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

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

26 sierpnia 2024

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 largely attributed to powerful and expressive type system and ownership based memory management system which guarantees memory safety without a garbage collector.

1.1 Data types

1.1.1 User defined types

Rust provides 3 ways to construct used defined aggregate types which. These are: - `structs` - `enums` - `unions`

We focus on the first two of these.

`structs` Struct is a heterogeneous product of other types, they are analogous to struct types in C, the record types of the ML family, or the struct types of the Lisp family. They constitute the basic building blocks for any user defined types.

A struct type is a heterogeneous product of other types, called the fields of the type.

The memory layout of a struct is undefined by default to allow for compiler optimizations like field reordering, but it can be fixed with the repr attribute. In either case, fields may be given in any order in a corresponding struct expression; the resulting struct value will always have the same memory layout.

The fields of a struct may be qualified by visibility modifiers, to allow access to data in a struct outside a module.

A tuple struct type is just like a struct type, except that the fields are anonymous.

A unit-like struct type is like a struct type, except that it has no fields. The one value constructed by the associated struct expression is the only value that inhabits such a type.

1.2 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.

1.2.1 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.

1.3 Generic types

As of version 1.80 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.

1.3.1 Type parameters

Type parameters can be used in function or type definition; they represent an abstract type which must be specified (or inferred) during compilation. Type generics

are most commonly used for collections since they can contain arbitrary object and don't need to know almost anything about the inner types.

However, one can't do much with trully arbitrary type, even collections require ordering for tree structures, hashing for hash based collections and even the simplest collections like vectors and queues need to know that types they contain have finite size, or can be shared across thread boundaries. Most languages either provide these kinds of behaviour inductively by the structure of a type but that's not what Rust does. Rust requires that pretty much all capabilities of a type are specified.

Capabilities of a type parameter are expressed using traits which we in the next section.

1.3.2 Constant parameters

Similarity to how types can be generic over type, rust allows types to be generic by a constant value. These, so called, dependent types provide brand new level of expressive power, statically sized arrays especially become much more useful. This makes stack based allocations much more common, improving performance and reducing heap fragmentation, but for our purposes it allows type system verify and enforce certain quantities or reason about them in an abstract way which, as we will show in this study, can be used to express very complex systems.

1.3.3 Lifetime parameters

Lifetime parameters are standout feature of Rust. They represent duration based on lexical scoping for how long reference remains valid, so being generic over lifetime means being generic to how long given reference can be held.

1.4 Traits

Traits provide an ability to express shared behavior in abstract way [1]. We are mostly interested in their use in *trait bounds* on types and type parameters. Trait bounds declare contracts that types must fulfil or else the program will be rejected. We used it to enforce use of valid data formats and proper sequencing of operations.

As mentioned in the previous section, type parameters don't have any capabilities unless explicitly declared. Trait bounds serve that exact purpose. Types and generic parameters have their requirements states in such where clause and these requirements are checked at call site.

What distinguishes Rust's traits from most other languages is its unique scheme of implementing functionality for types. Trait for a type is implemented in a very similar fashion to inherent impls using `impl Trait for Type { ... }` syntax.

Such impl block must contain definition for all items a trait provides. Traits can uniquely, be implemented generically for all types that salify bounds using an **blanket impl** `impl<T> Trait for T where T: ...` { ... }. This will even influence types from crate external to trait's definition. Blanket impls however come with significant downside - a blanket impl is the only impl for that trait that may exist. This requirement is overly conservative and stems from necessity to guarantee impl coherence which we discuss in the next section

1.5 Implementation Coherence

Rust must be always able to uniquely determine which method corresponds to which impl block that is, impl blocks must be coherent with each other, they must not interfere or overlap. That's the reason why as of 2024 Rust enforces one blanket impl - it cannot guarantee that two blanket impls of the same trait don't target some type twice. However, if inherent or trait impl's target a specific generic type with at least one type parameter differing between the two impls coherence is preserved and program passes orphan rule.

1.5.1 Orphan Rule

Chapter 2

OpenGL and GLSL

2.1 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.

2.2 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.

2.3 OpenGL objects

OpenGL exposes an abstraction over GPU's resources called objects. These roughly correlate with object oriented design as they aggregate data for appropriate subset of operations albeit with certain unique caveats. In all but the latest opengl versions, to use given object it first must have been bound to a binding point in global in current OpenGL context. In OpenGl 4.6 the `ARB_direct_state_access` extension was made part of core specification which introduces duplicates of all object manipulating functions to accept as one of parameters the name of object to operate on.

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 objects can be created with a call to

`void` **Gen*s**(`sizei` *n*, `uint` **objects*)

which will allocate the object's *name*. A subsequent call to

`void` **Bind***(`uint` *name*, ...)

will bind the given object to the context. If the object has never been bound before, this will also allocate its internal state. Alternatively, one can use

`void` **Create*s**(`sizei` *n*, `uint` **objects*)

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

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

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.

2.3.1 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.

2.3.2 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 viewed as two aspects: (1) memory layout and access frequency, and (2) data interpretation/conversion.

Originally all of that information was specified at one with

`void` **VertexAttrib*Pointer**(

)

2.4 Graphics pipeline

The modern OpenGL pipeline is a sequence of both programmable and fixed function stages that process geometric data to form discrete color values - pixels - that end up stored in a framebuffer.

2.4.1 Vertex Specification

Before rendering can begin, geometric information needs to uploaded to GPU memory along with its description as generic vertex attributes.

Generic Vertex is an abstract composition of values (attributes) that is supposed to represent a vertex of the triangular mesh of an object. Generic stands from the fact that data associated with vertices has no intrinsic meaning. Semantics of data are decided by client provided vertex shader.

OpenGL sources data for each vertex attribute form a buffer. Each attribute is assigned a unique numeric index. Association between attribute with given index and a buffer, from which that attribute should be sourced, is established by Vertex Array Object (abbr. VAO).

Once all vertex attributes have their data sources assigned and properly configured, vertex specification can be considered finished and one could precede with further pipeline configuration. In this instance vertices would be interpreted sequentially as appropriate geometric primitives. This forces vertex data to be specified redundantly for lines and especially raw triangles, since each triangle shares an edge with each neighboring triangle.

To better conserve memory one can use indexed rendering. This requires additional buffer filled with indices into main vertex buffer instead of inlined vertex data. In case of basic triangle rendering (without using compressed representations like triangle fan or triangle / line strip) will still cause repetition but now only few byte wide indices instead of whole attributes which are substantially larger.

2.4.2 Vertex Shader

Vertex shader is the first programmable stage of OpenGL Pipeline and is one of two required shaders to execute a draw call, the other being the fragment shader.

Most commonly vertex shader performs 3 translations. From initial model space, world space, view space to final clip space which we will now discuss briefly.

model space - when a 3D model is created in 3D modelling software its vertex positions are specified to some local coordinate system (commonly center of an object). These positions would commonly be loaded into gpu memory. Such objects can be easily placed in broader scene by providing a so called world transform performs transformation from model's local coordinate system to scene's coordinate system.

world space - world space refers to coordinate system of a scene that uses multiple models by translating, scaling or rotating them.

view space - its common for 3D rendering applications to provide means of interacting with the scene. Wether its a 3D computer game, CAD program or medical data visualization we would like to be able to control how scene is displayed by moving a virtual camera. This can be expressed as yet another transformation of the coordinate system - we would like to transform coordinate system to align with the position of our camera. This transformation is commonly called view transform.

clip space - having accounted for model position in a scene and user interactivity all that remains is to provide vertex data in form that subsequent fixed function pipeline stage - the rasterizer - expects. Namely once vertex shader is finished fixed function processing will clip all geometry then perform perspective divide to obtain vertices in normalized device coordinates (NDC). Output of vertex shader is a 4 component vector which corresponds to a 3D position in homogenious coordinate system used in computer graphics due to its ability to represent non linear transformations using matrices. The output position of vertex is divided by the forth component in order to introduce perspective

The main responsibility of vertex shader is to transform vertices to clip space, which will be discussed in future subsection.

2.4.3 Tessellation

Tesselation stages were added as graphics hardware compute capability grew. With raw compute throughput outperforming bus throughput GPUs were equipped with hardware tesselation unit which can subdivide a larger triangle into batch of smaller ones. This allows for efficient generation of geometric detail on chip alleviating the issue of limited PCI throughput. To drive the tesselation stage two new shaders were introduced:

- **Tesselation control shader** which configures how hardware tesselator should subdivide a triangle.
- **Tesselation evaluation shader** which performs transformations on vertices generated by the tesselator.

2.4.4 Geometry shading

Geometry shader was introduced prior to tesselation stage. They operate on assembled geometric primitives and may even access primitives neighbors. Given primitive input geometry shaders output one or more primitive of the same type.

2.4.5 Fixed function vertex post-processing

Once all programmable vertex progressing has concluded, a series of fixed-function operations are applied to the vertices of the resulting primitives before rasterization. These operations include transform feedback, which captures processed vertex data, primitive queries to gather information about the primitives being processed and flat shading which applies a uniform attribute value to a whole primitive.

Primitives then get clipped against clip volume and client-defined half-spaces. The clip coordinates undergo perspective division, followed by viewport mapping to adjust for screen coordinates and depth range scaling.

2.4.6 Rasterization

If neither tesselation stage nor geometry stage was used in vertex processing, primitive assembly takes place (presence of any of the aforementioned stages would necessitate early primitive assembly). OpenGL converts geometric primitives used in currently processed draw call into base primitives which are points, lines and triangles. Mathematical representation of primitives is used during rasterization to determine if given fragment falls inside of primitive being rasterized.

Process of rasterization requires determining if given pixel position falls inside of rendered primitive. This process needs to account for point and line thickness. Polygon rasterization is obviously the most complex of the three. Prior to insidness test face culling is performed. This optimization culls a polygon based on the sign of surface normal computed based on edge ordering as specified in vertex array. This helps reduce overdraw which can be one of two main bottlenecks in modern rendering system, the latter being insufficient memory bandwidth.

Once pixel location was deemed inside a primitive a fragment is generated. A Fragment is a collection of data corresponding to specific pixel location. Most commonly its perspective corrected barycentric interpolation of vertex data across

the primitive's surface. Tough interpolation can be disabled from within vertex shader using `flat` qualifier on output variable declaration, as well as perspective correction with `noperspective` qualifier.

Once fragments are computed early per-fragment tests take place.

- **Ownership test** - determines if pixel at location (x, y) falls into the portion of the screen that active OpenGL context owns.
- **Scissor test** - checks if pixel at location (x, y) is contained within client provided list of axis aligned rectangles
- **Early Fragment tests** - stencil test, depth test and occlusion query which are normally performed after fragment processing can optionally be performed early. We discuss them in subsection on fragment post processing.

If all tests passed fragment is submitted for programable fragment processing.

2.4.7 Fragment processing

Programable fragment processing is performed by client provided fragment shader. The most essential task that fragment shader should perform is assign pixel a color. For that purpose data interpolated from rasterization is used. Most commonly fragment shaders perform texture mapping, lighting calculations, parallax mapping to emulate geometric detail and screen space effects like ambient occlusion, use signed distance functions and implicit surface equations to render otherwise complex scenes all by itself or create volumetric effects like clouds or visualize CT scan results.

2.4.8 Fragment post processing

2.5 GLSL

GLSL, which stands for OpenGL Shading Language, is a high level shading language with c like syntax developed by OpenGL Architecture Review Board to power programable processing stages in OpenGL pipeline. GLSL code is still relevant as it can be compiled into SPIR-V and used with Vulkan API.

Shaders

Independent compilation units written in this language are called shaders. A program is a set of shaders that are compiled and linked together, completely creating one or more of the programmable stages of the API pipeline

In OpenGL 4.6 and GLSL 4.60 there exist 6 types of shaders: vertex, tessellation control and evaluation, geometry, fragment and compute. All shaders except compute shader control appropriate parts of OpenGL pipeline as described in subsections above.

Compute shaders operate completely outside of graphics pipeline. They can access same resources as fragment or vertex shader like textures, buffers, images and atomic counters but they are not expected to produce data with predetermined form or semantics. They offer general purpose compute capability on the GPU. They function similarly to other existing general purpose GPU compute APIs like CUDA or OpenCL.

2.5.1 Variables

The main purpose of shaders if to transform received data to some other form. The data that the shader expects is defined using global variables with appropriate qualifiers. During Program linking OpenGL matches outputs from previous stage with inputs of the next stage. In case of vertex shader `in` variables should match with vertex attribute definitions specified in vertex array object. Tough in case of mismatch if attribute is disabled constant value can be provided, however thats rarely desired behavior. Similarly, `out` variables from fragment shader should match with framebuffer configuration. This process can be quite error prone and can lead to undefined behavior which can be difficult to diagnose and may have different consequences depending on actual hardware, os or driver versions.

Under no circumstances erroneous pipeline configuration should be allowed. Program containing such malformed configuration should be rejected by static analysis, and that was one of most important aspects of this study. To achieve that we attempted to express both GLSL variable declarations along with full OpenGL pipeline in Rust's type system in sucha a way to force type errors for invalid pipeline configurations.

We determined that keeping track of three variable qualifiers is essential to achieve that.

2.5.2 Variable types

Expressing GLSL variable type in Rust types was the obvious first step. GLSL defines a set of built-in types along with ability to create aggregate data types with C-like array and struct definitions.

For this work we focused on builtin types and arrays and omitted structures due to Rust's inability to encode layout guarantees for arbitrary types. Rust's built-in numeric types and arrays have however have well defined memory layout and

create a close set of possible types which allowed us to enumerate them express their memory layout in type system using traits.

GLSL's built-in types are divided into two groups: transparent and opaque types. Transparent types represent numeric data (plain old data) whereas opaque types represent handles to different resources like texture image samplers.

In case of transparent types there are 5 base numeric types: `float`, `double`, `int`, `uint` and `bool`. Floating point types `float` and `double` are accordingly IEEE-754 single and double prevision numbers, integers are two's compliment 32bit values and bool undefined representation but it can take only two values `true` or `false`.

All base numeric types can be aggregated into 2, 3 or 4 component vector types. Each vector type is named `TvecN` where **T** depends on inner base type: `b` for `bool`, `d` for `double`, `i` for `int`, `u` for `uint` and for `float` nothing is prepended to `vecN`. **N** is the number of components vector should contain.

Finally, there are matrix types of form `TmatN`. Matrices can contain only `float`s (`matN`) or `double`s (`dmatN`). The **N** depends on matrix dimensions it can be a single number 2, 3 or 4 for square matrices or can be arbitrarily combined pair of these numbers of form `NxM`, i.e. `mat2x4`, `mat4x2` or `dmat3x3`.

Data types used by GLSL have quite large memory footprint. Thats why OpenGL provides conversion mechanisms for data stored in buffers. Buffer data can be low bitwidth integer or float which will be normalized on access or even completely new OpenGL defined packed formats like `UNSIGNED_INT_2_10_10_10_REV` - a 32bit value which will be expanded to 4 `float`s.

This indirect mapping of OpenGL data to GLSL types is also essential to be statically verified just like shader input / output matching.

2.5.3 Variable storage qualifiers

The origin of data for a variable is encoded by a storage qualifier. We have already discussed that data within a shader can originate from previous stage / vertex buffers and that it can be saved as input to subsequent stage or framebuffer. These sources correspond to `in` and `out` qualifiers. Shaders can also declare uniform variables which are data associated with program itself. Value of these variables remains the same across the entire primitive being processed. All uniform variables are read-only and are initialized externally either at link time or through the API.

For bidirectional communication between shaders and API there exists `buffer` qualifier. Variables with such qualification are stored in buffer objects and can be both written to and read from by shaders and API.

Remaining qualifiers are ignored as they are irrelevant for the scope of this study.

2.5.4 Variable layout qualifiers

Statically asserting that the data passed though the pipeline is correct was the main goal of this study. .... Ability to statically assert that data flow though the pipeline is configured correctly depends on our ability to match neighboring stage inputs and outputs. In newer OpenGL versions one can use `location` layout qualifier to assign a variable a numeric index. This integer will be checked for uniqueness among all other variables that it shares storage qualifier with, and generate compilation error in case of overlap. When using locations based interface matching each `out` variable must have

It assigns an variable location and along with its storage qualifier it creates unique

TODO: tie this subsection together

“ An output variable is considered to match an input variable in the subsequent shader if:

- the two variables match in name, type, and qualification, and neither has a location qualifier, or
- the two variables are declared with the same location and component layout qualifiers and match in type and qualification.

  For the purposes of interface matching, variables declared with a location layout qualifier but without a component layout qualifier are considered to have declared a component layout qualifier of zero. ”

/chapterExisting solutions

There are many qualities of any software library one could consider important. In this research we focused foremost on providing minimalistic wrapper and staying as faithful as possible to original specification of the API. By this we mean that appropriate GL functions take analogous parameters as in original spec and have their names and semantics preserved. Major benefit of this approach is that we could simply follow the OpenGL specification when creating type safe facades around procedures.

Starting from these minimalistic principles we focused on providing maximal level of type safety. The main goal was to enable rejection of as many ill-formed programs at compile time as possible.

There are many levels of safety guarantees we can expect from any software package. In this analysis we devise

Here we consider alternative ways of programming computer graphics with use of OpenGL as rendering backend.

We distinguish between a language of choice and any framework at use.

/sectionNative C / C++ bindings

The simplest way one can program with OpenGL is using platform provided C bindings contained within an os provided dynamic link library (.dll for MS Windows, .so) along with appropriate function pointer loader. The requirement for the latter stems from common practice among OS vendors to officially guarantee distribution of very dated version of the specification (1.1 for Windows). This poses a requirement for manual function pointer loading at runtime, an approach that has two main benefits - abstracts away details of dynamic library loading for different platforms, - provides unified mechanism for using optional core standard extensions.

Additionally to function pointer loading one needs to initialize OpenGL context following platform defined protocol.

Most commonly there exists a library for each task, some examples for PC are GLEW for function loading and GLFW for window creation and context creation. Once these actions are accomplished one can use OpenGL in C or C++ provided appropriate attention to C interoperability.

Writing C application provides no auto

/sectionRust with unsafe bindings

Rust toolchain provides a utility for automatically generating Rust Foreign Function Interface bindings to C called `bindgen`. In this case all the setup needed for a Native C / C++ bindings application still applies. There exist appropriate counterparts to GLEW and GLFW. Once context is initialized and function pointers loaded one can call C functions but Rust will require one to use these functions inside unsafe context.

/chapterExisting solutions

There are many qualities of any software library one could consider important. In this research we focused foremost on providing minimalistic wrapper and staying as faithful as possible to original specification of the API. By this we mean that appropriate GL functions take analogous parameters as in original spec and have their names and semantics preserved. Major benefit of this approach is that we could simply follow the OpenGL specification when creating type safe facades around procedures.

Starting from these minimalistic principles we focused on providing maximal level of type safety. The main goal was to enable rejection of as many ill-formed programs at compile time as possible.

There are many levels of safety guarantees we can expect from any software package. In this analysis we devise

Here we consider alternative ways of programming computer graphics with use of OpenGL as rendering backend.

We distinguish between a language of choice and any framework at use.

/sectionNative C / C++ bindings

The simplest way one can program with OpenGL is using platform provided C bindings contained within an os provided dynamic link library (.dll for MS Windows, .so) along with appropriate function pointer loader. The requirement for the latter stems from common practice among OS vendors to officially guarantee distribution of very dated version of the specification (1.1 for Windows). This poses a requirement for manual function pointer loading at runtime, an approach that has two main benefits - abstracts away details of dynamic library loading for different platforms, - provides unified mechanism for using optional core standard extensions.

Additionally to function pointer loading one needs to initialize OpenGL context following platform defined protocol.

Most commonly there exists a library for each task, some examples for PC are GLEW for function loading and GLFW for window creation and context creation. Once these actions are accomplished one can use OpenGL in C or C++ provided appropriate attention to C interoperability.

Writing C application provides no auto

/sectionRust with unsafe bindings

Rust toolchain provides a utility for automatically generating Rust Foreign Function Interface bindings to C called `bindgen`. In this case all the setup needed for a Native C / C++ bindings application still applies. There exist appropriate counterparts to GLEW and GLFW. Once context is initialized and function pointers loaded one can call C functions but Rust will require one to use these functions inside unsafe context.

Chapter 3

OpenGL wrapper library

In this chapter, we demonstrate how Rust's type system can be harnessed to create a safe wrapper library for modern OpenGL, specifically targeting version 4.6. Our goal is to cover the most essential components of the OpenGL specification and staying as close to the original spec as possible. In many cases, we implement a minimal subset of functionality to demonstrate that, once a specific feature is in place, it can be readily extended to encompass a broader scope of the API.

Besides the wrapper library the purpose of this study was to identify common patterns that arise during type driven design.

Overview

The resulting library was named *GPU bulwark* since it provides strong foundations for safer programming on the GPU, and could easily be extended to other GPU programming APIs.

Library at its root is logically divided into two halves: (1) main OpenGL wrapper and (2) general-purpose auxiliary modules which contain implementations of various patterns we have recognized.

3.1 External dependencies

Our library utilizes several publicly available crates from crates.io, we will briefly discuss their purposes below:

- `gl` - generates raw OpenGL bindings for Rust using build script. Additionally, it exposes a single function that loads function pointers using the provided routine. These bindings use C types and need to be invoked in `unsafe` context.

- `derive_move` - is a procedural macro crate that expands `derive` to support more built-in traits. It significantly reduces code boilerplate.
- `concat_idents` - provides singular procedural macro that allows to concatenate identifiers akin to C's `##` operator. We utilize this macro for identifier generation for certain OpenGL names that strictly follow a naming convection. This yet again helps to reduce boilerplate, makes code more succinct and minimizes risk of typos.
- `nalgebra` and `nalgebra-glm` define algorithms and types for linear algebra computations. They are not used directly in our library for their functionality but rather for optional integration with `gpu-bulwark`.

Remaining packages are imported for use in examples only.

- `thiserror` and `anyhow` - very popular crates that make error handling more ergonomic.
- `raw-window-handle`, `glutin` and `winit` allow for cross platform window creation and OpenGL context initialization.

3.2 Auxiliary modules and crates

All general purpose design patterns we encountered during development are implemented in these modules.

3.3 Identified design patterns

In our exploration we found that patterns which tend to emerge during programming with types can be broadly divided into two categories: (1) compensation for language limitations (2) validation of program structure at compile-time (CT).

3.3.1 Compensation for language limitations

Rust is in continuous development. Some features have been work-in-progress for over years and are still nowhere near completion. Others have seen minimal-viable-product releases, and some are merely the subject of wishful thinking and speculation. Features we found useful in type-based design fall into all of these categories. Most of them can be emulated with varying levels of complexity and user experience degradation.

Stemming from often contrived usage of type system and different language features resulting error messages are very verbose and difficult to interpret.

Variadic Generics

Problem It is common practice among programming language developers to support variadic function arguments - functions which can accept arbitrarily many arguments. This capability is a major syntactic convenience and serves as a tool for more complex abstractions.

It is substantially less common to support variadic type parameters in generic types; in fact, Rust has no such language feature. One highly desirable use case for such variadic generics was identified: non homogenous collections.

Solution Rust has an excellent support for recursive types thanks to the `impl` syntax, and since lists can be defined recursively we derived a variadic generics emulation scheme from that. We used type level recursion on binary tuples (recursive step) and unit type as `Nil`. We call such recursive type list a **HList**. HLists can be wound to the left: first tuple component contains n - 1 elements and second the n'th type or to the right in reverse order. These two schemes are equivalent in terms of functionality, but differ in terms of potential user experience. In our use case appending new types to the end of a HList was by far the most common use case, and as such we almost exclusively use left wound HLists (LHLists).

Implementation of functionality for HLists needs to mirror their abstract and recursive nature. This can be achieved using two `impl`s, one for unit in the base case and another recursive for a the binary tuple for recursive step.

These homogenous collections have been implemented as an independent module called `hlist` which can be found in the root of our crate.

Use case In `gpu-bulwark` HLists are used every time variable-length user configuration is required, most notably to represent shader inputs, outputs, used uniforms or external resources like textures. Almost always we create a facade marker trait which joins together predefined pieces of functionality from `hlist` module and adds specialized requirements for HList member types in order to prohibit creation of invalid type list.

3.3.2 Const generics in const expressions

Problem Const generics fall into the category of partially implemented features. Const generic types depend on a value of limited subset of types, most notably numeric types, bool and unit. This feature, being in its early stages, has a significant limitation: const parameters must be literals or expressions using only literals. Const parameters cannot be used in any type level const expressions, they have to be used directly. As a result, we cannot perform

arbitrary compile-time (CT) computations on these parameters for purposes of verification.

Solution However, there is one exception to that limitation: associated constants. Associated constants can have their values computed using CT `const fn`s and themselves be used in such computations as parameters. These functions can panic with static error message (no formatting) and may cause compilation errors based on programable logic. As a consequence, different limitation was imposed: associated constants cannot be used as const parameters in types, they can only be used as values in code.

Use case Due to the lack of negative reasoning, as of yet, in Rust compiler we cannot express type inequality. The only viable solution would be to a blanket impl stating that two types are different if they are not the same type; since such a blanket would apply to user defined types as well.

CT validation that types are all different is required to assert that glsl variable layout locations do not overlap. In certain scenarios when layout components are used this overlap may be valid; we ignored it in this work because it can be easily taken into account in future releases.

We use associated constants and conditionally panicking `const` function to check that location ranges do not overlap.

3.3.3 Effect system

Problem First class effect system is a non-existent feature that would be of immense value in the context of an OpenGL wrapper implementation. Ability to type check function invocation context in OpenGL would be especially useful as we could encode presence of appropriate object binding using an effect.

Solution We instead were forced to opt for more error prone and verbose approach. Objects like textures or buffers can produce binder objects which in their constructor bind, and in their destructors unbind, object appropriate binding point. This lets us control binding using lexical scope but does not in any way prevent overriding of the binding.

Usae Case As already mentioned effect system would greatly improve the handling of context bindings in terms of statically verifiable correctness, as well as, user and developer experience.

3.3.4 Application of existing features

3.3.5 Markers

Problem Enumeration types are a core component of almost all currently used programming languages. In recent years many languages even gained ability to store dynamic data in enum variants. Such enums provide simple mechanism for statically typed polymorphism with dynamic variants. However sometimes this dynamic-ness of enums is a hurdle causing constant match or switch statements to crop you all over the code base, producing clutter and boilerplate. Sometime one simply wishes to encode static configuration based on a closed set of possible values.

Solution Markers are traits and types which don't provide any runtime behavior, but rather exist for purposes of conveying information and constrains on a type level. Marker traits provide no useful functionality but rather serve to impose relations and logical division on types.

Marker types don't hold any data and as such don't exist at runtime (they occupy zero bytes and are formally called Zero Sized Types - ZST). It is even possible to marker types to have type, lifetime and const parameters by using special compiler intrinsic datatype - `PhantomData` - which binds parameters but does not hold any value.

Marker traits along with marker types can be used as:

- compile-time enums – by limiting access to a marker using item visibility qualifiers we strictly control what types implement given functionality.
- marker trait based relations – we can express relations between types and make unsound parameter combinations a compile-time error.
- typing external resources – by using `PhantomData` we can attach type information to otherwise untyped parts of API.

Use Case We make heavy use of markers to implement entirety of glsl module which consists almost elusively of ZSTs for purposes of modelling shader `in`, `out` and `uniform` variables. Types representing these variables aggregated into hlists are specified by the user with help of GLSL DSL implemented using lightweight declarative macros.

Marker traits in miscellaneous `_::valid` modules define relations between valid combinations of data types. Buffer in raw OpenGL, due to C's lack of generics, has its buffer populated using `*void` and the documentation enumerates valid types. To make things worse validity of data types changes depending on what's the buffer's target. It is illegal for index buffer to contain anything other than unsigned integers, pixel buffers can contain almost everything and

vertex buffers, yet again, can contain only specific combinations of data. By associating a phantom type with a Buffer and using marker trait based validation relations on uploaded data we solve both of these issues.

This methodology can be extended to form **many modes** pattern, in which one uses marker types that implement trait containing generic associated types (GAT) to control behavior in more complex fashion than using non-generic associated types.

many modes We use many modes in `Variable<S, L, T, Store>` to abstract over kind of storage used for variable's type member - `Phantom` or `Inline`. `Phantom` uses `PhantomData` as its associated type and effectively discards value and `Inline` keeps it as is.

3.3.6 Subtyping

Problem Subtyping or inheritance is a very common concept in object oriented programming. In these languages one can create a type, and inherit from it to produce more specialized subtype which can be used in all places the parent can. This is

Using generic type and automatic dereferencing via `Deref` we can emulate the relation of subtyping

3.3.7 Type State

Type state is very powerful pattern that takes advantage of how rust understands generic types and allows for tracking runtime capabilities at compile-time. For the duration of this subsection it is worth to think about generic types as of type constructors which can be partially applied to create another constructor or fully applied to produce a type.

Recall from chapter 1 that Rust determines if `impl`s are coherent, what it means is that each associated item resolves uniquely. Generic type with one of it's parameters supplied can have an inherent `impl` for

3.4 OpenGL wrapper

Bibliography

[1] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.