University of Brighton

Department of Computer Science

CI646 Programming languages, concurrency and client server computing

Technical report: Comparison of languages

Max Sherman

Module leader: Dr Almas Baimagambetov

A report submitted in partial fulfilment of the requirements of the University of Brighton for the degree of Bachelor of Science in *Computer Science*

09/05/2025

Contents

Chapter 1

Introduction

Software development depends heavily on programming languages that offer distinct features and paradigms that are beneficial for various applications. Understanding the differences between programming languages is crucial for determining the appropriate language for a given application. This report presents a comparative analysis of memory management in Java and C++, alongside a technical overview of concurrency, networking, and reflection. The analysis is contextualised through two open-source projects: FXGL, a Java/Kotlin-based game development framework, and XCube2D, a C++-based 2D game engine (Baimagambetov, 2025*a*, 2023). By exploring these projects, the report highlights both languages' strengths, limitations, and implementation in real-world applications.

1.1 Aims and Objectives

Aim: This technical report aims to provide a detailed analysis and comparison of two programming languages within the context of open-source projects. The report should demonstrate independent research and an in-depth understanding of language features, software architecture, and implementation techniques.

Objectives:

- Perform a technical comparison of Java/Kotlin and C++ by analysing their use in the FXGL and XCube2D projects.
- Examine memory management strategies employed in both languages and highlight key differences in their implementation across the two projects, providing code examples to support the discussion.
- Investigate concurrency, networking, and reflection tools within the FXGL project, evaluating available API's and their implementation.
- Provide code examples of the tools to support the technical evaluation.
- Conclude with insights gained from the comparison, summarising the key takeaways.

Chapter 2

Technical comparison

2.1 Memory management in FXGL (Java/Kotlin)

This section explores how Java/Kotlin, via the Java Virtual Machine (JVM), and the FXGL game library approach memory management. The JVM provides automatic memory handling through structured memory areas and garbage collection, removing manual management from the developer. However, its non-deterministic nature can lead to performance issues in real-time applications. FXGL addresses these challenges by introducing game-specific strategies such as asset caching, object pooling, and scene lifecycle management. These techniques complement the JVM's approach, enabling more predictable and efficient memory management.

2.1.1 JVM memory management

Understanding memory management in the Java Virtual Machine (JVM) provides essential context for analysing memory management in FXGL. Java's approach to memory management is focused on the automatic allocation and deallocation of objects. This task is handled by the JVM and consequently removes manual memory management responsibilities from the developer.

The JVM structures memory into several areas: the method area, heap, and stack. Each area has a distinct role in storing various components of a program during execution. The method area is a logical part of the heap and stores class-level information such as class structures, static variables, and interfaces. This area is created when the JVM is initiated and remains active for the duration of the program (GeeksforGeeks, 2025).

The heap is the primary memory area used for dynamic memory allocation. When objects and arrays are instantiated using the `new` keyword, they are stored in the heap, while references to these objects are stored in the stack. The heap is created alongside the JVM, and may increase or decrease in size while the application runs (Oracle, 2025). All threads have access to the heap, and its efficient management is critical for performance, especially in large-scale applications.

In contrast, the stack is thread-specific and is created when a new thread is started. It stores method call frames, local variables, parameters, and return addresses. Limiting each stack to a single thread enhances thread safety, an important consideration in concurrent systems (GeeksforGeeks, 2025).

Garbage Collection (GC) is a key feature of the JVM's memory management system. It operates in the background to remove any objects no longer reachable from any reference from the memory. This process is automatic, but developers can suggest a collection cycle using `System.gc()`. Despite this, the JVM remains in full control over the actual execution of GC. The system uses generational GC, which prioritises the collection of newer objects over older ones, improving overall memory management efficiency (Oracle, 2025).

Strengths	**Limitations**
Automated GC: Reduces manual memory management tasks and prevents errors such as memory leaks.	**Non-deterministic GC**: GC operations are not controlled by the application, but by the JVM. This unpredictability can lead to pauses in execution.
Improved developer productivity: Allows developers to focus on application logic rather than low-level memory concerns.	**GC pauses**: GC may occur during crucial moments, leading to unpredictable delays that impact performance, which is critical for fast-paced or latency-sensitive applications.
Generational GC: optimises memory by collecting newer objects more frequently than older ones.	**Performance overhead**: While GC improves memory management, it adds overhead, which can impact the performance of resource-intensive applications.
Memory safety: Stack-based memory management ensures thread isolation, preventing memory safety issues.	

Table 2.1: Strengths and limitations of memory management in Java/Kotlin

2.1.2 FXGL memory management

Understanding memory management in FXGL is crucial for analysing how it operates within the context of game development. While FXGL builds on the JVM's memory management, it utilises its own strategies to address the specific needs of game engines, where performance and responsiveness are critical. The non-deterministic nature of JVM GC can be problematic in real-time applications where predictable performance is essential (Romanazzi, 2018). FXGL manages memory in several areas, such as asset management, entity handling, and resource optimisation. The JVM's memory management features are complemented by FXGL's game-specific techniques, ensuring optimal memory usage during gameplay.

Asset management is one of the strategies employed in FXGL for managing game-related assets such as textures, sounds, and fonts. FXGL facilitates a lazy loading approach, where assets are only loaded when needed, preventing unnecessary memory consumption. To achieve this, FXGL introduces object pooling, which is a system where game entities are reused rather than recreated. This process reduces memory allocation overhead and minimises asset generation tasks. When an entity is no longer active, it is typically returned to a pool rather than destroyed, allowing for a faster entity retrieval without memory reallocation cost. The pooling logic can be found in the `pool` package, containing `Pool.java`, `Poolable.java`, `Pools.java`, and `ReflectionPool.java`.

To further optimise memory use, FXGL uses an asset cache for frequently accessed assets, avoiding repeated disk access and improving performance. The asset loader will check if the asset is present in the cache before loading it from the file system. If it is present, the loading task will be skipped, providing a more efficient asset retrieval approach. If the asset is not yet present in the cache, the asset loader will add it to the cache for future retrieval. Additionally, developers can clear the cache after use by calling `clearCache` (Baimagambetov, 2025*a*).

```kotlin
/** Lines 527-558 of FXGLAssetLoaderService.kt */
private fun <T> load(assetType: AssetType, loadParams: LoadParams): T {
    val data = assetData[assetType] as AssetLoader<T>

    if (loadParams.url === NULL_URL) {
        log.warning("Failed to load $assetType")
        return data.getDummy()
    }

    val cacheKey = loadParams.cacheKey

    val asset = cachedAssets[cacheKey]
    if (asset != null) {
        return data.cast(asset) // load from cache
    }

    return try {
        log.debug("Loading from file system: ${loadParams.url}")

        val loaded = data.load(loadParams)

        if (loadParams.isCacheEnabled) {
            cachedAssets[cacheKey] = loaded as Any
        }

        data.cast(loaded as Any)
    } catch (e: Exception) {
        log.warning("Failed to load ${loadParams.url}", e)
        data.getDummy()
    }
}
```

Listing 2.1: Cache logic in FXGLAssetLoaderService.kt

FXGL also manages scene switching effectively. When transitioning between scenes, FXGL ensures that resources no longer needed in the new scene are unloaded, freeing up memory. This is especially important for managing the memory of large, complex game scenes, which can contain numerous scene-specific assets and objects. The `SceneService` class is responsible for handling transitions. When a scene is no longer required, the scene can be popped from the stack, and `SceneService` ensures that the scene's resources are cleaned up appropriately.

In summary, FXGL incorporates several memory management techniques that complement the JVM's automatic garbage collection. Through strategies like asset caching, entity pooling, and scene management, FXGL ensures that game performance remains optimal and predictable, even for large-scale, complex game applications.

Strengths	**Limitations**
Efficient asset management: Lazy loading, automatic unloading, and asset caching reduce the overall memory footprint.	**Limited automatic asset unloading**: Assets cached to improve performance are not automatically unloaded, which could lead to high memory usage if not managed properly.
Built-in object pooling: Reuses entities and objects, minimising allocation overhead and reducing unpredictable GC.	**Memory persistence**: Improperly detached non-visual data, such as listeners or large collections, can persist across scenes.

Continued on next page

Strengths	Limitations
Scene lifecycle management: Ensures proper cleanup of resources during scene transitions, preventing memory leaks.	**Lazy loading overhead**: Loading assets only when necessary can cause gameplay interruptions if assets are not preloaded.
GC-friendly design: Scene-based architecture promotes natural garbage collection by the JVM.	**Asset cache saturation**: Excessive asset caching can overload memory, requiring developers to actively manage cache limits.

Table 2.2: Strengths and limitations of memory management in FXGL

2.2 Memory management in xcude2d (C++)

This section explores how memory is managed in both C++ and within the XCube2D game engine. While C++ offers low-level control and modern abstractions like smart pointers and RAII, XCub2D implements a custom resource-oriented approach using SDL and manual deallocation. By discussing both approaches, we gain a clearer understanding and how language-level features and engine-specific design choices affect memory efficiency and developer responsibilities.

2.2.1 C++ memory management

Memory management in C++ is a crucial aspect of ensuring application efficiency and performance. Unlike Java, which relies on automatic garbage collection, C++ provides developers with fine-grained control over memory allocation and deallocation. This section provides an overview of how memory management works in C++, with a focus on stack and heap memory, smart pointers, memory leaks, RAII (Resource Acquisition Is Initialisation), and memory fragmentation.

In C++, the stack is used for automatic storage and typically holds local variables declared within functions. When a function is called, memory for its local variables is automatically allocated, and when the function returns, the memory is automatically freed. The lifetime of stack-allocated variables is tied to the scope of the function or block in which they are defined, making stack memory fast and efficient. However, the stack is limited in size, which means that allocating large structures or arrays may result in stack overflow if the memory exceeds the available space (Stroustrup, 2013).

For dynamic memory allocation, C++ uses the heap. Unlike stack memory, where memory management is automatic, heap memory requires explicit handling (ISO C++, 2024). The `new` operator is used to allocate memory on the heap for objects or arrays, while `delete` is used to free that memory when it is no longer needed. This approach allows for flexible memory usage that can persist beyond a function's scope but places the responsibilities of memory deallocation entirely on the developer. Failing to use `delete` properly can result in memory leaks, which can lead to performance degradation and application crashes, as the system runs out of available memory. Additionally, incorrect usage can cause dangling pointers or double deletions, causing inefficient memory usage. Heap memory is commonly used when the size of the memory required is not known at compile time, or when the memory needs to persist beyond the scope of a single function.

From C++ 11 onwards, smart pointers have been introduced to simplify memory management and reduce the risk of memory leaks. Smart pointers automatically manage the lifetime of heap-allocated objects by deallocating memory when it is no longer needed (Programiz, 2025). `std::unique_pte` provides exclusive ownership of an object and frees the associated memory when it goes out of scope. `std::shared_ptr` allows multiple owners of an object, and the object is only destroyed when the last shared pointer goes out of scope. `std::weak_ptr` provides non-owning references that can break circular references without prematurely destroying the object. These tools are implemented based on the RAII (Resource Acquisition Is Initialisation) principle, which connects resource management with object lifetimes, ensuring resources are released when the owning object is destroyed (cppreference.com, 2024). Smart pointers are a prime example of RAII in practice, as they automatically manage memory through their lifetime and ensure proper clean-up when they go out of scope.

A major concern in memory management in C++ is memory fragmentation, which can occur in both the heap and stack. External fragmentation occurs when free memory is divided into non-contiguous blocks, making it difficult to allocate large contiguous blocks of memory. Internal fragmentation occurs when the allocated memory exceeds the amount requested, leaving unused space within allocated blocks. Techniques such as memory pooling, where a large memory block is pre-allocated and divided into smaller chunks, can mitigate issues by reducing the frequency and cost of dynamic allocations (Stroustrup, 2013).

In conclusion, C++ offers a variety of techniques for managing memory, from manual management using `new` and `delete` to automatic management with smart pointers. While C++ offers a high level of control over memory management, it requires disciplined programming practices. Understanding the discussed techniques, along with best practices such as RAII and memory pooling, is essential for avoiding memory leaks, reducing fragmentation, and ensuring efficient memory usage in C++ applications.

Strengths	**Limitations**
Fine-grained control: Direct management of memory using `new` and `delete` allows for precise optimisation of resource usage in performance-critical applications.	**Error-prone manual management**: Developer must explicitly manage memory, introducing risks such as memory leaks, dangling pointers, and double deletion.
High efficiency: Manual allocation supports custom memory layouts that can minimise overhead and maximise runtime performance.	**Increased complexity**: Manual memory handling increases cognitive load, making code harder to read and maintain.
Modern smart pointers: `std::unique_ptr` and `std::shared_ptr` offer automatic memory management, significantly reducing memory leaks and dangling pointers.	**No garbage collection**: Unlike languages such as Java or Python, C++ lacks built-in garbage collection, placing full ownership and lifecycle responsibility on the developer.
Deterministic clean-up: RAII ensures that resources are released immediately when objects go out of scope, improving predictability and reliability.	**Legacy code issues**: Older C++ projects that rely heavily on raw pointers are harder to modernise and more susceptible to memory-related bugs.

Table 2.3: Strengths and limitations of memory management in C++

2.2.2 XCude2D memory management

The XCube2D game engine employs a custom memory management model centred around resource caching and explicit deallocation rather than the use of advanced allocator schemes or automated garbage collection. Built on top of the SDL framework, the engine assigns low-level memory operations to the SDL library while leveraging a structured resource management system. This design ensures predictable control over the lifetime of textures, fonts, sounds, and MP3 files while minimising redundant allocation during runtime.

XCube2D's main memory management system is located in the `ResourceManager` class, which acts as a singleton and handles all asset loading, caching, and freeing (Baimagambetov, 2023). Resources such as textures, font, sounds, and MP3 files are loaded through dedicated `load*()` functions, which return raw pointers to the relevant SDL structures. Loaded resources are automatically added to the corresponding caching maps, storing their file path and SDL structure. Notably, the `loadTextures()` does not automatically cache textures, but only returns the loaded texture. Developers are responsible for manually adding textures to the map to ensure texture caching. This approach ensures that assets are loaded only once, even if requested multiple times during execution.

Once loaded and cached, resources are accessed through getter methods that return the cached resource and file path. Using this caching mechanism provides quick access to resources whilst maintaining centralised control over ownership and lifecycle. However, the system does not implement reference counting or lifetime tracking. Resources will remain in memory until explicitly released by the engine. Deallocating is performed by the `freeResources()` function, which iterates through each map and frees the associated memory using the appropriate SDL function. Developers should call this function during the engine shutdown sequence, ensuring that all assets are appropriately cleaned.

In addition to resource memory management, the engine manages subsystem lifetimes through a central `XCube2Engine` class. Subsystems such as graphics, audio, event, and physics engines are initiated using `std::shared_ptr`. During engine termination, all resources are freed and subsystems are reset, excluding the audio and physics engines, which might be an oversight. While shared pointers offer a level of safety through reference counting, the overall system relies on the assumption that no subsystem will outlive the engine singleton. This design keeps object ownership centralised and consistent, although it lacks the fine-grained control of allocator-driven engines.

Overall, XCube2D offers a straightforward and practical memory management system, which is suitable for small to medium-scale games. Its manual lifecycle management and explicit deallocation provides developer with control and predictability. However, the absence of advanced memory techniques such as pooling or allocator-based engines limits scalability.

```
std::map<std::string, SDL_Texture *> ResourceManager::textures;
std::map<std::string, TTF_Font *> ResourceManager::fonts;
std::map<std::string, Mix_Chunk *> ResourceManager::sounds;
std::map<std::string, Mix_Music *> ResourceManager::mp3files;

SDL_Texture * ResourceManager::loadTexture(std::string file, SDL_Color
    trans) {
        /** Omitted: Texture retrieval logic*/
        return texture;
}

TTF_Font * ResourceManager::loadFont(std::string file, const int & pt) {
        /** Omitted: font retrieval and caching logic*/
        return font;
}

Mix_Chunk * ResourceManager::loadSound(std::string file) {
        /** Omitted: sound retrieval and caching logic*/
        return sound;
}

Mix_Music * ResourceManager::loadMP3(std::string file) {
```

```cpp
        /** Omitted: MP3 retrieval and caching logic*/
        return mp3;
}

void ResourceManager::freeResources() {
        /** Omitted: debugging logic*/
        for (auto pair : fonts) {
                if (pair.second) {
                        TTF_CloseFont(pair.second);
                }
        }
        for (auto pair : textures) {
                if (pair.second) {
                        SDL_DestroyTexture(pair.second);
                }
        }
        for (auto pair : sounds) {
                if (pair.second) {
                        Mix_FreeChunk(pair.second);
                }
        }
        for (auto pair : mp3files) {
                if (pair.second) {
                        Mix_FreeMusic(pair.second);
                }
        }
}

SDL_Texture * ResourceManager::getTexture(std::string fileName) {
        return textures[fileName];
}

TTF_Font * ResourceManager::getFont(std::string fileName) {
        return fonts[fileName];
}

Mix_Chunk * ResourceManager::getSound(std::string fileName) {
        return sounds[fileName];
}

Mix_Music * ResourceManager::getMP3(std::string fileName) {
        return mp3files[fileName];
}
```

Listing 2.2: Overview of ResourceManager.cpp in XCube2D

Strengths	**Limitations**
SDL memory management: Delegates low-level memory tasks to SDL, reducing complexity.	**Raw pointer usage**: SDL resources are managed with raw pointers, increasing the risk of leaks and unsafe deallocation.
Custom resource caching: Centralised caching via `ResourceManager` avoid redundant allocations and ensures efficient asset reuse.	**No automated lifetime tracking**: Cached resource remain in memory until explicitly freed.

Continued on next page

Strengths	Limitations
Explicit lifecycle control: Manual `load*()` and `freeResources` functions provide predictable and transparent memory management.	**Manual texture caching**: Developers must manually manage texture reuse, increasing the risk of errors.
Shared pointer use in subsystems: Core systems use `std::shared_ptr` for basic memory safety.	**Lack of advanced memory techniques**: No pooling or allocator use, limiting scalability.
Centralised shutdown routine: Ensures consistent clean-up of resources and subsystems.	**Incomplete subsystem clean-up**: Audio and physics subsystems are not fully reset on shutdown.

Table 2.4: Strengths and limitations of memory management in XCube2D

2.3 Comparative Analysis

FXGL and XCube2D adopt fundamentally different memory management paradigms, portraying the differences between managed and unmanaged languages. FXGL, built on Java and Kotlin, benefits from the JVM's automatic memory handling, which relies on GC to allocate and reclaim memory. This model abstracts away most of the complexity for developers, promoting memory safety and simplifying development. However, it includes non-deterministic GC pauses, which can affect runtime consistency in latency-sensitive scenarios. Recent benchmarking research has shown that cloud systems using GC languages exhibit significantly worse tail latencies and resource efficiency compared to their C++ or Rust equivalents, even under similar workloads (Liang et al., 2024).

XCube2D, written in modern C++, opts for manual memory management. Allocation and deallocation are explicitly controlled by the developer, either through raw pointers or smart pointers such as `std::shared_ptr`. This approach allows for deterministic performance, essential in real-time applications, however, the responsibility lies with the developers to ensure memory safety. Without coding discipline, issues such as memory leaks, dangling pointers, or double deletions can occur.

For asset handling, FXGL include a robust system that includes built-in caching, lazy loading, and object pooling. Assets are automatically reused when available, improving performance by minimising redundancy. In contrast, XCube2D provides a centralised resource manager, but unconventionally excludes SDL textures from automatic caching. Texture caching requires manual handling, increasing the risk of errors.

Scene and object lifecycle management further highlight the engine's differences. FXGL automates clean-up via its scene service, which handles object destruction when scenes transition. XCube2D offers no built-in scene lifecycle management, requiring developers to manually free all associated resources, however, the engine does offer engine shutdown logic.

Collectively, the performance difference can be significant. FXGL can experience occasional pauses due to GC, particularly if object pooling is not used effectively. In contrast, XCube2D's manual memory framework ensures consistent UI behaviour when managed correctly, but increases the responsibilities of the developer.

Feature	FXGL (Java/Kotlin)	XCube2D (C++)
Memory framework	Automatic through GC	Manual with smart pointers
Allocation/ Deallocation	JVM handles both	Developer managed
Asset caching	Built-in with fallback	partially manual through the resource manager
Scene lifecycle handling	Automated clean-up	No built-in support
Performance predictability	Affected by GC pauses	Deterministic if managed well
Developer effort	Low due to abstraction	High due to manual management
Leak risk	Low due to automatic management	High due to error-prone manual control

Table 2.5: Comparison of FXGL and XCube2D memory management features

2.4 Code examples

This section shows code examples of applications leveraging the memory management tools available in FXGL and XCube2D, providing crucial context for the discussed techniques (Baimagambetov, 2025*b*).

2.4.1 FXGL

```java
package examples;

import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.asset.AssetLoaderService;
import com.almasb.fxgl.core.asset.AssetType;
import com.almasb.fxgl.core.pool.Pools;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.scene.SubScene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.util.Duration;

public class MemoryManagementExample extends GameApplication {
    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(500);
        settings.setHeight(500);
        settings.setTitle("Memory Management Example");
    }

    @Override
    protected void initGame() {
        // Asset caching example
        AssetLoaderService assetLoader = FXGL.getAssetLoader();
        Image image1 = assetLoader.load(AssetType.IMAGE, "player.png");
        Image image2 = assetLoader.load(AssetType.IMAGE, "player.png");
        System.out.println(image1 == image2);
        FXGL.getAssetLoader().clearCache();

        // Object pooling example
        fireBullet(100, 100, 2);
        fireBullet(200, 200, 4);

        // Sub scene creation example
        FXGL.getGameTimer().runOnceAfter(() -> {
            SubScene customSubScene = new SubScene() {
                @Override
                public void onCreate() {
                    Text text = FXGL.getUIFactoryService().newText("Press
                        button to change scene");
                    text.setFill(Color.BLACK);
                    text.setTranslateX(125);
                    text.setTranslateY(50);

                    Button resetButton = new Button("Reset");
                    resetButton.setOnAction(e -> resetScene());
                    resetButton.setTranslateX(225);
                    resetButton.setTranslateY(400);

                    getRoot().getChildren().addAll(text, resetButton);
                }
            };
            FXGL.getSceneService().pushSubScene(customSubScene);
        }, Duration.seconds(0));
    }

    // Function for firing the poolable bullet and freeing the bullet
        after the duration specified
    static void fireBullet(double x, double y, int duration) {
        Bullet bullet = Pools.obtain(Bullet.class);
        bullet.fire(x, y);
        FXGL.getGameTimer().runOnceAfter(() -> {
            Pools.free(bullet);
            System.out.println("Bullet at " + x + ", " + y + " returned to
                pool.");
        }, Duration.seconds(duration));
    }

    // Function for clearing the scene and creating a new scene
    private void resetScene() {
        FXGL.getGameWorld().getEntities().forEach(FXGL.getGameWorld()::
            removeEntities);
        FXGL.getGameScene().clearUINodes();

        while (FXGL.getSceneService().getCurrentScene() instanceof
            SubScene) {
            FXGL.getSceneService().popSubScene();
        }

        Text newText = FXGL.getUIFactoryService().newText("Scene reset
            complete!");
        newText.setFill(Color.BLACK);
        newText.setTranslateX(150);
        newText.setTranslateY(50);

        FXGL.getGameScene().addUINode(newText);
        System.out.println("Scene reset and cleaned up!");
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 2.3: Example of memory management in FXGL

```java
package examples;

import com.almasb.fxgl.core.pool.Poolable;

public class Bullet implements Poolable {

    private double x;
    private double y;

    public void fire(double x, double y) {
        this.x = x;
        this.y = y;
        System.out.println("Bullet fired at " + x + ", " + y);
    }

    @Override
    public void reset() {
        this.x = 0;
        this.y = 0;
    }
}
```

Listing 2.4: Poolable bullet class used in the FXGL memory management example

2.4.2 XCube2D

```cpp
#include "MyGame.h"

MyGame::MyGame() : AbstractGame() {
        // Resource loading and caching
        SDL_Color transparent = {255, 0, 255};
        ResourceManager::loadFont("res/fonts/arial.ttf", 72);
        ResourceManager::textures["res/textures/player.png"] =
            ResourceManager::loadTexture("res/textures/player.png",
            transparent);
        ResourceManager::loadSound("res/sounds/laser.wav");
        ResourceManager::loadMP3("res/mp3/music.mp3");

        // Get resources from cache, demonstrating cache retrieval
        TTF_Font * font = ResourceManager::getFont("res/fonts/arial.ttf");
        Mix_Chunk * sound = ResourceManager::getSound("res/sounds/laser.
            wav");
        Mix_Music * mp3 = ResourceManager::getMP3("res/mp3/music.mp3");

        // Use resources
        gfx->useFont(font);
        gfx->setVerticalSync(true);
        sfx->playSound(sound);
        sfx->playMP3(mp3, 10);
}

MyGame::~MyGame() {}

void MyGame::handleKeyEvents() {}

void MyGame::update() {}

void MyGame::render() {}

void MyGame::renderUI() {
        // Font resource example
        gfx->setDrawColor(SDL_COLOR_WHITE);
        std::string string = "Welcome to the game";
        gfx->drawText(string, 50, 50);

        // Texture resource example
        SDL_Rect* rect = new SDL_Rect{100, 100, 64, 64};
        gfx->drawTexture(ResourceManager::getTexture("res/textures/player.
            png"), rect, SDL_FLIP_NONE);

        // Free resources after use, commented out because resources are
            still in use (audio)
        // ResourceManager::freeResources();
}
```

Listing 2.5: Example of memory management in XCube2D through the resource manager

Chapter 3

Concurrency tools

3.1 Overview

FXGL provides a dedicated concurrency framework specifically designed for interactive, real-time game development. The concurrency tools are available in the `com.almasb.fxgl.core.concurrent` package and are designed to manage asynchronous tasks while maintaining the responsiveness and reliability of the JavaFX-based game loop (Baimagambetov, 2025*a*).

FXGL's main concurrency system is a centralised asynchronous executor service which is responsible for managing and scheduling asynchronous tasks on either the background or JavaFX thread. The `Async` singleton object implements the `Executor` interface, which extends Java's standard executor service (Oracle, 2012). The FXGL executor service supports delayed and immediate execution and is designed to prevent background operations from interfering with the game loop. This includes tasks that can remain scheduled independent of the game's state, offering granular control over when and how background operations are executed.

Additionally, the `AsyncTask<T>` abstract class enables blocking or awaiting task execution with synchronisation mechanisms such as `CountDownLatch`. This allows for safe and controlled retrieval of asynchronous results within game logic.

For I/O-related operations, FXGL provides an abstract task class named `IOTask<T>`, but this class can be used for any operation that can fail. The class supports failure handling, error handling, and task chaining. It is particularly suitable for operations such as resource loading and network communication, where robust error handling and cancellation support are critical.

Lastly, `AsyncService<T>` enables asynchronous processing integrated with the game loop by allowing background computations to run during the game loop. For the asynchronous processing thread and JavaFX thread, the service offers either synchronised or unsynchronised task execution through the `onPostGameUpdateAsync()` and `onGameUpdateAsync()` functions, respectively. This mechanism is crucial for heavy computational tasks, which must not interfere with key processes such as game rendering or input handling.

Collectively, these tools offer a comprehensive concurrency framework that enables developers to control background tasks safely and efficiently, without impacting game execution. This approach is fitting for game engines, as they require deterministic behaviour.

3.2 Strengths

FXGL provides a robust, integrated concurrency framework specifically designed for game development, ensuring responsive gameplay and thread safety. The framework manages asynchronous tasks and offers low-level control and high-level abstractions. The system supports flexible task scheduling, robust error handling, and safe integration, making it suitable for handling high complexity or I/O operations without affecting runtime performance.

Tool	**Responsibility**	**Strengths**
`Executor`	Interface abstraction for asynchronous task execution	Separation of concerns, decouples game logic from thread management, unified interface for concurrency
`Async`	Singleton class for asynchronous task management and scheduling	Thread safe, supports background and JavaFX thread, provides flexible delayed executions, includes shutdown function
`AsyncTask`	Asynchronous tasks that can block the current thread to wait for the result	Clean abstraction for synchronised result retrieval, enables synchronisation mechanisms such as `CountDownLatch`
`IOTask`	Abstract class for asynchronous I/O or failure-prone operations	Supports cancellation, failure, error handling, and task chaining
`AsyncService`	Asynchronous logic with game-loop integration	Executes heavy computational tasks, either synchronised or unsynchronised with the JavaFX thread.

Table 3.1: Responsibilities and strengths of concurrency tools in FXGL

3.3 Code example

```java
package examples;

import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.concurrent.IOTask;
import com.almasb.fxgl.dsl.FXGL;
import javafx.scene.text.Text;
import java.nio.file.Files;
import java.nio.file.Path;

public class ConcurrencyExample extends GameApplication {
    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(500);
        settings.setHeight(500);
        settings.setTitle("Concurrency Example");
    }

    @Override
    protected void initGame() {
        Text statusText = new Text("Loading...");
        statusText.setTranslateX(50);
        statusText.setTranslateY(50);
        FXGL.getGameScene().addUINode(statusText);

        // Start asynchronous task on background thread
        FXGL.getExecutor().startAsync(() -> {
            // Simulates background tasks such as loading a resource
            boolean result = loadResource();
            if (result) {
                statusText.setText("Concurrency loaded!");
            } else {
                statusText.setText("Failed to load!");
            }
        });

        // Can run shutdownNow to simulate resource loading failure
        //FXGL.getExecutor().shutdownNow();


        // Example IOTask
        IOTask<String> readFileTask = new IOTask<>() {
            @Override
            protected String onExecute() throws Exception {
                Path path = Path.of("data.txt");
                return Files.readString(path);
            }
        };

        // Success, failure, and cancellation handling
        readFileTask
                .onSuccess(content -> {statusText.setText("File content:\n
                    " + content);})
                .onFailure(e -> {statusText.setText("Failed to load file:
                    " + e.getMessage());})
                .onCancel(() -> {statusText.setText("Task was cancelled")
                    ;});
        // Run async task on JavaFX thread
        FXGL.getExecutor().startAsyncFX(readFileTask::run);
    }

    private boolean loadResource() {
        try {
            Thread.sleep(5000);
            return true;
        } catch (InterruptedException e) {
            return false;
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 3.1: Example of concurrency in FXGL

Chapter 4

Networking tools

4.1 Overview

FXGL offers a high-level, extendable networking framework tailored to real-time game development. The `com.almasb.fxgl.net` package offers clean abstractions for creating TCP and UDP clients and servers, designed to operate safely in a JavaFX environment. Networking functionalities can be retrieved through the `NetService` class, which provides a consistent, thread-safe API for managing multiplayer interactions. The system is designed to simplify common networking tasks while maintaining flexibility and performance through asynchronous operations and background threading (Baimagambetov, 2024, 2025*a*).

`NetService` is the central access point for TCP and UDP networking. It enables developers to create servers and clients with generic type-safe configurations. It also supports asynchronous file downloading and stream access through background tasks, ensuring that network operations do not affect the game loop. Servers and clients can be configured with `ServerConfig` or `ClientConfig` instances, allowing developers to specify the message types. The service uses FXGL's internal Bundle format by default, but can also support custom serialisable classes.

Additionally, the `HttpClientService` class in FXGL offers a high-level asynchronous interface for performing HTTP operations using Java's HttpClient, integrated with `IOTask` to ensure non-blocking execution. The service supports standard HTTP methods such as GET, PUT, POST, and DELETE, and provides multiple overloads for headers and body handlers. Internally, it uses lazy loading for the `HttpClient` instance and encapsulates each request in an `IOTask`, which runs on a background thread and returns `HttpResponse`. This design enables safe and efficient HTTP client networking, preserving responsiveness whilst simplifying HTTP communication.

`Server<T>` and `Client<T>` are abstract base classes used to implement TCP and UDP servers and clients. Both classes extend `Endpoint<T>`, which provides a shared interface for managing connection state and message routing. The `Server<T>` class offers asynchronous methods such as `startAsync()` and reveals connection state through observable properties. Similarly, the `Client<T>` provides methods such as `connectAsync()` to initiate connections on background threads. Both implementations are built using background threads to ensure concurrency does not interfere with key processes such as rendering or input processing.

`Endpoint<T>` is the base class for all network endpoints. It manages a collection of active `Connection<T>` instances and handles core tasks such as broadcasting messages and registering connection listeners. Developers can attach connection lifecycle callbacks (`setOnConnected`, `setOnDisconnected`) to respond to connection events. Internally, `Endpoints<T>` handles thread operations and supports both TCP and UDP communication through protocol-specific implementations.

Connection<T> represents an active link between a client and a server. It allows for message transmission using a thread-safe queue and supports registering multiple MessageHandler<T> instances for processing incoming messages. To maintain UI consistency, FXGL provides addMessageHandlerFX, which ensures that message callbacks run on the JavaFX thread. Each connection also includes a PropertyMap that stores session-specific metadata.

MessageHandler<T> is a functional interface that defines how messages are processed upon receipt. Handlers can be registered to run either on a background thread or the UI thread, depending on application requirements. This flexibility enables real-time message handling without compromising UI responsiveness or thread safety.

FXGL support both the Bundle class and custom message types for network transmission. The Bundle offers a key-value format ideal for rapid prototyping, while more complex applications can define structured message classes and configure clients or servers with the appropriate serialisable class.

In summary, FXGL's networking API provides a reliable and intuitive method for developing networked games. Its high-level abstraction, robust thread safety, integration with JavaFX, and support for both TCP and UDP make it a reliable solution for real-time multiplayer functionality.

4.2 Strengths

FXGL's networking framework offers a comprehensive set of tools for implementing real-time multiplayer functionality. Built around extendable abstractions for HTTP, TCP, and UDP communication, it simplifies client-server architecture whilst ensuring thread safety and consistent performance. Each tool addresses a distinct aspect of the networking framework, from connection lifecycle management to message handling and data transmission. The system's design prioritises clarity, responsiveness, and integration with JavaFX, allowing developers to build interactive, networked games without the complexity of low-level socket programming.

Tool	**Responsibility**	**Strengths**
NetService	Network service for creating TCP/UDP clients and servers, provides file downloading and stream access	High-level abstraction, seamless FXGL integration, supports both default and custom message types
HttpClientService	Handles asynchronous HTTP communication, uses HttpClient through IOTask	Non-blocking requests, built-in support for HTTP methods, seamless integration with FXGL's concurrency framework
Server<T>/ Client<T>	Classes for TCP/UDP servers and clients, manages connections and message transmission	Asynchronous connection handling, supports multiple concurrent clients, extends Endpoint<T> class
Endpoint<T>	Common superclass for clients and servers, manages active connections and broadcasting	Unified API, handles both TCP and UDP, supports lifecycle hooks

Continued on next page

Tool	Responsibility	Strengths
`Connection<T>`	Represents a single client-server connection, manages message queue and handlers	Thread-safe queueing and message dispatching, supports session metadata and JavaFX-thread message handling
`MessageHandler<T>`	Interface for handling received messages	Decouples message processing, enables safe message dispatching via background or JavaFX thread
`Bundle`/ Custom classes	Data formats used for message transmission, either default key-value or structured types	Encapsulates transmitted data, flexible serialisation, extendable via custom serialisable classes and `ClientConfig`/ `ServerConfig`

Table 4.1: Responsibilities and strengths of networking tools in FXGL

4.3 Code example

```java
package examples;

import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.serialization.Bundle;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.net.Server;
import javafx.scene.control.CheckBox;

public class NetworkingServerExample extends GameApplication {
    private Server<Bundle> server;
    private CheckBox cb;

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(500);
        settings.setHeight(500);
        settings.setTitle("Networking Example");
    }

    @Override
    protected void initUI() {
        // Add checkbox to game scene
        cb = new CheckBox();
        // Add listener which broadcast a bundle when the checkbox
            property changes
        cb.selectedProperty().addListener((observable, oldValue, newValue)
             -> {
            var bundle = new Bundle("CheckBoxData");
            bundle.put("isSelected", newValue);
            server.broadcast(bundle);
        });
        cb.setTranslateX(100);
        cb.setTranslateY(100);
        FXGL.getGameScene().addUINode(cb);

        // Create new TCP server
        server = FXGL.getNetService()
                .newTCPServer(50000);

        // Print message when connected or disconnected
        server.setOnConnected(connection -> {
            System.out.println("New client connected: " + connection);
        });
        server.setOnDisconnected(connection -> {
            System.out.println("Client disconnected: " + connection);
        });

        // Start server asynchronously
        server.startAsync();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 4.1: Example of networking server in FXGL

```java
package examples;

import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.serialization.Bundle;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.net.Client;
import javafx.scene.control.CheckBox;

public class NetworkingClientExample extends GameApplication {
    private Client<Bundle> client;
    private CheckBox cb;

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(500);
        settings.setHeight(500);
        settings.setTitle("Networking Example");
    }

    @Override
    protected void initUI() {
        // Add checkbox to game scene
        cb = new CheckBox();
        cb.setTranslateX(100);
        cb.setTranslateY(100);
        FXGL.getGameScene().addUINode(cb);

        // Create new TCP client
        client = FXGL.getNetService().newTCPClient("localhost", 50000);

        // Add message handler to connection
        client.setOnConnected(connection -> {
            System.out.println("Connected to server.");

            connection.addMessageHandlerFX((conn, bundle) -> {
                if (bundle.exists("isSelected")){
                    Boolean isSelected = bundle.get("isSelected");
```

```
                        cb.setSelected(isSelected);
                    }
                });
            });

            // Print message when disconnected
            client.setOnDisconnected(connection -> {
                System.out.println("Disconnected from server.");
            });

            // Connect client asynchronously
            client.connectAsync();
        }

        public static void main(String[] args) {
            launch(args);
        }
}
```

Listing 4.2: Example of networking client in FXGL

Chapter 5

Reflection tools

5.1 Overview

FXGL provides a lightweight reflection framework in the `com.almasb.fxgl.core.reflect` package. This framework is designed for use cases common in game development, such as runtime method invocation, dynamic type resolution, and object serialisation. The framework emphasises flexibility and scalability, enabling developers to perform reflective operations without the overhead typically associated with Java's standard reflection API (Baimagambetov, 2025*a*).

FXGL's reflection framework is built around the `ReflectionFunctionCaller` class. The Kotlin class enables runtime method invocation of any object using its method name and parameters as strings. It registers target objects and dynamically resolves and invokes methods, including private ones. The system automatically converts string parameters to matching Java types and supports overloaded methods by pairing method names with the data types and number of arguments. Developers can extend this mechanism with custom converters for additional data types. If no matching method is found, a default handler is triggered, ensuring graceful fallbacks during reflection. All method calls follow consistent return logic, using Elvis operators to handle void methods, which return `null`. To maintain compatibility with the unified `Any` return type, void method calls return default values (in this case, 0).

The `ReflectionUtils` class offers a variety of static methods for more granular reflection tasks. The class finds methods and fields based on annotations or type, offering a flexible approach to class identification. Developers can locate methods with specific annotations, find declared fields, and perform field injections regardless of access level. Additionally, the class also includes methods for calling private methods (`callInaccessible`), and creating new instances using Java's default no-argument constructor (`newInstance`). A particularly useful feature is the `getCallingClass()` method, which finds the class that invoked a specific method at runtime. It searches the stack trace for the method's caller and returns the class type.

Furthermore, FXGL offers reflection with native libraries via the `ForeignFunctionCaller` class. This class leverages the Java Foreign Function and Memory API (FFM API) to load and call functions from native libraries as if they were regular Java methods. The class loads libraries through a dedicated thread, using a thread-safe queue to avoid concurrency issues. The associated `ForeignFunctionContext` class facilitates memory allocation and native function calling, both with and without arguments. This wrapper offers an alternative to the Java Native Interface (JNI), using modern Java techniques to simplify foreign library integration.

To standardise error handling during reflection operations, FXGL defines a dedicated exception class, `ReflectionException`. This simplifies the process of identifying and debugging reflection-related issues by grouping low-level exceptions in a uniform structure with support for custom messages and causes.

Collectively, FXGL's reflection tools offer a lightweight solution for runtime tasks such as dynamic method invocation, field access, and native integration. The framework simplifies reflection by offering tools that reduce redundant code and handle common use cases efficiently. Integration with the Java FFM API offers developers the tools to extend game behaviour dynamically. Its straightforward design makes it a useful package for tasks that require access to methods and fields at runtime.

5.2 Strengths

FXGL's reflection framework provides a focused and efficient set of tools for handling runtime operations such as dynamic method invocation, field access, object creation, and native function integration. Each tool covers specific runtime needs without introducing unnecessary complexity. Together, these tools allow developers to extend game behaviour dynamically and interact with otherwise inaccessible structures efficiently, whilst maintaining safety and readability.

Tool	**Responsibility**	**Strengths**
`ReflectionFunction-Caller`	Dynamically call object methods by name and arguments, converts string inputs to typed arguments	Supports private and overloaded methods, extendable type conversion, consistent return types
`ReflectionUtils`	Locate and change fields and methods, call private methods, create objects reflectively	Broad functionality, supports annotation and type-based identification, proxy-based functional mapping
`ForeignFunction-Caller`	Load and call native functions via the Java FFM API, manages native memory	Provides modern alternative to JNI, ensures thread safety
`ReflectionException`	Encapsulates and standardises error handling for reflection-related operations	Simplifies debugging, ensures consistent exception handling

Table 5.1: Responsibilities and strengths of reflection tools in FXGL

5.3 Code example

```java
package examples;

import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.core.reflect.ReflectionFunctionCaller;
import javafx.scene.control.TextField;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import java.util.Arrays;

```

```java

public class ReflectionExample extends GameApplication {
    private final ReflectionFunctionCaller reflectionCaller = new
        ReflectionFunctionCaller();

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(500);
        settings.setHeight(500);
        settings.setTitle("Reflection Example");
    }

    @Override
    protected void initGame() {
        // Add rectangle to game scene
        FXGL.entityBuilder()
                .at(100, 100)
                .view(new Rectangle(50, 50, Color.BLACK))
                .buildAndAttach();
        // Set the score to 0
        FXGL.set("score", 0);
    }

    @Override
    protected void initUI() {
        // Retrieve game commands
        GameCommands commands = new GameCommands();
        reflectionCaller.addFunctionCallTarget(commands);

        // Add input box, in this example a debugger
        TextField inputBox = new TextField();
        inputBox.setTranslateX(50);
        inputBox.setTranslateY(450);
        inputBox.setPrefWidth(400);

        // Retrieve input when enter is pressed
        inputBox.setOnAction(e -> {
            // Input filtering
            String input = inputBox.getText().trim();
            String[] tokens = input.split(" ");
            String methodName = tokens[0];
            String[] args = Arrays.copyOfRange(tokens, 1, tokens.length);

            // Call the method through the reflection caller, push
                notification accordingly
            try {
                Object result = reflectionCaller.call(methodName, args);
                FXGL.getNotificationService().pushNotification("Executed "
                     + methodName + " -> " + result);
            } catch (Exception ex) {
                FXGL.getNotificationService().pushNotification("Error:" +
                    ex.getMessage());
            }

            inputBox.setText("");
        });
        FXGL.getGameScene().addUINode(inputBox);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 5.1: Example of reflection in FXGL

```java
package examples;

import com.almasb.fxgl.dsl.FXGL;

// Example game commands for reflection
public class GameCommands {
    public void teleportPlayerToStart() {
        System.out.println("Teleporting player to (0,0)");
        FXGL.getGameWorld()
                .getEntities()
                .forEach(e -> {
                    e.setPosition(0, 0);
                });
    }

    public void teleportPlayer(int x, int y) {
        System.out.println("Teleporting player to (" + x + ", " + y + ")")
            ;
        FXGL.getGameWorld()
                .getEntities()
                .forEach(e -> {
                    e.setPosition(x, y);
                });
    }

    public void printScore() {
        System.out.println("Current score: " + FXGL.geti("score"));
    }
}
```

Listing 5.2: Example game commands which can be called through reflection

Chapter 6

Conclusion

This report explored core technical areas of FXGL and XCube2D, with a focus on memory management, concurrency, networking, and reflection. A key takeaway is the contrast between manual and automatic memory management across languages. Whilst Java and Kotlin offer automatic garbage collection that reduces overhead, C++ requires explicit memory control, offering more granular control at the cost of increased cognitive load on the developer. The choice between these models should depend on application requirements, with memory management playing a crucial role in performance and predictability.

In researching concurrency tools, I gained a deeper understanding of thread management and the need for thread safety in real-time applications. Implementing and testing code examples also revealed how thread-related errors can occur unexpectedly, emphasising the importance of clearly defined thread boundaries and dedicated thread-safe methods.

The networking framework of FXGL shows the value of high-level abstractions in simplifying complex tasks. The API offers a clean and efficient interface that removes much of the typical overhead involved in socket programming, making it well-suited for rapid game development.

Through reflection research, I came to appreciate both its power and versatility. FXGL's reflection tools, especially when applied to dynamic method invocation and native function interfacing, revealed reflection's wide range of practical use cases in modern game engines.

This project also allows me to engage deeply with both Java and Kotlin. Although I started with no experience in Kotlin, I became comfortable with the language and its syntax through FXGL codebase exploration. Overall, I developed a better understanding of how modern game engines are structured and how to apply best practices during game development and engine design.

6.1 Estimated grade

Comparison: A, Concurrency: A, Networking: A, Reflection: A-

References

Baimagambetov, A. (2023), ‘xcube2d’. Accessed February 7, 2025.
URL: *https://github.com/AlmasB/xcube2d*

Baimagambetov, A. (2024), ‘Fxgl wiki’. Accessed February 14, 2025.
URL: *https://github.com/AlmasB/FXGL/wiki*

Baimagambetov, A. (2025*a*), ‘Fxgl game engine’. Accessed Febuary 7, 2025.
URL: *https://github.com/AlmasB/FXGL*

Baimagambetov, A. (2025*b*), ‘Youtube channel’. Accessed February 7, 2025.
URL: *https://www.youtube.com/channel/UCmjXvUa36DjqCJ1zktXVbUA*

cppreference.com (2024), ‘Resource acquisition is initialization (raii)’. Accessed February 21, 2025.
URL: *https://en.cppreference.com/w/cpp/language/raii*

GeeksforGeeks (2025), ‘Java memory management’. Accessed February 14, 2025.
URL: *https://www.geeksforgeeks.org/java-memory-management/*

ISO C++ (2024), ‘C++ core guidelines’. Accessed February 21, 2025.
URL: *https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines*

Liang, Z., Jabrayilov, V., Charapko, A. and Aghayev, A. (2024), ‘The cost of garbage collection for state machine replication’. Accessed March 28, 2025.
URL: *https://arxiv.org/pdf/2405.11182*

Oracle (2012), ‘Javafx threads’. Accessed March 14, 2025.
URL: *https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm*

Oracle (2025), ‘Garbage collection’. Accessed February 14, 2025.
URL: *https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html*

Programiz (2025), ‘C++ memory management’. Accessed February 21, 2025.
URL: *https://www.programiz.com/cpp-programming/memory-management*

Romanazzi, S. (2018), From manual memory management to garbage collection, Technical report, University of Bologna. Accessed March 28, 2025.
URL: *https://www.researchgate.net/publication/326369017_From_Manual_Memory_Management_to_Garbage_Collection*

Stroustrup, B. (2013), *The C++ Programming Language*, 4 edn, Addison-Wesley, Upper Saddle River, NJ.