FileLoader Module - Programmer's Guide

Introduction

The **FileLoader** module is a comprehensive, cross-platform file handling system designed for UltraCanvas applications. Its primary purpose is to provide unified file loading, saving, and format conversion capabilities across all supported file types including images, audio, video, documents, and 3D models. The module abstracts away platform-specific file handling complexities while offering powerful format conversion and built-in security scanning.

Key Benefits:

- **Universal Format Support** - Load and save files in any supported format
- **Cross-Format Conversion** - Convert between compatible file types seamlessly
- **Security Integration** - Built-in virus and malware scanning
- **Plugin Architecture** - Extensible system for adding new format support
- **Performance Optimized** - Caching, memory management, and GPU acceleration
- **Platform Agnostic** - Same API works on Windows, Linux, macOS

Function Reference Table

Function Category	Function Name	Description
Core Loading	`LoadFile`	Load any supported file format
Core Saving	`SaveFile`	Save data in specified format
Format Conversion	`ConvertFile`	Convert between compatible formats
Format Detection	`DetectFormat`	Identify file format from path/data
Security	`ScanFileForThreats`	Scan loaded files for malware
Plugin Management	`RegisterPlugin`	Register new format plugin
Cache Control	`SetCacheSize`	Configure memory cache limits
Batch Operations	`LoadMultipleFiles`	Load multiple files efficiently
Stream Support	`LoadFromStream`	Load from memory streams
Metadata	`GetFileMetadata`	Extract file metadata and properties

User Functions (Public API)

Core File Operations

`LoadFile`

Purpose: Load a file of any supported format into memory
Signature: `FileLoadResult LoadFile(const std::string& filePath, FileLoadOptions options = {})`

Parameters:

- `filePath` - Path to the file to load
- `options` - Optional loading configuration (security scanning, caching, etc.)

Returns: `FileLoadResult` containing success status, loaded data, metadata, and error information

Usage:

```cpp
// Basic file loading
FileLoadResult result = FileLoader::LoadFile("photo.jpg");
if (result.IsSuccess()) {
    ImageData* imageData = result.GetImageData();
    // Use loaded image data
}

// Advanced loading with options
FileLoadOptions options;
options.enableSecurity = true;
options.useCache = true;
options.maxResolution = Size(1920, 1080);
FileLoadResult result = FileLoader::LoadFile("document.pdf", options);
```

`SaveFile`

Purpose: Save data to a file in the specified format
Signature: `bool SaveFile(const std::string& filePath, const FileData& data, FileSaveOptions options = {})`

Parameters:

- `filePath` - Target file path with desired extension
- `data` - Data to save (ImageData, AudioData, VideoData, etc.)
- `options` - Save configuration (quality, compression, etc.)

Returns: `true` if saved successfully, `false` otherwise

Usage:

```cpp
// Save image with quality control
FileSaveOptions options;
options.quality = 90;
options.compressionLevel = 8;
bool success = FileLoader::SaveFile("output.webp", imageData, options);

// Save 3D model
FileLoader::SaveFile("model.obj", modelData);
```

`ConvertFile`

Purpose: Convert a file from one format to another

Signature: `bool ConvertFile(const std::string& inputPath, const std::string& outputPath, FileConvertOptions options = {})`

Parameters:

- `inputPath` - Source file path
- `outputPath` - Target file path with desired format extension
- `options` - Conversion settings (quality, resize, etc.)

Returns: `true` if conversion successful, `false` otherwise

Usage:

```cpp
// Convert image formats
FileLoader::ConvertFile("photo.png", "photo.webp");

// Convert with resize
FileConvertOptions options;
options.targetSize = Size(800, 600);
options.maintainAspectRatio = true;
options.quality = 85;
FileLoader::ConvertFile("large_photo.jpg", "small_photo.avif", options);
```

Format Detection & Information

`DetectFormat`

Purpose: Identify file format from file path or data

Signature: `FileFormat DetectFormat(const std::string& filePath)`

Usage:

```cpp
FileFormat format = FileLoader::DetectFormat("unknown_file.dat");
if (format == FileFormat::PNG) {
    // Handle as PNG image
}
```

GetFileMetadata

Purpose: Extract comprehensive metadata from loaded file

Signature: `FileMetadata GetFileMetadata(const std::string& filePath)`

Returns: Metadata including dimensions, creation date, EXIF data, audio properties, etc.

Usage:

```cpp
FileMetadata meta = FileLoader::GetFileMetadata("photo.jpg");
std::cout << "Resolution: " << meta.width << "x" << meta.height << std::endl;
std::cout << "Creation Date: " << meta.creationDate << std::endl;
if (meta.hasExif) {
    std::cout << "Camera: " << meta.exif.camera << std::endl;
}
```

Security Functions

ScanFileForThreats

Purpose: Scan file content for viruses and malware

Signature: `SecurityScanResult ScanFileForThreats(const std::string& filePath)`

Returns: Scan result with threat detection status and details

Usage:

```cpp
SecurityScanResult scan = FileLoader::ScanFileForThreats("suspicious.exe");
if (scan.isThreatDetected) {
    std::cout << "Threat found: " << scan.threatType << std::endl;
    // Handle security threat appropriately
}
```

Batch Operations

LoadMultipleFiles

Purpose: Efficiently load multiple files with parallel processing

Signature: `std::vector<FileLoadResult> LoadMultipleFiles(const std::vector<std::string>& filePaths, FileLoadOptions options = {})`

Usage:

```cpp
std::vector<std::string> files = {"img1.png", "img2.jpg", "img3.webp"};
auto results = FileLoader::LoadMultipleFiles(files);
for (const auto& result : results) {
    if (result.IsSuccess()) {
        // Process loaded file
    }
}
```

Stream Support

LoadFromStream

Purpose: Load file data from memory stream or buffer

Signature: `FileLoadResult LoadFromStream(const uint8_t* data, size_t size, FileFormat format, FileLoadOptions options = {})`

Usage:

```cpp
// Load from memory buffer
std::vector<uint8_t> fileData = GetDataFromNetwork();
FileLoadResult result = FileLoader::LoadFromStream(
    fileData.data(), fileData.size(), FileFormat::PNG
);
```

Module Support Functions (Internal API)

Plugin Management

RegisterPlugin

Purpose: Register a new format handler plugin
Signature: `bool RegisterPlugin(std::shared_ptr<IFileLoaderPlugin> plugin)`

Usage:

```cpp
// Register custom format plugin
auto customPlugin = std::make_shared<MyCustomFormatPlugin>();
FileLoader::RegisterPlugin(customPlugin);
```

UnregisterPlugin

Purpose: Remove a plugin from the system
Signature: `void UnregisterPlugin(const std::string& pluginName)`

GetRegisteredPlugins

Purpose: Get list of all registered plugins
Signature: `std::vector<std::shared_ptr<IFileLoaderPlugin>> GetRegisteredPlugins()`

Cache Management

SetCacheSize

Purpose: Configure maximum memory cache size
Signature: `void SetCacheSize(size_t maxSizeBytes)`

ClearCache

Purpose: Clear all cached file data
Signature: `void ClearCache()`

GetCacheStats

Purpose: Get current cache usage statistics
Signature: `CacheStats GetCacheStats()`

Resource Management

InitializeModule

Purpose: Initialize the FileLoader module and load plugins
Signature: `bool InitializeModule(const FileLoaderConfig& config = {})`

ShutdownModule

Purpose: Clean shutdown of module and release resources
Signature: `void ShutdownModule()`

SetTemporaryDirectory

Purpose: Configure temporary file location for conversions
Signature: `void SetTemporaryDirectory(const std::string& tempPath)`

Programming Examples

Basic File Loading Example

cpp

```cpp
#include "FileLoader.h"

int main() {
    // Initialize FileLoader
    FileLoader::InitializeModule();

    // Load an image
    FileLoadResult result = FileLoader::LoadFile("vacation_photo.heic");

    if (result.IsSuccess()) {
        ImageData* image = result.GetImageData();
        std::cout << "Loaded image: " << image->width << "x" << image->height << std::endl;

        // Use image data in your application
        DisplayImage(image);

        // Convert and save in different format
        FileLoader::SaveFile("vacation_photo.webp", *image);
    } else {
        std::cout << "Failed to load: " << result.GetErrorMessage() << std::endl;
    }

    // Cleanup
    FileLoader::ShutdownModule();
    return 0;
}
```

Advanced Format Conversion Example

cpp

```cpp
// Batch convert photos with resizing and quality control
std::vector<std::string> inputFiles = {
    "raw_photos/IMG_001.ARW",
    "raw_photos/IMG_002.ARW",
    "raw_photos/IMG_003.ARW"
};

FileConvertOptions convertOptions;
convertOptions.targetFormat = FileFormat::JPEG_XL;
convertOptions.quality = 92;
convertOptions.targetSize = Size(2048, 1536);
convertOptions.maintainAspectRatio = true;
convertOptions.enableGPUAcceleration = true;

for (size_t i = 0; i < inputFiles.size(); ++i) {
    std::string outputPath = "web_photos/photo_" + std::to_string(i+1) + ".jxl";

    bool success = FileLoader::ConvertFile(inputFiles[i], outputPath, convertOptions);
    if (success) {
        std::cout << "Converted: " << outputPath << std::endl;
    } else {
        std::cout << "Failed: " << inputFiles[i] << std::endl;
    }
}
```

Security-Aware File Processing Example

cpp

```cpp
// Process user-uploaded files with security scanning
void ProcessUploadedFile(const std::string& filePath) {
    // Always scan untrusted files first
    SecurityScanResult scan = FileLoader::ScanFileForThreats(filePath);

    if (scan.isThreatDetected) {
        std::cout << "Security threat detected: " << scan.threatType << std::endl;
        std::cout << "Description: " << scan.description << std::endl;

        // Delete suspicious file and alert user
        std::filesystem::remove(filePath);
        ShowSecurityAlert("Malicious file detected and removed");
        return;
    }

    // File is clean - proceed with loading
    FileLoadOptions options;
    options.enableSecurity = true;  // Continue monitoring during load

    FileLoadResult result = FileLoader::LoadFile(filePath, options);
    if (result.IsSuccess()) {
        // Process clean file data
        ProcessFileData(result);
    }
}
```

Plugin Development Example

cpp

```cpp
// Custom plugin for proprietary format
class MyCADFormatPlugin : public IFileLoaderPlugin {
public:
    std::string GetPluginName() const override {
        return "MyCAD Format Plugin";
    }

    std::vector<std::string> GetSupportedExtensions() const override {
        return {"mycad", "mcd"};
    }

    bool CanHandle(const std::string& filePath) const override {
        std::string ext = GetFileExtension(filePath);
        return ext == "mycad" || ext == "mcd";
    }

    FileLoadResult LoadFromFile(const std::string& filePath, FileLoadOptions options = {}) override {
        FileLoadResult result;

        // Implement custom loading logic
        std::ifstream file(filePath, std::ios::binary);
        if (!file.is_open()) {
            result.SetError("Cannot open file");
            return result;
        }

        // Parse your custom format
        ModelData* modelData = ParseMyCADFormat(file);
        if (modelData) {
            result.SetModelData(modelData);
            result.SetSuccess(true);
        } else {
            result.SetError("Invalid MyCAD file format");
        }

        return result;
    }

    bool SaveToFile(const std::string& filePath, const FileData& data, FileSaveOptions options = {}) override {
        // Implement custom saving logic
        const ModelData* model = data.GetModelData();
        if (!model) return false;

        std::ofstream file(filePath, std::ios::binary);
```

```
        std::ofstream file(filePath, std::ios::binary);
        return WriteMyCADFormat(file, *model);
    }
};

// Register your plugin
void InitializeMyCADSupport() {
    auto plugin = std::make_shared<MyCADFormatPlugin>();
    FileLoader::RegisterPlugin(plugin);
}
```

Stream Processing Example

cpp

```
// Load image from network without saving to disk
void LoadImageFromNetwork(const std::string& imageUrl) {
    // Download image data
    NetworkResponse response = HTTPClient::Get(imageUrl);
    if (response.IsSuccess()) {

        // Detect format from data
        FileFormat format = FileLoader::DetectFormatFromData(
            response.data.data(), response.data.size()
        );

        // Load directly from memory
        FileLoadResult result = FileLoader::LoadFromStream(
            response.data.data(),
            response.data.size(),
            format
        );

        if (result.IsSuccess()) {
            ImageData* image = result.GetImageData();
            DisplayImageInUI(image);
        }
    }
}
```

Comprehensive File Manager Example

cpp

```cpp
class MediaFileManager {
private:
    std::string mediaDirectory;
    std::vector<FileLoadResult> loadedFiles;

public:
    MediaFileManager(const std::string& directory) : mediaDirectory(directory) {
        // Configure FileLoader
        FileLoaderConfig config;
        config.maxCacheSize = Size::Megabytes(500);
        config.enableBackgroundScanning = true;
        config.securityLevel = SecurityLevel::High;
        FileLoader::Configure(config);
    }

    void LoadAllMediaFiles() {
        // Get all files in directory
        std::vector<std::string> allFiles = GetFilesInDirectory(mediaDirectory);

        // Filter supported formats
        std::vector<std::string> supportedFiles;
        for (const auto& file : allFiles) {
            if (FileLoader::IsFormatSupported(file)) {
                supportedFiles.push_back(file);
            }
        }

        // Batch load with progress tracking
        FileLoadOptions options;
        options.enableSecurity = true;
        options.generateThumbnails = true;

        loadedFiles = FileLoader::LoadMultipleFiles(supportedFiles, options);

        // Report results
        int successCount = 0;
        for (const auto& result : loadedFiles) {
            if (result.IsSuccess()) {
                successCount++;
            } else {
                std::cout << "Failed to load: " << result.GetFilePath()
                          << " - " << result.GetErrorMessage() << std::endl;
            }
        }
```

```
        }

        std::cout << "Loaded " << successCount << " of "
                  << supportedFiles.size() << " files" << std::endl;
    }

    void ConvertAllToWebFormat() {
        FileConvertOptions webOptions;
        webOptions.targetFormat = FileFormat::WEBP;
        webOptions.quality = 85;
        webOptions.targetSize = Size(1200, 800);
        webOptions.maintainAspectRatio = true;

        for (const auto& result : loadedFiles) {
            if (result.IsSuccess() && result.GetDataType() == FileDataType::Image) {
                std::string outputPath = ChangeFileExtension(result.GetFilePath(), ".webp");
                FileLoader::ConvertFile(result.GetFilePath(), outputPath, webOptions);
            }
        }
    }

    void GenerateReport() {
        FileLoaderStats stats = FileLoader::GetStatistics();
        std::cout << "=== FileLoader Statistics ===" << std::endl;
        std::cout << "Files loaded: " << stats.totalFilesLoaded << std::endl;
        std::cout << "Cache hits: " << stats.cacheHitRate << "%" << std::endl;
        std::cout << "Memory usage: " << stats.memoryUsage << " bytes" << std::endl;
        std::cout << "Threats detected: " << stats.threatsDetected << std::endl;
    }
};
```

Error Handling Best Practices

cpp

```cpp
// Comprehensive error handling example
void RobustFileProcessing(const std::string& filePath) {
    try {
        // Validate file exists
        if (!std::filesystem::exists(filePath)) {
            throw FileLoaderException("File does not exist: " + filePath);
        }

        // Check file size limits
        auto fileSize = std::filesystem::file_size(filePath);
        if (fileSize > FileLoader::GetMaxFileSize()) {
            throw FileLoaderException("File too large: " + std::to_string(fileSize) + " bytes");
        }

        // Attempt to load
        FileLoadResult result = FileLoader::LoadFile(filePath);

        if (!result.IsSuccess()) {
            // Handle specific error types
            FileLoaderError error = result.GetError();
            switch (error.type) {
                case FileLoaderErrorType::UnsupportedFormat:
                    std::cout << "Format not supported: " << error.details << std::endl;
                    break;

                case FileLoaderErrorType::CorruptedFile:
                    std::cout << "File appears corrupted: " << error.details << std::endl;
                    break;

                case FileLoaderErrorType::SecurityThreat:
                    std::cout << "Security threat detected: " << error.details << std::endl;
                    // Take appropriate security measures
                    break;

                case FileLoaderErrorType::InsufficientMemory:
                    std::cout << "Not enough memory to load file" << std::endl;
                    // Try reducing cache or splitting operation
                    break;

                default:
                    std::cout << "Unknown error: " << error.details << std::endl;
            }
            return;
        }
```

```
        }

        // Successfully loaded - process data
        ProcessLoadedFile(result);

    } catch (const FileLoaderException& ex) {
        std::cout << "FileLoader exception: " << ex.what() << std::endl;
    } catch (const std::exception& ex) {
        std::cout << "General exception: " << ex.what() << std::endl;
    }
}
```

Configuration & Initialization

Module Setup

```
cpp

// Initialize FileLoader with custom configuration
FileLoaderConfig config;
config.maxCacheSize = Size::Gigabytes(2);
config.temporaryDirectory = "/tmp/ultracanvas_fileloader";
config.enableBackgroundScanning = true;
config.securityLevel = SecurityLevel::High;
config.maxConcurrentOperations = 4;

bool success = FileLoader::InitializeModule(config);
if (!success) {
    std::cerr << "Failed to initialize FileLoader module" << std::endl;
}

// Register additional plugins
FileLoader::RegisterPlugin(std::make_shared<MyCustomPlugin>());
```

Performance Tuning

```cpp
// Configure for high-performance batch operations
FileLoader::SetCacheSize(Size::Gigabytes(4));
FileLoader::SetMaxConcurrentOperations(8);
FileLoader::EnableGPUAcceleration(true);

// Configure for memory-constrained environment
FileLoader::SetCacheSize(Size::Megabytes(50));
FileLoader::SetMaxConcurrentOperations(2);
FileLoader::EnableGPUAcceleration(false);
```

Integration with UltraCanvas

The FileLoader module integrates seamlessly with UltraCanvas components:

```cpp
// Load and display image in UltraCanvas
FileLoadResult result = FileLoader::LoadFile("logo.svg");
if (result.IsSuccess()) {
    auto imageElement = std::make_shared<UltraCanvasImageElement>();
    imageElement->SetImageData(result.GetImageData());
    canvas->AddElement(imageElement);
}

// Load 3D model for UltraCanvas 3D viewer
FileLoadResult modelResult = FileLoader::LoadFile("asset.fbx");
if (modelResult.IsSuccess()) {
    auto viewer3D = std::make_shared<UltraCanvas3DViewer>();
    viewer3D->LoadModel(modelResult.GetModelData());
    canvas->AddElement(viewer3D);
}
```

Best Practices

1. **Always initialize** the module before use: `FileLoader::InitializeModule()`
2. **Check return values** - all operations can fail gracefully
3. **Use security scanning** for untrusted files: `options.enableSecurity = true`
4. **Configure cache appropriately** for your application's memory constraints
5. **Handle format-specific errors** using the comprehensive error system
6. **Use batch operations** for multiple files to improve performance
7. **Register custom plugins** early in application startup
8. **Call shutdown** to properly release resources: `FileLoader::ShutdownModule()`

Supported File Formats

The FileLoader module supports extensive format coverage through its plugin system:

Images: PNG, JPEG, WEBP, AVIF, HEIC, GIF, BMP, TIFF, SVG, PSD, HDR, TGA
Audio: MP3, FLAC, WAV, OGG, AAC, M4A, OPUS
Video: MP4, AVI, MKV, WEBM, MOV, FLV
Documents: PDF, DOCX, RTF, TXT, MD
3D Models: OBJ, FBX, GLTF, STL, 3DS, PLY
Archives: ZIP, RAR, 7Z, TAR, GZ

Additional formats can be added through the plugin system without modifying core module code.