Thouv'Run

Un Jeu de Plateforme Multijoueur

Paul Vernus

Janvier 2026

Résumé

Thouv'Run est un jeu de plateforme dynamique et compétitif développé en **Python 3.12**. Le joueur incarne Bastien qui se déplace en rollers et doit éviter des obstacles (voitures, camions, policiers) tout en collectant des bonus ("bedos"). Le jeu propose deux interfaces : une version graphique moderne (Pygame) et une version rétro en terminal. Les scores sont synchronisés en ligne via une API Flask et accessibles sur un leaderboard web en temps réel.

Table des matières

1 Présentation du Projet

1.1 Concept et Objectif

Thouv'Run est un jeu arcade rapide où le joueur doit :
— **Esquiver** trois types d'obstacles : voitures, camions et policiers
— **Courir** le plus loin possible pour maximiser son score
— **Collecter** des bonus appelés "bedos" pour gagner des points
— **Compétitionner** avec d'autres joueurs via un leaderboard en ligne

Le titre "Thouv'Run" est une référence ludique : le personnage principal, Bastien Thouverez, fait sa première rentrée avec des rollers le premier jour des cours et découvre qu'il est en retard !

1.2 Caractéristiques Principales

Fonctionnalités Clés

✓ Deux interfaces de jeu : Graphique (Pygame) et Terminal (Curses)
✓ Deux niveaux de difficulté : NORMALE et DIFFICILE (x1.5 points)
✓ Synchronisation automatique des scores avec serveur (API Flask)
✓ Leaderboard web avec recherche, tri et statistiques
✓ Gestion de la musique et des effets sonores
✓ Sauvegarde locale (JSON) + serveur de secours
✓ Support hors-ligne (scores synchro à la reconnexion)

2 Gameplay et Mécanique

2.1 Contrôles

Le joueur contrôle Bastien avec les touches suivantes :

Touche	Action	Obstacle
SPACE / FLECHE HAUT	Saut Court	Voiture, Policier
Z / W	Saut Long	Camion
ESC	Pause / Menu	—
R	Redémarrer	Après Game Over
F11	Plein écran	Mode Graphique

2.2 Mécanique de Jeu

2.2.1 Génération d'Obstacles

Les obstacles apparaissent de manière pseudoaléatoire selon la formule :

$$\text{spawn\_timer} > \frac{90}{\text{vitesse}} + \text{rand}(0, 40)$$

En mode DIFFICILE, les obstacles sont 2× plus fréquents :

$$\text{spawn\_timer} > \frac{45}{\text{vitesse}} + \text{rand}(0, 30)$$

Les obstacles se divisent en trois catégories avec les dimensions suivantes :

Type	Hauteur	Largeur	Couleur
Voiture	Basse (1 ligne)	2-3 caractères	ROUGE
Camion	Haute (3 lignes)	4-5 caractères	BLEU
Policier	Moyenne (2 lignes)	3-4 caractères	MAGENTA

2.2.2 Système de Score

— **Distance** : +0.1 points/frame × vitesse actuelle
— **Bonus "Bedo"** : +50 points par collecte
— **Vitesse Progressive** : augmente de 0.05 toutes les 500 points
— **Multiplicateur Difficulté** : ×1.5 en mode DIFFICILE

2.3 Système de Collision

La détection de collision utilise un rectangle AABB simplifié :

```
def rect_collision(self, autre):
    col_x = (self.x < autre.x + autre.width - 2) and \
            (self.x + self.width > autre.x + 2)
    col_y = (self.y < autre.y + autre.height) and \
            (self.y + self.height > autre.y)
    return col_x and col_y
```

Un léger offset de `-2` est appliqué pour rendre le jeu plus "jouable" (hitbox légèrement plus petite que l'affichage).

3 Architecture Technique

3.1 Structure du Projet

```
Projet Thouv\
|- src/
|   |- main_graphique.py      # Interface Pygame (1200+ lignes)
|   |- main_terminal.py       # Interface Curses (400+ lignes)
|   |- moteur_jeu.py          # Logique de jeu (Classes Entité, Jeu)
|   |- gestion_scores.py      # Sync scores + sauvegarde
|   |- tache_fond.py          # Thread de musique
|- server/
|   |- api_server.py          # API Flask + Leaderboard
|- web/
|   |- scores.html            # Leaderboard web (Pixel Art)
|- data/
|   |- thouv_scores.json      # Scores locaux
|   |- server_scores.json     # Backup persistant
|- assets/
|   |- images/                # Sprites, icônes
|   |- sounds/                # Effets sonores
|   |- music/                 # Musiques de fond
|   |- fonts/                 # Polices (Jersey10, Press Start 2P)
|- SETUP.bat                  # Installation auto (Python + pip)
|- Thouv-Run-Graphique.bat    # Lanceur Pygame
|- Thouv-Run-Terminal.bat     # Lanceur Terminal
```

3.2 Stack Technologique

Composant	Technologie
Langage Principal	Python 3.12
Interface Graphique	Pygame 2.5.2
Interface Terminal	windows-curses 2.4.1
Backend Serveur	Flask 3.0.0 + Flask-CORS
Base de Données	JSON (SQLite éphémère en production)
Hébergement	Railway.app (container Docker)
Frontend Web	HTML5 + CSS3 + JavaScript Vanilla
Domaine Custom	thouvrun.com (Ionos, Railway)

3.3 Architecture des Classes

3.3.1 Classe `Entite`

Classe parent pour tous les objets du jeu (joueur, obstacles, bonus) :

```
class Entite:
    def __init__(self, x, y, width, height, type_entite, art):
        self.x = x           # Position X (relative au viewport)
        self.y = y           # Position Y (relative au sol)
        self.width = width   # Largeur en caractères
        self.height = height # Hauteur en lignes
        self.type = type_entite  # "voiture", "camion", etc.
        self.art = art       # ASCII art pour affichage

    def rect_collision(self, autre):
        # Détection collision AABB
        pass
```

3.3.2 Classe `Thouverez` (Joueur)

Hérite de `Entite`, gère les sauts et l'état du joueur :

```
class Thouverez(Entite):
    def __init__(self):
        super().__init__(10, SOL_ECRAN, 2, 3, "joueur", ART_JOUEUR)
        self.is_jumping = False
        self.jump_vel = 0

    def sauter_court(self):
        if not self.is_jumping:
            self.is_jumping = True
            self.jump_vel = -2  # Hauteur: ~3 lignes

    def sauter_long(self):
        if not self.is_jumping:
            self.is_jumping = True
            self.jump_vel = -3  # Hauteur: ~5 lignes

    def update(self):
        # Appliquer gravité et détecter sol
        pass
```

3.3.3 Classe `Jeu`

Moteur central gérant la physique, les collisions et le game loop :

```
class Jeu:
    def __init__(self, nom_joueur, version_jeu, difficulte="NORMALE"):
        self.joueur = Thouverez()
        self.obstacles = []     # Liste des obstacles actifs
        self.bonus = []         # Liste des bonus actifs
        self.score = 0
        self.distance = 0
        self.bedos = 0
        self.vitesse = 1.0
        self.running = True
        self.difficulte = difficulte

    def update(self):
        # Appliquer physique
        # Générer obstacles
        # Vérifier collisions
        # Calculer score
        pass

    def fin_partie(self, cause):
        # Enregistrer score dans JSON + API
        pass
```

3.4 Gestion des Scores et Synchronisation

La synchronisation fonctionne en trois couches :

1. **Couche Locale** : `data/thouv_scores.json`
   - Sauvegarde immédiate après chaque partie
   - Toujours accessible, même hors-ligne
   - Backup utilisé au démarrage du jeu
2. **Couche Serveur** : API Flask à `https://thouvrun-production.up.railway.app`
   - POST `/api/scores` : envoyer un nouveau score
   - GET `/api/scores?limit=500` : récupérer les N meilleurs scores
   - Base de données SQLite éphémère (regénérée à chaque redémarrage)
3. **Couche Web** : Leaderboard HTML + API (actualisation 10s)
   - Frontend pur (Vanilla JS, pas de framework)
   - Récupération de 191 scores dedupliqués
   - Tri par Points/Distance/Bedos/Date
   - Recherche par nom de joueur

3.4.1 Synchronisation Automatique du Backup

Après chaque partie, la fonction `_synchroniser_backup_depuis_serveur()` exécutée en thread :

```
def sauvegarder_nouveau_score(...):
    # 1. Sauvegarder localement
    scores.append(nouvelle_entree)
    with open(FICHIER_SCORES, 'w') as f:
        json.dump(scores, f)

    # 2. Envoyer à l'API (thread)
    _envoyer_score_api(nouvelle_entree)

    # 3. Télécharger TOUS les scores du serveur
    # et fusionner dans le backup local
    _synchroniser_backup_depuis_serveur()
```

Cela garantit qu'aucun score n'est perdu lors d'un déploiement ou redémarrage serveur.

4 Choix de Conception

4.1 Pourquoi Python ?

Justification

- **Rapidité de développement** : Syntaxe claire, moins de boilerplate
- **Écosystème** : Pygame mature pour graphisme 2D, Flask pour serveur
- **Cross-plateforme** : Code identique sur Windows/Linux/macOS
- **Flexibilité** : Facile de switcher entre GUI (Pygame) et TUI (Curses)
- **Prototypage** : Idéal pour game jam et projets académiques

4.2 Architecture Monolithique vs Microservices

Choix : Monolithique avec séparation concerns

- Chaque module a une responsabilité unique
- `moteur_jeu.py` : Logique pure, pas de GUI
- `main_graphique.py` / `main_terminal.py` : Couches présentation
- `gestion_scores.py` : Logique métier (persistance)
- Facile à tester en isolé

4.3 Interface Unique vs Multiples

Choix : Deux interfaces

- **Pygame (Graphique)** : Moderne, accès grand public
- **Curses (Terminal)** : Rétro, hommage aux jeux 80s-90s
- Partagent la même logique de jeu (`moteur_jeu.py`)
- Minimise duplication, permet expérimentation UI

4.4 Stockage : JSON vs Base de Données

Choix : JSON pour local, SQLite en production

- **Local (JSON)** : Zéro dépendance, fichier texte, versionnage Git facile
- **Serveur (SQLite)** : Léger, intégré à Python, suffisant pour 191 scores
- Pas besoin de MySQL/PostgreSQL pour ce cas d'usage
- Les scores sont sauvegardés dans Git → backup automatique

4.5 Déploiement : Railway.app

Choix : Conteneur Docker sur Railway

- Déploiement simple : `git push` déclenche build + deploy
- Certificat SSL automatique
- Scaling gratuit pour petit trafic
- Alternative : Heroku (fermé), Replit, PythonAnywhere

5 Processus de Développement

5.1 Phases du Projet

Phase	Objectifs	Durée
1. Prototype	Logique jeu + première interface	2-3 jours
2. Polissage	Graphisme, son, UI	1-2 jours
3. Sync Scores	API, leaderboard, stockage	2-3 jours
4. Déploiement	Railway, domaine, optimisations	1-2 jours
5. Bug Fixes	Alignement tableau, backup sync	1 jour

5.2 Itérations Principales

5.2.1 Animation du Fond

Problème : Les weeds (mauvaises herbes) du fond n'animaient pas correctement.
Solution : Après 8 tentatives, implémentation d'une animation diagonale en CSS :

```
@keyframes weedsDiagonal {
    0% { background-position: 0 0; }
    100% { background-position: 200px -200px; }
}
animation: weedsDiagonal 6s linear infinite;
```

Apprentissage clé : En CSS, `background-position: X Y` où X positif = droite, Y négatif = haut.

5.2.2 Responsive Design

Problème : Le leaderboard n'était pas lisible sur mobile/tablet.
Solution : Media queries multiples avec colonnes adaptées :
— Desktop : 6 colonnes à 80px/1fr/120px/120px/120px/140px
— Tablet (768px) : Réduction à 50px/1fr/100px/120px/120px/130px
— Mobile (480px) : Scroll horizontal avec hint "(← glisse →)"
— Very small (320px) : Nouvelles réductions

5.2.3 Perte de Données

Problème : À chaque `git push`, les scores joués localement étaient perdus.
Solution : Fonction `_synchroniser_backup_depuis_serveur()` qui télécharge les scores du serveur après chaque partie et les fusionne avec le backup local.
Impact : Zéro perte de données, même lors de déploiements.

5.3 Gestion des Défauts

Défaut	Cause Racine	Correction
Favicon ne s'affiche pas	Cache navigateur	Ajout `?v=1`
Date mal triée	Format DD/MM/YYYY interprété comme texte	Parser JS personnalisé
Colonnes misalignées	CSS dupliqué et conflictuel	Consolidation CSS
Doublons leaderboard	Scores importés plusieurs fois	Clé composite (nom, score, date)

6 Résultats et Performances

6.1 Métriques du Leaderboard

— **Scores actifs** : 191 entrées uniques (après déduplication)
— **Joueurs distincts** : 45-50
— **Meilleur score** : 751 points (401 mètres, 7 bedos)
— **Temps de chargement** : <500ms (API + parse JSON)

6.2 Performance Moteur Jeu

— **FPS cible** : 60 (60 ms/frame)
— **FPS réel** : 55-60 sur hardware modéré

— **Mémoire** : 80-120 MB (Pygame + assets)
— **Startup** : 2-3 secondes (chargement assets + sync)

6.3 Uptime Serveur

— **Disponibilité** : 99.5
— **Response time API** : <100ms en moyenne
— **Leaderboard** : Actualisation 10 secondes

7 Améliorations Futures

7.1 Échelle Court Terme

Power-ups : Invincibilité, ralentissement temps
Skins personnalisables pour le joueur
Achievements et badges
Mode multijoueur local (splitscreen)

7.2 Échelle Moyen Terme

Application mobile (Flutter/React Native)
Matchmaking en ligne (1v1)
Boutique cosmétique ($)
Replay/Spectate system

7.3 Échelle Long Terme

Engine personnalisé (Godot/Unreal)
Campagne histoire
Éditeur de niveaux utilisateur
Intégration Steam

8 Conclusion

Thouv'Run démontre comment construire un jeu complet avec :
— Architecture modulaire et testable (séparation logique/UI)
— Déploiement production avec synchronisation robuste
— Interface responsive multi-plateforme
— Pipeline de développement itératif (Git, Railway)
Le projet illustre les bonnes pratiques du développement jeu indie :
— ✓ Scope limité et réalisable
— ✓ Itérations rapides et feedback utilisateur
— ✓ Passion + polish (pixel art, musique)
— ✓ Code propre et maintenable

« Un jeu simple, mais bien fait. Voilà Thouv'Run ! »