SEP Project
Development Environment

CodeCatalyst UG33

Team Documentation

Version 1.0

August 30, 2025

Contents

1 Quick Reference

Essential commands for immediate productivity.

1.1 One-Command Setup

Automated Environment Setup

```
# Complete development environment setup
./scripts/setup-dev-environment.sh

# Script automatically handles:
# - Platform detection (Ubuntu/macOS/Windows)
# - Dependency installation
# - vcpkg setup and configuration
# - IDE configuration files
# - Build verification and testing
```

1.2 Manual Setup Commands

Essential Manual Commands

```
# Linux development builds (native)
cmake --preset debug            # Debug build (Linux)
cmake --preset release          # Release build (Linux)
cmake --build build/release     # Build Linux executable

# Windows submission (choose one method)
cmake --preset windows-mingw    # Windows cross-compilation (CMake)
cmake --build build/windows-mingw
# OR (easier for submission)
make windows-package            # One-command Windows .exe.zip creation

# Testing and validation
ctest --test-dir build/release --output-on-failure
cmake --build build/release --target test-all
```

1.3 Pre-Setup Checklist

Before Starting

Verify you have:

- Git installed and configured
- Admin/sudo access for package installation
- Internet connection for downloading dependencies
- At least 2GB free disk space
- Text editor or IDE of choice

2 Automated Setup (Recommended)

The fastest and most reliable way to set up your development environment.

2.1 Setup Script Overview

What the Script Does

The automated setup script (`./scripts/setup-dev-environment.sh`) handles:

Platform Detection:

- Detects Ubuntu/Debian, macOS, or Windows (WSL)
- Chooses appropriate package manager
- Configures platform-specific settings

Dependency Installation:

- C++ toolchain (compiler, build tools)
- CMake 3.16+ and Ninja build system
- Code quality tools (clang-format, clang-tidy)
- Cross-compilation tools (MinGW for Windows)

Environment Configuration:

- vcpkg setup and integration
- IDE configuration files (VSCode, CLion)
- Shell environment variables
- Compile commands generation

2.2 Running the Setup Script

Automated Setup Process

```
# Navigate to project directory
cd SEP-UG-33

# Make script executable (if needed)
chmod +x scripts/setup-dev-environment.sh

# Run automated setup
./scripts/setup-dev-environment.sh

# Expected output:
# - Dependency installation progress
# - vcpkg bootstrap and configuration
# - Build verification
# - Success confirmation with next steps
```

Option	Description
--skip-vcpkg	Skip vcpkg setup (use system packages only)
--vcpkg-dir DIR	Custom vcpkg installation directory
--skip-deps	Skip system dependency installation
-h, --help	Show help message with all options

Table 1: Setup Script Command Line Options

2.3 Script Options

Custom Setup Examples

```
# Skip vcpkg if you don't need package management
./scripts/setup-dev-environment.sh --skip-vcpkg

# Use custom vcpkg location
./scripts/setup-dev-environment.sh --vcpkg-dir /opt/vcpkg

# Skip system dependencies (if already installed)
./scripts/setup-dev-environment.sh --skip-deps
```

3 Manual Setup by Platform

Step-by-step manual setup for each supported platform.

3.1 Ubuntu/Debian Setup

3.1.1 System Dependencies

Ubuntu/Debian Package Installation

```
# Update package database
sudo apt update

# Install essential development tools
sudo apt install -y \
    build-essential \
    cmake \
    ninja-build \
    git \
    curl \
    zip \
    unzip \
    tar

# Install code quality tools
sudo apt install -y \
    clang \
    clang-format \
    clang-tidy

# Install cross-compilation tools
sudo apt install -y mingw-w64
```

3.1.2 Verify Installation

Ubuntu/Debian Verification

```
# Check installed versions
gcc --version           # Should be 7.0+
cmake --version         # Should be 3.16+
ninja --version         # Any recent version
clang-format --version  # Any recent version

# Verify cross-compilation
x86_64-w64-mingw32-g++ --version  # MinGW cross-compiler
```

3.2 macOS Setup

3.2.1 Prerequisites

macOS Prerequisites

Required first steps:

- Install Xcode Command Line Tools: `xcode-select --install`
- Install Homebrew if not present: `https://brew.sh/`
- Ensure adequate disk space (Xcode tools are large)

3.2.2 Homebrew Package Installation

macOS Package Installation

```
# Verify Homebrew installation
brew --version

# Install development tools
brew install cmake ninja git curl

# Install code quality tools
brew install clang-format

# Note: clang-tidy comes with Xcode Command Line Tools
# Note: Cross-compilation to Windows not directly supported on macOS
```

3.2.3 Verify Installation

macOS Verification

```
# Check installed versions
clang --version         # Should be recent Xcode version
cmake --version         # Should be 3.16+
ninja --version         # Any recent version
clang-format --version  # Any recent version
```

3.3 Windows Setup

3.3.1 Option 1: Visual Studio (Recommended)

Visual Studio Installation

Install Visual Studio 2019 or later:

- Download from `https://visualstudio.microsoft.com/`
- Select "Desktop development with C++" workload
- Include CMake tools component
- Include Git for Windows component

Additional tools:

- Install CMake separately: `https://cmake.org/download/`
- Add CMake to system PATH
- Install Git for Windows if not included: `https://git-scm.com/`

3.3.2 Option 2: Command Line Tools

Windows Command Line Setup

```
# Install Chocolatey package manager (as Administrator)
Set-ExecutionPolicy Bypass -Scope Process -Force
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.
    ServicePointManager]::SecurityProtocol -bor 3072
iex ((New-Object System.Net.WebClient).DownloadString('https://community.
    chocolatey.org/install.ps1'))

# Install development tools via Chocolatey
choco install cmake ninja git

# Install Build Tools for Visual Studio
choco install visualstudio2019buildtools --package-parameters "--add
    Microsoft.VisualStudio.Workload.VCTools"
```

3.3.3 Windows Subsystem for Linux (WSL)

WSL Alternative

For Linux-like development on Windows:

- Install WSL2 with Ubuntu distribution
- Follow Ubuntu setup instructions within WSL
- Use Windows IDE with WSL backend
- Cross-compilation to Windows works from WSL

Benefits:

- Native Linux toolchain
- Better package management
- Consistent with CI/CD environment

4 vcpkg Package Management

Modern C++ dependency management system.

4.1 vcpkg Overview

What is vcpkg?

vcpkg is Microsoft's C++ package manager:

- Cross-platform C++ library management
- Integrates with CMake and Visual Studio
- Source-based builds for reliability
- Supports versioning and feature selection

Benefits for our project:

- Consistent dependency versions across team
- Easy library integration in future
- Reproducible builds
- Professional development workflow

4.2 vcpkg Installation

vcpkg Setup Process

```
# Clone vcpkg (recommended: adjacent to project)
cd ..  # Go to parent directory
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg

# Bootstrap vcpkg
./bootstrap-vcpkg.sh    # Linux/macOS
# .\bootstrap-vcpkg.bat   # Windows

# Set environment variable
export VCPKG_ROOT=$(pwd)

# Add to shell profile for persistence
echo 'export VCPKG_ROOT="'$(pwd)'"' >> ~/.bashrc
# OR for zsh users:
echo 'export VCPKG_ROOT="'$(pwd)'"' >> ~/.zshrc

# Integrate with build systems
./vcpkg integrate install
```

4.3 vcpkg Project Configuration

Project Integration

The project includes `vcpkg.json` manifest file:
Current configuration:
- No dependencies yet (standard library only)
- Ready for future library additions
- Features defined for testing and benchmarking
- Baseline pinned for reproducible builds

To add dependencies in future:
- Edit `vcpkg.json` dependencies array
- Run CMake configure to install packages
- Use `find_package()` in CMakeLists.txt

4.4 vcpkg Usage Examples

Common vcpkg Operations

```
# Search for packages
./vcpkg search json         # Find JSON libraries
./vcpkg search testing      # Find testing frameworks

# Install packages manually (for experimentation)
./vcpkg install fmt         # Fast formatting library
./vcpkg install catch2      # Testing framework

# List installed packages
./vcpkg list

# Build with vcpkg-managed dependencies
cmake --preset vcpkg-release
cmake --build build/vcpkg-release
```

5 Build System Configuration

Modern CMake-based build system setup and usage.

5.1 CMake Build System Overview

CMake Advantages

Why CMake over Makefile:

- True cross-platform support
- Better IDE integration and IntelliSense
- Modern dependency management
- Built-in testing framework (CTest)
- Industry standard build system
- Easier maintenance and scaling

Backward compatibility:

- Original Makefile still supported
- Gradual migration path available
- Same functionality with better implementation

5.2 Build Presets

Preset	Description
default	Basic build using Ninja generator
debug	Debug build with debugging symbols
release	Optimized release build (Linux)
vcpkg	Build with vcpkg dependency management
vcpkg-debug	Debug build with vcpkg
vcpkg-release	Release build with vcpkg
windows-mingw	Cross-compile for Windows using MinGW

Table 2: Available CMake Build Presets

5.3 Build Commands

CMake Build Workflow

```
# Configure builds (choose appropriate preset)
cmake --preset debug            # Debug configuration
cmake --preset release          # Release configuration (Linux)
cmake --preset windows-mingw    # Windows cross-compilation

# Build project
cmake --build build/release
cmake --build build/debug
cmake --build build/windows-mingw

# Clean builds
cmake --build build/release --target clean
```

5.4 Testing Integration

Testing with CTest

```
# Run all tests
ctest --test-dir build/release --output-on-failure

# Run specific test types
ctest --test-dir build/release -R "Compression"
ctest --test-dir build/release -R "Integration"

# Run tests with verbose output
ctest --test-dir build/release --verbose

# Custom test targets
cmake --build build/release --target test-all
cmake --build build/release --target run-case1
cmake --build build/release --target run-case2
```

6 IDE Configuration

Setting up popular IDEs for optimal C++ development experience.

6.1 Visual Studio Code (Recommended)

6.1.1 Required Extensions

VSCode Extension Setup

Essential extensions:
- **C/C++ Extension Pack** – Microsoft's official C++ support
- **CMake Tools** – CMake integration and IntelliSense
- **clangd** – Language server (recommended over C/C++)

Installation:
- Open VSCode Extensions (Ctrl+Shift+X)
- Search and install each extension
- Reload VSCode after installation

6.1.2 Workspace Configuration

Automatic Configuration

The setup script creates `.vscode/settings.json` with:

CMake integration:
- Configure on open enabled
- Ninja generator preference
- Build directory configuration

IntelliSense:
- clangd configuration with compile commands
- Header file associations
- Include path resolution

Code quality:
- Format on save enabled
- clang-format integration
- C++ standard configuration

6.1.3 Manual VSCode Setup

Manual VSCode Configuration

```
# Open project in VSCode
code .

# Generate compile commands for IntelliSense
cmake --preset release -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cp build/release/compile_commands.json .

# Configure CMake Tools extension
# 1. Open Command Palette (Ctrl+Shift+P)
# 2. Run "CMake: Select a Kit"
# 3. Choose your preferred compiler
# 4. Run "CMake: Select Variant" -> Release
```

6.2 CLion

6.2.1 Project Import

CLion Setup Process

Import CMake project:

- File → Open → Select `CMakeLists.txt`
- Choose "Open as Project"
- CLion will automatically configure CMake

Toolchain configuration:

- File → Settings → Build → Toolchains
- Verify CMake and compiler paths
- Configure vcpkg toolchain if using

Code style:

- File → Settings → Editor → Code Style → C/C++
- Scheme → Import → Select project `.clang-format`
- Enable "Format code on save"

6.3 Vim/Neovim

6.3.1 LSP Configuration

Vim/Neovim Setup

```
-- For Neovim with nvim-lspconfig
require'lspconfig'.clangd.setup{
    cmd = {"clangd", "--compile-commands-dir=build"},
    filetypes = {"c", "cpp", "objc", "objcpp"},
    root_dir = require'lspconfig.util'.root_pattern(
        '.clangd',
        '.clang-tidy',
        '.clang-format',
        'compile_commands.json',
        'compile_flags.txt',
        'configure.ac',
        '.git'
    ),
}
```

Vim/Neovim Plugins

Recommended plugins:

- **nvim-lspconfig** – LSP configuration
- **nvim-cmp** – Autocompletion
- **telescope.nvim** – Fuzzy finder
- **nvim-treesitter** – Syntax highlighting

7 Verification and Testing

Ensuring your development environment is properly configured.

7.1 Environment Verification

Complete Environment Test

```
# 1. Verify all tools are installed
cmake --version     # Should be 3.16+
ninja --version     # Any recent version
clang-format --version  # Any recent version

# 2. Test CMake configuration
cmake --preset release

# 3. Test build process
cmake --build build/release

# 4. Test executable creation
ls -la build/release/block_model  # Should exist

# 5. Test with sample data
cmake --build build/release --target run-case1

# 6. Run all tests
ctest --test-dir build/release --output-on-failure

# 7. Test cross-compilation (if on Linux)
cmake --preset windows-mingw
cmake --build build/windows-mingw
ls -la build/windows-mingw/block_model.exe  # Should exist
```

7.2 Troubleshooting

7.2.1 Common Issues

Build Environment Problems

Problem: CMake not found

- Install CMake 3.16+ from official website
- Add to system PATH environment variable
- Restart terminal/IDE after installation

Problem: Ninja not found

- Install: `sudo apt install ninja-build` (Ubuntu)
- Alternative: Use Make: `cmake -G "Unix Makefiles"`
- Windows: Install via Visual Studio or Chocolatey

Problem: vcpkg integration fails

- Ensure `VCPKG_ROOT` environment variable is set
- Run: `$VCPKG_ROOT/vcpkg integrate install`
- Restart terminal to pick up environment changes

7.2.2 IDE Issues

IDE Configuration Problems

Problem: IntelliSense not working

- Generate: `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
- Copy: `cp build/compile_commands.json .`
- Reload IDE/window

Problem: Headers not found

- Verify `include/` directory exists
- Check CMakeLists.txt target_include_directories
- Ensure compile commands are up to date

Problem: Build configuration errors

- Delete build directory: `rm -rf build`
- Reconfigure: `cmake --preset release`
- Check for conflicting IDE configurations

7.3 Performance Optimization

Build Performance Tips

Faster builds:

- Use Ninja generator (default in presets)
- Enable parallel builds: `cmake --build build --parallel $(nproc)`
- Use ccache: `export CMAKE_CXX_COMPILER_LAUNCHER=ccache`
- Incremental builds: only rebuild changed files

Development workflow optimization:

- Use IDE's built-in build commands
- Configure editor to format on save
- Set up automatic test running on file changes
- Use build caching for faster CI/CD

8 Next Steps

8.1 After Environment Setup

Ready to Develop!

You're now ready to:

Build the project with modern CMake

Run comprehensive tests

Cross-compile for Windows submission

Use professional development tools

Follow consistent coding standards

Next recommended reading:

- `docs/coding-standards.tex` – Code style and quality guidelines
- `README.md` – Daily development command reference
- `docs/Git & Github Workflow.pdf` – Version control workflow