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

29 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.

Od kilku lat w branży programistycznej rośnie zainteresowanie językami silnie typowanymi. Przejawia się to przez powstawanie zupełnie nowych technologii, w których silne systemy typów są fundamentem budowy języka lub w zmianach wprowadzanych do istniejących języków. Przykładami takich nowoczesnych języków są TypeScript, jako alternatywa dla JavaScript w świecie stron internetowych lub Rust w dziedzinie programowania systemowego zamiast C i C++. Starsze technologie rozszerzyły swoje systemy typów, na przykład w C# 8 wprowadzono jawne adnotacje o tym, że typ może przyjmować wartości `null`. Nawet dynamicznie typowany Python doczekał się znacznych ulepszeń w swoim systemie adnotacji typów [3].

Niniejsza praca — implementacja biblioteki opakowującej API OpenGL dla języka Rust — będzie próbą zademonstrowania w jaki sposób jego system typów może zostać wykorzystany do zwiększenia niezadwodności, łatwości utrzymania oraz rozwoju oprogramowania i jego niskopoziomowego bezpieczeństwa; a także w jaki sposób wpływa on na projektowanie API i strukturę kodu.

Contents

1 The Rust programming language **11**

1.1 Data types . . . 13

1.2 Built–in Rust Types . . . 13

1.2.1 Scalars . . . 13

1.2.2 Tuples . . . 14

1.2.3 Array . . . 14

1.2.4 Slice . . . 14

1.2.5 `str` . . . 14

1.2.6 User defined types . . . 15

1.3 Implementations . . . 15

1.3.1 Inherent implementations . . . 16

1.3.2 Trait implementations . . . 16

1.4 Generic types . . . 16

1.4.1 Type parameters . . . 16

1.4.2 Constant parameters . . . 17

1.4.3 Lifetime parameters . . . 17

1.4.4 Type Aliases . . . 18

1.5 Traits . . . 18

1.5.1 Associated functions and methods . . . 18

1.5.2 Associated types . . . 19

1.5.3 Associated Constants . . . 19

1.6 Implementation Coherence . . . 20

1.7 Orphan rules 20

2 OpenGL and GLSL **21**

2.1 Introduction 21

2.2 History 21

2.3 OpenGL objects 21

2.3.1 Buffer object 22

2.3.2 Vertex Array object 22

2.4 Graphics pipeline 23

2.4.1 Vertex Specification 23

2.4.2 Vertex Shader 23

2.4.3 Tessellation 24

2.4.4 Geometry shading 24

2.4.5 Fixed function vertex post–processing 25

2.4.6 Rasterization 25

2.4.7 Fragment processing 26

2.4.8 Fragment post processing 26

2.5 GLSL 27

2.5.1 Variables 27

2.5.2 Variable types 28

2.5.3 Variable storage qualifiers 29

2.5.4 Variable layout qualifiers 29

3 Existing solutions **31**

3.1 Native C / C++ bindings 31

3.2 Rust with unsafe bindings 32

3.3 Rust graphics libraries 33

3.4 Conclusion 33

4 OpenGL wrapper library **35**

4.1 External dependencies 35

4.2 Identified design patterns 36
4.2.1 Compensation for language limitations 36
4.2.2 Const generics in const expressions 38
4.2.3 Effect system 39
4.2.4 Subtyping 40
4.3 Validation of program structure at compile–time 41
4.3.1 Markers 41
4.3.2 Type State 43
4.4 OpenGL wrapper 47
4.4.1 GLSL module 47
4.4.2 Shaders 48
4.4.3 Program 48
4.4.4 Buffer 49
4.4.5 Memory Mapping 50
4.4.6 Vertex Array 50
4.4.7 Textures 50
4.5 Examples 52

5 Conclusions **53**

5.1 Identified benefits of type–driven design for systems programming . 53
5.2 Identified downsides of type–driven design for systems programming 53
5.3 What works 54
5.4 What could be improved 54

Bibliography **57**

Introduction

This study aims to demonstrate how extensive use of advanced type systems can change the way software is developed. Advancements in type systems allow their users to express increasingly more complex invariants and contracts for their programs using logic rules, and enable type checkers to proof that all these invariants are upheld at compile–time.

This is very desired as type systems not only offer extensive compile–time verification, but just as importantly logic based rules are substantially less susceptible to bugs due to programming errors like regular validation code is.

For over a decade now in many languages we have seen a gradual shift toward both stronger typing, and more of a functional approach to program specification. with strong and expressive types systems as a foundation of this trend.

We chose to implement wrapper library for OpenGL using Rust due to several important factors.

Rust has: a state of the art type system comparable to Haskell's, is natively compiled language with very good support for C interoperability, its ownership based model of resource lifetime management provides strong guarantees about when exactly destructor will be called, borrow checker allows to better express how different pieces of program interact with each other. Rust is also a systems programming language, and we wanted to showcase that it can be successfully applied to graphics programming [11]

OpenGL is a very mature, cross–platform, well understood and widely supported graphics specification. Its also arguably the best API for beginners in hardware accelerated graphics programming to start with. With all that being said its also quite dated, and its sequential model of execution does accurately represent the realities of graphics hardware.

We realized that Rust's type system can help to amend certain aspects of OpenGL which make it sometimes difficult to use. We tried to emphasize important aspects of the API like: control flow or graphics pipeline and impose a structure for different APIs by generalizing them, as well as, using strongly typed procedures and types. so users can focus on graphics programming and not have to worry about

different OpenGL obscurities and verifying by hand that numerous contracts are upheld.

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 which guarantees memory safety without a garbage collector.

1.1 Data types

1.2 Built–in Rust Types

1.2.1 Scalars

Scalars are the most basic types in Rust, representing single numeric values. There are four primary scalar types in Rust:

- **Integers**: These represent whole numbers and come in signed (`i8`, `i16`, `i32`, `i64`, `i128`, `isize`) and unsigned (`u8`, `u16`, `u32`, `u64`, `u128`, `usize`) varieties. The numbers in the names indicate the number of bits used for storage. `isize` and `usize` depend on the architecture of the machine, being 32–bit or 64–bit.

- **Floating–Point Numbers**: These represent numbers with fractional parts. Rust provides two floating–point types defined in IEEE754, namely single precision 32–bit `f32` and double precision 64–bit `f64`.

- **Char**: The `char` type represents a single Unicode code point. It can represent a wide range of characters, including emojis, and is always four bytes in size.

- **Boolean**: The `bool` type has two possible values: `true` and `false`.

1.2.2 Tuples

Tuples are a compound type that allows you to group together multiple values of different types into a single entity. The values are ordered, and each element can be of a different type. Tuples have a fixed size, meaning once they are declared, they cannot grow or shrink.

```
let person: (i32, &str, bool) = (25, "Alice", true);
```

You can access the elements of a tuple using dot notation with a zero–based index:

```
let age = person.0;  // 25
let name = person.1; // "Alice"
```

1.2.3 Array

An **array** is a collection of elements of the same type stored in a contiguous block of memory. Arrays in Rust have a fixed size, meaning once they are defined, their size cannot be changed. Arrays are useful when you need a collection of items that are all of the same type and the exact number of elements is known at compile–time.

```
let numbers: [i32; 3] = [1, 2, 3];
```

Here, `[i32; 3]` indicates an array of three 32–bit integers.

1.2.4 Slice

A **slice** is a reference to a contiguous sequence of elements within an array or another slice. Unlike arrays, slices are dynamically sized. Slices are used when you want to reference a portion of an array without needing to own the entire array. Slices have two forms: **borrowed slices** (`&[T]`) and **mutable slices** (`&mut [T]`).

```
let numbers: [i32; 3] = [1, 2, 3];
let slice: &[i32] = &numbers[1..];
```

Here, `slice` refers to the elements `[2, 3]` in the `numbers` array.

1.2.5 `str`

The `str` type is a string slice, which represents a view into a string data. It's typically seen as a reference, `&str`, and is used for passing strings around without needing to own them. `str` is an immutable sequence of UTF–8 encoded bytes.

```
let greeting: &str = "Hello, world!";
```

A `&str` is often created from string literals or from string manipulation functions, and it is the most commonly used string type when you don't need to modify the string contents.

Both `str`s and slices are unsized data types

1.2.6 User defined types

Rust provides 3 ways to construct user defined aggregate types, these are:

- `struct`s
- `enum`s
- `union`s

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. 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. 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. [12]

`enums`

An enumeration, also referred to as an enum, is a simultaneous definition of a nominal enumerated type as well as a set of constructors, that can be used to create or pattern–match values of the corresponding enumerated type [12].

One noteworthy aspect is that enums without any constructors cannot be ever instantiated.

1.3 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.

1.3.1 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.

1.3.2 Trait implementations

Trait implementations are discussed in section dedicated to traits 1.5.

1.4 Generic types

As of version 1.80 Rust provides 3 kinds of generic parameters types can use. These are: (1) type parameters, (2) constant (const) parameters and (3) lifetime parameters. Type which uses any generic parameters is said to be generic.

1.4.1 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.

```
struct Collection<T> { inner: Vec<T> }
fn add<T>(element: T) { todo!() }
```

However, one can't do much with truly 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 behavior 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 describe in section dedicated to traits 1.5.

1.4.2 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.

Constant generics, could for example be used to type–check dimensions of matrix multiplication like so:

```
struct Matrix<const N: usize, const M: usize> {
    array: [[f32; M]; N]
}
fn mat_mul<const N: usize, const M: usize, const K: usize>(
    lhs: Matrix<N, M>, rhs: Matrix<M, K>
) -> Matrix<N, K> { todo!() }
```

1.4.3 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 for.

```
struct Adapter<'a>(&'a mut String);

fn foo<'a>(used: &'a mut String, ignored: &mut String) -> Adapter<'a> {
    Adapter(used)
}
```

As shown in the listing above lifetime parameters need to be used in both types - so that during type's instantiation a relation can be established as to what data is

being borrowed. If lifetime annotations in the above listing were removed this code would fail to compile with error message stating:

```
help: this function's return type contains a borrowed value, but the
  signature does not say whether it is borrowed from ‘used‘ or ‘ignored‘
```

1.4.4 Type Aliases

A type alias defines a new name for an existing type, similarly to C's `typedef`. Type aliases are declared with the keyword `type`. Type aliases can have generic parameters which must be used in the aliased type.

```
type IntVec = Vec<i32>;
type MyVec<T> = Vec<T>;
```

1.5 Traits

Traits provide an ability to express shared behavior in abstract way [17]. 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, these items consist of: (1) functions or methods, (2) associated types and (3) associated constants.

1.5.1 Associated functions and methods

Associated functions and methods are the basic mean to express common behavior. In this regard they function like interfaces in most object oriented languages. Both methods and functions can have default implementations which implementors can override.

```
pub trait Foo {
    fn function() -> String;
```

```
    fn method(&self);
    fn default_impl() -> String {
        String::from("hello world!")
    }
}
```

1.5.2 Associated types

Associated types are type aliases associated with another type. Associated types cannot be defined in inherent implementations nor can they be given a default implementation in traits [12].

```
pub trait Foo {
    type Bar;
}
```

In associated type definition traits bounds can be defined. Compliance with these bounds will be checked during trait implementation.

```
pub trait Foo {
    type Bar: Clone;
}
```

Since only one trait implementation per concrete type (non–generic type) is allowed, traits with associated types create a type–level function from implementing types to associated types. Most importantly from the perspective of this study, in trait bounds, associated types can be constraint by equality to given specified type.

```
fn foo<T>(bar: T) where T: Foo<Bar=i32> { }
```

In the listing above `bar` must be a value of type `T` which implemented `Foo` with its associated type `Bar` equal to 32–bit signed int.

1.5.3 Associated Constants

Traits can be implemented generically for all types that satisfy given bounds using a generic implementation, often referred to as a blanket impl: `impl<T> Trait for T where T: ... { ... }`. This will even influence types from external crates. 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.

1.6 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. This is overly restrictive

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 is not rejected.

1.7 Orphan rules

To guarantee impl coherence between all version of a crate rust imposes, so called, orphan rules. Implementation of a trait for a type is passes an orphan rule check if either (1) trait or (2) type is defined in current trait. This is not an exact definition of these rules, but it will suffice for our purposes. This enables rust package manager **cargo** to handle different versions of the same dependency. What it means for us is that we cannot impl traits from standard library for types from standard library as such items would be external to any crate.

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 [5] [6].

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 a scalar or a vector type. Each attribute is assigned a zero–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 [14].

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 the Vertex Array Object.

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 [19].

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 [19].

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 transformations [22]. 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 in some local coordinate system (commonly center of

an object). These positions would commonly be loaded into GPU memory, in order to keep only single version of the mesh in memory. Such objects can be easily placed in the broader scene by providing a so called world transform.

world space — world space position of an object refers to its destined position in the scene.

view space — its common for 3D rendering applications to provide means of interacting with the scene. Whether 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 or perspective transform in case when perspective computations are applied at the same time.

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 homogenous coordinate system used in computer graphics due to its ability to represent non linear transformations using matrices.

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 geometric primitives [19]. Output primitives all must be of the same type which can be different then the input primitive type. For example given a single point, geometry shader can emit 2 triangles.

2.4.5 Fixed function vertex post–processing

Once all programmable vertex processing 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 [19].

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 have necessitated an earlier 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 the inside–ness 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. Though 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 during 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] [22].

2.4.8 Fragment post processing

After all fragments are processed fixed function processing takes over for the final time. Aforementioned ownership, scissor tests take place followed by stencil and depth test. Each of these test require additional buffer to be allocated, called appropriately stencil buffer and depth buffer.

Once enabled, the stencil test works by comparing a reference value against the value stored in the stencil buffer according to a specified comparison function. Depending on the result of this comparison, the pixel may be drawn, modified, or discarded. Functions `glStencilFunc` and `glStencilOp` customize when fragment should pass the test and whether to update current values in stencil buffer.

The depth test is much more commonly used. When fragments are rendered and depth test is enabled, based on their `xy` coordinates, their `z` coordinate is compared against current value in the depth buffer. Typically, if tested fragment's depth value is greater then the one stored in the buffer we can reject it since a fragment we processed earlier occludes current fragment. However, we can customize test outcome based on comparison result using `glDepthFunc`.

Occlusion queries count how many fragments passed depth test. This information can be read and used to implement amongst others: post processing effects, occlusion culling or Level Of Detail [2].

Finally alpha blending takes place. This process blends together values of current fragment with what's already stored in the framebuffer. Exactly what operation is performed can be configured to a limited degree similarly to functions determining outcome of depth and stencil test.

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 [16]. 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

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 is 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. Though in case of a 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 [16].

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 such a way to force type errors for invalid pipeline configurations. We determined that keeping track of the following 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, in the type system. 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. 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 precision 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. OpenGL specification calls this process *shader interface matching*.

Historically interface matching would match variables based on their names and types. Wiliest still valid, in newer OpenGL versions there exist a better way to match shader stages. One can use `location` layout qualifier to assign variable an integer index. This integer will be checked for uniqueness among all other variables that it shares storage qualifier with; program linking will fail in case of overlap. When using locations based interface matching each `out` variable in considered to match an input variable in the subsequent shader if the two variables are declared with the same location and component layout qualifiers and match in type and qualification.

Together `location`, storage and type qualifiers uniquely define a variable for the purposes of interface matching.

Chapter 3

Existing 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.

3.1 Native C / C++ bindings

The simplest way to program with OpenGL is by using platform–provided C bindings, typically contained within an OS–provided dynamic link library (e.g., .dll for MS Windows or .so for Unix–based systems), along with an appropriate function pointer loader. The necessity for a function pointer loader arises from the common practice among OS vendors of officially supporting only very dated versions of the OpenGL specification (e.g., version 1.1 on Windows). As a result, manual function pointer loading at runtime becomes essential.

This approach offers two main benefits:

- It abstracts away the intricacies of dynamic library loading across different platforms.
- It provides a unified mechanism for utilizing optional core standard extensions.

In addition to function pointer loading, an OpenGL context must be initialized according to platform–specific protocols. This context serves as the environment in which OpenGL commands are executed and keeps track of allocated resources.

Though both can be implemented manually by interfacing with raw OS APIs, typically there are specialized libraries available to handle each of these tasks. For example, on PC platforms, the OpenGL Extension Wrangler Library (GLEW) [8] is commonly used for function pointer loading, while the Graphics Library Framework (GLFW) [9] is often employed for window creation and OpenGL context management.

Once these tasks — function pointer loading and context creation — are completed, OpenGL can be utilized in C or C++ applications, provided that care is taken to ensure proper C interoperability.

Writing an application in C, however, does not benefit from features such as automatic resource management and type safety that higher–level languages may offer. Therefore, developers must manually manage resources such as memory and OpenGL objects, ensuring proper cleanup to avoid memory leaks and other potential issues. This can be beneficial as automatic binding and unbinding of OpenGL objects can result, if not done carefuly, in a very non linear and invisible control flow. In C++ the situation is a bit better due to RAII mechanism, object orientation and templates which all may be utilized to improve user experience but the critical limitation of C++ is its relatively weak type system (partially amended with Cpp20 concepts feature) [4]. C++'s templates are essentially advanced macros, all type variables are simply inserted and type analyses in performed on concrete types. There is no way to express abstraction in a way that type system could reason about it. To C++'s credit its compile time capabilities are much more expansive than Rust's, and could possibly achieve similar results though with notable downside of validation having to be hand written and not scaling by default, as opposed to type based rules.

3.2 Rust 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.

3.3 Rust graphics libraries

Among rust native graphics programming libraries most notable are `glium` [10], `wgpu` [13] and `rust-gpu` [21].

Whilst both provide varying levels of safety, non of them achieve nearly the amount of static verification we do. `glium` is specifically an OpenGL adapter but it operates on much higher level of abstraction. Shaders are untyped containers for code, there is a limited number of predefined buffers that are implemented separately, and drawing operates on very high level of abstraction. Underlying OpenGL is not very clearly visible, which in our work was a crucial design aspect.

`wgpu` is a very large project with over 400 contributors. Its is a cross–platform, safe, pure–rust graphics API, that does not expose any specific hardware programming API but rather creates a unified abstraction over GPU and uses graphics APIs as a backed. This again is not what we have strived for with `gpu-bulwark`, we tried to stay as true to OpenGL spec as possible and accentuate its design characteristics.

`rust-gpu` is a very interesting crate. Currently in heavy development, it compiles rust code to SPIR–V which is a binary, intermediate, assembly–like representation for graphics shader stages and compute kernels [15]. Perhaps we could integrate it with `gpu-bulwark` to contain the entirety of graphics programming entirely inside rust with perhaps build scripts using external tools, macros or `const fn`s when their capabilities are expanded.

3.4 Conclusion

Currently no other solution on the market provides the same level of static validation we're aiming for, while staying as true to original API design as possible.

Chapter 4

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, and stay as close to the original OpenGL specification 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

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.

4.1 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`.
- `thiserror` — very popular crate which provides declarative macros for faster error type generation.

4.2 Identified design patterns

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

4.2.1 Compensation for language limitations

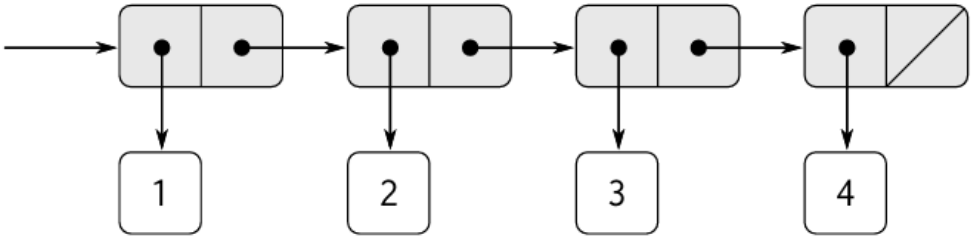
Rust is in continuous development. Some features have been work–in–progress for over the years and are still nowhere near completion. Others have seen minimal–viable–product releases, and some are merely the subject of wishful thinking and speculation. Features we found useful in type–based design fall 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. Rust does not no have such a language feature, and yet one highly desirable use case for variadic generics was identified: non–homogenous collections. Typical generic collections can contain one of more values of the same type. Structs and tuples can contain arbitrary data but the number, and names of aggregated types are fixed so the collection cannot grow.

Solution Lists can be defined recursively using pairs and special list termination symbol [1].



An empty list is simply the termination symbol and a list of length n is a pair containing an element in one slot and a list of length n - 1 in the other. Rust's type system understands recursion and provides a singleton unit type along with an ordered pair type - a binary tuple. Functionality for such lists would need to be expressed using traits since our very goal is to abstract over lists of different types — so no concrete type could possibly suffice. With all that in mind, we present a scheme for non–homogenous collections of types in Rust.

Starting with the most basic type list, we define a trait simply states that given type is a type list (called HLists henceforth [7]), and provide no functionality.

```
pub trait HList { }
```

Then we provide implementations for base and recursive cases.

```
impl HList for () { }
impl<Head, Tail> HList for (Head, Tail) where Tail: HList { }
```

We verify that our implementation works with a no–op function that simply checks if passed value is of type that implements `HList`.

```
fn test<HL>(_: HL) where HL: HList { }

fn main() {
  let list_1: ()                          = ();
  let list_2: (i32, ())                   = (1, list_1.clone());
  let list_3: (f32, (i32, ()))            = (42.0, list_2.clone());
  let list_4: (String, (f32, (i32, ()))) = (String::from("wow"), list_3.clone());

  test(list_1);
  test(list_2);
  test(list_3);
  test(list_4);
}
```

Type annotations are redundant, as types can be inferred, and were purposefully added to better visualize the mechanism. This sample can be run interactively in the rust playground — an online service hosted by rust foundation that provides a browser interface to the Rust compiler to experiment with the language. To run this listing follow this permalink.

One can create increasingly complex traits and implement them in a similar

manner, for example below we implement HList length computation.

```
pub trait HList { fn len(&self) -> usize; }
impl HList for () {
  fn len(&self) -> usize { 0 }
}
impl<H, T> HList for (H, T) where T: HList {
  fn len(&self) -> usize { self.1.len() + 1 }
}
```

[playground parmalink]

By deciding in which slot to place the tuple and in which the current element we can change the winding order of the HLists. 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).

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.

4.2.2 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 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 compile–time evaluated `const fn`s and themselves be used in such computations as parameters. Example with HList length could be adapted to use consts instead of methods like so.

```
pub trait HList { const LENGTH: usize; }
```

```
impl HList for () {
    const LENGTH: usize = 0;
}
impl<H, T> HList for (H, T) where T: HList {
    const LENGTH: usize = T::LENGTH + 1;
}
```

[playground permalink]

These functions can panic with static error message (no formatting) and may cause compilation errors based on programable logic. Lack of compile–time formatting makes it difficult to provide useful error messages, yet another advantage of using type–based validation. 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, conditionally panicking `const` function and environment variables to allow for compile–time validation that HLists of glsl variables of arbitrary length do not overlap.

We prepared an another example in the playground where HLists are checked to contain types with increasing numeric indices assigned to them.

4.2.3 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 binds, and in their destructors unbinds, object from appropriate binding point. This lets us control context bindings using lexical scope, but does not in any way prevent

distinct objects with the same target from overriding the global binding. A minimal example showing of effect emulation can be found on playground.

Use Case As already mentioned above, effect system would greatly improve the handling of context bindings in terms of statically verifiable correctness, as well as, user and developer experience. Even though in OpenGL 4.6 with direct state access we could circumvent this problem altogether, we purposefully chose to keep it and tried tackling it.

4.2.4 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 version of the original type – a subtype. Subtype extends functionality of base type and can seamlessly delegate all base type's method calls to the base type, and can be used in all places the base type can.

Solution Using generic type and automatic dereferencing via `Deref` we can emulate the relation of subtyping. The base type has a generic parameter which corresponds to any potential subtype, and implements `Deref` and `DerefMut` targeting that subtype.

```
struct Base<Child> { base: i32, child: Child }

impl<Child> Deref for Base<Child> {
    type Target = Child;
    fn deref(&self) -> &Self::Target { &self.child }
}

impl<Child> DerefMut for Base<Child> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.child
    }
}

struct Subtype;
type Inherited = Base<Subtype>;
impl Inherited { /* ... */ }
```

[playground permalink]

Subtype is obtained by defining a type alias for the generic base type with concrete subtype state specified. Subtype–specific functionality can now be specified

using an `impl` block on this alias. Such an `impl` would be coherent since all other subtypes have different nominal types representing their states and have full access to both base type's and subtype's states.

Use Case We applied this emulation of inheritance to model OpenGL objects. `ObjectBase<T>` contains base state and functionality and `T` provides implementation for subtype specific allocation, deallocation and context binding in a scheme very similar to template method pattern.

4.3 Validation of program structure at compile–time

4.3.1 Markers

Problem Enumeration types are a core component of almost all currently used programming languages. In recent years, many languages have even gained the ability to store variable–size data in their dynamic 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 pollute 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 possible for marker types to have type 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 an API.

```
use std::marker::PhantomData;

/// This pattern of pub trait in private module provides the sealing behaviour
/// Downstream users cannot use this trait but other public traits can
/// list it in its super traits.
mod sealed {
    /// A marker trait that will be implemented for predefined set of types
    /// from which downstream clients should choose pixel channel types in Pix
    pub trait Channels { }
}

// Four marker types used by pixel format to statically dispatch impl blocks.

pub struct R;
pub struct RG;
pub struct RGB;
pub struct RGBA;

impl sealed::Channels for R { }
impl sealed::Channels for RG { }
impl sealed::Channels for RGB { }
impl sealed::Channels for RGBA { }

/// Using marker types and traits we provide typing information to external re
pub struct PixelFormat<C, const COMPONENT_SIZE: usize>(PhantomData<C>)
where
    C: sealed::Channels
;

// Now we provide functionality for PixelFormats with channels from predefined
// set of possible configurations.

impl<const N: usize> PixelFormat<R, N> { pub fn r() { println!("R{}", N); } }
impl<const N: usize> PixelFormat<RG, N> { pub fn rg() { println!("RG{}", N); }
impl<const N: usize> PixelFormat<RGB, N> { pub fn rgb() { println!("RGB{}", N)
impl<const N: usize> PixelFormat<RGBA, N> { pub fn rgba() { println!("RGBA{}",

fn main() {
    PixelFormat::<R, 4>::r();
    PixelFormat::<RG, 8>::rg();
    PixelFormat::<RGB, 16>::rgb();
    PixelFormat::<RGBA, 32>::rgba();
```

```
    // Error — we prevent the very creation of nonsensical type.
    // PixelFormat::<i32, 32>;
}
```

[playground permalink]

Use Case We make heavy use of markers to implement entirety of glsl module which consists almost exclusively 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 to control behavior in more complex fashion than using non–generic associated types [18].

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.

4.3.2 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. Its name refers to static verification technique of the same name proposed by Robert E. Strom and Shaula Yemini [20]. Which allowed to determine the subset of operations permitted on a data object in a particular context, detecting and preventing semantically undefined execution sequences at compile–time.

This behavior can be elegantly implemented in rust using generics. 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 the section on coherence 1.6 that Rust determines if `impl`s are non–overlapping, what it implies is that each associated item can be resolved uniquely. Additionally `impl` blocks can be generic and implemented for generic types. These `impl`s can be generic over a subset of parameters, by providing fixed types for certain parameters.

```
struct Generic<T, H, F>(T, H, F);

impl<F, H> Generic<i32, F, H> { /* ... */ }
impl<F, H> Generic<u32, F, H> { /* ... */ }
impl<H> Generic<u32, (), H> { /* ... */ }
```

Rust understands that `Generic<i32, F, H>` and `Generic<u32, F, H>` must be different types (note the difference `i32` and `u32`), and as such any implementations for these two types must be disjoint. This behavior is what empowers type state pattern. We can intentionally add or use an existing generic parameter (const parameters can be used as well) to encode certain states and provide functionality for the main type in that specific state. Along with state specific type capabilities state transitions between the states are provided, allowing to encode a compile–time state machine.

In the below listings we show how type state, along with markers could be used to statically guarantee, that only users who were successfully authenticated can access secret information. We start by creating marker types `Anonymous` and `Verified` which represent different states a user session can be in. We constrain the possible set of state types, and assign each state its own data with `AuthenticationStatus`.

```
pub trait AuthenticationStatus {
    type Ctx: Sized;
}

pub struct Verified;
pub struct Anonymous;

/// Empty state is associated with an anonymous user.
impl AuthenticationStatus for Anonymous {
    type Ctx = ();
}

/// Representation of a user.
pub struct User {
    pub user_name: String,
    /* ... */
}
```

```
/// If user is logged in, User data can be obtained.
impl AuthenticationStatus for Verified {
    type Ctx = User;
}
```

To represent a permanent connection (like a TCP socket) we created the `Connection` type. Its a simple wrapper around IPv4 socket address thats used solely as a identifier, no actual network connections are made. Its purpose it to demonstrate that type state can accommodate persistent data by moving it between states.

```
struct Connection(Ipv4Addr);

impl Connection {
    pub fn new(client: Ipv4Addr) -> Self {
        println!("connection established with: {:?}", &client);
        Self(client)
    }
}

impl Drop for Connection {
    fn drop(&mut self) {
        println!("connection closed with: {:?}", &self.0);
    }
}
```

Finally we define type state based `Session` type.

```
struct Session<AS> where AS: AuthenticationStatus {
    type_state: PhantomData<AS>,
    connection: Connection,
    ctx: AS::Ctx,
}

impl Session<Anonymous> {
    fn new(ip: Ipv4Addr) -> Self {
        Self {
            type_state: PhantomData,
            connection: Connection::new(ip),
            ctx: (),
        }
    }
}

impl Session<Verified> {
```

```
    fn secret(&self) -> String {
        "'a secret'".into()
    }
}
```

Session in each state holds a `Connection` instance, phantom data and ctx to store current state type and any data associated with it. We then precede to define semantics of our type state. We allow direct creation of only anonymous sessions, as well as access to secrete information for authenticated users. Now all that remains are the state transitions.

```
#[derive(Error, Debug)]
enum LoginError {
    #[error("invalid password for {user_name}: {password}")]
    InvalidPassword { user_name: String, password: String },
}

fn db_check_password(user: &User, password: &str) -> Result<(), LoginError> {
    (password == "password")
        .then_some(())
        .ok_or(LoginError::InvalidPassword {
            user_name: user.user_name.clone(),
            password: password.into(),
        })
}

impl Session<Anonymous> {
    /// We provide ability to log in only for anonymous users.
    pub fn log_in(self, user: User, password: &str) -> Result<Session<Verified
        db_check_password(&user, password).map(|_| Session {
            type_state: PhantomData,
            connection: self.connection,
            ctx: user,
        })
    }
}
```

We provide some basic mocks of the authentication mechanism and use it to define the `log_in` method on `Session<Anonymous>`. It consumes an anonymous session, validates given credentials and constructs validated session's instance if validation was successful, which is the only way to obtain an instance of the `Session<Verified>` type. That's how type state enforces that only logged in users can access secret information. This invariant is guaranteed at compile-time. However, if the actual transition succeeds is determined at runtime. We statically en-

forced a sequence of operations, and prevented function invocation sequence which is semantically incorrect.

A full sample along with few examples can be accessed via [playground permalink].

4.4 OpenGL wrapper

Scope

Our goal with this study was foremost to explore how Rust's type system can be utilized to improve static validation of OpenGL programs. We by no means meant to cover the entirety of OpenGL functionality only the most essential aspects like shaders, programs, vertex arrays and buffers and textures. Nevertheless, a great deal of consideration was taken before every major design decision to ensure that one could simply duplicate existing solution from a single API variant to all others and obtain the same level of functionality and protection. Additionally, we tried staying as true to original OpenGL API as possible. We preserves most of the semantics, function names and parameters that made sense. All objects don't retain any parameters, they simply forward them to appropriate GL procedure calls.

In the end we arrived at minimal working example of a type–driven OpenGL 4.6 core wrapper that provides ability to create all the aforementioned GL objects and program even moderately complex computer graphics in a safer fashion.

4.4.1 GLSL module

Programming in OpenGL consists of GL API and GLSL shaders. This division is at very core of our library as it's essential to correctly capture how glsl and opengl interact.

We settled on a design were shaders and pipeline configuration are the original source of truth and that's were `gpu-bulwark`'s user must start development. Description of graphics pipeline and everything related to glsl is encompassed in the `glsl` module. The most important component of that module is the `Variable<S, L, T, Store = md::Phantom>` type. It represents an AST–like node for glsl variable at a type level. It contains type parameters for 3 qualifiers we discussed in chapter 2: Type, Storage and Layout. These qualifiers are most essential from the perspective of correctness verification. We use the `Variable`'s type parameters as follows:

- type qualifier — it is used most notably in matching vertex array's attribute definitions and shader matching during program construction.

- storage qualifier — since variables with different storage qualifiers can share the same locations its imperative to distinguish between storage qualifiers during location overlap checking.
- layout qualifier — both location and binding values are used along with previous qualifiers to determine if shader inputs, outputs or uniforms are defined in a valid way.

Additionally `glsl` defines compatibility between glsl `in` types and gl attribute types.

4.4.2 Shaders

`gpu-bulwark` provides type state builders for complex objects like shaders or programs. Type state builder has a state for each type parameter we determined that object requires. In case of shader it has one state parameter that corresponds to compilation status. The `create` method creates a shader in `Uncompiled` state.

In this state the only operation one can perform is provide shader source code. In order to attach a shader to a program, shader needs to be compiled. Compilation can either succeed and return shader object now in `Compiled` state or a compilation error back to the user. Once shader has been compiled successfully one can specify what uniform variable this shader requires or precede forward by converting shader to either shader containing a pipeline stage entry point `main`, or to `Lib` shader for use for linking. `Main` shader require specification of inputs or outputs using appropriate `glsl::Variable`s defined in global scope using declarative macros.

Such shaders are ready for attachment to program Objects.

4.4.3 Program

In this study we implement non–separable program objects, which means that at least vertex and fragment shaders need to be specified, as well as, that all stages have to match during linking.

Builder

Programs build using the most complex type state builder. It contains six type parameters corresponding to

- target shader of most recently attached shader with entrypoint
- vertex input
- outputs of most recently configured stage

- program uniform definitions
- uniform declarations from most recently configured stage
- declarations of external resources used by the program

Program building is divided into two parts: (1) uniform specification and (2) pipeline configuration.

Uniform Specification

During uniform specification one can define uniforms that program contains by defining their type and providing their initial values, along with declarations of external resources that program uses. Although both are represented by uniforms in GLSL transparent uniforms differ in their meaning from opaque types which all represent handles to resources external to the program instead of plain old data like matrices or vectors.

Pipeline Configuration

Once uniforms and resources builder transitions to vertex shader stage where obligatory main entrypoint for vertex stage needs to be provided. From here type state will force the user to follow valid configuration paths for the pipeline as defined in the specification, validating that outputs from most recently attached entrypoint shader match inputs to the newly provided one using location layout qualifiers on shader provided inputs and outputs, resulting in compilation error in case of a mismatch. This traversal always concludes with specification of the fragment stage entrypoint shader where linking can be performed and actual Program object obtained.

If program is obtained it means that pipeline is configured correctly, program is linked and ready for use. Otherwise either compilation or runtime program link error was generated. In the current version we do not validate that representation of shader interfaces actually matches shader status, but it can be easily achieved by either parsing shader source code or querying shaders interfaces using OpenGL introspection API.

4.4.4 Buffer

Buffer objects heavily leverage phantom types and markers to provide type information and content type validation to otherwise untyped integers.

Buffers are created, as for all objects, with `create` function. Once buffer is created its data store can be allocated and populated with `data` function which expects one type parameter for usage hint.

4.4.5 Memory Mapping

Buffers have an interesting feature that they can be mapped into memory and used with Direct Memory Access. We provide this functionality using auxiliary `Mapped_` types which borrow a buffer and are smart pointers that provide `Deref` and `DerefMut` to slices, allowing for idiomatic store access. A noteworthy implementation detail is that `Mapped_` smart pointers leverage the borrow checked to safeguard against an error condition during rendering where draw call was issues using vertex array that sources vertex data from memory mapped buffer. Smart pointers hold references to buffers borrowed in turn from VAO, and since `draw_arrays` method also requires mutable reference to VAO borrow checker will reject a program if any buffer used as vertex source is mapped during drawing.

4.4.6 Vertex Array

Vertex Array can be created with `create`. It exposes a single method `vertex_attrib_pointer` which takes ownership of a Buffer bound to `BUFFER_ARRAY` and stores it in VAOs internal HList of `Attribute`s. Attributes encompass buffer, vertex format and attribute index. This is a complete set of information needed to match vertex shader input variables.

Draw Calls

There are three variants of regular `draw_arrays`. First is designed for programs with no inputs, outputs, uniforms and resources — ones which essentially have the entire scene encoded in shader sources code. Second expects vertex array object which it binds and draws as many triangles as VAO specifies. Types of attributes are checked for compatibility against program inputs defined using glsl variables. Finally the third version, the most general, expects VAO and handles to texture bindings which are matched against external resources program uses.

4.4.7 Textures

Textures have the broadest API of all the OpenGL objects. Textures consist of three components: (1) sampling parameters, (2) texture parameters and (3) texture images, but only the storage benefits from strong typing due somewhat complex allocation and wide range of rules regarding valid parameter combinations which all can be easily expressed and checked at compile–time.

Textures support mipmaps in order better antialias textures during sampling. Mipmaps are a sequence of images generated from original image by halving its dimensions until they all reach 1px.

Storage

When texture is created its storage kind (immutable, mutable, buffer) and textures dimensionality (1D, 2D or 3D) are defined by the function name.

Textures can have 3 types of their storage

- mutable texture owned — allocated using `TexImage*` family of functions. These are the earliest form of textures. Storage can be later reallocated, hence mutability. Mipmaps need to be manually allocated and uploaded which is quite error prone. Pixel data for mutable storage can be specified directly using `TexImage* data` parameter of a Buffer bound to `PIXEL_UNPACK_BUFFER`. Data may also be modified using `TexSubImage*` functions.

- immutable texture owned — allocated using `TexStorage*` family of functions. This is the most recently added kind of texture backing storage. Once texture with this storage is allocated it cannot be reallocated. Major benefit of using immutable textures is that mipmaps are allocated automatically. Storage immutability allows to create views into the texture providing an opportunity for better memory conservation, and substantially simplifies work for the driver. Pixel data for these kinds of storage must be supplied using `TexSubImage*` functions.

- buffer backed texture — memory and content of texture come from buffer bound to `BUFFER_TEXTURE`, and similarly such a texture must be also bound to `BUFFER_TEXTURE`.

In our work we focussed on immutable storage textures due to their automatic mipmap allocation which frees us from implements compile–time mipmap completeness validation.

Image Format

Image formats (texture's internal format) are implemented in `texture::image` module. They describe memory layout pixels in GPU's memory and their interpretation. We focused on implementing sized formats as they specify precisely component count and size and such precision lends itself well to type level validation. They are represented by a `Format<Components, ComponentType, Interpretation>`. Besides component count and type, image formats have one additional parameter: `Interpretation`, it symbolizes how shader should interpret these values: either integers or floats.

Pixels

Once texture is allocated it needs to be initialized with pixels, by means of a pixel transfer operation. The simplest way to transfer pixel data is by using `TexSubImage*` functions. Ranges along each texture dimension have to be specified where pixels will be substituted, along with pixel type and format parameters both of which we encoded in types and validate them against internal image format of the targeted texture. Format parameter has dual meaning it describes: (1) what texture channels to target with current pixel transfer (which indirectly determines number of components in provided data) with formats like `RED`, `GREEN`, `BLUE`, `BGRA` or `BGR`, and how pixels are to be interpreted during texture sampling — as integers or floats — which needs to match texture image's `Interpretation` parameter.

Pixel transfer operations are implemented in such a way that type inference treats internal format's types as valid ones and will validate pixels that user is trying to transfer against that and report a compilation error if pixels are not compatible.

4.5 Examples

Examples demonstrating usage of `gpu-bulwark` are provided in `examples` directory, located in the root of the project. There are four samples which increase in complexity as follows:

- `hello_triangle` — the simplest example that shows the most basic rendering scenario where program is hard coded to produce a white triangle on the screen
- `hello_vertices` — this sample demonstrates basic vertex attribute configuration using a vertex buffers, vertex array and program input and VAO attribute matching. Two vertex attributes are used: one for vertex positions and another for vertex color. This sample is interactive, user can use A, S and D keys to modify color values stored in buffers which is implemented using buffer memory mapping and using buffers via normal rust slices.
- `hello_uniforms` — sample that shows how more complex programs which use uniforms may be constructed. Program implements an interactive 3D camera which can be controlled with mouse and keyboard to navigate in 3D space and view the scene: a single rainbow triangle. This interactivity is produced by using two uniform variables

Chapter 5

Conclusions

5.1 Identified benefits of type–driven design for systems programming

Type–driven design forces greater attention during the initial phase of software creation. The logical principles underlying the type system allow for a very precise description of the program's structure, algorithm invariants, as well as the interactions between different components.

Enforcing greater attention during system architecture design brings significant benefits in later stages of development. Type annotations also serve as a very good source of self–documentation.

The biggest benefit of extensive use of the type system is the static verification of the correctness of the analyzed programs. Rust's type system allows for the expression of a vast number of contracts.

5.2 Identified downsides of type–driven design for systems programming

This approach to programming requires expert knowledge of the language and an understanding of advanced concepts in logic and type theory, which significantly limits the accessibility of this method of software development due to the potential lack of specialists.

A large portion of the patterns and techniques presented in this publication requires writing a lot of code that does not necessarily carry much meaning — produces boilerplate. Error messages generated for highly recursive types are often unreadable and convey very little useful information. Specifically, in the case of Rust, the lack of certain features significantly complicates programming and forces

the addition of even more unreadable code, bending the language's functionalities beyond the creators' intent.

Even for a small project like our library, compilation times become noticeably longer, making code iteration more difficult.

Reverting form an erroneous design decision causes cascading errors and requires changes in many places of the codebase. Due to this it is difficult to maintain backwards compatibility.

5.3 What works

We managed to create a minimal wrapper for OpenGL that allows for the realization of a vast majority of even quite complex applications.

The library protects against misuse the most commonly used and critical functionalities of OpenGL, such as data flow through the pipeline, memory allocations, and resource management. A large part of the specification would not benefit from extensive type coverage because it operates on non–programmable parts of the pipeline where we have data from guarantees, there is no chance to invoke undefined behavior. These parts, when improperly configured, will only produce incorrect computation results rather than critical errors.

5.4 What could be improved

The minimalism of our wrapper lies in covering disjoint interfaces in the smallest possible way, which can be mechanically generalized to other, often simply more general, cases without encountering any problems.

Not all objects provided by OpenGL were implemented because they are quite niche and solve highly specialized problems. They are not necessary for constructing all applications. Such functionalities include immutable buffers, mutable and buffer–backed textures, asynchronous pixel transfer operations, constructing a context within our package, instanced rendering, supporting arbitrary types through macros and traits as vertex attributes.

From a user experience perspective, many aspects could be improved. With substantial effort, we could wrap all important type parameters in a wrapper type, with all validation traits implemented for it, ensuring that every program type-checks and moving validation to be run–time checked. Which mode of validation if used could be controlled using an environment variable and a type alias.

However, this approach would reimplement verification functionality to occur at runtime. From the user's perspective, this change would be partially transparent.

Since, during compile-time validation, these types would be appear, potentially causing even more confusion. At the same time, it would provide much better runtime diagnostics. Perhaps switching between these modes of validation could enhance user experience by improving error messages.

With future Rust releases some missing, incomplete or limited features could be improved allowing us to convey the same ideas more simply, and with better compiler support.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

[2] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real–Time Rendering*. AK Peters/CRC Press, 2019.

[3] Python Documentation Authors. *Python Typing Module*. `https://docs.python.org/3/library/typing.html`.

[4] CppReference Contributors. *C++ Reference*. `https://en.cppreference.com/w/`.

[5] Wikipedia Contributors. Iris gl wikipedia article. `https://en.wikipedia.org/wiki/IRIS_GL`.

[6] Wikipedia Contributors. Opengl wikipedia article. `https://en.wikipedia.org/wiki/OpenGL`.

[7] Frunk Developers. *Documentation of hlist module from frunk crate*. `https://docs.rs/frunk/latest/frunk/hlist/index.html`.

[8] GLEW Developers. Glew website. `https://glew.sourceforge.net/`.

[9] GLFW Developers. Glfw website. `https://www.glfw.org/`.

[10] Glium Developers. Glium github repository. `https://github.com/glium/glium`.

[11] The Rust Project Developers. Rust lang website. `https://www.rust-lang.org/`.

[12] The Rust Project Developers. *The Rust Reference*. `https://doc.rust-lang.org/stable/reference/`.

[13] gfx-rs Developers. wgpu github repository. `https://github.com/gfx-rs/wgpu`.

[14] Khronos Group Inc. *OpenGL Wiki Hosted by Khronos*. `https://registry.khronos.org/OpenGL-Refpages/gl4/`.

[15] The Khronos Group Inc. *SPIR-V Specification*. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

[16] John Kessenich, David Baldwin, and Randi Rost. *The OpenGL Shading Language, Version 4.60.8*. The Khronos Group Inc., 2020. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[17] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2023.

[18] Niko Matsakis. Many modes: A gats pattern, 2022. https://smallcultfollowing.com/babysteps/blog/2022/06/27/many-modes-a-gats-pattern/.

[19] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile)-October 22, 2019)*. The Khronos Group Inc., 2019. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[20] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[21] Embark Studios. rust-gpu github repository. https://github.com/EmbarkStudios/rust-gpu.

[22] Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Pearson Education, 2007.