Jenga Build System — Guide Complet d'Utilisation

Version 2.0 — Février 2026

Table des matières

1. Présentation

Jenga est un système de build multi-plateforme pour C/C++ (et autres langages) écrit en Python. Il utilise des fichiers de configuration `.jenga` — du Python standard enrichi d'un DSL (Domain-Specific Language) — pour décrire vos projets.

Philosophie

- **Un seul fichier** décrit votre projet pour toutes les plateformes
- **Python natif** : toute la puissance de Python est disponible dans `.jenga`
- **Zéro dépendance** pour la logique de build (pas de CMake, Ninja, Make)
- **Incremental** : seuls les fichiers modifiés sont recompilés
- **Parallèle** : compilation multi-cœurs automatique

Plateformes hôtes

rihen.universe@gmail.com • teuguiasederis@gmail.com
(+237) 693-761-773

Hôte	Cibles disponibles
Windows	Windows, Linux (cross), Android, Web, Xbox
Linux	Linux, Android, Web
macOS	macOS, iOS, tvOS, watchOS, Android, Web

---

2. Installation

Depuis le dépôt source

```
git clone https://github.com/votre-org/jenga.git
cd jenga
pip install -e .
```

Vérification

```
jenga --version
# ou
python Jenga/jenga.py --version
```

Raccourcis (recommandés)

Ajoutez `jenga.bat` (Windows) ou `jenga.sh` (Linux/macOS) au PATH pour utiliser `jenga` directement :

```
# Windows — ajouter au PATH :
C:\chemin\vers\Jenga\Jenga\

# Linux / macOS :
export PATH=$PATH:/chemin/vers/Jenga/Jenga/
```

---

3. Concepts fondamentaux

Workspace

Le **workspace** est le conteneur racine. Il regroupe tous les projets, les toolchains et la configuration globale.

```
with workspace("MonProjet"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.WINDOWS, TargetOS.LINUX])
    ...
```

Project

Un **project** est une unité de compilation : exécutable, bibliothèque statique, bibliothèque partagée, ou suite de tests.

```
with project("MaLib"):
    staticlib()
    language("C++")
    files(["src/**.cpp"])
```

Toolchain

Un **toolchain** décrit un compilateur : chemin, cibles, flags. Jenga détecte automatiquement les toolchains disponibles sur le système.

Filter

Un **filter** permet d'activer du code de configuration uniquement pour certaines conditions (plateforme, config, options).

```
with filter("system:Windows"):
    links(["user32", "gdi32"])
```

---

4. Structure d'un fichier .jenga

```
#!/usr/bin/env python3
# Mon projet — MonProjet.jenga
import os
from Jenga import *
from Jenga.GlobalToolchains import RegisterJengaGlobalToolchains

with workspace("MonWorkspace"):
    RegisterJengaGlobalToolchains()       # Détection auto des toolchains
    configurations(["Debug", "Release"])
    targetoses([TargetOS.WINDOWS, TargetOS.LINUX, TargetOS.ANDROID,
TargetOS.WEB])
    targetarchs([TargetArch.X86_64, TargetArch.ARM64, TargetArch.WASM32])
    startproject("MonApp")                # Projet démarré par défaut

    androidsdkpath(os.getenv("ANDROID_SDK_ROOT", ""))
    androidndkpath(os.getenv("ANDROID_NDK_ROOT", ""))

    with project("MonLib"):
```

```
        staticlib()
        language("C++")
        cppdialect("C++17")
        location("lib")
        files(["src/**.cpp"])
        includedirs(["include"])

    with project("MonApp"):
        consoleapp()
        language("C++")
        cppdialect("C++17")
        location("app")
        files(["src/**.cpp"])
        includedirs(["../lib/include"])
        links(["MonLib"])
        dependson(["MonLib"])

        objdir("%{wks.location}/Build/Obj/%{cfg.buildcfg}-%{cfg.system}/%
{prj.name}")
        targetdir("%{wks.location}/Build/Bin/%{cfg.buildcfg}-%{cfg.system}/%
{prj.name}")

        with filter("system:Windows"):
            usetoolchain("clang-mingw")
            links(["user32"])

        with filter("system:Linux"):
            usetoolchain("zig-linux-x64")
            links(["pthread", "dl"])

        with filter("system:Android"):
            windowedapp()
            usetoolchain("android-ndk")
            androidapplicationid("com.exemple.monapp")
            androidminsdk(24)
            androidtargetsdk(34)
            androidabis(["arm64-v8a", "x86_64"])
            androidnativeactivity(True)

        with filter("system:Web"):
            usetoolchain("emscripten")
            emscripteninitialmemory(32)

        with filter("config:Debug"):
            defines(["_DEBUG"])
            optimize("Off")
            symbols(True)

        with filter("config:Release"):
            defines(["NDEBUG"])
            optimize("Speed")
            symbols(False)
```

5. Commandes disponibles

`jenga build`

Compile le workspace ou un projet spécifique.

```
# Compilation de base
jenga build

# Avec configuration
jenga build --config Release

# Plateforme spécifique
jenga build --platform windows-x86_64
jenga build --platform linux-x86_64
jenga build --platform android-arm64
jenga build --platform web-wasm32
jenga build --platform xbox-x86_64

# Toutes les plateformes déclarées
jenga build --platform jengaall

# Projet spécifique
jenga build --target MonApp

# Compilation parallèle (N threads)
jenga build --jobs 8
# Auto-détection (CPU - 1 threads)
jenga build -j 0

# Mode verbeux (affiche les commandes)
jenga build --verbose

# Sans cache
jenga build --no-cache

# Options personnalisées
jenga build --with-sdl=/opt/sdl3
jenga build --no-headless
```

`jenga clean`

Supprime les artefacts de build.

```
jenga clean
jenga clean --config Release
jenga clean --platform android-arm64
```

jenga rebuild

Clean + Build en une commande.

```
jenga rebuild --platform windows-x86_64 --config Release
```

jenga run

Exécute le projet démarré (startproject).

```
jenga run
jenga run --config Release
```

jenga test

Exécute les suites de tests Unitest.

```
jenga test
jenga test --config Debug --verbose
```

jenga watch

Surveille les fichiers sources et recompile automatiquement.

```
jenga watch
jenga watch --config Debug --platform linux-x86_64
```

jenga info

Affiche les informations sur le workspace et les projets.

```
jenga info
jenga info --verbose
```

jenga init

Crée un nouveau workspace.

```
```

```
jenga init MonWorkspace
jenga init MonWorkspace --template console
```

jenga create

Ajoute un nouveau projet au workspace.

```
jenga create MonProjet
jenga create MaBib --kind staticlib
```

jenga keygen

Génère une keystore pour signer les APK Android.

```
jenga keygen
jenga keygen --alias mon-alias --validity 3650
```

jenga sign

Signe un APK Android.

```
jenga sign
jenga sign --apk Build/Bin/Release-Android/MonApp-Release.apk
```

jenga package

Empaquète l'application pour distribution.

```
jenga package
jenga package --config Release
```

jenga install

Installe/déploie les artefacts.

```
jenga install
jenga install --target /usr/local
```

jenga docs

Génère la documentation du projet.

```
jenga docs
```

jenga bench

Lance des benchmarks.

```
jenga bench
```

jenga profile

Lance le profiler sur l'application.

```
jenga profile
```

jenga gen

Génère des fichiers de build pour d'autres systèmes (CMake, Ninja...).

```
jenga gen --format cmake
jenga gen --format ninja
```

jenga config

Gère la configuration globale Jenga.

```
jenga config list
jenga config set toolchain.default clang-mingw
```

jenga examples

Liste et gère les exemples.

```
jenga examples list
jenga examples run 01_hello_console
```

---

6. Plateformes supportées

Identifiants de plateformes

Plateforme	**Identifiant `--platform`**
Windows x64	`windows-x86_64`
Windows x86	`windows-x86`
Linux x64	`linux-x86_64`
Linux ARM64	`linux-arm64`
macOS ARM64	`macos-arm64`
macOS x64	`macos-x86_64`
Android ARM64	`android-arm64`
Android x86_64	`android-x86_64`
WebAssembly	`web-wasm32`
Xbox Series	`xbox-x86_64`
Xbox One	`xboxone-x86_64`
iOS	`ios-arm64`
PS4	`ps4-x86_64`
PS5	`ps5-x86_64`

`TargetOS` disponibles

```
TargetOS.WINDOWS       # Windows
TargetOS.LINUX         # Linux
TargetOS.MACOS         # macOS
TargetOS.ANDROID       # Android (NDK)
TargetOS.IOS           # iOS (Xcode)
TargetOS.TVOS          # tvOS
TargetOS.WATCHOS       # watchOS
TargetOS.WEB           # WebAssembly (Emscripten)
TargetOS.XBOX_ONE      # Xbox One
TargetOS.XBOX_SERIES   # Xbox Series X|S
TargetOS.PS4           # PlayStation 4
TargetOS.PS5           # PlayStation 5
TargetOS.SWITCH        # Nintendo Switch
TargetOS.HARMONYOS     # HarmonyOS (Huawei)
TargetOS.FREEBSD       # FreeBSD
```

`TargetArch` disponibles

```
```

```
TargetArch.X86       # x86 (32 bits)
TargetArch.X86_64    # x86_64 (64 bits)
TargetArch.ARM       # ARMv7
TargetArch.ARM64     # AArch64
TargetArch.WASM32    # WebAssembly 32 bits
TargetArch.WASM64    # WebAssembly 64 bits
TargetArch.MIPS      # MIPS
TargetArch.POWERPC   # PowerPC
TargetArch.RISCV64   # RISC-V 64 bits
```

---

7. Toolchains personnalisés

Déclaration d'un toolchain inline

```
with workspace("MonWorkspace"):
    with toolchain("mon-clang", "clang"):
        settarget("Linux", "x86_64", "gnu")
        targettriple("x86_64-unknown-linux-gnu")
        ccompiler("/usr/bin/clang-17")
        cppcompiler("/usr/bin/clang++-17")
        linker("/usr/bin/clang++-17")    # Le compilateur comme driver de link
        archiver("/usr/bin/llvm-ar-17")
        sysroot("/opt/sysroot-x64")      # Optionnel
        cflags(["-O2", "--target=x86_64-unknown-linux-gnu"])
        cxxflags(["-O2", "-std=c++20", "--target=x86_64-unknown-linux-gnu"])
        ldflags(["--target=x86_64-unknown-linux-gnu"])
```

Familles de compilateurs

```
CompilerFamily.CLANG          # clang / clang++
CompilerFamily.GCC            # gcc / g++
CompilerFamily.MSVC           # cl.exe (MSVC)
CompilerFamily.ANDROID_NDK    # NDK clang
CompilerFamily.EMSCRIPTEN     # emcc / em++
CompilerFamily.APPLE_CLANG    # Apple clang
```

Registre global de toolchains (.jenga/toolchains_registry.json)

Vous pouvez enregistrer un toolchain une fois pour toute votre machine :

```
{
  "toolchains": [
    {
      "name": "clang-17-linux",
```

```
      "compilerFamily": "clang",
      "targetOs": "Linux",
      "targetArch": "x86_64",
      "targetEnv": "gnu",
      "targetTriple": "x86_64-unknown-linux-gnu",
      "ccPath": "/usr/bin/clang-17",
      "cxxPath": "/usr/bin/clang++-17",
      "arPath": "/usr/bin/llvm-ar-17",
      "ldPath": "/usr/bin/clang++-17",
      "cflags": ["-O2"],
      "cxxflags": ["-std=c++20"],
      "ldflags": [],
      "sysroot": ""
    }
  ],
  "sdk": {
    "androidSdkPath": "/opt/android-sdk",
    "androidNdkPath": "/opt/android-sdk/ndk/27.0.12077973"
  }
}
```

Le registre est dans `<JENGA_ROOT>/.jenga/toolchains_registry.json`.

`RegisterJengaGlobalToolchains()` — Toolchains prédéfinis

Cette fonction enregistre automatiquement les toolchains disponibles sur votre machine :

```
from Jenga.GlobalToolchains import RegisterJengaGlobalToolchains

with workspace("MonWorkspace"):
    RegisterJengaGlobalToolchains()
    # Enregistre automatiquement :
    # - android-ndk    (si ANDROID_NDK_ROOT défini)
    # - emscripten     (si EMSDK ou emcc dans PATH)
    # - zig-linux-x64  (si zig dans PATH)
    # - clang-mingw    (si clang dans PATH + MSYS2/UCRT64)
    # - clang-native   (si CLANG_BASE ou clang dans PATH)
    # - clang-cross-linux (si clang+sysroot Linux)
    # - clang-cl       (si clang-cl sur Windows)
```

Toolchains individuels

```
from Jenga.GlobalToolchains import (
    ToolchainAndroidNDK,
    ToolchainEmscripten,
    ToolchainZigLinuxX64,
    ToolchainClangMinGW,
    ToolchainClangNative,
```

```
    ToolchainClangCl,
)

with workspace("MonWorkspace"):
    # Enregistrement individuel avec chemin explicite
    ToolchainAndroidNDK(ndk_root="/opt/android-ndk")
    ToolchainEmscripten(emsdk_root="/opt/emsdk")
    ToolchainClangNative(clang_base="/usr/lib/llvm-17")
```

Variables d'environnement reconnues

Variable	Toolchain	Utilisation
ANDROID_NDK_ROOT	android-ndk	Chemin NDK
ANDROID_NDK_HOME	android-ndk	Chemin NDK (alternative)
ANDROID_SDK_ROOT	android-ndk	Chemin SDK
EMSDK	emscripten	Répertoire racine emsdk
EMSCRIPTEN	emscripten	Répertoire emscripten/
CLANG_BASE	clang-native	Répertoire racine LLVM
MSVC_BASE	clang-cl	Répertoire MSVC
MINGW_ROOT	clang-mingw	Répertoire MSYS2/MinGW
ZIG	zig-*	Chemin de l'exécutable zig

8. Système de filtres

Les filtres permettent de conditionner la configuration selon le contexte de build.

Syntaxe de base

```
with filter("system:Windows"):
    links(["user32", "gdi32", "kernel32"])

with filter("config:Debug"):
    defines(["_DEBUG", "DEBUG"])
    optimize("Off")
    symbols(True)

with filter("config:Release"):
    defines(["NDEBUG"])
    optimize("Speed")
    symbols(False)
```

```
with filter("arch:arm64"):
    cflags(["-march=armv8-a"])
```

Tokens de filtre disponibles

Token	Exemples	Description
`system:X`	`system:Windows`	Système cible
`config:X`	`config:Debug`	Configuration de build
`arch:X`	`arch:arm64`	Architecture cible
`platform:X`	`platform:android-arm64`	Plateforme complète
`target:X`	`target:MonApp`	Projet cible
`options:X`	`options:headless`	Option CLI personnalisée
`verbose`	`verbose`	Mode verbeux actif
`no-cache`	`no-cache`	Cache désactivé

Opérateurs logiques

```
# ET logique
with filter("system:Windows && config:Debug"):
    defines(["WIN_DEBUG"])

# OU logique
with filter("system:Windows || system:Linux"):
    defines(["DESKTOP_PLATFORM"])

# NON logique
with filter("!system:Web"):
    links(["pthread"])
```

Alias système

Alias	Système correspondant
`Windows`	`system:Windows`
`Linux`	`system:Linux`
`Android`	`system:Android`
`Web`, `Emscripten`	`system:Web`
`Debug`	`config:Debug`
`Release`	`config:Release`

Alias	Système correspondant
`xbox`, `xboxseries`	`system:XboxSeries`

9. Variables dynamiques

Tokens disponibles

Variable	Description	Exemple
`%{wks.name}`	Nom du workspace	`MonWorkspace`
`%{wks.location}`	Chemin du workspace	`/home/user/project`
`%{prj.name}`	Nom du projet	`MonApp`
`%{cfg.buildcfg}`	Configuration	`Debug`
`%{cfg.system}`	Système cible	`Windows`
`%{cfg.arch}`	Architecture	`x86_64`
`%{cfg.platform}`	Plateforme complète	`Windows-x86_64`
`%{Jenga.Unitest.Source}`	Source Unitest	chemin interne
`%{Jenga.Unitest.Include}`	Include Unitest	chemin interne

Utilisation

```
with project("MonApp"):
    objdir("%{wks.location}/Build/Obj/%{cfg.buildcfg}-%{cfg.system}/%
{prj.name}")
    targetdir("%{wks.location}/Build/Bin/%{cfg.buildcfg}-%{cfg.system}/%
{prj.name}")
```

10. Tests unitaires avec Unitest

Jenga intègre son propre framework de tests C++.

Configuration du workspace

```
with workspace("MonWorkspace"):
    with unitest() as u:
        u.Compile(cxxflags=["-fexceptions"])  # Compiler Unitest depuis les
sources
        # ou
        # u.Precompiled()  # Utiliser la version précompilée
```

Ajout d'une suite de tests à un projet

```
with project("MonLib"):
    staticlib()
    language("C++")
    files(["src/**.cpp"])
    includedirs(["include"])

    with test():
        testfiles(["tests/**.cpp"])
```

Écrire des tests C++

```
// tests/test_math.cpp
#include <Unitest/TestMacro.h>

TEST_CASE("Addition") {
    int result = add(2, 3);
    ASSERT_EQ(result, 5);
}

TEST_CASE("Division par zéro") {
    ASSERT_THROWS(divide(1, 0), std::runtime_error);
}

// Benchmarks
BENCHMARK("Performance Add") {
    for (int i = 0; i < 10000; ++i) {
        add(i, i + 1);
    }
}
```

Macros disponibles

Macro	Description
`TEST_CASE("nom")`	Déclare un test
`ASSERT_EQ(a, b)`	Égalité
`ASSERT_NE(a, b)`	Inégalité
`ASSERT_LT(a, b)`	Inférieur
`ASSERT_GT(a, b)`	Supérieur
`ASSERT_LE(a, b)`	Inférieur ou égal

Macro	Description
`ASSERT_GE(a, b)`	Supérieur ou égal
`ASSERT_TRUE(cond)`	Condition vraie
`ASSERT_FALSE(cond)`	Condition fausse
`ASSERT_THROWS(expr, type)`	Exception attendue
`ASSERT_NO_THROW(expr)`	Pas d'exception
`ASSERT_NEAR(a, b, eps)`	Égalité approchée (float)
`BENCHMARK("nom")`	Déclare un benchmark

Exécution

```
jenga test
jenga test --verbose
jenga test --config Release
```

---

11. WebAssembly / Emscripten

Prérequis

Installez emsdk : https://emscripten.org/docs/getting_started/downloads.html

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
# Définir EMSDK ou EMSCRIPTEN dans les variables d'environnement
```

Workspace WebAssembly

```
from Jenga import *
from Jenga.GlobalToolchains import RegisterJengaGlobalToolchains

with workspace("WebApp"):
    RegisterJengaGlobalToolchains()
    configurations(["Debug", "Release"])
    targetoses([TargetOS.WEB])
    targetarchs([TargetArch.WASM32])

    with project("MonApp"):
        consoleapp()
```

```
        language("C++")
        cppdialect("C++17")
        files(["src/**.cpp"])
        usetoolchain("emscripten")

        emscripteninitialmemory(32)         # 32 MB RAM initiale
        emscriptenstacksize(8)               # 8 MB stack
        emscriptenexportname("MyModule")    # Nom du module JS
        emscriptenextraflags(["-s ASYNCIFY"]) # Flags extra
```

Options Emscripten disponibles

```
emscriptenshellfile("shell.html")           # Template HTML personnalisé
emscriptencanvasid("mycanvas")              # ID du canvas HTML
emscripteninitialmemory(16)                 # RAM initiale en MB
emscriptenstacksize(5)                      # Stack en MB
emscriptenexportname("Module")             # Nom export JS
emscriptenextraflags(["-s ASYNCIFY"])       # Flags additionnels
emscriptenuseFullscreenShell(True)         # Shell plein écran
```

Fichiers générés

Après compilation, dans le répertoire de sortie :

```
Build/Bin/Release-Web/MonApp/
├── MonApp.html          ← Page HTML principale
├── MonApp.js            ← Glue code JavaScript
├── MonApp.wasm          ← Binaire WebAssembly
├── run_MonApp.bat       ← Lanceur Windows (serveur HTTP local)
└── run_MonApp.sh        ← Lanceur Linux/macOS (serveur HTTP local)
```

Exécution sans erreurs CORS

IMPORTANT : N'ouvrez jamais `MonApp.html` directement en double-cliquant. Utilisez les scripts générés :

```
# Windows
Build/Bin/Release-Web/MonApp/run_MonApp.bat

# Linux / macOS
chmod +x Build/Bin/Release-Web/MonApp/run_MonApp.sh
./Build/Bin/Release-Web/MonApp/run_MonApp.sh

# Avec port personnalisé
run_MonApp.bat 9090
./run_MonApp.sh 9090
```

Puis ouvrez : `http://localhost:8080/MonApp.html`

12. Android NDK

Prérequis

1. **Android SDK** : https://developer.android.com/studio
2. **NDK** : via Android Studio ou `sdkmanager --install "ndk;27.0.12077973"`
3. **JDK 17+** : https://adoptium.net/

Variables d'environnement

```
# Windows
set ANDROID_SDK_ROOT=C:\Android\sdk
set ANDROID_NDK_ROOT=C:\Android\sdk\ndk\27.0.12077973

# Linux / macOS
export ANDROID_SDK_ROOT=$HOME/Android/sdk
export ANDROID_NDK_ROOT=$HOME/Android/sdk/ndk/27.0.12077973
```

Configuration Android

```
with workspace("AndroidApp"):
    RegisterJengaGlobalToolchains()
    configurations(["Debug", "Release"])
    targetoses([TargetOS.ANDROID])
    targetarchs([TargetArch.ARM64, TargetArch.X86_64])

    androidsdkpath(os.getenv("ANDROID_SDK_ROOT", ""))
    androidndkpath(os.getenv("ANDROID_NDK_ROOT", ""))

    with project("MonApp"):
        windowedapp()                    # NativeActivity = windowed
        language("C++")
        files(["src/**.cpp"])

        with filter("system:Android"):
            usetoolchain("android-ndk")

        androidapplicationid("com.exemple.monapp")
        androidminsdk(24)
        androidtargetsdk(34)
        androidcompilesdk(34)
        androidabis(["armeabi-v7a", "arm64-v8a", "x86", "x86_64"])
        androidnativeactivity(True)
        androidversioncode(1)
        androidversionname("1.0")
```

```
        androidscreenorientation("sensorLandscape")
        androidpermissions(["android.permission.INTERNET"])
        androidassets(["assets/**"])
```

ABIs Android

ABI	Description	Usage
`arm64-v8a`	ARM 64 bits	Smartphones modernes (recommandé)
`armeabi-v7a`	ARM 32 bits	Appareils anciens
`x86`	x86 32 bits	Émulateurs anciens
`x86_64`	x86 64 bits	Émulateurs modernes (MEmu, BlueStacks)

Build et signature

```
# Build Debug (APK non signé)
jenga build --platform android-arm64 --config Debug

# Génération keystore
jenga keygen --alias monapp-key --keystore monapp.keystore

# Build Release + signature
jenga build --platform android-arm64 --config Release
jenga sign
```

APK universel

Quand plusieurs ABIs sont définies, Jenga génère automatiquement un **APK universel** contenant tous les `.so` :

```
Build/Bin/Debug-Android/MonApp/android-build-universal/
└── MonApp-Debug.apk    ← APK universel (toutes ABIs)
```

---

13. Xbox (GDK / UWP)

Prérequis

- Windows 10/11 avec Visual Studio 2022
- Microsoft GDK (Gaming Development Kit) : `winget install Microsoft.Gaming.GDK`
- GDKX pour Xbox Series X|S (licence EA requise)

Configuration Xbox

```
with workspace("XboxGame"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.XBOX_SERIES])
    targetarchs([TargetArch.X86_64])

    xboxmode("gdk")              # "gdk" ou "uwp"
    xboxplatform("Scarlett")     # "Scarlett" (Series) ou "XboxOne"

    with project("MonJeu"):
        windowedapp()
        language("C++")
        cppdialect("C++20")
        files(["src/**.cpp"])
        includedirs(["include"])
        links(["GameRuntime", "xgameruntime"])

        xboxsigningmode("test")
        xboxpackagename("MonJeu")
        xboxpublisher("MaCompagnie")
        xboxversion("1.0.0.0")
```

Option CLI Xbox

```
# Mode GDK (défaut)
jenga build --platform xbox-x86_64

# Mode UWP Dev Mode
jenga build --platform xbox-x86_64 --xbox-mode=uwp
```

---

14. Compilation incrémentale et cache

Jenga utilise un système de cache multi-niveaux pour accélérer les builds :

1. Cache par timestamps

Le fichier objet `.o` est plus récent que le `.cpp` → pas de recompilation.

2. Cache par fichiers de dépendances (.d)

GCC/Clang génèrent des fichiers `.d` (format Make) listant tous les headers inclus. Si un header est modifié, tous les `.o` qui en dépendent sont recompilés.

3. Cache par signature

Chaque fichier objet a un fichier sidecar `.jenga_sig` (SHA256) contenant une empreinte de :

- Nom du compilateur et version

- Tous les flags de compilation
- Chemin des includes
- Defines actifs
- Configuration (Debug/Release)
- Plateforme cible

Si la signature change (ex: ajout d'un `-DNOUVEAU_FLAG`), le fichier est recompilé.

Performances observées

Scénario	Temps initial	Temps incrémental
Hello World (Windows)	0.90s	**0.12s** (7.5x)
Bibliothèque + App (Windows)	0.98s	**0.12s** (8x)
OpenGL Triangle (Linux)	1.03s	**0.51s** (2x)
WasmApp (Web)	2.43s	**0.28s** (8.7x)

Contrôle du cache

```
# Désactiver le cache (forcer recompilation)
jenga build --no-cache

# Forcer une recompilation complète
jenga rebuild
```

---

15. Compilation parallèle

Jenga utilise `ThreadPoolExecutor` pour compiler plusieurs fichiers en parallèle.

Contrôle du parallélisme

```
# Auto-détection (CPU - 1 threads)
jenga build -j 0

# N threads explicites
jenga build --jobs 8
jenga build -j 4

# Compilation séquentielle
jenga build --jobs 1
```

Comportement

- Les modules C++20 (BMI) sont toujours précompilés séquentiellement (dépendances)

- Les `.cpp` réguliers sont compilés en parallèle
- L'édition de liens est toujours séquentielle (ordre des dépendances)
- La progression est affichée en temps réel

---

16. Projets multi-plateformes

Pattern recommandé

```
with workspace("CrossPlatform"):
    RegisterJengaGlobalToolchains()
    configurations(["Debug", "Release"])
    targetoses([
        TargetOS.WINDOWS,
        TargetOS.LINUX,
        TargetOS.ANDROID,
        TargetOS.WEB,
    ])
    targetarchs([TargetArch.X86_64, TargetArch.ARM64, TargetArch.WASM32])

    with project("Game"):
        consoleapp()
        language("C++")
        files(["src/**.cpp"])

        # Toolchain par plateforme
        with filter("system:Windows"):
            usetoolchain("clang-mingw")
            links(["user32", "gdi32", "opengl32"])

        with filter("system:Linux"):
            usetoolchain("zig-linux-x64")
            links(["GL", "X11", "pthread"])

        with filter("system:Android"):
            windowedapp()
            usetoolchain("android-ndk")
            androidapplicationid("com.exemple.game")
            androidminsdk(24)
            androidtargetsdk(34)
            androidabis(["arm64-v8a", "x86_64"])
            androidnativeactivity(True)
            links(["EGL", "GLESv3", "android", "log"])

        with filter("system:Web"):
            usetoolchain("emscripten")
            emscripteninitialmemory(32)
            links(["openal", "EGL"])

        # Configuration Debug/Release
        with filter("config:Debug"):
```

```
            defines(["DEBUG"])
            optimize("Off")
            symbols(True)

        with filter("config:Release"):
            defines(["NDEBUG"])
            optimize("Speed")
            symbols(False)
```

Build toutes plateformes

```
# Construire pour toutes les plateformes déclarées
jenga build --platform jengaall --config Release

# Construire pour chaque plateforme individuellement
jenga build --platform windows-x86_64 --config Release
jenga build --platform linux-x86_64 --config Release
jenga build --platform android-arm64 --config Release
jenga build --platform web-wasm32 --config Release
```

---

17. C++20 Modules

Configuration

```
with project("ModulesApp"):
    consoleapp()
    language("C++")
    cppdialect("C++20")
    files(["src/**.cpp", "modules/**.cppm"])
```

Fichiers modules supportés

Extension	Description
`.cppm`	Module C++20 (standard)
`.ixx`	Module C++20 (MSVC style)
`.mpp`	Module C++20 (alternative)
`.c++m`	Module C++20 (GCC style)

Exemple de module

```
```

```cpp
// modules/math.cppm
export module math;

export int add(int a, int b) {
    return a + b;
}

export double sqrt_approx(double x) {
    return x * 0.5;
}
```

```cpp
// src/main.cpp
import math;
#include <iostream>

int main() {
    std::cout << add(3, 4) << "\n";  // 7
    return 0;
}
```

BMI (Binary Module Interface)

Jenga précompile automatiquement les BMI :

- **Clang** : `.pcm` (dans `Build/Obj/modules/`)
- **MSVC** : `.ifc` (dans `Build/Obj/modules/`)
- **GCC** : direct `.o` (pas de BMI séparé)

---

18. Exemples de référence

#	Exemple	Plateformes	Description
01	hello_console	Windows, Linux, Android, Web	Hello World
02	static_library	Toutes	Bibliothèque statique
03	shared_library	Toutes	Bibliothèque partagée (.dll/.so)
04	unit_tests	Windows, Linux	Tests unitaires Unitest
05	android_ndk	Android	APK NativeActivity
06	ios_app	iOS (macOS requis)	App iOS
07	web_wasm	Web	WebAssembly + scripts runner
08	custom_toolchain	Windows	Toolchain clang personnalisé
09	multi_projects	Windows, Linux, Android, Web	Dépendances multi-projets

#	Exemple	Plateformes	Description
10	modules_cpp20	Windows, Linux	Modules C++20
11	benchmark	Toutes	Benchmarks Unitest
12	external_includes	Toutes	Inclusions système
13	packaging	Toutes	Packaging distribution
14	cross_compile	Linux croisé	Cross-compilation
15	window_win32	Windows	Fenêtre Win32
16	window_x11_linux	Linux	Fenêtre X11
17	window_macos_cocoa	macOS	Fenêtre Cocoa
18	window_android_native	Android	Fenêtre NativeActivity
19	window_web_canvas	Web	Canvas HTML5
20	window_ios_uikit	iOS	Fenêtre UIKit
21	zig_cross_compile	Linux (cross)	Zig toolchain
22	nk_multiplatform	Windows, Linux, Android, Web	UI Nuklear
23	android_sdl3	Android	SDL3 NDK
24	all_platforms	Toutes	Toutes plateformes
25	opengl_triangle	Windows, Linux, Android, Web	OpenGL triangle
26	xbox_project_kinds	Xbox	Variantes Xbox
27	nk_window	Windows, Linux, Android, Web	UI Nuklear avancé

19. GlobalToolchains — Registre global

Utilisation simple

```
from Jenga.GlobalToolchains import RegisterJengaGlobalToolchains

with workspace("MonWorkspace"):
    RegisterJengaGlobalToolchains()
```

Fonctions individuelles

```
from Jenga.GlobalToolchains import (
    ToolchainAndroidNDK,      # android-ndk
    ToolchainEmscripten,      # emscripten
```

```
    ToolchainZigLinuxX64,      # zig-linux-x64
    ToolchainClangMinGW,       # clang-mingw
    ToolchainClangNative,      # clang-native
    ToolchainClangCrossLinux, # clang-cross-linux
    ToolchainClangCl,          # clang-cl (Windows MSVC)
)
```

Toolchains disponibles après `RegisterJengaGlobalToolchains()`

Nom	Famille	Cible	Condition
`android-ndk`	android-ndk	Android ARM64	ANDROID_NDK_ROOT défini
`emscripten`	emscripten	Web WASM32	EMSDK ou emcc dans PATH
`zig-linux-x64`	clang	Linux x86_64	zig dans PATH
`clang-mingw`	clang	Windows x86_64 (MinGW)	clang++ dans PATH
`clang-native`	clang	Host (auto)	CLANG_BASE ou clang dans PATH
`clang-cross-linux`	clang	Linux x86_64 (cross)	CLANG_BASE défini
`clang-cl`	clang	Windows x86_64 (MSVC)	Windows + clang-cl

Toolchain personnalisé via registre JSON

```
# Créer le fichier registre
mkdir -p .jenga
cat > .jenga/toolchains_registry.json << 'EOF'
{
  "toolchains": [
    {
      "name": "mon-gcc-12",
      "compilerFamily": "gcc",
      "targetOs": "Linux",
      "targetArch": "x86_64",
      "targetEnv": "gnu",
      "ccPath": "/usr/bin/gcc-12",
      "cxxPath": "/usr/bin/g++-12",
      "arPath": "/usr/bin/ar",
      "ldPath": "/usr/bin/g++-12",
      "cflags": ["-O2"],
      "cxxflags": ["-std=c++20", "-O2"]
    }
  ],
  "sdk": {}
}
EOF
```

```
from Jenga.Core.GlobalToolchains import ApplyGlobalRegistryToWorkspace

with workspace("MonWorkspace") as wks:
    ApplyGlobalRegistryToWorkspace(wks)
    # "mon-gcc-12" est maintenant disponible
```

---

20. FAQ et résolution de problèmes

`Toolchain 'xxx' not defined`

Cause : Vous appelez `usetoolchain("xxx")` dans un bloc projet (hors filtre) mais le toolchain n'est pas enregistré.

Solution 1 : Appeler `RegisterJengaGlobalToolchains()` avant l'utilisation.

Solution 2 : Envelopper dans un filtre (validation différée) :

```
with filter("system:Android"):
    usetoolchain("android-ndk")  # ✓ Validation différée
```

Solution 3 : Définir le toolchain manuellement :

```
with toolchain("mon-tc", "clang"):
    settarget("Linux", "x86_64")
    ccompiler("/usr/bin/clang")
    cppcompiler("/usr/bin/clang++")
    linker("/usr/bin/clang++")
    archiver("/usr/bin/ar")
```

`Android NDK not found`

Solution :

```
export ANDROID_NDK_ROOT=/chemin/vers/ndk/27.0.12077973
```

Ou dans le `.jenga` :

```
androidndkpath("/chemin/vers/ndk/27.0.12077973")
```

CORS errors lors de l'exécution d'une app WebAssembly

Cause : Ouverture de `MonApp.html` directement via `file://`.

Solution : Utiliser les scripts runner générés :

```
# Windows
run_MonApp.bat

# Linux / macOS
./run_MonApp.sh
```

`No source files found`

Cause : Les patterns de fichiers ne correspondent pas.

Solution : Vérifiez le chemin relatif à `location()` :

```
with project("MonApp"):
    location("src")           # Répertoire de base
    files(["**.cpp"])         # Relatif à location()
    # ou absolu depuis le workspace :
    files(["%{wks.location}/src/**.cpp"])
```

Build lent (pas de cache)

Le cache incrémental est automatique. Si les builds sont toujours lents :

1. Vérifiez que les fichiers `.jenga_sig` sont créés dans le répertoire `objdir`
2. Utilisez `-j 0` pour la compilation parallèle
3. Vérifiez que `objdir` et `targetdir` sont cohérents entre les builds

Xbox : `GDK not found`

Solution :

```
winget install Microsoft.Gaming.GDK
# Redémarrer VS + terminal après installation
```

Modules C++20 non trouvés

Solution : Assurez-vous que `cppdialect("C++20")` est défini et que le compilateur supporte les modules :

- Clang 16+ (`--precompile`)
- MSVC 19.28+ (`/interface`)
- GCC 12+ (`-fmodules-ts`)

Document généré par l'analyse complète du projet Jenga v2.0 — Février 2026

rihen.universe@gmail.com • teuguiasederis@gmail.com
(+237) 693-761-773
CEO – RIHEN