KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Business Information Technology

Aki Kaisanlahti

APPLICABILITY OF COMMON LISP IN GAME DEVELOPMENT

Thesis
February 2017

THESIS
February 2017
All Degree Programmes

Tikkarinne 9
80220 JOENSUU
FINLAND
(013) 260 600

Author (s)
Aki Kaisanlahti

Title
Applicability of Common Lisp in Game Development

Commissioned by
NKUAS and KUAS

Abstract

The aim of the this thesis was to find out how applicable Common Lisp is in game development through examining the language and game development as well as trying to build a core of a game development framework with Common Lisp.

Common Lisp was inspected as a programming language on its own, through comparing it with C family languages as well as in game development context through earlier use in video games. Game development was covered through examining tools and methods based on a programming perspective. Criteria for the final evaluation of the framework and Common Lisp in game development in general were based on what distinguishes game development from general software as far as development is concerned.

The core of the programming library is based on Common Lisp macros; its API implementation is based on lower level programming libraries. The game development library is designed very strongly based on a programmer's perspective and aimed at other programmers in order to focus on distinguishing points of Common Lisp in game development.

The applicability of Common Lisp and the game development library in game development are based on the flexibility of the language due to its design. Future development of the library and with the library would make it possible to observe its applicability on a larger scale in game development. It would also enable broadening the possibilities of the API and fixing current issues such as foreign memory problems with foreign language library bindings or replacing them with native Common Lisp solutions.

Language	Pages 51
English	Appendices 1

Keywords

Common Lisp, game development, macros

OPINNÄYTETYÖ
Helmikuu 2017
Kaikki koulutukset

Tikkarinne 9
80220 JOENSUU
(013) 260 600

Tekijä(t)
Aki Kaisanlahti

Nimeke
Common Lispin soveltuvuus pelinkehitykseen

Toimeksiantaja
PKAMK ja Karelia-amk

Tiivistelmä

Opinnäytetyössä tutkittiin Common Lisp -ohjelmointikieltä, pelinkehitystä ja Common Lispin edellytyksiä siihen. Common Lispillä toteutettiin myös pienimuotoisen pelinkehitykseen tarkoitetun ohjelmointikirjaston pohja.

Common Lisp -kieltä käsiteltiin itsenäisesti, C-pohjaisiin kieliin verraten sekä pelinkehityskontekstissa. Modernia pelinkehitystä käsiteltiin metodien ja työkalujen näkökulmasta ohjelmointikeskeisesti. Tämän perusteella määriteltiin kriteerit, jotka erottavat pelinkehityksen normaalista ohjelmistokehityksestä. Kriteerien avulla voitiin arvioida ohjelmointikirjastoa ja Common Lispiä pelinkehityksessä

Ohjelmointikirjaston pohja tukeutuu Common Lispin makroihin; ohjelmointirajapinnan sisäinen toteutus perustuu muihin alemman tason ohjelmointikirjastoihin. Ohjelmointikirjasto on suunniteltu vahvasti ohjelmoijanäkökulmasta ohjelmoijille, jotta voitiin keskittää huomio Common Lispin erityispiirteisiin pelinkehityksessä.

Common Lispin ja ohjelmointikirjaston vahvuudet pelinkehityksessä perustuvat kielen joustavuuteen kielisuunnittelun ansiosta. Jatkokehitys selventäisi laajemmin kielen soveltuvuutta pelinkehitykseen. Jatkokehityksen myötä olisi myös mahdollista korjata yhteensopivuusongelmia toteutukseen käytettyjen muiden ohjelmointikielten kirjastojen kohdalla tai toteuttaa näistä Common Lispillä omat kirjastot.

Kieli englanti	Sivuja 51 Liitteet 1

Asiasanat

Common Lisp, pelinkehitys, makrot

Content

Glossary....6
1 INTRODUCTION....7
2 COMMON LISP....9
2.1 History and notable current implementations of Common Lisp....9
2.2 Common Lisp as a language....11
2.2.1 Language paradigms....11
2.2.2 Common Lisp syntax and language design....13
2.2.3 Language comparison....15
2.3 Programming ecosystem in Common Lisp....17
2.4 Lisp in games....17
2.5 Performance....19
3 GAME DEVELOPMENT....20
3.1 Game development methods....21
3.2 Game development tools....22
3.3 Criteria for game development....24
4 LUCERNA: PREPARATION....25
4.1 Tests....26
4.1.1 Classimp....26
4.1.2 SDL....27
4.1.3 Cl-opengl....28
4.1.4 Combinatory tests....29
4.1.5 Deliverable executable....30
4.2 Component choices....31
4.3 Development tools....32
5 LUCERNA: IMPLEMENTATION....33
5.1 Framework component default implementation....34
5.2 Framework component custom implementation....35
5.3 Framework component extension....36
5.4 Demo application....37
6 CONCLUSION....39
6.1 What is Common Lisp?....39
6.2 What is modern game development?....39
6.3 Tools and libraries in small scale game development....40
6.4 Applicability of Common Lisp in game development....40
6.5 Evaluation and discussion....41
6.6 Future development....42
References....44

Appendices
Appendix 1

Glossary

XNA A framework for making games made by Microsoft. Intended to run on Microsoft platforms such as Xbox 360 and Windows.

SFML Simple and Fast Multimedia Library is composed of five parts that cover system, window, graphics, audio and network usage. Originally developed for C++, it has multiple language bindings today. Developed by Laurent Gomila (Gomila, 2016).

SDL Simple DirectMedia Layer is similar to SFML, but is older, more well known and used. Used partly in Unreal Engine 4 and e.g. In FTL: Faster Than Light.

OpenGL Open Graphics Library is a specification made by Khronos group along with others. Used widely in many different platforms.

Vulkan is a new, lower level specification made by Khronos group aimed to give more control in graphics over the GPU.

CFFI Common Foreign Function Interface, Common Lisp's foreign function interface that enables handling foreign code and memory. The cornerstone of Common Lisp language bindings of common libraries such as SDL.

1 INTRODUCTION

The aim of this thesis is to examine Common Lisp as a programming language and reflect on its applicability in game development. In order to examine this, the main questions set were: What is Common Lisp? What distinguishes game development? What tools and libraries are necessary to enable small scale game development? How applicable is Common Lisp in game development? The choice of small scale game development and low level library testing is based on the fundamentals. If you can establish reliable rendering, windowing, input and output and such, other components of game development are more about scaling and abstraction.

The game development framework development process started from seeing a video on live editing code with Common Lisp. Seeing it is possible to compile code, run and see changes live while working on the code seemed like a huge quality of life improvement compared to more static environments like those generally found in the C++ ecosystem. This is within using the same language, scripting languages and interpreters are a different matter entirely.

Having the desire to obtain a centralised, easy to use and easy to extend or modify framework was where I started. If the environment made it possible to change and see the changes live, I wanted to see if it was possible to make a core framework for the environment and make working with it even easier. If the user of the framework should not be happy with all parts of it or wanted to integrate some other systems into it, it would make adapting the framework easier as long as the core API and functionality was deemed acceptable.

Trying to learn Common Lisp basics while still working on C# and C++ code trying to separate implementation details from the abstracted API, I did some experiments to see how easy it would be to change some parts in C++ world. Separating the API from the implementation by a layer proved to be fairly easy, but I found the basic idea to be even easier in Common Lisp.

After testing components of lower level libraries through the foreign function interface in Common Lisp, I started to work on the idea of separating the structures and function calls of the API from the implementation. In spite of having it possible to type out the arguments for speed, one could also just as easily choose not to type them and have them be automatically typed. As long as functions had easily determinable input and output, it would be fairly easy to control the API in order to make it possible to extend or to change the actual implementation without touching the API itself. As long as classes or structures reflected this by having clearly marked data types that they required, they could be constructed by just having proper input to their constructing functions.

After going through this, I came upon the idea to make the first step in making the tools for extending and changing implementations. This was because even if I was personally happy with how the separation of the API and implementation was going, I would never know what components could be developed using the same principle in separate projects, nor could I possibly expect to have the best implementation of even my own functionality. As long as the whole project wouldn't be unusable and it had a good design behind it API wise, ease of extensions, integrations and implementation changes would make it easier to adopt the framework into use in the potential future.

In order to answer the questions put forward, the thesis is constructed into two main parts. In the first part, the theory behind Common Lisp and game development is explored. In the second part, a framework basis for future game development is implemented.

In the section about Common Lisp, its history is briefly looked at. After that, the section focuses on the programming language itself, as well as its ecosystem, previous usage in games and performance.

Game development section is about seeing the current state of game development and defining criteria for the practical implementation part of the thesis. Looking at game development, we define what it is, what methods are usually used in game development, take a look at tools used in it and define the criteria

for game development as a whole so we have a comparison basis after having implemented the framework.

The practical part of the thesis is divided into two parts. In the first part, the design and tools base for the framework is introduced in order to limit the scope of it.

The second part is all about the actual implementation of the framework, taking a look at it from a high level perspective by looking at the system as a whole and how the subsystems are tied into it.

In the final part of the thesis, results are examined and discussed based on the criteria defined earlier. Future development options and improvements are discussed. Problems and areas that the framework fell short on are also pointed out.

2 COMMON LISP

This section concentrates on Common Lisp. The contents include a short history of Common Lisp as a language from design to standardisation. The language is introduced as well as main similarities and differences to mainly C-family languages are discussed, but the focus will be on the language itself. Comparison is mostly made due to their prevalence in software development and the games industry. Common Lisp's community and ecosystem are discussed with game development focus in mind. The section finishes with some known examples on the use of Common Lisp in games.

2.1 History and notable current implementations of Common Lisp

Common Lisp is a multi-paradigm programming language that began to form in the 1980s (Gabriel & Steele 1993, 20-22). The first ANSI standard was formed in 1994 (ANSI, 1994). After standardisation, there have been multiple Common Lisp implementations of the standard, although there are no standardised tests each implementation must fulfill and thus their compatibility with the standard has not been recorded. Besides providing implementation of the features required in the standard, many implementations provide other features that aren't in it, e.g. threads, making for a “robust and vibrant language”. (Weinreb, 2010.)

Some of the more notable Common Lisp implementations are Steel Bank Common Lisp, Allegro Common Lisp, LispWorks and Armed Bear Common Lisp. This is due to having a fairly optimised compiler, commercially used and liked toolkit or having the possibility to include code from another programming language and ecosystem in it.

Steel Bank Common Lisp was originally forked from CMU Common Lisp (SBCL, 2004). CMU itself was a project in Carnegie Mellon University that started as a part of their Spice project in 1980. Originally named Spice Lisp, it was renamed CMU Common Lisp afterwards as Common Lisp's first standard had come out. (MacLachlan, 1999.) SBCL is known for having a compiler that compiles Common Lisp into fairly optimised machine code.

Allegro Common Lisp is a Common Lisp implementation made by Franz Inc. It's a commercial Common Lisp available on multiple platforms such as Windows, Linux and Mac OS X. It has been used in a number of commercial applications and projects, such as the 3D software Mirai used in Lord of the Rings movie trilogy, Game Oriented Object Lisp in Naughty Dog. (Franz Inc, 2015b.) Furthermore, Allegro Common Lisp was used as the base for multiple products of Nichimen Graphics used in the games industry in the late 1990s to early 2000s. (Franz, 2015a.)

Armed Bear Common Lisp is notable as being a Common Lisp implementation that runs on the Java Virtual Machine. What this effectively means is that ABCL lives inside the huge Java ecosystem, having access to the libraries that

operate on the same JVM platform while still being an implementation of Common Lisp itself. (ABCL, 2015)

2.2 Common Lisp as a language

Common Lisp is a multi-paradigm language. In this chapter some of the more well known and used aspects of Common Lisp are explored and compared to other mainstream languages of mainly the C-language family due to them being ubiquitous in general software as well as video games.

2.2.1 Language paradigms

Common Lisp supports object-oriented programming through its Common Lisp Object System (CLOS). When Common Lisp was standardised, CLOS was included in it. Although the language offers object-oriented features, they differ from languages like C++. (Seibel, 2003b.) In Common Lisp, methods that may operate on a class do not belong to the class itself, but to a generic function instead. Generic function defines a name and a lambda-list, but does not have an implementation and cannot be invoked. Instead, a method that specialises one or more parameters of the lambda-list defines an actual operation that will be invoked for the proper type(s). A method can have multiple specialisations, making it a multimethod. (Seibel, 2003b.) Multimethod somewhat looks like a function or method overloading in C++, but is not exactly the same. The method with the most specialised arguments is guaranteed to be called and the code placement for handling objects of different classes doesn't have to reside within them, but outside of them.

Common Lisp has first-class functions. This means being able to create new functions dynamically and being able to bind them to variables like other values

and entities. Higher-order functions are functions that can take functions as parametres as well as return them from the function itself. Since the language allows higher-order functions and it has first-class functions, it can be said to include functional programming as one of its paradigms, although a programmer isn't forced to program in a purely functional fashion.

In Common Lisp, parametres for functions and e.g. macros are defined through lambda-lists. A lambda-list may contain none or all of the following five parts: normal specifiers for required parametres, followed by lambda-list keywords **&optional**, **&rest**, **&key** and **&aux**. Optional parametres are as the name implies, optional. Rest takes a list of arguments that can be anything in size. Key parametres are named and correspond to a key. Auxiliary variables aren't technically parametres, but rather variables that can be included in the lambda-list. (Steele, 1990.)

Listing 1 includes functions that have basic examples of all of the five different parts that can be included in a lambda-list. Function *example-params* uses normal named parametres and multiplies its parametres. In *example-optional* **&optional** is used in naming a parametre that is initialised in the definition as being five in case it's not provided. Otherwise the functionality is the same. Keyword **&rest** is demonstrated in *example-rest,* where all of the given values to the function will be picked one by one and multiplied by five. Although it doesn't show inside the function, *example-key* does define the function differently in use. It requires the programmer to use it by calling the parametre name with a colon before it, in this case **:param2** or **:param3**. As mentioned before, **&aux** is technically not even a parametre, so it only shows up in the function definition and is used inside the function, it cannot be called from the outside, *example-aux* shows a basic example of this, it multiplies the given parametre with 5 which is the value of **auxiliary1**.

```
(defun example-params (param1 param2)
  (* param1 param2))

(defun example-optional (param1 param2 &optional (optionalparam 5))
  (* param1 param2 optionalparam))

(defun example-rest (&rest paramlist)
  (dolist (x paramlist) (* x 5)))

(defun example-key (param1 &key param2 param3)
  (* param1 param2 param3))

(defun example-aux (param1 &aux (auxiliary1 5))
  (* param1 auxiliary1))
```

Listing 1. Examples of function definitions with different lambda-lists, including all of the five types of keywords. All of the functions do basic multiplication.

2.2.2 Common Lisp syntax and language design

The Common Lisp syntax is based on symbolic expressions, shortened S-expressions. S-expressions consist of either an atom or a list. Naturally, the list can be a nested one, including other lists inside of it. Everything that is not a list is an atom, including symbols and numbers. The only entity that is both an atom and a list is the empty list, also known as NIL. Although the syntactic tree is based on s-expressions, not every s-expression is a valid Lisp form. (Luger & Stubblefield, 2009.) Listing 2 has five examples of S-expressions. The first one is a basic string, second and third examples feature numbers. The fourth one is a list with three members in it and the fifth one is a list as well with four numbers in it.

```
1. "hello"

2. 1

3. 1/3

4. (format t "hello")

5. (1 2 3 4)
```

Listing 2. Examples of S-expressions. First three are atoms, 4. and 5. are examples of lists.

Valid Lisp forms are either Common Lisp atom elements or lists that start with a symbol. Symbol is a named object that can either refer to an operator or a variable and when evaluated on its own will return the value of the variable tied to it. (Seibel, 2003a.) There are three different kinds of valid Lisp forms that are not atoms and they start with an operator; function, macro or a special operator. Listing 3 contains examples of valid and invalid Lisp forms.

```
1.   (+ 1 2 3)

2.   (+ 1 (* 2 3) 3)

3.   (1 + 2 + 3)
```

Listing 3. Examples 1. and 2. are proper Lisp forms, they have a function as the first element of a list, while 3. is a valid s-expression, but not a proper Lisp form, since it starts out with an atom that's not a macro, function or a special operator.

Lisp has list as a built-in type and has multiple functions associated with lists (Reddy, 2008). This is due to Lisp evolving as a language alongside rapid prototyping and including features such as lists as a fundamental type alongside other features such as keyword parametres (Graham 1993, 284). This does imply that some of the features do come at the cost of speed and efficiency in certain cases.

One of Common Lisp's more prominent features is macros. Macros in Common Lisp work very differently compared to Macros in C or C++. C++ macros are written exclusively for the proprocessor that reads them before the compiler and replaces the defined identifier with text (C++ reference, 2011). In Common Lisp there are three different types of macros: symbol macros, read macros and compilation macros. Symbol macros look like symbols instead of function calls and as such don't use parametres, but they can substitute any Common Lisp code in a manner that compilation macros can (Graham 1993, 105). Read macros work with the Lisp reader that reads the code before compiling it (Graham 1993, 225). Read macros enable reading code differently or for example making it possible to embed JSON syntax into Common Lisp (Gupta, 2014). Compile-time macros are generally expanded at compile time, although this is not defined by the standard. They must return a valid S-expression.

Macros enable using regular functions and Common Lisp as a language itself instead of relying on its own macro language which makes it possible to define a new domain specific language while still using the same syntax as the base language, Common Lisp. This means that the programmer doesn't have to jump hoops or go outside the base language to define new constructs. It also allows to write code that writes code itself, whether it's a macro that defines an anonymous function or even a macro that defines another macro that in turn returns code to be actually run. (Seibel, 2003c.)

2.2.3 Language comparison

Although C# started to support functional features since version 2.0 and increased it later on, the language is based on the family of other C style languages and thus is based on imperative programming. Since Common Lisp is very flexible due to its macros, it would be possible to write code that resembles imperative languages, but at its core the language is more oriented towards functional programming, which is showcased against imperative programming below in listing 4. The Common Lisp example applies a function

on each element of the list with **mapcar**, taking the function as a parametre. In this case it's an anonymous function that multiplies the element with itself.

```
(mapcar #'(lambda (x) (* x x)) list)

for (int i = 0; i < list.Count; i++)
{
    list[i] *= list[i];
}
```

Listing 4. An example of a way to multiply every member of a list by itself in Common Lisp (first line) and C# (last four lines).[1]

In both examples the code ends up with a list that has its members raised to the second power. The Common Lisp example starts out with the default function mapcar which takes as its parametres a function and a list and applies the function to the members of the list given to it. The function given to it is created through the use of lambda macro. The function takes x as its parametre and multiplies it by itself, making the number a second power of the original. The final element in the expression is the list given to mapcar.

The C# version uses the basic for loop due to the fact that mutating a changing collection with something like foreach will not work. In the loop, there's a temporary variable **i** that is incremented on every iteration of the loop until it reaches the last element of the list. In the loop itself, the element is multiplied by itself and assigned to the same place in the list, effectively raising it to the second power.

As mentioned above, there are multiple ways to solve the issue of applying an operation to each element in a list, depending on the operation and situation. In C# one can use ForEach designed for generic collections and apply a function through it. Common Lisp can also use user-defined or standardised looping macros to achieve somewhat similar code when compared to C#.

[1] It is possible to use e.g. iteration macros to make the Common Lisp code seem more like the C# code, but that is not the conventional way of doing it.

2.3 Programming ecosystem in Common Lisp

Common Lisp has often been claimed absent of enough libraries to support productive development of many applications, including games. This is especially the case when considering a well known and centralised system for getting and installing libraries. In the case of an appropriate library existing, it might require a lot of work to set up, discouraging from using it. One project that aims to resolve this is Quicklisp which makes installing most libraries in the project a simple process. The Quicklisp project currently includes more than 1000 libraries in it that are easily set up in a Common Lisp working environment across multiple implementations of the language. (Beane, 2011.) Installing a library into your working environment requires just a single call: **(ql:quickload ”library-name”), ql** being shorthand for Quicklisp. If the library depends on some other system, it's downloaded automatically for the user. Libraries that use the Common Lisp foreign function interface need their dynamically linked libraries to work.

As is fairly apparent though, Common Lisp is not one of the mainstream languages at the time of writing. In late 2015, Github had nearly 9500 repositories using Common Lisp as a language based on their search system. Meanwhile, C++ had over 400000 repositories. Although it is hard to estimate fully how a programming language ecosystem works and how lively it is, the numbers do point out that the ecosystem is very likely to be a lot smaller based on public, open source code available.

2.4 Lisp in games

The games industry has adopted C++ as its de facto language for development, as can be seen from many game engines and job listings (Klint, 2016). Although the rise of indie development beside the AAA development studios has given

raise to other languages, such as C# and Java, Common Lisp hasn't taken off in a similar manner as of yet. In fact, many of the cases where a Lisp or Common Lisp variant has been used in games is from the time when C++ wasn't always the obvious choice. In this section Naughty Dog and its GOOL as well as Crack Dot Com's Abuse are taken a loot at.

Naughty Dog is commercially a very successful game development studio founded in 1984, currently owned by Sony Computer Entertainment. It has developed many commercially successful titles including e.g. Crash Bandicoot series and the Jak and Daxter series.

Crash Bandicoot, for the Playstation console, was developed using a language called GOOL, Game Oriented Object Lisp. The language as well as the associated tools of the studio were created using Allegro Common Lisp provided by Franz Inc. GOOL and the associated tools were developed to overcome many of the contemporary limitations of technology in game development. Many problems, such as inconsistency of syntax and text based macros of C, were solved by GOOL, making for rapid development (Gavin, 1996.)

GOOL allowed to solve many problems that other programming languages had at the time. The language had, among others, the following features: LISP macros, light threading and dynamic linking. These allowed for many of the things some contemporary languages, such as C or the early C++, didn't. Macros made extending the language easier while having consistent syntax and small memory usage is helpful on consoles. Since GOOL compiled to assembly in the end, it had speed as well, not having to make sacrifices to that end. (Gavin, 1996.)

For their other game series, Jak and Daxter, Naughty Dog developed another language and a batch of tools. The language and its compiler were still developed with Allegro Common Lisp. The language was reformed, named Game Oriented Assembly Lisp (GOAL) and it had several features its predecessor did not, including the titular ability to write assembly within the Lisp expressions. (White, 2002.)

Abuse is another example of the usage of Common Lisp. It's a PC game released in mid 1990s, made by a company called Crack dot Com. The game features a Common Lisp interpreter that implements most of the Common Lisp ANSI standard, although it does not support structures or objects. The game engine of Abuse was mostly written in C++ while Common Lisp was mostly used for game scripting. After the development of the engine, the actual game development only lasted four months. (Perry, 1995).

As can be seen above, many of the games that use Common Lisp were made in 1990s or early 2000s. Although Naughty Dog used it successfully in AAA titles for many years, it faded away from use. This is partially due to C++ being more mainstream, so programmers for it were easier to find compared to Common Lisp (White, 2002). This might explain why the language hasn't been seen in video games in the last decade.

2.5 Performance

Performance is important to an extent in game development, especially in the AAA game industry. This is due to the fact that games are by their very nature interactive and their core game loop is often even run 60 times per second. Even if the game is updated less than 60 times a second, the game might be rendered that many times on screen (Nystrom, 2009b).

Focus on performance is further compounded by the fact that currently and historically AAA games run on video game consoles which have fixed hardware (Garney & Preisz 2010, 293). As markets expect more from games every year, companies have to attempt to squeeze as much performance as they can from the console they're developing the game for. This is less so in the case of smaller games from smaller teams, but interactivity is still crucial, so truly abhorrent performance on the game code won't cut it even in the case of indie titles.

Performance for a language is hard to measure as it depends on multiple factors. There are some benchmarks that put Steel Bank Common Lisp most of

the time at roughly 2-4 times execution time to that of C++ (Fulgham & Gouy, 2015). The benchmarks themselves are based on certain algorithms and their implementations and thus are in isolation. This implies that taking into account the differences currently in compiler optimisations and such, Common Lisp is slower than C++ in general cases. This doesn't mean that Common Lisp code is always slower without a question, nor that it isn't comparable to other languages used for game development, but gives a starting point for evaluating performance.

3 GAME DEVELOPMENT

This chapter aims to look at game development and how it generally works. There's a general overview, but the criteria that I base my thesis on are mostly based on programming, although there's a large overlap in general game development and game development as far as programming goes. This section is not focused on e.g. quality assurance, content creation or marketing.

Game development is highly iterative. Many game development studios employ agile methods for development, iterating the project in small timeframes. This is not the case for all studios however, some employing some currently less known methods such as the Cerny method or even waterfall. (Tozour, 2014.)

Since game projects may employ people from different disciplines from design and storytelling to art and programming, studios either make or license tools in order to get input from multiple disciplines without always having a programmer piece the game together.

These tools range from simple frameworks such as XNA or SDL all the way to full blown environments and editors such as with Unreal Engine 4 or StingRay. There are also more specialised tools that only tackle a single problem or integrate two tools together, such as Bullet physics engine or A.R.T for Unreal Engine 4 and Autodesk Maya.

In this chapter the basis for evaluating how well Common Lisp and its ecosystem can fit in game development is also evaluated. The basis is determined based on examining game development, its tools and methods.

3.1 Game development methods

Game development especially in large scale is often divided into five phases: concept, preproduction, production, postproduction and aftermarket (Sloper 2009, 791). These five stages may include different things depending on the exact development method chosen, but all of them are generally part of creating a game.

In the concept phase, the general concept of the phase is explored and decided as well as written down. In larger franchises and premade intellectual property of a publisher, the concept for a game can already be fairly ready, not needing a designer to work on it. (Sloper, 2009, 791-794.) A concept document is written in this phase and it's generally kept brief, spanning e.g. a few pages.

Preproduction consists of forming the team and writing a design document for the game (Sloper 2009, 794). In the Cerny Method, this is also the phase in which the first playable is made. The reason to push a first playable version of the game as early as possible is to reduce overall costs for a studio or a publisher. If the project doesn't seem promising, only time and money up to this point will be sacrificed instead of going through full production. (McLean-Foreman, 2002.)

After preproduction, production naturally begins. This is the phase where the majority of the work on the game is done. If the first playable wasn't built in the preproduction phase, it is generally done here as soon as possible in order to evaluate the upcoming product (Sloper 2009, 816). Late in the production phase, the majority of the content, especially art, has been completed, but programming is not fully done (Sloper 2009, 821). This depends on the scale and type of a game built.

Postproduction is after the game has been completed in its design and art, but some programming pieces are generally still missing or bugs have yet to be fixed (Sloper 2009, 824). In some definitions postproduction is extended to be a phase that lasts even after the initial launch of a product, covering bug fixes and even content patches. Most of the time postproduction is about polishing and finishing the product for consumers.

The majority of game studios employ agile methods of development while working on a game (Tozour, 2014). According to the findings of Koutonen and Leppänen (2013, 6) based on a survey made to Finnish game developers, practices used in Scrum are most commonly used in preproduction and production as opposed to postproduction or the concept phase.

The concept and and definition may differ, but many studios use an idea of a first playable. Mark Cerny defines this as a ”publishable” first playable in his description of the method named after him. Publishable first playable version of a game has two finished levels or areas and all needed features implemented for those levels. (Academy of Interactive Arts & Sciences, 2012.) This allows to make a more rational decision on whether to continue making to game or to scrap it. The publishable first playable might also be called minimum viable product or MVP (York, 2012).

3.2 Game development tools

Since game development is most of the time very iterative, tools need to be easy to use and content has to fit in to the game with little effort once the actual production starts. In the preproduction it isn't as crucial to have an easy to use content pipeline, but once production ramps up, due to the multi-discipline nature of development e.g. must have an easy access to the tools to import their creations from specialised software, or someone else has to use their possibly productive time to put the content in to the game. This can cause a bottleneck in production that will waste resources.

Game development tools are tools that are specifically designed to be used in developing games, although there are no formal definitions. In general, tools such as 3D modeling software or digital audio workstation software are left out of this, as they can be used for other purposes as well, unlike a game specific editor. The definition will likely never be robust, as there have been examples of 3D short films being developed with game engines. One example of these is The Butterfly Effect made with Unity (Zioma, 2012).

On the lower abstraction level of tools, there are game development frameworks. They are there purely to help the programmers of a project. They abstract certain low level systems and have an API for programmers to interact with, but they have little value to non-programmers. Examples of frameworks include SDL, XNA and SFML.

Frameworks and libraries do not necessarily include any code for implementing or automating the process of developing a certain type of game and instead only abstract low level systems, such as rendering, input & output operations or playing sound. It's generally left to the programmer to implement the actual engine code in this case. The engine code will be responsible for making sure the low level systems work together as intended.

On a higher level, there are editors that interact with the engine and framework code. These editors might not be game specific yet, such as a general level editor or an object editor. They are also useful for other disciplines, such as game designers and artists that want to tweak objects or levels in the game to reflect their vision.

On an even higher level of abstraction, there are tools to help make a specific game. If the game needs a lot of hand crafted content or its procedural generation requires a lot of hand crafted pieces in order to start generating levels or content, it might have at least one editor for a specific type of content the game requires. These include quest or mission editors that have specific formats, or AI behaviour editors that are game specific, i.e. non-navigational editors, unless the game uses a special pathfinding for its non-playable characters.

3.3 Criteria for game development

In order to distinguish game development from other software, there are certain criteria that have to be met. Game development can come in many sizes all the way from one man studio to thousands of developers on a single project in a AAA studio. Since their needs are slightly different, we will mostly take a look at things they have in common, not taking too specific requirements from either end. The criteria are based on programming and the technical side of development.

As has been discussed earlier, game development is considered fairly iterative by nature, so the easier it is to iterate parts of the game or its assets while developing the faster the development speed itself is. So one criterion is clearly **flexibility.** (Nystrom, 2009a.) One example could be flexibility in technical design like having a Lua scripting above C++ engine makes it possible to reiterate the game code or level layouts while the game is being tested.

Game development often works in multiple abstraction levels in the actual production phase. This can involve code very close to hardware as well as very high level gameplay code in terms of scripting and abstracting different hardware specific APIs. (Llopis 2009, 184-185.) Thus, **abstraction** is one of the criteria as well.

Scalability is also often important due to different hardware that the end user might have on PC or different mobile devices (Llopis 2009, 181-182). Console hardware doesn't usually change, but other platforms may vary wildly in their power and configuration. For example PCs have wildly variable hardware configurations from low end integrated graphics cards to high end double graphics cards.

Multi-platform development is also fairly common in game development, especially in contemporary game development. There often have to be multiple configurations of tools and assets to compile and distribute the game on

multiple platforms due to different underlying hardware (Llopis 2009, 184-185). This includes PC, mobile platforms as well as consoles, e.g. X86 processor versus the old PS3 Cell processor. Not all companies can do this, but from medium sized indies up to big AAA studios often port their games on multiple platforms.

Game development isn't only defined by these criteria. Other points of view from a technical standpoint could include e.g. performance. The criteria could also be split into smaller sections such as scalability concerning CPU and GPU separately as well GPU capabilities within certain graphics APIs. This could have implications on a more detailed level in a longer paper concerning a bigger project, for example in optimisation. A full breakdown of game development is outside of the scope of the thesis however, which is why these four higher level criteria were chosen.

4 LUCERNA: PREPARATION

As a part of the thesis a rough core of a game development framework in Common Lisp was made. This was done in order to find out in a qualitative manner how working in the Common Lisp ecosystem works in practice. The framework core was made use of in a small demo application in order to ensure its components worked as intended.

In order to build the demo application and the framework, the basics had to be established first, including tools, components and basic tests to ensure that the actual work was feasible due to the fact that the Common Lisp ecosystem has not established itself to the mainstream developer community.

As has been established earlier in discussion on Common Lisp programming ecosystem, it does not have a huge number of developers nor standard frameworks in game development. As of the late 2016, there are not that many game oriented libraries or frameworks, but there are some. In order to establish

how well the language is suitable for game development though, a basic framework will be implemented out of lower level libraries in order to find out how easy it is to establish technological base for developing games.

4.1 Tests

Although not strictly made to be a certain model of test driven development, I approached building the framework and its components through making very small tests and slowly expanding on them. This was due to my starting point with the project: not having a lot of expertise in a language as large as Common Lisp, not much documentation was available for the language bindings of common libraries and the fact that the bindings were not converted in any way 1:1 with the originals, but many authors had made changes so some of the calls were renamed, changed, added or modified to look similar to some Common Lisp conventions.

The tests do not cover every functionality of the libraries that were used. This is due to the fact that the aim of the thesis is not to make a benchmark program nor is it to make a full test suite or a full test framework for a given library. The tests were conducted to ensure that the Common Lisp language bound libraries were working as intended on a base level so development could start.

4.1.1 Classimp

In Assimp it was most important to test that importing files itself would work and thus was tested. There were a couple of the most common post processing flags used, but testing all of them wasn't important, since Assimp is only used as a temporary solution to loading 3D file formats. More optimised frameworks and engines can use custom file format and code to go along with it.

In order to see if the library worked correctly, there were three models loaded. Stanford Bunny and Dragon as well as a random 3D .obj model. The Stanford

models' vertices were checked against an online diff tool to see if there were differences, none were found.

The only issue is that Assimp itself can't load really large files, especially when it comes to the Stanford Polygon File Format .ply. Testing it on the highest resolution Stanford dragon, the file fails to load with Assimp, coming up only nil in Common Lisp. The issue was fixable in very large models by splitting them into submeshes with Assimps SplitLargeMeshes post processing flag.

4.1.2 SDL

SDL had to cover my windowing, basic looping and the input in this case, so I tested these sections in parts as well as combining them. For the core of the framework I did not think sound was necessary, atlhough in an actual game sound is almost always crucial and SDL supports playing sound by default. All of the tests were done a desktop computer, one monitor setup with a basic mouse and keyboard.

```
(defun test-sdl-window-flags (flags)
  (sdl2:create-window :title "FlagWindow" :flags flags))

;;;Flag list, in CL binding only the last part, e.g. :fullscreen-desktop, full list below
;SDL_WINDOW_FULLSCREEN Tested
;fullscreen window
```

Listing 5. Snippet from window testing with different flags. Full file included as appendix 1.

Since SDL in this project was used for its windowing, input and OpenGL context capabilities, they had to be ensured to work to a degree. Testing window part proved to be difficult at start since the window is not visible by default in some working environments due to having set the Windows' own flag for showing windows to be false. Since SDL does not check this flag when launched, it will be different and has to be fixed by toggling the window invisible and back to visible.

Events that are related to input or the window seem to be working normally and SDL's GL context creation seemed to produce the results that the graphics card supported as well.

Basic windowing works. Since the framework uses SDL, I tested basic windowing functionality with it. I did get some issues when opening the window in SLIME with Emacs open. After much digging this might happen if something sets the operating system flags for window visibility off while SDL itself has not toggled any of its own flags. In development environment one can use a hack to bypass this issue by toggling window visibility off and on while creating a window. This ensures that the SDL flagging of the window visibility is the same as the one the operating system uses.

4.1.3 Cl-opengl

I did basic testing to ensure that using OpenGL in Common Lisp itself seemed to work normally. This involved setting up basic vertex and fragment shaders and rendering a triangle. This was done in order to ensure that the basics were working. Extensive testing was not covered due to drivers and graphics cards being outside of the scope for this thesis.

```
;;Basic shader tests
(defun test-basic-vertex-shader ()
  (let ((source
        "#version 330 core
         layout (location = 0) in vec3 position;
         void main()
         {
             gl_Position = vec4(position.x, position.y, position.z, 1.0);
         }")
        (vertex-shader))
  (progn
    (setf vertex-shader (cl-opengl:create-shader :vertex-shader))
    (cl-opengl:shader-source vertex-shader source)
    (cl-opengl:compile-shader vertex-shader)
    (print (cl-opengl:get-shader-info-log vertex-shader))
    (return-from test-basic-vertex-shader vertex-shader))))
```

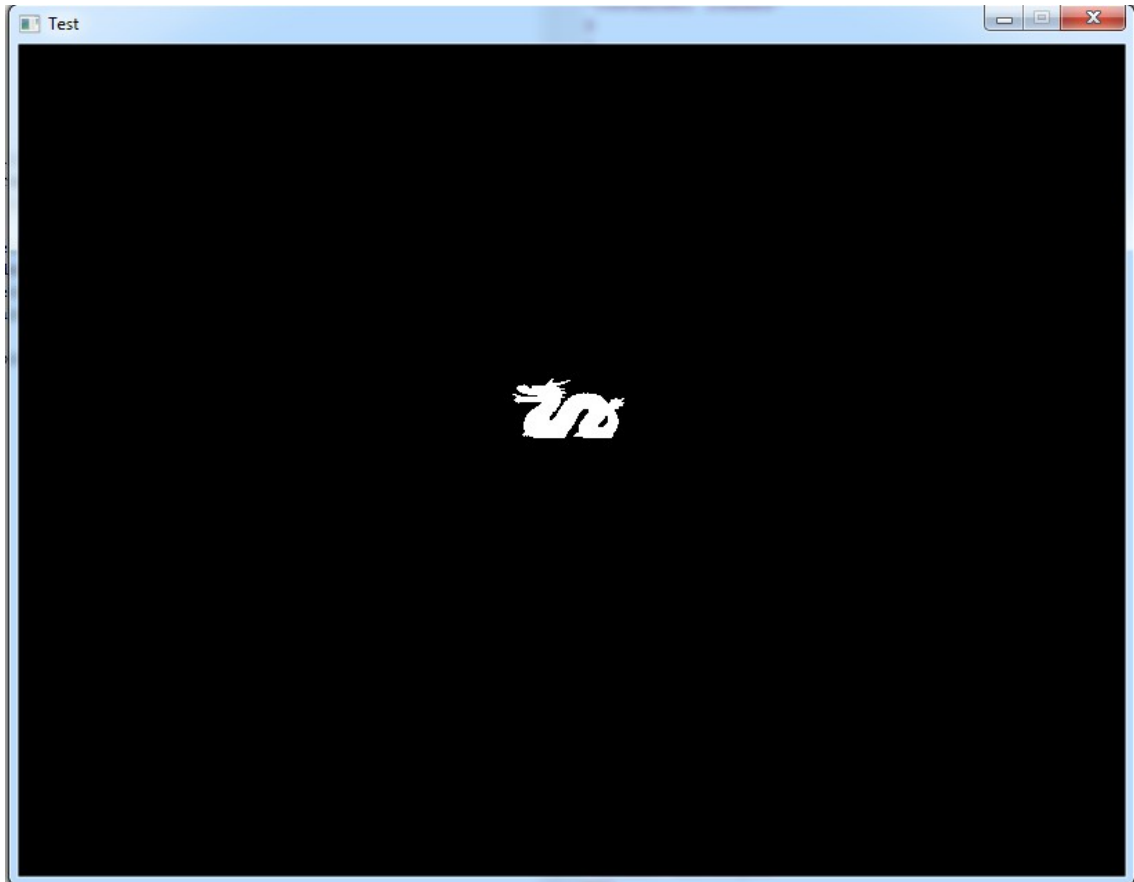
Listing 6. Snippet from OpenGL tests, featuring a basic vertex shader and its compilation and info logging.

Before creating the demo program with test meshes, basic rendering with cl-opengl was tested in order to ensure that the bindings themselves worked as expected. The basic rendering consisted of using modern OpenGL and GLSL to render a basic polygon with basic shaders to ensure basic functionality of the graphics pipeline, in this case OpenGL 3.3. was used. The rendering was part of the following combinatory tests since visual verification of rendering naturally requires a window to work with and it has to have some permanence. There were no issues with basic rendering.

4.1.4 Combinatory tests

Besides the very small tests for individual functionality, some tests were made that combined these into small programs and see them in full action. The combinatory tests used parts from other tests, such as creating a window or checking for a specific type of event and including basic rendering and model loading. The main difference to the demo program is that the test programs include code as is from the given frameworks instead of working using the framework code itself.

Picture 1. A test that had a basic window, one version of the Stanford Dragon



model loaded and rendered with minimalistic shaders, no lighting involved.

4.1.5 Deliverable executable

Deliverable executable with Steel Bank Common Lisp essentially always works through save-lisp-and-die which takes multiple parametres and saves the environment in a certain state to be opened from a specified function. There are some tools to help with this, namely Buildapp by Zachary Beane, though it does call the same function in the end. It is just designed to reduce necessary build script work from a developer. The final demo program uses a custom buildscript which is run from command prompt instead of Buildapp. Intial tests on this had satisfactory results, although there were some issues when loading foreign libraries to the environment for the executable. Even with two libraries in the exact same location were loaded in exactly the same way, only one of them loaded properly. This is mostly an inconvenience in development, since in the final version all libraries are in the current working directory of the executable

and are able to be loaded from there regardless of loading complications in development.

4.2 Component choices

In this section I will go through what lower level libraries I used. Due to time constraints as well as in the interest of scope, I chose to glue together existing components for the framework and focus more on non-networking components as opposed to network components, they are left out of the thesis. In the end, I focus on the very basics, such as working with basic windowing, input & output, file system, importing, rendering, sound. Libraries and frameworks chosen for these components include: classimp, cl-opengl, cl-sdl2.

Classimp is the Common Lisp language binding of Open Asset Import Library shortened Assimp. It is an open source library that supports importing multiple 3D file formats including most of the common ones (.3ds, .dae, .obj, .blend, .fbx) as well as some rarer ones. It also has some exporting functionality as well as other capabilities, but it is mostly used for importing and the intended use is to convert the imported formats to a custom file format for faster use. The design and implementation of a 3D file format is outside of the scope of this thesis. Wavefront's open .obj format is an example of what goes into making a specification for a 3D file format (Boulos, 2003).

OpenGL has its own binding in cl-opengl. OpenGL is at the time of writing still one of the two bigger standards of 3D programming interfaces that graphics card manufacturers implement drivers for. It is a fairly low level interface, although the upcoming Vulkan works on an even lower level and allows for more control. OpenGL itself generally is built upon in frameworks and beside DirectX works to provide a base for a graphics engine.

Naturally cl-sdl2 is the binding of the second version of Simple DirectMedia Layer SDL. It is a library that offers many basic functionalities packed into one library, e.g. windowing, input, sound and it even interacts well with OpenGL,

which saves effort and time due to not having to have a separate library for just setting up OpenGL to the window like GLFW.

4.3 Development tools

The tools for implementing Lucerna include Emacs, Superior Lisp Interaction Mode for Emacs (SLIME), and various Common Lisp bindings of libraries, such as cl-sdl2, classimp, cl-opengl. These tools are used in order to provide the basis for combining utilities into a single framework without having to write everything from scratch.

Emacs is a text editor. Its job in the project is to host the development process. It works well with Common Lisp for spacing and matching parentheses as well as providing the platform for compiling and running an instance of SBCL with SLIME. Emacs itself is also extendable through its own Lisp. This can be used to configure it. Emacs was chosen due to its support of Lisp through SLIME mostly, as Visual Studio and other conventional tools don't currently support creating a similar environment for development on Lisp. They may support basic syntax highlighting and parentheses matching, but not an environment where you can move withing the editor and Lisp itself, reflecting changes in one to the other.

SLIME is an interaction mode in Emacs for Common Lisp. It's designed to integrate Common Lisp runtime with Emacs, making it possible to work with Common Lisp in Emacs while writing, compiling, running and changing code on the fly.

As the name implies, cl-sdl2 is the Common Lisp language binding of second major version of Simple DirectMedia Layer. Classimp is binding of asset importing library Assimp.

5 LUCERNA: IMPLEMENTATION

The base and the implementation is based on a few principles and macros with in-depth descriptions below. These are called **defextension** and **defimplementation** that follow the general naming scheme in Common Lisp that has definitions starting with def, followed by the actual thing being defined. The goals of **defextension** are mainly the following:

- Make it easy to extend the development of the framework with a module and have it be separate from the main framework.
- Have the API be consistent and make it easy to include extensions in the same or similar packages as the original ones.
- Make it easy to define a custom extension for a game to the framework while leveraging the possibilities of multiple implementations for other development as well as having the code ready within development environment.
- Easy to look up full code without having to separately go to a source file while in development.
- Easy to manipulate code ready in hand.

The goals of **defimplementation** are mainly the following:

- Make it easy to swap out implementations of functionality. This can be another framework or just a different version using same underlying code if there's a need for comparison.
- Consistent API regardless of underlying implementation code.
- Easy live editing of code while making it possible to run multiple versions of the same code due to only running a different implementation of the function call. Easy to swap implementations live for comparisons or just return to the default in case there's a problem with the new implementation of functionality. This is possible since the parametres are the same, the only difference is in calling the implementation name in the parametre list if you don't want the default one.

The current state is examined in the following parts of this chapter. It was made after making personal research into Common Lisp, framework source code from SDL and SFML for example.

5.1 Framework component default implementation

The default implementation currently uses macros to set it up just like extension options, with the only difference being that the code for the default implementation is provided for the user out of the box. In the thesis phase, it depends on the component libraries such as classimp and cl-opengl.

The default implementation is decoupled from the libraries on the inner API in order to make it easibly replacable by future code. It's layered so that the actual implementation of the functionality uses library calls, but the API has an in-between layer that calls the actual implementation in order to separate the implementation from the inner API. The user API has macros that use the intermediate layer.

The basic design follows the principles of making the framework flexible by allowing extensions and custom implementations of the API.

Chart 1. An example of the chain of calls happening from the top level down to the implementation level.

As shown in chart 1, the first call is naturally made to the Lucerna API. In order for it to fetch the Window object in this case, it will call a macro that has implementation possibilities. The macro then calls a custom macro or a function that the underlying implementation uses. The next call will actually use the platform specific calls to construct the object that is then returned in the chain to the top. In this fashion, the Lucerna window does not have to worry about platform code nor the underlying implementation library in any way. In the thesis version, the underlying implementation is done by SDL which naturally calls its own platform specific code to construct the SDL Window that Lucerna uses in its implementation handle. The implementation could be easily swapped to use e.g. SFML windowing system, since the window handle type is not specified and the macros of the API could be defined to use custom implementations, in this case the SFML ones.

5.2 Framework component custom implementation

Custom implementation of components is directly supported by the **defimplementation** macro. The macro enables one to redefine parts of the framework without touching the API. This makes it easier to adopt and use with existing code as well as change the functionality of the framework without having to touch the source code directly.

As the **defimplementation** macro can be used to redefine a macro, and is currently set to do that, the compiler should be able to cut the cost of using it out, so it enables flexibility without having a significant overhead or possibly overhead at all.

5.3 Framework component extension

The extendability of the Lucerna framework uses **defextension** as its core functionality. The **defextension** macro currently defines a new macro with the standard call **defmacro**. In order to make custom implementations easier from the get go, it also defines given code to it as the default implementation and makes branching with **cond**. The code is saved to a hash table so it can be checked live as well as edited, this also enables editing it for a custom implementation, since the code is not saved by Common Lisp image itself. It could be extended to save the extension macro to a file in the future.

```
(defmacro defextension (name parametres &body code)
  `(progn (defmacro ,name ,(append-key-implementation parametres)
             (cond ((eq implementation :default) ,@code)))
           (setf (gethash ',name *extension-code*)
                 '(defmacro ,name ,(append-key-implementation parametres)
                    (cond ((eq implementation :default) ,@code))))))
```

Listing 7. Defextension macro.

This regardless of whether the extension is made for the future of the framework or just for a single project that wants a quality of life improvement while having the consistent API calls and code on demand for live checking and editing.

The defextension is mostly made for programmers. It does not offer any particular feature when it comes to game code, but its use comes in developing tools for a project or the framework itself. It is intended to make adapting and, as the name implies, extending framework easier.

5.4 Demo application

The demo application has a basic window with a test model rendered as a default with basic controls that control rendering. The main reason to build a similar demo to the largest combinatory test is to demonstrate the core of the framework in action. Many of the macros used could be defined as custom implementations or new functionality for the API could be written as extensions.

```
(in-package :demo)

;;;Called main due to legacy from
;;;other languages
(defun main ()
  (let ((window)
        (demo-mesh)
        (demo-running t)
        (rendering-type :default)
        (shader))
  (progn
    (print "Initialising")
    (print "Opening window")
    (setf window (lucerna:create-window "Demo Application" 0 0 640 480)) ;open by default
    (print "Loading testmodel")
    (setf demo-mesh (lucerna:3d-import "../Test assets/Test models/dragon_recon/dragon_vrip_res3.ply"))
    (print "Preparing mesh for rendering")
    (lucerna:prep-simple-mesh demo-mesh)
    (print "Creating basic shader")
    (setf shader (lucerna:create-shader-from-source (*vertex-source* *fragment-source*)))
    (loop
       while (eq demo-running t)
       do
          (progn
             (loop
                with event = (lucerna:empty-event)
                do
                   (prog2 (setf event (lucerna:pop-event))
                       (setf rendering-type (handle-event event)))
                until (eq (lucerna:event-type event) :NONE))
             (print "Rendering")
             (lucerna:render (list demo-mesh) shader window :implementation rendering-type)))
    (lucerna:destroy-window window)
    (print "Exiting main"))))

(defun handle-event (event)
  (cond ((eq (event-type event) :keyboard)
         (cond ((eq (key-virtual) :d) (return-from handle-event :debug-wireframe))
               ((eq (key-virtual) :n) (return-from handle-event :default))))))
```

Listing 8. Demo application code with debug prints and comments.

What's mostly done in the demo is a showcase of the API. Although not the most expansive use and test of it, the rendering is made slightly different in its alternative implementation. Since making a rendering library itself is a huge task

and the demo is meant to just show the principles, the rendering functionality has been implemented mostly with the **defextension**. It's easy to look at, modify and change implementation of it. Of course, just like regular code, it can just be deleted to start from scratch.

Listing 8 has the demo code included. Progn and prog2 allow sequential functions or macros to be called instead of just one function at a time. Cond allows for conditional execution of code and loop naturally loops a body of code. The code goes to a basic loop that handles events and when there are no proper events, the framework returns an empty event as a sign of no events, it renders the test model depending on the rendering mode chosen by keyboard prompts or the default it starts out with.

```
(defimplementation render debug-wireframe :extension
  (progn
    (print "Debug rendering")
    (cl-opengl:clear :color-buffer-bit)
    (cl-opengl:use-program shader)
    (cl-opengl:polygon-mode :front-and-back :line)
    (dolist (current-mesh simple-meshes
             (prog2 (cl:opengl:bind-vertex-array (array-object current-mesh))
             (cl-opengl:draw-elements
              :lines (cl-opengl:make-null-gl-array :unsigned-int)
              :count (list-length (mesh-indices current-mesh))))))
    (swap-buffers renderwindow)))
```

Listing 9. Debug rendering implementation of basic rendering used in the demo program.

As seen in listing 9, the debug rendering is a simple wireframe rendering of the meshes given to it implemented through defimplementation macro. It is a simple demonstration of the principles of the API core that does not represent a complex use case of switching a library for a different implementation, but rather a simple implementation that one can use to test a small difference in execution while keeping the main code exactly the same.

6 CONCLUSION

In this thesis the following questions were asked: What is Common Lisp? What distinguishes game development? What tools and libraries are necessary to enable small scale game development? How applicable is Common Lisp in modern game development? The following sections aim to dissect the content displayed earlier on in this thesis.

6.1 What is Common Lisp?

As established in section 2, Common Lisp is a multi-paradigm programming language based around simple syntax with symbolic expressions, very influenced by functional programming. It has a small history of being used in video games from smaller games to bigger ones, but is not very widely used in or outside of games right now. Decent performance, flexibility and library solutions such as Quicklisp make it a potential language for development.

6.2 What is modern game development?

Game development was largely covered in section 3. Based on its iterative nature, game development is involved with multiple platforms, needs flexibility, is concerned with scalability due to changing hardware on mobile and PC and due to working on multiple different levels of code and configurations, is subject to many abstractions and abstraction levels. More on the criteria for game development in section 3.3.

6.3 Tools and libraries in small scale game development

Covered mainly in game development tools in 3.2. The lowest level tools needed for development are generally low level libraries that programmers can use to build other layers and tools. Higher level abstraction and tools are useful for speeding up development.

6.4 Applicability of Common Lisp in game development

Based on the criteria, examining the language of Common Lisp, its use and personal experience with the language through starting a framework, Common Lisp does seem applicable as a game development language even in contemporary use.

The strengths of the language in game development lie in its simple design which allows for a great deal of flexibility in the form of different type of macros especially. Macros in turn create possiblities for domain specific languages which can be used to create tools or game specific building blocks without having to jump outside Common Lisp to use scripting languages or tacked on data formats.

Besides macros, Common Lisp offers a handy read-eval-print loop that can be leveraged in a live manner, editing code and running it in a really short feedback loop. Contemporary tools allow for compilation and reflecting that to the live Lisp image as well, making iterating really easy and quick.

There are difficulties and challenges when developing with Common Lisp. As established in 2.3 and 2.4, there is a Common Lisp community, but it is smaller than a mainstream language such as C++ has. The same goes for more well known and bigger titles developed with Common Lisp. With less developers, there are fewer examples, learning sources and co-developers to work with when developing a game. Overcoming the initial learning curve of not using a mainstream C-family language without as many resources as those languages have makes development more difficult.

A smaller community does affect the developer scene through tools and libraries as well, especially where more specialised tools are concerned. Although Common Lisp ecosystem has many general libraries and even has come around to having an easy way to get them to a developer's own environent through Quicklisp, the scene is still very reliant on low level foreign libraries and their language bindings through the CFFI. It is slower to start the progress of development with no higher level tools to speak of such as Löve2D, Unity or Unreal Engine for example. Language agnostic tools do make it possible to make content for games, but programming is somewhat hindered by the current situation.

6.5 Evaluation and discussion

Testing the basic libraries used as the basis for the framework was relatively simple although some steps required stepping in to the source of the bindings due to the fact that the syntax has been changed to resemble other Common Lisp code and there is very little documentation in any of the libraries. It also required some further digging into source, examples and other developers' code to see how the bindings worked when trying to dig for data that the bindings didn't explicitly account for but what was present in the original library. This was definitely the case with SDL, for example in its event data.

Unreal 4 has small use of SDL splashed in its HTML 5 and Linux portions. So the use of SDL in the actual implementation in the beginning seems justified even in tools, not just in games themselves. Other library choices such as the Common Lisp binding of Assimp seemed to work fine as a temporary tool to work with different 3D file formats while the current version lacks its own pipeline for 3D files.

Framework core was developed. The core was intended to leverage Common Lisp features and design. There were some issues with the core considering foreign memory and the intention of extensive uses of compile-time macros. As objects that were used in compile-time macros needed to have instructions on

how to construct them at that point, they broke down when using foreign memory and different frameworks that didn't have directions on how to initialise those objects.

The issue of initialisation in compilation could be fixed in the future by developing the implementations of the basic functionality of the framework natively. There might be workarounds by extending the language bindings of existing libraries. Some features and tools that help develop the framework and applications could be used even without fixing the problem as they aren't relying purely on compile-time macros, but rather the ease of development through easy access to code and basically a domain specific language that abstracts parts of simple programming out of the way, while still retaining the possibility to examine said programming easily and quickly within the environment.

The core of the framework tying together other libraries and its aim to make development easier is a bit lackluster due to its issues with foreign memory initialisation in compile-time. It shows potential, as the code is easy to write and examine in SLIME, but the current solution can not fully leverage the principles set forward in the beginning of section 5 as well as 5.1 and 5.2. At current state the API core can not make working with classes and data in general as flexible and easy as it aims to do for functions and macros.

6.6 Future development

Future developments of the framework include working with more file formats in importing assets, including sound, adding animation capabilities. Developing the API and underlying systems, possibly switching some implementation frameworks out for own implementations. Graphics API could take some influence from CEPL frameworks' Varjo and jungl.

Other improvements would make extensions and implementations even easier, plausibly giving the option to choose which kind of extensions the user would want and give macro help in writing them. An example of this idea would be to make a function, macro or a method based on what the programmer requires.

An extension of the principle applied in this thesis could be a library that is more generic and aimed at making development cycles easier and faster with more options on reading and modifying code as well as creating a Common Lisp package and system definition files for them.

The demo application is mostly a demonstration of the API on a basic level. The next target would be to make a small game while extending the framework in order to ensure its functionality and design so they serve actually developing games. After that would be a bigger game and possible tech demos to polish it up.

Due to time contstraints the extensions and implementations only used single frameworks instead of multiple ones. This is a very natural way of using the idea and could be explored in the future. The demo application didn't use it, but a bigger demonstration could use multiple underlying implementations of e.g. windowing with SDL, GLFW or even qt.

References

ABCL, 2015. Armed Bear Common Lisp. https://common-lisp.net/project/armedbear/ 14.8.2016.
Academy of Interactive Arts & Sciences. 2012. D.I.C.E. Summit 2002 – Mark Cerny. https://www.youtube.com/watch?v=QOAW9ioWAvE. 27.10.2015.
ANSI. 1994. X3.226-1994. http://webstore.ansi.org/RecordDetail.aspxsku=ANSI+INCITS+226-1994+(R2004), referenced 26.2.2014
Beane, Z. 2011. Quicklisp Beta. http://www.quicklisp.org/beta/. 3.6.2014.
Boulos, S. 2003. B1. Object Files (.obj). http://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf. 27.10.2016.
C++ reference. 2011. Replacing text macros. http://en.cppreference.com/w/cpp/preprocessor/replace. 7.9.2016.
Franz Inc. 2015a. Game Development Systems. http://franz.com/success/customer_apps/animation_graphics/nichimen.lhtml
Franz Inc. 2015b. Animation/Graphics. http://franz.com/success/customer_apps/animation_graphics/
Fulgham, B., & Gouy, I. 2015. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/u32/compare.phplang=sbcl&lang2=gpp 9.6.2015
Gabriel R., & Steele, G. 1993. The Evolution of Lisp. Cambridge, Massachusetts, USA. ACM SIGPLAN.
Gavin, A. 1996. AI and Character Control in Crash Bandicoot. http://all-things-andy-gavin.com/2011/03/12/making-crash-bandicoot-gool-part-9/. 19.03.2014
Garney, B & Preisz, E. 2010. Video Game Optimization. Cengage Learning.
Graham, P. 1993. On Lisp. Prentice Hall.
Gupta, C. 2014. Reader Macros in Common Lisp. http://lisper.in/reader-macros/ . 7.9.2016.
Klint, J. 2016. What Programming Language Should You Learn to Get a Job in the Game Industryhttp://www.gamasutra.com/blogs/JoshKlint/20160718/277319/What_Programming_Language_Should_You_Learn_to_Get_a_Job_in_the_Game_Industry.php
Koutonen, J & Leppänen, M. 2013. How are agile methods and practices deployed in video game development? A survey into Finnish game studios. http://www.researchgate.net/publication/255823539_How_are_agile_methods_and_practices_deployed_in_video_game_development_A_survey_into_Finnish_game_studios
Llopis, N. 2009. Game Programming: Languages and Architecture. In Introduction to Game Development by Rabin, S. (ed.). Charles River Media. USA.
Luger, G., & Stubblefield, W. 2009. AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java. Addison-Wesley.
McLean-Foreman, J. 2002. Conversations From GDC Europe: Mark Cerny, Jonty Barnes, Jason Kingsley. Gamasutra.

http://www.gamasutra.com/view/feature/2959/conversations_from_gdc_europe_.php
MacLachlan, R. 1999. CMUCL: Project History. http://www.cons.org/cmucl/doc/cmucl-history.html. 22.05.2014
Nystrom, R. 2009a. Game Programming Patterns: Architecture, Performance, and Games. http://gameprogrammingpatterns.com/architecture-performance-and-games.html
Nystrom, R. 2009b. Game Loop. http://gameprogrammingpatterns.com/game-loop.html
Perry, M. 1995. Abuse FAQ. http://abuse.zoy.org/wiki/doc/faq. 26.05.2014
Reddy, A. 2008. Features of Common Lisp. http://random-state.net/features-of-common-lisp.html. 3.6.2014
Seibel, P. 2003a. 4. Syntacs and Semantics. http://www.gigamonkeys.com/book/syntax-and-semantics.html
Seibel, P. 2003b. 16. Object Reorientation: Generic Functions. http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html
Seibel, P. 2003c. 8. Macros: Defining your own. http://www.gigamonkeys.com/book/macros-defining-your-own.html
Sloper, T. 2009. Game Production and Project Management. In Introduction to Game Development by Rabin, S. (ed.). Charles River Media. USA.
Steele, G. 1990. Common Lisp the Language, 2nd Edition. Woburn, Massachusetts, USA. Digital Press.
SBCL. 2004. History & Copyright. http://www.sbcl.org/history.html. 22.05.2014
Tozour, P. 2014. The Game Outcomes Project, Part 1: The Best and the Rest. http://gamasutra.com/blogs/PaulTozour/20141216/232023/The_Game_Outcomes_Project_Part_1_The_Best_and_the_Rest.php 9.6.2015
Weinreb, D. 2010. Common Lisp Implementations: A Survey. http://common-lisp.net/~dlw/LispSurvey.html 26.2.2014
White, S. 2002. Postmortem: Naughty Dog's Jak and Daxter: the Precursor Legacy. http://www.gamasutra.com/view/feature/131394/postmortem_naughty_dogs_jak_and_.php
York, T. 2012. Making Lean Startup Tactics Work for Games. http://www.gamasutra.com/view/feature/168647/making_lean_startup_tactics_work_.php 23.9.2016.
Zioma, R. 2012. THE BUTTERFLY EFFECT PROJECT. http://blogs.unity3d.com/2012/11/07/the-butterfly-effect-project/ 2.5.2016

Library tests

```
;;;SDL basic tests (sdl2)

;;Window tests

(defun test-sdl-window-flags (flags)
  (sdl2:create-window :title "FlagWindow" :flags flags))

;;;Flag list, in CL binding only the last part, e.g. :fullscreen-desktop, full list below
;SDL_WINDOW_FULLSCREEN Tested
;fullscreen window
;SDL_WINDOW_FULLSCREEN_DESKTOP Tested
;fullscreen window at the current desktop resolution
;SDL_WINDOW_OPENGL Tested
;window usable with OpenGL context
;SDL_WINDOW_SHOWN Tested, doesn't work
;window is visible
;SDL_WINDOW_HIDDEN Tested, is hidden by default
;window is not visible
;SDL_WINDOW_BORDERLESS Tested
;no window decoration
;SDL_WINDOW_RESIZABLE Tested
;window can be resized
;SDL_WINDOW_MINIMIZED Tested
;window is minimized
;SDL_WINDOW_MAXIMIZED Tested
;window is maximized
;SDL_WINDOW_INPUT_GRABBED Tested
;window has grabbed input focus
;SDL_WINDOW_INPUT_FOCUS Tested, generally not relevant
;window has input focus
;SDL_WINDOW_MOUSE_FOCUS Tested, generally not relevant
;window has mouse focus
;SDL_WINDOW_FOREIGN
;window not created by SDL
;SDL_WINDOW_ALLOW_HIGHDPI
;window should be created in high-DPI mode if supported (>= SDL 2.0.1)
;SDL_WINDOW_MOUSE_CAPTURE
;window has mouse captured (unrelated to INPUT_GRABBED, >= SDL 2.0.4)

;;:ALLOW-HIGHDPI :BORDERLESS :CENTERED :FOREIGN :FULLSCREEN
;   :FULLSCREEN-DESKTOP :HIDDEN :INPUT-FOCUS :INPUT-GRABBED :MAXIMIZED
;   :MINIMIZED :MOUSE-FOCUS :OPENGL :RESIZABLE :SHOWN

(defun test-sdl-opengl-context (major minor window)
  (progn
    (sdl2:gl-set-attr :context-major-version major)
    (sdl2:gl-set-attr :context-minor-version minor)
    (sdl2:gl-create-context window)))
;;;Tested with multiple context major and minor versions

(defun test-input-events ()
  (let (sdl-window)
  (progn
    (setf sdl-window (sdl2:create-window :title "Event Window"))
    (sdl-windows-windowhack sdl-window)
    (loop with sdl-event = (sdl2:new-event)
       until (eq (sdl2:get-event-type sdl-event) :quit)
```

Library tests

```
        finally (sdl2:free-event sdl-event)
        do (prog2
               (sdl2:next-event sdl-event)
               (cond ((eq (sdl2:get-event-type sdl-event) :keyup) (print "Keyboard used"))
                     ((eq (sdl2:get-event-type sdl-event) :mousemotion) (print "Mouse moved"))
                     ((eq (sdl2:get-event-type sdl-event) :mousewheel) (print "Mouse wheel used"))
                     ((eq (sdl2:get-event-type sdl-event) :mousebuttonup) (print "Mouse button used"))
                     ((eq (sdl2:get-event-type sdl-event) :windowevent) (print "Window event")))))
    (sdl2:destroy-window sdl-window))))

;;;Assimp basic tests (classimp)

;;;Tested with what amounts to a random .obj model
;;; as well as the Stanford Dragon and Stanford Bunny
;;; Filepath either as double \\ or /
(defun test-model-load (filepath processing-flags)
  (ai:import-into-lisp filepath :processing-flags processing-flags))


;;;OpenGL basic tests (cl-opengl)

;;Basic shader tests
(defun test-basic-vertex-shader ()
  (let ((source
         "#version 330 core
          layout (location = 0) in vec3 position;
          void main()
          {
              gl_Position = vec4(position.x, position.y, position.z, 1.0);
          }")
         (vertex-shader))
  (progn
    (setf vertex-shader (cl-opengl:create-shader :vertex-shader))
    (cl-opengl:shader-source vertex-shader source)
    (cl-opengl:compile-shader vertex-shader)
    (print (cl-opengl:get-shader-info-log vertex-shader))
    (return-from test-basic-vertex-shader vertex-shader))))

(defun test-basic-fragment-shader ()
  (let ((source
         "#version 330 core
         out vec4 color;
         void main()
         {
             color = vec4(1.0f, 1.0f, 1.0f, 1.0f);
         }")
         (fragment-shader))
    (progn
      (setf fragment-shader (cl-opengl:create-shader :fragment-shader))
      (cl-opengl:shader-source fragment-shader source)
      (cl-opengl:compile-shader fragment-shader)
      (print (cl-opengl:get-shader-info-log fragment-shader))
      (return-from test-basic-fragment-shader fragment-shader))))
```

Library tests

```
;;;Combination tests (all of the above)

;;;Test function for window and loop
;;;(eq (sdl2:next-event event) 0) this works, at some point there are no events
(defun gametest ()
  (let (window)
  (progn
    (sdl2:init :everything)
    (print "SDL initialised")
    (setq window (create-sdl-window))
    (sdl-windows-windowhack window)
    (print "Window created")
    ;;(sdl2:push-quit-event)
    (loop
        with event = (sdl2:new-event)
        until (eq (sdl2:get-event-type event) :quit)
        finally (sdl2:free-event event)
       do (sdl2:next-event event))
    (print "Out of loop")
    (sdl2:destroy-window window)
    (print "Window destroyed")
    (sdl2:quit))))

;;;Test function for
;;;OpenGL context gameloop
(defun gl-gametest ()
  (let ((window)
        (gl-context))
  (progn
    (sdl2:init :everything)
    (print "SDL initialised")
    (setq window (create-sdl-window))
    (sdl-windows-windowhack window)
    (print "Window created")
    (sdl2:gl-set-attr :context-major-version 3)
    (sdl2:gl-set-attr :context-minor-version 1)
    (print "OpenGL version defined")
    (setq gl-context (sdl2:gl-create-context window))
    (print "OpenGL context created")
    (print (sdl2:gl-get-attr :context-major-version))
    (print (sdl2:gl-get-attr :context-minor-version))
    (cl-opengl:clear-color 1.0 1.0 1.0 1.0)
    (print "OpenGL clear colour set")
    (cl-opengl:clear :color-buffer-bit)
    (sdl2:gl-swap-window window)
    (loop
        with event = (sdl2:new-event)
        until (eq (sdl2:get-event-type event) :quit)
        finally (sdl2:free-event event)
       do (sdl2:next-event event))
    (print "Out of loop")
    (sdl2:gl-delete-context gl-context)
    (print "OpenGL context deleted")
    (sdl2:destroy-window window)
    (print "Window destroyed")
    (sdl2:quit))))
```

Library tests

```
;;;Test function for importing
;;;and having the base loop
(defun import-gametest ()
  (let ((window)
        (gl-context)
        (testmodel))
  (progn
    (sdl2:init :everything)
    (print "SDL initialised")
    (setf window (create-sdl-window))
    (sdl-windows-windowhack window)
    (print "Window created")
    (setf testmodel (ai:import-into-lisp
                     "../Test assets/Test models/AK_SmilingAlien.obj"
                     :processing-flags '(:ai-process-preset-target-realtime-quality)))
    (print "Testmodel loaded")
    (print testmodel)
    (sdl2:gl-set-attr :context-major-version 3)
    (sdl2:gl-set-attr :context-minor-version 1)
    (print "OpenGL version defined")
    (setf gl-context (sdl2:gl-create-context window))
    (print "OpenGL context created")
    (print (sdl2:gl-get-attr :context-major-version))
    (print (sdl2:gl-get-attr :context-minor-version))
    (cl-opengl:clear-color 1.0 1.0 1.0 1.0)
    (print "OpenGL clear colour set")
    (cl-opengl:clear :color-buffer-bit)
    (sdl2:gl-swap-window window)
    (loop
        with event = (sdl2:new-event)
        until (eq (sdl2:get-event-type event) :quit)
        finally (sdl2:free-event event)
       do (sdl2:next-event event))
    (print "Out of loop")
    (sdl2:gl-delete-context gl-context)
    (print "OpenGL context deleted")
    (sdl2:destroy-window window)
    (print "Window destroyed")
    (sdl2:quit))))())

;;;Test function for rendering
(defun render-gametest ()
  (let ((window)
        (gl-context)
        (testmodel)
        (vertex-shader)
        (fragment-shader)
        (basic-shader)
        (gl-array-triangle (cl-opengl:alloc-gl-array :float 9))
        (triangle-vertices
         '(-0.5 -0.5 0.0 0.5 -0.5 0.0 0.0 0.5 0.0))
        (vertex-buffer-object-triangle)
        (vertex-array-object-triangle)
        )
  (progn
    (sdl2:init :everything)
    (print "SDL initialised")
```

Library tests

```
(setf window (create-sdl-window))
(sdl-windows-windowhack window)
(print "Window created")
(setf testmodel (ai:import-into-lisp
                 "../Test assets/Test models/AK_SmilingAlien.obj"
                 :processing-flags '(:ai-process-preset-target-realtime-quality)))
(print "Testmodel loaded")
(print testmodel)
(sdl2:gl-set-attr :context-major-version 3)
(sdl2:gl-set-attr :context-minor-version 3)
(print "OpenGL version defined")
(setf gl-context (sdl2:gl-create-context window))
(print "OpenGL context created")
(print (sdl2:gl-get-attr :context-major-version))
(print (sdl2:gl-get-attr :context-minor-version))
(loop
   for i from 0
   for vertex in triangle-vertices
     do (setf (cl-opengl:glaref gl-array-triangle i) (coerce vertex 'float)))
(setf vertex-shader (basic-vertex-shader))
(setf fragment-shader (basic-fragment-shader))
(setf basic-shader (cl-opengl:create-program))
(cl-opengl:attach-shader basic-shader vertex-shader)
(cl-opengl:attach-shader basic-shader fragment-shader)
(cl-opengl:link-program basic-shader)
(print "Basic shader created")
(cl-opengl:delete-shader vertex-shader)
(cl-opengl:delete-shader fragment-shader)
(print "Cleaned up vertex shader, fragment shader")
(setf vertex-buffer-object-triangle (car (cl-opengl:gen-buffers 1)))
(setf vertex-array-object-triangle (car (cl-opengl:gen-vertex-arrays 1)))
(cl-opengl:bind-vertex-array vertex-array-object-triangle)
(cl-opengl:bind-buffer :array-buffer vertex-buffer-object-triangle)
(cl-opengl:buffer-data :array-buffer :static-draw gl-array-triangle)
(cl-opengl:vertex-attrib-pointer 0 3 :float nil 0 (cffi:null-pointer))
(cl-opengl:enable-vertex-attrib-array 0)
(cl-opengl:bind-vertex-array 0)
(cl-opengl:clear-color 0.0 0.0 0.0 0.0)
(print "OpenGL clear colour set")
(cl-opengl:clear :color-buffer-bit)
(cl-opengl:bind-vertex-array vertex-array-object-triangle)
(cl-opengl:use-program basic-shader)
(cl-opengl:draw-arrays :triangles 0 3)
(sdl2:gl-swap-window window)
(loop
    with event = (sdl2:new-event)
    until (eq (sdl2:get-event-type event) :quit)
    finally (sdl2:free-event event)
   do (sdl2:next-event event))
(print "Out of loop")
(cl-opengl:free-gl-array gl-array-triangle)
(cl-opengl:delete-program basic-shader)
(print "Shader deleted")
(sdl2:gl-delete-context gl-context)
(print "OpenGL context deleted")
(sdl2:destroy-window window)
(print "Window destroyed")
```

Library tests

```
    (sdl2:quit)))))

;;;Function with model rendered in it, in this case the Stanford Dragon
(defun model-gametest ()
  (let ((window)
        (gl-context)
        (testmodel)
        (vertex-shader)
        (fragment-shader)
        (model-shader)
        (gl-array-model)
        (gl-array-indices)
        (model-vertices)
        (model-indices)
        (vertex-buffer-object-model)
        (vertex-array-object-model)
        (index-buffer-object-model)
        )
    (progn
      (sdl2:init :everything )
      (print "SDL initialised")
      (setf window (create-sdl-window))
      (sdl-windows-windowhack window)
      (print "Window created")
      (setf testmodel (ai:import-into-lisp
                       "../Test assets/Test models/dragon_recon/dragon_vrip_res3.ply"
                       :processing-flags '(:ai-process-preset-target-realtime-quality)))
      (print "Testmodel loaded")
      (sdl2:gl-set-attr :context-major-version 3)
      (sdl2:gl-set-attr :context-minor-version 3)
      (setf gl-context (sdl2:gl-create-context window))
      (print (sdl2:gl-get-attr :context-major-version))
      (print (sdl2:gl-get-attr :context-minor-version))
      (loop
         for vertex in (coerce (ai:vertices (aref (ai:meshes testmodel) 0)) 'list)
         do (setf model-vertices (nconc model-vertices (coerce vertex 'list))))
      (loop
         for face in (coerce (ai:faces (aref (ai:meshes testmodel) 0)) 'list)
         do (setf model-indices (nconc model-indices (coerce face 'list))))
      (setf gl-array-model (cl-opengl:alloc-gl-array :float (list-length model-vertices)))
      (setf gl-array-indices (cl-opengl:alloc-gl-array :int (list-length model-indices)))
      (loop
         for i from 0
         for vertex in model-vertices
         do (setf (cl-opengl:glaref gl-array-model i) (coerce vertex 'float)))
      (loop
         for i from 0
         for index in model-indices
         do (setf (cl-opengl:glaref gl-array-indices i) (coerce index 'integer)))
      (print "Model vertices and indices set")
      (setf vertex-shader (test-basic-vertex-shader))
      (setf fragment-shader (test-basic-fragment-shader))
      (setf model-shader (cl-opengl:create-program))
      (cl-opengl:attach-shader model-shader vertex-shader)
      (cl-opengl:attach-shader model-shader fragment-shader)
      (cl-opengl:link-program model-shader)
      (cl-opengl:delete-shader vertex-shader)
```

Library tests

```
(cl-opengl:delete-shader fragment-shader)
(setf vertex-buffer-object-model (car (cl-opengl:gen-buffers 1)))
(setf vertex-array-object-model (car (cl-opengl:gen-vertex-arrays 1)))
(setf index-buffer-object-model (car (cl-opengl:gen-buffers 1)))
(cl-opengl:bind-vertex-array vertex-array-object-model)
(cl-opengl:bind-buffer :array-buffer vertex-buffer-object-model)
(cl-opengl:buffer-data :array-buffer :static-draw gl-array-model)
(cl-opengl:bind-buffer :element-array-buffer index-buffer-object-model)
(cl-opengl:buffer-data :element-array-buffer :static-draw gl-array-indices)
(cl-opengl:enable-vertex-attrib-array 0)
(cl-opengl:vertex-attrib-pointer 0 3 :float nil 0 (cffi:null-pointer))
(cl-opengl:bind-vertex-array vertex-array-object-model)
(print "OpenGL buffers set")
(cl-opengl:clear-color 0.0 0.0 0.0 0.0)
(cl-opengl:clear :color-buffer-bit)
(cl-opengl:use-program model-shader)
(cl-opengl:draw-elements
 :triangles (cl-opengl:make-null-gl-array :unsigned-int)
 :count (list-length model-indices))
(sdl2:gl-swap-window window)
(loop
   with event = (sdl2:new-event)
   until (eq (sdl2:get-event-type event) :quit)
   finally (sdl2:free-event event)
   do (sdl2:next-event event))
(print "Out of loop")
(cl-opengl:delete-program model-shader)
(print "Shader deleted")
(sdl2:gl-delete-context gl-context)
(print "OpenGL context deleted")
(sdl2:destroy-window window)
(print "Window destroyed")
(sdl2:quit))))

;;;Basic build test (for building demo)

;;; This is intended to be its own script to be called when invoking Lisp
;;; It's assumed that the code for main is loaded before this
(save-lisp-and-die "Demo.exe"
                   :toplevel #'main
                   :executable t
                   :application-type :console)
```