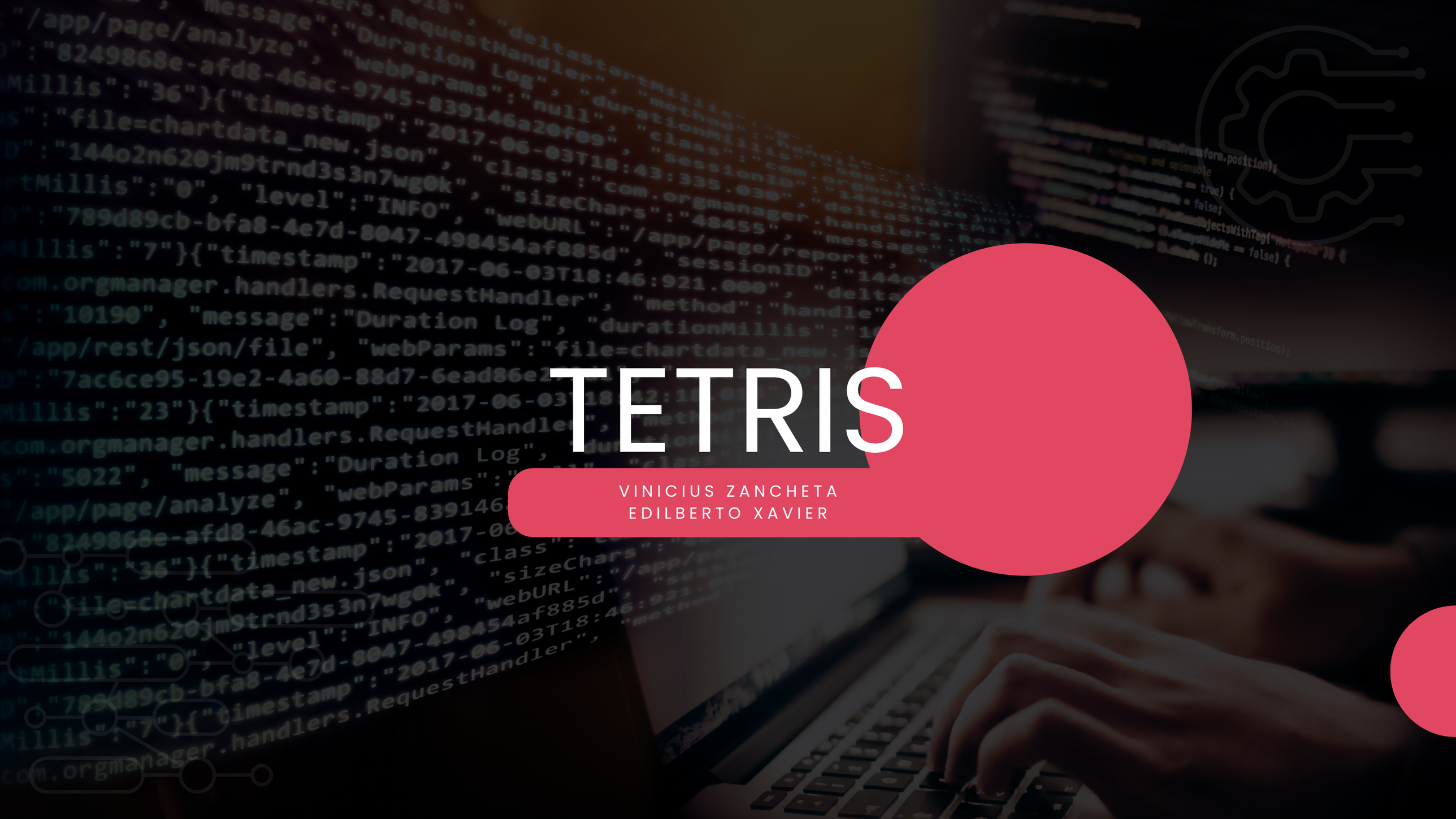
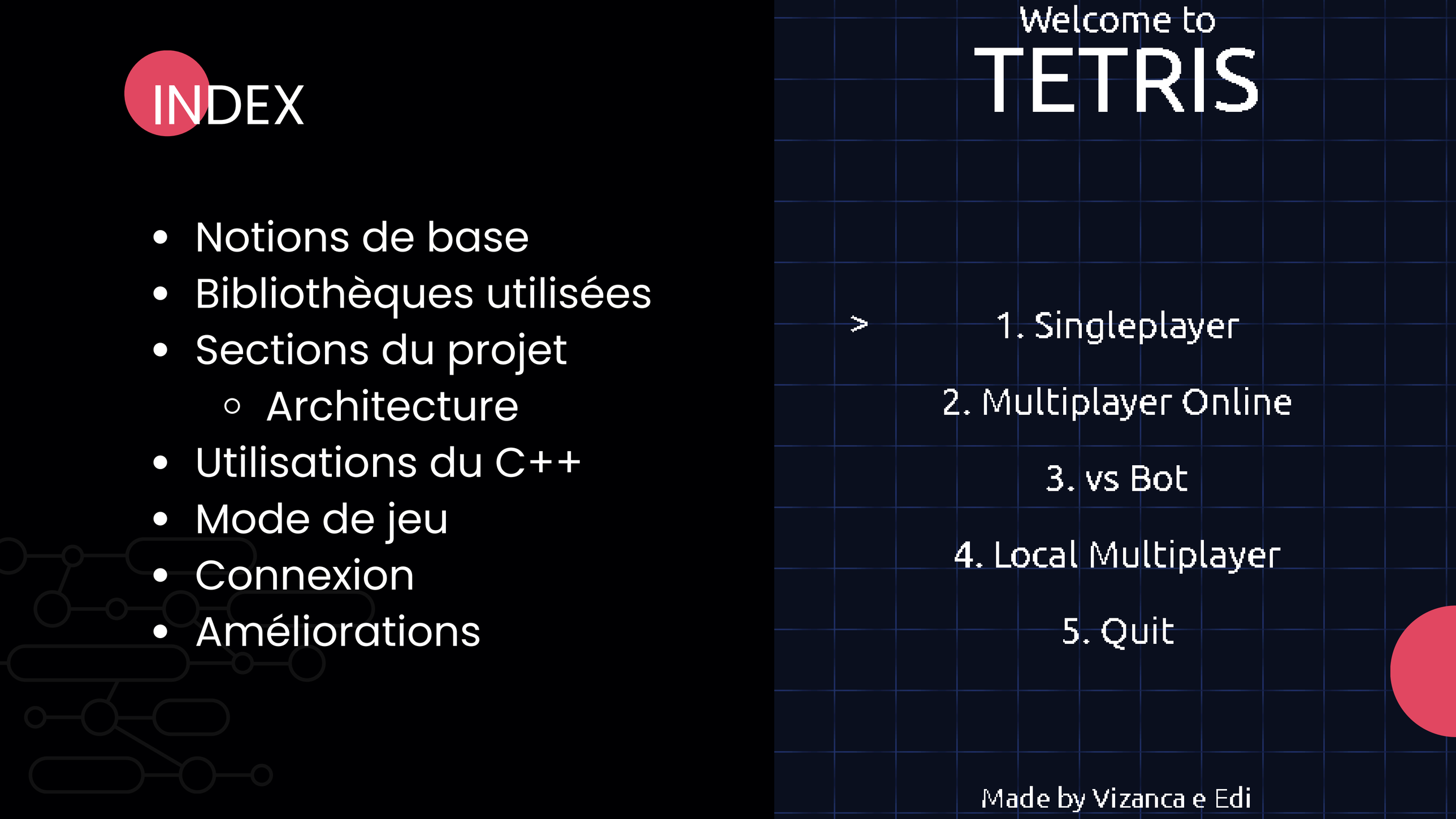


TETRIS

VINICIUS ZANCHETA
EDILBERTO XAVIER

INDEX



BESOINS FONDAMENTAUX

- **Environnement de Compilation :** Utilisation du compilateur g++ avec le flag -pthread pour la gestion des processus légers et -g pour le débogage.
- **Dépendances Logicielles :** Présence impérative des bibliothèques SDL2 pour le rendu graphique, SDL2_ttf pour la gestion des polices de caractères
- **Architecture Découplée** : Le système nécessite deux processus indépendants (Serveur et Client) communiquant via des sockets TCP.
- **Organisation du Code** : Respect d'une structure modulaire stricte avec des répertoires séparés pour les sources (./src) et les en-têtes (./include).

SAVOIR OÙ IL FAUDRAIT FAIRE DES CONCESSIONS

- **Échelle de Joueurs :** Limitation volontaire à 2 joueurs par salle
- **Synchronisation par État :** Le client envoie l'intégralité de la matrice du jeu (200 caractères) plutôt que les touches pressées, garantissant une cohérence visuelle parfaite entre les adversaires malgré la latence réseau.
- **Simplification de l'IA :** Choix d'un algorithme basé sur des heuristiques (trous, hauteur, irrégularité) au lieu d'une recherche exhaustive

COMMENT DÉMARRER

Initialisation du Serveur
Compiler et lancer le binaire du serveur en définissant la taille de la salle : `g++ ./*.cpp -o main -pthread && ./main 2.`

Lancement du Client
Compiler les sources graphiques et se connecter à l'hôte local : `g++ -g ./src/*.cpp -lSDL2 -lSDL2_ttf -o main && ./main localhost`

Phase d'Attente
Le programme démarre sur la classe Menu, puis bascule vers la MenuRoom jusqu'à ce que le serveur envoie le code de démarrage du jeu.

BIBLIOTHÈQUES

1. Graphismes et Interface (SDL2)

- **SDL2** : Bibliothèque principale pour la création de fenêtres, la gestion des événements clavier et le rendu graphique 2D.
- **SDL2_ttf** : Extension utilisée pour le chargement des polices TrueType et l'affichage dynamique des scores et niveaux.

2. Bibliothèque Standard C++ (STL)

- **Conteneurs** (vector, array, map) : Gestion dynamique de la file d'attente des pièces, de la matrice du jeu et des rotations.
- **Parallélisme** (thread, mutex) : Implémentation du mode multijoueur avec des threads séparés pour le réseau et l'affichage, sécurisés par des verrous mutuels.

3. Système et Communications (POSIX)

- **Sockets POSIX** : Communication réseau de bas niveau via le protocole TCP/IP pour le mode duel.
- **En-têtes Système** (unistd.h) : Accès aux fonctions système pour la lecture/écriture sur les sockets et la gestion du temps.
- **Bibliothèques Mathématiques** (cmath, cstdlib) : Calculs des heuristiques pour l'IA et génération aléatoire des pièces (Virtual Bag).

SECTIONS

- **Utilités :**
  - Composants (Mino & Pos)
  - Texte & Background
  - Grille de Jeu (TetrisMap)
- **États (States) :**
  - Menus (Principal & Salle d'attente)
  - Phases de Jeu (Falling, Pattern, Animation)
  - Écran de Résultats
- **Vues (Views) :**
  - Rendu Solo
  - Rendu Multi (Mirror Layout)
- **Cœur du Jeu (Game Logic) :**
  - Tétriminos (Polymorphisme & SRS)
  - Intelligence Artificielle (Heuristiques)
- **Réseau (Network) :**
  - Architecture Client / Serveur (TCP Sockets)
  - Gestion des Salons (Room & Player)

UTILISATION DU C++

01 Encapsulation et Gestion de Vie

Utilisation stricte de private pour les attributs et public pour les interfaces (Getters/Setters) dans les classes como Mino, Pos et Player

```
class Mino
{
private:
    Color m_color;
    char m_value;

    // Animation
    float animationProgress = 0.0;

public:
    Mino();
    Mino(char val);
    Color &color();
    const Color &color() const;
    char &value();
    const char &value() const;
    void draw(SDL_Renderer *renderer, int x, int y);
    void incrementAnimationProgress();
    bool animationEnded();
};
```

02 Heritage et Polymorphisme

Hiérarchie de Classes : La classe Tetrimino sert de base abstraite définissant le comportement générique d'une pièce.

Liaison Dynamique : Utilisation de méthodes virtuelles (virtual getMinos()) permettant au moteur de jeu (TetrisMap) de manipuler n'importe quelle pièce de manière générique en temps de réel.

```
class TetriminoI : public Tetrimino
{
public:
    TetriminoI();
    virtual array<Pos, 4> getMinos();
    virtual map<Orientation, map<int, Pos>> g
    virtual int getSize();
};

class TetriminoJ : public Tetrimino
{
public:
    TetriminoJ();
    virtual array<Pos, 4> getMinos();
    virtual map<Orientation, map<int, Pos>> g
    virtual int getSize();
};
```

03 Conteneurs Standards

Utilisation de std::vector pour la file d'attente des pièces et std::array para les formes fixes des Tétriminos, assurant performance et sécurité.

```
vector<int> rowsDestroyed;
vector<char> virtualBag = {'I', 'O', 'T', 'L', 'J', 'Z', 'S'};
vector<Tetrimino *> tetriminoQueue;
```

04 Simplification de la Logique

Surcharge des opérateurs + et - dans la classe Pos, permettant des calculs de coordonnées intuitifs : Pos p = m_pos + offset;

```
Pos &Pos::operator+=(Pos &anotherPos)
{
    m_row += anotherPos.m_row;
    m_col += anotherPos.m_col;
    return *this;
}

Pos &Pos::operator-=(Pos &anotherPos)
{
    m_row -= anotherPos.m_row;
    m_col -= anotherPos.m_col;
    return *this;
}
```

UTILISATION DU C++

05 Parallélisme (Multithreading)

Implémentation de std::thread pour séparer le flux réseau (bloquant) du rendu graphique (fluide à 60 FPS).

thread t_client(handleClient, client, tetrisMap);

06 Mutex

Utilisation de std::mutex pour protéger les données partagées (liste de joueurs) contre les accès concurrents.

```
class TetrisServer
    void enter_game_phase();

private:
    /*
        Store client socket file descriptors.
        They are used to send and receive messages to and from other users
    */
    Room *room;

    int players_alive = 0;

    Server *server;

    Status m_status;

    /*
        Mutex used to lock and unlock when doing critical work through threads
    */
    mutex mtx;
};

#endif
```

07 Ponteuirs Intelligents

Gestion automatique de la mémoire pour éviter les fuites (Memory Leaks)

```
// Ponteiros Inteligentes
std::unique_ptr<Background> bg;
std::unique_ptr<Text> textSmall;
std::unique_ptr<Text> textSmaller;
std::unique_ptr<Text> textBig;
```

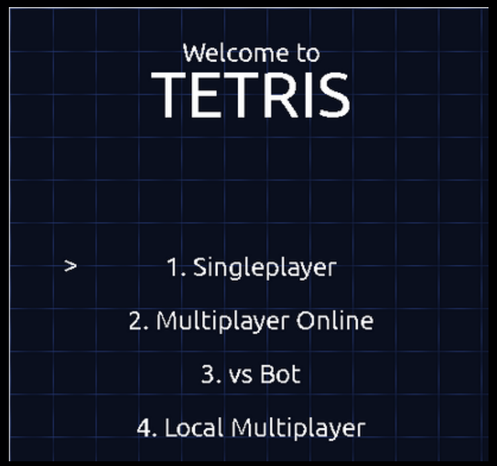
MODE DE JEU

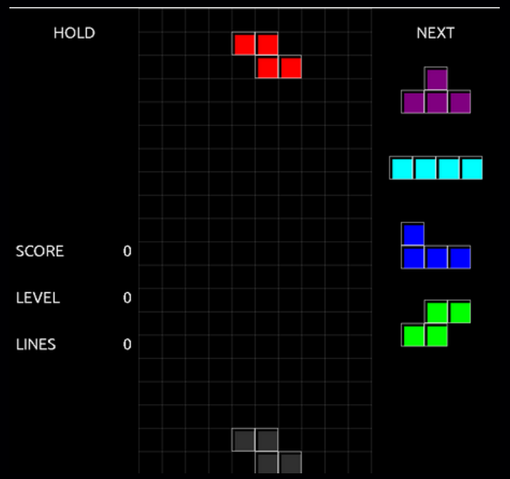
01 **Singleplayer**

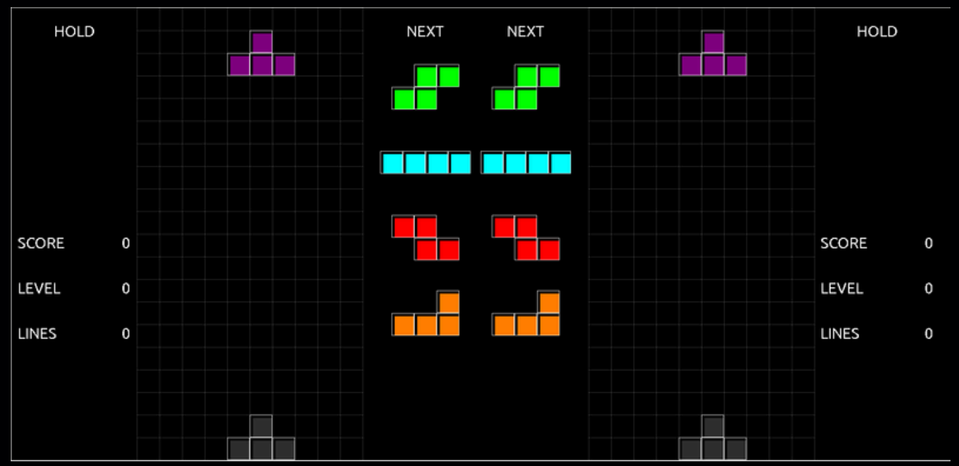
02 **Multiplayer “online”**

03 **vs BOT**

04 **Multiplayer locale**







CONNEXION

Modèle Client-Serveur (Sockets TCP)

- Protocole Fiable : Utilisation de sockets TCP/IP pour garantir l'intégrité des données, assurant qu'aucun bloc ou ligne de malus ne soit perdu lors de la transmission.
- Handshake Initial : Le client établit la connexion via la classe Client.
- Architecture Découplée : Le serveur fonctionne de manière autonome, gérant les salons (Room) et les joueurs (Player) sans impacter la logique de rendu graphique.

Protocole de Communication (Messaging)

- CODE_PLAYER_MAP : Envoi de la matrice complète du jeu (200 blocs) pour une synchronisation visuelle absolue.
- CODE_PLAYER_LINES : Envoi d'attaques lorsque l'adversaire complète plusieurs lignes simultanément.
- CODE_GAME_OVER : Signalement immédiat de la fin de partie pour les deux clients.

Threads Dédiés : Le client utilise un thread séparé (handleClient) pour la réception des données réseau afin de ne pas bloquer la boucle principale.

Gestion des Conflits (Mutex) : Côté serveur, des instances protègent la liste des joueurs actifs, prévenant les erreurs mémoire lors de déconnexions imprévues.

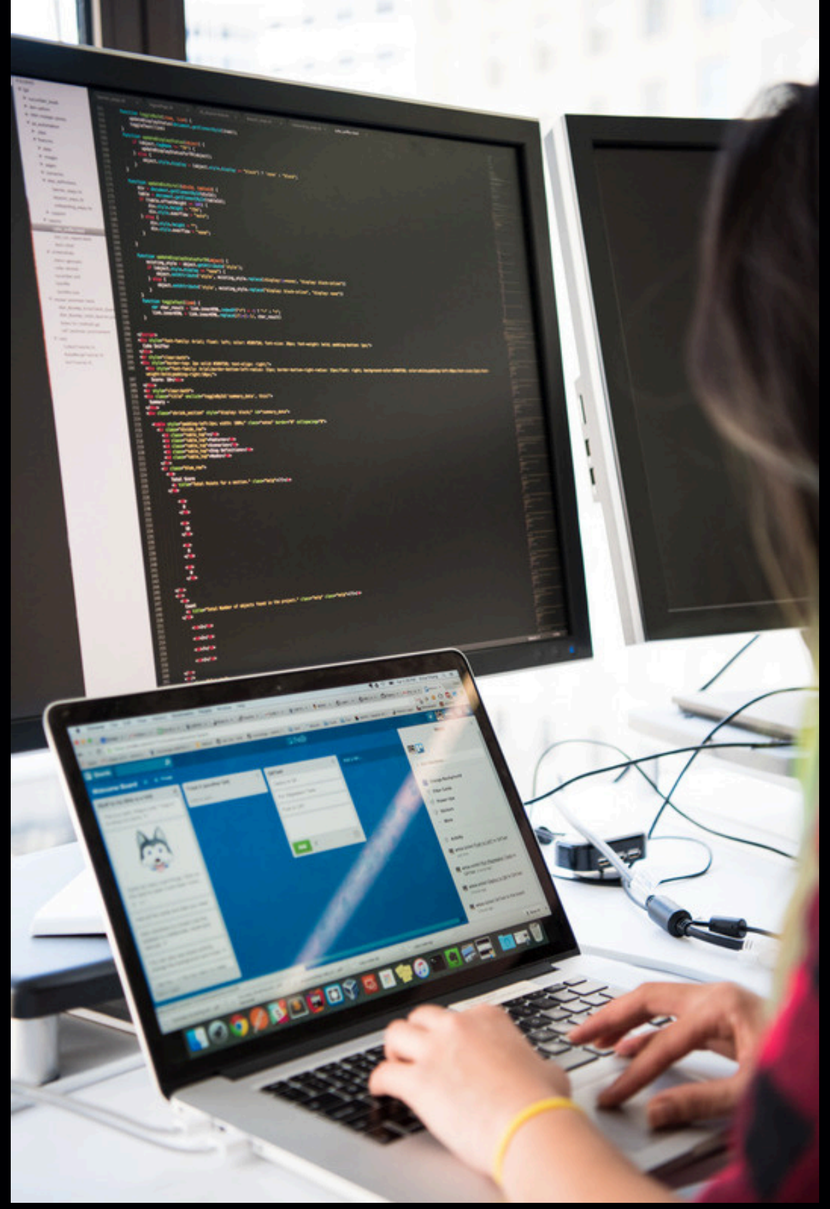
SALFORD & CO.

AMÉLIORATIONS

Évolution de l'Intelligence Artificielle

Évolutivité du Mode Multijoueur

Expérience Utilisateur



MERCI