UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor of Computer Engineering

BACHELOR THESIS

Graphics Engine Built on OpenGL in Common Lisp

Author: Daniel Litvak
Advisor: Francisco Jurado Monroy

June 2023

Daniel Litvak
Graphics Engine Built on OpenGL in Common Lisp

Daniel Litvak

PRINTED IN SPAIN

To my parents

Never forget what you are, for surely the world will not.
Make it your strength. Then it can never be your weakness.
Armour yourself in it, and it will never be used to hurt you.

George R. R. Martin, A Game of Thrones

Preface

In this preface, I wish to provide some context for the journey that led to the development of this graphics engine. As a lifelong video game enthusiast with a penchant for programming, I've always had an interest in developing graphics software. Be it just for the enjoyment of getting visual results from writing code, or working towards a larger goal like publishing a video game.

As for why I chose Common Lisp as a programming language, I could spell out many reasonable and pragmatic arguments, but the truth is that ever since I learned how to use it, I've been looking for a project to use it in.

The development of this project was not an easy task. However, it was driven by personal curiosity, and a desire to understand and craft the low-level aspects of a graphics engine. Apart from being an academic project, it has been a project of passion.

In sharing this work, I hope to provide insight into the process of developing a graphics engine from scratch. May it inspire other enthusiasts to embark on their journey of exploring the fascinating intersection of programming, graphics, and gaming.

Acknowledgements

Thank you David and Ineke, my parents, for always sticking by me and for always believing in me.

Resumen

Este trabajo de fin de grado presenta el desarrollo de un motor gráfico programado en Common Lisp, haciendo uso de las capacidades de OpenGL. El objetivo principal del proyecto era diseñar e implementar un software funcional y sencillo de usar, para renderizar escenas 3D. Estuvo motivado por un deseo personal de tener una herramienta versátil y directa para la programación gráfica, sin tener que aprender a utilizar un editor o framework complejo.

El motor presenta una interfaz sencilla de utilizar para cargar y renderizar mallas de ficheros, generar mallas procedurales con ruido Perlin, enlazar texturas a mallas, y simular iluminación.

A pesar de los desafíos que existen en la programación gráfica, el proyecto fue exitoso a la hora de producir un motor gráfico funcional. Se mantuvo dentro del umbral definido y se implementaron todos los requisitos que se especificaron.

En conclusión, el proyecto contribuye significativamente al campo de la programación gráfica, específicamente dentro de la comunidad de Common Lisp. Ofrece una herramienta que además de cumplir con sus objetivos, presenta una plataforma excelente para futuras mejoras o expansiones.

Palabras clave

Common Lisp, Motor gráfico, Programación de gráficos, OpenGL Videojuegos

Abstract

This bachelor thesis presents the development of a graphics engine written in Common Lisp, leveraging the capabilities of OpenGL. The primary goal of the project was to design and build a functional, user-friendly engine to render 3D scenes. The project was personally motivated by the desire to have a flexible and direct tool for graphics programming without the need to learn a complex framework or editor.

The engine provides a simple-to-use interface for loading and rendering meshes from files, generating procedural meshes using Perlin noise, mapping textures to meshes, and simulating lighting.

Despite the challenges inherent in graphics programming from scratch, the project was successful in producing a working graphics engine, adhering closely to the initial scope and achieving the planned feature set.

This project thus makes a significant contribution to the field of graphics programming, especially within the Common Lisp community, by providing a tool that not only meets the project's goals but also offers an excellent platform for further expansion and enhancement.

Keywords

Common Lisp, Graphics engine, Graphics programming, OpenGL, Videogames

Table of Contents

Lists

List of codes

List of figures

1 Introduction

Computer graphics and 3D rendering have rapidly evolved in the last few decades. This evolution has led to sophisticated tools and engines. While these resources are powerful, they can often be complex and challenging for developers, especially those who wish to rapidly prototype or focus primarily on rendering. As an example, the third edition of *Game Engine Architecture* is over one thousand pages long [1], covering a wide range of distinct topics. This thesis presents a project focused on creating a streamlined and intuitive graphics engine, based on OpenGL and written in Common Lisp, prioritizing ease-of-use for the developer.

1.1 Project Purpose

The purpose of this project is not to compete with industry-standard graphics or game engines. Rather it seeks to provide developers with a simple and accessible tool set, that empowers them to dive straight into rendering. The ubiquity of OpenGL ensures that the resulting software is likely to be cross-platform [2]. The flexibility and the powerful macro system of Common Lisp allow for an intuitive library interface [3].

1.2 Motivation

The inspiration for this project stems from an interest in computer graphics, and an appreciation for Common Lisp's programming style. Whenever I would try to start on a graphics project, I would be faced with two possibilities. Either learn how to use a complex engine, or start completely from scratch developing against a low level Application Programming Interface (API). Upon the completion of this project, I hope to have a tool that will allow me to pursue those interests.

Originally the Engine was going to be written in Rust, because of its many stated benefits like a strong type system, borrow and move semantics, and its ability to produce highly performant code. However after some months working on the project, Rust's type system proved too rigid for the flexibility and freedom I was looking for. At that point, I decided to rewrite what I had up to that point in Common

Lisp, and continued working from there.

1.3 Project Scope

The main purpose of the engine is to render 3D scenes. Nevertheless, it should be possible to use the tools to render 2D scenes as well, though using the engine in this way is left as an exercise to the reader. In order to provide comprehensive tools for developers, the scope of the project encompasses the following aspects of 3D rendering:

- **Bitmap Font Rendering.** A very helpful module that will allow a user to render debug information directly on the screen in real time.
- **3D Model Rendering.** The most essential feature for rendering 3D scenes. The engine is able to import 3D meshes from Wavefront OBJ files and render them.
- **Texture Mapping.** Furthermore, users can import PNG texture files, and map them to meshes for further detail and aesthetics.
- **Lighting.** In order to provide a better depth perception, the Phong reflection model is implemented to model different types of lights.
- **Procedural Generation.** While not a fundamental aspect of 3D rendering, algorithms like Perlin Noise are implemented to allow a user to render arbitrarily large or detailed scenes.

Note that the scope of the project is far from exhaustive. There are many other aspects of graphics rendering like shadow mapping, physics based rendering, physics simulations and many others [4]. While they are outside of the scope of this project, the hope is that the implementation will allow these features to be integrated in future iterations without much additional effort.

1.4 Organization

This document is organized in the following sections, each focused on a different aspect of the project.

1.– **Introduction.** Provides an overview of the project, explaining its purpose and scope. Sets the stage for the rest of the thesis and provides context for the work.

2.– **State of the Art.** Describes the current landscape of graphics engines in general, OpenGL as a graphics API, and Common Lisp as a programming language. Finally it highlights existing work in the space.

3.– **Methods and Materials.** Outlines the different aspects of the developer experience during the evolution of the project. Gives readers an understanding of the project setup.

4.– **Design and Implementation.** Describes the steps taken during design and development of the graphics engine, going into specifics about integrating OpenGL and implementing aspects of a 3D renderer.

5.– **Testing.** Documents the testing approaches taken to ensure the engine is functional and useful.

6.– **Discussion and Analysis.** Interprets the results of the project.

7.– **Conclusion and Future Work.** Summarizes the project’s findings and offers some suggestions for possible future work.

2 State of the Art

First I will present some general background about different aspects of this project. Once this background has been established, the final section will mention the existing work in the intersection of these worlds.

2.1 Graphics Engines Basics

Graphics engines are some of the most complex pieces of software that exist. This is because of the many systems they tend to implement, and these systems having to integrate well with each other. As a consequence, they have become an important part of a wide array of applications and industries. Scene graphs and scripting systems allow for rapid development. Developers can use detailed lighting and texturing models for virtual reality visualizations or architectural design. In short, any type of creator who wants to express their idea in some three-dimensional way, will likely use a graphics engine. The following is a more detailed list of the different systems that can be available in such an engine. Some of these systems could qualify a piece of software as a game engine rather than just a graphics engine.

- **Resource Management.** In charge of loading assets from storage into memory, and mapping them onto the relevant data structures the engine uses.
- **Meshes.** Holds references to 3D models and the GPU buffers where they are stored. Also allows a program to manipulate a mesh, so that a model can change shape or be animated.
- **Lighting.** A collection of shaders that get compiled into a rendering pipeline. This pipeline instructs the GPU how to render a mesh.
- **Sound.** An API to define different sound sources, and mix them together.
- **Animation.** Handles keyframes and different mesh states to animate between them.
- **Networking.** A higher level abstraction on top the operating system's network stack, to share game state over the network for different players.
- **Input Handling.** Can be done by specifying callbacks that should be run on any input event, or by polling the engine for the current input state at a fixed rate.
- **Physics.** Many subsystems actually comprise the physics system. Some of them are detailed below.
  - Rigid body physics. Models meshes as rigid bodies that react to gravity and collide with each other.
  - Soft body physics. Models meshes as malleable bodies that can deform upon collision.

    - ○ Cloth simulations. Model a plane as a piece of cloth that can be deformed and ripped.
    - ○ Fluid simulations. Model a collection of points as if they were particles in a fluid.
- Navigation. Offers implementations of navigation meshes and agents to easily model autonomous behavior.

This list is again by no means exhaustive, many other features are made available by some engines.

2.2 Graphics Libraries

A graphics library is a collection of abstractions, which allow developers to make use the capabilities of rendering hardware, such as graphics cards. Many such libraries exist, ranging from low level APIs that only offer rendering capabilities, to higher level ones that provide additional functionality. To provide one example of a high level library, BGFX [5] is a “cross-platform, graphics API agnostic rendering library”. Graphics API agnostic means that it targets many low level graphics APIs.

Most of these libraries and APIs are implemented for many programming languages, and when they aren’t, bindings are usually available. In the case of Common Lisp however, OpenGL is the only API available with a somewhat actively maintained library. Some others exist, and those will be touched upon in section 2.4.

2.2.1 OpenGL

OpenGL [2] is a cross-language, cross-platform graphics API for rendering vector graphics. By the nature of the specification, it has support for 2D and 3D graphics, but the library itself doesn’t distinguish those.

In order to function, an OpenGL context needs to be instantiated. This context then maintains state, which will instruct the graphics hardware how to operate. To render a mesh for example, the context needs to be manipulated into the corresponding state, and then a rendering command can be executed.

OpenGL example

Working directly with this API is very cumbersome, especially as the scale of a project grows. As an example, here follows a program for rendering a single triangle, which could be considered the simplest possible OpenGL program. The example is implemented using the `cl-opengl` [6] library used for the rest of this project.

Before the program can even start, mesh data and shader source code needs to be imported. In the case of this example it has been hard-coded (code 2.1). A detailed listing of the shader source code can be found in codes A.4 and A.5.

Code 2.1: Mesh data and vertex source code

```
;; A mesh represented as a list of points and a list of indices
(defparameter *vertices* #(-0.5 -0.5 0.0 0.5 -0.5 0.0 0.0 0.5 0.0))
(defparameter *indices* #(0 1 2))

;; Shader source code
(defparameter *vs-source* "#version␣330␣core\n␣...")
(defparameter *fs-source* "#version␣330␣core\n␣...")
```

To start off, a window needs to be created and an OpenGL context initialized. Furthermore, many OpenGL objects need to be generated to contain our data and state. In code 2.2 the Vertex Buffer Object (VBO) and Element Buffer Object (EBO) will hold vertex data and vertex indices respectively. Variables also need to be created to store references to the individual shaders and the shader program.

Code 2.2: Window creation and OpenGL object generation.

```
;; Needed for interacting with OS windows
(glfw:with-init
 ;; Creates a window and the OpenGL context
 (let* ((window (glfw:create-window :width 800
                                    :height 600
                                    :title "Example"
                                    :context-version-major 3
                                    :context-version-minor 3
                                    :opengl-profile
                                    :opengl-core-profile))
        ;; Create some OpenGL objects to store data
        (vbo (gl:gen-buffer))
        (ebo (gl:gen-buffer))
        (vao (gl:gen-vertex-array))

        ;; Create shader objects
        (vs (gl:create-shader :vertex-shader))
        (fs (gl:create-shader :fragment-shader))
        (shader-program (gl:create-program)))
```

Code 2.3 takes care of compiling the individual shaders and linking them together in a shader program. Error checking and handling is omitted in this example, but would add additional code and complexity. Code 2.4 sets up the buffers generated earlier to hold mesh data. Furthermore, the Vertex Array Object (VAO) is configured so that the vertex layout can be understood by the shaders.

Now that all the data is set up, and the OpenGL context has the correct state, some actual rendering can be done. In code 2.5 the only actual function that instructs the Graphics Processing Unit (GPU) to render anything, is `gl:draw-elements` on line 65. The rest of the code surrounding it is again needed to manipulate state.

Code 2.3: Shader compilation and linking

```
;; Bind and compile shader source code
(gl:shader-source vs *vs-source*)
(gl:compile-shader vs)

(gl:shader-source fs *fs-source*)
(gl:compile-shader fs)

;; Link compiled shaders to shader program
(gl:attach-shader program vs)
(gl:attach-shader program fs)
(gl:link-program program)
```

Code 2.4: Mesh setup.

```
;; Bind our Vertex Array Object so we can specify our mesh layout
(gl:bind-vertex-array vao)

;; Bind the buffers we created to the vao, specifying their type
(gl:bind-buffer :array-buffer vbo)
(gl:bind-buffer :element-array-buffer ebo)

;; Copy the vertex and index data into the buffers
(gl:buffer-data :array-buffer :static-draw (array-to-gl-array *vertices*))
(gl:buffer-data :element-array-buffer :static-draw (array-to-gl-array *indices*))

;; Specify Vertex Attribute Pointers
(gl:vertex-attrib-pointer 0 3 :float :false (* 3 4) 0)
(gl:enable-vertex-attrib-array 0)
```

Finally, the code in listing 2.6 cleans up the OpenGL context. The result of executing all this code can be seen in figure 5.1. Hopefully this example shows why a graphics engine with proper abstractions is not only useful, but an absolute necessity for even the most basic graphics programming.

Code 2.5: The actual rendering code. Of these lines, the only part that actually instructs OpenGL to render is the function call on line 65. The other functions only take care of getting the OpenGL context in the proper state. The final two lines instruct that the OS to draw the rendered framebuffer the window.

```
56  (loop until (glfw:window-should-close-p window)
57    do (progn
58         ;; Set the shader program we are using
59         (gl:use-program shader-program)
60
61         ;; Bind the Vertex Array Object for our mesh
62         (gl:bind-vertex-array vao)
63
64         ;; Draw triangles with the current state
65         (gl:draw-elements :triangles
66               (gl:make-null-gl-array :unsigned-int)
67               :count (length *indices*))
68
69         ;; Unbind our vertex array
70         (gl:bind-vertex-array 0)
71
72         ;; Tell the OS to draw the framebuffer to the screen
73         (glfw:swap-buffers window)
74         (glfw:poll-events)))
```

Code 2.6: These final lines free up the OpenGL objects that were created at the beginning of the program.

```
76  (gl:delete-buffers (list vbo ebo))
77  (gl:delete-vertex-arrays (list vao))
78  (gl:delete-program shader-program)
79  (gl:delete-shaders vs fs)))
```

2.3 Common Lisp

Common Lisp exists as a dialect in a large family of Lisp programming languages. It is important to note that Common Lisp itself is not a language implementation, but rather a language specification. It has been published in ANSI standard *ANSI INCITS 226-1994 (S20018)* [7]. There is also the Common Lisp HyperSpec, an HTML version which has been derived from the standard with permission from ANSI and X3.

There are many features that set Common Lisp apart from other programming languages. Here are some that were particularly helpful for this project. Note that for the rest of this document, Common Lisp and lisp will be used interchangeably.

2.3.1 REPL driven development

The first thing that stands out from Common Lisp is that there is no real concept of “program”. Rather when you start the interpreter, you are faced with a Read-Evaluate-Print Loop (REPL). In this environment you can execute any lisp code, such as in listings 2.7 and 2.8. By default, any new functions or data structures you define will be confined to this REPL. There are two different ways to make your code persistent. The first is by saving the current state of the REPL to an image, which can be booted up at a later date to continue working with it. The second one, and the one used for this project, is by using an Integrated Development Environment (IDE) capable of linking code you write in files, with an existing REPL session. The benefits of the second approach are that you get to have a static copy of the code you write, which is beneficial for source control and collaboration. The setup used for this project will be explained in more detail in section 3.3.

Code 2.7: Some basic lisp functions being executed. In this case the + and `nth` functions

```
> (+ 1 2 3)
6

> (nth 2 '("foo" "bar" "baz" "quz"))
"baz"
```

Code 2.8: This function definition would be lost if the proper steps aren't taken before closing the REPL session.

```
> (defun fact (n)
    (if (eq n 0)
    1
        (* n (fact (1- n)))))
FACT
```

One major consequence of how the Common Lisp REPL works, is that you get code reloading for free. What this means is that while the code is running, you can modify a specific function, recompile it, and see the changes take effect without having to restart the entire program. This is also very useful for debugging, since whenever an error is encountered, the lisp debugger will allow you to fix it while execution is paused.

Another way this interactive development experience turned out useful for this project, is that rendering code could be modified while the application is running. This isn't always helpful since certain code in the engine only gets run once, but for code that is run every frame, it was possible to get immediate feedback without having to restart the application.

2.3.2 Homoiconicity

Homoiconicity is a property of certain programming languages, where the program structure can be manipulated as data in the same language. In the context of Common Lisp, this property emerges from the use of S-expression. The same notation is used to represent lisp code, such as function invocations, as well as the main lisp data structure, the list. For a more concrete example refer back to listing 2.7, where an Symbolic Expression (S-expression) is used for the `nth` function invocation, and also to define its second argument.

One other way this manifests in Common Lisp is in its powerful macro system. Since code and data are treated the same, no separate macro language is needed. This makes writing macros very easy.

2.3.3 Computer Graphics history

A final interesting point about Common Lisp is that it has a rich history in Computer Graphics. Lisp machines build in the 1980s by companies like Symbolics were some of the first capable of real time video I/O, and the first to produce HDTV video [8].

2.4 Previous Work

The previous sections of this chapter give context about the current state of the tools we're using. This next section discusses existing work at the intersection of Graphics Engines, OpenGL and Common Lisp. In this small slice, there are a few software packages that have to be mentioned.

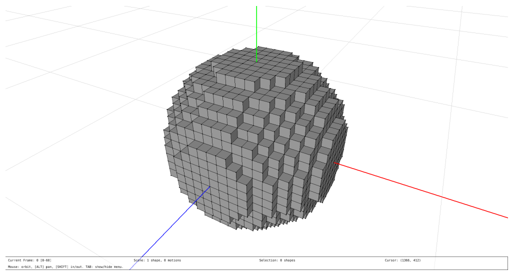
2.4.1 cl-opengl

`cl-opengl` [6] is a Common Lisp library that provides bindings for the OpenGL graphics API. The way it achieves this is using the Common Foreign Function Interface (CFFI). Besides offering low level bindings directly the OpenGL API, this library also creates its own interface on top of it. This higher level interface is very desirable, since it prevents us from having to call foreign functions directly, which generally means working with dynamic memory directly [9].

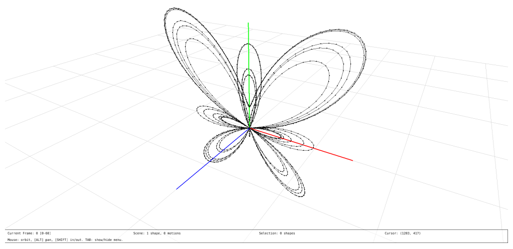
This library also includes bindings to the OpenGL Utility Toolkit (GLUT) API, however in this project `cl-glfw` is used instead, which provides bindings to Graphics Library Framework (GLFW). The reason for this is that GLFW gives full control of the render loop to the developer.

2.4.2 kons-9

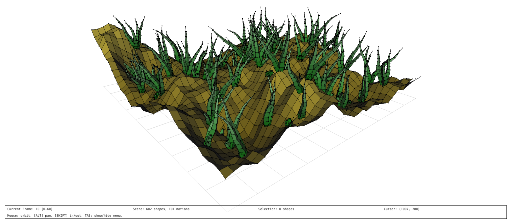
`kons-9` [10] is somewhat unique in this list because it is a fairly new project. It's self described as falling under the category of 3D digital content creation tool. It provides a flexible and extensible system to create abstract and arbitrary 3D visualizations. Its unique feature is that it combines the REPL-driven development workflow provided by Common Lisp with the visual tools of a 3D graphics system. Some examples of visuals generated using `kons-9` are shown in figure 2.1.



(a) A voxel sphere



(b) Bezier curves in the shape of a butterfly



(c) Procedural organic growth

Figure 2.1: Some visuals generated using kons-9

2.4.3 Trial

Trial [11] is a game engine for Common Lisp. It's structured as a loose connection of components that can be composed in any combinations, depending on the requirements of any particular game. Trial stands out in the Common Lisp ecosystem because it has been used to develop and publish full-fledged video games [11].

2.4.4 clinch

Clinch purports itself as *"a simple, yet powerful 3D game engine for Lisp"*. It is written on the same `cl-opengl` library as this project, although its feature-set is quite more extensive. The main features it offers that are not present in this engine are the following [12]:

- **3D physics with joints and motors**
- **2D vector graphics**
- **Animation of textures and 3D objects**

The library's homepage mentions that the project is still under development, however no changes have been uploaded since 2017 as of the writing of this text.

2.4.5 Wrapper libraries

Besides `cl-opengl`, there are some other libraries acting as wrappers by way of CFFI bindings. Examples include bindings for Horde3D [13] and Ogre [14]. Most of these libraries have been unmaintained for many years though.

3 Methods and Materials

This chapter provides an in-depth exploration of the strategies and resources employed during the development of the project. It also discusses the hardware and software requirements for developers wishing to work with the engine.

3.1 Software Development Life Cycle

As the only developer of this project, I approached the development cycle in a way that aligned with my personal style and habits. While no official methodology was followed, the closest approximation would be Iterative and Incremental Development. The idea behind this method is to have a system evolve over the course of small cycles. Even though it wasn't strictly followed, it still makes sense to talk about this project's lifecycle using the nomenclature this method provides.

3.1.1 Initialization step

The initialization step is meant to create a base version of the software system. If the entire project were reduced to small iteration cycles, there would be nothing to start from. For this project, this step consisted of implementing parts of *Learn OpenGL* [4] in Common Lisp. This allowed me to learn the basics of OpenGL, while at the same time working towards a base state for the project.

3.1.2 Project control list

The project control list is intended to guide the iteration process, by keeping a record of the tasks that need to be performed, and goals that wish to be achieved. It's made up of the items below. Furthermore, any features that were deemed necessary to implement one of these items were implemented on the spot during the respective cycle.

- **Rendering bitmap fonts.**
- **Rendering meshes.**

- **Phong lighting.**
- **Loading meshes from files.**
- **Mini-framework for easy bootstrapping.**
- **Generating procedural terrain.**
- **Generating procedural 3D environments.**

3.2 Hardware and Software Requirements

When it comes to software requirements, there isn't much to say. Any modern operating system capable of running an ANSI compliant Common Lisp implementation, should be capable of setting up the development environment and making modifications to it. Besides that, in order to actually run the final programs, necessary graphics drivers should be installed with support for the desired OpenGL version.

On the hardware side, so long as the rendering hardware has drivers with support for OpenGL, it will at least run the programs built with this engine. The minimum performance requirements for the hardware will depend on the complexity of the rendered scenes.

3.3 Common Lisp Programming Environment

3.3.1 Package management

Like many programming languages, Common Lisp does not have a native build and package management system. I used two pieces of software to help with that. The first is Quicklisp, a library manager. It is supported by many Common Lisp implementations, and provides access to over 1500 libraries. [15] In order to actually specify the libraries needed though, I used Another System Definition Facility (ASDF). It is the "*de facto* build facility for Common Lisp" [16]. With it, you can describe how a lisp project is organized, and how it should be built and loaded.

Together, Quicklisp and ASDF take care of downloading necessary dependencies, loading their packages so that their functions can be used, building and running the software that you write, and finally they can compile and link a standalone binary that can be run its supported platforms.

3.3.2 Integrated Development Environment

As mentioned in section 2.3, Common Lisp is developed against a REPL. For this purpose, I set up the Emacs editor to act as an IDE. Emacs is well-suited for Common Lisp development, since it is built with a different flavor of lisp, Emacs Lisp. Apart from that, there exist several packages for Emacs that integrate the editor with the REPL. I chose to install SLY, because of its many useful debugging features. I've included my full Emacs configuration in appendix B for reference.

3.3.3 SLY

The helpfulness of SLY is easily understated, so this section is dedicated to explaining some of its features in detail [17].

Full-featured REPL

The default REPL provided by most Lisp implementations is very bare. SLY provides a full-featured Command Line Interface (CLI). The additional functionality includes completion, reverse i-search and bash-style keyboard navigation.

Code reloading

As mentioned in section 2.3, Common Lisp supports reloading code during the execution of a program. SLY makes this extra easy by providing an Emacs command, `sly-compile-defun` (bound to `C-c C-c`, which will compile and load code currently under the cursor. There are many other commands for loading Lisp code that are explained in detail in the SLY manual [18].

Debugger

Whenever an error is encountered in a Lisp program, execution will pause and a debugger will show up in the REPL. SLY improves on this by letting the developer explore the full execution stack, inspect and modify variables, and evaluate arbitrary Lisp code before continuing or aborting execution.

Stickers

Stickers are similar to breakpoints in other debuggers, but they are set for arbitrary expressions. When they are enabled, SLY will pause execution after evaluating a marked expression, allowing you to inspect the result without having to create intermediary variables.

4 Design and Implementation

This chapter details the design decision that were made during the development lifecycle, and the implementation details for each of the engine's systems and modules.

4.1 Design

Based on the project scope and the project control list defined in section 3.1, the following requirements can be derived.

4.1.1 Functional requirements

FREQ1 Load meshes from files

FREQ2 Render meshes

FREQ3 Apply shading to meshes using the Phong reflection model

FREQ4 Support different types of light sources

FREQ5 Map textures to meshes

FREQ6 Transform meshes

FREQ7 Sample Perlin noise

FREQ8 Generate fractal Perlin noise

FREQ9 Render debug information on the screen using bitmap fonts

4.1.2 Non-functional requirements

NFREQ1 The resulting interface should be easy to use

NFREQ2 The engine should be platform independent

NFREQ3 The engine should show clear errors when they are encountered

4.2 System Architecture

To keep with the spirit of building a minimalist graphics engine, the architecture has been designed as a set of interfaces, bundled together in a library. The primary entry point into the engine is the `with-graphics` macro. By itself however, it will only create a window and open it. Everything else in the engine is optional to use, and the remaining sections in this chapter will describe each of these modules in detail. A high level overview of the internal architecture is available in figure 4.1

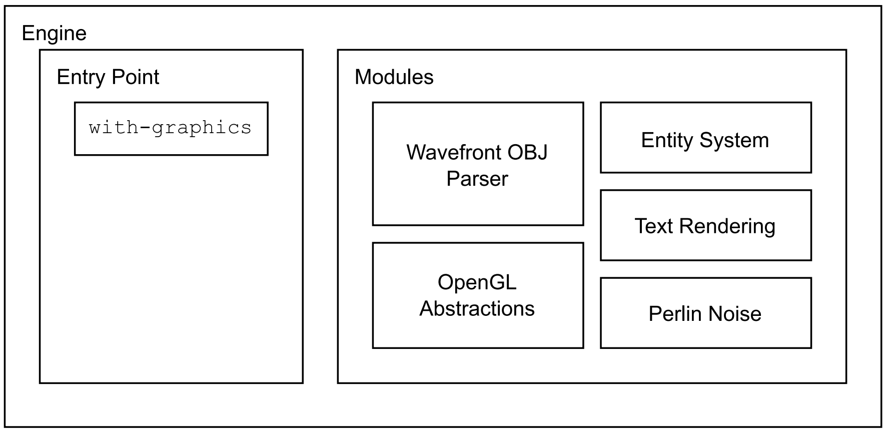


Figure 4.1: The engine architecture is split into two parts. One is the engine entry point, the `with-graphics` macro, and the other contains all the implemented modules.

4.3 OpenGL Integration with Common Lisp

The integration of OpenGL could be considered the core of the engine architecture.

4.3.1 with-graphics macro

The finished engine provides a macro called `with-graphics`, that takes care of the following steps. A visual overview can be seen in figure 4.2

OpenGL and GLFW initialization

GLFW is used to create an operating system window, and an OpenGL context to go with it. Context version 4.5 is specified to enable modern OpenGL features like rendering and compute shaders.

Graphics class

A graphics object is created to hold all the state relating to the engine. This includes the window size, current keyboard state, global objects like the debug font, whether debug mode is enabled, etc. This

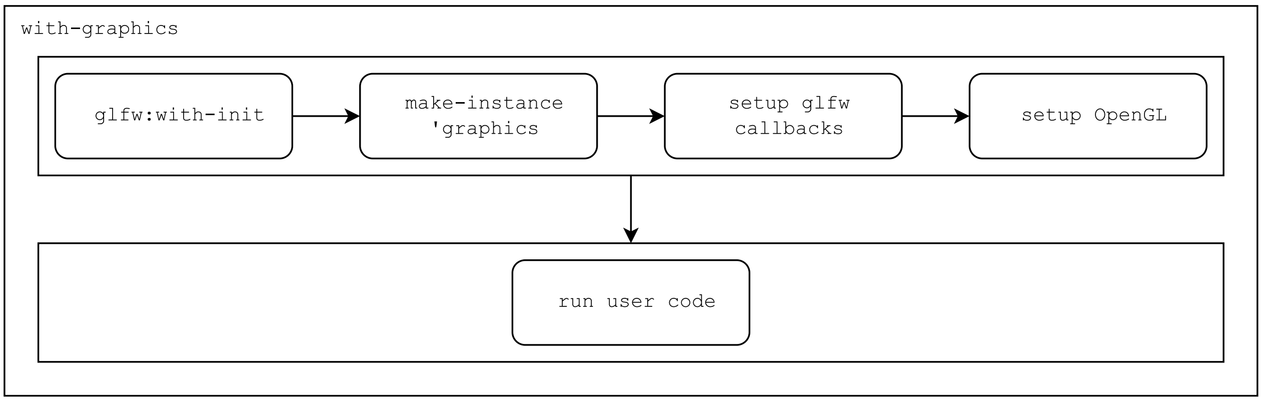


Figure 4.2: The different steps taken by the `with-graphics` macro before it starts running the user code

object is then exposed in the scope of the expanded macro, to allow any program to interact with it.

This object also has many methods which abstract away some of the complexities of working with OpenGL and GLFW. For example, creating and assigning GLFW callbacks is a multiple step process, because of the nature of the CFFI implementation. The `graphics-set-key-callback` method and its siblings, take care of this complexity for the developer.

OS Callbacks

GLFW is used to setup some basic callbacks to integrate with the Operating System (OS). The most important is the resize callback, which updates the window size in the graphics object. Another important one is setting up a keyboard callback to quit when the escape key is pressed. This can be overridden by a program using the macro, but before this is done, it's an easy way to quit a program during the development phase.

4.3.2 Data structures and buffers

Another small but important aspect of integrating OpenGL is how to move data between Common Lisp and OpenGL buffers. For that purpose the function `make-gl-array` and the macro `with-gl-array` were written. The function takes care of allocating memory for a C-style array, and copying data from a lisp array to it. The macro allows you to create a contained scope where that array exists, so you can use it to move data to a buffer or do other manipulations, and afterwards frees the memory that was allocated to it.

4.3.3 OpenGL error handling

Because of the state-machine-like nature of OpenGL, in many cases when it encounters an error it won't throw an exception or return an error code like in other languages or libraries. Rather, the state is modified to reflect the error, and it is the developer's responsibility to check the state for any errors.

When programming shaders, it is very easy to slip up and make a mistake. Since the result of a shader that couldn't compile is usually just a blank screen, utilities for error-checking are very helpful.

The `gl-assert` macro takes care of querying the current OpenGL state, and in case it detects an error, queries the respective error and triggers a Common Lisp error, that can be inspected and analyzed in the debugger.

4.4 3D Modeling and Rendering

To represent 3D models in software, and storing them in a way that can be understood by OpenGL, many different parts were written that all work together.

4.4.1 Vertex Primitives

In order to render an object, we need to know its visual properties. The best way to represent this is by sampling a series of points on the object's surface. If we group the visual properties of these points together, we get vertices. When talking about these properties in this context, we call them vertex attributes [1]. There are many attributes that are typically included, but in this engine we only use the following

- **Position vector** $(p_i = [p_{ix} \quad p_{iy} \quad p_{iz}])$. The position of this vertex in 3D space. This is usually relative to the object's origin, or what is also called *model space*.
- **Normal vector** $(p_i = [n_{ix} \quad n_{iy} \quad n_{iz}])$. This vector defines the surface normal at this point. It's used for dynamic lighting calculations.
- **Diffuse color** $(d_i = [d_R \quad d_G \quad d_B])$. This vector describes the diffuse color of the surface at this point, expressed in the RGB color space. If we were to implement blending in this project, we would also include an *alpha* value here.
- **Texture coordinates** $(t_{ij} = [u_{ij} \quad vij])$. This vector maps each vertex onto a texture. One way to visualize this is to imagine the object unwrapping, and then flattening onto a 2D plane.

Vertex formats

Depending on the mesh we want to render, we will want to include some vertex attributes in the vertices while omitting others. For example, when rendering a detailed 3D model, it is desirable to keep as much of that detail as possible in a texture, and then use a texture coordinate attribute to map that detail onto our mesh. If however, we are generating some procedural meshes like fractals, it might make more sense to use a diffuse color attribute, since all our detail will be generated at the same time as the mesh.

Keeping all this in mind, the way this is implemented is by defining a single struct (code 4.1). This struct will contain fields for all supported attributes. When writing the vertices to an OpenGL buffer, unneeded attributes will be skipped and the VAO will only have the relevant attributes mapped to attribute pointers.

Code 4.1: Vertex Struct Definition

```
(defstruct vertex
  position
  normal
  diffuse-color
  tex-coords)
```

4.4.2 Meshes

A mesh is nothing more than a collection of vertices. The vertices can be described in many ways, but the most common is to do so by triangles. When building a mesh up through triangles, you are guaranteed that all faces will be flat, since three distinct, non-co-linear points always describe a plane.

In OpenGL there are many ways to represent meshes in its state. The easiest way is to just have a VBO with all vertices in the correct order. This way quite inefficient though, since any vertex that is shared among multiple faces will appear multiple times in the buffer, taking up valuable space in memory.

$$\boxed{VBO = \{V_0, V_1, V_2, V_1, V_3, V_2, \dots\}} \tag{4.1}$$

Another way that is very useful to represent arbitrary 3D shapes, is by storing all unique vertices in the VBO, and then using an EBO to index into that buffer.

$$\boxed{\begin{aligned} VBO &= \{V_0, V_1, V_2, V_3, \dots\} \\ EBO &= \{0, 1, 2, 1, 3, 2, \dots\} \end{aligned}} \tag{4.2}$$

Mesh files

One important thing to consider is how to store meshes on disk, since their representation in memory can vary quite a bit. Luckily there are many file formats to store mesh data. One such format, and the one I decided to implement a parser for, is Wavefront OBJ. The two main reasons are that it is a text based format, making parsing it slightly easier than if it were binary, and the fact that it is an open format, making the specification very accessible.

The way the format works is that first it defines a set of vectors. There are three types of vectors I implemented: position vectors, texture coordinates, and normal vectors. Then once all these vectors are defined, all the surface faces get defined by indexing the vectors and combining them. Since this format has support for face shapes other than triangles, it is important to have mesh triangulation as part of the modeling workflow.

Procedural mesh generation

Sometimes it is desirable to render shapes that aren't modeled beforehand. In this document I will refer to the creation of those shapes as procedural mesh generation. In this project this process takes place in two steps, first a base mesh is generated, and then certain transformations are applied to every vertex in the mesh to reach a new shape. I explain this process in more detail in section 4.7

4.5 Lighting and Texture Mapping

4.5.1 Lighting

In order to achieve lighting effects, the Phong reflection model was implemented. It was developed by Bui Tuong Phong in 1975 [19]. It's an empirical model of local illumination. It describes surface reflections as the combination of diffuse reflection, which falls off gradually, and specular reflection, which causes small highlights. It also introduces a third *ambient* term, to account for light that is scattered about an entire scene.

The way it's implemented in this project is using an OpenGL shader program. The first stage, the vertex shader, runs for every vertex in the mesh and takes care of transforming it from world space to screen space. The second stage, the fragment shader, takes the transformed vertex and shades it. It calculates the three terms of the Phong model separately, and then just adds them together for the final result.

The final result will depend on the amount of lights that influence the mesh being lit. Since running the calculations for every pixel, and for every light can be very inefficient, there are many ways to simplify this. The easiest is to break a scene up into chunks, this way the rendering code can ensure that a mesh is only lit by the lights that are a certain vicinity, reducing the number of calculations that have to be made.

4.5.2 Texture Mapping

Another important feature when it comes to lighting effects is texture mapping. By mapping a diffuse texture as well as a specular texture, the Phong model detailed in the previous section shows its true power. When shading flat surfaces, it can give the impression of depth and reflections without extra geometry. An example is shown in figure 4.3 [1]

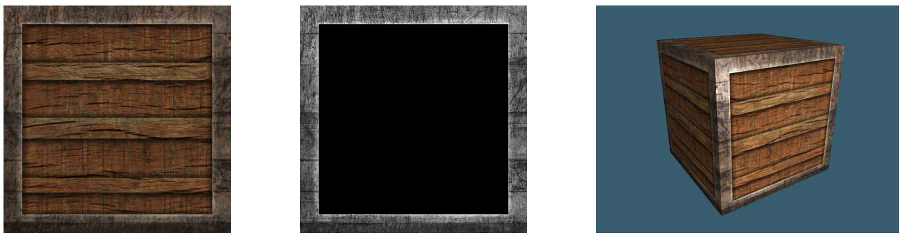

(a) Diffuse texture (b) Specular map (c) Rendered mesh

Figure 4.3: A cube mesh with diffuse and specular textures mapped to it, shaded using the Phong Reflection Model.

Other types of textures can be mapped to the meshes for even more detail. Normal maps, displacement maps and bump maps can create the illusion of higher detail geometry, without actually increasing the number of vertices. Bump maps give the impression of small raised details on a surface, making it look less flat. Normal maps improve upon bump maps, by giving raised details an angle relative to the surface they're on. Displacement maps create higher detail by transforming the underlying mesh. These types of texture maps and their corresponding lighting calculations are outside of the scope of this project.

4.6 Fractal Noise

One of the goals of this project is the ability to generate procedural meshes. A very important tool to that end is 3D smooth noise. When we talk about noise, we are talking about any n dimensional function that will output a random value. In the case of smooth noise, the desired effect is for those values to change gradually when the input parameters also change gradually. One such example can be seen in figure 4.4(a). [2] In order to generate higher detail and a more organic look, lower frequency, scaled up noise (or vice-versa) can be blended on top, like in figure 4.4(b). [3]

[1] Diffuse texture and Specular map obtained from Learn OpenGL © Joey de Vries CC BY 4.0
[2] Perlin noise example published by Lord Belbury under CC0 1.0
[3] 2D sample of Perlin noise released into the public domain by Burgercat

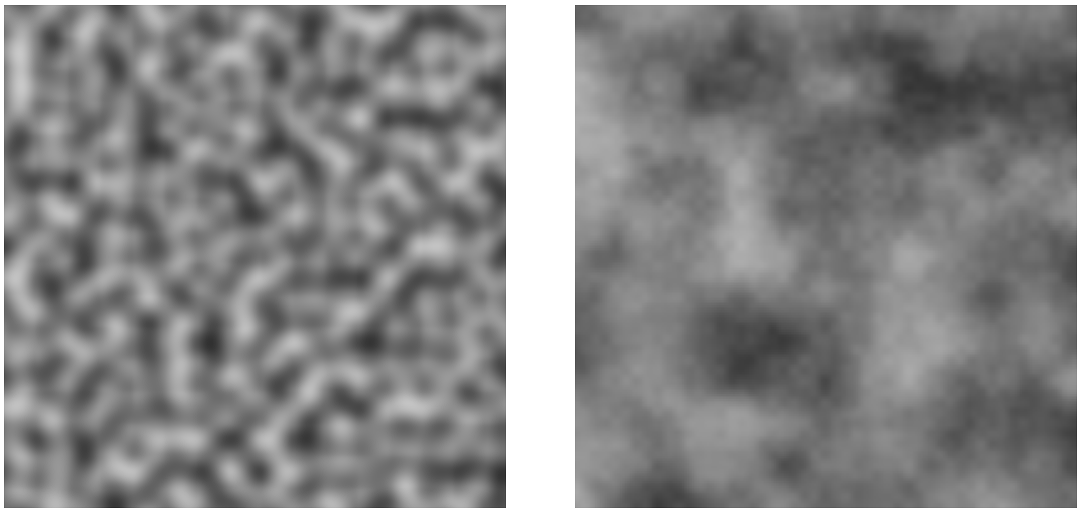

(a) 2D slice through 3D Perlin noise (b) Layered perlin noise

Figure 4.4: 2D slice through 3D Perlin noise.

4.6.1 Perlin noise

The noise function that has been chosen for this project is Perlin noise. It was developed by Ken Perlin in 1983 because he was frustrated with the "machine-like" look of imaged generated by computers at the time. [20] The algorithm to generate Perlin noise has been broken down below.

The algorithm

Grid definition

Define a three-dimensional grid, where each grid point has associated with it a random gradient vector. In order to get a uniform distribution of random vectors, I used the sphere point picking method described by Wolfram MathWorld [21].

Dot product

To determine the noise value of a candidate point, determine the grid cell it lies in. For each of the corners of that cell, calculate the position vector of the point relative to that corner. Take the dot product of this new vector with the gradient of the respective corner.

Interpolation

Finally interpolate between all the dot products calculated in the previous step. Interpolation if done using the smooth-step function described in equation 4.3

$$\boxed{f(min, max, x) = min + (max - min)(3 - 2x)x^2} \tag{4.3}$$

4.7 Procedural Meshes

Procedural mesh generation allows creating dynamic 3D shapes algorithmically. Instead of relying on static models, it facilitates the development of complex objects whose form can change based on predefined or random variables. This in turn enables the creation of organic and natural-looking environments. In this engine, the process is broken up into two steps.

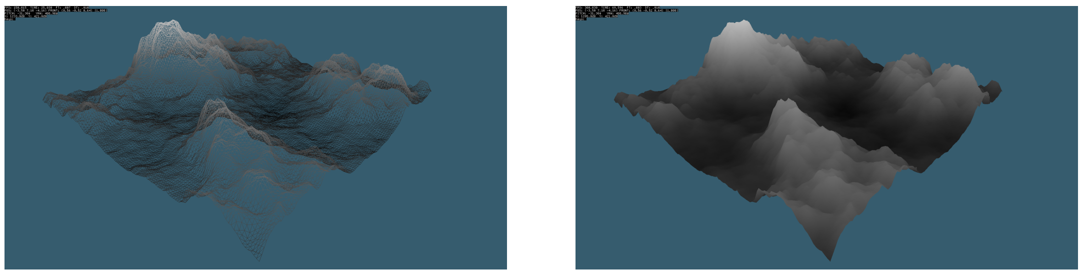
4.7.1 Base mesh

When generating procedural meshes, it is very useful to have a starting point. I found the easiest starting point was a simple plane. Multiple planes can be combined to form cubes, and the vertices of a cube can be normalized to form a sphere.

OpenGL supports a type of indexed mesh called a triangle strip. In this mesh, the VBO as always holds all the unique vertices. The EBO however, instead of indexing individual triangles, will make use of the fact that in a plane most vertices are shared by multiple triangles.

4.7.2 Mesh manipulation

Once a base mesh has been generated, transformations can be applied to it to get procedural or even organic shapes. If for example we sample a 2D plane of fractal noise based on perlin noise, and use the random values to modify the height of each vertex, we can easily create natural looking terrain. See figure 4.5



(a) Rendered showing underlying geometry (b) Rendered using triangles

Figure 4.5: A procedural mesh generated using a plane of triangle strips, and fractal noise.

4.8 Bitmap Font Rendering

One important but easily overlooked aspect of this project was rendering text. Once you are rendering complex scenes, getting real-time debug information in a console is very overwhelming. Since usually you only want to see the current state of certain variables, showing them on screen is the best option.

It is possible to render vector fonts like TTF or OTF formats, but since that requires a library for the rendering, and there aren't many to choose from for Common Lisp, I opted for bitmap fonts instead.

A bitmap font is a font that is entirely defined in one image file. The downside of this is that scaling the font by any fractional factor will result in blurry text, since the pixel size is fixed. The upside is that rendering any character comes down to locating it in a texture, and mapping that texture to some surface.

The font I chose is licensed under Creative Commons 0, and it has all its characters conveniently placed as a function of their Unicode code point. I took some measurements using GIMP, and came up with the following numbers.

- The width of each character is 7 pixels.
- The height of each character is 11 pixels.
- The X position of each character is $98 + 14C_{3,0}$, where C is the last four bits of the character's code point.
- The Y position of each character is $66 + 13C15, 4$, where C is the first 12 bits of the character's code point.

An example of some text rendered with this font can be seen in figure 4.5

5 Testing

Due to most of the code using OpenGL functions, writing tests was challenging at times. Because of this most of the testing was what I will call exploratory testing, where I set up small and contained scenes with the purpose of testing a singular module. However there were some parts where unit tests turned out to be not only possible to write, but also very helpful.

5.1 Unit Testing

To write the unit tests I used the `fiveam` [22] library. It's a very simple framework, which was important to me since I did not want to spend a lot of time setting up a complex testing framework. Test suites in `fiveam` work very similar to Common Lisp packages, first you define a suite using the `def-suite` macro, and when you want to write tests for that suite, you first declare that you are in that suite by using the `in-suite` macro.

One part where unit tests came in very handy was when parsing Wavefront OBJ files. Since a mesh gets broken down into very minimal pieces, (position vectors, texture coordinates, normal vectors and faces), the parsers for those elements turned out very small as well. By writing individual unit tests for each of those elements, and proving that those were correct, I could be quite certain that the final complete parser would be as well.

5.2 Exploratory Testing

The main way most of the code got tested was by writing small programs and prototypes. This way any implemented functionality could not only be verified, but also showcased.

5.2.1 Simple triangle

The most basic 3D scene possible. This example showcases that the `with-graphics` macro works properly, and the OpenGL context has been successfully initialized.

This example initializes the graphics context, and hard-codes the mesh data for a triangle. Furthermore it performs all the steps of getting the data into OpenGL buffers, configuring the VAO, and calling the necessary drawing functions manually.

The code for this test is included in appendix A.1. The final result can be seen in figure 5.1.

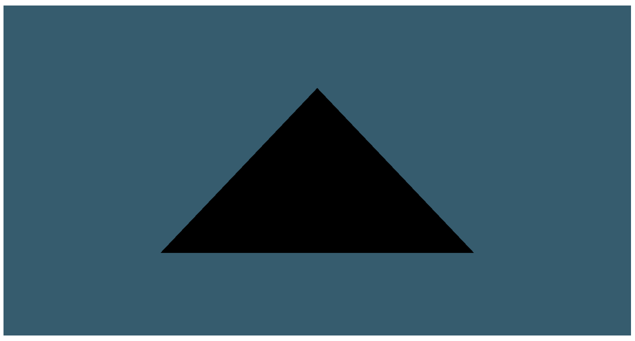
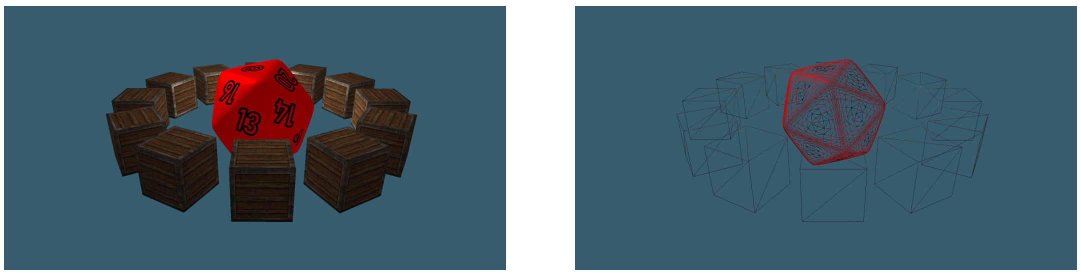


Figure 5.1: A render of a triangle

5.2.2 Loading and rendering textured meshes

This example improves upon the previous one by loading in 3D models from disk, and assigning them to entities that the engine can recognize. It also adds some lights to the scene, though only in the sense that it sends values to the shader program. Finally it renders the loaded models in the scene. This example also shows off the debug mode, which renders meshes using lines instead of triangles.

The code for this test is included in appendix A.2. The final result is shown in figure 5.2.



(a) Normal mode

(b) Debug mode

Figure 5.2: A render of multiple meshes with different textures. There are also some invisible lights in the scene causing diffuse and specular lighting. The multiple cubes are actually only one entity, rendered multiple times with a different transform applied to it.

5.2.3 Natural terrain using fractal Perlin noise

This example first creates a plane mesh build from triangle strips. Then for each of the vertices it samples six different frequencies of Perlin noise, and scales the results inversely proportional to the sampled frequency. For example the values that are sampled at one-fifth the frequency, (by multiplying the sample point position by 0.2), are multiplied by five. The final results for each point get added up, some more transformations are applied, and the resulting value is used to modify the height of vertex as well as to choose its color.

An example of a resulting mesh is shown in figure 4.4. The code for this test is included in appendix A.3.

6 Discussion and Analysis

This chapter aims to discuss and analyze the findings and experiences encountered during the project's lifecycle, addressing its successes, challenges, and its potential applications.

6.1 Results Interpretation

After completing the project's lifecycle, the result is a functional graphics engine. The interface it provides is simple to use, and easy to work with to create graphics visualizations. The project code can be found at `https://github.com/aximili-dev/learnopengl`.

The feature set aligns with the goals that were decided at the start of the project. The engine is capable of loading meshes from files, and creating procedural meshes. The Phong reflection model was implemented and meshes can have diffuse and specular textures mapped to them. Perlin noise was implemented and used to create procedural and natural-looking meshes. Finally, the project stayed inside the specified scope, without growing beyond control.

6.2 Summary of Findings

The main findings of this project are:

- Developing a functional graphics engine using Common Lisp and OpenGL is feasible and advantageous due to the combination of a feature-rich graphics API and a flexible programming language.
- The use of Perlin noise and procedural meshes enables the creation of organic and natural-looking meshes.
- The process of developing graphics software without prior experience or the help of an existing engine proved challenging, highlighting the complexity of graphics programming.

6.3 Challenges and Limitations

While the result of this project could be considered a success, there were numerous challenges encountered during its development. Some related the the design of the engine, others related to the actual implementation. This chapter covers some of those challenges.

6.3.1 Project scope

For a long time at the start of this project, the scope was quite undefined. I had many ideas for different features I wanted to implement, but no real structure or plan. As a consequence, the scope grew unbounded for some time. There were ideas for implementing physics models, a sound system, an Entity Component System (ECS), etc. This meant that the project kept growing, without ever really getting closer to being done.

At some point I decided to completely stop with development, and actually define a concrete goal and the steps to get there. This ended up being one of the most challenging aspects of the project, having to reconcile countless ideas with the fact the project had to actually finish at some point.

6.3.2 Common Lisp ecosystem

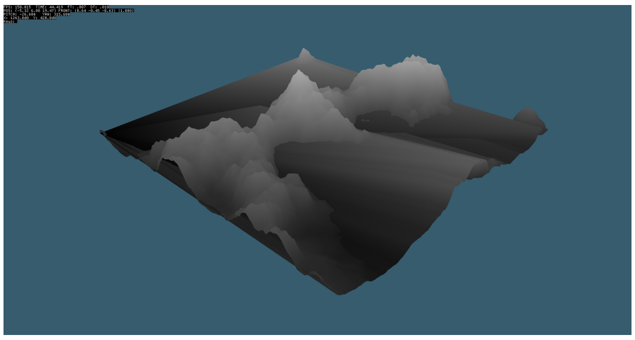
Another challenge that turned out quite tough, was dealing with the current state of the Common Lisp ecosystem. Due to the it having lost popularity over the last few decades, working with the language itself is already quite a challenge. Going by the 2022 Stack Overflow Developer survey, less then 2% of developers use Lisp. Searching the internet for answers to certain questions then, means you will be less likely to find them.

Another problem that arises from the current state of the ecosystem, is the distinct lack of libraries. Compared to a package manager like NPM for JavaScript, which in 2019 reported having over 1.3 million packages published [23], Quicklisp reports a number just over 15000 [15].

When looking for a solution to a common problem, like vector and matrix math, there usually exists only a single, actively maintained library that provides a working solution. In many cases however, there might not even be a library. As an example, the functionality to import Wavefront OBJ files is provided by the `classimp` library, a CFFI library providing bindings to the `Assimp` library. I found working with this library very cumbersome, and since I could not find any other solution in the Quicklisp registry, I ended up writing my own.

6.3.3 Debugging graphics applications

One final challenge that is worth mentioning is debugging graphics applications. Sometimes a small change suddenly causes a program to not start anymore, or crazy graphical glitches. Because of the complexity of the underlying software, and the error system of OpenGL, at times it could be difficult to determine the cause of such errors. An example is shown in figure 6.1.



```
FPS: 150.815  TIME: 44.419  FT: .007  DT: .010
POS: (-5.32 6.80 19.47) FRONT: (0.64 -0.45 -0.63) |1.000|
PITCH: -26.600   YAW: 315.600
X: 1263.0d0  Y: 420.0d0
Keys:
```

Figure 6.1: A graphical glitch caused by a mathematical error when building the initial triangle strip for this mesh. The only way to find the error in this case was reading the code multiple times until the error was caught.

6.4 Implications and Applications

This project shows that graphics programming can be quite accessible to any capable developer. The simplicity of this particular engine means in order to get started with any type of graphics programming, a developer only needs a few lines of code. This aligns with the main goal that was set out, to allow any developer to create prototypes for simple graphics and get immediate results.

However with some extra work, this engine could also be applied to heavier and more complex workloads. If one wanted to develop a video game, it provides almost all the necessary tools to get started with that. Any missing features can be added to the engine at any time or developed as separate components.

Another application that comes to mind is visualizations for physics simulations. Since the engine is built with mesh-manipulation in mind, certain physics simulation types like erosion or soft bodies could easily be implemented with it.

7 Conclusion and Future work

In conclusion, this project has achieved its initial objective of creating a useful and flexible graphics engine. By combining the strengths of OpenGL with the flexibility of Common Lisp, it enables developers to render a wide array of 3D scenes.

7.1 Recommendations for Future Projects

After having finished the development, there are still many areas where improvements are possible. There are also many features that could be implemented and integrated into the engine, so as to have a more complete feature set.

7.1.1 Code refactoring

Due to the nature of the project's lifecycle, most of the code has ended up in one shared package. Before adding any new features, there would be value in splitting the code up into multiple packages, maybe even extracting some libraries for some specific cases, like the Wavefront OBJ parser. This would not only make the current code base more digestible, but it would make it easier to expand on the work in the future.

7.1.2 Scene management

The project currently has a hierarchy where meshes belong to models, and models belong to entities, with each level in this hierarchy being responsible for a different part of the rendering process. Entities are located in the scene and can contain a model, models can have materials and textures and a mesh, and the mesh is just a collection of vertices and indices.

A feature that would allow for more complex scenes would be a scene or world graph, wherein entities can be parents or children to other entities. This way different components or elements can be composed to create complex objects. One consideration for this is that the transforms of entities have to be combined together for any operation that depends on their position, scale or rotation.

7.1.3 UI Toolkit

In the case where a graphics application has to be interactive, a UI toolkit goes a long way towards reaching that goal. Implementing such a toolkit in this engine would require some work, but it would be well worth the effort.

An alternative to implementing such a toolkit from scratch, would be integrating an existing library like Alloy. It's written in Common Lisp and contains code to interface with OpenGL and GLFW [24].

7.1.4 Rigid body physics

Another interesting area for expansion in the current project would be the implementation of a rigid body physics system. Graphics rendering is only one aspect of creating an immersive and interactive 3D environment. Realistic physics simulations can significantly enhance the overall feel and realism of the scene.

Rigid body physics would allow for the simulation of physical interactions between objects in the scene, including factors such as gravity, friction, and force-induced movement. This would bring another layer of realism to the rendered scenes, providing a more engaging and interactive experience for the users.

In addition to the physics system, a collision detection and response system would be an essential addition. Collision systems allow for the detection of interactions between objects in a scene, such as when they touch, overlap or collide. Once a collision is detected, appropriate responses such as bouncing or sliding can be simulated, adding another level of authenticity to the scene.

Bibliography

[1] J. Gregory, *Game Engine Architecture*. A K Peters/CRC Press, 3 ed., 2018.

[2] T. K. G. Inc, *About The Khronos Group*. Accessed: 2022-02-18.

[3] P. Seibel, *Practical Common Lisp*. Apress, 4 ed., 2005.

[4] J. de Vries, *Learn OpenGL - Graphics Programming*. Kendall & Welling, 2020.

[5] B. Karadžić, "Bgfx." https://bkaradzic.github.io/bgfx/overview.html. Accessed: 2023-06-18.

[6] 3b, "cl-opengl." https://cl-opengl.common-lisp.dev/. Accessed: 2023-06-18.

[7] K. Pitman, "Common lisp hyperspec." http://www.lispworks.com/documentation/HyperSpec/Front/index.htm Accessed: 2023-06-11.

[8] "The computer graphics essential reference." https://www.cs.cmu.edu/ ph/nyit/masson/companies.htm. Accessed: 2023-06-18.

[9] J. Bielman and L. Oliveira, "Cffi - the common foreign function interface." https://cffi.common-lisp.dev/. Accessed: 2023-05-29.

[10] kaveh808, "kons-9." https://github.com/kaveh808/kons-9. Accessed: 2023-06-15.

[11] Shirakumo, "Trial." https://shirakumo.github.io/trial/. Accessed: 2023-06-16.

[12] B. Beer, "Clinch." https://github.com/BradWBeer/clinch. Accessed: 2023-05-29.

[13] O. Arndt, "Horde3d." https://github.com/anwyn/cl-horde3d/. Accessed: 2023-05-29.

[14] Okra, "Okra." https://okra.common-lisp.dev/manual.html. Accessed: 2023-06-16.

[15] Z. Beane, "Quicklisp." https://www.quicklisp.org/beta/. Accessed: 2023-06-11.

[16] ASDF, "Another system definition facility." https://asdf.common-lisp.dev/. Accessed: 2023-06-11.

[17] J. Tavora, "Sly." https://github.com/joaotavora/sly. Accessed: 2023-06-16.

[18] http://joaotavora.github.io/sly/, "Sly manual." Joao Tavora. Accessed: 2023-06-18.

[19] B. T. Phong, "Illumination for computer generated pictures," *Communications of ACM 18*, no. 6, pp. 311–317, 1975.

[20] K. Perlin, "Making noise." https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/. Accessed: 2023-06-11.

[21] E. W. Weisstein, "Sphere point picking." From MathWorld – A Wolfram Web Resource.

[22] FiveAM, "Fiveam." https://fiveam.common-lisp.dev/. Accessed: 2023-06-15.

[23] A. Nassri, "So long, and thanks for all the packages!." https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages. From: npm Blog – Accessed: 2023-06-15.

[24] Shirakumo, "Alloy." https://shirakumo.github.io/alloy/. Accessed: 2023-06-16.

TERMINOLOGY

graphics engine	A set of tools and software abstractions for graphics programming..
struct	A common lisp data structure to aggregate data types with named elements. Very similar to structs in C..

Acronyms

API	Application Programming Interface.
ASDF	Another System Definition Facility.
CFFI	Common Foreign Function Interface.
CLI	Command Line Interface.
EBO	Element Buffer Object.
ECS	Entity Component System.
GLFW	Graphics Library Framework.
GLUT	OpenGL Utility Toolkit.
GPU	Graphics Processing Unit.
IDE	Integrated Development Environment.
OS	Operating System.
REPL	Read-Evaluate-Print Loop.
S-expression	Symbolic Expression.
VAO	Vertex Array Object.
VBO	Vertex Buffer Object.

Appendices

A Exploratory test code

This appendix presents the code used in the exploratory tests specified in section 5.2.

A.1 Triangle

Code A.1: Code used in exploratory test to render a triangle

```
(defun main ()
  (with-graphics (graphics "01␣Triangle"
          :hide-cursor nil
          :window-width 800
          :window-height 600)
    (let ((vertex-positions #(-0.5 -0.5 0.0 0.5 -0.5 0.0 0.0 0.5 0.0))
      (vao (gl:gen-vertex-array))
      (vbo (gl:gen-buffer))
      (shader (load-shader-from-disk #P"./triangle.vert" #P"./triangle.frag")))

      (setup vao vbo)

      (loop while (not (glfw:window-should-close-p (graphics-window graphics)))
        do (render vao shader graphics vertex-positions))

      (gl:delete-buffers (list vbo))
      (gl:delete-vertex-arrays (list vao))
```

Code A.2: Setup code for A.1

```
(defun setup (vao vbo)
  (gl:bind-vertex-array vao)

  (gl:bind-buffer :array-buffer vbo)

  (with-gl-array (gl-array vertex-positions :float)
        (gl:buffer-data :array-buffer :static-draw gl-array))

  (gl:vertex-attrib-pointer 0 3 :float :false (* 3 4) 0)
  (gl:enable-vertex-attrib-array 0)
```

Code A.3: Rendering code for A.1

```
(defun render (vao shader graphics vertex-positions)
  (gl:clear-color 0.21 0.36 0.43 1.0)
  (gl:clear :color-buffer :depth-buffer)

  (gl:bind-vertex-array vao)

  (shader-use shader)
  (gl:draw-arrays :triangles 0 (length vertex-positions))

  (gl:bind-vertex-array 0)

  (glfw:swap-buffers (graphics-window graphics))
```

Code A.4: Vertex shader used in code A.1

```
#version 330 core

layout (location = 0) in vec3 aPos;

out vec3 fragPos;

void main()
{
  gl_Position = vec4(aPos, 1.0);
  fragPos = aPos;
}
```

Code A.5: Fragment shader used in code A.1

```
#version 330 core

out vec4 FragColor;

void main()
{
  FragColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

A.2 Meshes

Code A.6: Code used in exploratory test to load entity models from disk, and render them using the Phong reflection model

```
(defun main ()
  (with-graphics (graphics "02␣Meshes"
          :hide-cursor t
          :window-width 800
          :window-height 600)
    (let* ((camera (make-instance 'fps-camera :sensitivity 0.1 :speed 10.0 :pos (vec 0 0 10)))
       (model-shader (load-shader-from-disk #P"./shaders/model.vert" #P"./shaders/model.frag"))
       (cube-model (load-model #P"./models/cube.obj" :diffuse-path #P"./container.png" :specular-path
            #P"./container.specular.png"))
       (cube-entity (make-instance 'entity :model cube-model :shader model-shader :transform (transform)))
       (d20-model (load-model #P"./models/Dice_d20.obj" :diffuse-path #P"./d20white.png"
            :specular-path #P"./d20white.png"))
       (d20-entity (make-instance 'entity :model d20-model :shader model-shader :transform (transform
            (vec 0 0 0) (vec 3 3 3)))))
      (graphics-setup-fps-camera graphics camera)
      (flet ((tick (time dt)
           (process-tick camera time dt (graphics-keys graphics)))
         (render (current-frame fps &key debugp)
           (gl:clear-color 0.21 0.36 0.43 1.0)
           (gl:clear :color-buffer :depth-buffer)

           (setup-lights model-shader)

           (if debugp
           (gl:polygon-mode :front-and-back :line)
           (gl:polygon-mode :front-and-back :fill))

           (push (format nil "FPS:␣~3,3f" fps) (graphics-debug-text graphics))))
    (graphics-run graphics #'render #'tick)))))
```

Code A.7: Lighting setup code for code A.6

```
(defun setup-lights (shader)
  (shader-use shader)

  (let ((i 0))
    (macrolet ((set-light-prop (i prop &rest values)
                   `(let ((loc (format nil "pointLights[~d].~a" ,i ,prop)))
                      (shader-set-uniform shader loc ,@values))))
            (dolist (light-pos *light-positions*)
          (with-vec (x y z) light-pos
                (set-light-prop i "position" x y z)
                (set-light-prop i "constant" 1.0)
                (set-light-prop i "linear" 0.09)
                (set-light-prop i "quadratic" 0.032)
                (set-light-prop i "ambient" 0.2 0.2 0.2)
                (set-light-prop i "diffuse" 0.5 0.5 0.5)
                (set-light-prop i "specular" 1.0 1.0 1.0))
          (incf i)))))

(defun render-entities (d20-entity cube-entity)
  (render-entity d20-entity
         (graphics-v-width graphics)
         (graphics-v-height graphics)
```

Code A.8: Rendering code for code A.6

```
         :debugp debugp)

  (dotimes (i 12)
    (let* ((angle (* i (/ 360 12)))
       (angle (* angle (/ pi 180)))
       (pos (vrot (vec 0 0 6) +vy+ angle)))
      (with-slots (transform) cube-entity
    (setf transform (transform pos
                     (vec 1 1 1)
                     (qfrom-angle +vy+ angle)))

    (render-entity cube-entity
               (graphics-v-width graphics)
               (graphics-v-height graphics)
               camera
               :debugp debugp))))
```

Code A.9: Vertex shader used in code A.6

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec3 fragPos;
out vec3 normal;
out vec2 texCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main ()
{
  gl_Position = projection *view *model *vec4(aPos, 1.0);
  gl_PointSize = 7;
  fragPos = vec3(model *vec4(aPos, 1.0));
  normal = mat3(transpose(inverse(model))) *aNormal;
  texCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Code A.10: Fragment shader used in code A.6

```
#version 330 core

struct Material {
   float shininess;
   sampler2D diffuse; sampler2D specular;
};

struct PointLight {
   vec3 position;
   vec3 ambient; vec3 diffuse; vec3 specular;
   float constant; float linear; float quadratic;
};

struct DirLight {
   vec3 direction;
   vec3 ambient; vec3 diffuse; vec3 specular;
};

out vec4 FragColor;

in vec3 fragPos; in vec3 normal; in vec2 texCoord;

uniform vec3 viewPos;

#define NR_POINT_LIGHTS 4

uniform DirLight dirLight;
uniform PointLight pointLights[NR_POINT_LIGHTS];
uniform Material material;

vec3 calcDirLight(DirLight light, vec3 normal, vec3 viewDir);
vec3 calcPointLight(PointLight light, vec3 normal, vec3 viewDir, vec3 fragPos);

void main()
{
      // Diffuse
      vec3 norm = normalize(normal);
      vec3 viewDir = normalize(viewPos -fragPos);

      vec3 result = calcDirLight(dirLight, norm, viewDir);

      for (int i = 0; i < NR_POINT_LIGHTS; i++)
         result += calcPointLight(pointLights[i], norm, viewDir, fragPos);

      FragColor = vec4(result, 1.0);
}
```

Code A.11: Supporting function for code A.10

```
vec3 calcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
  vec3 lightDir = normalize(-light.direction);
  vec3 reflectDir = reflect(-lightDir, normal);

  float diff = max(dot(normal, lightDir), 0.0);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

  vec3 ambient = light.ambient *vec3(texture(material.diffuse, texCoord));
  vec3 diffuse = light.diffuse *diff *vec3(texture(material.diffuse, texCoord));
  vec3 specular = light.specular *spec *vec3(texture(material.specular, texCoord));

  return (ambient + diffuse + specular);
}
```

Code A.12: Supporting function for code A.10

```
vec3 calcPointLight(PointLight light, vec3 normal, vec3 viewDir, vec3 fragPos)
{
  vec3 lightDir = normalize(light.position -fragPos);
  vec3 reflectDir = reflect(-lightDir, normal);

  float diff = max(dot(normal, lightDir), 0.0);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

  float distance = length(light.position -fragPos);
  float attenuation = 1.0 / (light.constant +
                  light.linear *distance +
                  light.quadratic *(distance *distance));

  vec3 ambient = light.ambient *vec3(texture(material.diffuse, texCoord));
  vec3 diffuse = light.diffuse *diff *vec3(texture(material.diffuse, texCoord));
  vec3 specular = light.specular *spec *vec3(texture(material.specular, texCoord));

  ambient *= attenuation;
  diffuse *= attenuation;
  specular *= attenuation;

  return (ambient + diffuse + specular);
}
```

A.3 Procedural Terrain

Code A.13: Code used in exploratory test to generate procedural terrain. Also defines some global variables to avoid functions with many arguments.

```
(defparameter *vao* nil)
(defparameter *vbo* nil)
(defparameter *ebo* nil)
(defparameter *num-indices* nil)

(defun main ()
  (with-graphics (graphics "03␣Terrain"
                 :hide-cursor t
                 :window-width 800
                 :window-height 600)
    (let* ((camera (make-instance 'fps-camera :sensitivity 0.1 :speed 10.0 :pos (vec 0 0 10)))
           (points (make-array 0 :adjustable t :fill-pointer 0))
           (indices (make-array 0 :adjustable t :fill-pointer 0))
           (shader (load-shader-from-disk #P"./colored.vert" #P"./colored.frag"))
           (width 150))

      (create-mesh points indices)
      (setup-gl-mesh points indices)

      (graphics-setup-fps-camera graphics camera)

      (flet ((tick (time dt)
               (process-tick camera time dt (graphics-keys graphics)))
             (render ()
               (render-mesh shader camera)))
        (graphics-run graphics #'render #'tick)))))
```

Code A.14: Creates the vertices for code A.13. Sets the vertex height using samples from fractal Perlin noise. Sets the vertex color as a function of the vertex height.

```
(defun create-mesh-vertices (points width)
  (dotimes (x width)
    (dotimes (z width)
      (let* ((seeds '(1 2 3 4 5 6))
          (wpos (vec (/ x 10) (/ y 10) (/ z 10)))
          (value (+ (* (perlin (v* wpos 0.2) :seed (nth 0 seeds)) 5)
                (* (perlin (v* wpos 0.4) :seed (nth 1 seeds)) 2)
                (perlin wpos :seed (nth 2 seeds))
                (/ (perlin (v* wpos 2) :seed (nth 3 seeds)) 2)
                (/ (perlin (v* wpos 4) :seed (nth 4 seeds)) 4)
                (/ (perlin (v* wpos 8) :seed (nth 5 seeds)) 8)))
          (value (if (< value 0)
              (/ value 4)
                value))
          (value (if (< value -1)
              -1.0
                value))
          (color (height-to-color value min max)))
    (when t
      (vector-push-extend (vx wpos) points)
      (vector-push-extend value points)
      (vector-push-extend (vz wpos) points)
      (vector-push-extend (vx color) points)
      (vector-push-extend (vy color) points)
      (vector-push-extend (vz color) points))))))

(defun height-to-color (y min max)
  (let* ((diff (- max min))
      (dist (- y min))
      (y (float (/ dist diff))))
    (vec y y y)))
```

Code A.15: Calculates the mesh indices for A.13. The indices are calculated to be rendered as a triangle strip.

```
(defun create-mesh-indices (indices width)
  (dotimes (z (1- width))
    (dotimes (x width)
      (vector-push-extend (+ (* z w) x) indices)
      (vector-push-extend (+ (* z w) x w) indices))

    (vector-push-extend #xFFFFFFFF indices))

  (setf *num-indices* (length indices)))
```

Code A.16: Sets up the mesh as OpenGL buffers for code A.13

```
(defun setup-gl-mesh (points indices)
  (setf *vao* (gl:gen-vertex-array))
  (setf *vbo* (gl:gen-buffer))
  (setf *ebo* (gl:gen-buffer))

  (gl:bind-vertex-array *vao*)

  (let ((gl-arr (make-gl-array points :float)))
    (gl:bind-buffer :array-buffer *vbo*)
    (gl:buffer-data :array-buffer :static-draw gl-arr)
    (gl:free-gl-array gl-arr))

  (let ((gl-arr (make-gl-array indices :unsigned-int)))
    (gl:bind-buffer :element-array-buffer *ebo*)
    (gl:buffer-data :element-array-buffer :static-draw gl-arr)
    (gl:free-gl-array gl-arr))

  (gl:enable-vertex-attrib-array 0)
  (gl:vertex-attrib-pointer 0 3 :float :false (* 6 4) 0)

  (gl:enable-vertex-attrib-array 1)
  (gl:vertex-attrib-pointer 1 3 :float :false (* 6 4) (* 3 4))

  (gl:bind-vertex-array 0))

(defun render-mesh (shader camera debugp)
  (shader-use shader)

  (let ((view (view-matrix camera))
    (proj (mperspective 45 (/ v-width v-height) 0.1 1000)))
    (shader-set-uniform shader "view" (marr view))
    (shader-set-uniform shader "projection" (marr proj))
```

Code A.17: Rendering code for code A.13

```
    (shader-set-uniform shader "projection" (marr proj))
    (shader-set-uniform shader "model" (marr (meye 4))))

  (when debugp
    (gl:polygon-mode :front-and-back :line))

  (cl-opengl-bindings:primitive-restart-index #xFFFFFFFF)

  (gl:bind-vertex-array *vao*)
  (gl:draw-elements :triangle-strip
            (gl:make-null-gl-array :unsigned-int)
            :count 44849)
  (gl:polygon-mode :front-and-back :fill)
  (gl:bind-vertex-array 0))
```

Code A.18: Vertex shader used in code A.13

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fragPos;
out vec4 vertColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main ()
{
  gl_Position = projection *view *model *vec4(aPos, 1.0);
  gl_PointSize = 5;
  fragPos = vec3(model *vec4(aPos, 1.0));
  vertColor = vec4(aColor, 1.0);
}
```

Code A.19: Fragment shader used in code A.13

```
#version 330 core

out vec4 FragColor;

in vec3 fragPos;
in vec4 vertColor;

uniform vec3 viewPos;

void main()
{
    FragColor = vertColor;
}
```

B
Emacs configuration

```
;; Package repositories
(require 'package)
(add-to-list 'package-archives '("melpa" . "https://melpa.org/packages/") t)
(package-initialize)

;; Straight.el
(defvar bootstrap-version)
(let ((bootstrap-file
       (expand-file-name "straight/repos/straight.el/bootstrap.el" user-emacs-directory))
      (bootstrap-version 6))
  (unless (file-exists-p bootstrap-file)
    (with-current-buffer
        (url-retrieve-synchronously
         "https://raw.githubusercontent.com/radian-software/straight.el/develop/install.el"
         'silent 'inhibit-cookies)
      (goto-char (point-max))
      (eval-print-last-sexp)))
  (load bootstrap-file nil 'nomessage))

;; use-package to simplify the config file
(straight-use-package 'use-package)

(use-package sly
  :ensure t
  :init
  (setq sly-lisp-implementations
    '((sbcl ("/usr/bin/sbcl") :coding-system utf-8-unix))))

(use-package magit
  :ensure t)

(use-package treemacs
  :ensure t
  :init
  (setq treemacs-is-never-other-window t)
  :config
  (treemacs-follow-mode -1))

(use-package glsl-mode
  :ensure t)
```

Index