



Translating a Game Engine

Improving 3D Games for the Android OS

Name: Ben Constable

Degree Programme: BSc Computer Science

School: Informatics

Candidate Number: 30903

Supervisor: Martin Berger

Year of Submission: 2011

Statement Of Originality

This report is submitted as part requirement for the degree of BSc Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: _____

Dated: _____

Sangwin Disclaimer

The original Sangwin Engine Java source code used and included with this submission is subject to the following disclaimer:

“The Sangwin Engine Java source code is for reference only, and is the property of Sangwin Games. It should not be distributed freely without the permission of Sangwin Games.”

Acknowledgements

I would like to thank the following people, for contributing to the success of this project.

Oli Winks: Oli is the programmer behind the original Sangwin Engine, and has helped me consistently throughout the project with his source code and the final translation.

Martin Berger: Martin constantly kept me on track with my project, and ensured that I chose interesting topics for the final dissertation.

Summary

This report discusses the development of 3D games for mobile devices with limited hardware capabilities, with a particular focus on choice of programming language. The Android OS is used as the platform for comparison. The OS uses Java as its main language, but also provides developers with the ability to program using native (C or C++) code.

A 3D Android game engine written in Java was provided by Sangwin, a small games company based at the University of Sussex. Translating the engine to C++ was a main aim of this project, which was successfully achieved. The resultant engine was built for desktop using the Nokia Qt framework, and for the Android OS. The Java Native Interface was used alongside the Android Native Development Kit to access the C++ code using the operating system.

The performance of both engines was profiled using a number of techniques, in order to establish the differences between Java and C++ when developing for mobile platforms. It was found that, despite the Android OS making use of Just In Time Java compilation, the C++ game engine was up to 3 times faster than the original Java engine.

The report also details the design changes required by the C++ translation. It discusses the added complexity of using native code with Android, and proposes a number of solutions to reduce this complexity. A useful and usable API was designed, which was complemented by an example game made using the engine, and extensive documentation.

All of the Primary Requirements for the project were met successfully, and all but one of the Secondary Requirements were also achieved. A number of new technologies were learnt throughout the project, including C++, OpenGL, game engine programming and Android OS programming. The project was both challenging and rewarding, and provided a number of insights into language choice for mobile platforms.

Table of Contents

1. Introduction	8
1.1 Description of the Project and Problem Area	9
1.2 Research and Technologies	10
1.2.1 Game Engines	10
1.2.2 C++ and Java Performance	11
1.2.3 Developing Natively for Android	11
1.2.4 The Java Native Interface	12
1.3 Professional Considerations	13
1.3.1 Code of Conduct	13
1.3.2 Code of Good Practice	13
2. Requirements Analysis	15
2.1 Understanding Requirements	16
2.1.1 Gathering Requirements	16
2.1.2 Software Engineering Approach	16
2.2 The Sangwin Java Engine	17
2.3 Requirements Specification	18
2.3.1 Primary Requirements	18
2.3.2 Secondary Requirements	19
3. System Design & Implementation	20
3.1 Using The Android OS	21
3.1.1 The Activity Lifecycle	21
3.1.2 Android 2.2 and 2.3	21
3.2 Managing Memory	23
3.2.1 Reference Counted Smart Pointers	23
3.2.2 Inheritance, Casting and Type Safety	23
3.2.3 Drawbacks	24
3.3 Native Android Limitations	25
3.3.1 The C++ Standard Template Library (STL)	25
3.3.2 Real Time Type Information (RTTI)	26
3.3.3 Exceptions	26
3.4 The Scenograph	27
3.4.1 What are Scenographs?	27
3.4.2 C++ Designs	27
3.4.3 The GetNode Class	29
3.5 Android Event Handling	30
3.5.1 C++ Designs	30

3.5.2 Concurrency	31
3.6 Android Resource Management	32
3.6.1 Loading Textures	32
3.6.2 Loading Models	33
4. Final Builds and Testing	34
4.1 Test Strategies	35
4.2 Low Level Unit Testing	36
4.3 Building and Testing the Desktop Engine	37
4.3.1 Desktop Implementation	37
4.3.2 Desktop Testing	37
4.3.3 Working Example	38
4.4 Building and Testing the Android Engine	39
4.4.1 Using the JNI	39
4.4.2 Android Testing	40
4.4.3 Implementation Technicalities	40
4.4.4 Working Example	40
4.5 An Example Game	41
5. Profiling	42
5.1 Profiling Methods	43
5.1.1 Profiled areas	43
5.1.2 Ensuring a Reliable Experiment	43
5.1.3 Android Traceview	44
5.1.4 Alternative	44
5.2 Results	46
5.2.1 Basic Comparisons	46
5.2.2 onDrawFrame()	47
5.2.3 Resource Loading	49
5.2.4 Event Handling	50
5.3 Optimisations	51
6. Evaluation & Conclusions	53
6.1 Critical Evaluation	54
6.2 Extensions and Further Work	56
6.3 Concluding Summary	57
7. References & Bibliography	58
7.1 References	59
7.2 Bibliography	61
8. Appendices	63
8.A Sangwin Engine Analysis	64

8.B Test Results	67
8.B.1 Desktop Tests	67
8.B.2 Android Tests	68
8.C Profiling Data	70
8.D Example Game Design	72
8.E Using the NDK with Netbeans	73
8.F Interim Log	76

1. Introduction



1.1 Description of the Project and Problem Area

This project aims to tackle the problem of developing hardware intensive 3D games for mobile phones. It focuses in particular on the use of Java and native (C++) code for the Android OS, a widely used mobile phone (and recently tablet) operating system developed by Google. It attempts to compare the two languages by analysing the design and performance of a 3D game engine originally written in Java and translated to C++ as part of the project. The Java engine has been supplied by Sangwin^[21], a small games company based at the University of Sussex.

Mobile phones are difficult to develop for due to their limited memory and processing power. However, in recent years advances in hardware have given developers the ability to build 3D games for mobile phones, while digital download services such as Apple's App Store and the Android Market have provided a way for applications to be released quickly and easily, and at little cost.

This has led to an explosion in the mobile gaming industry, with many companies opting to release major titles for mobile phones alongside dedicated games consoles. Electronic Arts, for example, released the latest game in their popular football franchise, *FIFA*, for the Apple iOS. This sudden interest in mobile gaming has made it even more important to get the most out of the limited hardware available.

The Android OS is a very widely used mobile phone operating system^[2]. It was first released in September 2008, and has since had many updates and changes that have led it to become one of the most popular smartphone operating systems. It makes use of the Java programming language for development, and but also allows developers to program in native code if needed. Java has traditionally been avoided in the context of game development, mostly due to its speed and efficiency. However, as compiler technology has improved, the speed at which Java code runs has greatly increased, and some successful 3D video games, such as *Minecraft*, have been built using the language. Despite this, C++ is generally the language of choice when developing 3D games, as it still provides mechanisms that allow programmers to get the most out of the available hardware.

This project will try to improve the performance of the Sangwin Java engine by translating it into C++. Profiling techniques will be used to provide an informed analysis of each engine's performance. As well as this, the project will highlight a number of benefits and disadvantages of using C++ over Java when developing 3D games for the Android OS. The resultant information will be used to hopefully provide a clear indication as to which language is the better choice for developers looking to build games for Android.

The project will also involve learning a number of new technologies. C++, OpenGL, game engine programming and the Android OS will all need to be understood and learned to a reasonably high level for the project to be successful.

1.2 Research and Technologies

1.2.1 Game Engines

It is a requirement of this project to gain an understanding of the processes behind a game engine, as this will help to make the translation easier to complete and improve the analysis of the resultant programs.

A game engine is a tool which is used to build and power video games. It performs all of the complex mathematic and rendering operations so that a game developer can concentrate on the design of the game, rather than the low level functions behind it. Programmers can build games with a number of tools that the game engine provides. These can typically be grouped as follows:

- **Scenegraphs and rendering**

A scenegraph uses a hierarchical tree structure to organise and render objects. Each object has spatial and visual information which dictate it's position, orientation and material in the scene.

- **Models, textures and animation**

A game engine allows programmers to load 3D or 2D model data from external sources. This data can also include material, texture and animation information.

- **Collision detection**

Collision detection is important for any game, as it adds believability and realism. It can be handled in a number of different ways by the game engine. One of these, which is used in the Sangwin engine, will be discussed further on this report.

- **Event handling**

Game events can either come from the user (pressing a button, for example) or from the game itself (such as a collision between two objects). A game engine allows the programmer to handle these events.

- **Physics**

Modern game engines tend to also include one or more physics libraries. Having realistic physics is a necessity in today's game market, and so a large amount of effort is normally invested in this area.

- A game engine often also includes the ability to build user interfaces, add artificial intelligence algorithms to objects and play sounds and music.

There are a number of feature-rich game engines available to Android. Some examples are Unity^[25], ShiVa 3D^[22] and AndEngine^[5]. Unity and ShiVa 3D provide 3D visual editors to build games, and are very well established and complex pieces of software built by large teams of programmers. These kinds of engines are used in the professional industry, and are far more sophisticated than the Sangwin Engine. AndEngine is a 2D game engine that was conceptualised and developed by a single programmer^[4]. It is a good example of a successful engine in a similar scope to Sangwin.



Fig 1. The ShiVa 3D Editor

1.2.2 C++ and Java Performance

C++ and Java are both object-oriented programming languages which can be used to build complex and intricate applications. Syntactically they are very similar, and they provide a range of comparable tools and constructs. However, there are a number of key differences that set the languages apart.

C++ is a statically compiled language. This means that programs using it are restricted to the architecture that they are compiled on. This can make porting code difficult, as the program must be separately compiled for each desired target architecture. In some cases, the code may also have to be changed if calls to the host operating system are required. For large scale projects this can be time consuming as some entire sections of code may have to be redesigned if the operating system is vastly different. Static compile times can also be very large when there are a lot of source files.

Java runs inside the Java Virtual Machine, or JVM. This is an intermediate code layer that lies between Java code and assembly language. The Java code is compiled into system-independent 'bytecode', which is in turn compiled into system-dependent assembly language.

There is a JVM for most system architectures, which means that once a Java program has been compiled it can be run on almost any system, making Java programs not only portable, but secure, as a VM can be used to enforce security policies. For example, Java applets (web based Java apps) are restricted to running in a 'sandbox', which limits the activities they can perform (like reading and writing to and from disk)^[18]. This helps to prevent malicious code from being executed unknowingly.



Fig 2. Java Language Design

The main downside of the virtual machine approach is speed. As the target architecture is not known at compile time, Java bytecode must be translated into machine code at runtime. This is known as interpreting, which incurs overhead due to the extra runtime code analysis. This can be particularly expensive with complex functions that are called frequently.

The performance of Java interpretation can be improved with the use of a Just In Time (JIT) compiler. This is an interpreter which caches translated code at runtime, so it can be used again without having to be recompiled. This produces large performance increases as the amount of translation is far less than with traditional interpreters. A JIT compiler can also take advantage of CPU instruction sets not known at static compile time, and perform runtime optimisations that are not available to static compilers.

This project will compare the performance of Java using a JIT compiler against C++, as the Android OS provides a JIT compiler with its Java implementation.

1.2.3 Developing Natively for Android

Android developers use the Android Software Development Kit (SDK) to build applications for the operating system. The SDK gives programmers access to the OS framework, and allows them to retrieve hardware input data and build interfaces using the Android UI components. The SDK uses

Java, and all Android applications run inside a JVM called Dalvik. A detailed description of a typical Android application lifecycle is provided in **section 3.1**.

Netbeans is a well-known and well-used IDE for C++ and Java development. It also provides functionality to code, compile and test Android applications using a plugin for the Android SDK. This makes it a good choice for Android developers, and it has a large amount of support if any issues occur. It is therefore the choice of development environment for this project.

The Android Native Development Kit (NDK) was released in June 2009, along with Android 1.5. The NDK provides tools which allow developers to implement **some** sections of code in C/C++. **Some** is highlighted here as, until very recently, applications could not make use of all of the Android framework solely in native code, and had to include a Java layer. The release of Android 2.3 in December 2010 rectified this, and specific details of this version can be found in **section 3.1**. The NDK includes scripts to build native code for specific architectures, C++ and C libraries, header files for native system calls and some header files for Android OS functionality that is available in native code.

The NDK must be used judiciously, as it can cause negative performance affects when used in some situations. Google Android developer David Turner gives a good description of appropriate NDK use in a blog post accompanying its initial release^[17]. He states that an application will be “more complicated, have reduced compatibility...and be hard to debug.” These are all changes that will have to be addressed in this project. However, he also says that applications involving “self-contained, CPU-intensive operations” could benefit from performance increases. A game engine is an excellent example of this kind of application, and so should see improvement from NDK use.

1.2.4 The Java Native Interface

To access native code from the Java layer, Android 2.2 uses the Java Native Interface (JNI). This is a pre-existing tool which allows both languages to make calls to each other. This works by giving C++ access to the running JVM implementation, by way of the ‘JNIEnv’ pointer. This pointer can either be passed from Java to C++ in a function, or created natively in C++ code. It contains a number of functions to make calls to Java objects from C++, convert types between the two languages and access variables declared in the other language.

Each call to a JNI function creates a certain amount of overhead, and so frequent calls to small functions in either language can create a negative affect on performance. Therefore, wherever possible JNI calls must be reserved for complex functions which will offset the overhead. For this project to be a success, a good balance will have to be found between Java and C++ in order to minimise the affects of the JNI.

1.3 Professional Considerations

It is a requirement of the University that this project adhere to the regulations described in the BCS Code of Conduct^[7] and Good Practice^[8]. Included here are the regulations in the Code that are relevant, and details of how this project will comply with them.

1.3.1 Code of Conduct

- Duty to Relevant Authority, 6 - This point describes how a difference of interest should be handled. This can occur between the authority in control of the project, the intended users and the employee. In this case, the relevant authority is Sangwin Games. If a difference of interest occurs, it will be handled by the Project Supervisor.
- Duty to Relevant Authority, 7 - This point is similar to the one above, in that it describes what should be done if a conflict of interest is foreseen at any point during the project. If this occurs, Sangwin and the Project Supervisor will be informed as quickly as possible.
- Duty to Relevant Authority, 8 - This point states that no confidential information should be disclosed except with the relevant permission. A Non-Disclosure Agreement has been signed with Sangwin, which prevents their code from being used for anything but this project.
- Professional Competence and Integrity, 14 - This point states that relevant technology must be used, and new technology should be used where appropriate. In order to adhere to this, this project will be based on as much appropriate literature as possible.
- Professional Competence and Integrity, 15 - This point ensures that no task is undertaken if it cannot be completed. In order to make sure that this is complied with, before beginning a task the three questions listed in the description of the regulation will be asked. This will help to assess developer competence. If advice is needed, it will be asked for from the Project Supervisor.

1.3.2 Code of Good Practice

- 2, Manage Your Workload Efficiently - This point is fairly self explanatory. This project will adhere to a strict project plan which will be agreed with the Project Supervisor.
- 2, Participate Maturely - By adhering to this point, criticism will be accepted constructively and the workload will be well managed. This point will also ensure that a good working relationship will be kept with Sangwin, which is a necessity if this project is going to be successful.
- 2, Respect the Interests of your Customers - This regulation will be complied with by regularly updating Sangwin with details of the project, and informing them of any appropriate information as quickly as possible. As previously stated, no confidential information will be disclosed to third parties.
- 5.1, Requirements Analysis and Specification - This is an important point, as it describes a solid foundation for gathering requirements. By adhering to the regulations described in this point, the **Requirements Analysis** will be successful, and allow the rest of the project to run smoothly and effectively.

- 5.2, When Programming - This point states that code should be maintainable and well restructured. Code in this project will use a regular format, and include informative and concise commenting. It also requires awareness of platform limitations, which is a key aspect of this project.
- 5.2, When Testing - When designing and carrying out tests, appropriate tools will be used. A wide range of tests will be run, and a detailed log of results will be kept.
- 5.3, When Porting Software - This point will be adhered to by analysing the differences between C++ and Java, and making appropriate design decisions based on these differences. This will be investigated further as code is profiled and tested.

2. Requirements Analysis



2.1 Understanding Requirements

In almost all software engineering projects, establishing a solid set of requirements from potential users and stakeholders is key to success. Without understanding the needs of the project before it begins, it can be extremely hard to design and build a project effectively within the given timeframe.

2.1.1 Gathering Requirements

In this project, the basic software design requirements have already been established in the building of the original software. The software is well designed, and so rethinking the original requirements is not necessary. However, these requirements need to be thoroughly understood in order to make a successful port of the original engine. **A high level description of the original requirements is as follows:**

1. **The game engine should have functionality to create both 2D and 3D games to a reasonably high standard.**
2. **The game engine must be compatible with the Android OS.**
3. **The game engine must be efficient enough to work across a range of hardware devices.**
4. **The game engine must be usable by developers, and provide a useful API that makes it easy to program with.**

All of these requirements apply directly to the C++ translation of the Java engine. This project also extends the original requirements list in order to improve the original software. These are shown in **section 2.3**.

The stakeholders for this project are Sangwin, as this project can improve their software, and the Project Supervisor. The users are the existing developer base (i.e Android Java developers), and a set of new developers (i.e Android C++ developers). Both sets of developers must be considered when building the C++ port of the engine.

The new software requirements were established through a series of **interviews** with Sangwin and the Project Supervisor. Most of these interviews occurred before and around the start of the project. Interviews were also arranged before and after most large sections of code were implemented, so that feedback and advice could be given. Requirements were also established through extensive **background research**, which brought to light the different technologies needed for the project to be successful, and the areas which could cause difficulty.

2.1.2 Software Engineering Approach

Agile was the software engineering approach adopted for this project. It was chosen as it works well with the module-based structure of the Sangwin Engine. Design did not take place entirely before programming, rather each section of the engine was examined, designed, translated and tested before moving on to the next. This reduced the possibility of errors, and meant that new programming skills learnt could be used when designing the next section of the program.

2.2 The Sangwin Java Engine

In order to understand the scope of this project, an in depth analysis of the Sangwin source code was required. This analysis was achieved through a mixture of self-study and regular meetings with Sangwin. A large amount of the analysis took place before any programming, but the process was generally ongoing throughout the project as more parts of the engine were translated.

The results of the analysis showed that almost every class in the Sangwin Engine would need to be translated into C++ for the project to be successful. Redesigns were needed due to language differences, and these are detailed in **section 3**.

The results of the analysis is included in **Appendix 8.A**. This includes descriptions of each package in the Sangwin Engine, along with brief descriptions of classes and high level class diagrams for inheritance hierarchies. Also listed are the classes that were not included in the translation, and the reasons for excluding them.

2.3 Requirements Specification

2.3.1 Primary Requirements

These are the functional and non-functional requirements that **must** be completed in order for the project to be successful.

1. A number of new technologies and techniques must be learned:

- 1.1. The C++ programming language must be learned effectively, to write a high quality code base for the engine
- 1.2. The OpenGL ES graphics API must be learned, in order to understand the process behind rendering using hardware acceleration
- 1.3. Game engines must be understood so that improvements can be made to the new software as each section of the original engine is translated
- 1.4. Developing for the Android OS, using both Java and C++, must be researched extensively

2. The Sangwin Engine must be translated from Java to C++:

- 2.1. Each of the required classes must be translated
- 2.2. Some areas of the engine must be redesigned so that they work effectively in C++, and any new classes that are required by this redesign must be built
- 2.3. As much Android-specific code as possible must be removed from the engine so that it can easily be ported to other platforms that support C++
- 2.4. A working desktop version of the engine should be created for testing, and to potentially provide a way of extending engine functionality in future
- 2.5. A working Android OS version of the engine, that includes the same, if not more, functionality as the original Java engine, must be created

3. Both engines must be compared to measure the differences between each:

- 3.1. The differences between the main render functions of each engine should be measured
- 3.2. The differences between the resource loading functions of each engine should be measured
- 3.3. The differences between the event handling functions of each engine should be measured

- 3.4. The areas outside of performance where C++ provides advantages and disadvantages over Java should be identified

4. C++ code should be optimised to improve performance:

- 4.1. The areas of code that would benefit most from a performance increase, such as expensive functions or functions that are called frequently, should be identified
- 4.2. These areas should be improved wherever possible by using C++ programming techniques discovered through research

2.3.2 Secondary Requirements

These are the functional and non-functional requirements which are not required for the project to be successful. They are potential extensions to the project, which will be undertaken if time allows.

1. A useful API that is easy to develop games with should be created:

- 1.1. The C++ code should be kept as similar to Java code as possible to retain usability for existing developers
- 1.2. Comprehensive and useful documentation should be generated using a relevant tool
- 1.3. Examples and tutorials should be provided, which show how to use the engine
- 1.4. The engine should be condensed into a single library that can be easily included by developers

2. A small example game should be developed using the C++ engine:

- 2.1. A game should be created which shows the different sections of the engine working effectively

3. The area of memory management should be investigated to further improve performance:

- 3.1. Background research should be performed to understand the advantages and drawbacks of different memory management techniques
- 3.2. A number of different techniques should be implemented and measured to improve engine performance and provide insight into managing memory on mobile devices



3. System Design & Implementation

3.1 Using The Android OS

3.1.1 The Activity Lifecycle

The Android OS is based around a simple framework known as 'The Activity Lifecycle'. An Activity is a process dedicated to a particular set of tasks. An application can have a number of Activities for handling areas such as UI and display. The Activity Lifecycle describes how an Activity is handled by the OS depending on different events. It consists of 7 methods which should be implemented in order for the Activity to work correctly. These methods deal with starting, switching-between and ending Activities. This diagram is taken from the Android SDK documentation^[3]:

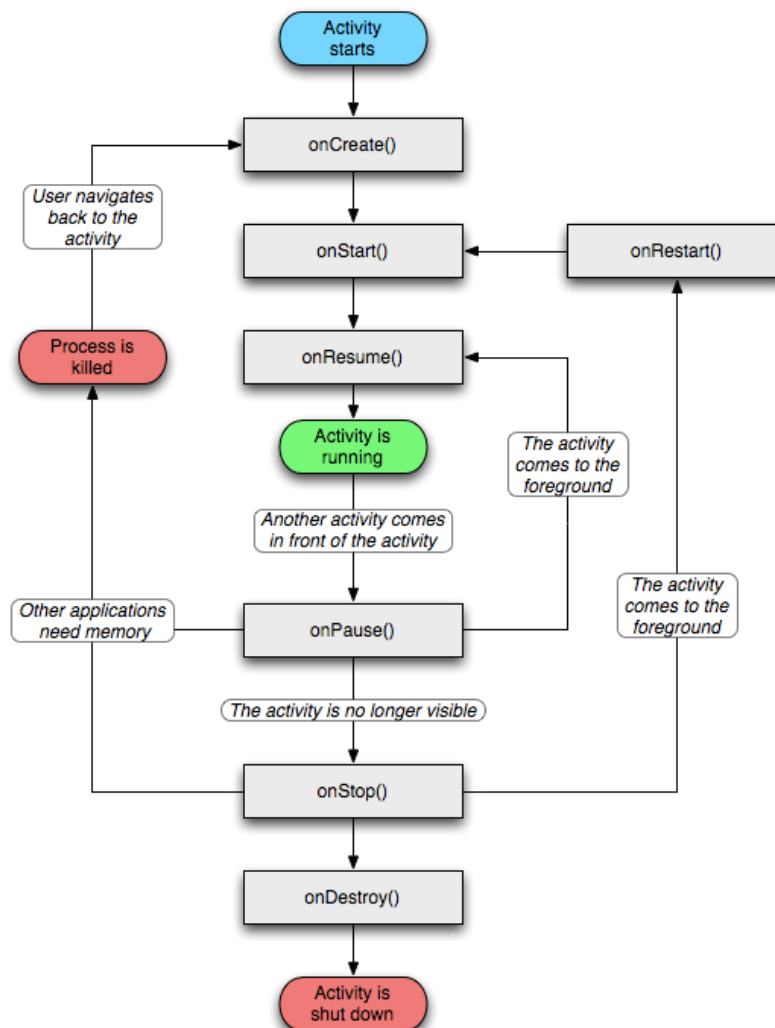


Fig 3. The Android Activity Lifecycle

A basic game created using the Sangwin engine consists of one Activity, a canvas, which displays the onscreen render. The Android `GLSurfaceView` and `Renderer` classes are used to implement this.

3.1.2 Android 2.2 and 2.3

Android 2.2 is the OS version that the Sangwin C++ engine is developed for. 2.2 is the most widely available version of the Android OS. It provides a recent release of the NDK which includes headers

for accessing Android Bitmaps and the Log system natively. This makes it a good option when deciding to build a native application for the system.

Android 2.3 has far more support for native development than the other OS versions. It introduced the ability to build a completely native application with the `native_activity` header file. This gives developers access to UI components, resources and the input event queue, all of which are needed by the Sangwin Engine. It also eradicates the need for frequent JNI calls, which improves the speed and efficiency of applications.

Android 2.3 was not used in this project for three reasons. The first was the timing of its release. 2.3 was released in early 2011, which was over half way into the lifecycle of this project. All planning had been made for 2.2, and so this would have involved a large project redesign. Secondly, native applications are complex, and require a thorough understanding of Android, JNI and C++, and using it would have added unnecessary complexity to an already difficult programming task. Finally, 2.3 is not yet available on many devices, meaning that applications built using it cannot be widely used. Testing would also have been restricted to the Android Emulator, which is not representative of actual hardware, as access to a device running 2.3 was not available.



Fig 4. Screenshot from an HTC Desire HD mobile phone running Android 2.2

3.2 Managing Memory

Unlike Java, C++ does not provide a Garbage Collector to automatically delete objects when they are no longer needed. This means that objects created on the heap must be deleted manually with the `delete` keyword. This can become difficult when an object is pointed to by multiple other objects, as deleting it in one place may create errors in another, and cause the program to crash. A `delete` statement can also easily be missed by the programmer, resulting in a memory leak.

3.2.1 Reference Counted Smart Pointers

A common and well tested solution to this problem is reference counting. This involves keeping a count of the number of pointers to an object. When the count is 0, the object can be safely deleted as nothing is pointing to it.

Reference counting can be combined with ‘smart pointers’. Smart pointers extend raw C++ pointers with added functionality. They can be made to automatically delete the raw pointer they hold when they go out of scope. This means that the programmer rarely or never has to use the `delete` command. Smart pointers can also be made to increment and decrement the reference count of the objects they point to, so as to ensure that they are not deleted at the wrong time. This solution is widely recognised amongst C++ programmers. For example, Boost, a widely used C++ library, has a number of smart pointer implementations for a range of uses^[9]. Unfortunately, this library is not available for use with the NDK, and so these implementations could not be used.

The C++ translation makes use of a reference counted smart pointer implementation suggested by Scott Meyers^[19]. This implementation uses two classes: `SPtr` and `Object`. The `Object` class is the base class for all objects that require reference counting. It contains the functionality to increment and decrement the reference count, and delete objects when the reference count reaches 0. The `SPtr` class is the actual pointer wrapper. It uses operator overloading to allow smart pointers to be dereferenced in the same way as raw pointers. The classes can be used as follows:

```
//Reference counted class
class Foo: public Object {
    //add functionality...
};

//Smart Pointer usage
SPtr<Foo> f (new Foo());
f->function();
```

This implementation is easy to use and relatively lightweight, as only one extra field is needed per pointer, for the reference count. It also makes the class structure more Java-like with the inclusion of a base class for all other classes. Abstracting the reference counting behaviour into a base class also means that it can be extended easily in future.

3.2.2 Inheritance, Casting and Type Safety

Inheritance with smart pointers can be tricky, as the compiler cannot implicitly see the relationship between classes that they wrap. To get around this, the `operator SPtr<newType>()`^[23] function

was included in the `SPtr` class. When an unexpected `SPtr` type is passed as a parameter, this function tries to convert the raw pointer it holds into the required parameter type. If this cannot be done, a compile time error will occur. If it can, a `SPtr` of the required type will be returned, and the function will work correctly.

Explicit casting is handled with the `smart_static_cast()` function, used as follows:

```
SPtr<Foo> f(new Foo());  
//where Foo can be casted to Bar  
SPtr<Bar> b(f.smart_static_cast(SPtr<Bar>()));
```

This function casts the raw pointer held by the smart pointer to the required type, and returns a new smart pointer. If this is not valid, an error will occur at compile time.

The `SPtr` constructor is `explicit`, meaning that it has to be physically called to be used. This stops C++ creating smart pointers from raw pointers implicitly. This may make programs more longwinded, but it means that functions expecting `SPtrs` can only accept `SPtrs`, which helps to clarify the way the function should be used to the developer.

3.2.3 Drawbacks

Despite its usefulness, this implementation does have some drawbacks. The inclusion of a base class means that all classes that require reference counting will have a vtable due to the virtual destructor in `Object`. This increases the size of the object in memory, and slows down deletion^[10].

Also, the size of the compiled binary increases, as all classes using smart pointers will have a `SPtr` template class. However, unless a developer creates a game with thousands of classes using reference counting this should not be a problem, as storage space on modern phones is normally very large.

3.3 Native Android Limitations

3.3.1 The C++ Standard Template Library (STL)

The C++ STL is a code library which provides a number of container classes and algorithms that are useful for C++ programmers. It is very widely used and considered a standard for almost all C++ development. However, the 4th revision of the Android NDK, which was the revision available for most of this project, does not supply a version of STL. This meant that where STL functionality was required, it had to be built from scratch.

There are a number of Java container classes that handle automatic resizing and item deletion in the Sangwin engine, and all of these had to be designed and implemented in the C++ port. Each class was designed to contain the same functionality as their Java counterparts, so as to make the use of the two engines as similar as possible. A class diagram of the container classes is shown below, without any constructors or destructors:

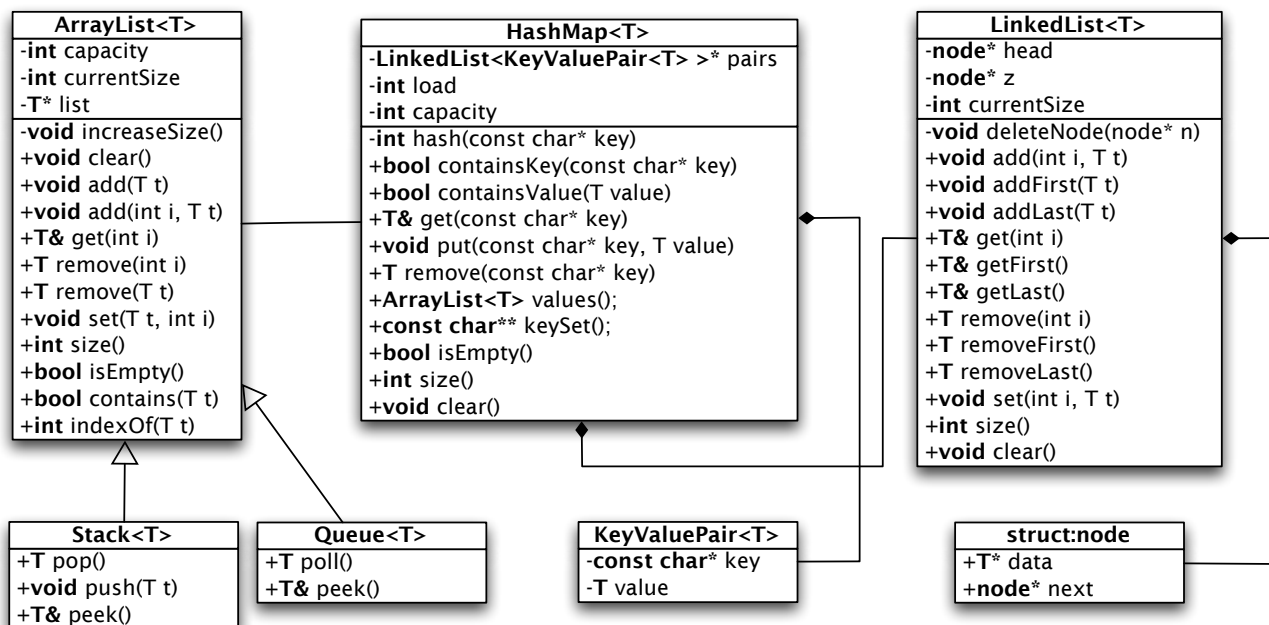


Fig 5. Class diagram of container classes

This class diagram shows how the container classes use each other to minimise code reuse. The inheritance structure of **ArrayList**, **Stack** and **Queue** also means that functions expecting an **ArrayList** as a parameter can accept either of the subclasses. This is in line with Java behaviour. **HashMap** uses **LinkedList** to implement a separate chaining algorithm to deal with collisions. A number of other algorithms were considered, but this was deemed the simplest and best to implement in the given timeframe. The **HashMap** class is also restricted to character based keys, as it uses a hashing algorithm that computes a hash based on the ASCII value of each character^[1].

The container classes are intended to be used with reference counted smart pointers. This is so that objects are removed from the containers correctly on destruction, and also so that the functions that make use of equality operators work as expected.

3.3.2 Real Time Type Information (RTTI)

The 4th revision of the NDK does not provide support for RTTI. RTTI is a C++ device that allows the checking of object or primitive types at runtime. This is useful for finding the type of objects stored in polymorphic pointers. In Java, RTTI is implemented through the use of the `instanceof()` keyword. This keyword is used numerous times in the Sangwin Engine to find the type of objects such as `Lights` and scenegraph `Nodes`, which have large inheritance hierarchies.

RTTI was achieved in the C++ translation through the use of enums. An enum exists for every type where RTTI is needed. Each object has a type field, which is set to the correct enum value in the constructor. This field can then be checked at runtime.

The problem with this approach is that the enum values have to be hard coded into the root superclass. This makes it impossible to extend inheritance hierarchies without having access to the source code, which is an issue if the engine is to be released without giving the source to developers. This will have to be rectified by using the NDK 5th revision in future.

3.3.3 Exceptions

Exception handling is not available in the 4th revision of the NDK. Thankfully, exceptions are not widely used in the Sangwin Java engine, and so designing workarounds was fairly straightforward.

The Java `UnsupportedOperationException` is used in instances where cloning is not available. This is handled in C++ by returning a null pointer, and appropriately documenting the function.

Java `IOExceptions` are used when loading or attempting to load file data from external sources. In the C++ translation, the loading of files is handled Java-side, and so these exceptions can be used and handled as normal.

In a future revision of the engine port, the 5th revision of the NDK will be used so that C++ runtime exceptions can be implemented in the code base.

3.4 The Scenegraph

3.4.1 What are Scenegraphs?

Scenegraphs are tree structures which consist of a number of nodes, each with a different purpose. Nodes are game objects, which can consist of characters, items, lights and cameras. There are also more abstract nodes, which can hold visual and spatial information for game objects to make use of. The tree structure allows for local (related to parent) and world (related to root) positioning. The renderer will traverse the scenegraph and perform appropriate operations depending on the node type.

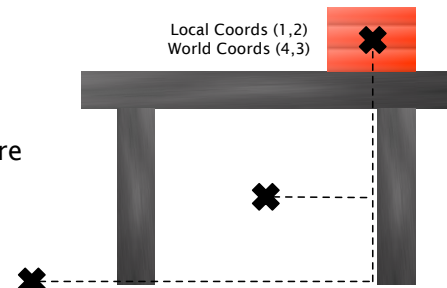


Fig 6. Simple scene example, showing a box parented to a table

The Sangwin scenegraph uses an inheritance hierarchy to define node types. The basic hierarchy is as follows:



Fig 7. Sangwin node structure

The node types are mostly self-explanatory, but **Bound** may need further explanation. It is a node which handles collision detection between engine objects using a bounding volume hierarchy. Bounding volumes are developer defined primitive shapes which wrap around nodes and their subtrees. The bounding volumes can be checked against each other to see if they come into contact, which defines a collision. Primitive shapes are used to make the collision calculation as efficient as possible. Every usable node extends **Bound**, and so can have a bounding volume added if collision detection is required.

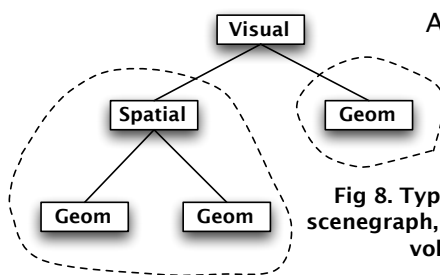


Fig 8. Typical Sangwin scenegraph, with bounding volumes

A typical Sangwin scenegraph will consist of a number of **Visual** and **Spatial** parent nodes which define position and appearance information. These will have **Geom** children, which contain geometry and are physically drawn to screen by the renderer. Some nodes will also have bounding volumes, if the developer assigns them.

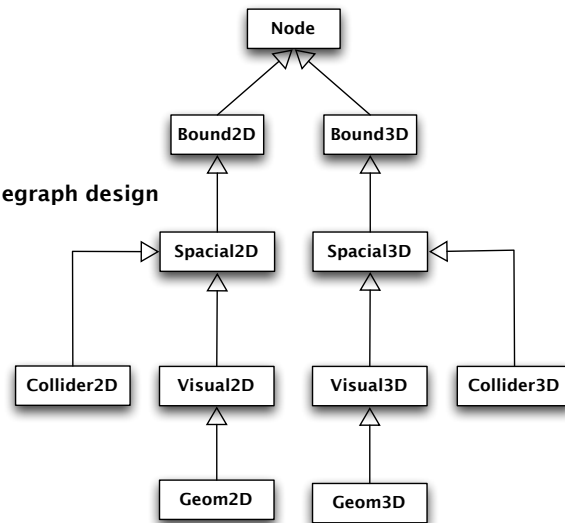
3.4.2 C++ Designs

The original Sangwin engine uses generic types to create a scenegraph which allows the creation of both 2D and 3D scenes. However, the use of generic types makes this design quite complex, so a number of different designs were considered for the C++ implementation.

Single Inheritance

The first design eradicated the need for generic typing by using single inheritance. This created a simple, branching scenegraph with completely separate classes for 2D and 3D nodes.

Fig 9. Single inheritance scenegraph design

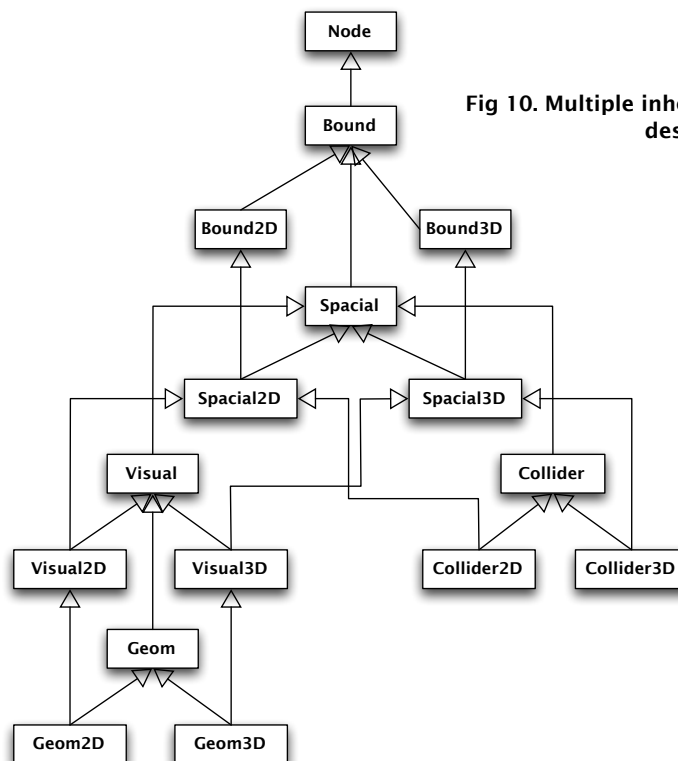


This design was straightforward to implement, but required a large amount of code reuse, not only in the scenegraph itself, but also in the other classes, such as the `Renderer` and `CollisionDetector`, that use the scenegraph in their code. Code reuse should be avoided if possible, and so this design was not implemented.

Multiple Inheritance

A design that made use of C++ multiple inheritance was also considered. This scenegraph removed the generic type system, and also partly solved the problem of code reuse with abstract parent classes for each node type.

Fig 10. Multiple inheritance scenegraph design



As can be seen from the class diagram, this design is complex, a typical problem with C++ multiple inheritance. Also, subclasses have multiple parents which inherit from the same base class, which creates issues with ambiguity and polymorphism^[11]. This was deemed a critical flaw, and so this design was not chosen.

Templates

The final design made use of C++ templates, which have similar functionality to Java generic types. This resulted in no unnecessary code reuse, and a scenegraph with the fewest classes.

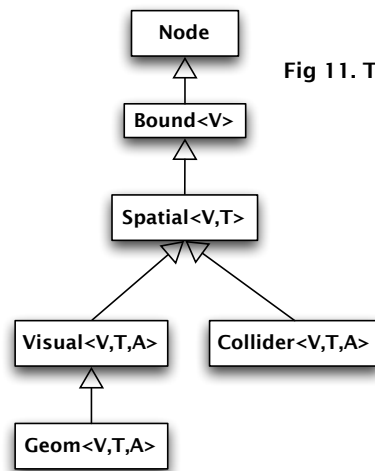


Fig 11. Templated scenegraph design

This design has some differences with the Java version. An extra template type is required throughout the scenegraph as C++ does not allow the use of templated types declared without template parameters, which is used in the Java `Bound` class. Also, templates cannot be restricted to a given set of types, which means that any valid type can be entered. However, this does not affect the classes as the constructors are protected and instances can only be created via static functions with preassigned template parameters.

Despite the above issues, this design was the most concise and was therefore chosen as the implementation option in the C++ translation. It is also similar to the Java engine design, so its use should be familiar to any existing developers.

3.4.3 The GetNode Class

The Sangwin Java engine includes static functions to create nodes, so as to avoid frequent use of generic constructors. This cannot work in C++, as templated classes cannot be used statically without the inclusion of template parameters. The proposed and implemented solution was to move all of the static node creation functions to a single class, called `GetNode`. Bounding volume constructors were also moved to the class. `GetNode` also defines `typedefs` for `SPtrs` to nodes. Using this functionality, nodes can be created like this:

```
Visual3D myVisual (GetNode::newVisual3D("myVisual")) ;
```

The above line of code creates a new visual node wrapped in a `SPtr`. This makes creating nodes concise and straightforward for the developer, as no templates for either `SPtrs` or `Nodes` have to be explicitly declared.

3.5 Android Event Handling

User input and in-engine events are both handled in the Sangwin Java engine by the `EventHandler` class. Any valid Java `Object` can be stored in the events buffer, which means that the engine is compatible with any Java input event system. The typical sequence of events in the `EventHandler` is shown below.

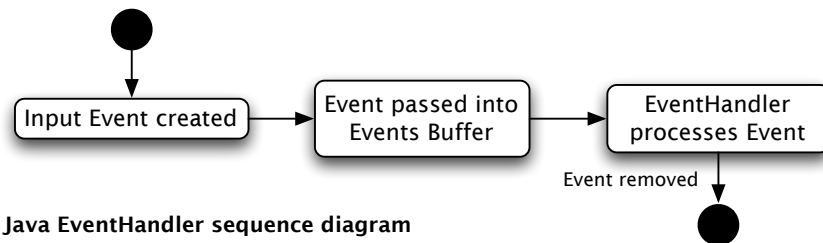


Fig 12. Java `EventHandler` sequence diagram

3.5.1 C++ Designs

In the C++ engine, handling in-engine events can work in exactly the same way as the Java engine. However, as there is no class that subclasses all C++ types, an `Event` class was created that all events can extend. A `void` pointer could have been used, but this cannot be reference counted by the `SPtr` class, and so was not a viable option.

Handling user input events required some design changes as Java objects cannot be stored in the C++ events buffer. Java events must be converted into C++ `Event` objects in order to be stored and processed correctly. This means that for every required Java input event, a corresponding `Event` subclass must be built. This is a downside of using Java and C++ together that cannot be avoided.

The conversion of the Java event to a C++ `Event` can happen both Java side and C++ side. The conversion could happen in C++ by passing the Java object directly to C++ using JNI, and using JNI function calls to access the event data from the Java object to build the C++ `Event`. This option would be very inefficient as there could be many JNI calls required to extract the data from large Java event objects.

Instead, it was decided that the conversion should happen Java side. This is achieved by extracting all of the primitive event data from the event in Java, and passing the data to C++ as parameters to a JNI function.

Although copying the event data and making a JNI call is an expensive operation, the affect on performance is minimal as user input happens very infrequently.

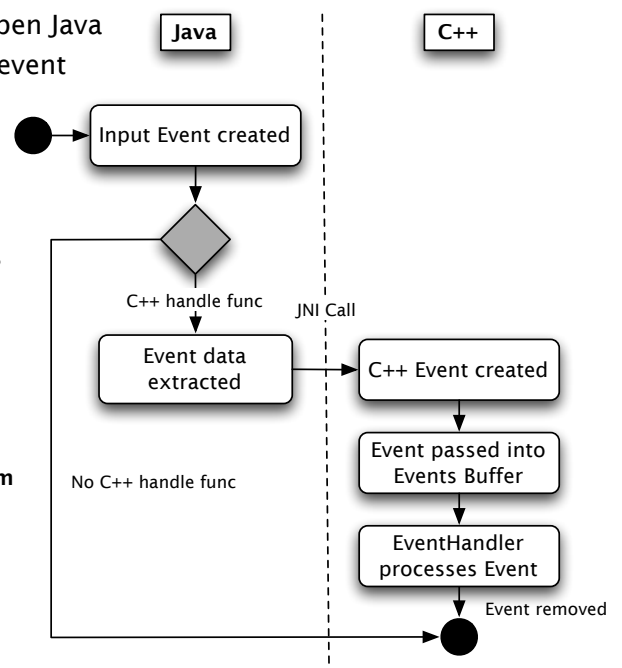


Fig 13. Java to C++ Event handling sequence diagram

3.5.2 Concurrency

The Android Event Queue runs in its own thread, which means `EventHandler` operations need to be handled concurrently. In the Java engine, this is achieved with the use of the `ConcurrentLinkedQueue` class, which automatically creates mutex locks when the queue is accessed.

There is no such class available to the C++ engine, so concurrency was handled in the `EventHandler` directly. Whenever an event is added or removed, a mutex lock is created on the event queue using the Posix Thread functionality of the `pthread` header file available in the NDK.

3.6 Android Resource Management

In the Sangwin engine, external game resources consist of texture files and model (.obj) files. These are loaded as they are needed by the `ResourceManager` class. Loading is hidden from the developer; a resource will simply be loaded if it is asked for and does not already exist in memory.

Game resources on Android are stored in the Assets directory. This is a directory that allows files to be accessed in Java by filename, by using the `getAssets().open("filename")` method. Assets must be opened in this way so that they are decompressed, as most file types are automatically compressed when the Android application package (.apk) is built.

3.6.1 Loading Textures

The Sangwin Java engine loads textures using the Android `Bitmap` and `BitmapFactory` classes. The `BitmapFactory` class retrieves the image data from a given file, and stores it in a `Bitmap` object. This object is then held in a `Texture` object, which is processed by the renderer with the `GLUtils.texImage2D()` method. This method is Android specific, and is tailored specifically to work with Android `Bitmap` objects. This texture loading functionality is part of the reason why the Sangwin engine is specific to Android, and so a design was looked for in the C++ engine which avoided the use of Android code.

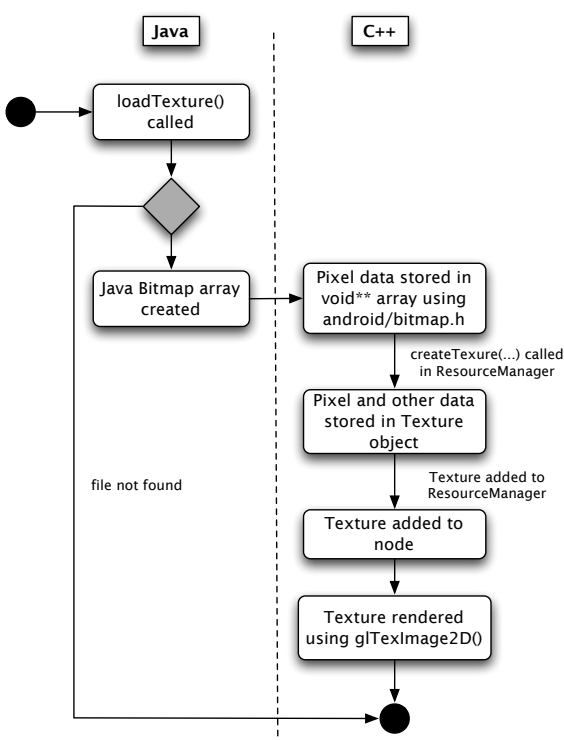


Fig 14. C++ engine texture loading

Loading textures entirely in native code would require the inclusion of a decompression and image handling library^[20]. This would drastically increase the binary size, and was deemed unnecessary as the Android OS already includes functionality to handle image files. It was therefore decided that retrieval and decompression of assets should be handled in Java.

So as to avoid the creation of a JVM pointer in native code and JNI calls from C++ to Java, assets are loaded when a game starts. This increases the memory a game uses, but reduces the complexity of the engine code and avoids texture loading during runtime, which can cause a drop in frame rate for large files. It also means that the engine remains independent of Android, which contributes towards **Primary Requirement 2.3**.

3.6.2 Loading Models

In the Sangwin Java engine, models are loaded from the Assets directory using the `AndroidOBJLoader` class. This is effectively a text parser which walks over the .obj file and performs appropriate operations depending on the textual commands. It incorporates the loading of .mtl files, which are sister files to .obj that specifically hold material and texture information. These files are referenced in the .obj file, and loaded when required. The resultant model is stored as a `Visual` node (or group of `Visual` nodes), which is passed to the `ResourceManager`.

In order to keep resource handling simple, model loading in the C++ engine also happens at game startup, with the files being retrieved and decompressed Java side. To avoid having to make JNI calls from C++ to Java, it was decided that textures and .mtl files needed by the model would be loaded before the .obj file, rather than as they are found in the file itself. To achieve this, the `loadModel()` method was designed, which walks over a directory containing all the files for a model and loads them in the right order.

Retrieving the file data in Java again means that the C++ code remains abstracted from Android. The `OBJLoader` class simply accepts an array of `char` data; it does not care where it comes from.

As a side note, the C++ `OBJLoader` was improved by separating functionality between it and the `OBJLexicalAnalyser`. This removes all tokenization functionality from the main loader, which simplifies the process and makes it easier to add new token types in the future.

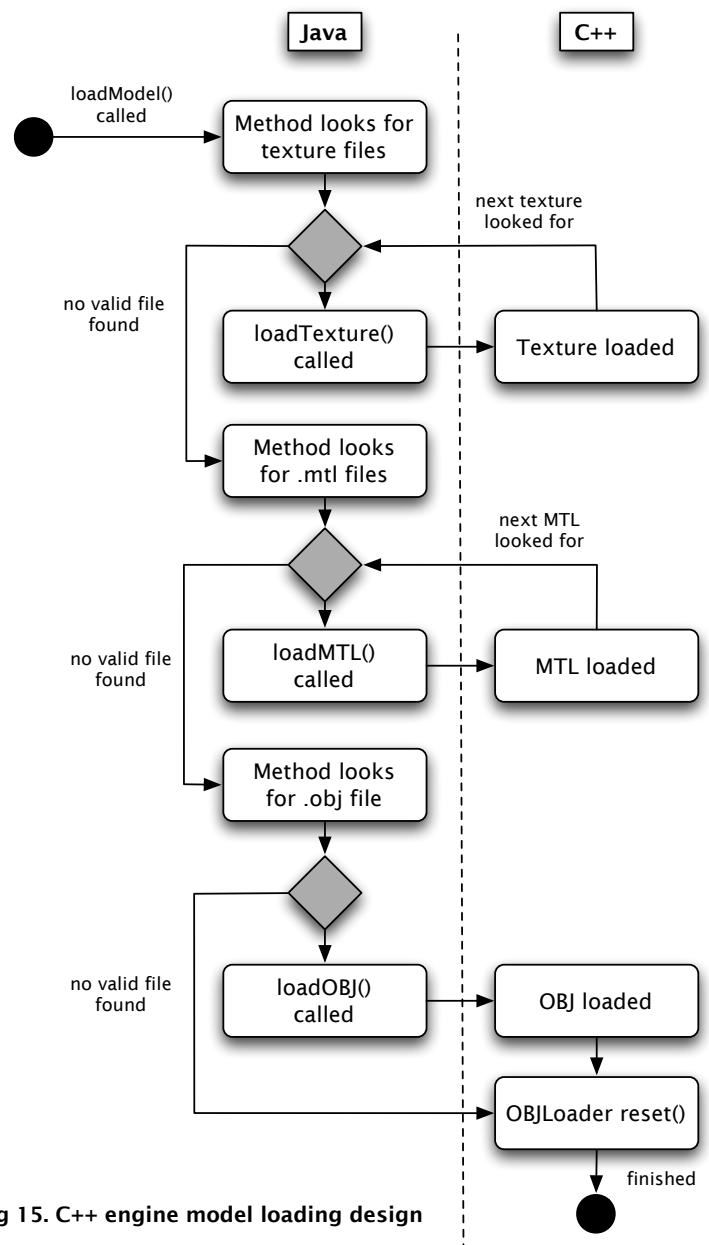


Fig 15. C++ engine model loading design

The Android robot logo, a green stylized figure with two antennae, is centered in the background. Overlaid on the robot is the section title in large, bold, black text.

4. Final Builds and Testing

4.1 Test Strategies

Testing is important for any software project, including one which involves porting existing software. It helps to bring to light any bugs in the code and address any issues that the end user may encounter.

Testing of the Sangwin C++ engine occurred at multiple stages of the development cycle:

1. **Low level Unit testing of the Math package**
2. **High level Unit and Integration testing of the Desktop engine**
3. **High level Unit and Integration testing of the Android engine**
4. **System Profiling of both the Android and Java engines**

Testing was a mixture of black and white box, and consisted of Unit and Integration tests. Each major area of functionality was tested, at a range of levels of granularity.

The 2D-specific functionality of the engine remained untested, as a 2D renderer was not built as part of the project. The reason for this is explained in **section 4.3.1**.

As the original engine was already known to be successful in its functional and non-functional requirements, high-level System testing was deemed to be less important, and was not performed to a great extent as part of this project. However, some System testing was performed in the form of profiling each version, in order to establish their performance differences. This is detailed in **section 5**.

End user testing was not performed in this project due to time limitations. The Sangwin Engine is a complex piece of software that needs a large amount of time to get to grips with, and so user testing was not viable.

4.2 Low Level Unit Testing

Low level unit testing of each of the Math classes was performed using the CppUnit plugin for Netbeans. This is an automated test framework which allows simple creation of tests for every function, and displays results on screen with a GUI.

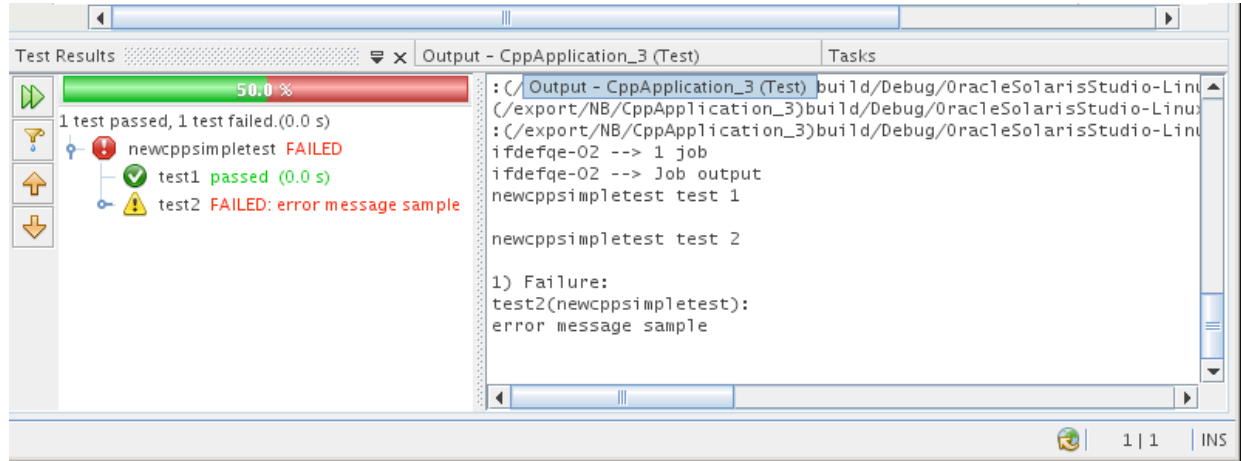


Fig 16. CppUnit plugin screenshot

The plugin allows use of assertions to dictate the passing criteria for a test. Multiple assertions can be used, and the framework will show which ones were successful and which ones failed.

Despite the usefulness of the plugin, it was not used to test the other parts of the engine code. This was due to the number of bugs with the software. There were multiple issues that involved manual editing of make and config files, as well as errors that could only be resolved by deleting and rebuilding all tests. This slowed down workflow and was very frustrating. Instead, manual testing methods were used, and are detailed in the coming sections.

4.3 Building and Testing the Desktop Engine

The Desktop version of the engine was used for testing the renderer, before adding the complication of Java and JNI in the Android version. This proved very useful, and helped to identify a number of bugs that would have been harder to track down on Android.

4.3.1 Desktop Implementation

The engine makes use of the cross-platform **Nokia Qt** OpenGL framework. This framework provides all of the required OpenGL functions and the ability to create windowed applications with a draw loop.

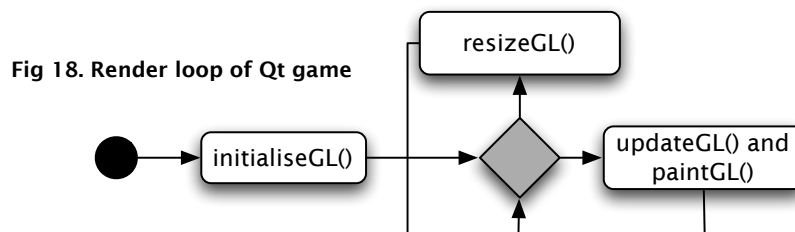


Fig 17. Nokia Qt logo

The Sangwin code is included as a static library, built using the Linux `ar` command from command line. This means that the source code does not have to be released to developers. The code can be given as a single binary, along with the header files. This makes for a neat release package.

The renderer built for the Desktop engine can only be used for 3D games. This was a design decision made to reduce the complexity and amount of testing. The renderer code was eventually used in the Android implementation, which meant that a 2D renderer was never created as part of this project. However, the renderer could be easily templated or a separate one built to work with the 2D components of the engine.

The basic render loop of a Qt-based Sangwin game is as follows. The loop runs until the game is ended, or until an error occurs that forces the loop to stop.



These are typical OpenGL functions which are synonymous with most OpenGL frameworks. This holds true for Android, albeit with different function names.

4.3.2 Desktop Testing

A series of high-level unit and integration tests were performed on the Qt implementation. The specific results of the testing data can be found in **Appendix 8.B.1**. Each set of test cases is grouped by category into tables, like the one below:

Test Case	Expected Result	Actual Result	Passed	Solution

Fig 19. Example test table

Where a test fails, the failed result is given along with a solution to the bug. If the bug could not be fixed, the reason is given instead of the solution. Resources and event handling were not tested on the Desktop version, due to the nature of their Android-based design.

4.3.3 Working Example

After the tests were completed and issues resolved, the engine worked as expected through Qt.

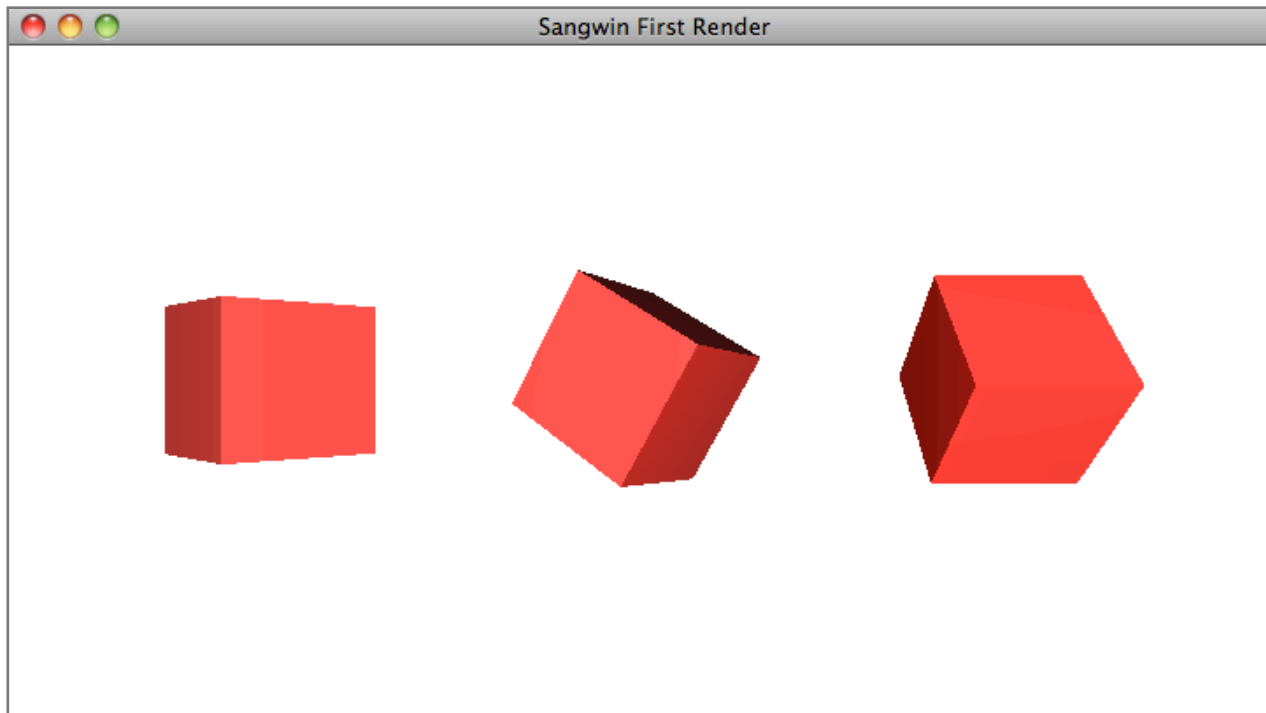


Fig 20. Desktop engine screenshot

The screenshot above shows three rotating cubes with materials, lit by an ambient light placed into the scenegraph.

4.4 Building and Testing the Android Engine

The Android engine is the main focus of **Primary Requirement 2**. It is needed to compare the differences between Java and C++ in mobile phone based 3D games.

Implementing the Sangwin C++ engine on Android was fairly straightforward after building and testing the Desktop version. The base engine code remained the same, and the renderer only had to be adapted slightly. The major differences lay in the use of the JNI to load resources and handle events, as described in **section 3**.

4.4.1 Using the JNI

Keeping the number of JNI calls to a minimum was a priority in the Android implementation. Aside from those described in **sections 3.5 and 3.6** for Resource Management and Event Handling, only 3 JNI entry points were needed. These are shown in the activity diagram below:

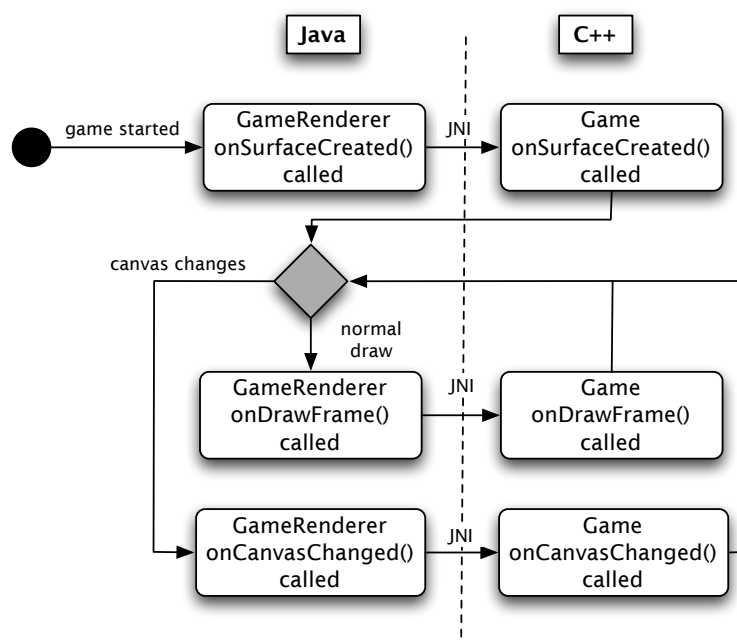


Fig 21. JNI implementation strategy

The JNI calls occur when a game starts, when the canvas changes size and when a frame is drawn. The starting and canvas change calls will only happen a few times during the run of a game, so their effects are minimal. `onDrawFrame()` must be called every frame to update the view. This means that an average frame will require only 1 or 2 JNI calls.

This implementation requires two Java classes; `GLSurfaceView` and `Renderer`. The `GLSurfaceView` is the panel which is used as the canvas. The `Renderer` renders OpenGL to the screen, although in this case it makes a JNI call to handle rendering. These classes are implemented in the Android engine as `GameView` and `GameRenderer` respectively. The `GameView` also handles UI events.

The JNI methods are declared Java-side in the `NativeExports` class. Each method is static, making them easily accessible from any class. In C++ there is a `GameExports` file. This is where each of the JNI functions are defined and made visible to Java. These files are both available to the developer so that they can be extended.

An example game using the engine should have a `Game` class, which should contain all of the game logic. It should use the Singleton design pattern to not only ensure that only one `Game` can be created per application, but to also make it easy to access via the JNI. The class should make calls to the `Android3DRenderer` through the appropriate functions.

4.4.2 Android Testing

Android testing consisted of a number of high-level unit and integration tests. These tests take the format of those in Desktop engine, and the results are shown in **Appendix 8.B.2**.

4.4.3 Implementation Technicalities

For the most part, the renderer code remains the same as in the Desktop version of the engine. However, the OpenGL `gluPerspective()` function (which calculates perspective view) is not available when using the NDK, so it had to be redefined using other OpenGL functions^[16].

The Sangwin C++ code was compiled into a single binary using the NDK. This binary can easily be included as a static library by the developer. The library includes the `Android3DRenderer`, to keep the number of separate files down to a minimum. If this engine were to be released, skeleton Java classes, makefiles and JNI files would be provided that show how to set up a basic game. Currently, the details of a basic setup are included in the documentation submitted with this report, including example Java and C++ classes.

4.4.4 Working Example

The screenshot below is taken from an **HTC Desire HD** phone running Android 2.2. It shows the Sangwin Engine displaying three cubes, each with a material and one with a texture map.

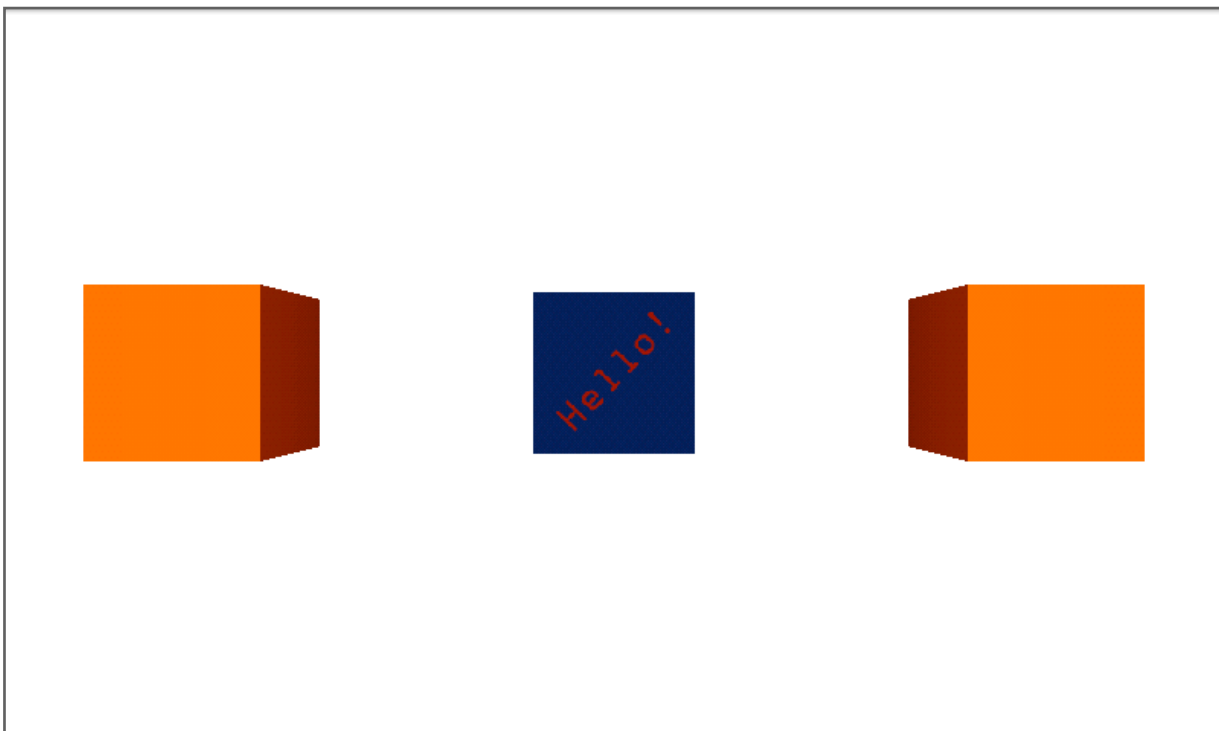


Fig 22. Android Example Screenshot

4.5 An Example Game

A small example game was built to show the different parts the Android C++ engine functionality working in a typical scenario. The game is very basic, but highlights resource loading, event handling, collision detection and rendering. Designs for the example game can be found in **Appendix 8.D**, and some screenshots of the game are shown below.

Due to the constant updating of the screen, some of the images are torn, and are not representative of the actual game. They do, however, give a rough representation of the game in progress.

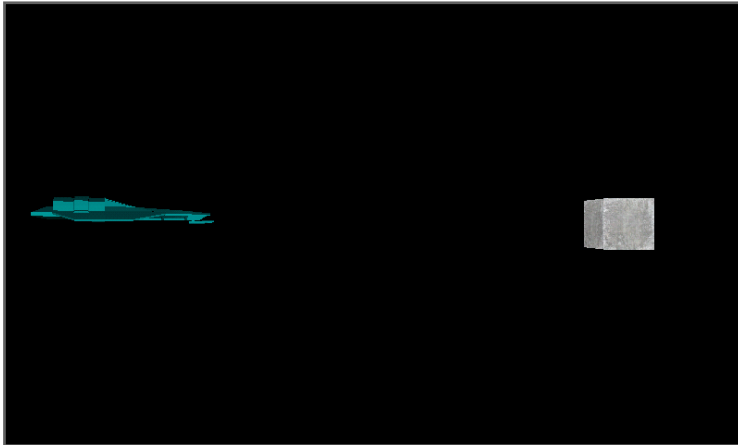


Fig 23. Game at startup, showing spaceship and target



Fig 24. Bullets being fired from the ship on screen-touch



Fig 25. Bullets coming into contact with target, and changing background colour



5.1 Profiling Methods

Profiling is an extremely important part of the testing process. It involves measuring the speed, memory usage, efficiency and any other performance-related parts of a program. The resultant information shows any areas where the program is performing badly, and needs to be improved. Profiling can happen throughout the software engineering process, but is most useful when a program is almost finished, as adding, changing or removing code can greatly effect the program profile.

Performance is extremely important, as a program that satisfies all other functional requirements is useless if it cannot be run to a reasonable standard on the target system. Profiling of the Sangwin game engines focused in particular on speed. The methodologies used to achieve this are detailed in this section.

5.1.1 Profiled areas

The profiling of the engines focused on the speed of specific areas of functionality. These areas were:

1. **The main draw loop**
2. **Resource loading**
3. **Event handling**

Different experiments with a variety of inputs were performed in each area, to try and provide measurements which highlight the strengths and weaknesses of each engine. Memory usage of each engine was not examined due to time constraints. This is something that could be performed as an extension to this project.

5.1.2 Ensuring a Reliable Experiment

To ensure fair results from each profile, the following methodology was used:

- For each separate test, the results were averaged over 5 individual runs of the program.
- Each run of the program lasted 30 seconds.
- Each area consisted of multiple test with different inputs, to understand the effect of input size on the code being tested.
- The Android application was explicitly forced to stop between each run.
- All tests were made using a **HTC Desire HD** mobile device, running **Android 2.2**.

These methods helped to ensure that the profile results were fair and representative. However, there are some factors that are out of developer control, particularly the number of background processes running on the device. While this usually does not greatly affect performance, it can cause problems when background processes are performing expensive operations. Therefore,

when a result occurred that was vastly different from those of other runs in the same test, the run was discounted and restarted.

5.1.3 Android Traceview

Android Traceview is a debugging and profiling tool. It shows a visual interpretation of an application state between given start and end points. These points can be defined anywhere inside a program, allowing the developer to focus the profile on a specific area of code. The output data shows the callgraph for the time period, as well as the time taken for function calls and the amount of calls for each function.

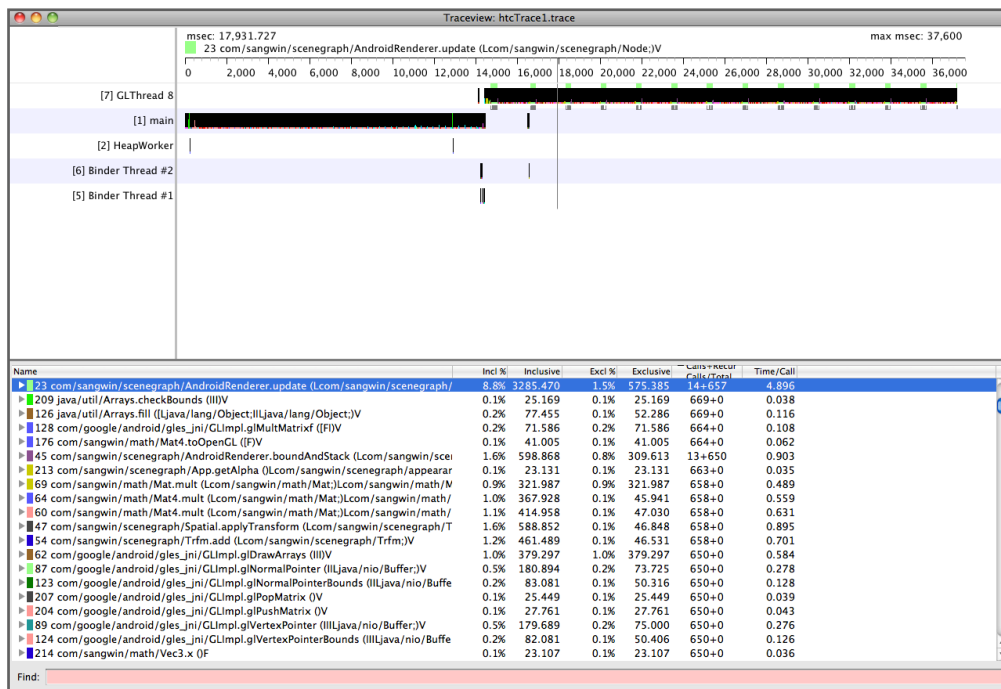


Fig 26. Android Traceview GUI

This tool would have been extremely useful for profiling the Sangwin engines, but unfortunately it could not be used for comparison. This is because the attached profiler affects the speed of the main Java thread, but not the native thread, which makes it impossible to accurately and fairly compare the speed of a native and non-native program.

The Traceview tool was still used to find the most frequently called and time consuming functions, and this is discussed further in the **Results section**.

5.1.4 Alternative

As Traceview could not be used, an alternative profiling solution was devised. This solution makes use of the Java `System.currentTimeMillis()` and `System.nanoTime()` methods. These methods retrieve the time of the system clock at the time of calling. By calling the method before and after a code block, the time taken to run the block can be found by finding the difference between each time. To increase the reliability of the result, these results can be average over many iterations of the code block.

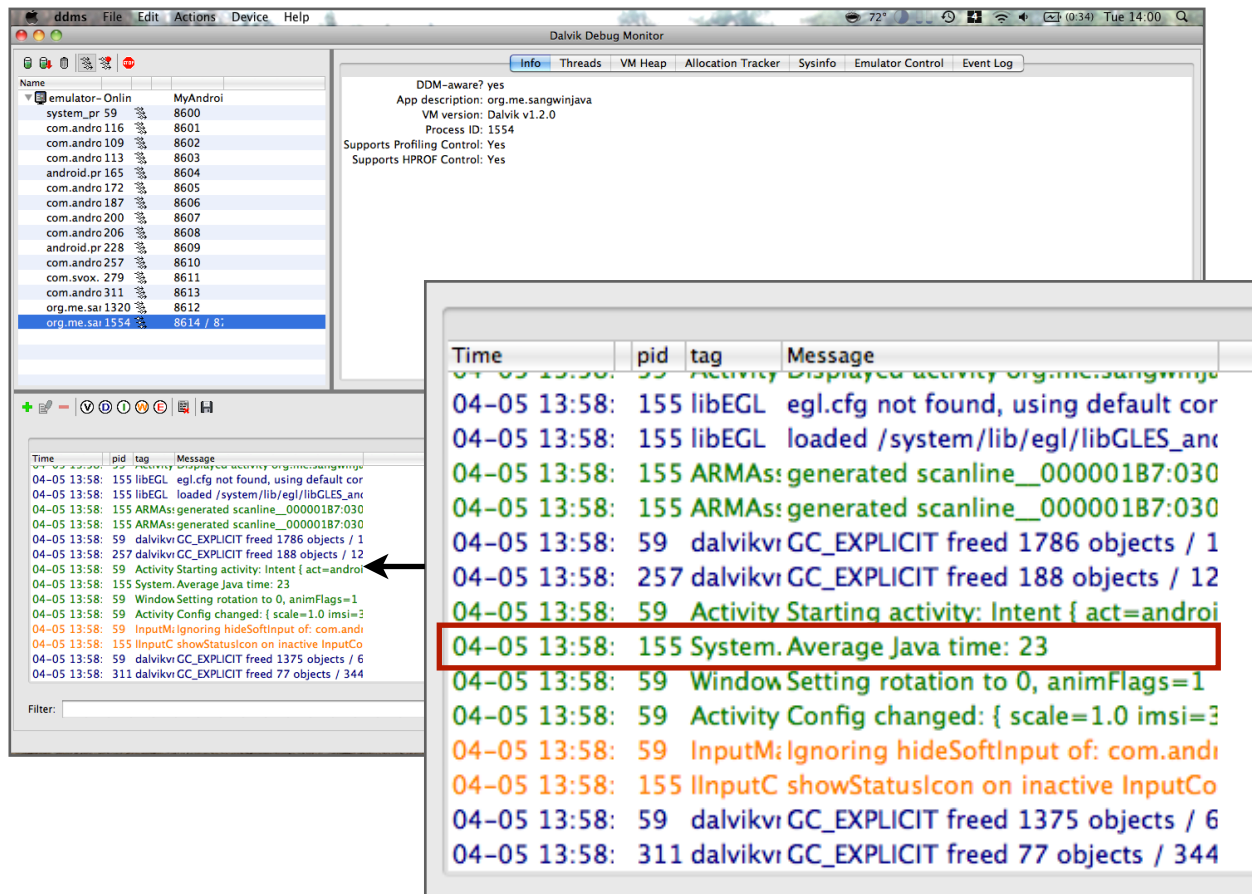


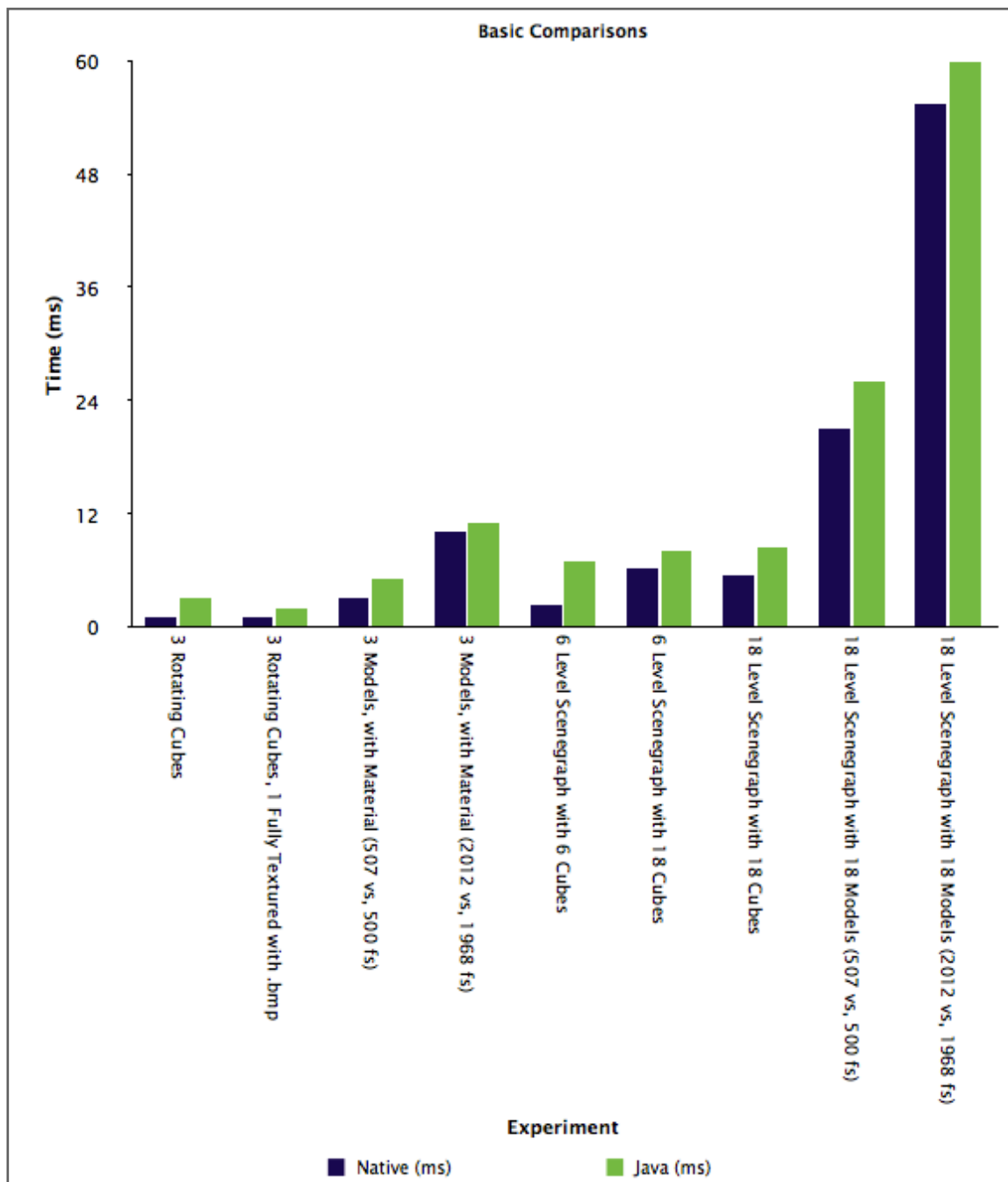
Fig 27. Android DDMS and Log view, highlighting profile output

The desired profile information was viewed using the Android **DDMS** tool. This is a debugging tool which lets the developer view information about the current device and application state. It includes a log output view which can be written to using either the Android log functionality or the Java `System.out...` methods.

5.2 Results

5.2.1 Basic Comparisons

The first set of comparisons focused on general, low load experiments across a range of inputs. This was to get an initial feel for the engine differences and fine tune the profiling methods. The tests were low granularity, and used the `System.currentTimeMillis()` method. The experiment data (and data for all other experiments) can be found in **Appendix 8.C**.



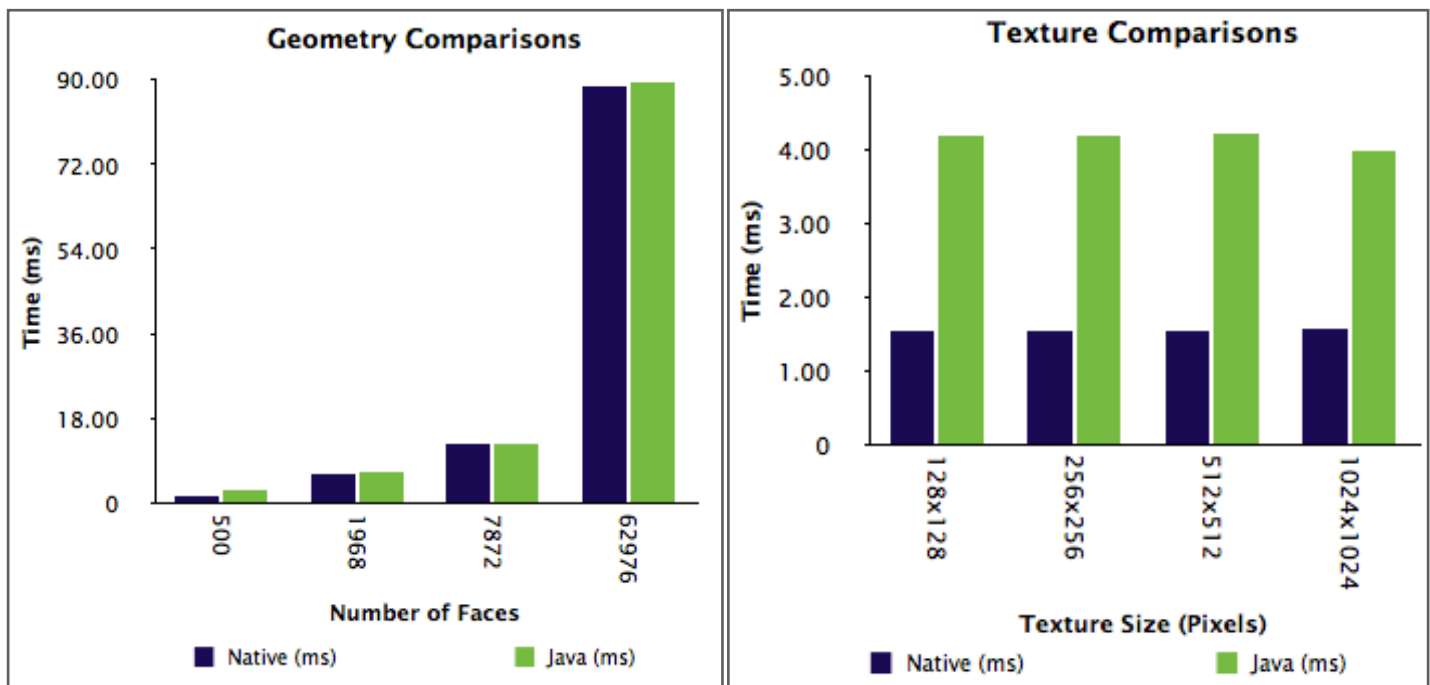
These results showed that the C++ engine was faster than the Java engine for all low load input, by roughly 10-20%.

5.2.2 onDrawFrame()

The `onDrawFrame()` method performs the update of the game loop and the scenegraph rendering, making it the most important function in the engine. It was profiled in both engines from Java, so that **a)** the experiments remained fair and **b)** the JNI overhead would be taken into account in the native code. After the **Basic Comparisons**, higher granularity tests were performed by using the `System.nanoTime()` method, which gives results in nanoseconds. This showed more accurate differences in performance. The results were converted to milliseconds after the averages were taken to give a more readable comparison.

Geometry and Textures

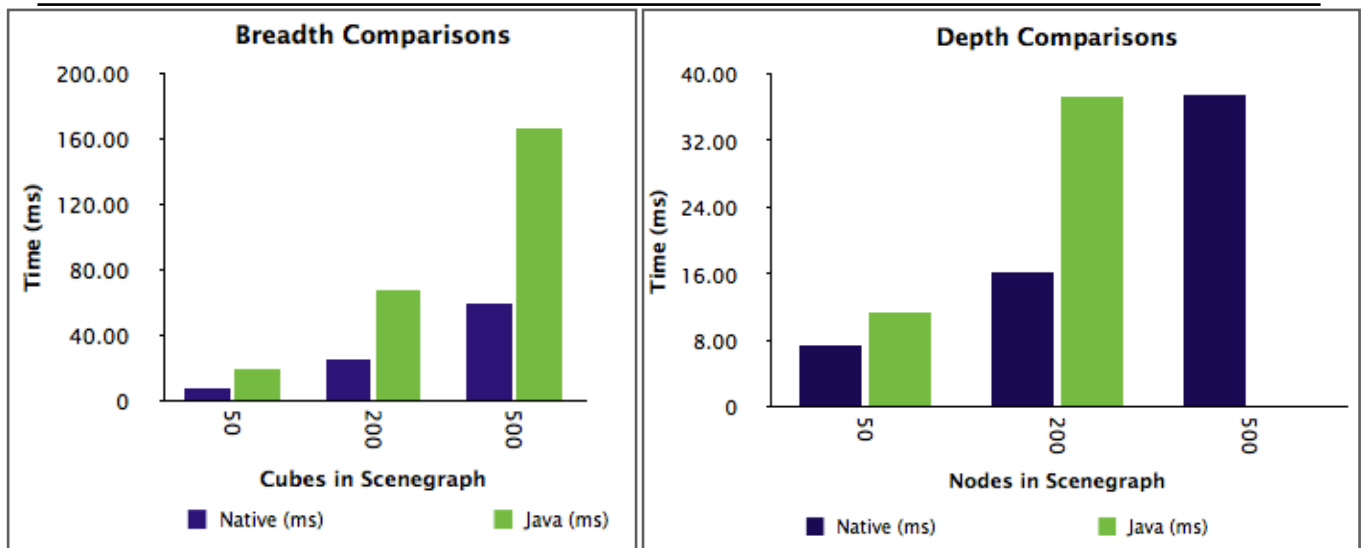
The geometry experiment focused solely on large numbers of faces, to put pressure on the OpenGL functionality of each engine. The experiment shows that the use of OpenGL results in almost the same performance whether used natively or through Java. This is to be expected, as the Java OpenGL implementation is simply a wrapper around C OpenGL functionality using JNI.



The texture experiment examined the differences in the speed of texture rendering of each engine, as the renderers use different OpenGL functions to render textures. The results showed that C++ was faster than Java, regardless of texture size.

Scenegraph Size

Scenegraphs of varying size were tested, to examine the effects of a large number of objects on engine performance. The scenegraph was populated both in a breadth (one node with many children) and depth (many nodes, each with two children) format, using automated methods. Low polygon geometry was used to minimise the effects of graphics rendering and focus entirely on the scenegraph.



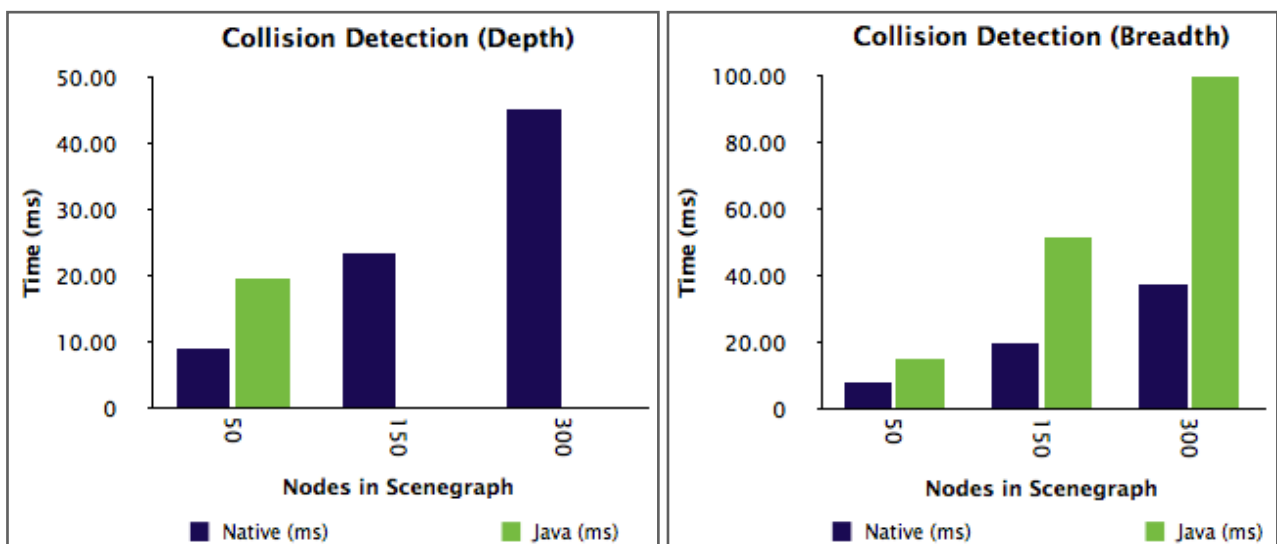
These experiments showed vast differences in performance between the two engines, with the C++ engine being around 3 times faster at high load. This suggests that native code handles expensive recursive operations far better than JIT compiled Java code. This is interesting, and highlights the benefits of using static compilation when many performance-critical, repetitive operations are required.

It should also be noted that Java threw a stack overflow error when populating the scenegraph with a high number of nodes, which shows that native stack is allocated more space than the Java stack on Android.

Collision Detection

Similar scenegraph experiments to those above were performed with inclusion of the collision detection algorithm. This algorithm requires another scenegraph traversal every frame, and is therefore expensive. The results were as expected, with collision detection having a negative effect on performance in both engines, but with a bigger hit shown in the Java engine.

When performing the depth-based comparisons, some Java results could not be obtained as the program threw a stack overflow error when populating the scenegraph.

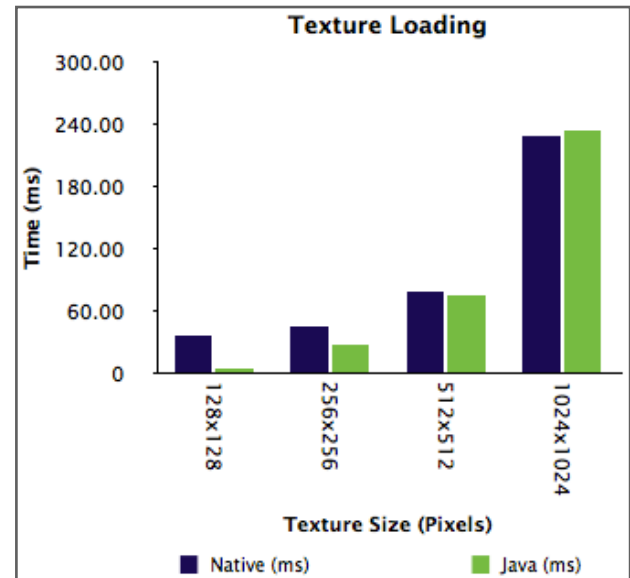


5.2.3 Resource Loading

Both the loading of models and textures were profiled using the same, high granularity method as with `onDrawFrame()`. The results are as follows.

Texture Loading

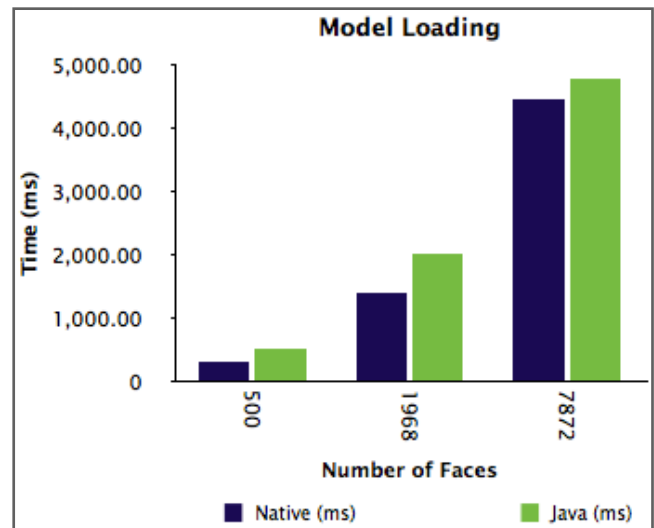
The time to load textures of a range of dimensions was measured. The results were interesting, and hard to interpret. They showed that for small textures, the Java engine was noticeably faster than the C++ engine, but for larger textures the performance was equal. This is most likely caused by the extended time taken to retrieve the image data from large files outweighing the extra calculation needed to handle images in the C++ engine.



Model Loading

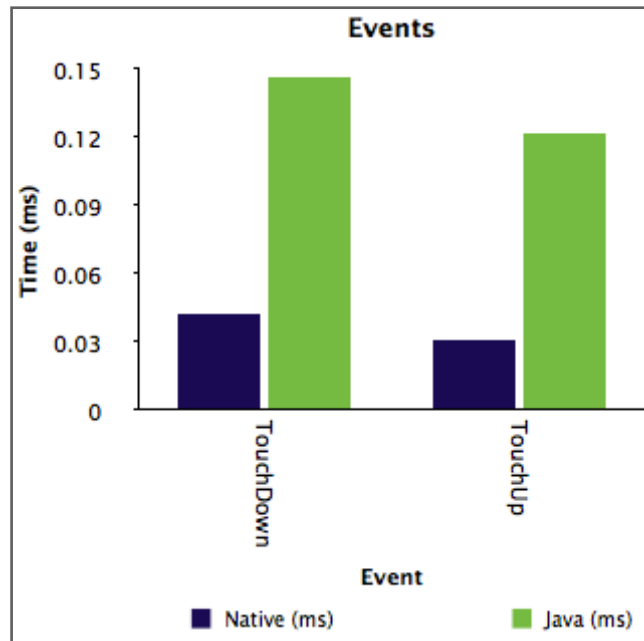
.obj files of a range of sizes were loaded and profiled. The resultant graph shows that model loading in the C++ engine was faster than the same behaviour in the Java engine. However, it can be seen that the relative speed of the Java engine improves as files get larger. This is perhaps due to the JIT compiler making increasingly efficient use of the code cache.

Loading models with textures and materials was not profiled due to time constraints, and could be examined as an extension to this project.



5.2.4 Event Handling

The events system was difficult to profile. The results were very erratic, and so it was hard to see the differences between the engines. This isn't much of a problem, as UI events happen very infrequently in a typical run of a game. The results graph is shown here regardless, and the data can be further examined in **Appendix 8.C**.



5.3 Optimisations

Primary Requirement 4 states that optimisations will be made to the original Java engine wherever possible. This section describes the attempts in optimisation that were made to the Sangwin codebase.

C++ optimisation is a large and well researched field, and discussing it in detail is beyond the scope of this report. *C++ For Game Programmers* was used to find a number of useful game engine optimisation techniques^[12]. Some of these, such as cache alignment and heap organisation, were too in depth to attempt within the given timeframe. Instead, smaller, more encapsulated methods were used. These included the following:

- **Inlining**

Inlining replaces a function call with the actual code the function contains at compile time. This is useful for small functions where the overhead of making the function call is as much as the time to perform the actual operation.

- **Removal of variables from loops**

Removing variables from loops reduces the amount of calculation required in each iteration. In some cases with very large loops that get called frequently, this can greatly improve performance.

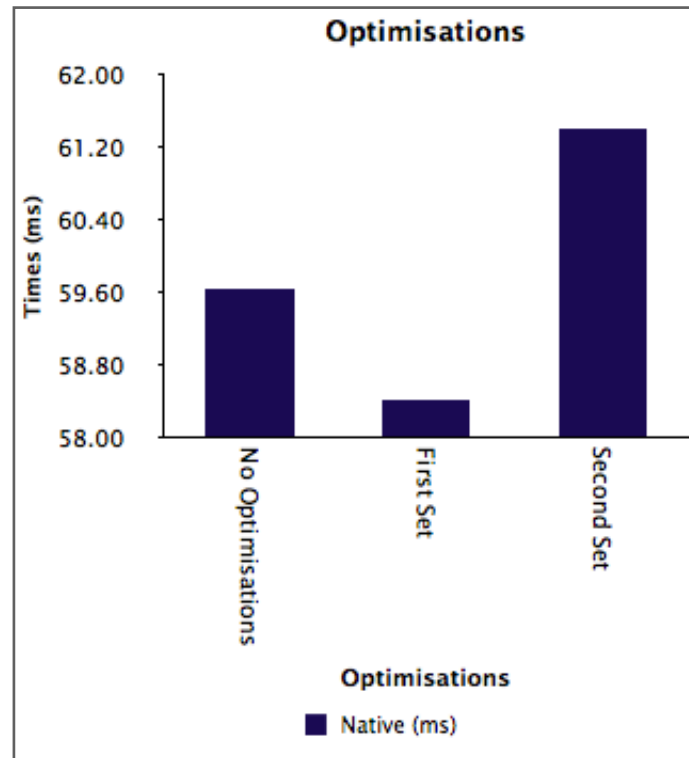
- **Good use of global variables**

Global variables can improve speed for two reasons. The first is that they remove the overhead associated with accessing variables through an object, and the second is that they are stored in a separate place in memory, making them less susceptible to heap fragmentation.

- **Initialisation Lists**


Initialisation lists ensure that hidden constructor calls are not made unnecessarily by C++. This can improve performance where objects are part of a large hierarchy and constructing can be expensive.

Initially, Android Traceview was used with the Java engine to find the functions that were called the most often or had the largest call times during a typical game run. Each function was examined and optimised using the above techniques wherever possible. After this, all of the base engine code was examined and optimised. Profiles were made after both sets of optimisations using a scenegraph populated in a breadth style with 500 nodes. The results were compared against the same scenario with no optimisation.



After examination, it was discovered that the original Sangwin code was already well optimised. However, there were some areas that still had room for changes. After the first round of optimisations, the code was improved by around 2% per frame. This was not a large improvement, but could be useful to squeeze the most out of the engine at high load.

After the second set of optimisations, the performance of the engine actually dropped by around 5%. This was most likely due to inefficient use of the code cache caused by inlining^[13]. This is one of the issues that can occur with over optimisation, and is a trade off that needs to be taken into consideration when programming in C++.



6. Evaluation & Conclusions

6.1 Critical Evaluation

This section aims to evaluate the project in terms of the initial requirements. The quality of the Sangwin C++ engine will be discussed in comparison to the original Java version, with a focus on performance and language design. The evaluation will also highlight any difficulties encountered during the project.

Primary Requirement 1 of this project stated that a number of new technologies should be learnt, in order to successfully translate the existing software. All of the required technologies were learnt to a standard which meant code could be created successfully. Almost all of the technologies were easy to get to grips with after enough background research, but C++ proved very challenging to learn. Code constantly had to be rewritten as more of the language was understood, which resulted in the translation taking far longer than expected.

Despite this, the translation was a complete success. All elements of **Primary Requirement 2** were achieved, including the removal of all Android code from the base engine library. This means that the engine is ready to be ported to other C++ platforms in future, including Apple's iPhone. Some sacrifices in usability had to be made, particularly in the loading of resources and the handling of events, but this was to be expected with the use of the JNI.

The engine was also transformed into a usable API. It was condensed into a single library for use by the developer, and extensive documentation was generated using the **Doxygen** tool. This was further helped with the inclusion of the example game, which works as an excellent tutorial for developers learning how to use the engine. Both of these elements mean that **Secondary Requirements 1 and 2** were achieved.

Language performance was a key focus of this project (described in **Primary Requirement 3**). Each of the engines were profiled extensively to assess their differences. The results showed a definitive improvement through the use of C++, even with the negative effects of the JNI. The Sangwin C++ engine was up to 3 times faster than the Java engine, and was never slower. Geometry handling performance differences were minimal as the OpenGL implementations are the same in both languages, but C++ showed vast improvements in function call overhead and recursion.

Optimisation was not very successful. The Sangwin Java engine was already well optimised, and so areas of improvement were hard to find. More research was needed into the behaviour of the Java engine prior to starting the project, and more knowledge in the field of C++ performance was required to understand the scope of optimising code using the language. This also meant that memory could not be investigated and **Secondary Requirement 3** could not be achieved. However, as the performance of C++ was better than that of Java without optimisation, optimisation was not as important as initially thought, and so **Primary Requirement 4** was not necessary for the other requirements to be successful.

Primary Requirement 3 was also concerned with ease of language use. This is important as improved performance is made largely redundant if a language is impossible to use. Translating the engine into C++ showed that the language is much more unforgiving than Java, as errors can be very hard to track down for an inexperienced user of the language. Also, with the NDK version used in this project the support for C++ was lacking, which meant that a lot of extra work had to be done to build a working version of the engine.

Despite these issues, as the language became more familiar it became far easier to program in. Particularly after the foundation work was finished, programming with C++ became comparable to Java. With the inclusion of managed memory through smart pointers and `typedefs` for complex types, the C++ engine can be used almost exactly like the Java one, aside from the restrictions the JNI imposes. The latest NDK revision also means that missing features (like RTTI and Exceptions) can easily be added to the codebase, making it even easier for developers to use.

The only major remaining complication is the added complexity of using two languages rather than one, and having to interface between them using the JNI. As described in **section 4.4**, the use of the JNI has been kept to a minimum, and the resulting framework does not differ greatly from that of the pure Java engine. Ultimately, it is up to the developer to decide whether the performance increases are worth the extra effort. As much has been done as possible to ensure that this added work is as straightforward as it can be.

6.2 Extensions and Further Work

There are a number of opportunities for extension with this project. The first is in the area of optimisation. Some complex optimisation methods were researched that were beyond the scope of this project, and implementing them could have a positive effect on performance. These are mostly concerned with cache alignment and memory management. Heap fragmentation is a common problem with C++, and so finding a way making more efficient use of the memory is important^[14]. Combined with effective cache use, a good memory management system can greatly improve the overall performance of a program.

The Desktop engine built as part of the project could be extended into a visual scenegraph editor for games. A developer could be able to drag and place items where they like in the scenegraph, and then save the generated code for use in a game. As discussed in the **Research and Technologies** section, visual editors are commonplace in modern game engines, and so the inclusion of one would help make the Sangwin Engine a far more attractive package for developers.

Finally, the existing functionality of the Sangwin engine could be extended. The performance improvements made through the use of C++ could be used to add functions such as animation, which require a large amount of calculation. Improvements like this would further help to strengthen the argument for C++ over Java when developing for Android.

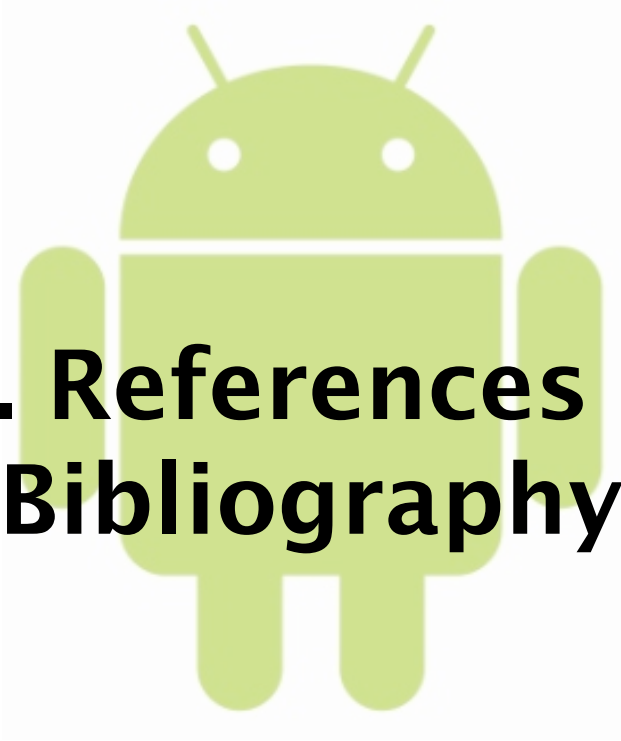
6.3 Concluding Summary

The Sangwin C++ translation was a complete success. The functionality from the pre-existing Java engine was ported to C++, and the resultant software can be used effectively by new and existing developers alike due to its design and similarity to its Java counterpart. A single, binary library was created for developers to use with minimal difficulty, and extensive documentation and an example game were created so that developers can easily make use of all of the engine functionality.

The translation allowed a number of challenges and difficulties to be discussed, and a critique on the use of the NDK to build native code for the Android OS was formed. The critique focused in particular on the use of the NDK in game development, and how it can be used to create a relatively straightforward and easy to use, high-performance game engine. This was presented in the form of a design document which highlighted particularly complex areas in the development cycle.

A number of comparisons were also made between C++ and Java, with the benefits of each language being highlighted. It was found that C++, when used correctly, can provide the same semantic behaviour of Java whilst vastly improving performance of the resultant application. Some shortcomings were also found with C++, and these were highlighted throughout the report.

Ultimately, the project provided an interesting insight into 3D video game development for Android, which can hopefully be used by developers looking to make high-performance games for the platform.



7. References & Bibliography

7.1 References

- [1] Sedgewick, R. (1992) *Algorithms in C++*. 1st edn., pp 231-243. Addison-Wesley.
- [2] *Android.com*
Available from:
<http://www.android.com/>
(Accessed 02/05/2011)
- [3] *Android Developers: The Activity Lifecycle*
Available from:
<http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle>
(Accessed 02/05/2011)
- [4] *AndEngine: About*
Available from:
<http://www.andengine.org/blog/about/>
(Accessed 02/05/2011)
- [5] *AndEngine: Free Android 2D OpenGL Game Engine*
Available from:
<http://www.andengine.org/blog/>
(Accessed 02/05/2011)
- [6] *Android, Netbeans, and the Assets Directory*
Available from:
<http://blog.yetisoftware.com/2009/04/02/android-netbeans-and-the-assets-directory/>
(Accessed 02/05/2011)
- [7] *BCS Code of Conduct*
Available from:
<http://www.bcs.org/category/6030>
(Accessed 02/05/2011)
- [8] *BCS Code of Good Practice*
Available from:
<http://www.bcs.org/category/6031>
(Accessed 02/05/2011)
- [9] *Boost: Smart Pointers*
Available from:
http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/smart_ptr.htm
(Accessed 02/05/2011)
- [10] Dickheiser, M. J. (2007) *C++ For Game Programmers*. 2nd edn., pp 16-19. Cengage Learning.
- [11] p 31.
- [12] pp 105-173.
- [13] pp 117-118.
- [14] pp 143-147.
- [15] *CG Textures*
Available from:
<http://www.cgtextures.com/>
(Accessed 02/05/2011)

- [16] *gluPerspective() Implementation*
Available from:
<http://bieh.net/svn/androidgl/src/androidgl.cpp>
(Accessed 02/05/2011)

- [17] *Introducing Android 1.5 NDK, Release 1*
Available from:
<http://android-developers.blogspot.com/2009/06/introducing-android-15-ndk-release-1.html>
(Accessed 02/05/2011)

- [18] *Java's Security Architecture - Javaworld*
Available from:
<http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html>
(Accessed 02/05/2011)

- [19] *More Effective C++ Item 29 Source Code*
Available from:
<http://www.aristeia.com/BookErrata/M29Source.html>
(Accessed 02/05/2011)

- [20] *NDK, OpenGL - Loading Resources from Native Code*
Available from:
http://www.anddev.org/ndk_opengl_-_loading_resources_and_assets_from_native_code-t11978.html
(Accessed 02/05/2011)

- [21] *Sangwin Games*
Available from:
<http://www.sangwingames.com/>
(Accessed 02/05/2011)

- [22] *ShiVa 3D: Game Engine with Development Tools*
Available from:
<http://www.stonetrip.com/>
(Accessed 02/05/2011)

- [23] Meyers, S. (1996) *Smart Pointers, Part 3*
Available from:
<http://www.aristeia.com/Papers/C++ReportColumns/sep96.pdf>
(Accessed 02/05/2011)

- [24] *Turbosquid - Free Wipeout Feisar Ship Model*
Available from:
<http://www.turbosquid.com/3d-models/free-wipeout-ship-feisar-3d-model/493665>
(Accessed 02/05/2011)

- [25] *Unity: Game Development Tool*
Available from:
<http://unity3d.com/unity/>
(Accessed 02/05/2011)

- [26] *Using NDK r4 with the Android SDK in Netbeans*
Available from:
http://www.jondev.net/articles/Using_NDK_r4_with_the_Android_SDK_in_Netbeans
(Accessed 02/05/2011)

7.2 Bibliography

Android Developers: Android SDK

Available from:

<http://developer.android.com/sdk/index.html>

(Accessed 02/05/2011)

Android Developers: Profiling with Traceview and dmtracedump

Available from:

<http://developer.android.com/guide/developing/debugging/debugging-tracing.html>

(Accessed 02/05/2011)

Android Developers: What is that NDK?

Available from:

<http://developer.android.com/sdk/ndk/overview.html>

(Accessed 02/05/2011)

C/C++ Pack Test Specification for C/C++ Unit Tests

Available from:

http://wiki.netbeans.org/TS_69_CNDUnitTests

(Accessed 02/05/2011)

Hahn, B. (1994) *C++: A Practical Introduction*. 1st edn., Massachusetts, Blackwell.

Dickheiser, M. J. (2007) *C++ For Game Programmers*. 2nd edn., Cengage Learning.

The C Book - Character Handling

Available from:

http://publications.gbdirect.co.uk/c_book/chapter5/character_handling.html

(Accessed 02/05/2011)

cplusplus.com - The C++ Resources Network

Available from:

<http://www.cplusplus.com/>

(Accessed 02/05/2011)

Creating UNIX Libraries

Available from:

<http://www.cs.duke.edu/~ola/courses/programming/libraries.html>

(Accessed 02/05/2011)

Doxygen

Available from:

<http://www.stack.nl/~dimitri/doxygen/index.html>

(Accessed 02/05/2011)

Meyers, S. (2010) *Effective C++*. 3rd edn., Addison-Wesley.

Games from Within | C++

Available from:

<http://gamesfromwithin.com/category/c>

(Accessed 02/05/2011)

Cormen, T. H., Leiserson, C. E., Rivest, R. L. (1990) *Introduction to Algorithms*. 1st edn., Massachusetts, MIT Press.

The Java Native Interface Programmer's Guide and Specification

Available from:

<http://java.sun.com/docs/books/jni/html/jniTOC.html>

(Accessed 02/05/2011)

Linux Tutorial: POSIX Threads

Available from:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

(Accessed 02/05/2011)

The Native Android API

Available from:

<http://mobilepearls.com/labs/native-android-api/>

(Accessed 02/05/2011)

Neon Helium - OpenGL Tutorials

Available from:

<http://nehe.gamedev.net/>

(Accessed 02/05/2011)

Online Matrix Calculator

Available from:

<http://www.bluebit.gr/matrix-calculator/>

(Accessed 02/05/2011)

OpenGL SDK Documentation

Available from:

<http://www.opengl.org/sdk/docs/>

(Accessed 02/05/2011)

Porting Applications Using the Android NDK

Available from:

<http://www.aton.com/porting-applications-using-the-android-ndk/>

(Accessed 02/05/2011)

Qt - Cross-platform application and UI Framework

Available from:

<http://qt.nokia.com/>

(Accessed 02/05/2011)

Singleton Design Pattern in C++

Available from:

http://sourcemaking.com/design_patterns/singleton/cpp/2

(Accessed 02/05/2011)

Wesley's Techblog >> Qt

Available from:

<http://wesley.vidigatch.org/category/programming/qt/>

(Accessed 02/05/2011)

8. Appendices



8.A Sangwin Engine Analysis

Below is the analysis of the Sangwin Engine, performed prior to the design and implementation stage. Included are details of each of the Java packages, with brief descriptions and high level class diagrams showing inheritance hierarchies.

The Math Package

The Math Package holds all of the classes that deal with Vector and Matrix mathematics in 2D and 3D space.

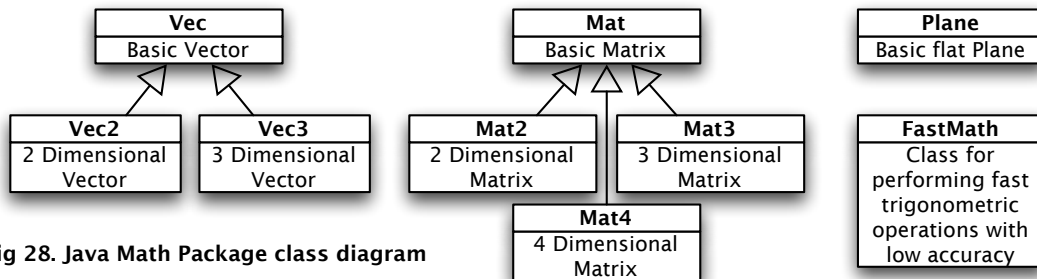


Fig 28. Java Math Package class diagram

The Lighting Package

The Lighting Package consists of a number of wrapper classes around OpenGL lighting functionality.

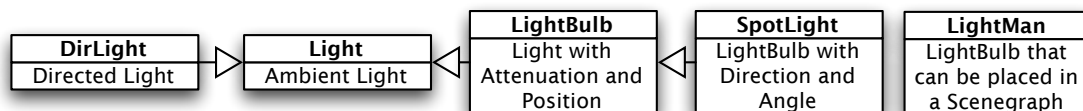


Fig 29. Java Lighting Package class diagram

The Appearance Package

The Appearance Package consists of a number of wrapper classes around OpenGL texture and material functionality.

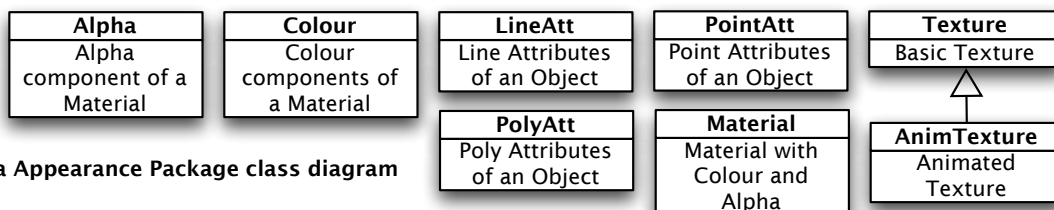


Fig 30. Java Appearance Package class diagram

The Camera Package

The Camera Package holds a number of camera classes for use in 2D and 3D scenes.

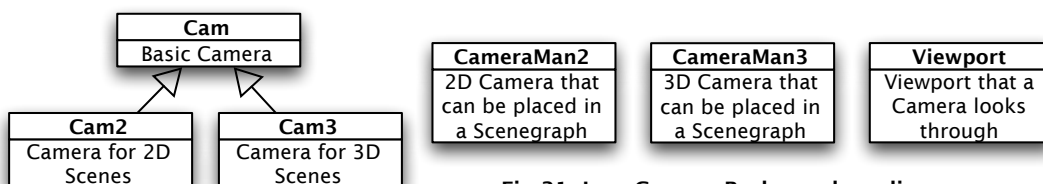


Fig 31. Java Camera Package class diagram

The Scenegraph Package

The Scenegraph Package holds all of the classes for building both 2D and 3D scenegraphs. It also contains the `AndroidRenderer`, and classes for creating and transforming meshes. An in depth

description and high level class diagram of this scenegraph and scenegraphs in general is included in **section 3.4**.

The Collision Package

The Collision Package contains classes to build bounding volumes around scenegraph nodes. Bounding volumes are simple way of performing collision detection, and are talked about in greater detail in **section 3.4**.

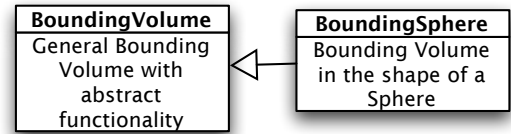


Fig 32. Java Collision Package class diagram

The Resources Package

The Resources Package handles the loading in of textures and 3D models to the game engine. Textures of any acceptable image format are allowed, and 3D models in the .obj file format.

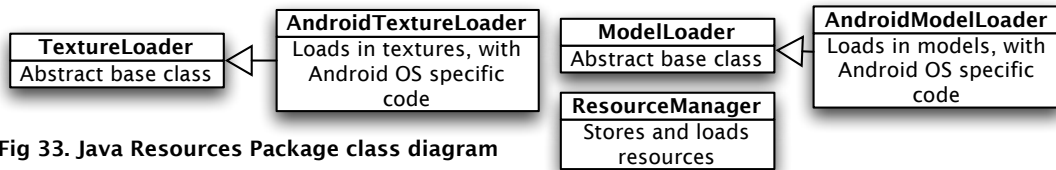


Fig 33. Java Resources Package class diagram

The Events Package

The Events Package deals with user input events and events from within the engine. These are managed by an EventHandler, which performs actions when a specific event occurs. Also, it provides a TaskScheduler which can perform tasks at regular intervals, or when an event or action from the EventHandler occurs.

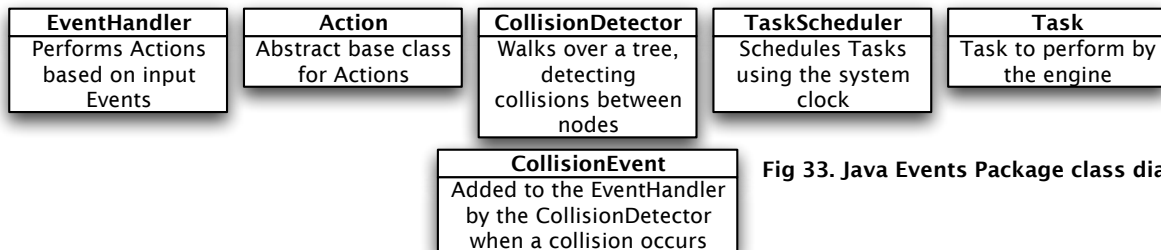


Fig 33. Java Events Package class diagram

The Game2D Package

The Game2D Package provides classes which are used specifically in 2D games.

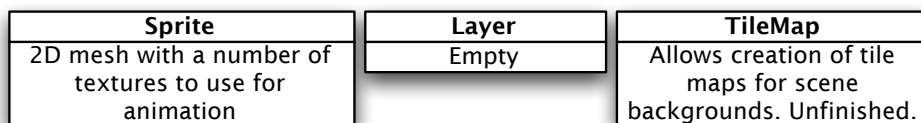


Fig 34. Java Game2D Package class diagram

Untranslated code

Noted here are the classes that have been left out of the above list, as they were not translated as part of this project.

TaskScheduler and Task

These classes were not translated, as they were not required to compare the differences between the two projects. They were initially going to be ported at the end of the translation, but due to time constraints they were left out.

- **AnimTexture**

This class was also left out as it was not required for comparison.

- **The Game2D Package**

This package was not translated as it was unfinished in the original engine. Also, the final renderer that was built focused solely on 3D games, and so these classes would have remained untested even if they had been translated.

8.B Test Results

This Appendix shows each of the individual tests performed on both the Desktop and Android versions of the C++ engines. Tests are grouped into categories, based on the area of functionality they were testing.

8.B.1 Desktop Tests

Camera

Test Case	Expected Result	Actual Result	Passed	Solution
Camera holds perspective when canvas size is changed	Perspective stays the same	Perspective changed	No	Boolean value for canvas changed not checked in renderer - added and fixed
Camera position updates when setPos(x,y,z) is called	Camera changed position	Camera changed position	Yes	N/A

Attaching Nodes

Test Case	Expected Result	Actual Result	Passed	Solution
Node::attachChild() is successful	Child attached to parent node	Random behaviour	No	Parent pointer not initialised in Node - fixed
Node::removeChild() is successful	Child is removed from parent node	Child removed from parent node	Yes	N/A
Trees of nodes can be built	Attach child at multiple levels works	Attach child at multiple levels works	Yes	N/A

Node Transformation

Test Case	Expected Result	Actual Result	Passed	Solution
get/updateLocalTransform() called on leaf node	Transformation affects single node	Transformation affects single node	Yes	N/A
get/updateLocalTransform() called on parent node	Children change when parent is modified	Children change when parent is modified	Yes	N/A

Node Appearance

Test Case	Expected Result	Actual Result	Passed	Solution
getLocalAppearance() called on node to modify Material, Material is set to override	Material of node modified	Material of node modified	Yes	N/A
updateLocalAppearance() called on parent node to modify Material	Material of node and its children is modified	Changed on parent, but not on children	No	Material needed to be overridden on parent node - fixed

Lighting

Test Case	Expected Result	Actual Result	Passed	Solution
addLight() called on node	Light added to scene	Light added to scene	Yes	N/A
addLight() called on node with children	Children meshes lit correctly	Children meshes lit incorrectly	No	Normals not added to static meshes in Trimesh class - added and fixed

8.B.2 Android Tests

Basic Setup

Test Case	Expected Result	Actual Result	Passed	Solution
onSurfaceCreated()	Game initialises all data properly	Game initialises all data properly	Yes	N/A
onDrawFrame()	Game redraws successfully on every frame	Game redraws successfully on every frame	Yes	N/A

Textures Usage

Test Case	Expected Result	Actual Result	Passed	Solution
Android_Bitmap_lockPixels called on Java Bitmap object	Bitmap pixel data stored in void** array	Bitmap pixel data stored in void** array	Yes	N/A
Textures are rendered correctly	Textures display with the same colour and dimension as the original file	Colours wrong, texture displayed twice at a quarter of the size, random pixel data shown also	No	Texture format in renderer generateTexturePointer() incorrect. Changed to GL_UNSIGNED_SHORT_5_6_5 to match RGB565 format of Bitmaps- fixed
Textures are displayed correctly on cube Trimesh	Textures have correct orientation on every face	Some faces correct, others not	No	Texture coordinates in cube Trimesh are incorrect - NOT FIXED
.png and .bmp Texture formats used	Both image formats display correctly	Both image formats display correctly	Yes	N/A

Model (.obj) Usage

Test Case	Expected Result	Actual Result	Passed	Solution
Model data extracted from file in loadOBJ()/loadMTL()	File data is intact in C++	File data is intact in C++	Yes	N/A
getNextTokenOBJ()/MTL() called on whole file, token data is printed	Tokens all print out correctly	Some tokens printed, others not, application crashes	No	strcpy() used instead of manually copying string data in lexical analyser. All strings correctly terminated with "\0" - fixed
Models displayed correctly onscreen	All faces displayed, normals and tex coords accurate	Last face missing	No	break commands missing in switch statement in OBJLoader::loadModel() - fixed

loadMTL()	Material loaded correctly and added to model	Material loaded correctly and added to model	Yes	N/A
loadModel() in Java loads a complete model from files in a directory	Model loads and displays correctly to screen	Application crash	No	Wrong filename given in C++ to access model, directory name must be included - fixed

User Input Events

Test Case	Expected Result	Actual Result	Passed	Solution
UI Event from Java created and passed to C++	Event added to EventHandler and any Actions listening for it are performed	Nothing happens	No	EventHandler::update() not called in draw loop - fixed

Collision Detection

Test Case	Expected Result	Actual Result	Passed	Solution
Two cubes with bounding volumes come into contact with each other	Collision events added to EventHandler in correct order (ENTER, COLLISION, EXIT)	Infinite collisions registered, bounding volumes stuck at origin and not updating position	No	Virtual keyword missing in Mat::mult(Vec& vec) function meaning that wrong version of the function was being called - fixed

8.C Profiling Data

This Appendix contains the profiling data gathered for each experiment. Only the averaged results are shown. The raw data can be viewed on request.

Basic Comparisons

Input	Native (ms)	Java (ms)
3 Rotating Cubes	1	3
3 Rotating Cubes, 1 Fully Textured with .bmp	1	2
3 Models, with Material (507 vs, 500 fs)	3	5
3 Models, with Material (2012 vs, 1968 fs)	10	11
6 Level Scenegraph with 6 Cubes	2.2	7
6 Level Scenegraph with 18 Cubes	6.2	8
18 Level Scenegraph with 18 Cubes	5.4	8.4
18 Level Scenegraph with 18 Models (507 vs, 500 fs)	21	26
18 Level Scenegraph with 18 Models (2012 vs, 1968 fs)	55.4	60

Geometry Comparisons

Number of Faces	Native (ms)	Java (ms)
500	1.60	2.78
1968	6.00	6.51
7872	12.39	12.70
62976	88.33	89.43

Texture Comparisons

Texture Size (Pixels)	Native (ms)	Java (ms)
128x128	1.54	4.18
256x256	1.53	4.17
512x512	1.53	4.22
1024x1024	1.57	3.97

Scenegraph Breadth Comparisons

Cubes in Scenegraph	Native (ms)	Java (ms)
50	7.80	18.49
200	25.14	67.06
500	59.63	166.37

Scenegraph Depth Comparisons

Nodes in Scenegraph	Native (ms)	Java (ms)
50	7.29	11.40
200	16.14	37.31
500	37.53	Stack Overflow

Collision Detection - Breadth Comparisons

Cubes in Scenegraph	Native (ms)	Java (ms)
50	8.92	19.55
150	23.35	Stack Overflow
300	45.14	Stack Overflow

Collision Detection - Depth Comparisons

Nodes in Scenegraph	Native (ms)	Java (ms)
50	7.68	15.04
150	19.56	51.13
300	37.21	99.70

Texture Loading Comparisons

Texture Size (Pixels)	Native (ms)	Java (ms)
128x128	35.91	4.08
256x256	44.19	26.15
512x512	78.18	74.96
1024x1024	228.36	233.51

Model Loading Comparisons

Number of Faces	Native (ms)	Java (ms)
500	309.62	492.10
1968	1,400.47	2,006.15
7872	4,344.46	4,784.45

Event Comparisons

Event	Native (ms)	Java (ms)
Touch Down	0.04	0.15
Touch Up	0.03	0.12

Optimisations - 500 Cube Breadth Populated Scenegraph

Optimisations	Native (ms)
No Optimisations	59.63
First Set	58.40
Second Set	61.40

8.D Example Game Design

The design of the example game was very straightforward. It was decided that the game must use the following functionality, in order to effectively show each part of the engine working correctly:

- Event Handling
- Resource (Model and Texture) Loading
- Collision Detection
- Rendering
- Node transformation
- Node appearance modification

The premise of the devised game was based on target shooting. A model would move up and down in a repeating pattern, with a bullet being fired from the model when the user presses the screen. If the bullet came into contact with a target, feedback would be given to the user in the form of a background colour change. This design shows all of the above functions working together. The model was downloaded as an .obj from a free website^[24], and was in the form of a spaceship. The target was textured using a texture downloaded from a different website^[15]. The basic sequence of game events is shown in the activity diagram, and screenshots can be seen in **section 4.5**.

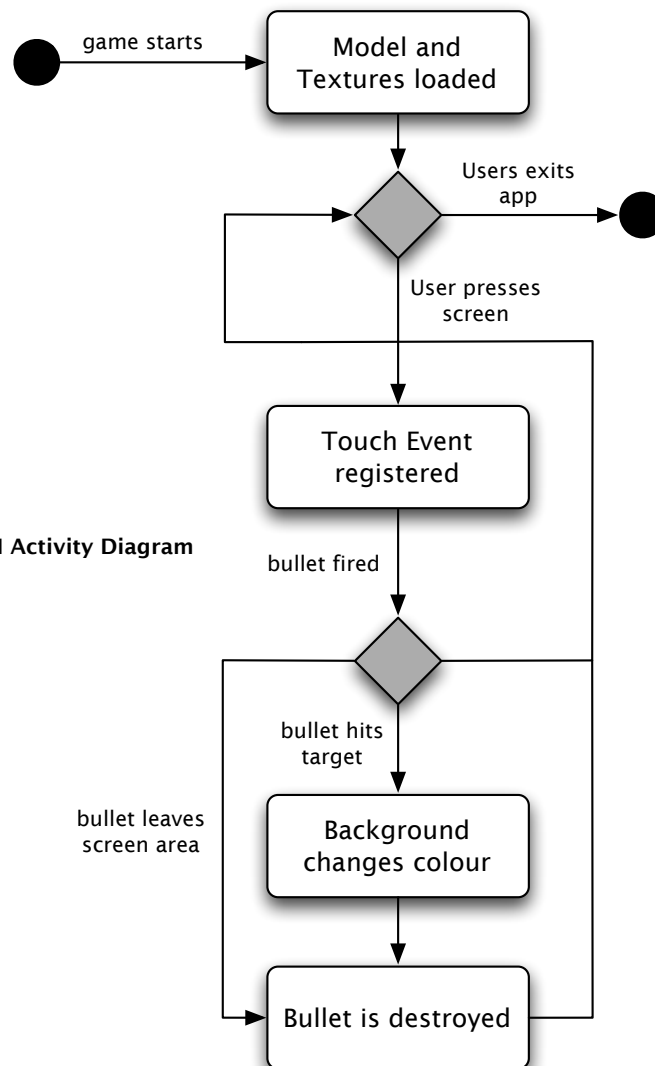


Fig 35. Example Game UI Activity Diagram

8.E Using the NDK with Netbeans

This document is included as setting up the NDK to work with Netbeans was not straightforward, and it may be useful for anyone interested in exploring native Android development.

Requirements:

- **Netbeans IDE** (downloadable from the Netbeans website)
- **NB Android Plugin** (downloadable from the Netbeans website)
- **Android SDK** (downloadable from the Android Developer website)
- **Android NDK** (downloadable from the Android Developer website)

Step-By-Step Guide

1. Set up an Android project using the NB Android plugin, as normal.
2. In the root of the directory created for the project, create a folder called 'jni'. This folder will hold all of the native code to use with the application.
3. In the `onCreate()` method of the Android `MainActivity` (or in a separate place that gets called at startup, if so desired), add in the following code:

```
static {
    System.loadLibrary("libraryname");
}
```

This code loads in the native library so it can be accessed with Java. `libraryname` is the name of the native library that will be built from the supplied native sources. Its is defined in the `Android.mk` file, described below.

4. Any native functions that are to be called from Java need to be defined. These can be defined in the `MainActivity` class, or in any class where they are going to be accessed. Native methods are defined as follows:

```
public native void nativeMethod();
```

The return type can be any type compatible with native code.

5. Native functions also need to be declared natively, and made visible to Java so that they can be called. Native functions are defined as follows (the above function is used as an example):

```
#include <jni.h>

extern "C" {

    //Declaration
    JNIEXPORT void JNICALL Java_packageName_ClassName_nativeMethod(JNIEnv*
env, jobject obj);
}
```

```
//Definition
```

```
JNIEXPORT void JNICALL Java_packageName_ClassName_nativeMethod(JNIEnv* env,
    jobject obj);
```

All functions MUST be declared in the `extern "C"` block to be made visible to Java. The file where these methods are declared and defined should be placed into the 'jni' directory created in **step 2**.

6. All of the required native code (both .h and .cpp files) should be placed in the 'jni' directory. They can be placed into sub directories, but this must be reflected in the Android.mk (described in **step 7**) file and in the source code (i.e if header files are placed into the 'includes' sub directory, they should be included with `#include "includes/myHeader.h"`).
7. The last file that needs to be created is the Android.mk file. This describes how to build the project to the NDK. The Android.mk file should be placed in the root of the 'jni' directory. An example file is show below:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := libraryname
LOCAL_SRC_FILES := sourceFile1.cpp \
                  sourceFile2.cpp \
                  subdir/sourceFile3.cpp \
                  ...
LOCAL_LDLIBS := -llog ...
include $(BUILD_SHARED_LIBRARY)
```

The `libraryname` should be the same as the one used in **step 2**. All source files, including pre-existing native code and any containing JNI export functions, should be listed.

`LOCAL_LDLIBS` lists any specific external libraries that should be included. In the example given, `log` is used, which is the library that allows the Android Log to be accessed natively.

8. There is a Netbeans specific issue that may not occur with other IDEs^[26]. The following code should be pasted into the build.xml in the project root:

```
<target name="-post-jar">
<zip update="true" destfile="${dist.apk}">
<zipfileset dir="libs/armeabi" includes="*.so" prefix="lib/armeabi"/>
</zip>
<zip destfile="tmp.apk">
<zipfileset src="${dist.apk}">
<exclude name="META-INF/*.*" />
</zipfileset>
</zip>
<move file="tmp.apk" tofile="${dist.apk}" />
<signjar jar="${dist.apk}" alias="androiddebugkey" storepass="android"
keypass="android" keystore="/Users/Username/.android/debug.keystore"/>
</target>
```

The Android debug key fields should be replaced with the correct values as required.

9. After all of the above steps have been completed, the project should be ready to be used. It should be compiled using the `ndk-build` command. To use this command, navigate to the project root from command line and enter: `path/to/ndk/ndk-build`. This should compile the native library, ready for use with the Java app. `ndk-build` should be used every time the native code needs to be compiled.

NOTE: For Mac OSX users, it may be easier to add the NDK directory to the `$PATH` variable. This can be done by manually adding the NDK directory path to the `/etc/paths` file. This means that `ndk-build` can be called without the path root.

10. As an aside, if access to Android Assets directory is required, the following code will need to be changed in the `/nbproject/project.properties` file (located near the top of the file)

```
assets.dir=
```

to

```
assets.dir=assets
assets.available=true
```

Also, in the `/nbproject/build-impl.xml` file, inside “`if=assets.available`”, there is this line:

```
<arg value="${asset.dir}"/>
```

which needs to be replaced with

```
<arg value="${assets.dir}"/>
```

After making these changes, the Assets directory should be usable, without any issues^[6].

8.F Interim Log

The Interim Log lists all of the meetings had with Sangwin and with the Project Supervisor. It also lists the major project milestones.

Date	Type	Details
Monday 4th October 2010	Project Supervisor Meeting	Discussion of initial project ideas
Tuesday 5th October 2010	Sangwin Meeting	Discussion of initial areas of work
Tuesday 12th October 2010	Sangwin Meeting	Discussion of NDA
Thursday 14th October 2010	Sangwin Meeting	Discussion of general progress and smart pointer implementation issues
Monday 18th October 2010	Project Supervisor Meeting	Discussion of general progress
Thursday 21st October 2010	Submission	Project Plan
Tuesday 2nd November 2010	Project Supervisor Meeting	Discussed start of main translation
Saturday 6th November 2010	Sangwin Meeting	Received Java code, started translation
Tuesday 9th November 2010	Project Supervisor Meeting	Discussed progress after start of main translation
Thursday 18th November 2010	Submission	Interim Report
Tuesday 30th November 2010	Project Supervisor Meeting	Pre-Christmas break catchup
Tuesday 11th January 2011	Project Supervisor Meeting	Post-Christmas break catchup
Tuesday 25th January 2011	Sangwin Meeting	Discussed scenegraph design ideas
Monday 31st January 2011	Sangwin Meeting	Discussion of general progress
Wednesday 9th February 2011	Sangwin Meeting	Initial discussion of Desktop renderer
Monday 14th February 2011	Project Supervisor Meeting	Discussed Desktop renderer progress
Tuesday 22nd February 2011	Project Supervisor Email	Sent across first successful Desktop render
Wednesday 2nd March 2011	Sangwin Meeting	Discussion of Android renderer
Monday 7th March 2011	Project Supervisor Meeting	Discussed progress and remaining code left to translate, in order to decide what was required for the project to be successful
Thursday 17th March 2011	Submission	Draft Report
Thursday 24th March 2011	Project Supervisor Email	Received Draft Report feedback, with suggested changes and ideas for improvement
Wednesday 13th April 2011	Sangwin Meeting	Discussion of profiling results and issue with Collision Detection
Wednesday 27th April 2011	Project Supervisor Meeting	Final discussion, including report overview and discussion of profiling results