Analysis of Type-Driven approach to systems programming:
Implementation of OpenGL library for Rust

(Analiza programowania systemowego z wykorzystaniem systemu typów:
Implementacja biblioteki do OpenGL dla języka Rust)

Mikołaj Depta

Praca inżynierska

Promotor: dr Andrzej Łukaszewski

Abstract

For the past few years in the software development industry there has been a growing interest in strongly typed languages. It manifests itself in emergence of brand-new technologies in which strong type systems were one of the core founding principles or in changes introduced to existing languages. The most common examples of modern languages with powerful type systems are TypeScript as an alternative to JavaScript in the world of web development or Rust in domain of systems programming in place of C and C++. More mature languages also had their type systems revised for example in C# 8 explicit type nullability annotations were introduced, or even dynamically typed Python has seen major improvements to its type annotation system.

This study - an implementation of the OpenGL graphics API wrapper library for Rust - will attempt to demonstrate how Rust's type system can be utilized to improve low-level software safety and maintainability as well as how it affects API design and codebase structure.

For the past few years in the software development industry there has been a growing interest in strongly typed languages. It manifests itself in emergence of brand-new technologies in which strong type systems were one of the core founding principles or in changes introduced to existing languages. The most common examples of modern languages with powerful type systems are TypeScript as an alternative to JavaScript in the world of web development or Rust in domain of systems programming in place of C and C++. More mature languages also had their type systems revised for example in C# 8 explicit type nullability annotations were introduced, or even dynamically typed Python has seen major improvements to its type annotation system.

This study - an implementation of the OpenGL graphics API wrapper library for Rust - will attempt to demonstrate how Rust's type system can be utilized to improve low-level software safety and maintainability as well as how it affects API design and codebase structure.

Contents

Chapter 1

The Rust programming language

Introduction

Rust is an open-source, general purpose, multi paradigm, compiled, statically and strongly typed language maintained by Rust Foundation.

Although general purpose, language lends itself particularly well to systems programming, where software reliability, scalability and efficiency are paramount. These qualities can be attributed to powerful and expressive type system and ownership based memory management which guarantees memory safety without a garbage collector.

In the following sections we describe

1.1 Language history

Language had it's initial inception in 2006 by Graydon Hoare, a software engineer at Mozzilla. Language's name was inspired by fungus called Rusts [1].

Fungus' lifecycle and reproduction cycle which is distributed, parallel and not prone to single point of failure seemed like qualifier – CITATION NEEDED

Over time more and more people joined Hoare on work on the project and in 2009, Mozilla decided to sponsor the project by founding the development. Rust 1.0 released on the 15th of May 2015.

1.2 Language Features

1.3 Data types

1.3.1 User defined types

Rust provides

As of version 1.79 Rust provides 3 Rust provides 3 ways to construct used defined aggregate types. These are: - `structs` - `enums` - `unions`

We focus on the first two of these.

structs   Structure is a named grouping of other data types (product type). They constitute the basic building blocks for any user defined types.

1.4   Implementations

Functionality of a type is not provided inline with it's definition like in most C like languages. Instead it's associated with a type by so called *implementations*. Implementations for a type are contained within a block introduced by the `impl` keyword.

There are two types of implementations: - inherent implementations - trait implementations

All items within impl block are called associated items

Functions defined within impl blocks are called associated functions and can be accessed with qualification `<type-name>::<function-name>`.

Within an implementation both `self` and `Self` keywords can be used. `self` can be used in associated function definitions within an impl block as first parameter. Such functions are called methods and the `self` parameter denotes the receiver of method call. `self` can be additionally qualified with immutable or mutable reference `&` or `&mut`.

`Self` is a type alias that refers to implementing type.

1.4.1   Inherent implementations

We will shorthand implementation to impl which is common in Rust terminology.

Inherent impls associate contents of an `impl` block with specified nominal type. Such blocks can contain definition of a constants or functions.

1.5   Generic types

As of version 1.79 Rust provides 3 kinds of generic parameters types can use. These are: - type parameters - constant parameters - lifetime parameters Type which uses any generic parameters is said to be generic.

1.5.1   Type parameters

Type parameters represent an abstract type

1.5.2   Constant parameters

1.5.3   Lifetime parameters

1.6   Type level abstraction

Traits provide an ability to express shared behavior in abstract way.

1.7   Implementation Coherence

1.7.1   Orphan Rule

Chapter 2

Basics of Computer Graphics

Introduction

The task of 3D computer graphics is to generate a 2D array of discrete color values from geometric representation of a scene. This process can be divided into two halves:

1. Geometry processing – manipulation of geometric data that

2. Pixel processing –

Between these two halves the rasterization process happens which transforms geometric data into set of discrete values

2.0.1 Programmable graphics pipeline

Release of Nvidia's GeForce 256 was a significant milestone for 3D graphics hardware. It was the first graphics accelerator that was capable of on-chip Transform and Light calculations which allowed it to process all of the pipeline without interacting with the CPU.

As power of, now called Graphics Processing Units (GPUs), grew it

Chapter 3

OpenGL

3.1 Introduction

OpenGL is an specification of an API for hardware accelerated computer graphics owned and maintained by the Khronos Group Inc.

Since it's inception and up until 2016 with release of Vulkan it has been the only widely supported cross platform graphics API.

TODO: opis opengl'a

3.2 OpenGL objects

OpenGL exposes an abstraction over GPU's resources called objects. In order to use given object it first must be bound to a binding point in current OpenGL context.

Objects contain internal state which can be queried using introspection. Objects are identified by a *name* which is an unsigned 32 bit integer. There exists common object model which describes how most types of objects are managed.

Most types of object can be created with either call to

$$\text{void } \textbf{Gen*s}(\texttt{sizei } n, \texttt{ uint } *objects)$$

which will allocate object's *name* and subsequent call to

$$\text{void } \textbf{Bind*}(\texttt{uint } name, ...)$$

which will bind given object to the context and, if given object has never been bound before, allocate their internal state. Alternatively one can use

$$\text{void } \textbf{Create*s}(\texttt{sizei } n, \texttt{ uint } *objects)$$

which will allocate both object's *name* and its state but it will not set any context bindings. There exist separate namespace for each object type.

The most notable outliers that do not conform to the rules above are program objects, shader objects

Objects can be deleted with `void` **Delete*s**(`sizei` *n*, `uint` *objects), bound with aforementioned `void` **Bind***(`uint` *name*, ...) which usually accepts additional parameter that specifies binding point.

OpenGL specification defines set of publicly available object parameters which can be queries using introspection with **GetInteger*** family of functions. One notable usage is determining compilation and linking status for shaders and programs.

3.2.1   Buffer object

Buffer objects provide means to allocate data stores in GPU memory. They can contain data of different format and purpose depending on buffer's target. Primary usage for buffers is to provide geometric information which includes vertex attribute values and indices for indexed rendering.

3.2.2   Vertex Array object

Modern OpenGL is generic over vertex format and only poses limitation on the number of such attributes and limits their values to glsl's scalar or vector types. Each attribute is assigned a zero-based index. Vertex Array object (VAO) assigns each active attribute information on how and where from to source vertex data, as well as, what is the data type of provided attribute in glsl.

This can be seen as two separate pieces of information: - in memory layout and access frequency - interpretation / conversion of the data.

Originally all of that information was specified at one with

$$\text{void VertexAttrib*Pointer( )}$$

3.3   History

IRIS GL, a proprietary graphics API, which later became OpenGL was initially developed by Silicon Graphics (SGI) during 1980's. SGI open sourced subset of their API as OpenGL due to mounting market pressure from SGI competitors (Sun Microsystems, Hewlett-Packard, IBM) who started providing their own APIs for hardware accelerated 3D graphics based on an existing open standard called PHIGS.

In 1992 OpenGL Architectural Review Board (OpenGL ARB) was established

and it was tasked with development and maintenance of the OpenGL specification. This task was passed on to Khronos Group in 2006 where it remained ever since.

3.4 Modern OpenGL

OpenGL release 3.0 was a turning point in the specification's development. It deprecated

3.5 Graphics pipeline

OpenGL 4.6 models graphics pipeline as follows:

3.5.1 Vertex Specification

Before any draw calls can be made geometric data must be transferred to the client and generic vertex attributes must be configured.

3.5.2 Vertex shading

3.5.3 Tesselation

3.5.4 Geometry shading

3.5.5 Primitive assembly

3.5.6 Rasterization

3.5.7 Fragment shading

3.5.8 Fragment post processing

3.5.9 Compute shaders

3.6 GLSL

Bibliography

[1] Clive Thompson. How rust went from a side project to the world's most-loved programming language. *s Interneta, https://www. technologyreview. com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language*, 14, 2023.