Type-safe(r) model of geometry in 3D graphics

University of Turku
Department of Computing
Master of Science Thesis
Computer Science
June 2026
Johannes Dahlström

UNIVERSITY OF TURKU
Department of Computing

Johannes Dahlström: Type-safe(r) model of geometry in 3D graphics

Master of Science Thesis, 104 p., 5 app. p.
Computer Science
June 2026

This thesis investigates the use of type-level techniques for reducing programming errors in 3D graphics programming, in particular as relates to the representation of geometry and coordinate transformations. Existing theoretical and practical work on the subject is fairly scarce, and a review of several popular 3D programming APIs reveals that none attempt to alleviate or prevent such errors either at runtime or compile time.

A case study of Retrofire, a software 3D rendering library written in the Rust programming language, is presented. Retrofire employs generic types (parametric polymorphism), traits (type classes), and tag (phantom) types to rule out many of the identified programming errors at compile time, with a goal of minimal or zero impact on runtime performance.

The library is evaluated from the viewpoints of performance and developer experience. The conclusion is that the API is useful and helps prevent bugs in real-world programming, but more experience in writing complex applications with Retrofire is required for deeper understanding of its ergonomics. The library also appears to attain its goal of minimal performance overhead compared to a more conventional API; questions requiring further study include optimal memory management and use of SIMD vector operations.

Keywords: 3D, rendering, geometry, type systems, type safety, Rust

Contents

1 Introduction **1**

2 Motivation and research goals **4**
2.1 Sources of programming errors 6
2.2 Research questions 8

3 3D graphics in brief **10**
3.1 Geometry 11
3.1.1 Some conventions 11
3.1.2 Vector spaces 12
3.1.3 Linear transforms 14
3.1.4 Affine spaces 17
3.1.5 Affine transforms 19
3.1.6 Euclidean spaces 20
3.1.7 Homogeneous coordinates 21
3.1.8 Perspective projection 21
3.2 Rendering 22
3.2.1 Geometry representation 23
3.2.2 The rendering pipeline 25
3.2.3 Transformations 25
3.2.4 Shading 29
3.2.5 Colors and color spaces 30
3.2.6 Rasterization 33

4 Type systems **35**
4.1 A brief history of types 36
4.2 Benefits of typing 38

4.3 The Rust programming language . . . 40
4.3.1 Memory safety . . . 40
4.3.2 Algebraic data types . . . 42
4.3.3 Pattern matching . . . 44
4.3.4 Special types . . . 44
4.3.5 Traits and generics . . . 45
4.4 Related work . . . 48
4.4.1 Modeling geometry . . . 48
4.4.2 Type-level approaches . . . 49
4.4.3 Dimensions and units of measure . . . 51
4.4.4 Practical work . . . 52

5 Typing a renderer **54**
5.1 Introducing Retrofire . . . 54
5.2 Modeling affine and linear spaces . . . 57
5.2.1 Points and vectors . . . 57
5.2.2 Spaces and bases . . . 59
5.2.3 `Affine` and `Linear` implementations . . . 60
5.3 Affine and linear transforms . . . 61
5.3.1 `AffineMap` and `Compose` traits . . . 61
5.3.2 Matrices . . . 63
5.3.3 Matrix operations . . . 64
5.4 Colors and color spaces . . . 66
5.5 Angular quantities . . . 68
5.6 Shaders . . . 70
5.7 Other related APIs . . . 71
5.7.1 Cameras . . . 72
5.7.2 Splines . . . 72
5.7.3 Intersection tests . . . 74

6 Analysis and discussion **75**
6.1 Performance . . . 75
6.1.1 Methods . . . 77
6.1.2 Data locality . . . 77
6.1.3 Vectorization . . . 79
6.1.4 Monomorphization . . . 81
6.1.5 Code inlining . . . 82

6.1.6 Memory copies and allocations 84
6.1.7 Benchmarks 86
6.2 Programmer ergonomics 88
6.2.1 Methods 88
6.2.2 Escape hatches 89
6.2.3 Type inference 91
6.2.4 Build speed 92
6.2.5 Compiler diagnostics 94
6.2.6 Generic parameters and trait bounds 95
6.3 Summary of findings 98
6.4 Final thoughts and future directions 101

7 Conclusion **103**

References **105**

Appendices

A Code listings **A-1**
A.1 A simple example program using Retrofire A-1
A.2 A long, cryptic compiler error A-3
A.3 A vertex shader written in two ways A-4

1 Introduction

The field of 3D graphics programming is concerned with turning three-dimensional geometry, specified in an abstract numeric form, into a two-dimensional image or animation that depicts the geometry from some specified perspective. Its many applications include computer games, animated films, special effects, and computer-aided design. The use of 3D graphics is so prevalent that essentially all modern consumer computers, including mobile devices, contain hardware specifically designed to accelerate 3D rendering.

The fundamentals of 3D rendering have not changed much since the early days of computer graphics in the 1960s. The standard representation of geometry is a triangle mesh, with vertex positions and associated data expressed as Cartesian vectors. Transformation matrices are used to represent mappings between several coordinate frames, from the so-called model space to the final pixel coordinates in the screen space.

Popular graphics programming libraries model vectors and matrices as simple tuples of floating-point numbers. This low level of abstraction is flexible and efficient, but opens the door for many logical errors. For instance, one data type is used to model both vectors and points, which enables erroneous operations such as the addition of two points. Another example is the fact that different

coordinate frames are not explicitly represented, again permitting many operations that are not geometrically meaningful.

Based on the experience of this author and others, these sorts of errors are not uncommon and are often difficult to debug. This gives an incentive to explore how a more strictly typed set of geometry data types might help to prevent errors. This thesis is the product of such an investigation. It presents a case study of Retrofire, a software rendering library developed by the author. It is written in Rust, a modern general-purpose programming language that emphasizes productivity, performance, and correctness. Retrofire implements a strongly typed model of 3D geometry, allowing many logic errors to be caught at compile time. The structure of the thesis is as follows.

Chapter 2 motivates the research by exhibiting several issues with traditional 3D programming APIs that might be alleviated with stricter typing. It presents the research questions and the methodology used. Chapter 3 provides background on the linear and affine algebra involved in representing and manipulating 3D geometry. It also gives an overview of the most essential stages of a typical 3D rendering algorithm, including transforms, shading, and rasterization. Chapter 4 discusses type systems and how typing is used to help prevent programming errors. It gives a brief introduction to Rust. Existing academic and practical work related to geometry representation and type-level techniques is also reviewed.

Chapter 5 introduces the Retrofire renderer and its strongly-typed geometry API. It explains how the API is designed to alleviate each of the problems identified in Chapter 2. Chapter 6 provides analysis on how well the Retrofire API achieves its goal of improving correctness without impacting performance or developer experience. Chapter 7 concludes the thesis.

Declaration of AI use

Claude.ai was used for editorial feedback and proofreading of a near-final draft version of this thesis. All feedback provided by the model was reviewed by the author and manually applied if deemed pertinent. The Retrofire library was designed and implemented solely by the author without the use of AI tools.

2 Motivation and research goals

Typing is a robust mechanism for ruling out a class of programming errors for every possible execution of a program, without actually having to run the program. Types are also a valuable tool for modeling domain entities and making program code more self-documenting and easier to develop and maintain. Many contemporary programming languages are equipped with type systems that include features such as bounded parametric polymorphism (generic types), algebraic data types (sum and product types), and pattern matching. These tools can be used to create powerful abstractions. [1]

In high-performance programming fields such as 3D graphics, however, complex abstractions may be frowned upon. Abstraction layers can incur unwanted time or space overhead, make it more difficult to reason about code and its performance, and hinder the ability of the compiler to optimize the code. High-level, strictly typed APIs may also be too rigid, impeding valid use cases, and the need to "appease" the type checker may slow down prototyping and exploratory programming. [2]

On the other hand, a low level of abstraction has several downsides. Besides programming errors, it may cause learner confusion and lead to code that is difficult to understand and maintain. Comments can be added to source code

to document conditions and invariants, but it is common to forget to keep documentation up to date as the code evolves. Automatic tests also help prevent errors and regressions, and function as documentation of expected behavior, but may leave many execution paths uncovered, and some properties are difficult to automatically verify.

For several decades, the two major languages favored in high-performance programming have been C and C++. C is a simple language with a very rudimentary type system, largely unchanged since the origins of the language in the 1970s. C++, on the other hand, has a rich and powerful type system, but is hampered by the requirement to retain backwards compatibility with C. The language has also grown complex enough that the use of some of its more advanced or obscure features have been discouraged or outright forbidden in many coding conventions. [3]

Susceptibility to errors has been accepted as the price to be paid for performance and low-level access to the hardware, as no seriously considered alternatives have existed. In the author's experience, mastering a low-level language may even be seen as a point of pride, with criticisms dismissed as lack of skill. With their vast ecosystem of libraries, compilers, and tools, few other languages have been able to challenge C and C++. [3]

Rust [4] is a modern general-purpose programming language that aims to combine safety and performance by means of some state-of-the-art type system features. Inspired by the ML family of functional programming languages, it also implements some features well-established in functional programming, but rare in mainstream industrial languages, such as algebraic data types, pattern matching, and type classes. It is perhaps the first seriously promising competitor to C++ and has already had some major "wins", such as being admitted

into the Linux kernel tree as the first and only language other than C [5]. This makes it a very interesting language from the viewpoint of game and graphics programming as well.

2.1 Sources of programming errors

The author of this work has frequently been frustrated by bugs and confusion related to geometric spaces and transforms in 3D graphics. As a motivating example, the author only recently realized that the rotation matrix routines in the Retrofire project in fact had a sign error. Tests did not help catch this issue because they were also incorrect.

Others have had similar woes. According to Lee [6], "[i]n computer graphics and animation courses, geometric programming with rotations and orientations seems one of the most difficult topics for lecturers to teach and for students to learn.". Geisler et al. [7] assert that it is "easy to confuse different coordinate system representations and to introduce subtle bugs", and also that these bugs are difficult to catch with runtime checks or automatic tests.

Chandra [8] laments that "one of the most frustrating debugging journeys [they] had was related to interpreting vectors in the wrong coordinate systems." In a blog post on a naming convention for matrices, Sylvan [9] poses the rhetorical question: "Ever randomly inverted a matrix or swapped the order of a matrix multiply until things look right?" Apple [10] has published an online article on debugging coordinate space issues. It notes that "[i]ssues related to assumptions or misinterpretations of a coordinate space are difficult to debug, or even identify."

Specific sources of confusion and programming errors identified by this author,

and corroborated by others, include:

No distinction between affine and linear spaces: vectors are used to represent points, even though not all vector operations are meaningful for points and vice versa. For instance, the addition of two vectors is well-defined, but the addition of two points is, in general, not. Furthermore, coordinate transforms act on points and vectors differently.

Coordinate frame transformations: Vertices and their data, such as positions and surface normals, are transformed from one coordinate frame to another several times in a rendering pipeline. It can be easy to lose track of the frame a point or vector is in, and unintentionally apply operations that are not geometrically meaningful. Examples include the subtraction of vectors in different frames, or transforming a point to an unintended frame.

Angular and linear quantities are typically not distinguished at the type level. This includes planar and solid angles as well as polar and spherical coordinate vectors. Additionally, two different units of angle – degrees and radians – are in common use, and unit mix-ups may occur, especially if some functions take angles in degrees and others in radians.

Color spaces and conversions: Colors are objects analogous to geometric vectors, and may be defined in different *color spaces*. Colors in different spaces, and conversions between color spaces, are prone to mix-ups similar to geometric objects and transforms. One common example is the difference between linear and gamma-corrected colors.

Several additional factors conspire to make reasoning about 3D geometry difficult. These include the distinction between right- and left-handed coordinate

systems, column-major and row-major matrix and data layout conventions, and matrix pre- and post-multiplication, as well as the three different interpretations of transforms, as discussed by e.g. DeRose [11]. The root cause of a graphical glitch can be difficult to track down, and some rendering errors are subtle enough that they may go unnoticed unless a known-correct point of comparison is available.

2.2 Research questions

As mentioned before, Rust is a language that emphasizes both performance and correctness through its expressive type system, and appears to be a viable candidate for many use cases that have traditionally called for C or C++. This makes it the language of choice in this thesis. The rest of this work explores some ways to alleviate the aforementioned problems by making common data types more precisely typed, so that more semantically invalid code is detected by the type checker. Specifically, the thesis seeks answers to the following questions:

RQ1. What tools does the type system of the Rust programming language offer for writing correct 3D graphics software?

RQ2. How to employ these type system tools in an API that prevents real-world programming errors without being too rigid or complex for practical use?

RQ3. What, if any, performance implications are there? Correctness improvements that negatively impact performance may be deemed unacceptable by game and graphics programmers.

To help answer RQ1, relevant features of Rust are reviewed in Section 4.3, and a survey of existing work on type systems and geometry representation is given in Section 4.4. A case study of the Retrofire renderer in Chapter 5 attempts to shed light on RQ2, with more detailed analysis on RQ2 and RQ3 following in Chapter 6.

3 3D graphics in brief

The aim of 3D computer graphics is to automatically convert an abstract numeric representation of three-dimensional geometry into a two-dimensional image, a projected "virtual camera view" of the geometry from a given vantage point. This image creation process is called *rendering*, and a program creating 3D images a *renderer*. [12, Ch. 1]

Considerable time and effort has been expended to make rendering as fast as possible and the output as lifelike as possible, within the limits imposed by available technology. An interactive application, such as a video game, typically has to deliver a new frame at least thirty – preferably sixty or more – times per second in order to keep the animation fluid and responsive to user input. At the other end of the scale, rendering all the frames of a feature-length animated film may take months even by a large cluster of high-performance computers. The many use cases for 3D computer graphics include:

- Computer games, simulations, and virtual and augmented reality
- Animated visual media, including TV shows and feature films
- Special effects in visual media added in post-production (computer-generated imagery, or CGI)
- Computer-aided design (CAD) and manufacturing (CAM)

- Illustrations of proposed architectural designs
- Visualization of scientific or medical imaging data such as molecular structures or MRI scans. [13, Ch. 1]

This chapter gives a brief overview of the mathematics of 3D rendering in Section 3.1. The design of an archetypal 3D renderer is presented in Section 3.2.

3.1 Geometry

The goal of 3D graphics is typically to depict solid, discrete objects in a three-dimensional space that obeys the familiar rules of Euclidean geometry. To create desired scenes, these shapes have to be *transformed* – for example, rotated and translated – to their desired positions in the virtual world, and eventually *projected* on a 2D plane for viewing. This section gives a brief review of the mathematics of representing geometry and geometric transformations in 3D graphics. A more in-depth treatise is offered by e.g. Schneider and Eberly [14].

3.1.1 Some conventions

This thesis uses column vectors and pre-multiplication. Inline vectors are notated as tuples: $(1, 2, 3) = (1 \;\; 2 \;\; 3)^\mathsf{T}$. Coordinate systems are right-handed unless otherwise specified, with $+x$ = right, $+y$ = up, and $-z$ = forward. The typographical conventions listed in Table 3.1 are used to denote variables of different sorts.

Table 3.1: Typographical conventions.

Object	Notation	Examples
Vectors	lowercase bold	$\mathbf{v}, \mathbf{u}, \mathbf{0}$
Unit vectors	lowercase bold with a hat	$\hat{\mathbf{n}}$
Scalars	lowercase italic	a, b, c
Points	uppercase roman	$\mathrm{P}_{\mathrm{VIEW}}, \mathrm{Q}, \mathrm{O}$
Sets, fields	uppercase italic or blackboard bold	$F, \mathbb{R}, \mathbb{R}^3$
Spaces, frames	uppercase bold	$\mathbf{S}, \mathbf{T}$
Matrices	uppercase bold	$\mathbf{A}_{\mathbf{S}\to\mathbf{T}}, \mathbf{B}$

3.1.2 Vector spaces

A *vector space* or *linear space* $\mathbf{V}$ over a field F of *scalars* is a set of objects, *vectors*, with two associated operations:

- Addition of two vectors, $(+) : \mathbf{V} \times \mathbf{V} \to \mathbf{V}$
- Multiplication of a vector by a scalar, $(\cdot) : \mathbf{V} \times F \to \mathbf{V}$

as well as an identity element for addition, a *zero vector* $\mathbf{0}$. Negation, subtraction, and division can additionally be defined in the familiar way in terms of $(+)$, $(\cdot)$, and $\mathbf{0}$. Geometrically, a vector can be understood as an "arrow" with a direction and magnitude (or length), but no fixed position or origin point. [15, Ch. 7]

Some very abstract vector spaces exist in mathematics, but computer graphics is primarily concerned with the familiar real spaces $\mathbb{R}^2$ and $\mathbb{R}^3$. However, for reasons discussed in Section 3.1.7, 3D vectors are often actually expressed in 4D *homogeneous coordinates*.

As a rule, a real n-vector $\mathbf{v}$ is represented as a Cartesian coordinate tuple

$$\mathbf{v} = (v_1, v_2, \ldots, v_n),\ v_i \in F,$$

with the field $F = \mathbb{R}$ (usually approximated as floating-point values). The values v_i are called the *(scalar) components* of $\mathbf{v}$. Vector addition and scalar multiplication are simply defined component-wise: [15, Ch. 7]

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1, \ldots, a_n + b_n)$$
$$x\,\mathbf{a} = (x\,a_1, \ldots, x\,a_n).$$

The rest of this work assumes Cartesian coordinate representation unless otherwise stated.

The *dot product* $(\cdot) : \mathbf{V} \times \mathbf{V} \to F$ endows a vector space with a notion of angles and lengths. It has several uses in computer graphics, particularly in lighting calculations. The dot product is defined as follows:[1]

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \|\mathbf{a}\|\,\|\mathbf{b}\| \cos\theta$$

where θ is the angle between the vectors, and $\|\mathbf{v}\|$ is the length of $\mathbf{v}$. The length, or magnitude, of a vector is the square root of the dot product of the vector with itself:

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}\,.$$

This is, of course, simply the n-dimensional generalization of the Pythagorean theorem. [15, Ch. 7]

A *unit vector* is a vector with length 1. Any vector $\mathbf{v}$ except $\mathbf{0}$ can be *normal-*

[1] In a real vector space; not all vector spaces are equipped with a dot product.

ized to a unit vector $\hat{\mathbf{v}}$ by dividing by $\|\mathbf{v}\|$. Unit vectors express pure directions without a length or magnitude component, and are thus useful for representing quantities such as surface normals. In particular, the equation of the dot product of two unit vectors $\hat{\mathbf{v}}$, $\hat{\mathbf{u}}$ simplifies to $\hat{\mathbf{v}} \cdot \hat{\mathbf{u}} = \cos\theta$.

A *basis* of a linear n-space is a set of n linearly independent vectors. The *standard basis* is the set of orthogonal unit vectors $\hat{\mathbf{e}}_1, \ldots \hat{\mathbf{e}}_n$:

$$\begin{aligned}
\hat{\mathbf{e}}_1 &= (1, 0, \ldots, 0) \\
\hat{\mathbf{e}}_2 &= (0, 1, \ldots, 0) \\
&\vdots \\
\hat{\mathbf{e}}_n &= (0, 0, \ldots, 1)
\end{aligned}$$

Given a basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, any vector $\mathbf{v}$ in the space can be uniquely specified as a linear combination of the basis vectors:

$$\mathbf{v} = (v_1, v_2, \ldots, v_n) = v_1\mathbf{b}_1 + v_2\mathbf{b}_2 + \cdots + v_n\mathbf{b}_n.$$

3.1.3 Linear transforms

A function $T : \mathbf{V} \to \mathbf{U}$, where $\mathbf{V}$ and $\mathbf{U}$ are vector spaces (possibly $\mathbf{V} = \mathbf{U}$), is called a *linear map* or *linear transform*[2] if and only if it has the property of *linearity*:

$$\begin{aligned}
T(\mathbf{v}) + T(\mathbf{u}) &= T(\mathbf{v} + \mathbf{u}) \\
a\,T(\mathbf{v}) &= T(a\,\mathbf{v})
\end{aligned}$$

[2]The term "transformation" is commonly shortened to "transform" in computer graphics.

Equivalently, linear transforms preserve linear combinations. A specific consequence of linearity is that a linear map must preserve the zero vector: $T(\mathbf{0}) = \mathbf{0}$. [15, Ch. 9]

Linear maps can be represented as *matrices*, 2D arrays of numbers. In particular, a *square matrix* of dimension n is an $n \times n$ array of scalars:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$$

An n-vector $(a_{i,1}, \ldots, a_{i,n})$ is called the ith *row vector* of $\mathbf{A}$, and an n-vector $(a_{1,j}, \ldots, a_{n,j})$ is called the jth *column vector* of $\mathbf{A}$. [15, Ch. 9]

Geometric transforms that are linear, and thus expressible as matrices, include rotations, scalings, reflections, and shears. The encoding of these transforms as matrices is outside the scope of this work; detailed derivations are given e.g. by Schneider and Eberly [14, Ch. 4].

Matrix multiplication is the (function) composition of two linear maps. It is associative but not commutative. It can be defined in terms of the dot product:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{AB} = \begin{pmatrix} \mathbf{a}_{1,} \cdot \mathbf{b}_{,1} & \ldots & \mathbf{a}_{1,} \cdot \mathbf{b}_{,n} \\ \vdots & \ddots & \vdots \\ \mathbf{a}_{n,} \cdot \mathbf{b}_{,1} & \ldots & \mathbf{a}_{n,} \cdot \mathbf{b}_{,n} \end{pmatrix}$$

where $\mathbf{a}_{i,}$ is the ith row vector of $\mathbf{A}$ and $\mathbf{b}_{,j}$ the jth *column* vector of $\mathbf{B}$. Multiplying a matrix with itself has the natural result of applying the same linear transformation twice, and repeated multiplication can be written with the expo-

nent notation: [15, Ch. 8]

$$\mathbf{A}^n = \underbrace{\mathbf{A}\mathbf{A}\cdots\mathbf{A}}_{n \text{ times}}$$

Matrix–vector multiplication applies a linear transform to a vector. It is defined

$$\mathbf{A}\mathbf{v} = \begin{pmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_n \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{v} \\ \vdots \\ \mathbf{a}_n \cdot \mathbf{v} \end{pmatrix}$$

where a_i are the row vectors of $\mathbf{A}$. This is a special case of matrix multiplication, where the vector is identified with a matrix of size $n \times 1$. [15, Ch. 9]

Because matrix multiplication is associative, multiplying a vector first by one matrix, then the result by another, is equivalent to multiplying the vector by the product of the matrices:

$$\mathbf{B}(\mathbf{A}\mathbf{v}) = (\mathbf{B}\mathbf{A})\mathbf{v}$$

This is a crucial property, as it means that any number of linear transforms can be precomposed into a single matrix and efficiently applied to any number of vectors. [15, Ch. 9]

The *inverse* of a square matrix, if it exists, applies the transformation in reverse:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I},$$

where $\mathbf{I}$ is the *identity matrix*. A matrix is invertible if and only if it preserves the dimension of its domain; for example, a matrix that projects 3D vectors onto a 2D plane clearly has no inverse. For the inverse of the product of two

matrices, the following identity holds:

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

Intuitively, to revert a composite transform, one must "undo" the constituent transforms by applying their inverses in *reverse order*. [15, Ch. 9]

3.1.4 Affine spaces

Vectors and vector spaces are familiar to most graphics programmers, but the concept of affine spaces – spaces of points – is rarely discussed by programming tutorials or even academic textbooks[3]. This is even though the most familiar of geometric spaces, the Euclidean space, is affine.

Some authors do enunciate the distinction. For instance, Schneider and Eberly give a fairly in-depth treatment of affine geometry in computer graphics in [14, Ch. 3–4]. Further discussion is provided by Goldman [16].

An *affine space* $\mathbf{A}$ is a set of elements, called *points*, with an associated vector space $\mathbf{T_A}$ of *translation vectors*, or simply translations, and the following operations:

- Translation of a point by a vector, $(+) : \mathbf{A} \times \mathbf{T_A} \to \mathbf{A}$
- Difference between two points, $(-) : \mathbf{A} \times \mathbf{A} \to \mathbf{T_A}$.

Points, unlike vectors, can thus be neither added together nor multiplied by a scalar. This matches the normal intuition of what a point is. In general, points can be seen as representing a "state" or "position", and vectors as a "change" or "displacement", of some sort. [14, Ch. 3]

[3] For example, Vince [15] uses the word "affine" only once in his 500-page textbook "Mathematics for Computer Graphics"!

Notably, every vector space $\mathbf{V}$ is an affine space where $\mathbf{T}_{\mathbf{V}} = \mathbf{V}$, with the translation and difference operations reducing to vector–vector addition and subtraction. An affine space can be understood as a vector space without a privileged zero vector, and many of the concepts generalize.

An *affine basis*, or *(coordinate) frame*, defines a coordinate system in an affine space. It is defined by a freely chosen *origin* point and a linear n-basis (or equivalently, by $n + 1$ points, not all of which lie on the same n-hyperplane). [14, Ch. 3]

Given an origin O, each point P in an affine space can be identified with the translation vector $\overline{\mathrm{OP}} = \mathrm{P} - \mathrm{O}$, making the space isomorphic with its associated vector space. This crucial property enables the common practice of representing points as vectors in programming, with the natural choice of *canonical* origin $\mathrm{O} \equiv \mathbf{0} = (0, \ldots, 0)$.

An *affine combination* is a weighted average of points P_i:

$$a_1\mathrm{P}_1 + a_2\mathrm{P}_2 + \cdots + a_n\mathrm{P}_n\,, \text{ such that } \sum a_i = 1.$$

This notation is intentionally somewhat sloppy in that it adds and multiplies points as if they were vectors. It can be made rigorous by rearranging and rewriting it in terms of the translation vectors $\overline{\mathrm{OP}_i}$, for some arbitrary origin point O (for example, P_1). [14, Ch. 3]

Goldman [17] has shown that in general, treating points as vectors in an expression is "legitimate" when either $\sum a_i = 1$, in which case the result is a point, or $\sum a_i = 0$, in which case it is a vector. Thus, for instance, $1.5\mathrm{P} - 0.5\mathrm{Q}$ and $2\mathrm{P} + \mathrm{Q} - 3\mathrm{R}$ are well-defined, while $2\mathrm{P} - 3\mathrm{Q}$ is not.

Linear interpolation ("lerping") between two points is an important special case of affine combination:

$$\text{lerp}(\mathrm{P}, \mathrm{Q}, t) \;=\; (1-t)\,\mathrm{P} + t\,\mathrm{Q} \;=\; \mathrm{P} + t\,(\mathrm{Q}-\mathrm{P}) \;=\; \mathrm{P} + t\,\overline{\mathrm{PQ}}\,,$$

where typically $t \in [0,1]$ [15, Ch. 10]. The definition above exhibits both the "sloppy" form and the more rigorous one, obtained with simple algebra. The lerp function is an indispensable and ubiquitous tool in computer graphics.

3.1.5 Affine transforms

As mentioned in Section 3.1.3, linear transforms can represent rotation, reflection, scaling, and shearing of vectors. However, they cannot represent translation, which does not preserve linearity (in particular, it clearly does not preserve the zero vector). Translation is meaningless for vectors, which do not have a defined position, but a crucial operation for points. [15, Ch. 9]

An *affine transform* (or affine map) is a mapping from an affine space to an affine space that preserves *affinity*, a condition weaker than linearity. Equivalently, a mapping is affine if it preserves affine combinations. [14, Ch. 3]

An affine transform between affine n-spaces can be expressed as a square matrix of dimension $n+1$, composed of an $n \times n$ linear submatrix $\mathbf{L}$ and a translation vector $\mathbf{t}$:

$$\begin{pmatrix} \mathbf{L} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix}$$

where the bottom row is always $(0, \ldots, 0, 1)$ and does not need to be explicitly stored. However, it is needed in perspective projection (see Sec. 3.1.8), so for simplicity it is typically included. [14, Ch. 4]

Affine transforms compose like linear transforms. In fact, an affine map in dimension n can be understood as a linear map in dimension $n + 1$, and the translation of a n-point $(p_1, \ldots, p_n)$ corresponds to a *shearing* transform of the $(n + 1)$-vector $(p_1, \ldots, p_n, 1)$, as discussed in section 3.1.7.

3.1.6 Euclidean spaces

A *Euclidean n-space* is an affine space that models the physical space (when $n = 3$) or its higher- or lower-dimensional analogues. A Euclidean 2-space is a plane, and a Euclidean 1-space is a line. All Euclidean spaces of a given dimension are isomorphic, so it is common to refer to *the* Euclidean n-space $\mathbb{E}^n$. [14, Ch. 3]

A point in a Euclidean space is identified by its Cartesian coordinates, unique up to a choice of frame. The standard frame is the tuple $(\hat{\mathbf{e}}_1, \ldots, \hat{\mathbf{e}}_n, \mathrm{O})$, where the unit vectors $\hat{\mathbf{e}}_i$ are the standard linear n-basis vectors and the origin $\mathrm{O} = (0, \ldots, 0)$. Thus, a Euclidean space has a "canonical" origin, but it is not privileged the way the zero vector of a linear space is.

The standard dot product gives Euclidean spaces the familiar notions of angle, length, and distance. The distance between two points is simply the length of the vector between the points:

$$\mathrm{dist}(\mathrm{P}, \mathrm{Q}) = \|\mathrm{Q} - \mathrm{P}\| = \|\overline{\mathrm{PQ}}\|.$$

Excluding special applications, such as modeling relativistic motion, the Euclidean 3-space is the model of physical space in 3D graphics. Planes and lines, as subspaces of $\mathbb{E}^3$, are also heavily used in graphics programming.

3.1.7 Homogeneous coordinates

It would be useful to transform both points and direction vectors with the same matrix. However, as mentioned before, points should be affected by translation, but vectors should not. This distinction can be expressed with *homogeneous coordinates*, originally introduced by Möbius. The utility of homogeneous coordinates in computer graphics was realized early on (e.g. Roberts [18]).

In homogeneous form, an n-vector $\mathbf{v}$ is encoded as (n+1)-vector $(v_1, \ldots, v_n, 0)$ and an n-point P as $(p_1, \ldots, p_n, 1)$. For 3-vectors and -points the extra component is usually called w. It is easy to see that vectors in this form are not affected by the translation component, but points are: [15, Ch. 9]

$$\begin{pmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ 1 \end{pmatrix}, \text{ but } \begin{pmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}.$$

It can be seen that the "legitimate" expressions of Section 3.1.4 can also be defined as exactly those expressions that result in either w=1 or w=0 in homogenenous representation.

3.1.8 Perspective projection

The third type of space used in 3D graphics – perhaps the least well understood by most practitioners – is the *projective space*. It is used to implement perspective projection and is built upon the basic rule of perspective that parallel lines appear to converge and to meet at a "point at infinity", called the *vanishing point* of those lines. [19]

A projective space can be understood as taking a Euclidean space and adding

the points at infinity (topologically, the *closure* of the space) as normal points, then identifying each pair of "opposite" points at infinity, such that any family of parallel lines intersect at exactly one such point. Points in a projective space can be expressed in homogeneous coordinates, and the projective transform as a matrix, which is convenient given that this representation was already used to implement affine transforms. [19]

Normally, perspective projection is only applied to points. This means that separate matrices are needed for points and vectors, such as normals or tangent vectors, after all: one matrix that includes the projection transform and other that does not. This duplication can be one source of confusion or programming errors. The topology of the projective space is rather interesting and unintuitive. It is discussed in more depth by e.g. Riesenfeld [20] and Goldman [16].

3.2 Rendering

For the purposes of this work, *rendering* is defined as any process of automatically creating two-dimensional images based on some programmatic input. One of the first mechanical devices capable of creating visual patterns based on predefined instructions was the *Jacquard loom*, patented by Joseph Maria Jacquard in 1804. The input was provided by means of a chain of punched cards. [21]

Early computer terminals and hardcopy printers were only able to recreate a fixed set of character glyphs. These could be used to produce crude graphics, similar to earlier *typewriter art*. More free-form computer graphics were first created using a *pen plotter*, a device that moves a physical pen over a paper based on programmed instructions. [22]

Vector monitors, developed from oscilloscopes, are the real-time equivalent of pen plotters. They were the first display technology capable of rendering arbitrary lines and polygons in real time. Ultimately they were replaced by raster displays based on a grid of *pixels*, capable of representing much richer, multi-color graphics at the expense of "jaggedness" caused by aliasing. [22]

One of the first video games employing real-time graphics was *Spacewar*, developed by Steve Russell and others for the PDP-1 minicomputer in 1962. Arcade video games, pioneered by *Pong* in 1972, quickly became extremely popular. *Battlezone* was the first arcade game featuring real-time 3D vector graphics. In the 1980s, home computers and video game consoles brought interactive computer graphics to the living room. [23].

Live action films pioneering the use of computer-generated imagery (CGI) composed with real scenes and actors include *Tron* (1982) and *Terminator 2* (1991). *Toy Story*, released in 1995, was the first fully computer-generated full-length animation film, and the first near-photorealistic, fully computer-animated feature film was *Final Fantasy: Spirits Within* in 2001.

The first consumer 3D accelerator cards (later branded "GPUs", or Graphics Processing Units) were introduced in 1995, opening the door for much richer, more complex 3D scenes in computer games than was possible with CPU-based rendering. They achieve this by exploiting the inherent parallelism in drawing a large number of independent pixels. [12, Ch. 3]

3.2.1 Geometry representation

By far the most common way to model three-dimensional geometry on a computer is a *polygon mesh*: a list of planar polygons, defined by their vertex points,

that share edges such that they form a continuous surface [12, Ch. 2]. Most commonly the polygons are triangles, or they are subdivided into triangles before rendering; the versatility of the triangle representation was noted early on by Wylie et al. [24]

A polygon mesh can reproduce curved surfaces only approximately, and a large number of faces may have to be used to closely replicate a smooth curve. The primary advantage of meshes is that rendering triangles is very straightforward and efficient compared to curved surfaces such as Bézier patches. Such surfaces can be rendered by first *tessellating* them. [12, Ch. 17]

Listing 3.1: Basic mesh and vertex data structures.

```
struct TriangleMesh {
    vertices: Vec<Vertex>,
    faces: Vec<[usize; 3]>,
}
struct Vertex {
    position: Point3,
    normal: Normal3,
    texcoord: TexCoord,
}
```

Because each vertex is shared by at least two faces, it makes sense to store the geometry as two lists, as shown in Listing 3.1: one that contains all the unique vertices of the mesh and another containing tuples of valid indices into the vertex list, each tuple representing a face. The vertex data type is a tuple of *attributes*, comprising the position of the vertex in space and optional associated data such as color, surface normal, or texture coordinates. [12, Ch. 16]

3.2.2 The rendering pipeline

A typical 3D renderer architecture can be thought of as a pipeline, where geometry data flows from one *pipeline stage* to another on its way to the screen or other output medium. The data-flow diagram is a simple directed acyclic graph, where one stage only depends on the preceding stages. A simplified rendering pipeline is depicted in Figure 3.1. Modern GPUs have several additional programmable stages, such as *geometry* and *tessellation* shaders, but they are beyond the scope of this work. [12, Ch. 2–3]

3.2.3 Transformations

Geometry submitted for rendering undergoes several transformations as it is passed through the pipeline. In practice, some of these steps are combined into one. Each of the transformations maps the geometry to a different coordinate frame, or space[4], as illustrated by Figure 3.2. The common, named frames are described in more detail below. [12, Ch. 2]

The "native" frame of a model is called the *model space* (also *object* or *local* space). It is the space in which the model was created, typically by a 3D artist using a 3D modeler application. The geometry is normally positioned so that its natural rotation or symmetry axes are aligned with the coordinate axes. The *world* or *global space* is where different models, terrain, cameras, lights, and other objects are positioned and oriented relative to each other in order to compose the scene to be rendered. [12, Ch. 2]

It makes intuitive sense to treat the virtual camera as an object in the scene, transformed in the world space like other entities. However, for rendering purposes it is more convenient to instead transform everything else from the world

[4]Some of these "spaces" are better understood as different frames in the same space.

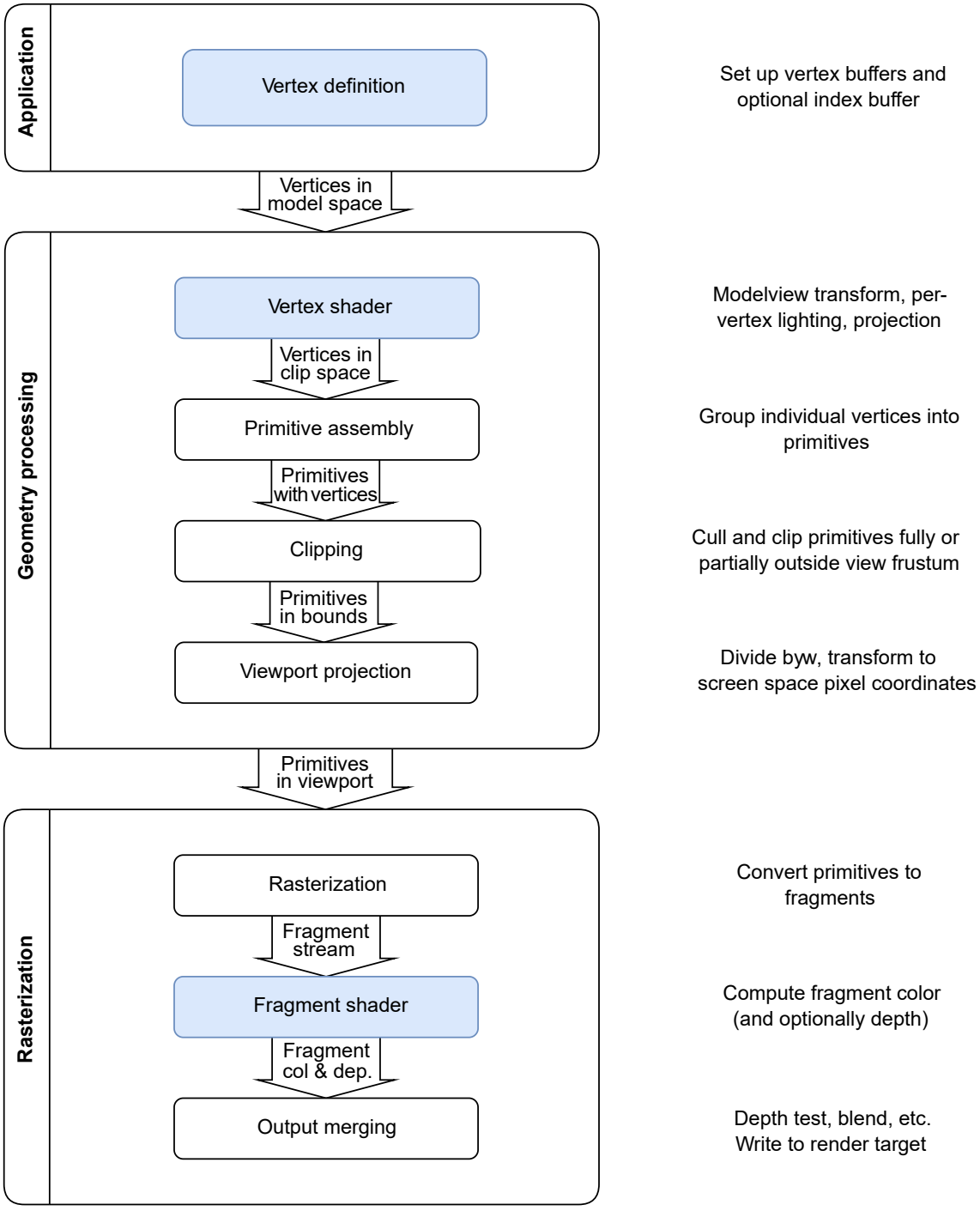

Figure 3.1: Pipeline overview, with programmable stages highlighted.

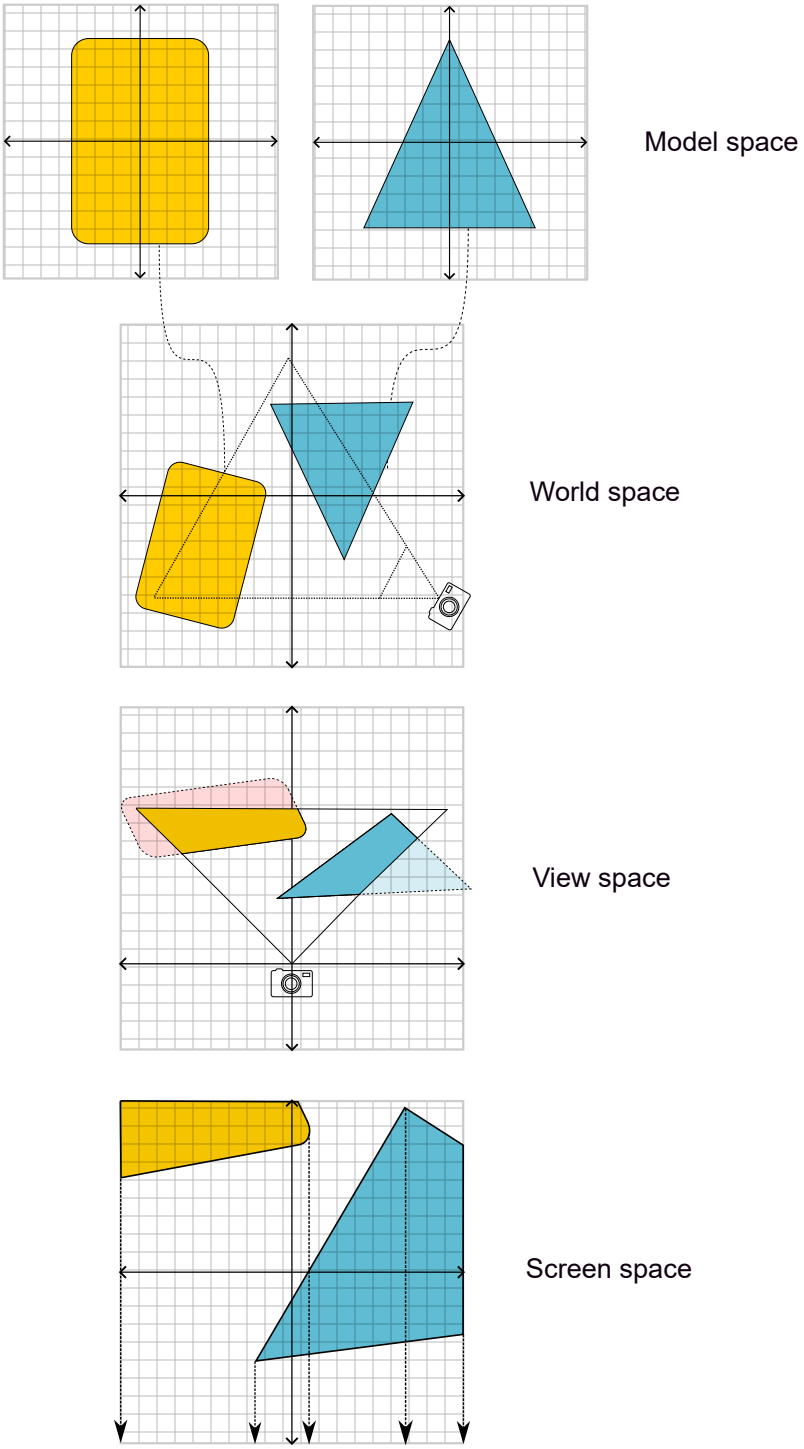

Figure 3.2: Spaces and transforms.

space to the camera-centric *view space* (also *eye* or *camera* space) [12, Ch. 2]. The inverse nature of this transform is one potential source of confusion and programming errors when writing camera routines and composing transforms.

A *projection* transform maps points from the view space to homogeneous coordinates in the *clip space*. The clip space models the the *view frustum*, a truncated pyramid that represents the volume of space that a virtual camera can see. The sides of the pyramid correspond to the edges of the viewport, the *near plane* is used to cull geometry behind the camera, and the *far plane* can be used to limit the amount of geometry that has to be rendered. [12, Ch. 2]

The mapping to the clip space is such that each side of the view frustum is mapped to one of the six clip planes $x = \pm w$, $y = \pm w$, $z = \pm w$.[5] This makes discarding vertices that are outside the frustum a simple matter of inequality checking. Primitives partially inside the view frustum have to be *clipped* against one or more clip planes. The Sutherland–Hodgman algorithm is commonly used. [25], [26]

After clipping, vertex coordinates are normalized by *perspective division* from (x, y, z, w) to $(x/w, y/w, z/w, 1)$. This nonlinear step is what creates the actual perspective-projected representation.[6] The normalized coordinate system is referred to as *normalized device coordinates*, or the NDC space for short. A *viewport* transform maps points from the NDC space to pixel coordinates in *screen space*, either filling the entire output buffer or a subregion of it (a viewport). This is the last transformation step before rasterization. [12, Ch. 2]

[5] Some renderers place the near plane at $z = 0$ instead.

[6] Some learning resources give the impression that the entire transform chain from model space to pixels can be represented with just matrix multiplications. This may confuse a learner – how can an affine transform handle the perspective division? The answer is, of course, that it cannot.

3.2.4 Shading

Computing the color of a point on the surface of an object, based on some approximation of the behavior of light in the physical world, is called *shading*. Shading algorithms have evolved from simple flat lighting in early renderers to highly detailed simulations of light and various surface properties.

The idea of a separate domain-specific language for writing shading routines, or *shaders*, was introduced in 1987 with the Reyes renderer developed by Pixar [27] and described in more detail in 1990 by Hanrahan et al. [28] A custom-written software renderer can support arbitrarily complex shading calculations. However, 3D programming interfaces such as OpenGL and Direct3D originally only supported a *fixed function* pipeline that was configurable to an extent but not programmable [12, Ch. 2].

The first 3D accelerator cards available to consumers, such as the 3Dfx Voodoo, only implemented the projection, clipping, and rasterization stages of the rendering pipeline, and were limited to interpolating colors and texture coordinates. As technology advanced, more pipeline stages were moved to the GPU, starting with vertex transforms. Support for programmable shaders was first implemented in consumer video cards in the early 2000s. At first, shaders were written in assembly-like languages, but higher-level C-like shading languages, such as HLSL and GLSL, were soon developed to help write more and more complex shader programs. [12, Ch. 3]

Much has been written on the topic of surface lighting calculations; for example Akenine-Möller et al. [12, Ch. 5, 6, and 9] discuss the topic in some detail. For the purposes of this thesis, shading is relevant because it is a function of three unit vectors: the direction of the light source $\hat{\mathbf{l}}$, the direction of the camera

viewpoint $\hat{\mathbf{v}}$, and the surface normal vector $\hat{\mathbf{n}}$. Typical shading equations are of the form

$$\mathrm{C}_{\text{shaded}} = f_{\text{unlit}}(\hat{\mathbf{n}}, \hat{\mathbf{v}}) + \sum_{i=1}^{n} \max(\hat{\mathbf{l}}_i \cdot \hat{\mathbf{n}}, 0)\ \mathrm{C}_i\, f_{\text{lit}}(\hat{\mathbf{l}}_i, \hat{\mathbf{n}}, \hat{\mathbf{v}}),$$

where $\hat{\mathbf{l}}_i$ and C_i are the direction and color of the ith light source, f_{unlit} gives the color of points not directly lit, and f_{lit} is the amount of light reflected from the light towards the camera. This is a discrete approximation of the full *reflectance equation* which integrates over all incident light in a hemisphere [12, Ch. 9].

The three vectors $\hat{\mathbf{l}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{n}}$ may be expressed in any basis, as long as it is the same for all three. Depending on the shading algorithm used, it can be convenient to work in either model, world, or view space. A fourth possibility is the *tangent space*, given by the Frenet–Serret, or TBN, basis. It comprises the surface normal vector at the point being shaded and two orthogonal vectors tangent to the surface, typically called the *tangent* and *bitangent* (or sometimes, inaccurately, *binormal* [7]). [12, Ch. 5–6]

3.2.5 Colors and color spaces

In computer graphics, colors are a data type very similar to vectors and points, but with their own intricacies. Typically, the representation of a color in memory is a tuple of components, called *(color) channels*, and in analogy to point and vectors, colors are elements of a *color space*, some subset of all perceptible colors.

The electromagnetic spectrum is a continuum. However, human color vision is based on three types of light-sensitive molecules, each with a sensitivity peak

[7] Note that the term "bitangent" can itself be confusing, as it has another, unrelated meaning as a line that is tangent to a curve at two distinct points.

at a different wavelength. Due to this fact, a very large number of colors can be expressed as points in a merely three-dimensional color space – a *tristimulus* value. Despite the apparent simplicity of the three-dimensional model, colors and color perception are a deep and complex topic. Hughes et al. caution that "[m]ost of what a majority of people 'know' about color is false, or at the very least, it is true only under very restrictive conditions of which they are unaware." [13, Ch. 28]

The gold standard of color models is the XYZ model, derived experimentally by the International Commission on Illumination (CIE) in 1931 [29]. More useful in most applications, however, is the standard RGB (sRGB) model based on red, green, and blue components, related to XYZ through a linear transform. When compositing images over background content, or rendering transparent objects in a 3D scene, a fourth channel, "A" or "alpha", can be included to represent opacity [13, Ch. 28].

The response of the human visual system to light intensity is nonlinear, approximated by the power law $I^{1/3}$. An important aspect of color rendering is *gamma correction* and the distinction between linear and gamma-encoded color coordinates, as linear operations like lerp yield incorrect results if performed in a nonlinear color space.[8] [12, Ch. 5]

Compared to geometric vectors and points, colors are different in that negative color values are unphysical. There is also a soft upper limit for color values – even though intermediate computations can work with arbitrary intensities, the result must eventually be reduced to the range that the video hardware expects and can physically reproduce. This reduction is usually called *tone mapping*. [12, Ch. 8]

[8] Historically, many applications, even professional ones, have gotten this wrong.

A technique called *premultiplied alpha* is often employed in real-time graphics. A typical way to render transparent surfaces is to lerp between the foreground and background colors using the equation

$$\mathrm{C}_{final} = \alpha \mathrm{C}_{fore} + (1 - \alpha)\mathrm{C}_{back}\,,$$

where α is the foreground alpha value. The first multiplication can be avoided by precalculating, "baking", it into textures with transparency. [30]

HLS (hue, lightness, saturation) and *HSV* (hue, saturation, value) are two closely related representations of the sRGB space in *cylindrical* coordinates. Operations such as adjusting the hue or saturation of a color are much simpler in HLS or HSV than in RGB coordinates. *Perceptual* color models, such as CIELAB and OKHCL, attempt to compensate for the fact that human eyes are the most sensitive to green and by far the least sensitive to blue light. Consequently, the pure RGB blue $(0, 0, 1)$ appears *much* less bright than the pure green $(0, 1, 0)$. [12, Ch. 8]

Finally, the standard sRGB space has a limited *gamut* – the subset of colors that it can represent out of all perceptible colors. Color spaces like Adobe RGB, Display P3, and ProPhoto RGB are examples of wide-gamut color spaces supported by some displays. *Color management* is concerned with trying to ensure color consistency and accuracy across media with different gamuts. [12, Ch. 8]

A program may need to juggle several of these color spaces and coordinate schemes. It seems evident that the same danger of confusion and errors is present as with geometric objects, spaces, and transforms.

3.2.6 Rasterization

The process of turning geometry data in the screen space into a discrete approximation is called *rasterization*. It involves determining exactly which pixels are *covered* by a geometric primitive, and then computing a color for each covered pixel.

Generally, attributes relevant to determining a pixel color are only stored for each vertex. They have to be interpolated in order to find the values at every individual pixel, or more properly *fragment* (in *multisampling*, used to alleviate the "jaggedness" of edges caused by aliasing, several fragments are sampled per output pixel). [12, Ch. 2]

The classic triangle rasterization algorithm walks along the edges of a triangle from the top to the bottom vertex, turning the triangle into a stack of horizontal *scanlines*. The scanlines are then filled by walking each scanline from left to right. This algorithm was likely first described by Wylie et al. [24]. It is easy to generalize to handle any convex polygon, but filling concave or self-intersecting polygons is somewhat trickier.

GPUs and many modern software renderers instead use an algorithm that employs *barycentric coordinates* (an affine combination of triangle vertex points) and tests whether every pixel in the bounding rectangle of a triangle is covered or not (a point-in-polygon test). The algorithm, introduced by Pineda [31], does more work than the scanline-based one, but is much easier to parallelize. This property is essentially what enables the massive parallelism of GPUs.

The final pipeline stage, *output merging*, composes each computed fragment color with the current color at the respective screen coordinate, performing operations such as blending transparent fragments with the background color

and depth testing to achieve proper occlusion of background objects. Some less common, optional operations, such as scissor and stencil testing, can also be performed at this stage.

4 Type systems

A *type system* is a logical system that associates a label, called a *type*, with every program phrase, such as a declaration or an expression, and defines a set of *typing rules* that constrain how types interact. A *type checker* is a tool, often integrated to a compiler, that accepts program source as input (usually pre-processed to an intermediate representation), and determines whether there exists a unique, globally valid assignment of types for the entire program. [32]

If a program is valid according to the type checker, it follows that the semantic rules provably hold for *any execution* of the program. This is a strong and clearly valuable property that excludes a class of semantic errors from ever occurring at runtime. However, programming languages vary considerably in how strong guarantees their type systems are able to express – how "rich" or "expressive" the type system is said to be. [1, Ch. 1]

Type systems occupy a middle ground between fully dynamic, run-time checking on one side and formal verification methods on the other. In practice, there are always properties that cannot be guaranteed by a static type checker and must be checked at runtime. Certain research languages, such as Idris [33], incorporate *dependent types* – types parameterized by[1] runtime values. Con-

[1] More properly, "indexed on"

versely, type checkers must necessarily be conservative: they can statically prove the absence of some erroneous runtime behaviors but cannot prove their *presence*, and thus must reject some programs that are actually "well-behaved". Every type system must thus make compromises between conservativeness and expressivity. [1, Ch. 1]

4.1 A brief history of types

Cardelli and Wegner give an overview of the evolution of type systems in [34]. Some developments and features relevant to this work are summarized below.

FORTRAN made a distinction between integer and floating-point number formats already in the 1950s. Integer arithmetic was more efficient and integers were more natural for use as indices or counters. The language distinguished between the two types of variables based on the first character of their names.

ALGOL 60 required explicitly declaring every variable as either integer, real, boolean, or string. It was the first major language to have an explicit notion of types and type checking. Its successor, ALGOL 68, included user-defined record types and *tagged unions*.

In 1967, Strachey [35] popularized several terms and concepts commonly used today, including the distinction between *ad-hoc* and *parametric polymorphism*. The former is also known as *overloading* – having multiple versions of an operation with different parameter types. For instance, most languages use the same set of arithmetic operators for all numeric types. The latter refers to the ability of parameterize functions and types with *type variables*.

Milner [36] reported in 1978 on a type system developed for the ML language.

It expanded upon earlier work by Hindley [37] and thus became known as the Hindley–Milner type system. ML included several significant, novel type system features, such as type inference, parametric polymorphism, algebraic data types, and pattern matching.

The first widespread language to use modularization for abstraction in a major way was Modula-2, developed by Wirth. It popularized the concepts of encapsulation, visibility control, and separating interfaces from implementations.

Wadler and Blott [38] introduced the notion of *type classes* in 1989 as a way to bridge ad-hoc and parametric polymorphism, allowing a more structured approach to overloading. A type class declares an abstract interface, to which type parameters can then be constrained. The concept was first adopted by the designers of the functional programming language Haskell.

Interfaces and abstract classes in object-oriented languages such as Java are similar to type classes, but there are some important differences. First, type classes are not themselves types, and there is no subtype relationship between a type class and its implementing types. Second, a type class instance can be written for a type *retroactively*, to make an existing type conform to a new interface. Extensions to the original idea include parametric type classes [39] and multi-parameter type classes [40].

Linear and *affine*[2] type systems stem from research on *substructural logics*. A value of a linear type must be "used" exactly once, and a value of an affine type must be used *at most* once [41]. These properties are particularly valuable for managing resources that must be eventually released, such as locks, file handles, and heap-allocated memory.

[2]This use of the two terms is only very vaguely related to the geometric meaning used elsewhere in this work.

4.2 Benefits of typing

Pierce [1] identifies the following strengths of type systems:

Error detection: Errors that are detected early can be fixed immediately. A type error usually also points directly to the problem; a runtime error might only become visible some time after the error occurred, possibly in a seemingly unrelated part of the code.

Type checking can be tremendously helpful in software maintenance and evolution. Wide-reaching changes to a code base can often be made with some assurance that after all the type errors reported by the compiler are fixed, the resulting program is correct. Cardelli asserts that "[t]ypes are essential for the ordered evolution of large software systems" [42].

Abstraction: Types are used to model the world and to express higher-level concepts. Rich type systems allow more expressive power and more precise models of domain rules and objects. Type systems usually include means of separating an interface from its implementation, facilitating the evolution of the latter without breaking code that uses the former. Furthermore, different types can be treated in a uniform manner if they share a common interface.

Documentation: Explicit typing is a form of documentation that aids reading and understanding code. A typed module interface makes it clear what kinds of values it expects and returns. This form of documentation cannot become outdated because it is checked every time it is compiled.

Several authors, including Martin [43], have noted that code is read many more times than it is written. Thus, investing time and effort to make

code easier to understand is likely to pay itself back in the long run.

Language safety: A *safe language* can guarantee the integrity of its own abstractions and those introduced by the programmer. Any high-level language designed to be run on a real computer must base its abstractions on the underlying untyped, unsafe machinery. A safe language is one that can make sure that those abstractions do not "leak" – for instance, that writing past the end of an array, corrupting another variable adjacent to the array in memory, cannot happen.

Efficiency: Type information was first introduced in programming languages such as FORTRAN in order to improve runtime efficiency: operations on typed values can be translated directly to their equivalent machine instructions. Many runtime checks and branches can be eliminated if the corresponding properties are statically known and proven. The optimizer and code generator stages of modern compilers rely heavily on information collected during type checking.

Pierce [1] points out that "[p]rogrammers working in richly typed languages often remark that their programs tend to 'just work' once they pass the typechecker, much more often than they feel they have a right to expect." This experience, shared by this author as well, is often expressed in the concise form "if it compiles, it works".

4.3 The Rust programming language

Rust is a general-purpose, statically typed, compiled programming language with a focus on performance, reliability, and productivity [4]. It was originally designed by Graydon Hoare as a personal project when he worked at Mozilla. The language became an official Mozilla-sponsored project in 2009, and after quite a bit of evolution, version 1.0 was released in 2015 [44]. Since then, new minor versions have been released at a steady six-week cadence, with the most recent release as of March 2026 being 1.94. Rust has strong backwards compatibility guarantees, and new minor releases are generally not allowed to break existing code.

Rust as a language, and its community and ecosystem, emphasizes the use of the type system to encode invariants and "making invalid states unrepresentable", helping ensure correctness and memory safety. This section gives a brief introduction to some high-profile features of Rust and its type system pertinent to the Retrofire API introduced in Chapter 5. A much more comprehensive treatment can be found, for example, in *The Rust Programming Language* by Klabnik et al. [45]

4.3.1 Memory safety

Memory safety refers to the guaranteed absence of program errors related to accessing memory. These include dereferencing null or dangling pointers, trying to free already freed memory, out-of-bounds array access (buffer overflow), and unsynchronized concurrent access (data races) [46]. A *memory model* is a formal description of the interaction of multiple threads of execution as they access shared memory. It is not possible to implement thread-based concurrency soundly without certain language-level guarantees, as shown by Boehm [47].

The memory model of Rust is essentially that of C and C++, as implemented by the LLVM compiler backend.

C and C++ are not memory-safe. This has led to several prominent security vulnerabilities (CVEs) where a programming error has given an attacker the ability to inject and execute arbitrary code over the network. Several studies have found that approximately 70% of vulnerabilities in large software systems are caused by memory safety issues [48].

Traditional memory-safe languages, such as Java, C♯, and Python, achieve safety by having a *runtime environment* that manages memory, including automatic garbage collection. This has a performance cost, both in time and space, and the use of memory-managed languages is uncommon particularly in real-time or "soft" real-time applications as well as in embedded programming with tight resource constraints.[3] [49], [50]

Rust attains memory safety via affine types and *borrow checking*, in addition to run-time checks for conditions that cannot be statically ensured, such as out-of-bounds array indexing. Like in C++, types can have destructors that deterministically handle cleanup when a value goes out of scope. The highly popular *RAII* idiom (Resource Acquisition Is Initialization) prevents resource leaks by taking advantage of the guaranteed cleanup [51].

Affine types have *move semantics:* a value of an affine type cannot be copied, only moved. The compiler prevents the use of a moved-from variable until and unless another value is assigned to it.[4] This semantics is particularly well-suited for types that hold resources that require eventual release, such as mem-

[3]To be fair, though, there is a version of Java that runs on smart cards of all things.

[4]This differs from C++ move semantics, where the value must remain in a "valid but unspecified" state.

ory allocations, file handles, or mutex locks. Primitive types in Rust are not affine; they can be freely and cheaply copied. User-defined types can opt-in to copyability if they are only composed of other copyable types.

In addition to moving and copying, values can be *borrowed*: passed to or returned from functions by either mutable or immutable reference. An object can have either any number of immutable or *shared* references or exactly one mutable or *exclusive* reference to it at a time. This constraint is often summarized as "shared XOR mutable"; the semantics are similar to a *read-write lock*, but the constraint is also enforced in single-threaded use and checked at compile time with zero runtime overhead. [45, Ch. 4]

Perhaps the most famous aspect of Rust's borrowing rules is reified *lifetimes*: references are statically prevented from existing beyond the scope of their referent by making object lifetimes part of the type system. Dangling references are thus impossible, as are references to uninitialized objects. [45, Ch. 10]

Static checking in Rust is necessarily conservative: it accepts no memory-unsafe programs but rejects some memory-*safe* programs. Additionally, interfacing with anything outside Rust is fundamentally memory unsafe, because there can be no guarantees as to what foreign code may do. *Unsafe blocks* are an "escape hatch" that give the programmer certain extra powers, the soundness of which cannot be ensured by the compiler. [52]

4.3.2 Algebraic data types

Rust has two basic sorts of user-defined types: `structs` and `enums`. Structs are record types containing zero or more *fields*; enums are discriminated unions, representing a choice between a bounded set of alternatives, or *variants*. A tag,

or discriminant, is internally stored in every enum value to keep track of which variant is “active”. [45, Ch. 5–6]

Struct fields can be either named or anonymous; in the latter case the term *tuple struct* is used. Rust also supports fully anonymous tuples of any arity, such as the type of integer–float pairs `(i32, f32)`. Array types are denoted `[T; N]` for some element type `T` and constant size `N`. *Slice* references `&[T]` and `&mut [T]` borrow subsequences of data with a runtime length. Some examples of structs and enums are given in Listing 4.1: the struct `User` representing a user of a computer system, the tuple struct `Vector3` modeling a geometric 3-vector, and the enum *Complex* that represents a complex number stored in either Cartesian or polar form.

Listing 4.1: Structs and enums.

```
struct User {
    id: u64,
    name: String,
    email: String
}

struct Vector3(pub f32, pub f32, pub f32);

enum Complex {
    Cartesian { re: f32, im: f32 }, // Real and imaginary parts
    Polar { abs: f32, arg: f32 }    // Absolute value and argument
}
```

Two enums in the standard library, `Option<T>` and `Result<T, E>`, are ubiquitous in Rust code. They are used to represent the possible absence of a value and the result of an operation that may fail, respectively – by design, Rust has neither exceptions nor nullable references. An `Option<T>` can be either `Some(T)` or `None`. A `Result<T, E>` can be either `Ok(T)`, denoting success, or `Err(E)`, re-

porting a failure. These types correspond to the `Maybe` and `Either` types in Haskell. [45, Ch. 9]

4.3.3 Pattern matching

Algebraic data types are almost inevitably accompanied by support for *pattern matching:* a type of flow control that tests an expression (a *scrutinee*) against one or more patterns, and jumps to the (lexically) first branch with a matching pattern. Patterns often involve *destructuring* aggregate values into their constituent values. A powerful feature of pattern matching in Rust is exhaustiveness checking: the compiler enforces that the match arms cover all the possible values of the scrutinee expression. Listing 4.2 presents a function that returns the absolute value, or magnitude, of a `Complex` value. [45, Ch. 19]

Listing 4.2: Matching and destructuring a `Complex`.

```
fn abs(c: &Complex) -> f32 {
    match c {
        Complex::Cartesian { re, im } => (re * re + im * im).sqrt(),
        Complex::Polar { abs, .. } => abs,
    }
}
```

4.3.4 Special types

The nullary tuple, `()`, is the canonical *unit type*, returned by functions that do not return anything else. Its sole value is also spelled `()`. Unlike the `void` "pseudo-type" in C-like languages, `()` is a regular, first-class type. The unit type can be seen as the identity element of product types. [45, Sec. 20.3]

Custom unit types can be declared as fieldless structs. All unit types are guaranteed to be zero-sized *(ZST)*: variables of unit types have no unique addresses,

do not consume storage, and when used as fields, do not add to the size of the parent type. Such types are useful as *tag types*, or *phantom type parameters* [53], in generic programming. [52, Sec. 2.2]

The "never type", spelled `!`, is a type with no values. It is the return type of functions that unconditionally *diverge*, that is, either panic, exit the program, or loop forever instead of returning. `!` is the identity element of sum types. It is as of this writing not a full first-class type due to certain unresolved design issues. [52, Sec. 2.2]

4.3.5 Traits and generics

Traits are the Rust implementation of type classes. A trait defines an abstract interface that can be implemented for types. A trait can contain three types of *associated items*: functions, constants, and types. Associated functions that take a special `self` parameter are called *methods* and can be called with the `object.method()` notation familiar from other languages. Listing 4.3 shows a trait similar to the `Iterator` trait in the Rust standard library. [45, Ch. 10]

Listing 4.3: An iterator trait.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Trait implementations (type class instances) are created in Rust with `impl Trait` blocks. A significant difference to interfaces in languages like Java is that traits can also be implemented "retroactively" for types, making it possible to adapt an existing type to a new interface without having to define a wrapper type. Similar to Haskell, common vocabulary traits such as `Clone`, `Debug`, or `Eq`

can be *derived*, auto-implemented with a macro. [45, Ch. 10]

In Rust, traits, functions, types, and `impl` blocks can be generic. Generic parameters can be either types, lifetimes, or constant expressions. Like in C++, generics are monomorphized. Lifetime parameters are erased and do not exist at runtime. Listing 4.4 exhibits a simple vector type generic over its dimension and scalar type, storing its components in an array and deriving various utility traits.

Listing 4.4: A simple generic vector type.

```
#[derive(Copy, Clone, Debug, Eq, PartialEq, Hash]
pub struct Vector<S, const DIM: usize>(pub [S; DIM]);
```

Generic traits and trait implementations raise the problem of *coherence*: making sure that in any given context there is a unique applicable instance [54]. Rust requires that instances are *globally* unique and non-overlapping, and enforces this with a set of *orphan rules*. In short, local traits can be implemented for foreign types and foreign traits for local types, with some more complex rules when generics are involved [55, Sec. 6.12]. Matsakis [56] discusses Rust's orphan rules in more detail.

There is little that one can do with a fully unconstrained type variable – it can be moved, but hardly more than that[5]. A *trait bound* attached to a type parameter constrains the caller of a function and enables the function to use the interface declared by the given trait or traits. Listing 4.5 defines a function `dot` which implements the dot product for vectors of arbitrary dimension whose component type supports multiplication and assigning addition.

[5]Still, this is enough for some types such as a resizable array.

Listing 4.5: A generic dot product.

```
fn dot<T, const N: usize>(v: &Vector<T, N>, u: &Vector<T, N>) -> T
  where
    T: Default + AddAssign + Mul<Output = T>
  {
  // For the sake of this example, assume that default()
  // gives the zero value (additive identity) of T
  let mut result = T::default();
  for (a, b) in zip(v.0, u.0) {
    result += a * b;
  }
  result
}
```

4.4 Related work

This section reviews existing literature and practical implementations of geometry types. It focuses on attempts to alleviate correctness problems such as those enumerated in Chapter 2. Academic work on the topic is rather scarce, and popular software libraries are all rather "traditionally" typed. Work on representing physical quantities and units of measure is reviewed as well, because both the problem and solution spaces are closely related to the main topic of this work.

4.4.1 Modeling geometry

A mathematical basis for computer 3D graphics was established early on by pioneers of the field such as Roberts and Sutherland. In particular, Roberts [18] was the first to publish the insight of exploiting homogeneous coordinates to be able to express linear, affine and perspective transforms with 4x4 matrices, acting on 4-coordinate tuples able to represent both points and vectors as needed. This uniform representation is computationally very convenient.

The bulk of research on computer graphics has focused on performance and output fidelity; implementation correctness has received limited attention from both researchers and practitioners. Perhaps the most prominent advocates of introducing affine geometry to distinguish between points and vectors, and of reifying the concept of coordinate spaces and frames to avoid confusion, are Goldman [16] and DeRose [11] and their collaborators.

Goldman [17] discusses the so-called "illicit operations" in graphics, such as the addition of two points, and shows that such operations are well-defined in specific cases, as seen in Section 3.1.4. Goldman also gives an in-depth presen-

tation of the "ambient spaces" of 3D graphics – linear, affine, and projective – in [16] and [57].

DeRose et al. [11] make a case for so-called "coordinate-free" approach to geometry, abstracting out explicit numerical coordinates with a set of abstract data types (ADTs) written in C. Mann et al. [58] further refine this work, translating the library of ADTs to C++. Unfortunately the source code does not seem to be easily available anymore.

Building on the work by DeRose, Goldman, and their collaborators, Lee [6] observes that the relationship of orientations and rotations is analogous to that between vectors and points: orientations are affine "points" and rotations are linear "translations" between orientations. He argues that rotations and orientations have different optimal representations in code.

4.4.2 Type-level approaches

While the abstractions implemented by DeRose et al. [11] and Mann et al. [58] are elegant, they appear to be largely based on runtime tags and checks, with time and space overhead that is likely to be unacceptable in production code. This is possibly a major reason why they have not gained traction – this author was not able to find any third-party libraries or renderers implementing such coordinate-free approaches. A handful of authors have instead proposed the use of types and parametric polymorphism to reach the same goals, but with little or no runtime overhead.

Ou and Pellacini [59] describe a method to improve the correctness of renderer code using static type checking and generic programming, and provide an implementation, the SafeGI library. SafeGI distinguishes between points, vectors,

and normals, and tracks different coordinate spaces by parameterizing points and vectors with a space, and matrices with a source and target space. The authors demonstrate the feasibility of SafeGI by implementing a CPU and a GPU renderer, as well as porting large portions of the open-source physically-based renderer PBRT[6] to use the library.

Hughes et al. [13, Ch. 12] introduce a strongly-typed C++ geometry library for pedagogical purposes in the textbook *Computer Graphics: Principles and Practices*, but note that it may be too inefficient for production use. The authors point out that "[i]n general, however, your programs will be easier to write and debug if you rely on a few carefully written and tested programs for building and applying transformations, and use your language's type system to help you keep track of the difference between points and vectors. The more your linear algebra module can support the expression of what rather than how [...] the easier it will be to both understand and maintain your programs."

Sampson [60] identifies six broad problems with the state of OpenGL and GLSL, among them the difficulty of keeping track of objects in different coordinate frames in shader code. He outlines a type-based solution very similar to that presented in [59]. Sampson also discusses visual correctness, how it can be difficult to verify, and observes that automatic or systematic tests seem to be rarely used in shader programming.

Geisler et al. [7] observe that programming errors related to coordinate systems and transforms are common in graphics programming, and moreover, often difficult to debug. They introduce a type system that encodes three properties: the "reference frame", such as model or world space; the type of "geometric object", such as a point or a direction vector; and the "coordinate scheme", such

[6] https://www.pbrt.org/

as Cartesian or spherical coordinates. They argue that all three properties must be modeled and that they interact in subtle ways. The authors also introduce a shader language, *Gator*, with an extended type system that encodes these properties at type level.

Chen et al. [61] introduce CoolerSpace, a strongly typed abstraction for professional work with colors, including transformations between many different color spaces. It is implemented in Python and not meant for real-time applications, so performance is not a principal concern.

On the practical side, Sylvan [9] notes that mixing up transform matrices is a common source of errors, and suggests adopting a naming scheme to keep track of them. Yunchao He [62] discusses coordinate systems and the confusion they can cause in the context of the WebGPU library. Apple [10] has published a knowledge base article on debugging coordinate system issues, particularly in the context of GUI programming. Shavit [63] gives a few common examples of affine spaces, including pointers and time points as well as geometric points. He observes that the concept of affine spaces is surprisingly poorly known, despite being common in everyday life. Chandra [8] briefly discusses the idea of using tag types to parameterize vectors by their space.

4.4.3 Dimensions and units of measure

Physical quantities are associated with a *dimension*, such as time or speed. The rules of *dimensional analysis* define how quantities can be mixed; for example, adding meters to seconds is not meaningful. Moreover, quantities of the same dimension, but measured in different *units*, must not be mixed without proper

conversion.[7] Working with physical quantities is similar to working with geometric objects. Both pertain to endowing "plain" numeric values with extra semantics that determine how those values can interact. These semantics are often left implicit and thus error-prone, but could be made explicit by encoding them in the type system.

Several authors, such as Ou and Pellacini [59], have pointed out this connection, and this author was also inspired by the work on type-based quantities and units libraries. Dimensional analysis is also in itself applicable to computer graphics, with one natural use case being physics-based animation. Another use case, implemented in SafeGI [59], is in physically-based rendering (PBR), which seeks to model how light interacts with surfaces in the real world. PBR thus works on physical quantities such as radiance ($\mathrm{W\,sr^{-1}\,m^{-2}}$) [65].

Modeling dimensions and units in programming has been discussed by Cmelik [66], Kennedy [67], and Brown [68], among others. Pusz et al. [69] have collaborated on a formal proposal for adding such an API to the C++ standard library. They discuss various complexities involved, such as the distinction of units, dimensions, quantities, and quantity *kinds*. Particularly relevant to this work, they also bring up the need to model both affine and vector spaces (as a familiar example, the Celsius temperature scale is affine but not linear).

4.4.4 Practical work

The basic geometry abstractions of mainstream software libraries and game engines have seen little change compared to the seminal work in the 1960s. DirectX Math [70], Unreal Engine [71], Unity [72], and Godot [73] all use their

[7] Perhaps the most famous (and expensive) unit confusion in programming resulted in the loss of the NASA spacecraft *Mars Climate Orbiter* in 1999. The root cause was found to be a mix-up SI and American customary units by a contractor [64].

`Vector` data types to represent both points and vectors. Positions in space are commonly referred to as "coordinate vectors" or "position vectors". The same applies to the popular shading languages GLSL [74] and HLSL [75]. There is also no attempt to model the concept of coordinate spaces or to prevent errors caused by mixing up different spaces.

Kreylos [76] has implemented a C++ geometry library with explicit support for affine geometry and a separate type for homogeneous coordinate vectors. Eigen [77] is a C++ linear algebra library that pioneered the use of *expression templates* to achieve FORTRAN-like performance. Its operations are lazy, building up expression trees that can be compiled to very efficient code. However, it does not concern itself with coordinate frames, and as a pure linear algebra library it has no support for affine geometry.

There are several Rust libraries concerned with type-safe modeling of geometry. Euclid [78] is a Rust library that provides strongly-typed mathematical tools, with focus on 2D geometry and layouting. It uses generics and tag types to represent different spaces. The Alga library [79] includes `AffineSpace` and `VectorSpace` traits. Nalgebra [80] is a linear algebra library aimed at the use in games and graphics. The library has separate types for points and vectors as well as unit vectors, and includes a typed hierarchy of transforms. The Sguaba library [81] implements strongly typed coordinate and transform types for use in geoinformation systems (GIS), a use case quite closely related to graphics.

Prominent practical implementations of strongly-typed quantities and units include Boost.Units [82] shipping with the popular C++ library collection Boost and the more recent designs Units [83] and MP-Units [84]. A similar library for Rust is UOM [85]. The functional programming language F♯, developed by Microsoft, has built-in support for units of measurement [86].

5 Typing a renderer

In the discussion on the mathematics and algorithms of 3D rendering in Chapter 3, several potential sources of confusion and programming errors were identified. This chapter presents an API design that aims to provide a (partial) solution for these problems, with a real-world proof-of-concept implementation. Further analysis and evaluation follows in Chapter 6.

5.1 Introducing Retrofire

Retrofire[1], stylized `retrofire`, is an open-source software 3D rendering library, written by the author of this thesis. Some aspects of the library are intentionally "retro" (hence the name), evocative of the state of real-time 3D graphics in games and demoscene works in the late 1990s, just prior to when hardware GPUs began gaining popularity. In the "do-it-yourself" spirit of the 90s, the core Retrofire library uses no non-optional third-party libraries. Moreover, it only requires the `core` and `alloc` components of the Rust standard library, enabling it to run on constrained platforms such as WebAssembly and many embedded systems.

[1]Source code repository at `https://github.com/jdahlstrom/retrofire`;
API documentation available at `https://docs.rs/retrofire_core/`.

Retrofire started as a hobby project, with the author's primary goal to improve his Rust programming skills and knowledge. The first incarnation of Retrofire was a fairly run-of-the-mill renderer with ordinary geometry types. Frustration with debugging geometry-related errors led the author to rewriting it almost from scratch, this time with explicit focus on strongly-typed, yet minimal-overhead abstractions.

This chapter reviews some of the APIs of Retrofire as a case study. Particular attention is paid to the ways that the library attempts to mitigate the issues introduced in Section 2.1:

Difference between linear and affine objects. Retrofire distinguishes affine objects from linear ones by representing them with two traits, `Affine` and its subtrait `Linear`. Vectors and points are modeled as separate types.

Confusing objects in different frames. The point and vector types are parametric over the frame relative to which they are defined, and operations on incompatible objects are disallowed.

Transforming objects with wrong matrices. Matrices, representing affine and linear transforms, are parametric over their source and target frames and only accept correctly-typed arguments.

Confusing colors in different color spaces. Colors in Retrofire are affine objects similar to points, generic over the color space in which they are defined.

Confusion between angular quantities. Scalar angles are represented by the `Angle` type which requires explicitly specifying the unit used. Angular vector types are distinct from Cartesian vectors.

Figure 5.1: The Stanford dragon.

A minimal example of the use of Retrofire is given in Appendix A.1. It renders a colorful triangle[2] into an in-memory buffer and saves it as a PPM image file. More complex example programs can be found in the `/demos` directory of the repository.[3] Figure 5.1 displays the Stanford dragon, one of the classic test models in 3D graphics, as rendered by Retrofire at real-time frame rates on a 2015 MacBook Pro. This model has 50,000 vertices and 100,000 faces in total.

In the code listings of this chapter, some nonpertinent technical details have been elided. The implementations of many functions have also been omitted for brevity and marked with an ellipsis in a comment: `/* ... */`.

[2]The de facto "Hello, World" of graphics programming.

[3]`https://github.com/jdahlstrom/retrofire/tree/master/demos/`

5.2 Modeling affine and linear spaces

The traits `Linear` and `Affine` represent vector-like and point-like types, respectively. Recall from Section 3.1 that an affine space $\mathbf{A}$ has the following properties:

- A companion linear space $\mathbf{V_A}$ of translation vectors, or simply translations;
- Difference of two points $(-) : \mathbf{A} \times \mathbf{A} \to \mathbf{V_A}$, giving the translation from one point to another; and
- Addition of a vector to a point $(+) : \mathbf{A} \times \mathbf{V_A} \to \mathbf{A}$, applying a translation to a point.

A linear space $\mathbf{L}$ is an affine space that is its own companion space ($\mathbf{V_L} = \mathbf{L}$) and additionally has:

- An associated field of scalars S,
- An additive identity element $\mathbf{0}$, and
- Multiplication of a vector by a scalar $(\cdot) : \mathbf{L} \times S \to \mathbf{L}$.

These mathematical definitions can be faithfully translated to two Rust traits in a straightforward manner, as seen in Listing 5.1. The additive inverse `Linear::neg`, with a default implementation, is provided for convenience.

5.2.1 Points and vectors

Points in Euclidean spaces are represented by struct `Point`:

```
#[repr(transparent)]
pub struct Point<Repr, Space>(Repr, PhantomData<Space>);
```

The type parameter `Repr` denotes the type that represents the component tuple,

Listing 5.1: Definition of the `Affine` and `Linear` traits.

```
pub trait Affine {
    type Diff: Linear;

    fn add(&self, diff: &Self::Diff) -> Self;
    fn sub(&self, other: &Self) -> Self::Diff;
}

pub trait Linear: Affine<Diff = Self> {
    type Scalar;

    fn zero() -> Self;
    fn mul(&self, scalar: Self::Scalar) -> Self;
    fn neg(&self) -> Self { Self::zero().sub(self) }
}
```

typically an array, and the type parameter `Space` is a type tag (a phantom type) that encodes the coordinate frame in which the point is specified. For reasons involving the variance of lifetimes, type parameters cannot be left "unused" in a type definition in Rust. Rather, the inclusion of a dummy field of type `PhantomData` is required. `PhantomData` is zero-sized, so it does not affect the size or alignment of the type.

The attribute `#[repr(transparent)]` tells the compiler that the layout and ABI of the type should be that of the sole non-zero-sized member, in this case `Repr` [52, Sec. 2.3]. This guarantee can have performance benefits, but importantly it also makes it sound (albeit still `unsafe`) to *transmute* or *type pun* a wrapper type to its wrapped type. Type punning is discussed in more detail in Section 6.1.6.

The struct `Vector` represents real Cartesian vectors. Aside from the name, its definition is identical to that of `Point`:[4]

[4]Indeed the two types, as well as `Color`, could simply be type aliases of a single more general type. However, this would impact ergonomics, as discussed in Section 6.2.5.

```
#[repr(transparent)]
pub struct Vector<Repr, Space>(Repr, PhantomData<Space>);
```

5.2.2 Spaces and bases

Retrofire defines tag types representing the common coordinate systems introduced in Section 3.2.3, as seen in Listing 5.2. The generic type `Real` represents real Euclidean spaces, parameterized by dimension and basis. The basis parameter is used to distinguish different coordinate frames; for example, the type `Real<3, World>` represents a 3D world space and is distinct from `Real<3, View>`. Types for the five standard Cartesian frames of 3D rendering are provided. The unit type `()` is used to denote a generic or "default" frame. Additionally, the $\mathbf{P}_3(\mathbb{R})$ projective clip space has its own type.

It should be emphasized that the different frames have no *inherent* geometric relationship with each other. A relationship is established contextually with a transform – for example, there is no single unique model space because every object in a scene likely has its own model-to-world transformation matrix.

Listing 5.2: Common spaces and bases.

```
struct Real<const DIM: usize, Basis = ()>(PhantomData<Basis>);

struct Model;
struct World;
struct View;
struct Ndc;
struct Screen;

struct Proj3;
```

5.2.3 `Affine` and `Linear` implementations

For the sake of clarity, the rest of this chapter will make use of the following, slightly more concrete type aliases:

```
type VectorN<Sc, Sp, const DIM: usize> = Vector<[Sc; DIM], Sp>;
type PointN<Sc, Sp, const DIM: usize> = Point<[Sc; DIM], Sp>;
```

As expected, `PointN` implements `Affine` (Listing 5.3), while `VectorN` implements both `Affine` and `Linear` (Listing 5.4)

The trait bounds of the scalar type parameters merit a closer look. A scalar is defined as a `Linear` type that is its own scalar type, making use of the fact that any field is, indeed, a vector space over itself. A separate `Scalar` trait is thus not needed, although it could be useful in some cases.

Listing 5.3: The `Affine` implementation for `PointN`.

```
impl<ScSelf, ScDiff, Sp, const DIM: usize> Affine for
    PointN<ScSelf, Sp, DIM>
where
    ScSelf: Affine<Diff = ScDiff> + Copy,
    ScDiff: Linear<Scalar = ScDiff> + Copy,
{
    type Space = Sp;
    type Diff = VectorN<ScDiff, Sp, DIM>;

    fn add(&self, other: &Self::Diff) -> Self { /* ... */ }
    fn sub(&self, other: &Self) -> Self::Diff { /* ... */ }
}
```

For convenience, `Point` and `Vector` implement the relevant operator overloading traits `Add`, `Sub`, `Mul`, `Div`, `Neg`, as well as the corresponding assignment variants. `Vector` naturally also provides the standard tools of vector algebra, including the dot product and cross product, scalar and vector projection, nor-

malization, and so on. `Point` has a different API, with dedicated methods for finding the distance between two points, for instance.

Listing 5.4: The `Affine` and `Linear` implementations for `VecN`.

```
impl<Sc, Sp, const DIM: usize> Affine for VecN<Sc, Sp, DIM>
where
    Sc: Linear<Scalar = Sc> + Copy,
{
    type Space = Sp;
    type Diff = Self;

    fn add(&self, other: &Self) -> Self { /* ... */ }
    fn sub(&self, other: &Self) -> Self { /* ... */ }
}

impl<Sc, Sp, const DIM: usize> Linear for VectorN<Sc, Sp, DIM>
where
    Self: Affine<Diff = Self>,
    Sc: Linear<Scalar = Sc> + Copy,
{
    type Scalar = Sc;

    fn zero() -> Self { [Sc::zero(); DIM].into() }

    fn mul(&self, s: Sc) -> Self { /* ... */ }
}
```

5.3 Affine and linear transforms

5.3.1 `AffineMap` and `Compose` traits

The `AffineMap` trait is a "marker trait" without behavior. It is used to encode the source and target spaces of a transform, so that only points in the correct source space are accepted and points in the correct target space are returned. Listing 5.5 shows the trait and its principal implementing types. The

`RealToReal` type is an `AffineMap` that represents transforms between two coordinate frames in an Euclidean space, while the `RealToProj` type represents projective transforms.

Listing 5.5: The `AffineMap` trait and implementers

```
pub trait AffineMap {
    type Source;
    type Dest;
}

pub struct RealToReal<const DIM: usize, Src = (), Dest = Src>(
    PhantomData<(Src, Dest)>,
);

pub struct RealToProj<SrcBasis>(PhantomData<SrcBasis>);

impl<const DIM: usize, S, D> AffineMap for RealToReal<DIM, S, D> {
    type Source = Real<DIM, S>;
    type Dest = Real<DIM, D>;
}

impl<S> AffineMap for RealToProj<S> {
    type Source = Real<3, S>;
    type Dest = Proj3;
}
```

The `Compose` trait, shown in Listing 5.6, is a *type-level function* that gives the composition of two affine maps. Specifically, if `A` and `B` are `AffineMaps` and `A: Compose<B>`, then `<A as Compose<B>>::Result` is an `AffineMap` from `A::Source` to `B::Dest`. This relationship for real-to-real and real-to-projection maps is expressed by the `impl` blocks in Listing 5.6.

Listing 5.6: Composition of affine maps.

```
pub trait Compose<Inner: AffineMap>
where
    Self: AffineMap<Source = Inner::Dest>
{
  type Result: AffineMap<Source = Inner::Source, Dest = Self::Dest>;
}

impl<const DIM: usize, S, I, D> Compose<RealToReal<DIM, S, I>>
    for RealToReal<DIM, I, D>
{
  type Result = RealToReal<DIM, S, D>;
}
impl<S, I> Compose<RealToReal<3, S, I>> for RealToProj<I> {
  type Result = RealToProj<S>;
}
```

5.3.2 Matrices

The `Matrix` type models standard linear, affine, and projection matrices, discussed in Sections 3.1.3 and 3.1.5. It is parametric over the `AffineMap` that it represents. The aliases `Mat2`, `Mat3`, and `Mat4` are provided for convenience:

```
#[repr(transparent)]
pub struct Matrix<Repr, Map: AffineMap>(pub Repr, PhantomData<Map>);

pub type Mat2<S, D> = Matrix<[[f32; 2]; 2], RealToReal<2, S, D>>;
pub type Mat3<S, D> = /* similarly */
pub type Mat4<S, D> = /* similarly */
```

Retrofire implements the standard matrix operations, including composition, inversion, transposing, and application to points and vectors. Functions are provided for constructing several types of transformation matrices, including scaling, translation, rotation about an axis, and change of frame, as well as creating projection and viewport transform matrices.

Listing 5.7: Composition of two matrices.

```
impl<const N: usize, Outer> MatN<N, Outer> {
    pub fn compose<Inner>(&self, other: &MatN<N, Inner>)
        -> MatN<N, <Outer as Compose<Inner>>::Result>
    where
        Inner: AffineMap,
        Outer: Compose<Inner>,
    {
        // Compute the product self * other
    }
}
impl<const N: usize, Inner> MatN<N, Inner> {
    pub fn then<Outer>(&self, other: &MatN<N, Outer>)
        -> MatN<N, <Outer as Compose<Inner>>::Result>
    where
        Outer: Compose<Inner>,
    {
        other.compose(self)
    }
}
```

5.3.3 Matrix operations

Matrix composition (multiplication) is defined for matrices with compatible `AffineMap` types, as seen in Listing 5.7. Composition is implemented as two named methods, `then` and `compose`, in order to minimize confusion about the order of application:

$$\text{compose}(\mathbf{A}_{S\to T}, \mathbf{B}_{R\to S}) = \text{then}(\mathbf{B}_{R\to S}, \mathbf{A}_{S\to T}) = \mathbf{A}_{S\to T} \cdot \mathbf{B}_{R\to S} = (\mathbf{AB})_{R\to T}$$

Recall from Section 3.1 that the matrix product $\mathbf{AB}$ is defined in terms of dot products of row vectors of $\mathbf{A}$ with column vectors of $\mathbf{B}$. For the dot product to be well-defined, the dimensions must match, but additionally the row vectors should be in the *source* and the column vectors in the *target* space of a matrix:

```
pub fn row_vec(&self, i: usize) -> VecN<N, M::Source> { /* ... */ }
pub fn col_vec(&self, i: usize) -> VecN<N, M::Dest> { /* ... */ }
```

This is a somewhat nonobvious result that, however, is consistent with the interpretation of row and column vectors as basis vectors in their respective spaces. Transposing or inverting a matrix swaps the row and column vectors, and consequently the source and target spaces, as expected.

The `invert` method computes the inverse of a matrix, swapping the source and destination spaces as expected. Listing 5.8 shows its signature for real-to-real matrices, but taking the inverse of a projection matrix can be useful as well, allowing one to compute the line of points in view space that correspond to given screen space coordinates (for example, the pointer position).

Listing 5.8: Inverting a matrix.

```
impl<const N: usize, S, D> MatN<N, RealToReal<S, D>> {
    pub fn invert(&self) -> MatN<N, RealToReal<D, S>> {
        // Compute the inverse of self
    }
}
```

Listing 5.9 shows the signature of the `apply` method, transforming a point by a square matrix of a compatible dimension. It takes a point in the source space and returns a point in the destination space of the affine map:

$$\mathrm{P}_T = \mathbf{A}_{S \to T} \cdot \mathrm{P}_S$$

Listing 5.9: Transforming a point with a matrix.

```
impl<const N: usize, M: AffineMap> MatN<N, M> {
    pub fn apply(&self, p: &PointN<N, M::Source>)
      -> PointN<N, M::Dest> {
        // Transform p by self
    }
}
```

5.4 Colors and color spaces

As discussed in Section 3.2, colors in computer graphics are represented as coordinate tuples that specify a position in a color space. The original assumption of the author was that points could be modeled as affine objects, but in fact colors are more similar to vectors than points. Light is additive and the sum of colors is a well-defined and necessary operation.

The standard basis vectors of the sRGB space are $\hat{\mathbf{r}} = (1, 0, 0)$, $\hat{\mathbf{g}} = (0, 1, 0)$, and $\hat{\mathbf{b}} = (0, 0, 1)$. In the ubiquitous eight-bit representation of color channels, these are expressed as fixed-point values, with $1 \equiv 255$.

The basic `Color` type is essentially identical to `Vector`. Constructor functions and convenience aliases are also defined, as seen in Listing 5.10. Colors with `u8` channels are typically used for output, whereas floating-point colors are used internally in calculation to ensure sufficient precision and range, as well as the ability to represent color *differences* which can be negative. Retrofire defines the color space types shown in Listing 5.11, as well as conversions between them.

Operator overloading traits like `Add` and `Sub` are implemented, similar to vectors. The channel-wise multiplication of two colors is not a standard linear operation, but is defined due to its importance in shading calculations.

Listing 5.10: The `Color` type, aliases, and constructors.

```
#[repr(transparent)]
pub struct Color<Repr, Space = ()>(pub Repr, PhantomData<Space>);

pub type Color3<Space = Rgb> = Color<[u8; 3], Space>;
pub type Color4<Space = Rgba> = Color<[u8; 4], Space>;

pub type Color3f<Space = Rgb> = Color<[f32; 3], Space>;
pub type Color4f<Space = Rgba> = Color<[f32; 4], Space>;

pub const fn rgb<Ch>(r: Ch, g: Ch, b: Ch) -> Color<[Ch; 3], Rgb> {
    Color::new([r, g, b])
}
/* Functions rgba, hsl, and hsla similarly. */
```

Listing 5.11: The color spaces supported by Retrofire.

```
/// (Gamma-encoded) RGB (red, green, blue) color space.
pub struct Rgb;
/// Gamma-encoded RGB color space with opacity.
pub struct Rgba;
/// Linear RGB color space.
pub struct LinRgb;
/// HSL color space (hue, saturation, luminance).
pub struct Hsl;
/// HSL color space with opacity.
pub struct Hsla;
```

Possible future additions include a space for colors with premultiplied alpha (see 3.2.5), as well as perceptual color spaces such as Oklhc. Various color blending modes for transparency and special effects should also be implemented. Retrofire does not currently support blending of fragments, partially because it is a rather performance-intensive operation in a software renderer and transparency was thus not a common effect in 1990s real-time 3D graphics.

5.5 Angular quantities

Angles are typically expressed in software as plain floating-point (or sometimes integer) values. A problem with this basic representation is that the unit of measurement is not explicit and must be separately documented, which may lead to programming errors. Two angular units, degrees and radians, are in common use, with the former more intuitive to humans and the latter easier to work with mathematically. Other units, such as *turns*, can be useful as well.

Listing 5.12: The `Angle` type and its constructors.

```
#[repr(transparent)]
pub struct Angle(f32);

pub fn rads(a: f32) -> Angle { Angle(a) }
pub fn degs(a: f32) -> Angle { Angle(a * RADS_PER_DEG) }
pub fn turns(a: f32) -> Angle { Angle(a * RADS_PER_TURN) }

pub fn asin(x: f32) -> Angle { Angle(x.asin()) }
/* ... */

impl Angle {
    pub fn sin(self) -> { self.0.sin() }
    /* ... */
    pub fn as_degs(self) -> f32 { self.0 / RADS_PER_DEG }
    /* ... */
}
```

Retrofire aims to avoid confusion by introducing the `Angle` newtype that wraps an `f32`, with constructors that make the choice of unit explicit (Listing 5.12). `Angle` defines the standard trigonometric functions as methods, and their inverse functions as free functions that return `Angle`s. Converting an `Angle` to a plain `f32` requires the use of one of the explicit `as_*` methods.

Vectors in polar and spherical coordinate systems are defined as type aliases of

`Vector`, with `Polar` and `Spherical` tag types respectively (Listing 5.13). These vector types have constructors that take `Angle`s, reducing the chance of mixing up argument order. The angular components have accessors that return `Angle`s.

Listing 5.13: Angular vectors.

```
pub struct Polar<B>(PhantomData<B>);
pub struct Spherical<B>(PhantomData<B>);

pub type PolarVec<B> = Vector<[f32; 2], Polar<B>>;
pub type SphericalVec<B> = Vector<[f32; 3], Spherical<B>>;

pub fn polar<B>(r: f32, az: Angle) -> PolarVec<B> { /* ... */ }
pub fn spherical<B>(r: f32, az: Angle, alt: Angle)
  -> SphericalVec<B> {
    /* ... */
}

impl<B> PolarVec<B> {
    pub fn r(self) -> f32 { self.0[0] }
    pub fn az(self) -> Angle { Angle(self.0[1]) }

    pub fn to_cart(self) -> Vec2<B> { /* ... */ }
    /* ... */
}
impl<B> SphericalVec<B> {
    pub fn r(self) -> f32 { self.0[0] }
    pub fn az(self) -> Angle { Angle(self.0[1]) }
    pub fn alt(self) -> Angle { Angle(self.0[2]) }

    pub fn to_cart(self) -> Vec3<B> { /* ... */ }
    /* ... */
}
```

5.6 Shaders

Any `Affine` type can be linearly interpolated, or "lerped", as seen in Listing 5.14 (as noted in Section 3.1.4, a lerp is simply an affine combination of two points). This means that they can be used as varying vertex attributes, that is, interpolated across polygon faces during rasterization[5]. The renderer accepts attributes of any `Affine` type.

Listing 5.14: A generic lerp implementation.

```
pub fn lerp<T, S>(a: &T, b: &T, s: S) -> Self
where
    T: Affine<Diff: Linear<Scalar = S>>
{
    self.add(&other.sub(self).mul(u))
}
```

As discussed in Section 3.2.4, shaders are user-defined functions or program fragments that are used to customize the behavior of certain pipeline stages. As of now, Retrofire supports the two "basic" types of shaders: vertex and fragment shaders.

A vertex shader is invoked for every vertex passed to the renderer. It should transform the input vertex to the clip space and return the result. It is allowed to change the type of the vertex attribute – for example, in old-fashioned per-vertex lighting[6], input vertices with normal vectors are mapped to output vertices with colors. A pixel shader, in turn, takes a fragment, containing the screen-space coordinates and the corresponding values of the varyings, as input and outputs an RGBA color. The type of the fragment's varyings matches the attribute type of the vertex shader's output.

[5]The actual implementation is more efficient than repeatedly invoking `lerp`.

[6]Also called Gouraud shading, after Henri Gouraud.

Shaders are represented by the `VertexShader` and `FragmentShader` traits, presented in Listing 5.15. These traits are implemented for the appropriate `Fn` and `FnMut` types, allowing the use of closures and functions as shaders. The `Shader` trait is a composition of a vertex and fragment shader, ensuring that their type parameters are compatible.

The *uniform* parameter passed to the vertex shader carries arbitrary user data that is constant across the entire primitive, rather than varying like the interpolated vertex attributes. The most common uses for the uniform parameter are passing transform matrices, texture references, and light source information to the shaders.

Listing 5.15: The shader traits.

```
pub trait VertexShader<In, Uni> {
    type Output;

    fn shade_vertex(&self, vertex: In, uniform: Uni) -> Self::Output;
}

pub trait FragmentShader<Var> {
    fn shade_fragment(&self, frag: Frag<Var>) -> Option<Color4>;
}

pub trait Shader<Vtx, Var, Uni>:
    VertexShader<Vtx, Uni, Output = Vertex<ProjVec3, Var>> +
    FragmentShader<Var>
{}
```

5.7 Other related APIs

The preceding sections of this chapter have reviewed the fundamental "vocabulary" API of Retrofire, implementing the system of geometry types that this

thesis is primarily concerned with. This section gives a brief look at some of Retrofire's higher-level interfaces that are built on top of the foundation.

5.7.1 Cameras

The `Camera` type encapsulates the handling of the world-to-view, projection, and viewport transforms. `Camera` has methods for easily setting up the viewport and projection transforms, as well as accessors for the various matrices and their inverses.

The world-to-view transform is further abstracted out to any type that implements the `cam::Transform` trait. This makes it easy to write different camera motion modes, each with an appropriate interface. Currently implemented are a first-person "mouse and keyboard" mode, a third-person "orbiting" mode, and an airplane-like mode based on pitch, yaw, and roll angles.

Writing camera transform code can be tricky. First, the "inverse" nature of the world-to-view transform can feel nonintuitive, and second, it can be useful to move and orient a camera in either world or view space, depending on the use case. The type-checked geometry API helps make the difference clear, and the author has discovered at least one latent bug by rewriting a camera routine to be more rigorously typed.

5.7.2 Splines

A *spline* is a type of piecewise-defined parametric curve that fulfills some desired smoothness and continuity properties. They have found many uses in CAD, graphic design, font design, and computer animation. [12, Ch. 17]

A spline of degree n is an affine combination of n points in some affine space,

with the coefficients given by a set of *basis polynomials*. For example, the commonly used cubic Bézier curve for the points P_0 to P_3 is defined by the *Bernstein polynomials* [87]:

$$B_3(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 \,.$$

These coefficients have the required property of adding up to unity for any t. The curve starts at the first point ($t = 0$) and ends at the last point ($t = 1$). The two intermediate points control the curve by "pulling" it towards them. Several Bézier curves can then be concatenated into a smooth spline by aligning their control points [12, Ch. 17]

Retrofire includes a generic implementation of several types of cubic splines[7] that accept any `Affine` type as the point type – for instance, one could create smooth color gradients with splines. It implements useful routines such as computing the gradient (derivative) and local curvature of a spline, important for animating the motion of an object, or the camera, along a curved path.

It is possible to define a specific type of frame at any point of a parametric curve. The tangent of the curve is chosen as one of the basis vectors, and the other two orthogonal components are selected depending on the use case; for example, animation is often concerned with so-called rotation-minimizing frames. This type of a curve-aligned frame is an example of a TBN, or Frenet–Serret frame, previously mentioned in Section 3.2.4.

[7] Specifically Bézier, Hermite, Catmull–Rom, and B-splines.

5.7.3 Intersection tests

A recent addition to the `geom` package of Retrofire is a collection of intersection testing routines between different geometric objects. These have several uses in games and animations, such as collision detection and picking (selecting objects under the cursor or targeting reticle). Particularly useful is finding intersection points between *rays* (half-lines) and other types of object, such as the bounding volume of a mesh.[8]

The affine–linear distinction is useful here. For example, a ray type can be implemented as follows, immediately yielding well-typed `Ray<Point2>` and `Ray<Point3>`:

```
struct Ray<T: Affine> {
    origin: T,
    direction: T::Diff
}
```

In the author's experience, the distinction was genuinely helpful when working on the intersection routines and also offered some new insights into the equations involved. The type-level encoding of coordinate frames is naturally also highly useful in avoiding meaningless intersection tests between objects in different frames. However, one problem that the current Retrofire implementation does not solve is that every object has its own model space, but there is only one `Model` tag type. Interaction of two objects, both in model space(s), is thus well-typed but the result is not geometrically meaningful. This may not be a big problem in practice – typically it is clear that any object interactions should be performed in the world (or view) space, and Retrofire makes the choice of space clear.

[8] This is, naturally enough, also the basis for the *ray-tracing* rendering algorithm.

6 Analysis and discussion

This chapter discusses the Retrofire API from the viewpoint of research questions 2 and 3. Section 6.1 examines several aspects of performance that might be impaired by abstraction layers, and describes how Retrofire attempts to avoid performance loss. Section 6.2 discusses programmer ergonomics, including aspects such as flexibility, build speed and helpfulness of compiler errors. Section 6.3 summarizes the findings of this chapter.

The basic goals that have driven the design include:

- Idiomatic Rust code and API design that follows guidelines such as [88],
- Use of "zero-overhead" features like monomorphized generics,
- Simple, transparent type layouts to help reason about performance, and
- "Escape hatches" to opt out of compile-time checks if necessary.

6.1 Performance

The time complexity of a standard non-parallelized software rendering pipeline is essentially $\mathcal{O}(\max(n, m))$, where n is the number of vertices and m is the number of fragments output. This reflects two important performance metrics: *vertex throughput* (vertices processed per second) and *fill rate* (fragments drawn per second). In practice, the bottleneck is usually fill rate.

For the rendering algorithm, linear time complexity is clearly optimal. Practical performance gains are all in paring down the constant factor, as well as trying to minimize n and m without impacting image quality by being smarter about what to draw. Techniques used to achieve the latter include early culling of non-visible geometry, dynamically adjusting the level of detail (LoD), and avoiding overdraw. These are largely outside the scope of this work, although many implementations of such algorithms could plausibly benefit from a Retrofire-like API.

GPUs achieve their tremendous rendering performance by massive parallelism, and a parallelized software renderer running on a modern multi-core CPU can also be an order of magnitude faster than a strictly serial renderer. Nonetheless, discussion of multi-threaded rendering is left out of this work because it appears to be largely orthogonal to the topic of strongly-typed geometry APIs: nothing appears to prevent parallelizing a library such as Retrofire while keeping the typed interface the same.

Optimization for modern hardware is discussed, among others, by Fog [3], Bakhvalov [89] and Dean & Ghemawat [90]. Some major topics brought up in these works are

- data locality,
- vectorization,
- and memory allocation,

which will be discussed in the following sections. Additionally, function inlining is discussed in its own section because it is a vital compiler optimization that functions as a "force multiplier", enabling a plethora of downstream optimizations.

6.1.1 Methods

Retrofire collects and reports some rendering statistics, such as the mean vertex throughput and fill rate of a program. A set of micro-benchmarks of performance-sensitive routines, as well as "end-to-end" benchmarks exercising the entire rendering pipeline, were implemented using the benchmarking library Divan[1].

To provide a baseline for benchmarks, a version of the renderer and geometry API was created that omits the reference frame type parameters and is thus similar to a traditional rendering API. Furthermore, a simplified, less generic rasterizer was written to study abstraction overhead not directly related to the geometry API.

Optimized assembly emitted by rustc was studied using `cargo-asm` (an add-on to the Rust build tool Cargo) as well as Matt Godbolt's superb website Compiler Explorer[2]. *Flamegraph* diagrams were used to visualize profiler output, revealing potential bottlenecks and hot spots that could benefit the most from further optimizations. Other tools used for performance analysis include `cargo-bloat`, `cargo-llvm-lines`, and `cargo-remark`. They will be discussed in more detail in the following sections.

6.1.2 Data locality

As noted in the classic paper "Hitting the Memory Wall" by Wulf and McKee[3] [91], a significant gap has grown between CPU and RAM performance in the recent decades. CPUs now contain several levels of fast on-die cache memory for data and instructions, and a cache miss may incur a latency of hundreds

[1] https://crates.io/crates/divan

[2] `https://rust.godbolt.org`

[3] Sally McKee sadly passed away on 21 Feb 2025, when this thesis was in its early stages.

of clock cycles in the worst case as the needed data is fetched from RAM. Consequently, organizing data such that good *memory locality* is maintained has become a major goal in high-performance computing [92].

The most efficient data layout is a contiguous array iterated linearly, containing only data relevant to the code in question. This achieves maximum locality and ensures that there is more data in the cache already prefetched when it is needed, thanks to the predictable memory access pattern. On the other hand, the pointer indirections inherent in object-oriented, dynamic-dispatch languages such as Java can lead to highly inefficient access patterns on modern hardware. Languages such as C, C++, and Rust store values of any type in arrays inline, without pointer indirection, and Retrofire takes full advantage of this. The exclusive use of static polymorphism ensures that there are no indirections to either code or data. Furthermore, Retrofire does not use node-based data structures, only the standard `Vec` dynamic array and built-in arrays.

More could be done, however. A linear contiguous data layout can still be inefficient if an operation only accesses some fields of a multi-field record type. The rest of the fields are still fetched and stored in caches even though they are not used. This layout is typically called *array-of-structures* or AoS. To achieve better utilization of cache and memory bandwidth, its transpose, *structure-of-arrays* or SoA, is often preferable. This is something that could be investigated in the future. [93]

In general, *data-oriented design* has become a major paradigm in high-performance programming. It is something of an antithesis to classic object-oriented design: rather than bundling code and data, which can encourage data inside objects being scattered around the heap, possibly behind multiple indirections and with extra metadata attached, data should be *separated* from code! Data-

oriented design has data locality as the first priority. [94], [95]

6.1.3 Vectorization

Modern CPU architectures are *superscalar:* they allow parallel execution even within a single processor core. *SIMD* (Single Instruction, Multiple Data) is a form of vector processing where mathematical operations can be applied to an array of several floating-point or integer values in parallel, without additional cost. For bulk data processing workloads this can bring an order-of-magnitude improvement in throughput "for free", as compared to equivalent scalar (that is, non-vectorized) code.

Writing SIMD code by hand is difficult and tedious, usually requiring the use of nonportable, unsafe intrinsics that map directly to machine instructions. The set of supported operations varies even within a processor family, and writing portable code requires runtime feature detection and *multiversioning*. Moreover, special and often unintuitive programming techniques must be employed to achieve optimal performance. [3]

Autovectorization is a class of optimizations where the compiler attempts to automatically translate scalar source code to an equivalent SIMD form. However, its capabilities are limited compared to skillful manual vectorization, as a best-effort optimization it cannot be guaranteed to happen, and care must be taken to write code in a way that is amenable to autovectorization – particularly to avoid data dependencies and branching within loops. [96]

The internals of a rendering library like Retrofire could in principle be written in highly optimized SIMD code, shielding the user from the complications involved. However, any APIs calling back to user code, such as Retrofire shaders,

present a conundrum. If the shaders – invoked from very hot inner loops – are written in normal scalar code, any SIMD optimizations are largely useless. On the other hand, if the library offers a SIMD-based interface, the user must also know how to write SIMD code.

A solution could be to write the interface in terms of high-level types that abstract out many of the pain points of vectorization. Mature third-party libraries for Rust exist that could be used, but like some other popular languages[4], Rust (as of version 1.94) has no standard high-level SIMD abstraction. This is an unfortunate state of affairs, given that mainstream processors have had some degree of SIMD support for over twenty years. Levien [97] gives an overview of the 2025 state of SIMD in Rust.

Retrofire, as of this writing (version 0.4), contains no explicit SIMD support and relies on the autovectorizer for any optimizations. The primitive geometry types are designed to be generic enough to support SIMD representation, and the author has verified that the renderer accepts custom SIMD-based types as varyings. However, the standard scalar `Point3<_>` type is used to represent all vertex positions; making the position types parametric might be worthwhile.

Analysis of generated assembly has revealed that the compiler is able to autovectorize the core rendering loop to an extent. One hindrance to what the optimizer can do is that strict adherence to IEEE/IEC 754 floating-point semantics disallows many reasonable-seeming transformations – for example, unlike real numbers, floats are not associative in general. C and C++ compilers typically have means to toggle more lenient floating-point semantics, but Rust currently lacks a safe way to do that. [98], [99]

[4]For instance, C++ is finally gaining a standard SIMD library in the 2026 version of the ISO standard.

6.1.4 Monomorphization

In Rust (like in C++), a generic function is compiled by creating a separate copy for every type parameter or combination of parameters with which it is instantiated in the program; this is called *monomorphization*. In this way the compiler is able to optimize every instance individually, which can have major performance benefits compared to dispatch based on runtime polymorphism. However, it can also lead to "code bloat", which negatively impacts executable size, but more pertinently, cache locality of the processor's instruction cache.

Some generic functions can be partially "polymorphized" manually by extracting parts independent of some or all generic parameters into less-generic helpers, a technique widely used by the Rust standard library [100, Ch. 19]. Use of runtime polymorphism, when applicable, eliminates code duplication at the expense of a slight performance cost. *Type erasure* can be employed to write a monomorphic but non-typesafe private implementation behind a safe, typed interface. These techniques, however, can make code more difficult to maintain and reason about.

The use of phantom type parameters, existing only at compile time with no effect on code generation, can lead to a compiled binary containing identical copies of generic functions. For instance, it would be unfortunate if a function like `Matrix::compose` were duplicated for every pair of source-target spaces it is invoked for, given that the machine code is identical for each copy.

The `cargo-bloat` tool analyzes a compiled binary and reports the size of functions within. Listing 6.1 shows an abridged output of `cargo bloat` when run on `solids` demo program[5] bundled with Retrofire. The report reveals that

[5] `https://github.com/jdahlstrom/retrofire/tree/master/demos`

Listing 6.1: Excerpt of cargo-bloat output, `solids` demo

```
File  .text    Size  Crate Name
...
0.3%  0.9%  3.6KiB  retrofire_geom retrofire_geom::io::parse_obj
0.3%  0.9%  3.6KiB  retrofire_geom retrofire_geom::io::parse_obj
0.2%  0.8%  3.0KiB  retrofire_front retrofire_front::minifb::Window::run
0.2%  0.5%  2.1KiB  retrofire_geom ...::solids::lathe::Lathe<P>::build
0.2%  0.5%  2.1KiB  retrofire_geom ...::solids::lathe::Lathe<P>::build
0.2%  0.5%  2.1KiB  retrofire_geom ...::solids::lathe::Lathe<P>::build
0.2%  0.5%  2.0KiB  retrofire_geom ...::solids::lathe::Lathe<P>::build
0.1%  0.4%  1.7KiB  retrofire_core retrofire_core::render::raster::scan
0.1%  0.4%  1.7KiB  retrofire_core retrofire_core::render::render
...
```

there are no less than four nearly-identical copies of the method `Lathe::build`, which creates surface-of-revolution objects, as well as two copies of the function `io::parse_obj`. These functions are not performance-critical, and the author was able to remove the duplication by rewriting them to use runtime polymorphism instead of generics. Other cases of redundant copies have also been alleviated or eliminated in the course of this analysis. For instance, some `Vector` and `Matrix` methods have been refactored as a result to delegate to minimally-generic lower-level helpers.

6.1.5 Code inlining

Thanks to monomorphization, generic code is amenable to *inlining*. Inlining a function call erases overhead involved in the call, but more crucially, it unlocks optimization across function boundaries and in general gives the optimizer more context to work with [101]. This can have considerable performance benefits. However, excessive inlining leads to code being duplicated many times over, which can put pressure on the instruction cache as mentioned above.

In Rust, the optimizer can be guided by annotating individual functions with the `#[inline]` attribute [55, attributes.codegen.inline], and Retrofire attempts to use this attribute judiciously and based on measured performance. For example, the attribute enables inlining between compilation units. At high optimization levels, LLVM inlines functions aggressively even without guidance, but during active development, when compiling with fewer optimizations to reduce build times, correct use of attribute can have a major effect on performance.

To evaluate the ability of the compiler to inline Retrofire code, a minimal `no_std` demo[6] exercising the entire rendering pipeline was compiled with full optimizations and analyzed with `cargo-bloat`. Listing 6.2 shows that the user code and the whole pipeline have been inlined into `main` and two other functions; all the rest are related to the OS runtime and panic handling. For a comparison, when the same program is compiled without optimizations, the size of the `.text` section is over 200 kiB, containing more than 300 functions from `retrofire_core` and over a thousand other functions!

Listing 6.2: Excerpt of `cargo-bloat` output, `no_std` demo.

```
File  .text  Size      Crate Name
24.4% 42.4% 4.0KiB      [Unknown]  main
7.0%  12.1% 1.1KiB           std  core::iter::traits::iterator::Iterator...
4.2%   7.4%   718B retrofire_core retrofire_core::render::raster::scan...
4.2%   7.3%   714B      [Unknown] __isa_available_init
2.2%   3.9%   380B      [Unknown] mainCRTStartup
13.0% 22.5% 2.1KiB           And 45 smaller methods. Use -n N to show more.
57.6% 100.0% 9.5KiB          .text section size, the file size is 16.5KiB
```

[6] https://github.com/jdahlstrom/retrofire/tree/master/demos/nostd

6.1.6 Memory copies and allocations

In a traditional geometry API, where vectors are not parameterized by space or frame, applying an in-place transform to every vector in a list is trivial. However, in Retrofire, a slice of model-space vectors cannot be suddenly turned into a slice of view-space vectors, even though their memory representations are identical. Allocating a new collection for the results may be necessary, which is not optimal.

However, if it can be guaranteed that the runtime representations are compatible, the extra allocation and copying can be optimized out. Rust's `Vec` collection type implements this optimization for types that have the same size and alignment. Thus, for example, the following code, converting a `Vec<i32>` to a `Vec<f32>` in fact reuses the allocation:

```
fn ints_to_floats(v: Vec<i32>) -> Vec<f32> {
    v.into_iter().map(|i| i as f32).collect()
}
```

More generally, reinterpreting a value of a type as another type is called *type punning*. This can be done in a few ways. Using the unsafe `std::mem::transmute` function [52, Sec. 4.4] is one method; another is doing an unsafe pointer cast. To avoid having to write unsafe code, a library such as `zerocopy`[7] or `bytemuck`[8] can be used, hiding the unsafety behind a (hopefully soundly implemented) abstraction.

As seen in Section 5.2.1, the `Point` and `Vector` types in Retrofire are annotated with the `#[repr(transparent)]` attribute. This allows reinterpreting, for example, a reference of type `&VecN<3, Model>` as a `&[f32; 3]`, and the latter in

[7] `https://crates.io/crates/zerocopy`

[8] `https://crates.io/crates/bytemuck`

turn as a `&VecN<3, View>`. The same applies to slices: a `&[VecN<3, Model>]` can be turned into a `&[VecN<3, View>]`, allowing in-place mutation while retaining a type-safe interface. It also enables reinterpreting "flat" slices of plain `f32` values (as deserialized from storage, for example) as slices of `Point` or `Vector` values, adding a type-safe layer over raw data with zero runtime overhead. The same reinterpretation can be done in reverse, making it possible, for instance, to pass slices of typed values to a legacy API that expects a list of vectors in the form of a flat array of `f32`s.

The Retrofire renderer makes a small, constant number of allocations (normally five) per drawing call, independent of the size of the input (the number of primitives or vertices). This was verified by using the tracking allocator included in the `divan` benchmarking library. The per-call overhead could be further reduced by retaining and reusing temporary buffers from one call to another. Alternatively, allocations could be made considerably less expensive by using a custom allocator, something that is very common in high-performance software. For example, an extremely fast *bump allocator* could be employed to handle the temporary memory space required by the renderer.

The finding most pertinent to this thesis was that all of the allocations internal to the renderer are independent of the use of strongly-typed geometry. In other words, there are no "extra" allocations only needed due to the use of the geometry types. However, using Retrofire types in *custom* geometry manipulation routines might still result in unwanted memory operations unless optimization measures such as those mentioned before are taken.

6.1.7 Benchmarks

Two special versions of Retrofire were developed and their benchmark results compared to the main development branch. The first, in branch `untyped`[9], removes the coordinate frame type parameters from points, vectors, and matrices, and is thus similar to a traditional rendering API. The change was largely mechanical in nature and, as often happens, the renderer worked as expected as soon as the code successfully compiled.

In the second branch `simple-render`[10], additionally a simplified, less generic rasterizer was written to study abstraction overhead not directly related to the geometry API. This rasterizer hardcodes the primitive type (triangles) and render target (only basic 2D framebuffers) and has a more straightforward inner loop.

The `e2e::sphere` benchmark draws a mesh approximation of a sphere at five different levels of detail, each at the same pixel resolution. Each sub-benchmark was run in a loop for five seconds in order to minimize noise. The benchmark output is shown in Listing 6.3. It is apparent that the "untyped" version has performance essentially identical to the main branch. On the other hand, the simplified renderer is decidedly faster. Memory usage was identical in all three cases.

The conclusion is that the Retrofire renderer has abstraction penalty caused by the generic, customizable rasterizer, but this penalty does not appear related to the typed geometry API in particular. The root causes of the performance gap are currently unclear; one plausible suspect is missed autovectorization opportunities.

[9] `https://github.com/jdahlstrom/retrofire/tree/untyped`

[10] `https://github.com/jdahlstrom/retrofire/tree/simple-render`

Listing 6.3: Divan end-to-end benchmarks.

```
e2e          fastest   slowest    median     mean      samples    iters
 sphere_main
  4          168.3 us  217.9 us   176.4 us   178.8 us     280     28000
  16         217.4 us  405.6 us   229.5 us   235.3 us     213     21300
  64         355.5 us  394.8 us   368.9 us   370.2 us     136     13600
  256        885.1 us  979.7 us   925.7 us   927.7 us      54      5400
  1024       3.05 ms   3.333 ms   3.09 ms    3.109 ms      17      1700

 sphere_untyped
  4          169.1 us  215.6 us   177.6 us   180.1 us     278     27800
  16         217.3 us  298.1 us   229.0 us   230.8 us     217     21700
  64         358.1 us  594.7 us   372.3 us   379.4 us     132     13200
  256        907.8 us  967.2 us   927.9 us   930.9 us      54      5400
  1024       3.073 ms  3.149 ms   3.107 ms   3.111 ms      17      1700

 sphere_simple_render
  4          73.18 us  121.7 us   78.60 us   79.84 us     626     62600
  16         95.49 us  178.1 us   103.0 us   105.0 us     476     47600
  64         146.0 us  818.7 us   155.0 us   161.2 us     311     31100
  256        350.6 us  596.6 us   371.1 us   380.5 us     132     13200
  1024       1.213 ms  1.295 ms   1.248 ms   1.251 ms     40       4000
```

6.2 Programmer ergonomics

Programmer ergonomics, and the near synonym "developer experience" refer to the user experience (UX) of a programmer using libraries, languages, and development tools to write software (as opposed to the experience of the *end user*). It is the ease (or difficulty) of getting the program to "do what the programmer means". Ergonomics can be negatively impacted by, for instance, having to write large amounts of repetitive "boilerplate" code, to reason about hidden or global state, or to fight an inflexible API (e.g. one typed so rigidly as to disallow valid use cases) or opaque compiler errors. A systematic literature review of the topic is given by Morales et al. [102]

Errors such as mixing up coordinate spaces are likely to be particularly common among those new to graphics programming (see, for example, [8]). Strong typing can both help them write correct code and gain a better mental model of what is happening. On the other hand, complex types and bounds can be intimidating, and compiler errors resulting from mismatched types may be frustratingly opaque.

6.2.1 Methods

Few studies on the ergonomics or productivity of typed APIs exist in the literature. Studying the ergonomics of a novel API is nontrivial. For example, a programmer used as a test subject cannot be expected to be instantly productive with an unfamiliar interface, and it can take time for benefits such as bug prevention to become visible. A longitudinal study could reveal real effects, but is outside the scope of this work. As a less formal measure, the author has posted about Retrofire on Internet fora such as Reddit and the Rust Users forum, but as of this writing has not been able to garner feedback beyond "likes"

and GitHub stars.

The analysis in this section is largely based on the test programs written in the course of the development of Retrofire to demonstrate and validate its APIs, as well as on the development of higher-level Retrofire components like those showcased in Section 5.7. These have been quite helpful in the design work and have provided some degree of assurance that the API is useful in real use cases and can prevent real-world bugs. However, to truly put the library to test, it would be important to use it to develop a larger-scale application, such as a game. It would likely provide many additional insights and highlight awkward or missing APIs. As in Section 6.1, the less-strictly typed variant of Retrofire in the `untyped` branch was used as a point of comparison in some parts of the analysis.

6.2.2 Escape hatches

Because type checking is necessarily conservative, it is important to provide convenient ways to opt out of strict typing when needed. Retrofire aims to keep its abstractions transparent, allowing access to the raw data, and to provide flexible conversions between data types.

The limited set of operations afforded by `Affine` objects can be inconvenient. In many cases it may be preferable, or even necessary, to do calculations with vectors, even if the end result is a point or points. The methods `Point::to_vec()` and `Vector::to_pt()` can be used to freely convert between the types. Similarly, in calculations, a programmer may not want to worry about the exact types of the intermediate values – only the type of the result is relevant. Accordingly, the relevant type parameters in `Matrix`, `Point`, and `Vector` default to `()`, the unit type, denoting a "default" or "generic" frame. Each type has a

`to()` method used to *coerce* the receiver to another frame as needed. By using the "default" types, it is largely possible to use Retrofire as a traditional, more weakly-typed graphics library.

The difference between more and less strictly typed Retrofire code is illustrated in Appendix A.3. Listing A.1 shows a vertex shader written with the unparameterized types of the `untyped` branch. It is a close analogue to how the shader would be written in a traditional API. By relying on the defaulted type parameters, very similar code could also be written on the main branch if it also implemented some helpers such as the `to_vec4` method. The comments point out many potential pitfalls in the code. On the other hand, Listing A.2 displays the same shader, but fully parameterized. It is not much more verbose, but rules out all of the highlighted errors at compile time and is also more self-documenting and easier to reason about.

There are a few cases in the API in which the author has found that practicality trumps theoretical purity. One is the operation of multiplying a point by a scalar. This is not well defined in an arbitrary affine space, but in a Euclidean space in which a standard origin $\mathrm{O} = (0, \ldots, 0)$ exists, it can be defined simply as a shorthand for the the affine combination

$$x\mathrm{P} \stackrel{\text{def}}{=} x\mathrm{P} + (1 - x)\mathrm{O} = \mathrm{O} + x\,\overline{\mathrm{OP}}\,.$$

Another convenience operator supported by Retrofire is the component-wise multiplication of two vectors, also called a Hadamard product. This is a linear transform (a nonuniform scaling), but useful often enough to warrant a shorthand.

6.2.3 Type inference

Type inference is crucial when dealing with generic types with complex signatures. On the other hand, relying on type inference too much can result in code that is difficult to read and reason about. Modern development tools, providing affordances such as displaying inferred types, can be very helpful.

Phantom type parameters present a challenge to type inference because they do not have corresponding values from which to infer the type. For example, the constructor function `vec2` can return a vector in any frame – but what should the exact type of `v` be in code like `let v = vec2(1, 2)`? It is ambiguous unless explicitly annotated or can be inferred from how `v` is used in the code that follows.

Listing 6.4: An example of a type inference issue.

```
let model_rot = rotate_z(degs(45.0));
let model_trans = translate(vec3(1.0, 0.0, 2.0));

let camera_trans = translate(vec3(0.0, 0.0, 3.0));

let modelview: Mat4<Model, View> = model_rot      // Model to ???
    .then(&model_trans)                           //  ???  to ???
    .then(&camera_trans);                         //  ???  to View
```

One example of such a type inference issue is shown in Listing 6.4. Based on the type annotation of `modelview`, the compiler can infer that the source space of `model_rot` is `Model` and the target space of `cam_trans` is `View`, but the types of the intermediate spaces in the transform chain cannot be inferred. Even though the type parameters have defaults, they are not used to resolve ambiguities in Rust.

As a compromise, in Retrofire matrix constructor functions such as `rotate_z`

and `translate` return "generic" matrices of type `Mat4<(), ()>`, which must be coerced to a more specific type with the `to()` method as needed. This is a somewhat annoying "papercut", but the alternative – having to explicitly specify the intermediate types – would be worse. The projection and viewport matrix constructors, however, have the expected return types `Mat4<View, Proj>` and `Mat4<Ndc, Screen>` respectively.

Another fairly common – and admittedly annoying – case of inference failures is when code is commented out during experiments or debugging, removing some expression that the compiler relied on for inference. This can lead to a "chain reaction" as more and more of transitively unused code has to be commented out.

In practice, having to explicitly annotate phantom type parameters is surprisingly infrequent. Most often it seems to occur in unit tests, where less context is available to help inference than in most real-world code. Testing-specific monomorphic helper functions can be defined to mitigate the problem.

6.2.4 Build speed

One of the most common criticisms of Rust is its compilation times, which can be an order of magnitude longer than those in languages such as Java and Go. Graphics programming often requires a fast edit–compile–test cycle, and if each build takes ten seconds, it can quickly add up. Much work has gone to making the compiler faster, including implementing incremental compilation, better use of parallelism, and even writing a new compiler backend purely aimed at improving the compilation speed of development builds.[11]

[11] `https://cranelift.dev/`

Compounding this problem is the fact that the "zero-overhead" abstractions in Rust fully depend on the optimizer to actually attain the promise of no overhead, and unoptimized code typically carries a heavy performance penalty. Fully unoptimized builds of Retrofire are generally too slow to render complex scenes at interactive frame rates. Thus, even development builds have to be optimized, which further slows down the development loop. A user of Retrofire can alleviate this to an extent by building the library with full optimizations and their own, actively developed code without.

The duplication of code caused by monomorphization increases the amount of LLVM IR (intermediate representation) that the compiler emits and the backend must handle – even if most of the code ultimately ends up being optimized out [100, Ch. 19]. This is another drawback of generics. The Rust compiler produces rather naive and unoptimal IR, and relies on LLVM to make it fast and compact. The mitigation techniques described in 6.1.4 apply. A tool, `cargo llvm-lines`, can report the size of functions in IR, allowing pruning of duplicate code that has no effect on the binary but merely slows down compilation.

The demo programs bundled with retrofire (package `retrofire-demos`) were built from scratch with timings enabled in development and release modes, in both the main development branch and the `untyped` branch (Listing 6.5). Each build was repeated several times to reduce noise and "warm-up" effects. The logged timings indicated no meaningful difference between the main and `untyped` versions.

Listing 6.5: Commands to measure build time.

```
$ cargo clean && cargo build --package retrofire-demos --timings
$ cargo clean && cargo build --package retrofire-demos --timings --release
```

Ultimately, the most effective way to speed up compilation is to have less code, and perhaps the best way to achieve that is to reduce the number of the project's transitive dependencies and replace large libraries with leaner alternatives. The Retrofire core library in fact depends on no (non-optional) third-party packages at all.

6.2.5 Compiler diagnostics

Complex type signatures may make compiler diagnostics difficult to understand. Source code can be made cleaner and more succinct by appropriate use of type aliases to abbreviate verbose type names, but the compiler still uses the unaliased name in diagnostic messages. Sometimes the exact type signature contains relevant information, but often it is a mere distraction.

For example, the concise type alias `Vec3` is `Vector<[f32; 3], Real<3, ()>>` to the compiler, and that is what is displayed in diagnostic messages. In addition to making the messages noisier, the programmer must also expend some mental energy translating between the aliased and full names. In the `untyped` branch, the corresponding type is the more succinct `Vector<[f32; 3]>`, and in an even less generic library `Vec3` might simply be its own data type rather than an alias.

The use of closures can also make compiler messages more difficult to understand due to the way the compiler prints their (anonymous) types. A particularly egregious example of an initially incomprehensible compiler error is shown in Appendix A.2, with line breaks added by the author to make it fit on the page horizontally. Rustc even gives up printing a long type name and chooses to write it to a temporary file instead, which is, in a word, suboptimal.

However, such extreme cases are fortunately outliers. The rustc compiler is well known for its highly helpful diagnostics, and Retrofire contains no code such as complex macros or deeply nested type-level expression trees that typically cause issues with error messages and IDE support. Rustc is often able to highlight the exact mismatch even in complex types. Moreover, complex compiler errors like the one shown in Appendix A.2 do not appear to be significantly more concise in the `untyped` branch. They are simply less common (that is, fewer errors are caught by the compiler)!

6.2.6 Generic parameters and trait bounds

Generic types are "infectious" in the sense that any interface dealing with them must in turn be generic, transitively. Trait bounds must be repeated, and this can be a significant ergonomics hurdle if the bounds are complex. A user of a library may end up copy-and-pasting generic signatures without even fully understanding them, just to make their program compile. One technique to simplify trait bounds and make them more understandable is creating a new trait (hopefully with a descriptive name) and writing a generic *blanket implementation* for all types that satisfy the bounds. *Trait aliases* are a longstanding experimental feature in Rust that enable doing the same thing more succinctly [103].

There is a danger of overabstraction to the point where the types of most struct fields or function parameters are generic. The loss of concreteness can make interfaces difficult to understand, and the issue can be exacerbated by the convention of omitting trait bounds in a type definition if not strictly needed. For example, the standard library type `HashSet<T>` is not constrained by `T: Hash` – only those methods that actually need to hash a value of `T` include the bound.

This idiom affords flexibility and reduces the repetition of bounds, but comes at the expense of clarity.

Based on the type definitions in Listing 6.6 alone, it is not at all clear what types `Tri` and `Vertex` can be meaningfully instantiated with. A user new to the library will likely have to consult the documentation. The names give hints, but one of the points of strong typing is to *not* have to depend on naming!

Listing 6.6: The triangle and vertex types.

```
pub struct Tri<V>(pub [V; 3]);

pub struct Vertex<P, A> {
    pub pos: P,
    pub attrib: A,
}
```

The `Vertex`, `Point`, and `Color` types also leave the inner representation entirely generic, even though in practice only array-based types such as `Vector<[T; N]>` are used, at least at the time of this writing. However, there are some potential future use cases for this genericity, including:

- SIMD representation, such as `Vector<SimdF32x4>`; however, better parallelism is likely achieved with a type like `Vector<[SimdF32x8; 3]>`, storing eight vectors in one "vector of vectors" (an AoSoA layout).
- Borrowing from a larger array, such as using a `Vector<&[f32]>` as a "cursor" to a raw array of `f32`s; this also allows dynamically sized vectors.
- Bit-packing small component types into a single value that fits in a register, such as representing a vector of booleans with a `Vector<u8>`.

It is also not entirely clear whether being generic over the dimension is worth it in a 3D graphics API, where `N` is only ever either 2, 3, or 4. Having a single

generic implementation does, however, reduce duplication or near-duplication of code and thus helps ensure correctness and improves maintainability, and the author believes that these advantages justify the extra abstraction. One alternative would be to use Rust macros to generate the different implementations, but that solution comes with its own problems.

6.3 Summary of findings

This section briefly summarizes the answers found to the research questions.

Type system features (RQ1): The type-safe geometry types of Retrofire are implemented in terms of generics (parametric polymorphism) and traits (type classes). Tag (phantom) type parameters are used to distinguish objects in different spaces or coordinate frames, and const generic parameters are used to abstract over spaces of different dimensions. This combination is idiomatic and expressive, and other authors and library designers have independently converged to similar solutions. Some "type-level metaprogramming" techniques are applied, although in simple ways compared to some advanced C++ or Rust libraries. Macros are another metaprogramming technique heavily used by some Rust libraries, but the author did not see a good use for them in this work.

Ergonomics (RQ2): Retrofire provides flexibility by keeping its abstractions as lightweight and transparent as possible, mostly simple zero-overhead wrappers of fundamental types. It implements "escape hatches" that allow dropping down to less strictly-typed operations. This also enables "gradual typing", opting in to stricter type guarantees step by step.

In general, Retrofire works well with type inference, and the need to explicitly disambiguate a type is fairly rare. Compiler diagnostics arising from type mismatches are typically easy to interpret, but can be long and opaque in the worst case. One weakness in the API is that it can be difficult to understand how to concretely use some highly generic Retrofire types, even though optimally types should be guiding the programmer on the "happy path". The "infectiousness" of generic parameters and trait

bounds may also present a hindrance.

Build times are a concern shared by most Rust programmers. Retrofire abstractions are relatively lightweight, and do not involve slow-to-compile constructs such as complex macros. The core Retrofire API has no third-party dependencies, keeping the amount of code that needs to be compiled fairly minimal. Compared to an alternative, less strictly typed implementation, there is no detectable difference in compile times.

To attain a more complete understanding of the ergonomics of the Retrofire API in the future, it would be vital to attract developers interested in trying out the API and to gather feedback from them. In addition to fairly minor test and demo programs, Retrofire should also be used in a larger-scale, moderately complex application such as a game.

Performance (RQ3): Unlike runtime polymorphism, parametric polymorphism and phantom types enable abstraction while permitting compact types and data locality (enabling efficient data access patterns) and function inlining (a major "force multiplier" enabling a host of other optimizations). Retrofire code appears to be highly inlineable, and the compiler is able to produce very compact code in spite of the abstraction. Benchmarks indicate that using Retrofire types incurs no meaningful runtime performance cost compared to the less strictly typed variant.

The more fine-grained typing can make it tricky to manipulate geometry objects in place. An analysis verified that the Retrofire renderer makes a small, constant number of allocations per drawing call. Allocator overhead could be further reduced, but pertinently there were no "extra" allocations required solely by the use of strongly-typed geometry. However,

using the Retrofire types in application code may still require extra copies and allocations if done “naively”.

A topic that merits further study in the future is vectorization. Use of low-level SIMD intrinsics is difficult and unsafe, and in the case of renderer with customizable shaders, SIMD use “leaks” to the API and cannot remain merely an implementation detail. The design choices of Retrofire make the code amenable to automatic vectorization by the optimizer; however, autovectorization can still be unreliable or suboptimal for many reasons. It would be worthwhile to study how the Retrofire API could interoperate with higher-level, portable SIMD libraries.

6.4 Final thoughts and future directions

Affine transforms can be subdivided into categories such as rigid motion, isometric, linear, and reflection. Specific transforms can be stored and computed more efficiently than a full 4×4 matrix. In particular, representing *rigid motions* (rotations and translations) is often sufficient. The Nalgebra library [80] implements such typed transforms that add a layer of abstraction over the underlying representation. 3D rotations especially have several different representations, such as Euler angles, axis–angles pairs, and quaternions, and which of them is the most convenient depends on the use case [13]. Additionally, as discussed by Lee [6], orientations and rotations have the same affine–linear relationship as points and vectors, and as such should perhaps be represented by separate types.

In many real-world applications a single local frame per model is not enough; objects can be composed of several individually moving parts. This necessitates a *hierarchy* of local frames and transforms. One example is the transform of the front wheel of a car. It is the composition of the spinning motion of the wheel around its axis, the rotation of the steering mechanism, the vertical motion of the suspension, the position of the entire assembly relative to the car body, and the world position and orientation of the car itself. Retrofire does not currently have explicit support for hierarchical transforms; there is only one built-in "model space" frame. However, users are free to add arbitrary custom frame types, such as `Wheel`, `Steering`, and `Body`.

A question worth asking is whether a CPU-based solution is relevant in a world where hardware-accelerated rendering is pervasive. On the other hand, the traditional untyped interfaces between a host application and shaders are clearly error-prone, requiring data such as uniform variables to be matched by

numeric indices or textual identifiers. One answer is SPIR-V [104], a standard intermediate representation language that allows one to write GPU shaders in any language for which a compiler to the IR exists. The RustGPU [105] project implements such a translator for Rust, enabling the same types to be used seamlessly on both sides of the bus.

There are related programming disciplines concerned with geometric spaces and transforms. These include physics solvers and simulations, robotics, and geoinformation systems (GIS). Physics problems often deal with moving objects in different frames of reference; robot arms and other manipulators may have complex hierarchies of nested coordinate frames; and GIS software packages often have to work with and convert between numerous different geographical coordinate systems. A Retrofire-like system of strongly-typed geometric objects could plausibly find use in these applications as well.

7 Conclusion

In this thesis, several sources of potential programming errors were identified that relate to how 3D geometry is typically represented and manipulated in 3D graphics. These include the lack of distinction between affine and linear objects (points and vectors), the lack of explicit representation of different coordinate frames, and others. Relatively little has been written on the topic of trying to automatically detect such errors either at runtime or at compile time. Runtime-based approaches are likely not considered practical due to performance concerns. Some practical implementations that utilize types exist, but they are not in common use.

The thesis introduced Retrofire, a software 3D rendering library written in Rust, with a strongly-typed API designed to mitigate these common programming errors while trying to incur minimal performance overhead over an equivalent conventional implementation. Retrofire employs generic types, traits, and phantom types to help solve the aforementioned problems at compile time, with minimal or no runtime overhead: in general, no additional metadata is stored in values, and no runtime branching or indirection is required. Benchmarks indicate that these are truly "zero-cost" abstractions; no noticeable overhead in build or runtime speed was observed compared to a more traditionally typed version. Still, there are aspects of performance that should be focused on in

future development. These include making better use of SIMD vectorization and further reducing memory copying and allocations.

Ergonomics, or developer experience, is another important aspect in determining whether an abstraction is practical in real-world use. Retrofire strives for flexibility by keeping its abstractions lightweight and transparent, providing simple ways to opt out of strict typing if necessary. It appears to work well with type inference, and compiler diagnostics are usually, though not always, comprehensible. However, more research is required for attaining an in-depth understanding of Retrofire ergonomics. It would be important to gather feedback from other programmers willing to test the API, and to use it to develop an application, such as a game, more complex and broader in scale than the existing demo programs that ship with Retrofire.

Some future improvements to the geometry API include a finer subdivision of different types of affine transforms, as well as explicit support for hierarchical transforms (nested relative coordinate frames), required for modeling many real-world objects made of parts that move in relation to each other. Retrofire code could be compiled to be run on the GPU via an intermediate representation such as SPIR-V, removing the need for a separate shader language and enabling a type-safe interface between CPU and GPU code.

The typed geometry API presented in this thesis could be be generalized to other related programming disciplines. Physics simulations, robotics, and geoinformation systems all deal with presenting and manipulating geometric objects defined in different coordinate frames. It is apparent that the benefits seen in graphics programming also apply in these fields.

References

[1] B. Pierce, *Types and Programming Languages*. MIT Press, 2002, ISBN: 9780262303828.

[2] G. Dévai, Z. Gera, and Z. Kelemen, "Language abstractions for low level optimization techniques", *Computer Science and Information Systems*, vol. 11, pp. 1499–1514, Jan. 2014. DOI: 10.2298/csis130224080d.

[3] A. Fog. "Optimizing software in C++". (2024), [Online]. Available: https://www.agner.org/optimize/optimizing_cpp.pdf (visited on 05/06/2026).

[4] "Rust Programming Language", The Rust Team. (n.d.), [Online]. Available: https://rust-lang.org/ (visited on 05/06/2026).

[5] J. Corbet, "The state of the kernel rust experiment", *LWN.net*, 2026. [Online]. Available: https://lwn.net/Articles/1050174/.

[6] J. Lee, "Representing rotations and orientations in geometric computing", *IEEE Computer Graphics and Applications*, vol. 28, no. 2, pp. 75–83, 2008. DOI: 10.1109/MCG.2008.37.

[7] D. Geisler, I. Yoon, A. Kabra, H. He, Y. Sanders, and A. Sampson, "Geometry types for graphics programming", *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428241.

[8] K. Chandra. "Systems for coordinating coordinate systems". (2020), [Online]. Available: https://hardmath123.github.io/systems-for-coordinating-coordinate-systems.html (visited on 05/06/2026).

[9] S. Sylvan. "Naming convention for matrix math". (2017), [Online]. Available: https://www.sebastiansylvan.com/post/matrix_naming_convention/ (visited on 05/06/2026).

[10] "Debugging coordinate transformations", Apple Inc. (n.d.), [Online]. Available: https://developer.apple.com/documentation/technotes/tn3124-debugging-coordinate-transformations (visited on 05/06/2026).

[11] T. D. DeRose, "A coordinate-free approach to geometric programming", in *Theory and Practice of Geometric Modeling*. Springer-Verlag, 1989, pp. 291–305, ISBN: 0387514724.

[12] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering*, 4th ed. CRC Press, 2018, ISBN: 9781138627000.

[13] J. F. Hughes, A. van Dam, M. McGuire, *et al.*, *Computer graphics: principles and practice (3rd ed.)* Boston, MA, USA: Addison-Wesley Professional, Jul. 2013, p. 1264, ISBN: 0321399528.

[14] P. J. Schneider and D. H. Eberly, *Geometric tools for computer graphics*. Morgan Kaufmann Publishers, 2003, ISBN: 1-55860-594-0.

[15] J. Vince, *Mathematics for Computer Graphics* (Undergraduate Topics in Computer Science), 6th ed. Springer London, Limited, 2022, ISBN: 9781447175193.

[16] R. Goldman, "The ambient spaces of computer graphics and geometric modeling", *IEEE Computer Graphics and Applications*, vol. 20, no. 2, pp. 76–84, 2000. DOI: 10.1109/38.824547.

[17] R. Goldman, "Illicit expressions in vector algebra", *ACM Trans. Graph.*, vol. 4, no. 3, pp. 223–243, Jul. 1985, ISSN: 0730-0301. DOI: 10.1145/282957.282969.

[18] L. G. Roberts, "Machine perception of three-dimensional solids", Ph.D. dissertation, Massachusetts Institute of Technology, 1963.

[19] I. Carlbom and J. Paciorek, "Planar geometric projections and viewing transformations", *ACM Comput. Surv.*, vol. 10, no. 4, pp. 465–502, Dec. 1978, ISSN: 0360-0300. DOI: 10.1145/356744.356750.

[20] R. F. Riesenfeld, "Homogeneous coordinates and projective planes in computer graphics", *IEEE Computer Graphics and Applications*, vol. 1, no. 1, pp. 50–55, Jan. 1981, ISSN: 0272-1716. DOI: 10.1109/MCG.1981.1673814.

[21] D. Anderson and J. Delve, "Biographies [f.c. williams; j. vaucanson; j.m. jacquard]", *IEEE Annals of the History of Computing*, vol. 29, no. 4, pp. 90–102, 2007. DOI: 10.1109/MAHC.2007.4407450.

[22] J. Belcher, "The evolution of computer displays", *Ars Technica*, 2011. [Online]. Available: https://arstechnica.com/gadgets/2011/01/the-evolution-of-computer-displaysthe-evolution-of-computer-displays/.

[23] D. Ryan, *History of computer graphics: DLR associates series*. Author House, 2011, ISBN: 9781456751166.

[24] C. Wylie, G. Romney, D. Evans, and A. Erdahl, "Half-tone perspective drawings by computer", in *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, ser. AFIPS '67 (Fall), Anaheim, California: Association for Computing Machinery, 1967, pp. 49–58, ISBN: 9781450378963. DOI: 10.1145/1465611.1465619.

[25] J. F. Blinn and M. E. Newell, "Clipping using homogeneous coordinates", *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 245–251, Aug. 1978, ISSN: 0097-8930. DOI: 10.1145/965139.807398.

[26] I. Sutherland and G. W. Hodgman, "Reentrant polygon clipping", *Communications of the ACM*, vol. 17, pp. 32–42, 1974. DOI: 10.1145/360767.360802.

[27] R. L. Cook, L. Carpenter, and E. Catmull, "The Reyes image rendering architecture", in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87, New York, NY, USA: Association for Computing Machinery, 1987, pp. 95–102, ISBN: 0897912276. DOI: 10.1145/37401.37414.

[28] P. Hanrahan and J. Lawson, "A language for shading and lighting calculations", in *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '90, Dallas, TX, USA: Association for Computing Machinery, 1990, pp. 289–298, ISBN: 0897913442. DOI: 10.1145/97879.97911.

[29] *Commission internationale de l'Eclairage proceedings*, International Commission on Illumination (CIE), Cambridge University Press, 1932.

[30] T. Porter and T. Duff, "Compositing digital images", *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 253–259, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808606. [Online]. Available: https://doi.org/10.1145/964965.808606.

[31] J. Pineda, "A parallel algorithm for polygon rasterization", in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '88, New York, NY, USA: Association for Computing Machinery, 1988, pp. 17–20, ISBN: 0897912756. DOI: 10.1145/54852.378457.

[32] L. Cardelli, "Type systems", *ACM Comput. Surv.*, vol. 28, no. 1, pp. 263–264, Mar. 1996, ISSN: 0360-0300. DOI: 10.1145/234313.234418.

[33] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation", *Journal of functional programming*, vol. 23, no. 5, pp. 552–593, 2013. DOI: 10.1017/S095679681300018X.

[34] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, Dec. 1985, ISSN: 0360-0300. DOI: 10.1145/6041.6042.

[35] C. Strachey, "Fundamental concepts in programming languages", *Higher-order and symbolic computation*, vol. 13, pp. 11–49, 2000.

[36] R. Milner, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978, ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4.

[37] J. R. Hindley, "The principal type-scheme of an object in combinatory logic", *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969.

[38] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc", in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 60–76.

[39] K. Chen, P. Hudak, and M. Odersky, "Parametric type classes", in *ACM Sigplan Lisp Pointers*, vol. V, Jan. 1992, pp. 170–181. DOI: 10.1145/141478.141536.

[40] S. Peyton Jones, M. Jones, and E. Meijer, "Type classes: An exploration of the design space", Microsoft, Tech. Rep., 1997.

[41] D. Walker, "Substructural type systems", in *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. MIT Press, 2024, pp. 3–43.

[42] L. Cardelli, "Typeful programming", in *Formal Description of Programming Concepts*. Springer–Verlag, 1991, ISBN: 9783540539612.

[43] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Prentice Hall PTR, 2008, ISBN: 0132350882.

[44] C. Thompson, "How Rust went from a side project to the world's most-loved programming language", *MIT Technology Review*, 2023. [Online]. Available: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/ (visited on 05/06/2026).

[45] S. Klabnik, C. Nichols, and C. Krycho, *The Rust Programming Language*, 2nd ed. No Starch Press, 2022, ISBN: 9781718503106.

[46] A. Azevedo de Amorim, C. HriŢcu, and B. C. Pierce, "The meaning of memory safety", in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds., Springer International Publishing, 2018, pp. 79–105, ISBN: 978-3-319-89722-6.

[47] H.-J. Boehm, "Threads cannot be implemented as a library", HP Laboratories, Tech. Rep., 2004. DOI: 10.1145/1064978.1065042.

[48] "What is memory safety and why does it matter?", Internet Security Research Group. (n.d.), [Online]. Available: https://www.memorysafety.org/docs/memory-safety/ (visited on 05/06/2026).

[49] S. Sylvan. "Garbage collection thoughts". (2012), [Online]. Available: https://sebastiansylvan.wordpress.com/2012/12/01/garbage-collection-thoughts/ (visited on 05/06/2026).

[50] S. Sylvan. "On GC in Games". (2013), [Online]. Available: https://www.sebastiansylvan.com/post/on-gc-in-games-response-to-jeff-and-casey/ (visited on 05/06/2026).

[51] "RAII". (n.d.), [Online]. Available: https://en.cppreference.com/w/cpp/language/raii.html (visited on 05/06/2026).

[52] *The Rustonomicon*, The Rust Team, n.d. [Online]. Available: https://doc.rust-lang.org/nomicon/ (visited on 05/06/2026).

[53] D. Leijen and E. Meijer, "Domain specific embedded compilers", *ACM Sigplan Notices*, vol. 35, no. 1, pp. 109–122, 1999.

[54] D. Racordon, E. Flesselle, and C. N. Pham, "On the state of coherence in the land of type classes", *The Art, Science, and Engineering of Programming*, vol. 10, 2025.

[55] *The Rust Reference*, The Rust Team, n.d. [Online]. Available: https://doc.rust-lang.org/stable/reference/ (visited on 05/06/2026).

[56] N. Matsakis. "Little orphan impls". (2015), [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2015/01/14/little-orphan-impls/ (visited on 05/06/2026).

[57] R. Goldman, "On the algebraic and geometric foundations of computer graphics", *ACM Transactions on Graphics*, vol. 21, pp. 52–86, 2002.

[58] S. Mann, N. Litke, and T. DeRose, "A coordinate free geometry ADT", University of Waterloo, Tech. Rep., 1997.

[59] J. Ou and F. Pellacini, "SafeGI: Type checking to improve correctness in rendering system implementation", *Computer Graphics Forum*, vol. 29, no. 4, pp. 1269–1277, 2010. DOI: 10.1111/j.1467-8659.2010.01722.x.

[60] A. Sampson, "Let's Fix OpenGL", in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, ser. Leibniz International Proceedings in Informatics, vol. 71, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 14:1–14:12, ISBN: 978-3-95977-032-3. DOI: 10.4230/LIPIcs.SNAPL.2017.14.

[61] E. Chen, J. Chang, and Y. Zhu, "Coolerspace: A language for physically correct and computationally efficient color programming", *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. DOI: 10.1145/3689741.

[62] Y. He. "Coordinate systems", W3C GPU for the Web Community Group. (2019), [Online]. Available: https://github.com/gpuweb/gpuweb/issues/416 (visited on 05/06/2026).

[63] A. Shavit. "Affine space types". (2018), [Online]. Available: http://videocortex.io/2018/Affine-Space-Types/ (visited on 05/06/2026).

[64] Mars Climate Orbiter Mishap Investigation Board, *Phase I Report*, NASA, 1999. [Online]. Available: https://llis.nasa.gov/lesson/641 (visited on 05/06/2026).

[65] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 4th ed. The MIT Press, 2023, ISBN: 9780262048026.

[66] R. F. Cmelik and N. H. Gehani, "Dimensional analysis with C++", *IEEE Software*, vol. 5, no. 3, pp. 21–27, Mar. 1988, ISSN: 1937-4194. DOI: 10.1109/52.2021.

[67] A. Kennedy, "Dimension types", in *Programming Languages and Systems — ESOP '94*, D. Sannella, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 348–362, ISBN: 978-3-540-48376-2.

[68] W. Brown, "Introduction to the SI library of unit-based computation", Fermi National Accelerator Laboratory, Tech. Rep., 1998.

[69] M. Pusz *et al.* "Quantities and units library", JTC1/SC22/WG21 - The C++ Standards Committee. (2024), [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3045r0.html (visited on 05/06/2026).

[70] "DirectXMath", Microsoft. (2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/dxmath/directxmath-portal (visited on 05/06/2026).

[71] "Unreal C++ API Reference", Epic Games. (2025), [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/API (visited on 05/06/2026).

[72] "Scripting API", Unity. (2025), [Online]. Available: https://docs.unity3d.com/ScriptReference/index.html (visited on 05/06/2026).

[73] "Godot Docs – 4.4 branch", Godot. (2025), [Online]. Available: https://docs.godotengine.org/en/4.4/ (visited on 05/06/2026).

[74] "OpenGL shading language", Khronos Group. (2021), [Online]. Available: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language (visited on 05/06/2026).

[75] "High-level shading language (HLSL)", Microsoft. (2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl (visited on 05/06/2026).

[76] "Templatized Geometry Library". (2006), [Online]. Available: https://web.cs.ucdavis.edu/~okreylos/ResDev/Geometry/MainPage.html (visited on 05/06/2026).

[77] "Eigen". (n.d.), [Online]. Available: https://libeigen.gitlab.io/ (visited on 05/06/2026).

[78] "Euclid", The Servo Project Developers. (n.d.), [Online]. Available: https://docs.rs/euclid/ (visited on 05/06/2026).

[79] "Alga", Dimforge. (n.d.), [Online]. Available: https://docs.rs/alga/ (visited on 05/06/2026).

[80] "Nalgebra", Dimforge. (n.d.), [Online]. Available: https://docs.rs/nalgebra/ (visited on 05/06/2026).

[81] "Sguaba", Helsing. (n.d.), [Online]. Available: https://docs.rs/sguaba/ (visited on 05/06/2026).

[82] M. C. Schabel and S. Watanabe. "Chapter 41. Boost.Units 1.1.0". (n.d.), [Online]. Available: https://www.boost.org/doc/libs/latest/doc/html/boost_units.html (visited on 05/06/2026).

[83] N. Holthaus. "Unit conversion and dimensional analysis library". (n.d.), [Online]. Available: https://nholthaus.github.io/units/ (visited on 05/05/2026).

[84] M. Pusz. "Mp-units". (n.d.), [Online]. Available: https://mpusz.github.io/mp-units/latest/ (visited on 05/05/2026).

[85] M. Boutin. "Units of measurement". (n.d.), [Online]. Available: https://docs.rs/uom/0.37.0/uom/index.html (visited on 05/06/2026).

[86] "Units of measurement", Microsoft. (2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure (visited on 05/06/2026).

[87] K. I. Joy. "Bernstein polynomials". (2000), [Online]. Available: https://www.academia.edu/download/49392061/Bernstein-Polynomials.pdf (visited on 05/06/2026).

[88] "Rust API Guidelines", The Rust Team. (n.d.), [Online]. Available: https://rust-lang.github.io/api-guidelines/about.html (visited on 05/06/2026).

[89] D. Bakhvalov, *Performance analysis and tuning on modern CPUs*, 2nd ed. self-published, 2024, ISBN: 979-8869584229.

[90] J. Dean and S. Ghemawat. "Performance hints". (2025), [Online]. Available: https://abseil.io/fast/hints.html (visited on 05/06/2026).

[91] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious", *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, ISSN: 0163-5964. DOI: 10.1145/216585.216588. [Online]. Available: https://doi.org/10.1145/216585.216588.

[92] D. Efnusheva, A. Cholakoska, and A. Tentov, "A survey of different approaches for overcoming the processor-memory bottleneck", *International Journal of Computer Science and Information Technology*, vol. 9, no. 2, pp. 151–163, 2017.

[93] A. Sharp. "Memory layout transformations", Intel Corporation. (2019), [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html (visited on 05/06/2026).

[94] N. Llopis, "Data-oriented design (or why you might be shooting yourself in the foot with OOP)", *Game Developer Magazine*, Sep. 2009. [Online]. Available: https://gamesfromwithin.com/data-oriented-design (visited on 05/06/2026).

[95] Y. Sharvit, *Data-oriented programming: reduce software complexity*, eng, [First edition]. Manning Publications Co., 2022, ISBN: 9781638356783.

[96] *Automatic vectorization*, Intel Corporation, n.d. [Online]. Available: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2025-2/overview.html (visited on 05/06/2026).

[97] R. Levien. "Towards fearless SIMD, 7 years later". (2025), [Online]. Available: https://linebender.org/blog/towards-fearless-simd/ (visited on 05/06/2026).

[98] S. Byrne. "Beware of fast-math", NVIDIA. (2021), [Online]. Available: https://simonbyrne.github.io/notes/fastmath/ (visited on 05/06/2026).

[99] M. Pedersen. "Imprecise floating point operations (fast-math) #21690". (2015), [Online]. Available: https://github.com/rust-lang/rust/issues/21690 (visited on 05/06/2026).

[100] N. Nethercote. "The rust performance book". (n.d.), [Online]. Available: https://nnethercote.github.io/perf-book/title-page.html (visited on 05/06/2026).

[101] F. Allen and J. Cocke, "A catalogue of optimizing transformations", IBM Thomas J. Watson Research Center, Tech. Rep., 1971.

[102] J. Morales, C. Rusu, F. Botella, and D. Quiñones, "Programmer experience: A systematic literature review", *IEEE Access*, vol. 7, pp. 71 079–71 094, 2019. DOI: 10.1109/ACCESS.2019.2920124.

[103] "The Rust Unstable Book", The Rust Team. (n.d.), [Online]. Available: https://doc.rust-lang.org/unstable-book/ (visited on 05/06/2026).

[104] "SPIR-V: The standard IR for parallel compute and graphics", Khronos Group. (n.d.), [Online]. Available: https://www.khronos.org/spirv/ (visited on 05/06/2026).

[105] "Rust-gpu", Rust GPU. (n.d.), [Online]. Available: https://github.com/rust-gpu/rust-gpu (visited on 05/06/2026).

Appendix A Code listings

A.1 A simple example program using Retrofire

```
use re::prelude::*;
use re::util::pnm::save_ppm;

fn main() {
  // Define a triangle with each vertex of different color.
  // Vertex type is inferred: Vertex<Point3<Model>, Color3f<Rgb>>
  let verts = [
    vertex(pt3(-0.8, -1.0, 0.0), rgb(1.0, 0.0, 0.0)),
    vertex(pt3( 0.8, -1.0, 0.0), rgb(0.0, 0.8, 0.0)),
    vertex(pt3( 0.0,   1.5, 0.0), rgb(0.4, 0.4, 1.0)),
  ];

  let shader = shader::new(
    // Vertex shader. The model-to-projection matrix is passed as the
    // uniform parameter. Rust requires explicit type annotations here.
    |v: Vertex3<_>, mvp: &Mat4<Model, Proj>| {
      // Transforms the vertex position, passes the color through
      // unchanged.
      vertex(mvp.apply(&v.pos), v.attrib)
    },
    // Trivial fragment shader. Simply converts the floating-point
    // color to 8-bit format for output.
    |frag: Frag<Color3f>| frag.var.to_color4(),
  );

  // Framebuffer size in pixels.
  let (w, h) = (640, 480);
```

```
  // Move the triangle away from the origin (camera position)
  // so it's visible. The type is inferred: Mat4<Model, View>.
  let modelview = translate3(0.0, 0.0, 2.0).to();

  // Set up perspective projection with focal ratio 1 (90 degree FoV),
  // aspect ratio w / h, and the near and far planes at 0.1 and 1000,
  // respectively.
  let project = perspective(1.0, w as f32 / h as f32, 0.1..1000.0);

  // Setup the viewport to span the whole buffer.
  let viewport = viewport(pt2(0, 0)..pt2(w, h));

  // Create a 2D memory buffer to function as the framebuffer.
  let mut framebuf = Buf2::<Color3>::new((w, h));

  render(
    // Render a single triangle...
    [Tri([0, 1, 2])],
    // ...with the given vertices.
    verts,
    &shader,
    // The uniform passed to the vertex shader.
    &modelview.then(&project),
    viewport,
    &mut framebuf,
    // Default rendering configuration.
    &Context::default(),
  );

  // Save the image as a PPM file.
  save_ppm("triangle.ppm", framebuf).unwrap();
}
```

A.2 A long, cryptic compiler error

```
    error[E0277]: the trait bound `Shader<{closure@crates.rs:27:9},
        {closure@crates.rs:31:9}>: Shader<Vertex<..., ...>, ..., ()>`
        is not satisfied
   --> demos/src/bin/crates.rs:109:25
    |
109 |                .shader(crate_shader)
    |                 ------ ^^^^^^^^^^^^ unsatisfied trait bound
    |                 |
    |                 required by a bound introduced by this call
    |
    = help: the trait `VertexShader<Vertex<_, _>, ()>` is not implemented
        for `retrofire_core::render::shader::Shader<
            {closure@demos/src/bin/crates.rs:27:9: 27:57},
            {closure@demos/src/bin/crates.rs:31:9: 31:30}>`
        but trait `VertexShader<Vertex<_, _>, &Matrix<[[f32; 4]; 4],
            RealToProj<Model>>>` is implemented for it
    = help: for that trait implementation, expected
        `&Matrix<[[f32; 4]; 4], RealToProj<Model>>`, found `()`
    = note: required for `retrofire_core::render::shader::Shader<
        {closure@demos/src/bin/crates.rs:27:9: 27:57},
        {closure@demos/src/bin/crates.rs:31:9: 31:30}>`
    to implement `retrofire_core::render::Shader<
        Vertex<retrofire_core::math::Point<[f32; 3], Real<3, Model>>,
            Vector<[f32; 3], Real<3>>>,
        retrofire_core::math::Color<[f32; 3], Rgb>, ()>`
note: required by a bound in `Batch::<Vtx, Uni, Shd, Tgt, Ctx>::shader`
   --> [...]retrofire/core/src/render/batch.rs:94:31
    |
94  |     pub fn shader<V: Vary, S: Shader<Vtx, V, Uni>>(
    |                               ^^^^^^^^^^^^^^^^^^^
        required by this bound in `Batch::<Vtx, Uni, Shd, Tgt, Ctx>::shader`
    = note: the full name for the type has been written to
        '[...]retrofire/target/release/deps/crates-b0f19617091c64f9.long-
        type-14995697767821216909.txt'
    = note: consider using `--verbose` to print the full type name
        to the console
```

A.3 A vertex shader written in two ways

Listing A.1: An untyped shader

```
struct Uniform {
  light_pos: Vec3,
  modelview: Mat4,
  normal: Mat4,
  project: Mat4,
}
struct Varying {
  normal: Vec4,
  light_dir: Vec4,
}
fn vertex_shader(v: Vertex<Vec3, Vec3>, uni: Uniform)
    -> Vertex<Vec3, Varying>
{
  // Hopefully pos is now in view space, no guarantees except the name
  // of the matrix. Remember that the homogeneous coordinate must be 1.
  let pos: Vec4 = uni.modelview.apply(&v.pos.to_vec4(1.0));

  // Is this correct? Yes if light_pos is already given in view space
  // but no if it's in world space (or the light's own local space...)
  let light_dir: Vec3 = uni.light_pos.to_vec4(1.0) - pos;

  // Pos is now in clip space - if project actually is a view-to-
  // project transform and if pos was actually in view space.
  let pos: Vec4 = uni.project.apply(&pos);

  // Where does the normal matrix take the normal?
  // Is it the frame that light_pos is given in?
  // Remember that the homogeneous coordinate must be 0
  let n: Vec4 = uni.normal.apply(&v.attrib.to_vec4(0.0));

  // pos should now be in clip space and the normal and light_dir
  // passed to the fragment shader in the same frame. Are they?
  Vertex { pos, attrib: Varying { normal: n, light_dir } }
}
```

Listing A.2: A typed shader

```
struct Uniform {
  light_pos: Point3<View>,
  modelview: Mat4<Model, View>,
  normal: Mat3<Model, View>,
  project: Mat4<View, Proj>,
}
// It is clear to the programmer and enforced by the compiler
// that the varyings are compatible.
struct Varying {
  normal: Normal3<View>,
  light_dir: Vec3<View>,
}

type VertIn = Vertex<Point3<Model>, Normal3<Model>>;
type VertOut = Vertex<Point3<Proj>, Varying>;

fn vertex_shader(v: VertIn, uni: Uniform) -> VertOut {
  // No explicit conversion to homogeneous needed
  let pos = uni.modelview.apply(&v.pos);

  // Vec3<View> = Point3<View> - Point3<View>: ok
  let light_dir = uni.light_pos - pos;

  // pos is in View and project only accepts points in View: ok
  let pos: Point3<Proj> = uni.project.apply(&pos);

  let n: Normal3<View> = uni.normal.apply(&v.attrib);

  // pos is known to be in Proj, n and light_dir in View: ok
  Vertex { pos, attrib: Varying { normal: n, light_dir } }
}
```