Practical Course Report: WasaBii

Cameron Reuschel, David Schantz

Julius-Maximilians-Universität Würzburg
reuschel@bii-gmbh.com
schantz@bii-gmbh.com

1 Introduction

The BII GmbH, founded in 2018, specializes in the topic of interactive construction process planning [4]. It provides many internships to students of the Julius-Maximilians-Universität Würzburg. Ever since its foundation, developers have been writing internal utilities and systems in order to speed up development and provide abstractions for common patterns. Previous interns often grow accustomed to these, and there have been many requests to release them for use in other personal, academic and commercial projects.

WasaBii is the result of these requests: an open source library, provided under the Apache 2.0 license [2], which includes most of the general purpose utilities and systems that people have been grown used to. Most of the code in WasaBii has been either polished or entirely rewritten in order to adhere to our high standard for open source code. This project is a direct evolution of the previously released *CoreLibrary* [7].

The WasaBii package focuses on work in the Unity engine [8], but most of the utilities and systems can also be used in any other dotnet projects without any additional dependencies. We prioritized type safety, discoverability and extensibility over low level performance.

The remainder of this report provides an overview of the available utilities and systems in WasaBii.

2 Project Structure

WasaBii is developed as an embedded Unity package [10] with nested packages. The project is hosted on GitHub [11]. The packages can be imported directly into Unity by providing an appropriate GitHub URL.

The "Full" package includes all other packages and can be conveniently imported as a package into Unity via the GitHub URL.

The "Core" package includes common core utilities which are used across all other packages.

The "Undo" package includes a fully fledged, standalone undo system that can easily be used in any type of C# or Unity project.

The "Units" package includes a full set of customizable, typesafe and lightweight unit wrappers for floating point numbers, with special compatibility when used in a Unity project.

The "Geometry" package includes typesafe and lightweight wrappers for vectors and quaternions, which explicitly differentiate between local and global values as well as between points, offsets and directions.

The "Splines" package includes typesafe and efficient yet generalized implementations of both Bézier splines as well as Catmull-Rom splines that can be used with any type of handle.

The "Unity" package includes a diverse set of utilities specifically for use in the Unity engine. It also contains extensions and utilities for using the other packages inside of Unity projects. This is the only package that cannot be used in a non-Unity C# project.

The "Extra" package includes more niche utilities, such as monads for modelling computations with a progress bar over time, and depends on all other packages.

The remainder of this report will provide an overview of the most interesting features of WasaBii, in a logical order that does not directly relate to to their grouping in the respective packages.

3 Larger Standalone Systems

WasaBii includes a number of larger standalone systems. These systems are mostly designed to be independent of one another. Each system has its own sub-package. These packages only depend on the core package and potentially the Units system.

3.1 Undo

Undo and Redo are common features in professional interactive software (Photoshop[1], Blender[3], Unity[8], dProB[4]). Implementing consistent undo and redo functionality in an application requires the use of a common undo system.

WasaBii provides an UndoManager class, which is generic on a label type. The labels can be used to visualize individual undoable and redoable actions in an undo history, and the type itself is unconstrained. Some applications might want a simple text label, others an additional description, and even other applications might require an image for each undoable step.

The system distinguishes between *operations* and *actions*: An action is what the user undoes or redoes on the push of a button, and it has a label. An operation describes an atomic unit of state modification that can be undone and redone. An action is therefore composed using a stack of operations.

When the user does something undoable, then the first step is to start recording an action. All operations registered from now are pushed onto a local stack. When the recording is stopped, the local stack is wrapped in an undo action, and this action is pushed onto the undo stack. When the user undoes, the top action is popped from the undo stack, the undo functions on the local stack are executed in order, and then a new action with an inverted local stack is pushed to the redo stack. When a new action is recorded, the redo stack is cleared.

The undo stack has a customizable maximum size, and when a new undo action is recorded, the oldest undo that would exceed the stack size is removed automatically to prevent memory leaks.

The undo system also provides many features to solve problems occurring in practice. A programmer can insert a placeholder into the undo stack, and later register an operation where the undo is inserted into the placeholder slot instead of at the top of the stack. The system also tracks a stack of undo buffers that allow temporary, more fine-grained local undo contexts. When a buffer is pushed, it temporarily overrides the undo history until popped. When popped, the programmer can use a callback in the buffer implementation to for example group all recorded actions into one large action, or discard them, or whatever else is required.

For all operations, the undo manager also provides events so that external systems such as UI can update themselves reactively. These events are not technically necessary, but make it much easier for programmers with less experience to integrate the undo system into their applications. The alternative to using these events would involve a proper layered software architecture which calls the undo manager methods and then updates all relevant dependencies in the appropriate order.

For registering operations, there are two different but intercompatible approaches:

The iterative approach allows calling *undoManager.RegisterAndExecute(do, undo)* with two functions (and with optional dispose callbacks that can free resources when the operation is removed from the undo or redo stack). For this approach, the undo manager is usually statically accessible (e.g. via LazySingleton). In a layered software architecture, the actions are recorded in the highest layers, close to the UI code. And the operations are registered in the lowest layers. Ideally, every undoable memory assignment should be wrapped in a call to RegisterAndExecute. Registering an operation in the *do* code (nested registering) is not allowed, and will fail at runtime.

The declarative approach allows composing objects of the SymmetricOperation monad. A symmetric operation is an object that holds both do and undo functions, as well as the optional dispose callbacks. Multiple symmetric operations can be merged into one. And a symmetric operation may return a value when it is executed, similarly to Haskell's IO monad. With this approach, the programmer can mark "symmetric" functions as returning a symmetric operation. These operations can then be composed up the callstack. Finally, the same code that records the action can then finally register and execute a single, composed symmetric operation. Because these operations are lazy, the order of the concrete lower level function calls does not matter, as long as the operations are composed in the correct order.

Note that a symmetric operation can be registered as a single operation along other operations that are registered and executed. Therefore, the programmer

can only use symmetric operation composition in the parts of the code that are especially complex, and simply register the do and undo functions in other trivial code.

3.2 Units

A common source of bugs and confusions when working with games or simulations is the representation of real world physical units. Usually, physical values are passed around simply as floating point numbers for performance reasons and because it is fast to write. This means that a **float** could refer to either a scalar, a duration, a length, a speed or whatever other physical unit. Naturally, the type itself does not inherently convey this context. As a consequence, developers need to be careful and thorough in naming parameters and variables to include information about which physical unit is represented. Developers especially need to document which specific unit the number is in. There is a difference between 13.37 meters and 13.37 kilometers. None of these names and comments are statically validated, which can lead to potential problems:

- Nonsensical calculations, such as adding a length to a speed.
- Wrong and unexpected factors in values when the wrong specific units are assumed.
- Stale names and documentations throughout the code when changing the underlying units.

To solve these issues, the F# programming language supports associating numbers with a unit of measure on a syntactical level, e.g. 55.0<miles/hour>. All calculations are validated at compile time, which solves the aforementioned problems.

WasaBii mimics F#'s functionality for use in C#, using custom types and extension methods. All supported units are defined in `.units.json` files which are included as resources in the compilation. The system uses Roslyn Source Generators [6] to generate all relevant types, operators, conversions and extension methods based on these unit definitions. WasaBii supports any number of `.units.json` files, as long as they adhere to a specified standard. Common units are included by default in the WasaBiiUnits.units.json file in the Units package.

Specifically, the developer can define a set of base units (e.g. length, mass, duration) as well as derived units which are either the product or the quotient of two other units (e.g. speed, area, volume, density). These different types of units each have a base unit (e.g. the SI unit like meters, kilograms, seconds) as well as a number of additional units which each have a long name, a short name, and a factor with which to derive them from the base unit (e.g. "Hours", abbreviated as "h", with a factor of 3600 seconds per hour).

For every defined unit, WasaBii generates one type. These types, such as Duration, are lightweight stack-allocated wrappers around a value. The value is a double precision floating point number corresponding to the unit's base unit, e.g. seconds. This means that the values are always stored as the base unit, and the actual unit is opaque to the developer.

Additionally to the generated types, WasaBii also includes a set of interfaces which can also be extended manually, which allow the use of many utilities commonly found on numbers. Existing extension utilities include rounding, interpolation, formatting, parsing, reinterpretation and finding the maximum, minimum, absolute value, sum, etc. . Units do not require Unity as a dependency, but all generated units have a custom property drawer for the Unity inspector and support Unity serialization.

For the basic usage, consult the following example:

```
1 Length distance = 100.Meters();
2 Duration time = 3.Seconds();
3 Speed averageSpeed = distance / time;
4 double kph = averageSpeed.AsKilometersPerHour();
```

3.3 Geometry

When dealing with calculations in 3D euclidean spaces, vectors and quaternions are powerful and versatile mathematical structures. As these are fundamental and essential concepts, every 3D game engine provides or utilizes an implementation of these, like the Vector3 or Quaternion in Unity.

Despite their broad operability, the data structures themselves are also quite trivial, merely encompassing three or four floating point variables of the required precision. While this fact contributes to their versatility, it can also lead to the same dilemma as **float**s representing units, as described in Section 3.2. For example, a 3D vector can represent a position, the offset between two positions, a direction, a velocity, and many more. When working with a vector, it is vital to understand what it represents, lest incorrectly applied mathematical calculations yield meaningless results. Such errors are hard to track and debug, since validation of the individual values as well as an understanding of the (possibly complex) calculations are required.

Similar to the aforementioned units, the Geometry framework combines a rigid type system with extensive utilities to solve this problem. It is built upon the former module.

Unit-like Wrappers are lightweight types, commonly backed by a single vector or quaternion. Similar to the unit types introduced above, they add context to the otherwise unspecific values they encompass, providing compile-time calculation validation and extension methods.

For example, one might want to move an object with a set velocity. If the object's position and velocity are both encoded as plain vectors, the type system allows simply adding them together. However, since velocity is a measure of change in position over time, it is necessary to first multiply the velocity with the time that has passed. Even in this trivial case, it is easy to overlook such a detail since the necessary context must be known. Stepping forward, the object does advance in the correct direction, so it might not be clear that the calculations

are incorrect. In a realtime application, the fault will manifest as the object moving with the wrong speed, which might also fluctuate with the framerate. These effects are difficult to track down, so it would be prudent to reduce the chance of this situation occurring in the first place.

The following wrapper types are included:

- Position: Vector
- Offset: Vector
- Direction: Vector
- Velocity: Vector
- Rotation: Quaternion
- Pose: Position + Rotation
- Bounds: Position (center) + Offset (size): axis-aligned, similar to Unity Bounds

Operators and methods are specifically designed to allow for interaction between types only where it is mathematically sound. For example: Since a Position represents a fixed position in space, there is no operator for adding two positions together. However, one can subtract one position from another to calculate the Offset between them. Naturally, the reverse operation of adding an Offset to a Position is possible, thus moving the Position by the given Offset. Since Offsets are not fixed in space, they can be freely added and subtracted from/to another.

In reference to the example outlined above, consider this interaction with the Unit system for moving a particle with an inherent velocity while being subject to gravity:

```
1 Speed speed = 343.MetersPerSecond();
2 particle.Velocity = speed * Direction.Forward;
3
4 Duration time = UnityEngine.Time.deltaTime.Seconds();
5 Velocity gravity = 9.81.MetersPerSecondSquared() * time *
      Direction.Down;
6 particle.Position += time * particle.Velocity + 0.5 * time *
      gravity;
7 particle.Velocity += gravity;
```

Conversions to and from the *System.Numerics* and (optionally) Unity geometry types are included in the module.

Relativity is another hidden property of objects like vectors. In a game engine specifically, it is vital to know whether a position is given relative to some parent or in world space. Without a type-system-based encoding, this information must be conveyed explicitly, e.g. in variable names or comments. Again, this can lead to the potential problems examined earlier. As a solution, the Geometry framework provides each of the aforementioned wrapper types in two distinct variants: Global and Local. Most operations can only be used when all values are given with the same relativity. Values can be converted using

```
local = global.RelativeTo(parent)
global = local.ToGlobalWith(parent)
```

respectively.

As an example, this code could be used for a particle that has just passed through a portal and should be teleported to the counterpart, for which the position, orientation and velocity relative to the exit portal after the event should be equal to the same properties relative to the entry portal before the event:

```
GlobalPose originalPose = particle.Pose;
GlobalVelocity originalVelocity = particle.Velocity;
LocalPose relativePose = originalPose.RelativeTo(portalA);
LocalVelocity relativeVelocity = originalVelocity.RelativeTo(
    portalA);

particle.Pose = relativePose.ToGlobalWith(portalB);
particle.Velocity = relativeVelocity.ToGlobalWith(portalB);
```

A parent can be given as a GlobalPose, a 4x4 transformation matrix or, if used inside the Unity engine, a Unity Transform.

3.4 Splines

Polynomial splines are a common trope in visual applications, both in 2D and 3D contexts. WasaBii offers a generalized spline architecture with explicit implementations of Catmull-Rom and Bézier curves while supporting custom implementations of any other polynomial-based spline type. A central goal in the design of this module is that it can be conveniently used in unison with the Geometry module as well as other math frameworks a user might employ. To this end, the splines are generic such that any type can be used for encoding the handles. The necessary mathematical operations are supplied through a type class that needs to be implemented for each required handle type. WasaBii offers default implementations for splines based on standard Unity vectors, as well LocalPosition and GlobalPosition. This allows for constructs such as a LocalSpline and a GlobalSpline respectively, which conserve the concept of typesafe relativity as explored in the previous section. Furthermore, seamless integration with the Geometry module is possible since all relevant operations, such as sampling the spline or calculating its length, are based on the corresponding WasaBii types (in this case: LocalPosition / GlobalPosition and Length).

Nonetheless, each variation is based on the same abstract code, which means that introducing a new handle type or spline type requires minimal effort while maintaining full compatibility. For example, when a user implements B-Splines using the type class definition mentioned earlier, it can automatically be used to construct Vector3-, LocalPosition- or GlobalPosition-based B-Splines. This makes the Splines module highly versatile since it can be easily extended and adapted to any situation.

3.5 Roslyn Analyzers

The C# toolchain enables package developers to provide extensive IDE feedback in the form of custom compile errors and warnings with potential automatic fixes. These feedback providers are called *Roslyn Analyzers* [5], and WasaBii uses them to provide some custom validations and suggestions at compile time. These validations and suggestions work out of the box when using the WasaBii package and require no additional setup.

Immutability Validation WasaBii allows the programmer to mark any type with the [MustBeImmutable] attribute. The type and all subtypes are then validated to be immutable at compile time. A type is immutable when there is no way for the programmer to mutate an instance of the type after construction. This is true when all fields are **readonly** and their types are either primitive or also known to be immutable. Violations are shown in any IDE just like any other compile error.

Immutability can be a useful property when working with interfaces that can be implemented anywhere. When an interface is immutable, then it can be assumed that all methods on that interface are pure, and (barring the access of static state) referentially transparent. In functional programming, these are useful properties to have, and this attribute can help guarantee these properties in interface implementations.

The validation itself is very conservative. For example, classes that are not **sealed** are prohibited, as one could inherit from them and add mutable fields. If a field has a non-sealed type, then that type must be marked with the attribute as well. Generic types must also be constrained by a type marked with the attribute. Only immutable collections from the standard library are allowed.

As an escape hatch, types and generic parameters can be marked with [__IgnoreMustBeImmutable], which disables the validation at that point.

4 Smaller Utilities

Aside from the aforementioned systems, WasaBii also includes a large number of small standalone utilities. These are explicitly sorted into utilities that require Unity, and utilities that are independent of Unity.

4.1 Core / Functional Programming Utilities

A core part of WasaBii's conception is enabling programmers to use modern, functional programming paradigms in their code. These paradigms emphasize immutability, statelessness, composability and static type safety. For this purpose, we provide a number of utilities which are generally applicable to programming in C#. These utilities require no additional dependencies and only depend on the standard library itself.

Custom Collections WasaBii includes some custom collections, which are generally applicable but not found in the standard library:

- MaxSizeStack: A stack which automatically removes the oldest entry when a max size is exceeded. Used in the undo system.
- ReadOnlyListSegment: Similar to a C# ArraySegment, but for IReadOnlyList. Useful to avoid collection copies.
- ReverseList: A lightweight wrapper that allows viewing an IReadOnlyList in reverse order efficiently.
- SequenceEqualityList: Collection built from copying an enumerable with special equality that checks for the same elements in the same order. Equality and hashing runtime depends on the number of elements, so be careful.

Option Idiomatic C# works with **null**, the "billion dollar mistake". Errors are often hard to detect at compiletime, and null values can propagate for a long time without causing an error, which results in a temporal disconnect between the cause of the error and the actual exception caused by accessing a null value at runtime.

To solve the problem of null values, languages that support a functional paradigm (as well as Java and C++, for example) provide an *Option* monad in the standard library (sometimes called *Optional* or *Maybe*). WasaBii provides an Option<T> type, which is a lightweight wrapper for either a value or no value. Programmers need to deal with the potential absence of a value when a method returns an =Option, which massively improves typesafety and reduces runtime errors. Options also help clarifying intent in method signatures without requiring fragile documentation.

In WasaBii, the Option implementation is a stack-allocated struct which contains a value as well as a boolean that indicates whether the value is present or the default value for its type (null for reference types, and zeroed memory for value types). The overhead of tracking an additional boolean is negligible in most applications. The implementation is also fully compatible with modern C#'s pattern matching features, such as **is** and **switch** expressions. WasaBii also provides a plethora of utilities that make working with options more comfortable, such as an idiomatic option.TryGetValue(**out** var value) as well as a universal Option.None constant that coerces to any type.

Result Functional programmers avoid exceptions: they circumvent the type system and make it hard to recover once invalid state is achieved. Instead, functional programming relies on a monad called *Either*, which can hold one value of either one of two distinct types. By convention, *Right* is the result (because it is "right"), and *Left* is the error value.

WasaBii provides an implementation of this common error handling pattern, but with names adjusted specifically to reflect the error handling semantics. The Result<R, E> type can either hold a result of type R or an error of type E. Similarly to Option, Result values are lightweight wrappers which are fully

compatible with pattern matching. There are many utilities and extensions for results which make working with them as convenient as possible.

Options can be converted to results by providing an alternative for either the result or error in case of absence. Conversely, results can be converted to either an optional result or an optional error.

4.2 Unity-specifics

Additionally to the many generally applicable utilities, WasaBii also contains a large number of utilites specific to working in the Unity engine:

Singleton / LazySingleton A common pain point when working with unity is dependency management, especially using the singleton pattern. By default, if a behaviour needs to reference some central "system", then one needs to add that system as a component to some object in the scene, and then add a field to the behaviour referencing it. Then the developer needs to drag the system object into the field in the inspector. This is really annoying for prototyping and does not scale well.

An alternative solution involves static state. Just store the system's state in a static context. But that causes a multitude of problems in Unity: static state is initialized when the editor starts and persists across scenes. It only resets when play mode begins, but only when Unity is configured appropriately. And this resetting of static state leads to more wait times during development.

As a solution, WasaBii provides two base classes: `Singleton<T>` and `LazySingleton<T>`, where `T` is the concrete type of the inheriting class. Both allow static access to the single instance from anywhere (e.g. `MySingleton.Instance`) but reset their state whenever a scene changes. A singleton needs to be attached to some game object in a scene. In contrast, a lazy singleton attaches itself to a special game object when it is first accessed, and does not need any tampering with the unity scene.

Note that these abstractions are great for quick prototyping. However, they do not have any support for explicit dependency management. Singletons are all initialized when the scene is loaded, either at a random order or the one defined using Unity features. Lazy singletons are instantiated in order of their first usages. This lack of control can lead to "spaghetti code" in larger codebases, and both kinds of singletons are only very rarely used in BII's corporate codebases. However, they work perfectly well for smaller projects and prototypes.

Coroutines framework Coroutines [9] are a large part of working in Unity. However, they are unweildy and do not compose well. In Unity, you define a coroutine by implementing a generator function, using the `yield return` feature. These generator functions cannot be nested, and they cannot capture any values from an outer scope. They also need to be started inside of an existing component. As unity generators use the `IEnumerator` interface, there aren't many available builtin utilities to compose them.

WasaBii provides a lot of functions in the static Coroutines class to supplement this. These functions allow building up coroutines through common patterns such as (conditional) repetition as well as composition, without a single arcane yield **return**. As a bonus, there's also a coroutine.Start() extension method, which allows starting the coroutine from anywhere, even outside of a mono behaviour.

Component Query Extensions When working in Unity, querying for components in game objects is a common occurrence. However, Unity's builtin methods have long and unwieldy names and cannot be parameterized well. They also limit searching for components to either the parents or the children or the object itself, which can often lead to boilerplate code. WasaBii provides additional query extension methods on game objects and components. These are easy to discover and can be parameterized with a Search enum value, which allows even more exotic searches such as in siblings. There is also a Find method on game objects which allows traversing the hierarchy using search parameters with a custom getter function.

4.3 And much more

In addition to all the aforementioned solutions, WasaBii also contains many more small utilities. Mentioning all of them would be outside of the scope of this report.

As a rough overview, these utilities include:

- Many utilities for composing, consuming and iterating IEnumerable<T>
- Functional utilities to model common control flow patterns in a more declarative manner
- Helpers for checking for null and absence of components
- Math support for double precision floating point numbers
- File and directory utilities
- Utilities for executing different code whether in an editor or a runtime context
- Immutable modifiers and swizzling for Unity colors and vectors
- Adapters that allow using async code in Unity
- A dynamic line renderer that updates reactively
- A custom exception for the default case when switching on an Enum, with tooling
- Monads for modeling computations with a "progress bar" over time

5 Conclusion

WasaBii is a package consisting of a large collection of independent standalone systems, utilities and tooling. The package is explicitly segmented in multiple sub-packages with appropriate dependencies between each other. This allows inclusion of only some sub-packages while ignoring other packages that are not

necessary or desired, such as e.g. the Unity utilities when working in a dotnet core project. All packages are open source and available under the Apache 2.0 license. The included systems and utilities have been tested over years in a real software product with hundreds of thousands of lines of code, and are proven to improve development velocity, maintainability and static validation of source code.

References

[1] *Adobe Photoshop*. https://www.adobe.com/de/products/photoshop.html. Accessed: 2023-12-06.

[2] *Apache 2.0 License*. https://www.apache.org/licenses/LICENSE-2.0. Accessed: 2023-12-06.

[3] *Blender*. https://www.blender.org. Accessed: 2023-12-06.

[4] *Building Information Innovator*. https://www.bii-gmbh.com. Accessed: 2023-12-06.

[5] *Roslyn Source Code Analysis Documentation*. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview?tabs=net-8. Accessed: 2023-12-06.

[6] *Roslyn Source Generators Documentation*. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview. Accessed: 2023-12-06.

[7] *Unity CoreLibrary Repository*. https://github.com/XDracam/unity-corelibrary. Accessed: 2023-12-06.

[8] *Unity Engine*. https://unity.com. Accessed: 2023-12-06.

[9] *Unity Engine Coroutines Documentation*. https://docs.unity3d.com/Manual/Coroutines.html. Accessed: 2023-12-06.

[10] *Unity Engine Packages Documentation*. https://docs.unity3d.com/Manual/Packages.html. Accessed: 2023-12-06.

[11] *WasaBii on GitHub*. https://github.com/BII-GmbH/WasaBii. Accessed: 2024-2-09.