Tampere University

Tatu Loikkanen

PROPERTY SYSTEMS IN GRAPHICS FRAMEWORKS

Implementation, benefits and comparison between Qt and Kanzi

Faculty of Information Technology and Communication Sciences
Bachelor of Science Thesis
May 2019

ABSTRACT

Tatu Loikkanen: Property systems in graphics frameworks
Bachelor of Science Thesis
Tampere University
Computing and Electrical Engineering
May 2019

A property system is a concept about having special class members, properties, in an object oriented programming language. Properties can be classified to partially implement reflection, which represents the program's ability to introspect its objects and their metadata. Reflection itself is a part of the programming concept known as metaprogramming. The C++ programming language offers a very limited amount of tools for reflection in its standard library, hence implementing a non-standard property system has became a viable solution to patch this need.

This Bachelor of Science thesis studies the background theory, implementation and usage of a property system written in C++. Two existing property systems are used as a study material: the property systems provided by Rightware and The Qt Company, in their product families *Kanzi* and *Qt*. A lightweight focus of this thesis is on comparing these two systems, specializing in how the Kanzi's implementation could be enhanced further. Both of the product families are graphics frameworks, although they are quite different in terms of codebase size, targeted market segments and release license.

The theoretical background of this thesis begins from the concepts of metaprogramming and reflection, falling all the way to the year 1982 and to Brian Cantwell Smith's Ph.D thesis *Procedural Reflection in Programming Languages*. In his thesis Smith coins and defines the term *reflection*. Several other scientific releases about modern day reflection and implementing a reflection library in C++ are used as a source material too. For the property-system-specific research the main sources are the associated documentations and the source codes of each system.

As a final result of this thesis, the research on the property systems did not reveal anything radical or revolutionary. Both compared systems turned out to be useful tools when working with user interface development. Some possible improvements for Kanzi's property system were discovered, but no critical design or implementation flaws were found, which was expected. One generic result worth noticing was that the property system as a separate third-party C++ library may possibly lose its usefulness as new C++ standards are released as time passes.

Keywords: reflection, property system, property, system, graphics framework, graphics, user interface, C++, Qt, QML, Kanzi, Kanzi Studio, Kanzi Engine

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

1 Introduction 1
2 Metaprogramming, reflection and properties 2
2.1 Reflection 2
2.2 Ways of reflection in C++ 3
2.3 Property system, a subset of reflection 3
2.4 Approaching properties in C++ 5
3 Kanzi engine and Qt 6
3.1 Overview of Kanzi 6
3.2 Overview of Qt 7
3.3 Property system implementation in Kanzi 9
3.4 Property system implementation in Qt 10
4 Analysis and comparison of property systems in Kanzi and Qt 11
4.1 Using the property system in Kanzi 11
4.2 Using the property system in Qt 13
4.3 Analysis and comparison 17
4.3.1 Features 17
4.3.2 Implementation 17
4.3.3 Usage 18
4.4 Possible enhancements for Kanzi 18
5 Conclusion 20
References 21

1 INTRODUCTION

A property system can be considered as a partial implementation of reflection. In general, reflection represents the ability to introspect and modify the metadata of a program. This metadata can consist from the elements of a program's objects, for example. Dynamic or runtime reflection is the type of reflection which takes place on a program's runtime. In this sense, a property is considered as a special class member, whose value and presence can be read or modified runtime. The abilities of properties thus include adding them to a certain object on the fly, or respectively, removing them. Properties provide ways to adjust and affect the behaviour of the objects, and if taken to a higher level of abstraction, the properties can also affect the visual look of UI (user interface) elements. Properties and reflection fall under the large category of metaprogramming, a way of programming where the code created by programmer has the ability to modify existing or produce new code.

As C++ itself doesn't provide too many ways to inspect objects runtime, implementing a property system (especially when developing a graphics framework) may be a viable choice. However, a system like this easily brings in at least some overhead and bloat, not to forget how important (and challenging) it is to pursue towards an API (Application Programming Interface) comfortable and efficient to use for the programmer.

This Bachelor of Science thesis analyzes the implementation, usage and benefit of a property system, focusing especially on the ones implemented in Kanzi and Qt. Comparison between these two graphics frameworks is also presented. Kanzi is a software family consisting of two parts: Kanzi Studio, a PC tool for UI design, and Kanzi Engine, a graphics engine and API for running the designed user interfaces on various target platforms. Kanzi is a product of Rightware, a software company founded in 2009.

A glimpse to the theoretical background of metaprogramming, reflection and properties is presented in chapter 2, overviews of Kanzi and Qt in chapter 3 and a deeper analysis on both frameworks and their property systems is in chapter 4. Some thoughts on improving the property system in Kanzi is also presented in the end of chapter 4. A conclusion of the main results is in the final chapter.

2 METAPROGRAMMING, REFLECTION AND PROPERTIES

The concept of metaprogramming is broad, even though its basic idea is simple: creating code that creates new or modifies existing code. If viewing in the timescale of software engineering, it is also a relatively old thing. With languages such as Lisp, metaprogramming played a crucial role in software development and research as early as in 1970-80's [18]. Modern day technologies and programming languages are, however, using relatively specific sets of the ideas presented back then, and things like Java's reflection and the template metaprogramming of C++ are probably among the first ones being mentioned if discussing the matter. One of the best everyday metaprogramming examples is *a compiler*, which is a program that generates target language code from the source code written by the programmer in another language.

2.1 Reflection

Reflection is a generic name for the ability to introspect and modify the metadata of a program. Runtime reflection enables the programmer to navigate through and use this metadata to create new instances of classes and call their methods. The available metadata includes at least the program's classes, interfaces, methods and attributes. For the software itself reflection provides the ways to adapt to runtime environments which may change on the fly. In dynamically typed languages, such as Python and Ruby, reflective operations can offer compact and generic ways to solve various problems. [1, 3, 6] The term *reflection* was first coined by Brian Cantwell Smith, in his doctoral thesis *Procedural Reflection in Programming Languages* from 1982 [18].

Reflection is a built-in feature in many programming languages, offered for example by the language's standard library. It is also possible that reflection comes as a side product from the way the object-oriented programming has been implemented in the programming language. For instance, Lua and Smalltalk represent the second way, while Java goes with the first approach. [1, 6]

2.2 Ways of reflection in C++

Unlike the programming languages discussed in the previous chapter, C++ offers a very limited amount of tools for reflection, especially for the type of reflection taking place during runtime. C++ has its own powerful template metaprogramming abilities, but they are limited for compile-time type introspection purposes. Several proposals for adding proper reflection to the C++20 standard were presented, but they were postponed for later releases [22].

The most common runtime type information (`rtti`) operation available is the `dynamic_cast`. It is used to convert pointers and references to different directions in the inheritance hierarchy, may the direction be either up, down or sideways. `dynamic_cast` takes two arguments: the destination type and the original pointer or reference to be converted. The object in tip of the original pointer must be an instance of the destination type class, otherwise a `nullptr` is returned and the cast has failed. This occurs to achieve the desired safety of the operation, but it has its usage-limiting costs. Forcing the two types to be in the same hierarchy prevents the programmer from finding out the actual type of an object when trying to convert a void pointer (`void*`). For example, dynamic cast cannot be used to find out if a `void*` is pointing to an integer or a double value: the cast will return a nullptr, even if the void pointer converted was actually pointing to an integer or a double.

Another tool for C++ `rtti` is the operator `typeid()`, which returns a reference to a special type, `type_info`, defined by the standard library. Two `type_info`'s can be compared with `operator==`, and thus an operation of `typeid(A) == typeid(B)` can be used to determine if both returned `type_info`'s refer to the same type. The problem with `typeid` is its ignorance of class hierarchies. The comparison operation returns false even if the right-hand type inherits the left-hand type (`B` inherits `A`). [1]

`dynamic_cast` and `typeid()` are pretty much everything C++ has to offer for runtime reflection. Java, as a textbook example, in its standard library, includes ways to find out at least the methods of a class and obtain information about constructors and the fields defined in a class [8]. It is expected that the amount of reflection tools in standard C++ may not satisfy every use case. Hence it is natural that some organizations have developed their own toolkits to patch this need.

2.3 Property system, a subset of reflection

One method to gain some reflection functionality in C++ is to use a property system. For example, Qt, an open-source cross-platform graphics framework and SDK (Software Development Kit) developed by The Qt Company, offers this kind of functionality. As in Qt and in many other programming languages or programming language extensions (see next chapter), properties can be considered as some sort of special class members.

A property holds a value of a certain type, may it be some of the basic data types or an object type. Functions for both setting and retrieving the value of a property can be provided, or the property can just be of a read-only kind. The main benefits of properties is that they can be added, modified or removed during runtime, or *dynamically*. [10] This is what links property systems to be one of the many varieties of metaprogramming.

The amount of use cases for a property system is limited only by the programmer's imagination. In Kanzi (a graphics framework, API and user interface SDK by Rightware), properties are mainly used to affect and control the look and behavior of the user interface elements, *nodes*. For example, a two-dimension text block element can have the properties for width and height, text font size, text color and even for the text string itself. Figure 2.1 illustrates an example of this kind of a two-dimensional text block in Kanzi Studio's preview-window. The available properties are listed on the left side of the figure, while the resulting text block is next to it on the right. In Kanzi's inter-node messaging system, special properties are also used as message arguments for the messages sent between nodes. [4]

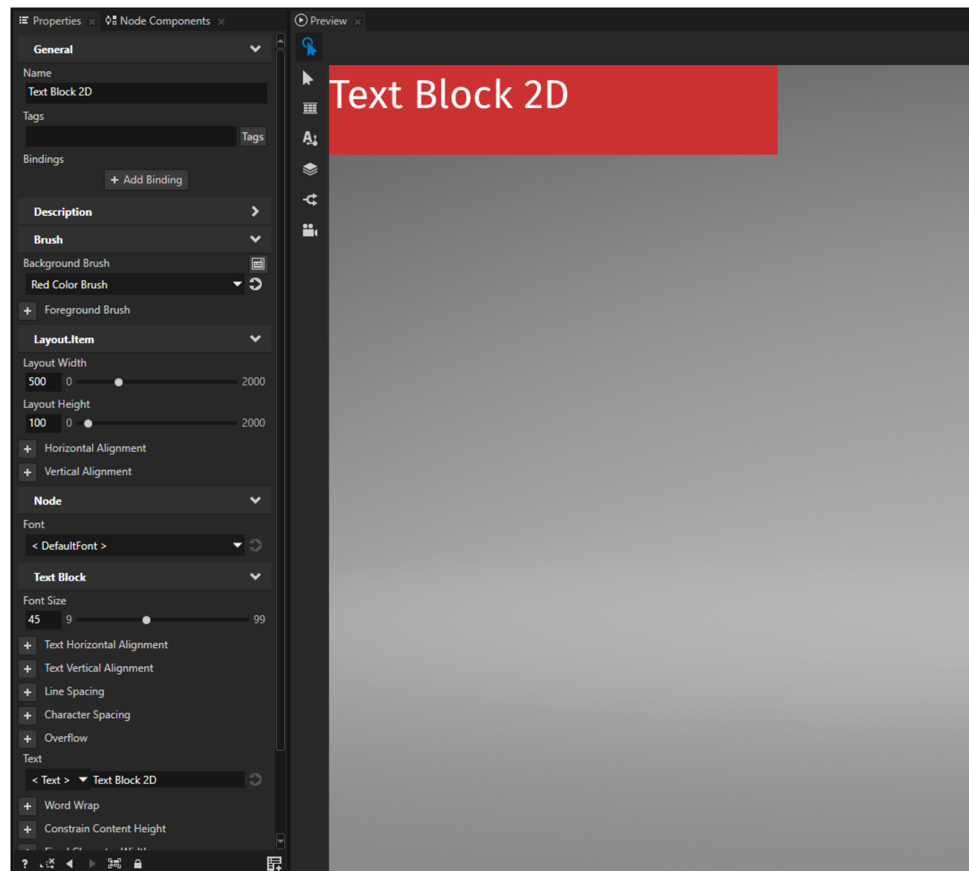

Figure 2.1. *A text block 2D and partial list of available properties in Kanzi Studio.*

In Qt, the properties are also playing a vital role in the user interface appearance and control logic. They can also be handy when the programmer wants to transfer data from C++ side to QML or vice versa. Figure 2.2 shows the Qt's vision of the text block and its properties, as they appear in Qt Creator's (version 4.7.2) design-mode.

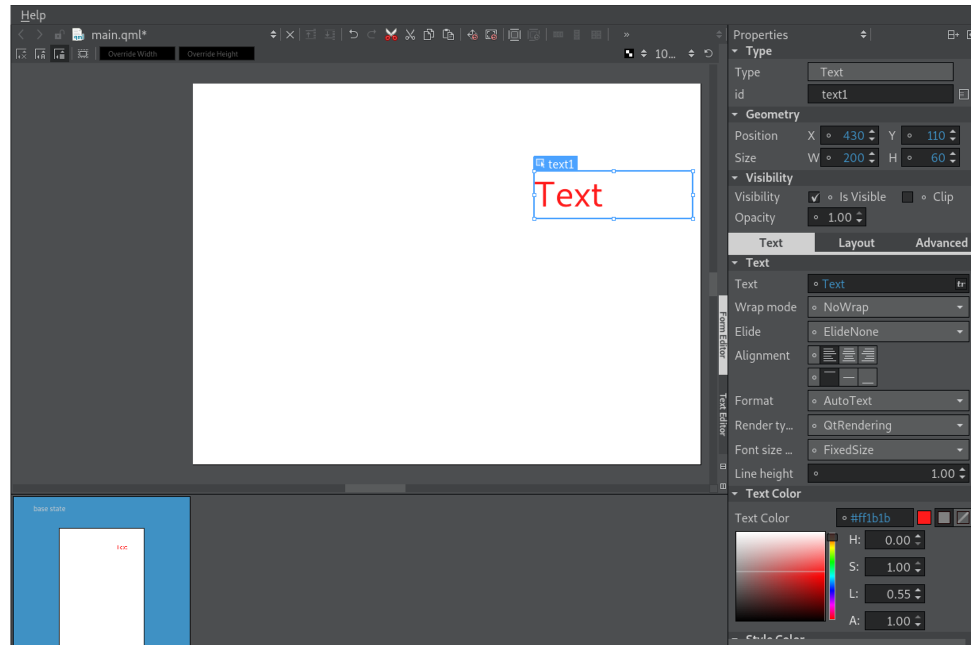

Figure 2.2. *A text block and some of its properties in Qt Creator's Design-mode.*

Since both Kanzi and Qt are cross-platform products, the properties are not platform-dependent. Therefore they should theoretically work the same way on every supported system or platform.

2.4 Approaching properties in C++

In programming languages other than C++, the property system is often just one tool among many others in the available reflection API. For C++, properties can be implemented with a specific library or as a part of some other API. Some compiler-specific properties are present, for example in Microsoft's Visual C++ language. With MSBuild C++ compiler, non-static virtual class or structure data members can be marked as properties with special `property` -attribute. Like other Microsoft-specific-storage-class attributes, the `property` can be applied with the special `__declspec` -keyword. [9] The usage of these properties is quite limited though, especially in the sense that they only work with Microsoft's C++ compiler.

In February 2019, the ISO C++ committee presented the results from their winter 2019 meeting. At that moment the design of C++20 standard was announced as feature-complete. According to the meeting results, no reflection nor any property-related features will be part of the C++20 standard. However, a working-draft of technical specification for C++ reflection extension was published. It is estimated that the extension will be part of the standard library in C++23 or C++26 standards, most likely with the latter one. [17, 20] Even though the specification doesn't explicitly mention dynamic runtime properties, the other suggested features can possibly patch the need and usage of properties pretty well, and the need for a specific property system should be re-considered.

3 KANZI ENGINE AND QT

While both Kanzi and Qt are GUI (Graphical User Interface) frameworks, they do have several significant differences in their ideology and targeted field of use. According to Rightware's website, "Kanzi enables the rapid design and development of user interfaces for the automotive industry and other embedded applications" [21]. A somewhat corresponding advertising slogan from Qt's documentation front page would be "Qt is a full development framework with tools designed to streamline the creation of applications and user interfaces for desktop, embedded, and mobile platforms" [10]. Deducing from these two statements it is clearly visible that Kanzi is targeted more to use cases in automotive and embedded device industry. Qt lists mobile and desktop platforms along the embedded devices, so Qt is marketing its product(s) to a larger market section than just embedded devices or automotive industry. This difference, however, doesn't affect this thesis that much, as the main objective here is to compare just two relatively small subsets of features existing in both frameworks: property systems.

3.1 Overview of Kanzi

As the Kanzi product family is targeted to the automotive and embedded industry use, its selection of products and features is not as wide and general-purpose as Qt's. All Kanzi products are released under commercial license.

The main products of the Kanzi family are the *Kanzi Studio* and *Kanzi Runtime*. *Kanzi Connect* is the newest addition along these two. [21]

Kanzi Runtime

Kanzi Runtime provides the C++ heart of Kanzi's ecosystem. Basically it is a cross-platform C++ API which is used to integrate the user interface with data sources and services. Kanzi Runtime is the home for Kanzi's graphics rendering engine while also providing programmatic access to all the features of Kanzi. A plug-in interface for extending the basic built-in Kanzi functionalities is part of Kanzi Runtime too. [21]

Kanzi Studio

The main tool for user interface design with Kanzi is the Kanzi Studio. Unlike Qt Creator from the Qt Company, Kanzi Studio is not used to write lines of code, but to design the user interfaces.

Among the UI editor, Kanzi Studio provides tools for 2D- and 3D composition. A built-in library of UI components and layouts is also part of the tool. [21]

Kanzi Connect

Kanzi Connect is the newest addition to Kanzi product family. According to Rightware, it is "Powering Tomorrow's Connected Car". [5] The term *connected car* comes from the automotive industry, and is basically used when referring to cars which are equipped with devices capable of internet access. The devices can be anything from integrated text message to collision avoidance systems. [2]

Kanzi Connect is used to share data, assets and services between multiple different devices. These multi-device setups can also be created and managed with Kanzi Connect. It is based on a server-client architecture. [5]

3.2 Overview of Qt

The repertoire offered by Qt is quite broad. It is worth noticing that most of Qt's libraries, developing tools and build tools are not connected to each other: one can use Qt's modules with any applicable IDE (or without IDE at all), or in the same way, write code with Qt Creator and leave Qt libraries and build systems unused completely [11, 12, 13].

Qt's main products include a set of libraries and API's, tools for developers and designers and the IDE (Integrated Development Environment), *Qt Creator*. [10] As Qt is available under both commercial and open source licenses it has been made possible to have a wide developer community, which (according to Qt) consists of more than 1 million developers.

Libraries: Qt Essentials, Add-Ons and Value-Add Modules

The Qt libraries consist of a pile of modules. Qt's documentation separates these modules to *Qt Essentials* and *Qt Add-Ons*. As deducible from their names, Qt Essentials make the general and foundative base of libraries used in Qt applications. The modules in Qt version 5.12 are:

- Qt Core - The core of Qt and non-graphical classes
- Qt GUI - Base classes for GUI components
- Qt Multimedia - Audio, video, radio and camera classes
- Qt Multimedia Widgets - Widget-based classes for multimedia
- Qt Network - Classes for network programming
- Qt QML - Classes for QML and JavaScript
- Qt Quick - Declarative framework for dynamic applications
- Qt Quick Controls - Lightweight QML types for simple and performant UIs
- Qt Quick Dialogs - Types for system dialogs from Qt Quick application
- Qt Quick Layouts - Items used for layouting in Qt Quick 2 applications
- Qt Quick Test - Unit test framework for QML
- Qt SQL - Classes for SQL database integration
- Qt Test - Unit test classes for Qt applications and libraries
- Qt Widgets - Classes for extending Qt GUI with C++ widgets.

The Qt Add-On modules serve needs for special purposes. Unlike Qt Essentials, the Add-Ons may be supported only by several development platforms, and not on every platform that Qt Essentials are working. The Add-On list is long, and has lots of things all the way from Qt Gamepad to Qt Virtual Keyboard modules.

In addition to Qt Essentials and Add-Ons, Qt Modules also include the *Value-Add Modules*. Value-Add modules consist of *Qt Automotive Suite*, *Qt For Automation* and *Qt for Device Creation*. Their idea is to provide additional value and evolve under their own schedules. It is also notable that Value-Add Modules are not licensed under open source licenses (like Qt Essentials and Qt Add-Ons are), but only commercial. [11]

Tools: Qt Creator and Qt Designer

Qt provides also the official IDE for Qt development: *Qt Creator*. The *Qt Designer*, tool for designing and building GUIs, is shipped with Qt Creator and jointed to the IDE.

The Qt libraries and modules are usable with any IDE, and the same way Qt Creator can be used to program without using the Qt libraries at all, if desired. Qt Creator is targeted for programming in languages C++, QML or Javascript. [13, 14]

Build: qmake, moc, uic

In addition to libraries and development tools, Qt offers several tools for the build process. *qmake* is the Qt's tool for automated creating of makefiles. It can be used with any

software project, may it be written using Qt libraries or not. [12]

Qt's *moc*, meta object compiler, is a crucial tool when building applications with Qt. Its main task is to handle the Qt's C++ extensions, such as the signal-slot system and most interestingly, the object properties. [16]

The third fundamental build tool in Qt is the *uic*, user interface compiler. Basically uic reads the `.ui` files created by Qt Designer and compiles a C++ header from it. The `.ui` -files are generated XML based on the user interface designed with Qt Designer. [15]

3.3 Property system implementation in Kanzi

A property in Kanzi is realized in the form of a `PropertyType` class. An instance of that class represents a single property type available in Kanzi's property system. `PropertyType` is a templated class, taking the `PropertyType`'s `type` as a templated parameter. The `type` can be any of C++'s basic data or class types: `int`, `float` or `std::string` for instance. Some of Kanzi's own classes can also be used.

In order to use Kanzi's properties in a new class it has to inherit the class `PropertyObject`. `PropertyObject` is the base class which brings in the support for setting and retrieving of property types, i.e. `PropertyObject` makes it possible for an object to have properties. In addition to properties, Kanzi has several other reflective features available: *message types* and *metamethods*. For the user's convenience, the base class `Object` links all these three reflective features (properties, metamethods and message types) into one class. To achieve this, `Object` inherits the previously mentioned `PropertyObject` and an another related class, `MetaObject`. Now the programmer can inherit their classes from `Object` and thus have access to all of Kanzi's reflective features. Figure 3.1 demonstrates an example of a 2D-image node's inheritance hierarchy, where the `Object` and its ancestors are visible.

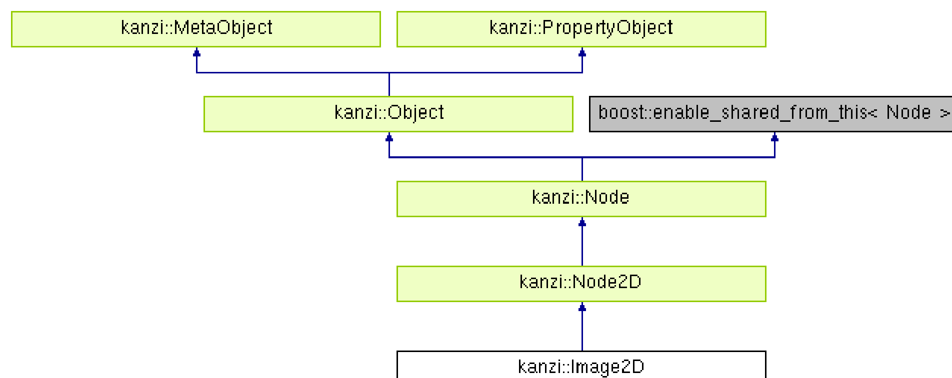

Figure 3.1. *An inheritance diagram of kanzi::Image2D. [4]*

In addition to type inspection tools, the `Object` also implements the class' association to a *domain*. A domain in Kanzi can be considered as a main frame for one Kanzi application.

It binds the objects (such as UI nodes of an application) and Kanzi subsystems (resource manager, renderer, task dispatcher etc.) together.

The value of a `PropertyObject`'s property can be retrieved, modified or listened to. Because of this it is possible to create systems that react when values of certain properties change. With this the programmer can create dependencies between different objects' properties. This is one of the key benefits when creating user interfaces with properties. [4]

3.4 Property system implementation in Qt

Qt has a bit different approach to properties. Basically a Qt property of an object is always linked to an ordinary class member variable that provides the value for the property [19]. The value's type must be something that is supported by the `QVariant` class [10]. `QVariant` is, shortly expressed, a class which holds a reference to an object, while the referenced object and its type can dynamically change and be changed [10]. It somewhat resembles a generic variable of a dynamically typed programming language. Thus there is a plethora of types available for a property in Qt, as most of Qt's classes can be used as a variant type.

A property isn't capable of much if it only has a value. Inside Qt's property system the class `QMetaProperty` serves this need by carrying the metadata of a property. As is deducible, `QMetaProperty` can be used to query information such as property's name and type. Together with a class' member variable value the `QMetaProperty` makes an instance of a property in Qt. The fundamental base class, `QObject`, offers the required logic to combine these two and create a new Qt class that is able to have properties. Similar to Kanzi, a class using properties must be inherited from QObject (`Object` in Kanzi). [10, 19]

Like Kanzi, Qt offers several other reflection tools in addition to properties. *Metamethods*, *metaenums* and the *signal-slot system* are some examples of such. The `QObject` uses the struct `QMetaObject` to control all these features (compare to Kanzi's `Object` class). [19] In Qt, the most notable feature gained with usage of properties is their visibility to QML side, as the properties work as a bridge between Qt's C++ and QML's JavaScript programming. [10]

4 ANALYSIS AND COMPARISON OF PROPERTY SYSTEMS IN KANZI AND QT

At this point we have at least a faint understanding of the property system in general and how one can be designed with an object-oriented language. This chapter presents a deeper analysis and comparison between the two discovered systems, their implementations and hands-on usage.

4.1 Using the property system in Kanzi

In order to create and use new properties in Kanzi, the user has to *declare*, *register* and *define* them. Naturally lots of properties are offered by the framework, but e.g. when creating new node types, creation of new property types correspondingly is often reasonable or even required to gain the desired functionalities. The method of creating new properties presented in this section applies when creating properties by coding, which in practice means using the API for developing Kanzi Engine plugins. Property types can also be created in Kanzi Studio (without writing any code), but this thesis focuses on inspecting the property system from a programmer's point of view.

Creating a Kanzi property type

A property type is declared inside a C++ class just like any other class member. Property types are declared as `public` and `static` and thus they exist throughout the lifetime of an application. In most cases the properties are declared and defined in classes which use them, but for a property it is also possible to affect the appearance, behaviour or state of another object too. For example, a property type named `ExampleProperty` could be declared among other class members as following:

```
static PropertyType<float> ExampleProperty;
```

The property types can be declared before or after the declarations of "normal" member variables and functions. In addition to declaring the properties, they must be registered to the Kanzi Engine. If following the usual C++ convention of declaring variables and functions in a header and then defining them in a source file, the property declarations and registrations should take place in the header and definitions in the source.

Kanzi Engine offers the macros `KZ_METACLASS_BEGIN()`, `KZ_METACLASS_PROPERTY_TYPE()` and `KZ_METACLASS_END()` to be used when registering properties. The previously declared `ExampleProperty` can now be registered with these macros. Let's consider that the property type and registration takes place inside the declaration of a new node class, `ExampleNode2D`, which is derived from the `Node2D` class. With these assumptions, the `ExampleProperty` can be registered to the Kanzi Engine as with the code example 4.1.

```
1 KZ_METACLASS_BEGIN(ExampleNode2D, Node2D, "Kanzi.ExampleNode2D")
2         KZ_METACLASS_PROPERTY_TYPE(ExampleProperty)
3         // Register more properties here ..
4 KZ_METACLASS_END()
```

Snippet 4.1. *Example of registering a property type to Kanzi Engine.*

Here it is nice to notice again that the registered property type is associated with a class (`ExampleNode2D`). This still doesn't limit the property's effects to that class only.

To finish the creation of a new property, it must be defined. The definition of our `ExampleProperty` could look like it does in another code example, 4.2, and could be placed in the `.cpp` source file of the `ExampleNode2D`.

```
 1 PropertyType<float> ExampleNode2D::ExampleProperty(
 2         kzMakeFixedString("ExampleNode2D.ExampleProperty"),
 3         2.2f,
 4         PropertyTypeChangeFlagMeasure,
 5         true,
 6         KZ_DECLARE_EDITOR_METADATA(
 7                 metadata.displayName = "Example property";
 8                 metadata.tooltip = "Property used as an example.";
 9                 metadata.host = "Node:user, Node2D:context";
10                 metadata.editor = "Slider";
11                 metadata["LowerBound"] = "0";
12                 metadata["UpperBound"] = "10";
13                 metadata["Step"] = "0.1";
14         )
15 );
```

Snippet 4.2. *Example of defining a property type in Kanzi Engine.*

The parameters given to the property type definition in 4.2 are the property's *name* in the metadata system (as generated with `kzMakeFixedString()`), *default value* (floating point number "2.2" in this example), the *change flags* (`PropertyTypeChangeFlagMeasure`, meaning that this property affects the layout of items) and a boolean value telling if the property is *inheritable* or not (`true` here). The rest of the stuff in the property type definition, meaning the macro `KZ_DECLARE_EDITOR_METADATA()` actually expands into a lambda function, which returns an *editor info* shared pointer. Basically that editor info provides

metadata to Kanzi Studio and Engine, such as the property's display name and tooltip. [4]

Using the created property type

The `PropertyObject` offers convenient methods for getting and setting the property values: `setProperty()`, `getProperty()`, `getOptionalProperty()` and `getPropertyBase()`. While `setProperty()` simply sets a property's value for a node, the `getProperty()` and its variations perform various operations to calculate the final returned value. As the `Node` inherited `PropertyObject`, the methods are available for use there too.

When retrieving the property's value with `getProperty()`, several factors affect the final result. First the property's *base value* is calculated and after that any existing *modifiers* are applied. The base value is its local (specific for this node instance) value, either set with `setProperty()` or loaded from a *Kanzi binary, kzb,* file. If neither of these is present, then the base value is the value set by any *styles* that affect the property. If no styles will apply, the value defined in the property's metadata (as in example 4.2) is determined to be the base value.

There are two modifiers which may then affect the base value: property values defined in a *state manager's state* and an *animation*. The modifiers will be evaluated to the base value in the same order the modifiers were added or applied. Now the property's final value is calculated and it can be returned. The `getOptionalProperty()` performs the same kind of operations as `getProperty()`, but if the base value would be evaluated by the metadata, a `nullopt` (empty optional) is returned. This responds to the ideology present in Kanzi, which kind of means that every property type registered to the engine can be present in any node, and the value can be retrieved or modified accordingly. It is just up to the node if it decides to use the property for anything or not. If `getOptionalProperty()` returns a `nullopt`, it is most likely not used by the node, at least in that moment of time.

The third property value getter, `getPropertyBase()` returns just the property's base value, without any modifiers applied, determined the same way as with `getProperty()` (local, style or metadata value). The `setProperty()` and these three property getters provide an uniform way to handle data in every Kanzi component and subsystem. This way of handling data is the main motivation for property system usage in Kanzi, and it enables the use of properties in actual UI development with Kanzi Studio. [4]

4.2 Using the property system in Qt

In Qt, the properties are rather "extended" from normal member variables of a class. As a reminder, in Kanzi they were created as whole new and separate variables.

Creating a property in Qt

A private, protected or public class member `QString m_exampleString` can be registered as a property by giving it and a function returning its value to the macro `Q_PROPERTY()` in a class' declaration. So by stating `Q_PROPERTY(QString exampleProperty READ getExample WRITE setExample)` a property named `exampleProperty` with data type `QString` is created. In this case the `getExample` and `setExample` would stand for the property's (class member variable's) *getter* and *setter* functions, which are simple methods for getting and setting the value of the property. The minimum required parameters for `Q_PROPERTY()` are the property's data type, name and the getter function, so providing a setter (like in this example) for a property is not obligatory. [10]

In addition to `WRITE` other optional parameters are available, such as `RESET` (function to reset the property to its context specific default value) and `NOTIFY` (the signal emitted when the property's value changes). [10] As an interesting side note, the `Q_PROPERTY()` macro doesn't expand into anything interesting with a normal C++ compiler: it is just used by the Qt's meta-object compiler to identify properties from a class declaration. [19]

For a more complete example, let's consider an example declaration of a created Qt class, `PointModel`, which represents a single measured value on a certain time point. The snippet 4.3 declares this class as following:

```
1  #include <QObject>
2  #include <QDateTime>
3
4  #include "currentvalues.hh"
5
6  // PointModel is the model for a single measured point.  It has a single value,
7  // unit and timestamp.
8  class PointModel : public QObject
9  {
10     Q_OBJECT
11     Q_PROPERTY(QString value READ getValue NOTIFY valueChanged)
12     Q_PROPERTY(QString name READ getName NOTIFY nameChanged)
13     Q_PROPERTY(bool enableAnalog READ getAnalogFlag CONSTANT)
14     Q_PROPERTY(QString unit READ getUnit CONSTANT)
15
16 public:
17     /// Constructor.
18     PointModel(CurrentValues* mainValue, QString name,
19                bool enableAnalog=false, QObject* parent=nullptr);
20
21     QString getValue() const;
22     QString getName() const;
```

```
23      QDateTime getTimestamp() const;
24      QString getUnit() const;
25      bool getAnalogFlag() const;
26
27  signals:
28      void nameChanged();
29      void valueChanged();
30
31  private:
32      QString m_value;
33      QString m_name;
34      QDateTime m_timestamp;
35      QString m_unit;
36      bool m_enableAnalog;
37  };
```

Snippet 4.3. *Declaring a `PointModel` class with Qt properties. [7]*

As visible in snippet 4.3, a total of four properties are declared: `value`, `name`, `enableAnalog` and `unit`. They are all linked to corresponding member variables, which are listed on the private -section of the class. The getters, setters and value change notifying signals are declared right below the constructor of the class.

It is worth noticing that the macro `Q_OBJECT` must be stated before declaring any properties. The macro indicates to meta-object compiler that the class is using Qt's meta-object features, such as properties or signal-slot mechanisms. [10]

Using the properties in QML

A project declaring properties will also have them available in the QML side of the project. QML is an abbreviation for *Qt Modeling Language* which is basically a markup language used to create user interfaces for Qt applications [10]. It is typical to link a C++ "backend" to a QML "frontend", and Qt's properties are one way to exchange information between these two. To use the properties from example 4.3 in QML, corresponding "QML properties" must be declared like in snippet 4.4:

```
1      property string mainValue: "MainValue"
2      property string cellTitle: "Title"
3      property bool enableAnalog: false
4      property string unit: ""
5
6      // Using the values of the properties in UI components
7      Text{
8          text: mainValue + " " + unit
```

```
9           font.pointSize: 50
10          anchors.verticalCenter: parent.verticalCenter
11          anchors.horizontalCenter: parent.horizontalCenter
12      }
13
14      // ...
```

Snippet 4.4. *Declaring QML properties corresponding to the ones in `PointModel`. [7]*

However, the declared QML properties are not yet linked to the ones defined in C++ side: they only have default values specified. For larger entities (such as a project with multiple QML files and linked properties) the linking is possible with Qt's approach to the *model-view-controller* pattern, with *model-view-delegate*. This is demonstrated in snippet 4.5. It should be noted that the `PointModel` must also be exposed to QML in C++ side, for example in the very main.cpp file of the project. The function for this is the `QQmlContext::setContextProperty()`, which can also be used to link only certain individual properties. This may be a reasonable choice when working with a smaller project.

```
1   GridView {
2       anchors.topMargin: 30
3       id: homeGrid
4       // ...
5
6       // The "pointModel" must be defined in C++ side for
7       // the used QML context to be visible here.
8       model: pointModel
9
10      delegate: HomeScreenComponent {
11          cellTitle: model.modelData.name
12          mainValue: model.modelData.value
13          enableAnalog: model.modelData.enableAnalog
14          unit: model.modelData.unit
15      }
16  }
```

Snippet 4.5. *Linking C++ property values to QML properties. [7]*

The lines 11-14 of snippet 4.5 are the crucial ones linking the values of C++ properties to the values of properties in QML side. The delegate `HomeScreenComponent` is the "QML object" declared in example 4.4, even though it is not directly visible from the code itself.

This is where one of the key benefits in Qt properties lay. The C++ -side properties can get their values updated by an other component and the changes will be reflected to the user's eyes via the QML "bridge". This is just one practical example of the power of properties.

4.3 Analysis and comparison

The property systems in Kanzi and Qt share some features, such as cross-platform support, but they turn out to be significantly different quite fast. The first thing a user will notice is the different syntax when creating and using properties. If the user has actually done some background research on the framework before beginning to use it, they should also be aware of the differences on the targeted use cases for properties. When digging deeper, one will encounter major differences in the implementations and architectures of both systems.

4.3.1 Features

Kanzi and Qt offer ways to set, get and inspect the available properties during runtime. For getting and setting the property values the interfaces are almost identical: In Kanzi, the `Node` offers `Node::getProperty()` and `Node::setProperty()`, while Qt's `QObject` has the methods `QObject::Property()` and `QObject::setProperty()`. Properties can be created by the user and then be used to extend the framework's basic functionalities in the created applications. The common ideology in both systems is that the created properties can affect and define the visual look of the final user interface and the application logic.

One of the most significant differences between the property systems in Kanzi and Qt is the repertoire of available data types. As Qt limits the property's data type to be a type supported by `QVariant`, the list of possible data types is quite long. The amount in Kanzi is measurably smaller. Nevertheless, the properties seem to play a somewhat "larger role" in Kanzi than they do in Qt. This can be deduced from the two system's different ways of exposing the properties to the UI design side, for example. In Kanzi, the properties are registered and exposed to every Kanzi subsystem and Kanzi Studio upon creation of the property types; in Qt, every property whose value is used in QML, has to be explicitly connected between C++ and QML. This means that by default, a created Qt property is visible and usable in C++ -side only.

4.3.2 Implementation

In both Kanzi and Qt, a class using or declaring properties has to be inherited from a generic object class: from `Object` in Kanzi and from `QObject` in Qt. These classes implement the base for the framework-specific metaprogramming features, such as properties and metamethods. Both systems aim to hide the required extra code by offering macros to be used when declaring a property or a property type. These macros typically expand into functions which are used e.g. when querying the metadata of an object.

On the deeper end, Kanzi seems to rely more on a traditional object-oriented class hie-

rarchy. The `PropertyType` is an own and independent class and the `Object` inherits from `MetaObject` and `PropertyObject`. A property in Qt is simply a class member variable, declared to be a property (even though the related object needs to be inherited from `QObject` similarly to Kanzi). Qt is also using its moc -system in the build process of a Qt application, while Kanzi's properties don't require any "Kanzi-specific" compilers in order to work.

4.3.3 Usage

Declaring and registering properties does not differ very much between Kanzi and Qt, although it's nice to notice again the conceptual difference between Kanzi's *property types* and Qt's *properties*. With the examples provided in this thesis the getter and setter functions for Qt's properties are written by hand and linked to the property in its registration. In newer versions of Qt these methods are automatically created [10].

The defining of a property is something that the user doesn't have to do in Qt, but in Kanzi it is required. The property type definition example 4.2 shows a somewhat complicated-looking syntax for defining a property type in Kanzi. Especially the amount of information written with the definition is quite big, and this is simply because the created property type will appear visible and usable in Kanzi Studio. It is up to the user how much information they want to provide with the property type. For example, the tooltip is not necessary. The two syntaxes for registering an attribute value (`metadata.attribute = "value"` and `metadata["Attribute"] = "value"`) exist in parallel because the "key-value -syntax" enables the user to create custom attributes [4].

The main use case for properties is the same in both systems: to affect and control the look and behaviour of UI elements. In Kanzi, the idea of accessing and handling data uniformly between different subsystems is also in a significant role.

4.4 Possible enhancements for Kanzi

The biggest issue in Kanzi's property system is probably its property type's definition syntax. As visible from example 4.2, it can be a thick and complex package. The multiple arguments given to `PropertyType's` constructor can vary as the constructor has several overloads, but that doesn't really clarify the situation very much.

One way to clean up the syntax a bit would be a more uniform metadata attribute registration. Fortunately, this has been taken into account on Kanzi development and the trend is to push the syntax more towards the `metadata.attribute = "value"` -syntax. As the definition is a quite compact package already, it is difficult to modify it much further. Probably the best option at the moment is just to aim for good documentation on each overload of `PropertyType's` constructor.

Another thing worth pointing out is the amount of supported data types for a Kanzi's `PropertyType`. It covers a notably smaller set of data types than Qt's `QVariant`. As Kanzi Engine has its own `Variant` class, which Kanzi uses internally as a property's data type, the amount of its possible classes is "just" 19. However, would be wise trying to find out if there's a real need for that. Since Kanzi is mainly developed towards the wishes of customers, it doesn't make much sense to add features which don't have any real use cases. But sometimes a new innovation or a moment spent on out-of-box -thinking could yield surprising results.

5 CONCLUSION

A property system is one way to extend the reflection functionalities in C++. It can be especially handy when working with larger ecosystems, which possibly use several different programming languages and lots of smaller components or subsystems. Property system combined with a graphics framework opens up a whole new page of possibilities, of which the most important one is the properties' ability to affect, modify and fine-tune the elements in an user interface.

This Bachelor of Science thesis inspected, analyzed and compared two different property systems. The targeted frameworks were *Kanzi*, provided by Rightware, and *Qt*, provided by The Qt Company. Theoretical background and an overall look on the two systems were also in the scope of the subject. As a compressed result, both property systems turned out to be interesting fields of study and viable choices for extending the standard library of C++. The comparison between the two systems didn't reveal anything revolutionary or radical in either ones. Small differences between syntax and inner implementation were expected and found, but based on them it is difficult (if not impossible) and unpleasant to state which one of the frameworks would implement, use or enable the use of a property system better than the other. Both work well on their targeted fields of use, but Kanzi would possibly benefit from a larger set of available property data types, cleaner or less complex property type definition syntax and a more complete documentation.

As C++23 or C++26 aims to bring extended reflection support for C++, the concept of a separate property system might become obsolete. In that sense any further studies focusing only on non-standard C++ property systems do not seem very relevant, even though the reflection proposals don't specifically mention anything about properties. A study comparing a "traditional" third-party property system and the new tools coming with C++23/26 could be sensible, but if the results are not in favor for the new standard library's reflection, then something relevant has been left missing. After all, a property system isn't that much of an irreplaceable feature, but more like a patch for a notable amount of special use cases, which should be coverable within a comprehensive set of basic reflection tools. It is a whole different story then if the conversion work would actually be worth doing (if it turned out to be possible), considering all the work spent on creating the property system in the first place. Most likely not, at least in the very first ten years after the release of C++26.

REFERENCES

[1] M. de Bayser and R. Cerqueira. A system for runtime type introspection in c++. In: *Brazilian Symposium on Programming Languages*. Springer. 2012, 102–116.

[2] *Definition of Connected Car – What is the connected car? Defined*. AUTO Connected Car News. URL: http://www.autoconnectedcar.com/definition-of-connected-car-what-is-the-connected-car-defined/ (visited on 03/13/2019).

[3] S. Ducasse, S. Marr, and C. Seaton. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In: *ACM SIGPLAN Notices* 50.6 (2015), 545–554.

[4] *Kanzi 3.6.4 Documentation*. Rightware. 2019.

[5] *Kanzi Connect - Rightware*. Rightware. URL: https://www.rightware.com/kanzi-connect (visited on 03/13/2019).

[6] Y. Li, T. Tan, and J. Xue. Understanding and Analyzing Java Reflection. In: *arXiv preprint arXiv:1706.04567* (2017).

[7] T. Loikkanen, V. Nguyen, and J.-M. Ryttyläinen. *TUT Course TIE-20200, Project work of group 40 source code*. 2018.

[8] G. McCluskey. Using Java Reflection. In: *Oracle Technology Network* (1998). URL: https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html (visited on 02/17/2019).

[9] *Microsoft Docs: property (C++)*. Microsoft. Nov. 4, 2016. URL: https://docs.microsoft.com/en-us/cpp/cpp/property-cpp?view=vs-2017 (visited on 02/17/2019).

[10] *Qt 5.12 Documentation*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5 (visited on 02/17/2019).

[11] *Qt Documentation - All Modules*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5/qtmodules.html (visited on 02/27/2019).

[12] *Qt Documentation - qmake Manual*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5/qmake-manual.html (visited on 02/28/2019).

[13] *Qt Documentation - Qt Creator Manual*. The Qt Company. 2019. URL: https://doc.qt.io/qtcreator/index.html (visited on 02/27/2019).

[14] *Qt Documentation - Qt Designer Manual*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5/qtdesigner-manual.html (visited on 02/27/2019).

[15] *Qt Documentation - User Interface Compiler (uic)*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5/uic.html (visited on 02/28/2019).

[16] *Qt Documentation - Using the Meta-Object Compiler (moc)*. The Qt Company. 2019. URL: https://doc.qt.io/qt-5/moc.html (visited on 02/28/2019).

[17] D. Sankel. *Working Draft, C++ Extensions for Reflection*. Standard C++ Foundation. Aug. 11, 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf (visited on 02/26/2019).

[18] B. C. Smith. Procedural reflection in programming languages. PhD thesis. Massachusetts Institute of Technology, 1982.

[19] *Source code of qt5 - Woboq Code Browser*. The Qt Company. URL: https://code.woboq.org/qt5/ (visited on 03/30/2019).

[20] H. Sutter. *Trip report: Winter ISO C++ standards meeting (Kona)*. Feb. 23, 2019. URL: https://herbsutter.com/2019/02/23/trip-report-winter-iso-c-standards-meeting-kona/ (visited on 02/26/2019).

[21] *Ui design software - Rightware Kanzi*. Rightware. URL: https://www.rightware.com/kanzi (visited on 02/26/2019).

[22] J. Weller. *Reflections on the reflection proposals*. Meetingcpp GmbH. Mar. 2, 2017. URL: https://meetingcpp.com/blog/items/reflections-on-the-reflection-proposals.html (visited on 02/24/2019).