colors:
    DARK_RED = (200, 0, 0)
    DARK_BLUE = (0, 0, 150)
    BLACK = (0, 0, 0)
    WHITE = (255, 255, 255)
...
```
```

```

GREEN = (176, 242, 194)
BLUE = (169, 203, 215)
RED = (255, 105, 97)
LIGHT_GRAY = (240, 240, 240)
DARK_GRAY = (100, 100, 100)
HIGHLIGHT = (255, 220, 120)
...

```

MENUS ADAPTATIFS (hub.py, game\_modes.py) :

```

``python
# Configuration dynamique des boutons
screen_rect = screen.get_rect()
texts = ["Jouer", "Quadrant", "Paramètres", "Quitter"]
max_width = max(font.render(text, True, WHITE).get_width() for text in texts)
button_width = max_width + 40
button_height = 50

# Centrage automatique
start_y = (screen_rect.height - total_height) // 2
buttons = [
    pygame.Rect((screen_rect.width - button_width)//2,
                start_y + i * (button_height + spacing),
                button_width, button_height)
    for i in range(len(texts))
]

# Rendu avec effets visuels
def draw_centered_text(text, rect, color):

```

```

text_surf = font.render(text, True, color)
text_rect = text_surf.get_rect(center=rect.center)
screen.blit(text_surf, text_rect)

for i, (rect, color) in enumerate(zip(buttons, colors)):
    pygame.draw.rect(screen, color, rect)
    # Bordure pour effet visuel
    pygame.draw.rect(screen, tuple(min(c + 30, 255) for c in color), rect, 2)
    draw_centered_text(texts[i], rect, BLACK)
...

```

FOND SEMI-TRANSPARENT :

```

```python
# Création de fonds transparents pour les textes
bg_surface = pygame.Surface((title_rect.width + 20, title_rect.height + 10),
pygame.SRCALPHA)
bg_surface.fill((255, 255, 255, 200)) # Blanc semi-transparent
screen.blit(bg_surface, (title_rect.x - 10, title_rect.y - 5))
screen.blit(title, title_rect)
...

```

## 4.6 GESTION AUDIO INTÉGRÉE

Le système audio est intégré aux composants graphiques pour fournir un retour sonore cohérent.

GESTIONNAIRE AUDIO (audio\_manager.py) :

```

```python
class AudioManager:

```

```

def __init__(self):
    self.sounds = {}
    self.settings = {'enabled': True, 'volume': 0.7}

def play_sound(self, sound_name):
    """Joue un son si audio activé"""
    if not self.settings['enabled']:
        return

    if sound_name in self.sounds:
        try:
            self.sounds[sound_name].play()
        except pygame.error as e:
            print(f"Erreur lecture {sound_name}: {e}")

```

```

# Instance globale
audio_manager = AudioManager()
...

```

## INTÉGRATION DANS LES COMPOSANTS :

```

```python
# Dans les boutons
if button.collidepoint(event.pos):
    audio_manager.play_sound('button_click')

```

# Dans les animations de pions

```

def start_move(...):
    # [Code d'animation...]

```



```
audio_manager.play_sound('pawn_move')
...
```

## 4.7 MESSAGES DE VICTOIRE ET DIALOGUES

Le système de dialogue utilise des superpositions graphiques avec transparence pour les messages importants.

MESSAGE DE VICTOIRE (game\_board.py) :

```
```python
def display_victory_message(screen, winner):
    """Affiche un message de victoire avec boutons interactifs"""
    message_color = (0, 150, 0)
    text_color = (255, 255, 255)

    # Texte principal
    font = pygame.font.Font(None, 48)
    message = f"Victoire du joueur {'Rouge' if winner == 1 else 'Bleu'} !"
    text = font.render(message, True, text_color)
    text_rect = text.get_rect(center=(screen.get_width() // 2, screen.get_height() // 2 - 60))

    # Fond semi-transparent
    message_rect = pygame.Rect(
        (screen.get_width() - message_width) // 2,
        (screen.get_height() - message_height) // 2,
        message_width, message_height
    )
```

```

s = pygame.Surface((message_rect.width, message_rect.height), pygame.SRCALPHA)
s.fill((message_color[0], message_color[1], message_color[2], 200))
screen.blit(s, message_rect)

# Bordure et contenu
pygame.draw.rect(screen, message_color, message_rect, 3)
screen.blit(text, text_rect)

# Boutons interactifs
rejouer_button = pygame.Rect(...)
quitter_button = pygame.Rect(...)

pygame.draw.rect(screen, Colors.BLUE, rejouer_button)
pygame.draw.rect(screen, Colors.RED, quitter_button)

# Textes des boutons
rejouer_text = button_font.render("Rejouer", True, Colors.WHITE)
quitter_text = button_font.render("Quitter", True, Colors.WHITE)

screen.blit(rejouer_text, rejouer_text.get_rect(center=rejouer_button.center))
screen.blit(quitter_text, quitter_text.get_rect(center=quitter_button.center))
...

```

## PRÉSENTATION DES ALGORITHMES DE GESTION DU DÉPLACEMENT DES PIONS

## 5.1 ARCHITECTURE GÉNÉRALE DU SYSTÈME DE DÉPLACEMENT

Notre système de déplacement utilise une approche unifiée basée sur la couleur de la case où se trouve le pion, permettant de gérer les trois modes de jeu avec une logique cohérente.

PRINCIPE FONDAMENTAL :

Le type de mouvement d'un pion dépend de la couleur de la case sur laquelle il se trouve :

- Case jaune (1) : Mouvement de fou (diagonales)
- Case verte (2) : Mouvement de cavalier (en L)
- Case bleue (3) : Mouvement de roi (8 directions, 1 case)
- Case rouge (4) : Mouvement de tour (lignes droites)

FONCTION PRINCIPALE (pawn.py) :

```
```python
def get_valid_moves(row, col, board_grid, pawn_grid, game_mode=None):
    """
    Obtenir les mouvements valides d'un pion à une position donnée.
    Gestion unifiée pour tous les modes de jeu avec grille 10x10 harmonisée.
    """
    # Utiliser le mode global si non spécifié
    if game_mode is None:
        game_mode = GLOBAL_SELECTED_GAME

    # Isolation: aucun déplacement de pions, seulement placement
    if game_mode == 2:
        return []

    # Vérifier s'il y a un pion à cette position
    if pawn_grid[row][col] == 0:
        return []

    # Couleur du pion et de la case
    pawn_color = pawn_grid[row][col]
    cell_color = board_grid[row][col]
```

```

possible_moves = []

# Déterminer les limites du plateau selon le mode
if game_mode == 0: # Katarenga - grille complète 10x10
    min_coord = 0
    max_coord = 10
    playable_min = 1
    playable_max = 8
else: # Congress - zone 8x8 dans grille 10x10
    min_coord = 1
    max_coord = 9
    playable_min = 1
    playable_max = 8
...

```

## 5.2 ALGORITHMES DE DÉPLACEMENT PAR TYPE DE CASE

Chaque couleur de case correspond à un algorithme de déplacement spécifique inspiré des pièces d'échecs.

MOUVEMENT DE ROI (CASE BLEUE) :

```

```python
if cell_color == 3: # Bleu: déplacement en roi
    directions = [
        (-1, -1), (-1, 0), (-1, 1),
        (0, -1),      (0, 1),
        (1, -1), (1, 0), (1, 1)
    ]

    for move_row, move_column in directions:
        r, c = row + move_row, col + move_column
        if min_coord <= r < max_coord and min_coord <= c < max_coord:
            # Zone de jeu normale

```

```

if playable_min <= r <= playable_max and playable_min <= c <= playable_max:
    # Si la case est vide, toujours autorisé
    if pawn_grid[r][c] == 0:
        possible_moves.append((r, c))
    # Si la case contient un pion ennemi et que le mode est Katarenga
    elif pawn_grid[r][c] != pawn_color and game_mode == 0:
        possible_moves.append((r, c))
    # Congress: pas de capture
    elif game_mode == 1 and pawn_grid[r][c] == 0:
        possible_moves.append((r, c))
...

```

MOUVEMENT DE TOUR (CASE ROUGE) :

```

```python
elif cell_color == 4: # Rouge: déplacement en tour
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for move_row, move_column in directions:
        r, c = row + move_row, col + move_column
        # Continuer dans cette direction jusqu'à rencontrer un obstacle
        while min_coord <= r < max_coord and min_coord <= c < max_coord:
            # Zone de jeu normale
            if playable_min <= r <= playable_max and playable_min <= c <= playable_max:
                # Si la case est vide
                if pawn_grid[r][c] == 0:
                    possible_moves.append((r, c))
            else:
                # Si la case contient un pion ennemi et que le mode autorise la capture (Katarenga)
                if pawn_grid[r][c] != pawn_color and game_mode == 0:
                    possible_moves.append((r, c))
                break # On ne peut pas aller plus loin

        # RÈGLE SPÉCIALE: Si c'est aussi une case rouge, c'est la dernière case accessible
        if board_grid[r][c] == 4:

```

```

        break

    else:

        break

    # Avancer d'une case dans la même direction
    r += move_row
    c += move_column
...

```

MOUVEMENT DE FOU (CASE JAUNE) :

```

```python
elif cell_color == 1: # Jaune: déplacement en fou
    directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]

    for move_row, move_column in directions:
        r, c = row + move_row, col + move_column

        # Continuer dans cette direction jusqu'à rencontrer un obstacle
        while min_coord <= r < max_coord and min_coord <= c < max_coord:

            # Zone de jeu normale
            if playable_min <= r <= playable_max and playable_min <= c <= playable_max:

                # Si la case est vide
                if pawn_grid[r][c] == 0:
                    possible_moves.append((r, c))
                else:
                    # Si la case contient un pion ennemi et que le mode est Katarenga
                    if pawn_grid[r][c] != pawn_color and game_mode == 0:
                        possible_moves.append((r, c))
                    break # On ne peut pas aller plus loin

            # RÈGLE SPÉCIALE: Si c'est aussi une case jaune, c'est la dernière case accessible
            if board_grid[r][c] == 1: # Jaune
                break
    else:
        break

```

```

        # Avancer d'une case dans la même direction
        r += move_row
        c += move_column
    ...

```

#### MOUVEMENT DE CAVALIER (CASE VERTE) :

```

'''python
elif cell_color == 2: # Vert: déplacement en cavalier
    knight_moves = [
        (-2, -1), (-2, 1),
        (-1, -2), (-1, 2),
        (1, -2), (1, 2),
        (2, -1), (2, 1)
    ]

    for move_row, move_column in knight_moves:
        r, c = row + move_row, col + move_column

        if min_coord <= r < max_coord and min_coord <= c < max_coord:
            # Zone de jeu normale

            if playable_min <= r <= playable_max and playable_min <= c <= playable_max:
                # Si la case est vide
                if pawn_grid[r][c] == 0:
                    possible_moves.append((r, c))

                # Si la case contient un pion ennemi et que le mode est Katarenga
                elif pawn_grid[r][c] != pawn_color and game_mode == 0:
                    possible_moves.append((r, c))

                # Congress: pas de capture
                elif game_mode == 1 and pawn_grid[r][c] == 0:
                    possible_moves.append((r, c))
    ...
'''

```

### 5.3 GESTION SPÉCIALE POUR KATARENGA - ACCÈS AUX CAMPS

Le mode Katarenga ajoute une mécanique spéciale permettant l'accès aux camps depuis la ligne de base adverse.

VÉRIFICATION DE LA LIGNE DE BASE (katarenga.py) :

```
```python
def is_on_enemy_baseline(row, player):
    """Vérifie si le pion est sur la ligne de base ennemie"""
    if player == 1: # Joueur rouge
        return row == 8 # Ligne de base du joueur bleu
    else: # Joueur bleu
        return row == 1 # Ligne de base du joueur rouge
...

```

ACCÈS AUX CAMPS :

```
```python
# Dans get_valid_moves() - Section Katarenga
camp_moves = []
if game_mode == 0:
    from jeux.katarenga import is_on_enemy_baseline, get_camp_positions, is_camp_occupied
    if is_on_enemy_baseline(row, pawn_color):
        camp_positions = get_camp_positions(pawn_color)
        for camp_row, camp_col in camp_positions:
            # Vérifier que le camp n'est pas déjà occupé (limite 1 pion par camp)
            if not is_camp_occupied(camp_row, camp_col, pawn_color):
                camp_moves.append((camp_row, camp_col))

# Ajouter les mouvements vers les camps
possible_moves.extend(camp_moves)
...

```

POSITIONS DES CAMPS :

```
```python
def get_camp_positions(player):
    """Retourne les positions des camps pour un joueur"""

```



```

if player == 1: # Joueur rouge vise les camps rouges (en bas)
    return [(9, 0), (9, 9)] # Camps rouges en bas
else: # Joueur bleu vise les camps bleus (en haut)
    return [(0, 0), (0, 9)] # Camps bleus en haut
...

```

## 5.4 ALGORITHME DE VALIDATION DES MOUVEMENTS

Le système inclut une fonction de validation pour vérifier la légalité d'un mouvement avant son exécution.

VALIDATION PRINCIPALE (pawn.py) :

```

```python
def is_valid_move(from_row, from_col, to_row, to_col, board_grid, pawn_grid, game_mode=None):
    """
    Vérifie si un mouvement d'une position à une autre est valide
    """
    # Vérifier s'il y a un pion à la position de départ
    if pawn_grid[from_row][from_col] == 0:
        return False

    # Calculer les mouvements possibles
    possible_moves = get_valid_moves(from_row, from_col, board_grid, pawn_grid, game_mode)

    # Vérifier si la position d'arrivée est dans les mouvements possibles
    return (to_row, to_col) in possible_moves
...

```

UTILISATION DANS LE JEU (game\_board.py) :

```

```python
# Lors du clic sur une case

if selected_pawn:
    selected_row, selected_col = selected_pawn

```

```

# Vérifier si le clic est sur l'un des mouvements possibles
if (row, col) in possible_moves:
    # Mouvement valide - démarrer l'animation
    animation.start_move(selected_row, selected_col, row, col,
                          board_x, board_y, cell_size,
                          pawn_grid[selected_row][selected_col])
...

```

## 5.5 GESTION DES DIFFÉRENCES ENTRE MODES DE JEU

Chaque mode de jeu applique des règles spécifiques sur les mouvements calculés.

RÈGLES DE CAPTURE :

```

```python
# Katarenga : Capture autorisée
if pawn_grid[r][c] != pawn_color and game_mode == 0:
    possible_moves.append((r, c))

# Congress : Pas de capture, cases vides uniquement
elif game_mode == 1 and pawn_grid[r][c] == 0:
    possible_moves.append((r, c))

# Isolation : Pas de déplacement de pions (géré séparément)
...

```

ZONES DE JEU :

```

```python
# Katarenga : Accès à la grille complète 10x10 + camps
if game_mode == 0:
    min_coord = 0
    max_coord = 10
    # + gestion spéciale des camps

# Congress/Isolation : Zone restreinte 8x8

```

else:

```
    min_coord = 1
    max_coord = 9
    playable_min = 1
    playable_max = 8
```

...

## 5.6 OPTIMISATIONS ET PERFORMANCES

Le système inclut plusieurs optimisations pour améliorer les performances lors du calcul des mouvements.

CACHE DES MOUVEMENTS :

```
```python
# Les mouvements sont recalculés uniquement lors de la sélection d'un nouveau pion
if selected_pawn != previous_selected_pawn:
    possible_moves = get_valid_moves(row, col, board_grid, pawn_grid, current_game_mode)
    previous_selected_pawn = selected_pawn
...
```
```

ARRÊT PRÉCOCE :

```
```python
# Dans les mouvements linéaires (tour, fou), arrêt dès qu'un obstacle est rencontré
if pawn_grid[r][c] != 0:
    # Traiter la capture si autorisée, puis arrêter
    if pawn_grid[r][c] != pawn_color and game_mode == 0:
        possible_moves.append((r, c))
    break # Arrêt immédiat
...
```
```

VÉRIFICATION DES LIMITES :

```
```python
# Vérification rapide des limites avant calculs complexes
if not (min_coord <= r < max_coord and min_coord <= c < max_coord):
    continue # Ignorer cette direction
...
```
```

```
...
```

RÉUTILISATION DE CODE :

```
```python
# Fonction utilitaire réutilisée dans game_board.py
def get_valid_moves_with_mode(row, col, board_grid, pawn_grid, game_mode):
    """Version locale qui utilise pawn.py pour tout"""
    from plateau.pawn import get_valid_moves
    return get_valid_moves(row, col, board_grid, pawn_grid, game_mode)
...

```

## 5.7 INTÉGRATION AVEC L'INTERFACE GRAPHIQUE

Les algorithmes de déplacement sont étroitement intégrés avec le système graphique pour fournir un retour visuel immédiat.

AFFICHAGE DES MOUVEMENTS POSSIBLES (pawn.py) :

```
```python
def highlight_possible_moves(screen, possible_moves, board_x, board_y, cell_size):
    """Met en surbrillance les mouvements possibles sur l'écran avec des petits points noirs"""
    dot_radius = cell_size // 8

    for r, c in possible_moves:
        x = board_x + c * cell_size + cell_size // 2
        y = board_y + r * cell_size + cell_size // 2
        pygame.draw.circle(screen, (0, 0, 0), (x, y), dot_radius)
...

```

GESTION DES CLICS :

```
```python
# Dans game_board.py - Boucle d'événements
if event.type == pygame.MOUSEBUTTONDOWN:
    if not game_over and not animation.is_moving():

```

```

mouse_x, mouse_y = event.pos
col = (mouse_x - board_x) // cell_size
row = (mouse_y - board_y) // cell_size

if selected_pawn:
    # Vérifier si le clic est sur un mouvement valide
    if (row, col) in possible_moves:
        # Exécuter le mouvement
        animation.start_move(...)
    elif pawn_grid[row][col] == current_player:
        # Sélectionner un nouveau pion
        selected_pawn = (row, col)
        possible_moves = get_valid_moves(row, col, board_grid, pawn_grid, current_game_mode)
else:
    # Première sélection
    if pawn_grid[row][col] == current_player:
        selected_pawn = (row, col)
        possible_moves = get_valid_moves(row, col, board_grid, pawn_grid, current_game_mode)
...

```

## PRÉSENTATION DES ALGORITHMES DE GESTION DE VICTOIRE

### 6.1 ALGORITHMES DE VICTOIRE - KATARENGA

Le jeu Katarenga propose deux mécanismes de victoire distincts : l'occupation des camps et l'élimination stratégique de l'adversaire.

#### 6.1.1 VICTOIRE PAR OCCUPATION DES CAMPS

PRINCIPE :

Un joueur remporte la partie en plaçant un pion dans chacun de ses deux camps dédiés.

STRUCTURE DES CAMPS (katarenga.py) :

```
```python
# Variables globales pour les camps (en dehors du plateau 8x8)
camps_player1 = {"camp1": [], "camp2": []} # Camps du joueur 1 (rouge)
camps_player2 = {"camp1": [], "camp2": []} # Camps du joueur 2 (bleu)

def reset_camps():
    """Réinitialise les camps au début d'une partie"""
    global camps_player1, camps_player2
    camps_player1 = {"camp1": [], "camp2": []}
    camps_player2 = {"camp1": [], "camp2": []}
...
```
```

ALGORITHME DE DÉTECTION :

```
```python
def check_katarenga_victory():
    """Vérifie si un joueur a gagné en occupant ses 2 camps"""

    # Joueur 1 (rouge) gagne s'il occupe ses 2 camps rouges
    camps_occupied_by_player1 = 0
    if len(camps_player1["camp1"]) > 0:
        camps_occupied_by_player1 += 1
    if len(camps_player1["camp2"]) > 0:
        camps_occupied_by_player1 += 1

    if camps_occupied_by_player1 >= 2:
        return 1 # Rouge gagne

    # Joueur 2 (bleu) gagne s'il occupe ses 2 camps bleus
    camps_occupied_by_player2 = 0
    if len(camps_player2["camp1"]) > 0:
        camps_occupied_by_player2 += 1
...
```
```

```

if len(camps_player2["camp2"]) > 0:
    camps_occupied_by_player2 += 1

if camps_occupied_by_player2 >= 2:
    return 2 # Bleu gagne

return 0 # Pas de gagnant
...

```

#### MÉCANISME D'ACCÈS AUX CAMPS :

```

```python
def place_in_camp(row, col, pawn_grid, player):
    """Place un pion dans un camp et le retire du jeu - LIMITE À UN PION PAR CAMP"""
    global camps_player1, camps_player2

    camp_positions = get_camp_positions(player)

    if (row, col) in camp_positions:
        if player == 1: # Rouge va dans ses camps rouges
            if (row, col) == (9, 0):
                if len(camps_player1["camp1"]) == 0: # Camp vide
                    camps_player1["camp1"].append(player)
                    return True
            else:
                return False # Camp déjà occupé
        else: # (9, 9)
            if len(camps_player1["camp2"]) == 0: # Camp vide
                camps_player1["camp2"].append(player)
                return True
            else:
                return False # Camp déjà occupé
    else: # Bleu va dans ses camps bleus
        if (row, col) == (0, 0):
            if len(camps_player2["camp1"]) == 0: # Camp vide

```

```

        camps_player2["camp1"].append(player)
        return True
    else:
        return False # Camp déjà occupé
else: # (0, 9)
    if len(camps_player2["camp2"]) == 0: # Camp vide
        camps_player2["camp2"].append(player)
        return True
    else:
        return False # Camp déjà occupé

return False
...

```

## 6.1.2 VICTOIRE PAR ÉLIMINATION STRATÉGIQUE

PRINCIPE MODIFIÉ :

Un joueur perd s'il n'a plus qu'un seul pion sur le plateau ET qu'il n'a aucun pion dans ses camps.

ALGORITHME DE VÉRIFICATION :

```

```python
def check_minimum_pawn_victory_condition(pawn_grid, game_mode):
    """
    NOUVELLES RÈGLES :
    - Si un joueur a un pion dans un camp, il peut continuer même avec 1 seul pion sur le plateau
    - Si un joueur n'a pas de pion dans un camp ET qu'il ne lui reste qu'1 pion sur le plateau, il perd
    """
    if game_mode != 0:
        return 0 # Ne s'applique qu'à Katarenga

    # Compter les pions sur le plateau
    red_pawns = sum(row.count(1) for row in pawn_grid)
    blue_pawns = sum(row.count(2) for row in pawn_grid)

```



```

# Vérifier si les joueurs ont des pions dans leurs camps
red_has_camp_pion = has_pion_in_camp(1)
blue_has_camp_pion = has_pion_in_camp(2)

# Règles modifiées pour la victoire par élimination
if red_pawns == 1 and not red_has_camp_pion:
    return 2 # Bleu gagne : Rouge n'a qu'1 pion et aucun pion dans un camp
elif blue_pawns == 1 and not blue_has_camp_pion:
    return 1 # Rouge gagne : Bleu n'a qu'1 pion et aucun pion dans un camp
elif red_pawns == 0:
    return 2 # Bleu gagne : Rouge n'a plus de pions
elif blue_pawns == 0:
    return 1 # Rouge gagne : Bleu n'a plus de pions

return 0 # Aucun gagnant encore

def has_pion_in_camp(player):
    """Vérifie si le joueur a au moins un pion dans ses camps"""
    if player == 1: # Rouge
        return len(camps_player1["camp1"]) > 0 or len(camps_player1["camp2"]) > 0
    else: # Bleu
        return len(camps_player2["camp1"]) > 0 or len(camps_player2["camp2"]) > 0
    ...

INTÉGRATION DANS LE JEU (game_board.py) :
```python
# Après chaque mouvement, vérifier les conditions de victoire
if not animation.is_moving() and animation.has_pending_move():
    winner_result, connected_result, game_over_result = animation.execute_pending_move(pawn_grid,
current_game_mode)

    if winner_result is not None:
        winner = winner_result
        game_over = game_over_result

```

```

# Vérifier si un joueur n'a plus assez de pions pour gagner
if current_game_mode == 0 and not game_over:
    forced_victory = check_minimum_pawn_victory_condition(pawn_grid, current_game_mode)
    if forced_victory > 0:
        winner = forced_victory
        game_over = True
...

```

## 6.2 ALGORITHMES DE VICTOIRE - CONGRESS

Le jeu Congress se base sur un objectif unique : connecter tous ses pions pour former un bloc continu.

### 6.2.1 PRINCIPE DE CONNEXION

RÈGLE DE VICTOIRE :

Un joueur gagne lorsque tous ses pions forment un groupe connecté par adjacence horizontale ou verticale (pas diagonale).

ALGORITHME PRINCIPAL (congress.py) :

```

```python
def check_victory(pawn_grid):
    """
    Vérifie si l'une des conditions de victoire du mode Congress est remplie.
    """
    # Vérifier la victoire pour chaque joueur
    for player in [1, 2]:
        player_pawns = []

        # Collecter toutes les positions des pions du joueur
        for row in range(8):
            for col in range(8):
                if pawn_grid[row][col] == player:
                    player_pawns.append((row, col))

```

```

# Si le joueur n'a pas de pions, il a perdu
if not player_pawns:
    continue

# Vérifier si tous les pions forment un bloc connecté
connected_pawns = find_connected_pawns(pawn_grid, player_pawns[0], player)

# Si tous les pions du joueur sont connectés, c'est une victoire
if len(connected_pawns) == len(player_pawns):
    return player, connected_pawns

# Aucune victoire
return 0, []
...

```

## 6.2.2 ALGORITHME DE RECHERCHE DE CONNEXION

EXPLORATION RÉCURSIVE :

```

```python
def connected(row, col, pawn_grid, player, visited, directions):
    """
    Fonction récursive pour explorer les pions connectés.
    """
    # Si la position est déjà visitée ou n'appartient pas au joueur, on arrête
    if (row, col) in visited or pawn_grid[row][col] != player:
        return

    # Marquer la position comme visitée
    visited.append((row, col))

    # Explorer les 4 directions adjacentes (pas de diagonales)
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc

```

```

        # Vérifier si dans les limites de la grille 8x8
        if 0 <= new_row < 8 and 0 <= new_col < 8:
            connected(new_row, new_col, pawn_grid, player, visited, directions)

def find_connected_pawns(pawn_grid, start_pos, player):
    """
    Trouve tous les pions connectés à partir d'une position de départ.
    """
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Haut, Bas, Gauche, Droite
    visited = [] # Liste pour stocker les positions visitées

    # Démarrer l'exploration récursive
    connected(start_pos[0], start_pos[1], pawn_grid, player, visited, directions)

    return visited

```

### 6.2.3 AFFICHAGE VISUEL DE LA VICTOIRE

SURBRILLANCE DES PIONS CONNECTÉS (congress.py) :

```

```python
def highlight_connected_pawns(screen, connected_pawns, board_x, board_y, cell_size, player_color):
    """
    Met en surbrillance les pions connectés sur l'écran pour montrer la victoire.
    """
    highlight_color = (255, 255, 0) # Jaune
    highlight_thickness = 3

    # Dessiner un cercle de surbrillance autour de chaque pion connecté
    for row, col in connected_pawns:
        x = board_x + col * cell_size + cell_size // 2
        y = board_y + row * cell_size + cell_size // 2

```

```

radius = cell_size // 3 + 5

pygame.draw.circle(screen, highlight_color, (x, y), radius, highlight_thickness)
...

```

INTÉGRATION DANS L'INTERFACE (game\_board.py) :

```

```python
# Mettre en surbrillance les pions connectés si le jeu est terminé en mode Congress
if game_over and winner > 0 and current_game_mode == 1:
    player_color = DARK_RED if winner == 1 else DARK_BLUE
    highlight_connected_pawns(screen, connected_pawns, board_x, board_y, cell_size, player_color)
...

```

=====

## 6.3 ALGORITHMES DE VICTOIRE - ISOLATION

Le jeu Isolation repose sur un principe d'élimination progressive : le joueur qui ne peut plus jouer perd.

### 6.3.1 PRINCIPE DE BLOCAGE

RÈGLE DE VICTOIRE :

Un joueur gagne lorsque son adversaire ne peut plus placer de pion sur le plateau de manière sûre.

ALGORITHME PRINCIPAL (isolation.py) :

```

```python
def check_isolation_victory(pawn_grid, current_player, board_grid):
    """
    Vérifie les conditions de victoire pour le mode Isolation.

    Le joueur qui vient de jouer gagne si l'adversaire ne peut plus jouer.
    Utilise la grille 10x10 harmonisée.
    """
    next_player = 2 if current_player == 1 else 1
    safe_positions = get_all_safe_positions_isolation(pawn_grid, board_grid, next_player)

```

```

if not safe_positions:
    # L'adversaire ne peut plus jouer, le joueur actuel gagne
    return True, current_player

return False, 0
...

```

### 6.3.2 CALCUL DES POSITIONS SÛRES

DÉFINITION D'UNE POSITION SÛRE :

Une position est sûre si aucun pion existant ne peut l'attaquer selon les règles de mouvement des cases.

ALGORITHME DE VÉRIFICATION :

```

```python
def get_all_safe_positions_isolation(pawn_grid, board_grid, player):
    """Retourne toutes les positions sûres pour un joueur en mode Isolation."""
    safe_positions = []
    for r in range(1, 9): # Zone de jeu 8x8
        for c in range(1, 9):
            if is_position_safe_isolation(pawn_grid, r, c, board_grid):
                safe_positions.append((r, c))
    return safe_positions

def is_position_safe_isolation(pawn_grid, row, col, board_grid):
    """
    Vérifie si une position est sûre pour le placement d'un pion en mode Isolation.
    Une position est sûre si aucun pion existant ne peut l'atteindre.
    """
    if pawn_grid[row][col] != 0:
        return False # Case déjà occupée

    # Vérifier tous les pions existants sur le plateau dans la zone de jeu 8x8
    for r in range(1, 9):

```

```

for c in range(1, 9):
    if pawn_grid[r][c] != 0: # Il y a un pion à cette position
        # Obtenir la couleur de la case où se trouve le pion
        cell_color = board_grid[r][c]

        # Vérifier si ce pion peut attaquer la position testée
        if can_attack_position(r, c, row, col, cell_color, pawn_grid, board_grid):
            return False

return True
...

```

### 6.3.3 RÈGLES D'ATTAQUE PAR TYPE DE CASE

ATTAQUE SELON LA COULEUR DE CASE :

```

```python
def can_attack_position(pion_row, pion_col, target_row, target_col, cell_color, pawn_grid, board_grid):
    """Détermine si un pion peut attaquer une position cible"""

    if cell_color == 3: # Bleu = Roi (8 directions, 1 case)
        dr, dc = abs(target_row - pion_row), abs(target_col - pion_col)
        return dr <= 1 and dc <= 1 and (dr > 0 or dc > 0)

    elif cell_color == 4: # Rouge = Tour (lignes droites AVEC arrêt sur cases rouges)
        if target_row == pion_row: # Même ligne
            start_col = min(target_col, pion_col) + 1
            end_col = max(target_col, pion_col)
            # Vérifier le chemin
            for c in range(start_col, end_col):
                if pawn_grid[target_row][c] != 0:
                    return False # Chemin bloqué par un pion
                if board_grid[target_row][c] == 4: # Arrêt sur case rouge
                    return False
            return True

```

```

elif target_col == pion_col: # Même colonne
    start_row = min(target_row, pion_row) + 1
    end_row = max(target_row, pion_row)
    # Vérifier le chemin
    for r in range(start_row, end_row):
        if pawn_grid[r][target_col] != 0:
            return False # Chemin bloqué par un pion
        if board_grid[r][target_col] == 4: # Arrêt sur case rouge
            return False
    return True

elif cell_color == 1: # Jaune = Fou (diagonales AVEC arrêt sur cases jaunes)
    dr, dc = target_row - pion_row, target_col - pion_col
    if abs(dr) == abs(dc) and dr != 0: # Même diagonale
        step_r = 1 if dr > 0 else -1
        step_c = 1 if dc > 0 else -1

        temp_r, temp_c = pion_row + step_r, pion_col + step_c
        while temp_r != target_row and temp_c != target_col:
            if pawn_grid[temp_r][temp_c] != 0:
                return False # Chemin bloqué par un pion
            if board_grid[temp_r][temp_c] == 1: # Arrêt sur case jaune
                return False
            temp_r += step_r
            temp_c += step_c
        return True

elif cell_color == 2: # Vert = Cavalier (mouvement en L)
    dr, dc = abs(target_row - pion_row), abs(target_col - pion_col)
    return (dr == 2 and dc == 1) or (dr == 1 and dc == 2)

return False
...

```



## 6.3.4 PLACEMENT ET VÉRIFICATION

MÉCANISME DE PLACEMENT :

```
```python
def place_pawn_isolation(pawn_grid, row, col, player, board_grid):
    """
    Place un pion en mode Isolation si la position est valide et sûre.
    """

    # Vérifier que nous sommes dans la zone de jeu 8x8
    if not (1 <= row <= 8 and 1 <= col <= 8):
        return False

    if pawn_grid[row][col] != 0:
        return False # Case déjà occupée

    # Vérifier si la position est sûre (pas en prise)
    if not is_position_safe_isolation(pawn_grid, row, col, board_grid):
        return False

    # Placer le pion
    pawn_grid[row][col] = player
    return True
```
```

INTÉGRATION DANS LE JEU (game\_board.py) :

```
```python
# Mode Isolation - Gestion du clic pour placement
if current_game_mode == 2:
    if (1 <= grid_row <= 8 and 1 <= grid_col <= 8 and
        pawn_grid[grid_row][grid_col] == 0):

        # Test de sécurité de la position
        can_place = True

        # [Logique de vérification détaillée...]
```
```

```

if can_place:
    pawn_grid[grid_row][grid_col] = current_player
    audio_manager.play_sound('pawn_move')

    # Vérifier victoire
    game_over, winner = check_isolation_victory(pawn_grid, current_player, board_grid)

    if not game_over:
        current_player = 2 if current_player == 1 else 1
...

```

### 6.3.5 IA POUR L'ISOLATION

ALGORITHME SIMPLE POUR L'ORDINATEUR :

```

```python
def isolation_ai(pawn_grid, board_grid, current_player):
    """
    IA simple pour le mode Isolation qui choisit une position aléatoire valide
    """
    # Utiliser la même logique que pour le joueur humain
    valid_positions = []
    for row in range(1, 9):
        for col in range(1, 9):
            if (pawn_grid[row][col] == 0 and
                is_position_safe_isolation(pawn_grid, row, col, board_grid)):
                valid_positions.append((row, col))

    # Si aucune position valide, retourner None
    if not valid_positions:
        return None

    # Choisir une position aléatoire
    return random.choice(valid_positions)

```

...

# EXPLICATION DU PROCÉDÉ DE COMMUNICATION RÉSEAU

## 7.1 ARCHITECTURE DE COMMUNICATION GÉNÉRALE

Notre système de jeu en réseau utilise une architecture client-serveur basée sur le protocole TCP, permettant à deux joueurs de jouer ensemble depuis des ordinateurs différents connectés au même réseau local ou via Internet.

PRINCIPE FONDAMENTAL :

- Un joueur "héberge" la partie (serveur)
- L'autre joueur "rejoint" la partie (client)
- Communication bidirectionnelle en temps réel
- Synchronisation des états de jeu entre les deux machines

TECHNOLOGIES UTILISÉES :

- Protocol TCP : Communication fiable avec garantie de livraison
- Sockets Python : Interface de programmation réseau native
- Format JSON : Sérialisation des données de jeu
- Threading : Gestion asynchrone des messages

## 7.2 ÉTABLISSEMENT DE LA CONNEXION

Le processus de connexion suit une séquence précise pour établir la communication entre les deux ordinateurs.

### 7.2.1 CÔTÉ SERVEUR (HÉBERGEMENT)

INITIALISATION DU SERVEUR (network\_manager.py) :

```
```python  
def start_server(self, port=12345):
```

```
"""Démarré le serveur pour attendre une connexion"""
```

```
try:
```

```
    # Création du socket serveur
```

```
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
    # Liaison à toutes les interfaces réseau disponibles
```

```
    self.socket.bind(("", port)) # " = toutes les interfaces
```

```
    self.socket.listen(1) # Attendre maximum 1 connexion
```

```
    self.is_server = True
```

```
    print(f"Serveur démarré sur le port {port}")
```

```
    print("En attente de connexion...")
```

```
    # ATTENTE BLOQUANTE d'une connexion client
```

```
    self.connection, addr = self.socket.accept()
```

```
    self.is_connected = True
```

```
    print(f"Connexion établie avec {addr}")
```

```
    # Démarrer le thread de réception des messages
```

```
    receive_thread = threading.Thread(target=self._receive_messages)
```

```
    receive_thread.daemon = True # Thread daemon (se ferme avec le programme)
```

```
    receive_thread.start()
```

```
    return True
```

```
except Exception as e:
```

```
    print(f"Erreur lors du démarrage du serveur: {e}")
```

```
    return False
```

```
...
```

OBTENTION DE L'ADRESSE IP LOCALE :

```
```python
```

```
def get_local_ip(self):
```

```

"""Obtient l'adresse IP locale pour communication"""

try:
    # Technique : connexion factice vers un serveur externe
    temp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    temp_socket.connect(("8.8.8.8", 80)) # DNS Google
    local_ip = temp_socket.getsockname()[0]
    temp_socket.close()
    return local_ip
except:
    return "127.0.0.1" # Fallback vers localhost
...

```

## 7.2.2 CÔTÉ CLIENT (CONNEXION)

CONNEXION AU SERVEUR :

```

```python
def connect_to_server(self, host, port=12345):
    """Se connecte à un serveur existant"""
    try:
        # Création du socket client
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # CONNEXION vers l'adresse IP du serveur
        self.socket.connect((host, port))
        self.connection = self.socket
        self.is_connected = True
        self.is_server = False

        print(f"Connecté au serveur {host}:{port}")

        # Démarrer le thread de réception des messages
        receive_thread = threading.Thread(target=self._receive_messages)
        receive_thread.daemon = True
        receive_thread.start()

```

```

        return True

    except Exception as e:
        print(f"Erreur lors de la connexion: {e}")
        return False
...

```

### 7.2.3 INTERFACE UTILISATEUR DE CONNEXION (network\_menu.py)

MENU DE SÉLECTION :

```

```python
def show_network_menu(screen):
    """Interface graphique pour établir la connexion réseau"""

    # États possibles : "menu", "host", "join", "connecting", "connected"
    mode = "menu"

    network_manager = NetworkManager()
    input_text = "" # Pour saisie IP

    # Interface pour hébergement
    if mode == "host":
        # Affichage de l'IP locale à communiquer
        local_ip = network_manager.get_local_ip()
        # "Communiquez cette IP à votre adversaire : [IP]"

        # Démarrage automatique du serveur
        connection_thread = threading.Thread(target=start_server_thread)
        connection_thread.start()

    # Interface pour connexion
    elif mode == "join":
        # Zone de saisie pour l'IP du serveur
        # Validation de l'IP saisie

```

```

# Tentative de connexion
connection_thread = threading.Thread(target=connect_to_server_thread, args=(input_text,))
connection_thread.start()
...

```

## 7.3 ÉCHANGE DE MESSAGES EN TEMPS RÉEL

Une fois la connexion établie, les deux ordinateurs échangent des messages JSON pour synchroniser le jeu.

### 7.3.1 MÉCANISME D'ENVOI

FORMAT STANDARD DES MESSAGES :

```

```python
def send_message(self, message_type, data):
    """Envoie un message structuré à l'autre joueur"""
    if not self.is_connected:
        return False

    try:
        # Construction du message avec horodatage
        message = {
            'type': message_type,    # Type de message
            'data': data,            # Données spécifiques
            'timestamp': time.time() # Horodatage pour debug
        }

        # Sérialisation JSON et envoi
        json_message = json.dumps(message)
        self.connection.send(json_message.encode('utf-8'))
        return True

    except Exception as e:
        print(f"Erreur lors de l'envoi: {e}")

```

```

        self.is_connected = False # Marquer comme déconnecté
    return False
...

```

EXEMPLES D'ENVOI SELON LE CONTEXTE :

```

```python
# Mouvement de pion (Katarenga/Congress)
def send_move(from_pos, to_pos):
    network_manager.send_message("move", {
        "from": from_pos,
        "to": to_pos,
        "player": my_player
    })

# Placement de pion (Isolation)
def send_placement(position):
    network_manager.send_message("placement", {
        "position": position,
        "player": my_player
    })

# Synchronisation du mode de jeu
network_manager.send_message("game_mode", {"mode": selected_game_mode})

# Annonce de victoire
network_manager.send_message("victory", {
    "winner": winner,
    "game_mode": current_game_mode
})
...

```

## 7.3.2 MÉCANISME DE RÉCEPTION

THREAD DE RÉCEPTION ASYNCHRONE :



```

python
def _receive_messages(self):
    """Thread dédié pour recevoir les messages en continu"""
    while self.is_connected:
        try:
            # RÉCEPTION BLOQUANTE (attente d'un message)
            data = self.connection.recv(1024) # Buffer de 1024 octets

            if not data:
                break # Connexion fermée côté distant

            # Décodage et désérialisation JSON
            message = data.decode('utf-8')
            parsed_message = json.loads(message)

            # STOCKAGE THREAD-SAFE dans le buffer
            with self.message_lock:
                self.received_messages.append(parsed_message)

        except Exception as e:
            print(f"Erreur lors de la réception: {e}")
            break

    # Marquer comme déconnecté en cas d'erreur
    self.is_connected = False
    ...

```

#### RÉCUPÉRATION DES MESSAGES :

```

python
def get_messages(self):
    """Récupère tous les messages reçus (thread-safe)"""
    with self.message_lock:
        # Copier et vider le buffer
        messages = self.received_messages.copy()
        self.received_messages.clear()

```

```

    return messages
...

```

### 7.3.3 TRAITEMENT DES MESSAGES REÇUS

BOUCLE DE TRAITEMENT PRINCIPAL (network\_game.py) :

```

```python
def process_messages():
    """Traite tous les messages reçus avec gestion complète des victoires"""
    nonlocal pawn_grid, current_player, game_over, winner, connected_pawns
    nonlocal selected_pawn, possible_moves, victory_shown, victory_message_sent, game_ended

    messages = network_manager.get_messages()
    for msg in messages:

        if msg['type'] == 'move':
            from_pos = msg['data']['from']
            to_pos = msg['data']['to']

            print(f"👉 Mouvement reçu: {from_pos} -> {to_pos}")

        # KATARENGA - Gestion spéciale des camps
        if current_game_mode == 0:
            from jeux.katarenga import get_camp_positions, place_in_camp, check_katarenga_victory
            camp_positions = get_camp_positions(opponent_player)

            if to_pos in camp_positions:
                if place_in_camp(to_pos[0], to_pos[1], pawn_grid, opponent_player):
                    pawn_grid[from_pos[0]][from_pos[1]] = 0
                    # Vérifier victoire après placement dans camp
                    winner = check_katarenga_victory()
                    if winner > 0:
                        game_over = True
                        game_ended = True

```

```

        victory_shown = False

    else:

        current_player = my_player

    return

# CONGRESS - Exécution immédiate sans animation
if current_game_mode == 1:

    pawn_grid[to_pos[0]][to_pos[1]] = pawn_grid[from_pos[0]][from_pos[1]]
    pawn_grid[from_pos[0]][from_pos[1]] = 0

    from jeux.congress import check_victory

    temp_grid = [[0 for _ in range(8)] for _ in range(8)]
    for row in range(1, 9):
        for col in range(1, 9):
            temp_grid[row-1][col-1] = pawn_grid[row][col]

    winner, connected_pawns_temp = check_victory(temp_grid)
    if winner > 0:
        connected_pawns = [(row+1, col+1) for row, col in connected_pawns_temp]
        game_over = True
        game_ended = True
        victory_shown = False
    else:
        current_player = my_player

else:

    # Animation pour Katarenga - SANS vérification immédiate de victoire
    animation.start_move(from_pos[0], from_pos[1], to_pos[0], to_pos[1],
                        board_x, board_y, cell_size,
                        pawn_grid[from_pos[0]][from_pos[1]])
    animation.pending_move = {
        'from': from_pos, 'to': to_pos,
        'pawn_color': pawn_grid[from_pos[0]][from_pos[1]]
    }
}

```

```

elif msg['type'] == 'placement': # Isolation
    row, col = msg['data']['position']
    player = msg['data']['player']
    pawn_grid[row][col] = player

    from jeux.isolation import check_isolation_victory
    game_over_temp, winner_temp = check_isolation_victory(pawn_grid, player, board_grid)

    if game_over_temp:
        game_over = True
        game_ended = True
        winner = winner_temp
        victory_shown = False
    else:
        current_player = my_player

elif msg['type'] == 'victory':
    # CORRECTION CRITIQUE : Réception message de victoire
    winner = msg['data']['winner']
    game_over = True
    game_ended = True
    victory_shown = False # FORCER l'affichage
    print(f"📬 Message de victoire reçu: Joueur {winner}")

    # Confirmer réception pour synchronisation
    network_manager.send_message("victory_received", {"winner": winner})

elif msg['type'] == 'victory_received':
    # Confirmation que l'adversaire a reçu la victoire
    print(f"✅ Adversaire a reçu le message de victoire")

elif msg['type'] == 'disconnect':
    print(f"🚫 Adversaire déconnecté")

```

```

    if not game_ended: # Seulement si partie en cours
        game_over = True
        game_ended = True
        victory_shown = False
    ...

```

## 7.4 SYNCHRONISATION DES CONFIGURATIONS

Avant de commencer la partie, les deux ordinateurs doivent synchroniser leurs configurations de quadrants.

### 7.4.1 ÉCHANGE DES QUADRANTS

PRINCIPE :

- Chaque joueur configure ses propres quadrants
- Serveur (Rouge) : quadrants du haut
- Client (Bleu) : quadrants du bas
- Échange des configurations pour assembler le plateau final

ENVOI DE LA CONFIGURATION (network\_quadrant\_setup.py) :

```

```python
def send_my_quadrants():
    """Envoie ma configuration de quadrants à l'adversaire"""
    if None in selected_quadrants:
        return False

    # Construire les grilles avec rotations appliquées
    my_config = []
    for i, quad_id in enumerate(selected_quadrants):
        grid = get_quadrant_grid(quad_id, quadrant_rotations[i])
        my_config.append(grid)

    # Message de configuration complet
    config_data = {
        "player": "server" if is_server else "client",

```

```

    "quadrants": my_config,          # Grilles 4x4 finales
    "quadrant_indices": quadrant_indices, # Positions sur le plateau
    "selected_ids": selected_quadrants,   # IDs des quadrants choisis
    "rotations": quadrant_rotations      # Rotations appliquées
}

network_manager.send_message("quadrant_config", config_data)

return True

...

```

## 7.4.2 ASSEMBLAGE DU PLATEAU FINAL

CONSTRUCTION COORDONNÉE :

```

```python
def build_final_board_config(my_quadrants, my_rotations, opponent_config, is_server):
    """
    Construit la configuration finale du plateau selon la perspective de chaque joueur
    """

    final_quadrants = [None, None, None, None]

    # Mes grilles (avec rotations appliquées)
    my_grids = []
    for i, quad_id in enumerate(my_quadrants):
        if quad_id:
            grid = get_quadrant_grid(quad_id, my_rotations[i])
            my_grids.append(grid)

    # Grilles de l'adversaire (reçues par réseau)
    opponent_grids = opponent_config["quadrants"]

    if is_server:
        # SERVEUR (Rouge) : Mes quadrants EN HAUT, adversaire EN BAS
        print(" Assemblage serveur: Haut=Mes quadrants, Bas=Adversaire")
        if len(my_grids) >= 2:

```

```

        final_quadrants[0] = my_grids[0]    # Haut gauche
        final_quadrants[1] = my_grids[1]    # Haut droite
    if len(opponent_grids) >= 2:
        final_quadrants[2] = opponent_grids[0] # Bas gauche
        final_quadrants[3] = opponent_grids[1] # Bas droite
    else:
        # CLIENT (Bleu) : Adversaire EN HAUT, mes quadrants EN BAS
        print(" Assemblage client: Haut=Adversaire, Bas=Mes quadrants")
    if len(opponent_grids) >= 2:
        final_quadrants[0] = opponent_grids[0] # Haut gauche
        final_quadrants[1] = opponent_grids[1] # Haut droite
    if len(my_grids) >= 2:
        final_quadrants[2] = my_grids[0]    # Bas gauche
        final_quadrants[3] = my_grids[1]    # Bas droite

    return final_quadrants
'''

```

## 7.5 GESTION DES TOURS ET SYNCHRONISATION

Le système assure que les deux joueurs restent synchronisés et que chacun joue à son tour.

### 7.5.1 ATTRIBUTION DES RÔLES

IDENTIFICATION DES JOUEURS :

```

'''python
# Dans network_game.py
if network_manager.is_server:
    my_player = 1      # Serveur = Joueur Rouge
    opponent_player = 2 # Client = Joueur Bleu
else:
    my_player = 2      # Client = Joueur Bleu
    opponent_player = 1 # Serveur = Joueur Rouge

```

...

## 7.5.2 GESTION AVANCÉE DES ÉTATS DE VICTOIRE

VARIABLES D'ÉTAT ENRICHIES :

```
```python
# Variables de contrôle de fin de partie

victory_shown = False      # Interface de victoire affichée
victory_message_sent = False # Éviter envois multiples de victoire
game_ended = False        # État final de partie

# Initialisation Katarenga spécifique
if current_game_mode == 0:
    from jeux.katarenga import reset_camps
    reset_camps()
    print("🎮 Camps Katarenga initialisés")
...
```
```

VÉRIFICATION CONTINUE DES VICTOIRES :

```
```python
# VÉRIFICATION KATARENGA - Version corrigée

if current_game_mode == 0 and not game_over and not game_ended:
    from jeux.katarenga import check_katarenga_victory, check_minimum_pawn_victory_condition

    # Vérifier victoire par camps (2 camps occupés)
    potential_winner = check_katarenga_victory()
    if potential_winner > 0 and not victory_message_sent:
        winner = potential_winner
        game_over = True
        game_ended = True
        victory_shown = False
        victory_message_sent = True
        print(f"🏆 Victoire Katarenga par camps: Joueur {winner}")
        network_manager.send_message("victory", {"winner": winner})
...
```
```



```

# Attendre confirmation avec timeout
confirmation_timeout = time.time() + 2.0
while time.time() < confirmation_timeout:
    temp_messages = network_manager.get_messages()
    for temp_msg in temp_messages:
        if temp_msg['type'] == 'victory_received':
            print("✔ Confirmation de victoire reçue")
            break
    else:
        pygame.time.wait(50)
        continue
    break

# Vérifier victoire par élimination si pas de victoire par camps
elif not victory_message_sent:
    elimination_winner = check_minimum_pawn_victory_condition(pawn_grid, current_game_mode)
    if elimination_winner > 0:
        winner = elimination_winner
        game_over = True
        game_ended = True
        victory_shown = False
        victory_message_sent = True
        print(f"🏆 Victoire Katarenga par élimination: Joueur {winner}")
        network_manager.send_message("victory", {"winner": winner})
...

```

## 7.6 GESTION DES ERREURS ET DÉCONNEXIONS

Le système inclut des mécanismes pour gérer les erreurs réseau et les déconnexions inattendues.

### 7.6.1 DÉTECTION DE DÉCONNEXION

SURVEILLANCE DE LA CONNEXION :

```

```python
# Vérification régulière dans la boucle de jeu
if not network_manager.is_connected and not game_over:
    game_over = True
    winner = my_player # Je gagne par forfait

# Dans le thread de réception
def _receive_messages(self):
    while self.is_connected:
        try:
            data = self.connection.recv(1024)

            if not data: # Connexion fermée côté distant
                break

        except Exception as e:
            print(f"Erreur réseau: {e}")
            break

    self.is_connected = False # Marquer comme déconnecté
...

```

## 7.6.2 INTERFACE DE VICTOIRE AMÉLIORÉE

AFFICHAGE PERSISTANT ET ADAPTATIF :

```

```python
# VICTOIRE - AFFICHAGE DÉFINITIF ET PERSISTANT
if game_ended and not victory_shown:
    victory_shown = True # Marquer comme affiché une seule fois

if victory_shown and game_ended:
    # Affichage spécial pour Congress avec surbrillance
    if current_game_mode == 1 and winner > 0:
        player_color = DARK_RED if winner == 1 else DARK_BLUE

        from jeux.congress import highlight_connected_pawns
        connected_pawns_display = [(row-1, col-1) for row, col in connected_pawns]

```

```
highlight_connected_pawns(screen, connected_pawns_display, board_x, board_y, cell_size,
player_color)
```

```
# Fond avec transparence COMPLET
```

```
overlay = pygame.Surface((current_width, current_height), pygame.SRCALPHA)
```

```
overlay.fill((0, 0, 0, 180)) # Plus opaque
```

```
screen.blit(overlay, (0, 0))
```

```
# Message selon la situation
```

```
if winner == 0:
```

```
    victory_text = "🏴‍☠️ PARTIE INTERROMPUE"
```

```
    color = (255, 165, 0) # Orange
```

```
    info_text = "Connexion perdue"
```

```
elif winner == my_player:
```

```
    victory_text = "🏆 VICTOIRE ! 🏆"
```

```
    color = GREEN
```

```
    mode_names = ["Katarenga", "Congress", "Isolation"]
```

```
    info_text = f"Mode {mode_names[current_game_mode]} - Bien joué !"
```

```
else:
```

```
    victory_text = "💀 DÉFAITE 💀"
```

```
    color = RED
```

```
    mode_names = ["Katarenga", "Congress", "Isolation"]
```

```
    info_text = f"Mode {mode_names[current_game_mode]} - Dommage..."
```

```
# Interface graphique enrichie avec émojis et couleurs
```

```
font_big = pygame.font.Font(None, 72) # Police plus grande
```

```
font_medium = pygame.font.Font(None, 36)
```

```
# Boîte de message plus visible
```

```
box_width = max(text_rect.width + 100, 500)
```

```
box_height = 200
```

```
box_rect = pygame.Rect(...)
```

```
# Bordure épaisse colorée selon le résultat
```

```

pygame.draw.rect(screen, WHITE, box_rect)
pygame.draw.rect(screen, color, box_rect, 6)

# Boutons d'action améliorés
btn_width = 140 # Plus larges
btn_height = 45 # Plus hauts

rejouer_btn = pygame.Rect(...) # Bouton "REJOUER"
quitter_btn = pygame.Rect(...) # Bouton "QUITTER"
...

GESTION DES ACTIONS POST-VICTOIRE :
```python
# Actions après victoire avec nettoyage complet
if victory_shown and game_ended:
    if rejouer_btn.collidepoint(event.pos):
        print("🔄 REDÉMARRAGE → Menu de connexion réseau")

# Réinitialisation complète avant déconnexion
game_over = False
game_ended = False
victory_shown = False
victory_message_sent = False
winner = 0

# Déconnexion propre avec délai
if network_manager.is_connected:
    network_manager.send_message("disconnect", {})
    pygame.time.wait(100) # Laisser le temps d'envoyer

network_manager.disconnect()

# Retour au menu de connexion réseau
from réseaux.network_menu import show_network_menu

```

```

        return show_network_menu(screen)

elif quitter_btn.collidepoint(event.pos):
    print("🏠 RETOUR AU HUB PRINCIPAL")

    # Déconnexion propre
    if network_manager.is_connected:
        network_manager.send_message("disconnect", {})
        pygame.time.wait(100)

    network_manager.disconnect()
    return # Retour vers hub.py
...

```

## 7.7 SPÉCIFICITÉS PAR MODE DE JEU

Chaque mode de jeu nécessite des adaptations spécifiques du protocole réseau.

### 7.7.1 KATARENGA

GESTION DES CAMPS :

```

```python
# Mouvement vers un camp (traitement spécial)
if to_pos in camp_positions:
    if place_in_camp(to_pos[0], to_pos[1], pawn_grid, current_player):
        # Retirer le pion du plateau
        pawn_grid[from_pos[0]][from_pos[1]] = 0

    # Vérifier victoire par camps
    winner = check_katarenga_victory()
    if winner > 0:
        game_over = True
        # Annoncer la victoire à l'adversaire

```

```

        network_manager.send_message("victory", {
            "winner": winner,
            "game_mode": 0
        })
    ...

```

## 7.7.2 CONGRESS

TRANSMISSION DES PIONS CONNECTÉS :

```

```python
# Lors de la victoire, transmettre les pions connectés pour l'affichage
if winner > 0:
    connected_pawns = [(row+1, col+1) for row, col in connected_pawns_temp]
    game_over = True

    network_manager.send_message("victory", {
        "winner": winner,
        "game_mode": 1,
        "connected_pawns": connected_pawns # Pour l'affichage côté adversaire
    })
...

```

## 7.7.3 ISOLATION

VÉRIFICATION DISTRIBUÉE :

```

```python
# Chaque joueur vérifie si l'adversaire peut encore jouer
if current_game_mode == 2:
    # Vérifier si l'adversaire peut jouer après mon mouvement
    can_opponent_play = len(get_all_safe_positions_isolation(pawn_grid, board_grid, opponent_player)) > 0

    if not can_opponent_play:
        game_over = True
        winner = my_player

```

```
network_manager.send_message("victory", {  
    "winner": winner,  
    "game_mode": 2  
})  
...
```