Qt Group

User Interface Design: Functionality, Tooling, and Workflow

The key ingredients to efficiently navigate the complexity of modern UI software

2024

Foreword

Rich functionality, reliability, and seamless usability are key software features inherent to modern users' collective imagination about digital devices. But user expectations are constantly rising and today high performance, connectivity, advanced graphics, and multi-platform operability have become key competitive advantages between products that do their job and products that set **new market standards**.

In successful digital devices, an enticing visual appearance encloses powerful functionality that is unleashed in a simple and natural way. As a defining factor, a plain user experience and neat visuals conceal the complexity of the software architecture. Consequently, the amount of work—in terms of UI and UX design, lines of code, third-party solutions, and background services,—the hours of testing, bug fixing, and optimization that led to the ultimate result cannot be imagined by most people.

However, software *developers* and *managers* know how complex and wearing it is to deliver a product that just feels like a natural extension of our senses and abilities. The right tools make the real difference not only in reducing production cost and time-to-market but in securing the overall success of the enterprise.

At Qt Group, we have tackled such complexities for about thirty years and identified the key features that a UI framework should include to deliver iconic digital experiences across all types of electronic devices. The Qt framework pioneered UI app creation when wireframes were drawn on paper. Today that the user experience is defined by real-time collaborative authoring tools, Qt empowers the automatic conversion of designs into fully functional applications and their validation in immersive 3D environments.

We take pride in being at the forefront of the digital revolution that is transforming industries and impacting our daily lives. Our experience and expertise organized in this new eBook shall help our reader choose the right tools for succeeding in the modern hyper-competitive market.

Qt Group

Contents

1. Foundations **4**
The UI Application 6
The UI Framework 6
The Middleware 7
Cross-Platform Deployment 8

2. Performance **10**
Event-Driven Architecture 12
Profiling 13
Quality Assurance 14

3. Graphics **15**
Real-Time Graphics 17
Data Binding 18
The Interplay of 2D and 3D Graphics 19
Immersive 3D Graphics 20
Physically Based Rendering 21
HDR Lighting 22
Postprocessing Effects 22
Theming 22
Embedding Game-Engine Graphics 23

4. Workflow **24**
An End-To-End Solution 26
Design and Development 27
Automated Testing 28
Standardization and Customization 28
Platform Engineering 29
Ownership 30

1. Foundations

User interface (UI) applications are an essential component of modern software and of the fast-growing set of software-driven devices that inhabit our daily lives. Intuitive design, cohesive user experience, prompt responsiveness, high degree of customizability, compatibility with different devices and platforms are all core features of UI applications.

Ultimately, their purpose is to provide users with a flawless and intuitive experience while interacting with software and digital devices. And despite their wide use, there may be misconceptions about what qualifies as a true UI application, and what not. Such mistakes may negatively impact our judgment in the choice of the **right development framework** for the creation of UI software.

The UI Application

Consider **graphics.** It is a defining aspect of UI applications, but it is one aspect among many others. A UI application is not just graphics rendered on a screen—rather, it is a visual interface to live data, services, and functions that run in real time in the backend. Nor is it a standalone software that, while rendering a scene on the screen, is allowed to consume all the available system resources—but a bundle of UI elements, like widgets, charts, interaction areas, and views, that coexist on one or multiple screens to provide users with the information they need for several different tasks in parallel.

Several processes compete with one another for a limited set of system resources—such as memory, processing power, storage, and often also power on low-energy devices—and we'll see some smart solutions to handle concurrent tasks efficiently.

More in general, the internal structure, visual layout, and logic flow of UI applications are unique and radically different from those of other types of visual software like, for instance, computer generated imagery (CGI) or video games. As such, their proper and efficient management requires dedicated tooling.

The UI Framework

The rich set of features and services that enter a UI application needs a development environment providing generic functionality and resources that designers and engineers can use as a foundation for the creation of UI applications. The UI framework offers the creative environment for UI designers and backend developers alike to realize their product vision to the full extent.

On the **design** side, it includes content authoring tools, assets libraries, configurable graphics pipelines and visual effects, real-time preview and testing on emulator, and much more to devise the end product's user experience.

For **developers,** it offers reference code and applications, APIs, compilers, profilers, code toolsets, libraries, and support for connectivity protocols that turn designs and prototypes into fully functional, future-proof UI applications.

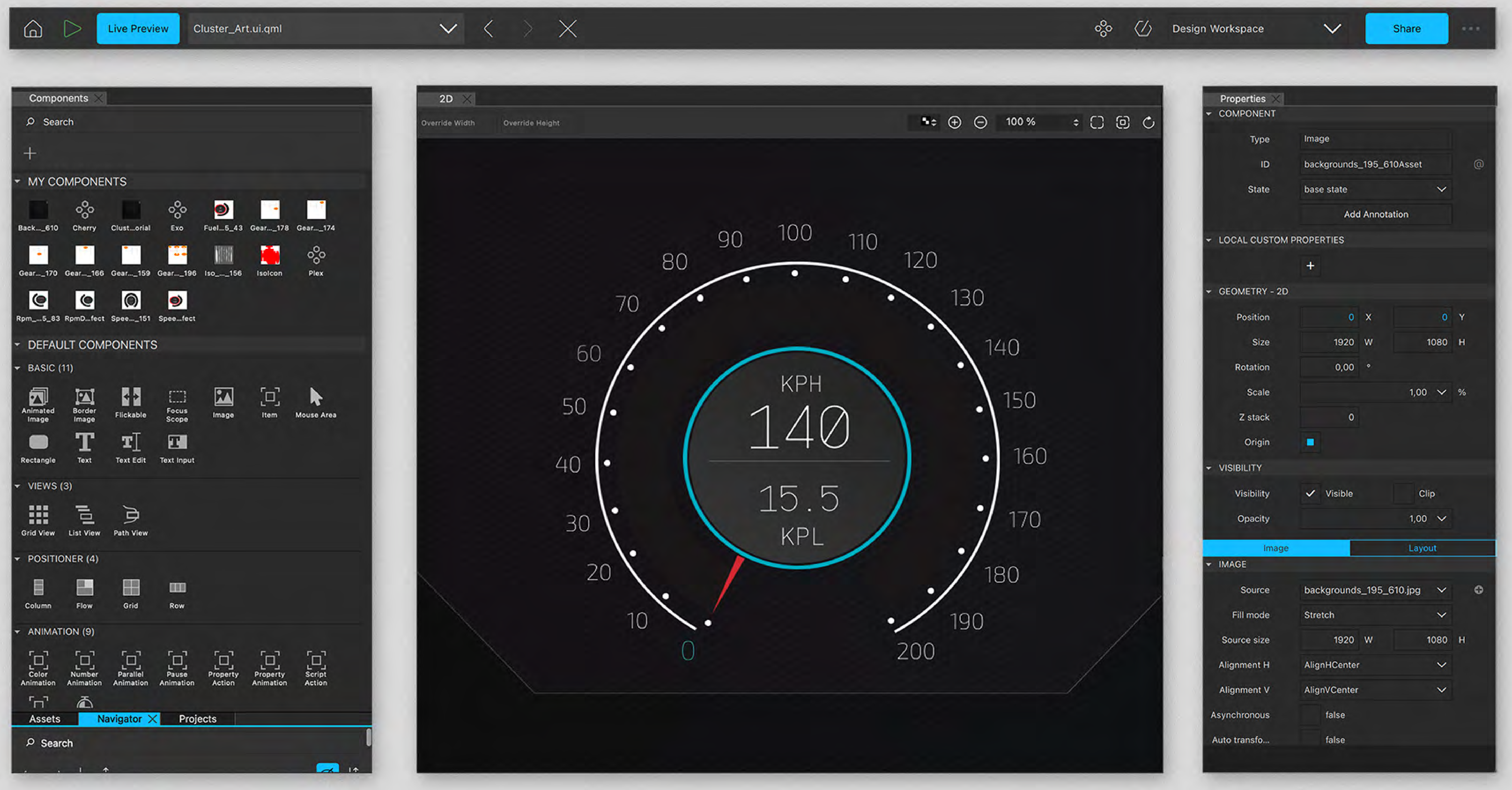
Each function that the framework provides represents less code that developers need to write, hence faster time-to-market and increased reliability. Wizards guide users interactively through the project stages, for instance, by creating the necessary files, solving dependencies, and specifying settings based on the use case. Semantic highlighting on code, syntax check, auto-completion, refactoring actions are other useful features that help create quality code faster.

Clearly, the more complete the UI framework offering is across all stages of software development, the less work is required for designers and developers to implement their ideas. Adopting a comprehensive framework relieves users of complex clerical activities and error-prone routines to let them focus on what really matters—the ultimate user experience.

The Middleware

Besides providing tools and functionality, the UI framework works as *middleware,* namely as a neutral development environment providing common APIs for software creation, *abstracting away from the low-level details of the hardware and operating system.* For instance, it provides cross-platform development interfaces for accessing resources (e.g. file system, connectivity, IO/sensors) on any system and pre-built components for common tasks—such as UI elements, networking, and data management systems.

This so-called *compatibility layer* acts as a bridge between the application and the underlying architecture, and it is at the heart of the **platform engineering** strategy that is being successfully implemented across industries to ease the developer workload while increasing efficiency and quality, especially when working on varied product portfolios and multiple devices.



COMPONENT
Type: Image
ID: backgrounds_195_610Asset
State: base state
Add Annotation
LOCAL CUSTOM PROPERTIES
GEOMETRY - 2D
Position: 0 X 0 Y
Size: 1920 W 1080 H
Rotation: 0,00 °
Scale: 1,00 %
Z stack: 0
VISIBILITY
Visibility: ✓ Visible, Clip
Opacity: 1,00
IMAGE
Source: backgrounds_195_610.jpg
Fill mode: Stretch
Source size: 1920 W 1080 H
Alignment H: AlignHCenter
Alignment V: AlignVCenter
Asynchronous: false
Auto transfo...: false

Cross-Platform Deployment

Modern software applications run on a variety of devices. Smartphones, tablets, laptops, and desktops have been out already for decades and enjoy a broad ecosystem of cross-platform applications. Today, familiar appliances that used analog interfaces till recent times are being equipped with rich digital UIs and new functionality stemming from digitalization.

An interesting example of software that needs to operate on different environments is offered by so-called **companion apps** that extend the capabilities of home appliances by enabling distant interaction and cloud-based services. Analogous use cases stem from the industrial, medical, and automotive domains where the same or largely similar user experiences are deployed on devices that have distinct form factors, varying computational resources, and different hardware and operating systems.

Cross-platform software development has become a necessity across industrial verticals to ensure broad usability, extra functionality, and market traction. And while this may sound like a further challenge, with the right tools, it is an opportunity.

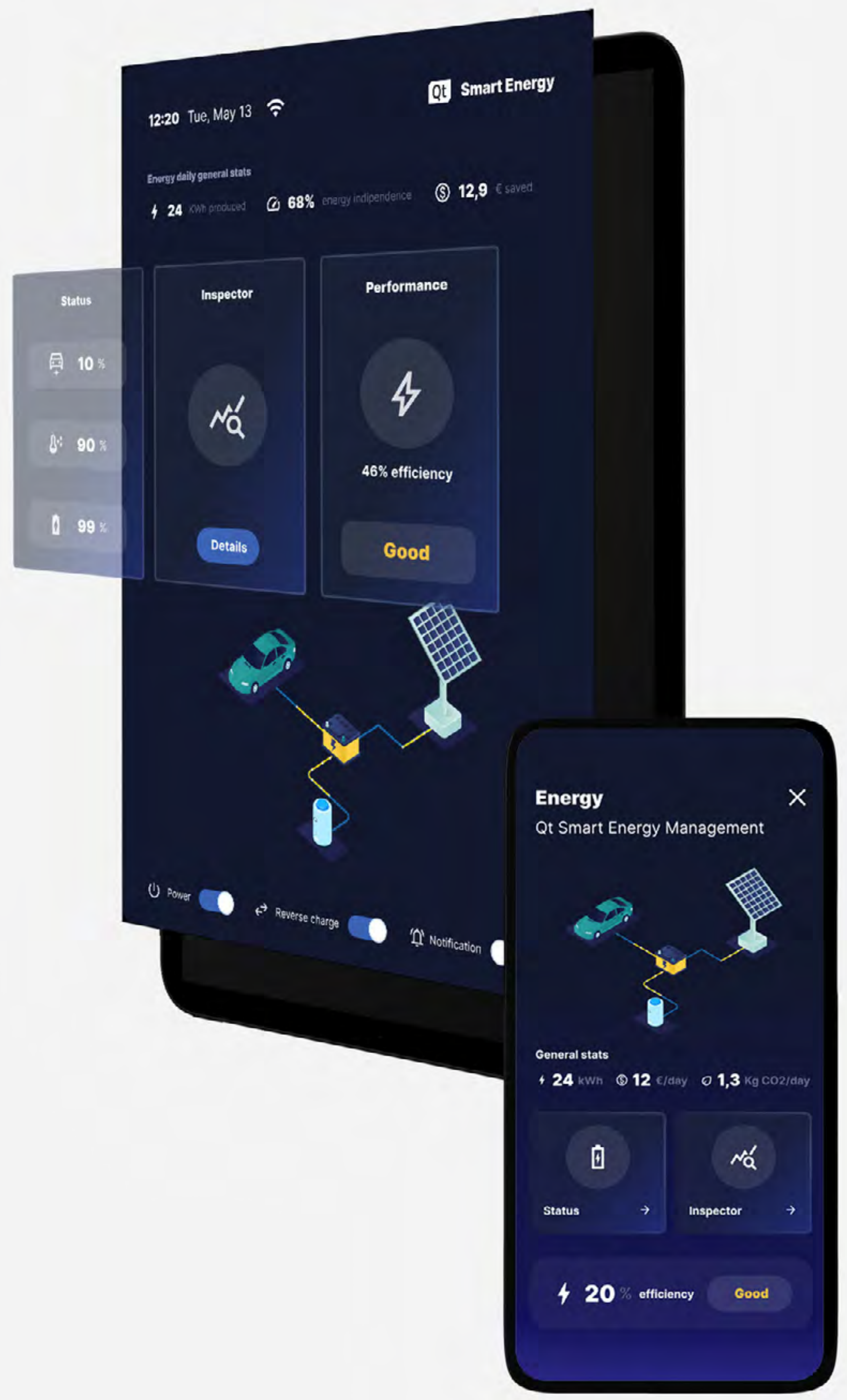
Clearly, re-writing code for each different target is not a viable solution. If the aim is to *code once and deploy on many platforms*, adopting a **platform-agnostic framework** grants the required flexibility and resources. Minimally, this implies

- reduced costs—time and materials
- independence from hardware suppliers
- faster time-to-market
- higher quality at scale

For end users and branding, *cross-platform design* ensures that a uniform user experience reaches a wide audience, strengthening brand identity via a consistent look-and-feel across the entire product portfolio.

Finally, pre-defined **software packaging** solutions enable deployment on a wide range of hardware types and operating systems, ensuring optimal performance on each target.

As the complexity of targeting multiple environments can be tamed by using a comprehensive cross-platform UI framework, the benefits of such an approach are vast, both in terms of *creative opportunities*—as in the case of companion apps—and *efficiency*.



2. Performance

While on the development side, the UI framework provides libraries and solutions for app creation on PC, on the deployment side, it provides the **UI engine** running the applications on the end-user devices, also known as **runtime.** A smartwatch, a digital automotive cockpit, or the display of smart-home appliances all include an instance of the framework's UI engine running the app on their specific hardware and software architecture.

Considering the variety of devices and their often very limited computational power and energy supply, it looks almost magic that, no matter the underlying system, a single runtime engine may control every function's and service's lifecycle, their interaction with one another, and the resource allocation for each task. The UI engine is specifically designed to *handle competing processes* with particular attention to performance.

Event-Driven Architecture

Digital devices are usually resource-constrained devices with very limited processing power and storage capabilities. The devices powering our appliances, cars, and wearables have, *for instance, much more limited capabilities than the powerful desktop PCs used for home entertainment and gaming*. And more in general, with UI software, it is rather exceptional that a single application is allowed to use all the resources available or to have unlimited startup time—as in the gaming case. The embedded devices used for industrial applications need to snap-boot and have only scarce computational resources to run several functions and services in parallel.

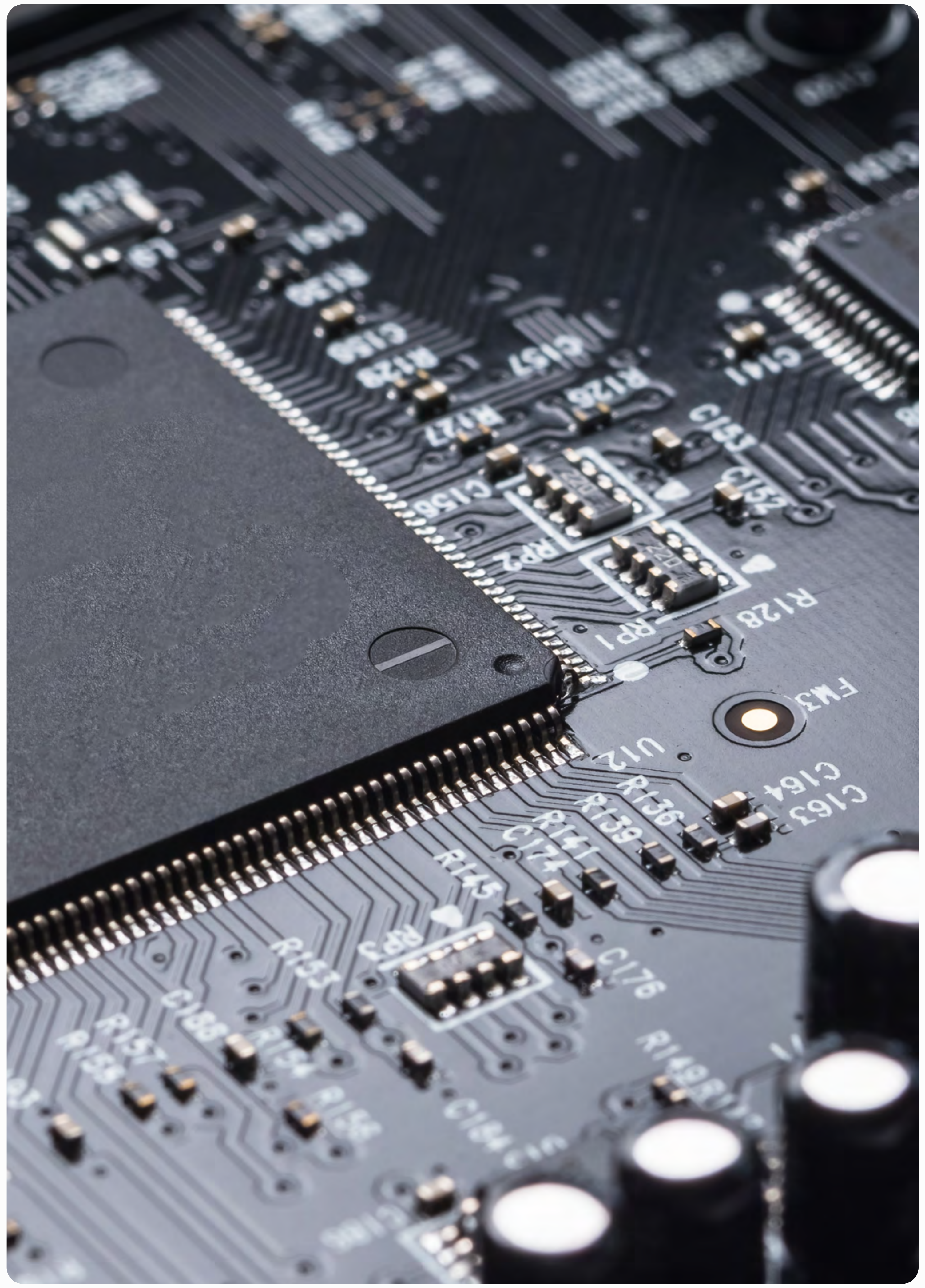
Nonetheless, the final performance of the application and the quality of its graphics are not dictated solely by the hardware. The UI framework offers a series of ready-made solutions to optimize resource consumption in constrained environments, enabling a multitude of parallel services and advanced graphics with a **low footprint.**

The underlying **event-driven architecture** uses events to trigger functionality and to communicate across decoupled services. Applications remain idle when not in use, minimizing the amount of memory, CPU, and GPU needed. Such **on-demand behavior** frees computational resources that can be dynamically allocated to other applications—and reduces energy consumption when the supply is limited.

In addition to UI virtualization, the UI framework supports:

- **Background loading** of UI and resources
- **Threading** to handle the concurrent execution of tasks across multiple CPU cores
- **Memory management** to automatically handle the allocation and deallocation of memory for objects
- **Caching** for images, fonts, and other resources, helping to speed up the loading and display of these resources
- **Optimized data structures** that are more efficient than the standard C++ data structures
- **Compilation options** including link-time optimization to find opportunities for optimization of the overall program.

The UI framework helps developers improve the efficiency and responsiveness of the UI by means of mechanisms that streamline the UI's internal processes and reduce the memory footprint.

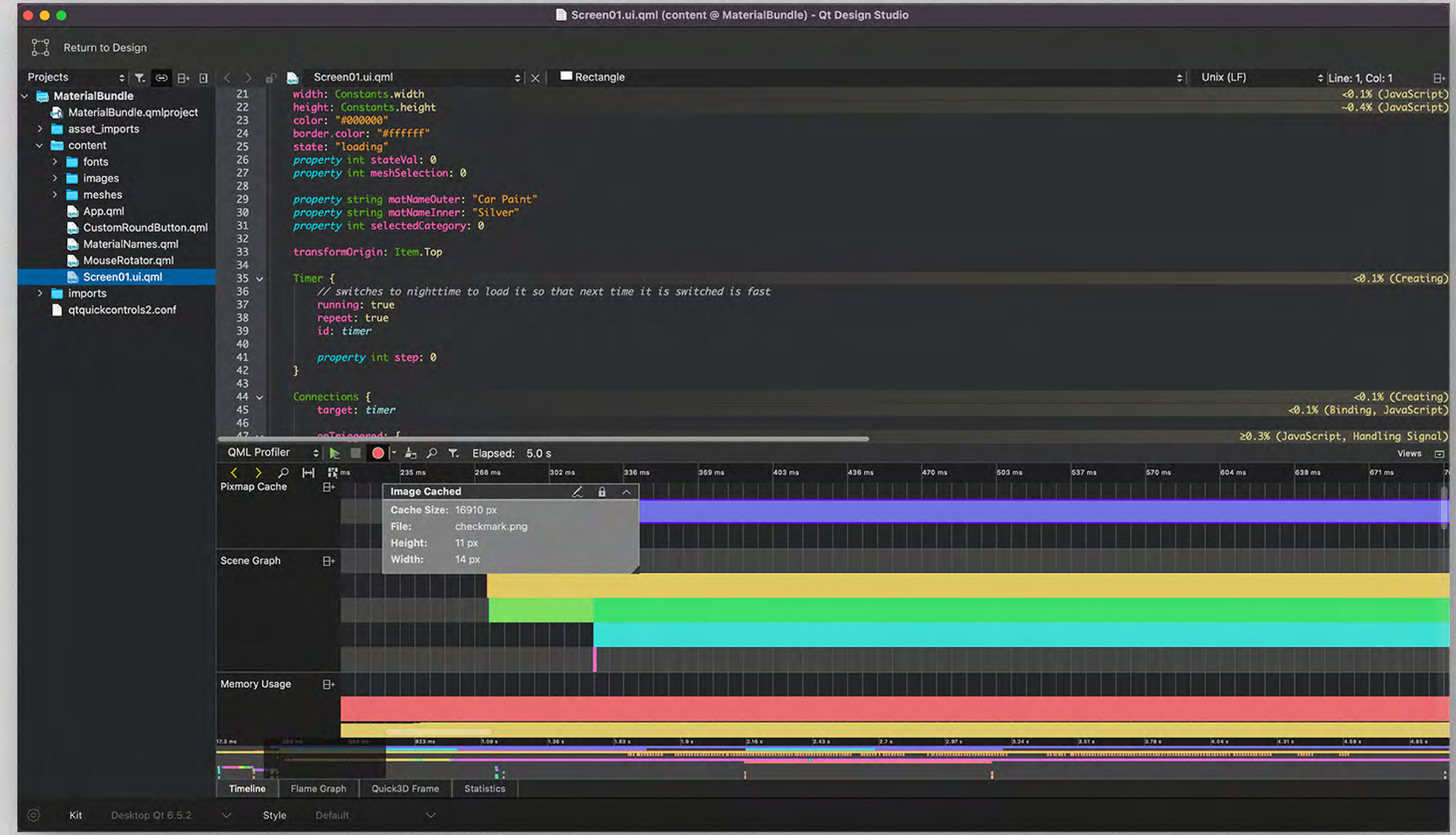


Profiling

While a **prototype** can show the application's intended behavior and look on a desktop PC—a sort of *unconstrained environment* that abstracts over the resource needs—rarely can such an application be ported to the actual target at this stage.

Profiling involves collecting data about how the application uses resources such as CPU, memory, and network bandwidth and analyzing this data to identify bottlenecks, lags, and memory leaks. Too complex 3D models, high-density shaders, extensive use of shader effects are frequent examples of such bottlenecks that may prevent an application from performing steadily at the optimal 60 frames per second. The profiler helps detect such rendering bottlenecks but also suboptimal code, background tasks that consume too many resources, memory leaks, unoptimized graphics, too frequent UI updates, and other similar issues that, without the right tools, would require a long time and unnecessary effort to be detected.

When the actual target hardware is not accessible, the UI framework's **emulating capabilities** allow developers to examine the app's behavior on the target's form factor and processing resources from their PC *as if it was running on the actual device.*



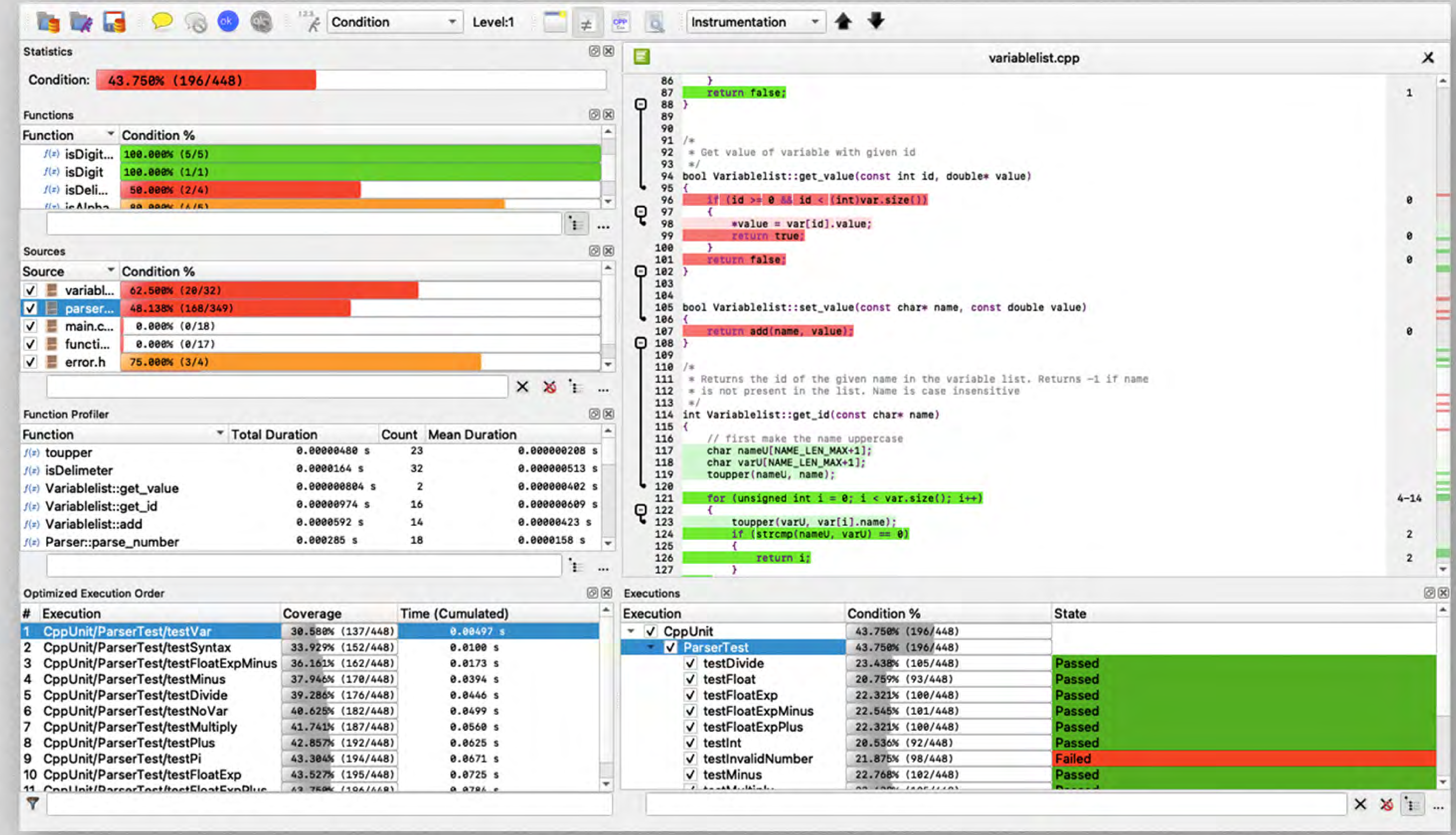
```
width: Constants.width                                         <0.1% (JavaScript)
height: Constants.height                                       ~0.4% (JavaScript)
color: "#000000"
border.color: "#ffffff"
state: "loading"
property int stateVal: 0
property int meshSelection: 0

property string matNameOuter: "Car Paint"
property string matNameInner: "Silver"
property int selectedCategory: 0

transformOrigin: Item.Top

Timer {                                                        <0.1% (Creating)
    // switches to nighttime to load it so that next time it is switched is fast
    running: true
    repeat: true
    id: timer

    property int step: 0
}

Connections {                                                  <0.1% (Creating)
    target: timer                                   <0.1% (Binding, JavaScript)

    onTriggered: {                        ≥0.3% (JavaScript, Handling Signal)
```

```
Image Cached
Cache Size:  16910 px
File:        checkmark.png
Height:      11 px
Width:       14 px
```

In case the hardware is not yet agreed upon, ready, or available (as in the case of chip shortage), the possibility to develop and test software on PC independently from the final hardware ensures that production can proceed faster and steadily with the least impact on time-to-market.

Quality Assurance

If in addition to a fluid performance, the application needs to meet the industry standards of safety and reliability and comply with the requirements set by notified bodies, the presence of integrated quality assurance tools helps automate the testing process, identify untested code, and detect deviations from coding directives. While in general beneficial to raise the quality and trust of the software product, such QA component becomes essential in case of safety-critical applications and certification thereof as, for instance, in the medical sector.



Statistics

Condition: 43.750% (196/448)

Functions

Function	Condition %
isDigit...	100.000% (5/5)
isDigit	100.000% (1/1)
isDeli...	50.000% (2/4)

Sources

Source	Condition %
✓ variabl...	62.500% (20/32)
✓ parser...	48.138% (168/349)
✓ main.c...	0.000% (0/18)
✓ functi...	0.000% (0/17)
✓ error.h	75.000% (3/4)

Function Profiler

Function	Total Duration	Count	Mean Duration
toupper	0.00000480 s	23	0.000000208 s
isDelimeter	0.0000164 s	32	0.000000513 s
Variablelist::get_value	0.000000804 s	2	0.000000402 s
Variablelist::get_id	0.00000974 s	16	0.000000609 s
Variablelist::add	0.0000592 s	14	0.00000423 s
Parser::parse_number	0.000285 s	18	0.0000158 s

Optimized Execution Order

#	Execution	Coverage	Time (Cumulated)
1	CppUnit/ParserTest/testVar	30.580% (137/448)	0.00497 s
2	CppUnit/ParserTest/testSyntax	33.929% (152/448)	0.0100 s
3	CppUnit/ParserTest/testFloatExpMinus	36.161% (162/448)	0.0173 s
4	CppUnit/ParserTest/testMinus	37.946% (170/448)	0.0394 s
5	CppUnit/ParserTest/testDivide	39.286% (176/448)	0.0446 s
6	CppUnit/ParserTest/testNoVar	40.625% (182/448)	0.0499 s
7	CppUnit/ParserTest/testMultiply	41.741% (187/448)	0.0560 s
8	CppUnit/ParserTest/testPlus	42.857% (192/448)	0.0625 s
9	CppUnit/ParserTest/testPi	43.304% (194/448)	0.0671 s
10	CppUnit/ParserTest/testFloatExp	43.527% (195/448)	0.0725 s

variablelist.cpp

```
    }
    return false;
}


/*
 * Get value of variable with given id
 */
bool Variablelist::get_value(const int id, double* value)
{
    if (id >= 0 && id < (int)var.size())
    {
        *value = var[id].value;
        return true;
    }
    return false;
}


bool Variablelist::set_value(const char* name, const double value)
{
    return add(name, value);
}

/*
 * Returns the id of the given name in the variable list. Returns -1 if name
 * is not present in the list. Name is case insensitive
 */
int Variablelist::get_id(const char* name)
{
    // first make the name uppercase
    char nameU[NAME_LEN_MAX+1];
    char varU[NAME_LEN_MAX+1];
    toupper(nameU, name);

    for (unsigned int i = 0; i < var.size(); i++)
    {
        toupper(varU, var[i].name);
        if (strcmp(nameU, varU) == 0)
        {
            return i;
        }
```

Executions

Execution	Condition %	State
✓ CppUnit	43.750% (196/448)	
✓ ParserTest	43.750% (196/448)	
✓ testDivide	23.438% (105/448)	Passed
✓ testFloat	20.759% (93/448)	Passed
✓ testFloatExp	22.321% (100/448)	Passed
✓ testFloatExpMinus	22.545% (101/448)	Passed
✓ testFloatExpPlus	22.321% (100/448)	Passed
✓ testInt	20.536% (92/448)	Passed
✓ testInvalidNumber	21.875% (98/448)	Failed
✓ testMinus	22.768% (102/448)	Passed

3. Graphics

Graphics are an essential element of UI applications. Indeed, what differentiates UI applications from other types of software is the presence of a visual interface enhancing user interaction.

Visuals make software more intuitive and immediate, easier to use and to understand. Colors, icons, and animations provide visual cues or status updates in a non-invasive way. Charts, graphs, or maps help display complex information and data. Advanced 3D graphics represent real-world objects and life-like situations on the UI. And in addition to their utility, there is an undisputed aesthetic value in the creation of visually appealing software—good UX design and pleasant visuals attract more users and reinforce brands.

COLOR
#a80420

To deliver the level of interactivity and the life-like situations that UI apps aim to model, the UI framework provides rich 2D and 3D tooling for the creation of various types of visual content, ranging from basic layout design to complex 3D scenes.

This is also why, in terms of graphics capabilities, the UI framework includes so many features typical of modern game engines, but also many others that are relevant to deliver high-quality graphics on a variety of devices and for the most diverse use cases.

Real-Time Graphics

In UI applications, the images on the screen *dynamically change based on user interaction or live data flow.* Real-time animations give feedback to the user in a series of subtle ways that contribute to making the *user experience engaging and natural.* A button expanding while hovering over it, popping up when pressed may pass almost as unnoticed but is an important cue of the device's responsivity to the user's action. Real-time graphics enable **interactive visualizations** of data, both in the form of 2D charts and graphs and in that of complex 3D objects.

These types of visuals are substantially different from the computer graphics seen in films and video games (CGI) for the creation of characters, environments, and effects, where the level of interactivity is null or very limited and visual elements are rendered once and for all. On embedded devices, UI applications rely on *real-time rendering* to provide a dynamic and interactive user experience.

Data Binding

At the core of the possibility for the UI application to dynamically and interactively represent the world, there is the ability to l*ink and sync visual properties of the UI to real objects and data:* the fuel level displayed on the car's dashboard represents the real car's fuel level, the heart rate chart on a smartwatch shows that of the user, and so forth.

Advanced data-binding mechanisms are a core UI framework's tool enabling such connection between UI visuals and real-world data by simple drag-and-drop and automatic synchronization.

Another crucial and rather unique feature of the UI framework is the ability to integrate 2D with 3D elements and have them play harmoniously together.

12:00
22°
Anthony
5G
1.5 km
Grand road avenue
44 min
12:44
89 km
km/h
25
90 km/h
Now Playing
Bon Jovi
Self driving
D
NORMAL
Odo 17321 km
50 %
164 km

The Interplay of 2D and 3D Graphics

While the UI layout is typically created by means of 2D graphics tools enabling the easy setup of ground elements, like frames, buttons, icons, charts, or text, there is a growing trend across various industries to **increase photorealism** by including 3D el-ements within the UI. Flat 2D elements are typically used to represent controls and other types of abstract or symbolic information, while 3D objects represent real-world entities and even complex situations.

2D and 3D graphics play complementary roles and provide together the right balance between performance and realism.

Where 3D graphics add an extra dimension of expressivity, 2D elements are simpler to create and use and have a lower footprint.

Today, the interplay of 2D and 3D graphics has reached such a level of quality and fluidity that complex situations can be represented in the 3D space and seamlessly accessed by means of simple 2D controls. For these two types of graphics to play well together, the UI framework should ensure perfect **synchronization** between the 2D and 3D elements.

Immersive 3D Graphics

With 3D graphics, complex life-like situations can be quickly and easily grasped at a glance. As a well-known example, advanced driving assistance systems (ADAS) in automotive displays increasingly include the real-time rendering of the car in its actual driving environment. To provide drivers with a clear, blind-spot-free view of the surrounding of the car, a 360° camera feed from the physical car creates the 3D scene where the rendered car lives. Real-time reflections, shadows, and other effects contribute to making the resulting 3D scene on the dashboard hardly distinguishable from the physical car's situation.

Similar applications that take advantage of real-time 3D graphics for the creation of **avatars** or **digital twins,** often within the space of virtual or augmented reality, are being developed across industries. In the medical sector, 3D graphics provide interactive visualizations of organs and tissues that can be used to aid in diagnosis, surgical planning, and medical research. Here, diagnostic imagining essentially relies on hyper-realistic 3D visuals to identify the causes of an illness or to confirm a diagnosis.

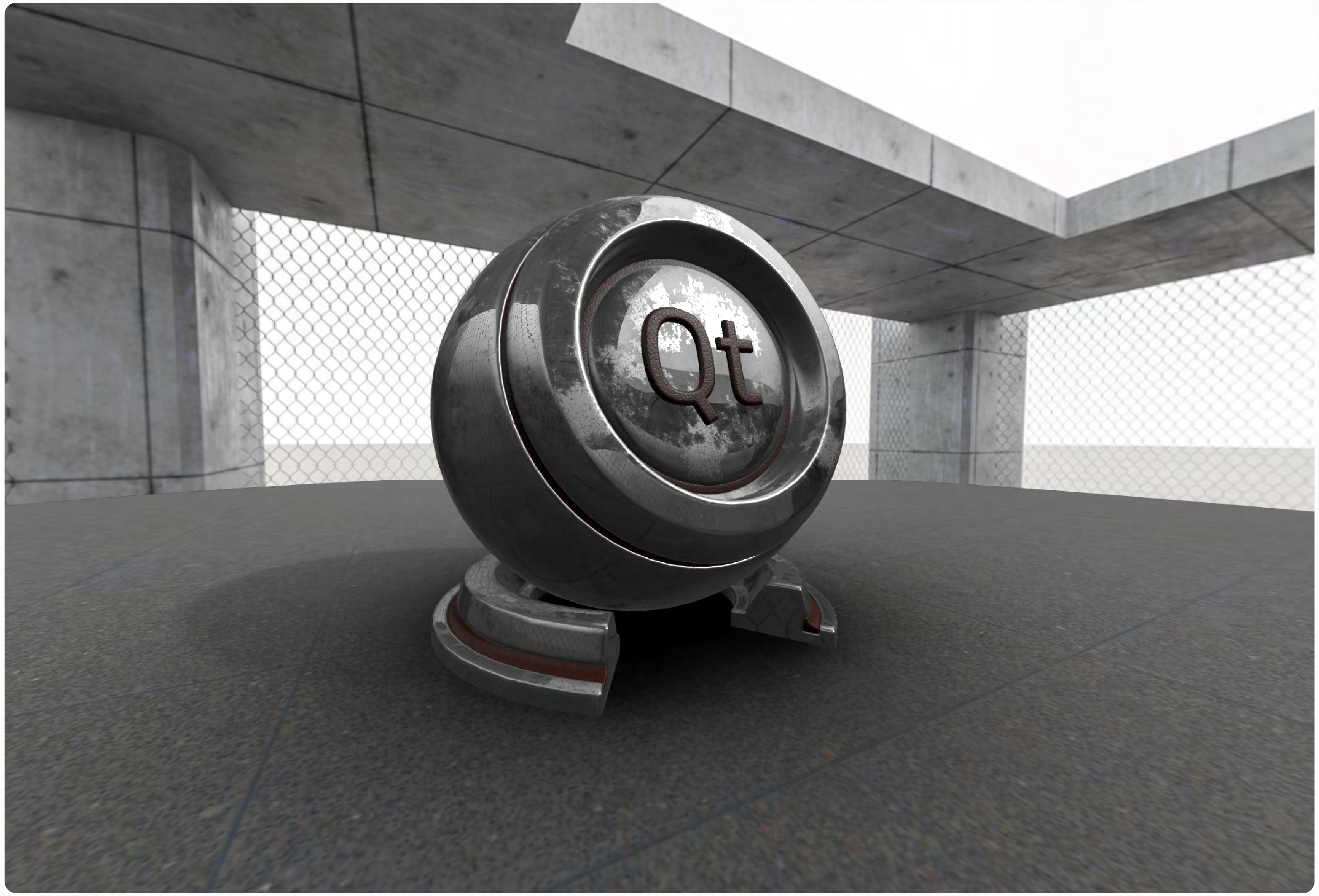
Everywhere, highly detailed and articulated 3D representations of the world increase safety, improve efficiency, reduce costs, and enhance the overall quality of the product or service.

Clearly, the development of such visually rich UI applications requires powerful design tools for 2D and 3D graphics and rendering engines capable of displaying with high fidelity and in real time. This is why, in terms of graphics capabilities, the UI framework includes many features typical of the most advanced **graphics engines.**

Physically Based Rendering

By including a physically based rendering (PBR) pipeline, the UI framework ensures a more accurate and realistic rendering of materials and light that simplifies the creation of high-quality graphics. By adhering to such a *standard,* as specified for instance in the glTF format, UI designers readily gain **compatibility** with general-purpose material models—like wood, leather, rubber, metal, etc. PBR allows for more consistent and predictable results across different lighting conditions and materials resulting in a reduced need for technical artists to create custom shaders for different surface types.

As the PBR standard is increasingly popular in the 3D design community, designers also readily get access to thousands of 3D models available online that can be used and customized according to the UI application's needs.



HDR Lighting

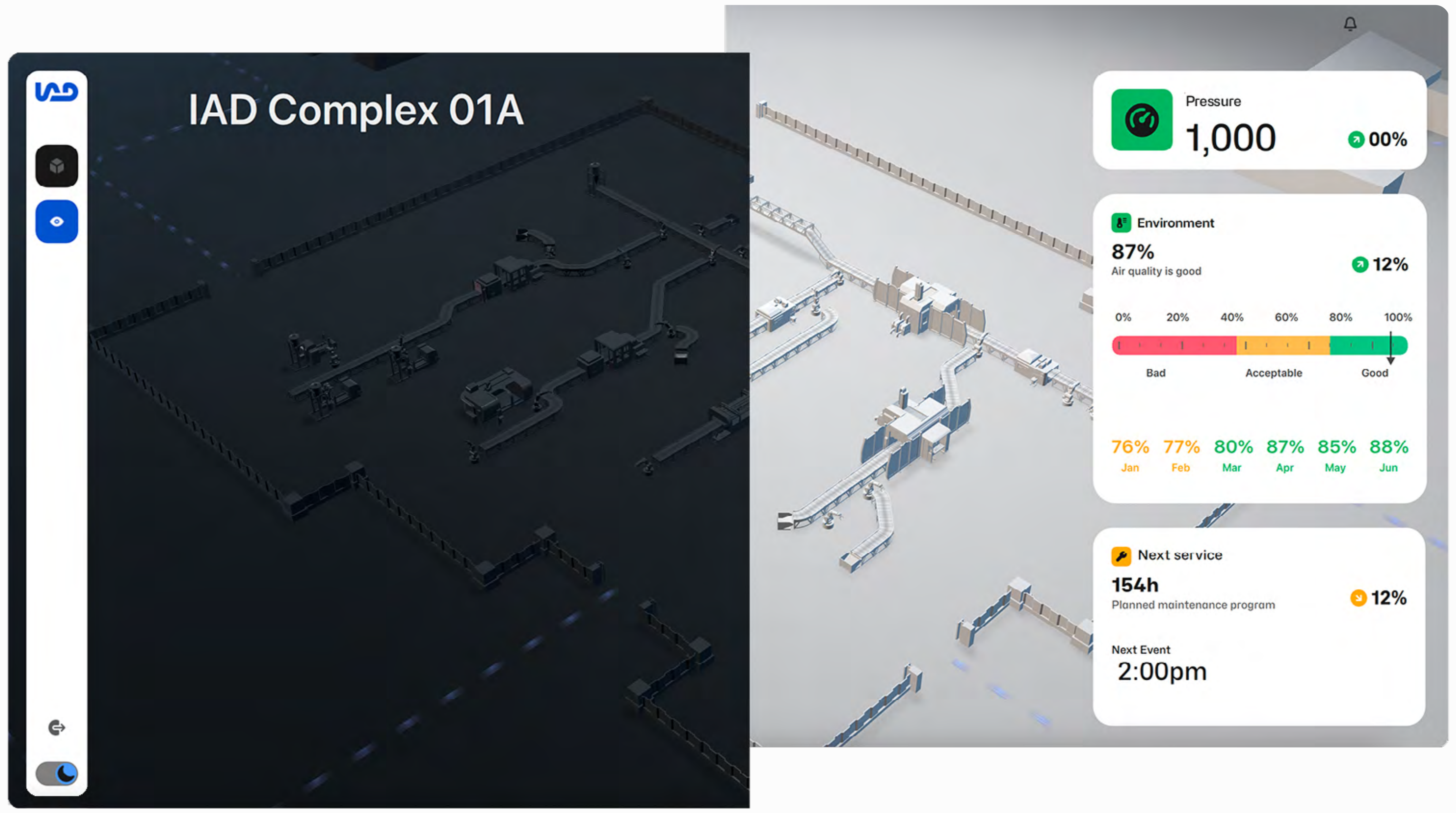
High Dynamic Range lighting allows the creation of **photoreal-istic,** lifelike 3D environments that are highly detailed, vivid, and accurate. By capturing and displaying a wider range of bright-ness and colors than traditional pictures, HDR images provide a basis for more accurate lighting and shading calculations in the 3D scene, resulting in more realistic and dynamic lighting effects.

Postprocessing Effects

Postprocessing effects apply before each frame is rendered by the actual graphics hardware to improve its visual quality or tweak the overall look and feel of the scene with little setup time. As the quality standard is largely set by the CGI offering—which, anyhow, is not real-time—the UI framework's graphics subsystem needs to include advanced algorithms to eliminate any glitches that can reduce photorealism. For instance, an-ti-aliasing removes from the rendering the jagged edges that may appear when a coarse model geometry is used.

Theming

Control over the theme and overall appearance of the UI is fundamental for ensuring a consistent look and feel of the UI across applications and platforms. Theming can include chang-es in colors, fonts, and icons, but also the overall restructuring of the UI by adding or removing widgets for better adaptation to different form factors. Such operations are essential to ensure, for instance, consistent branding and core functionalities across a varied product portfolio.

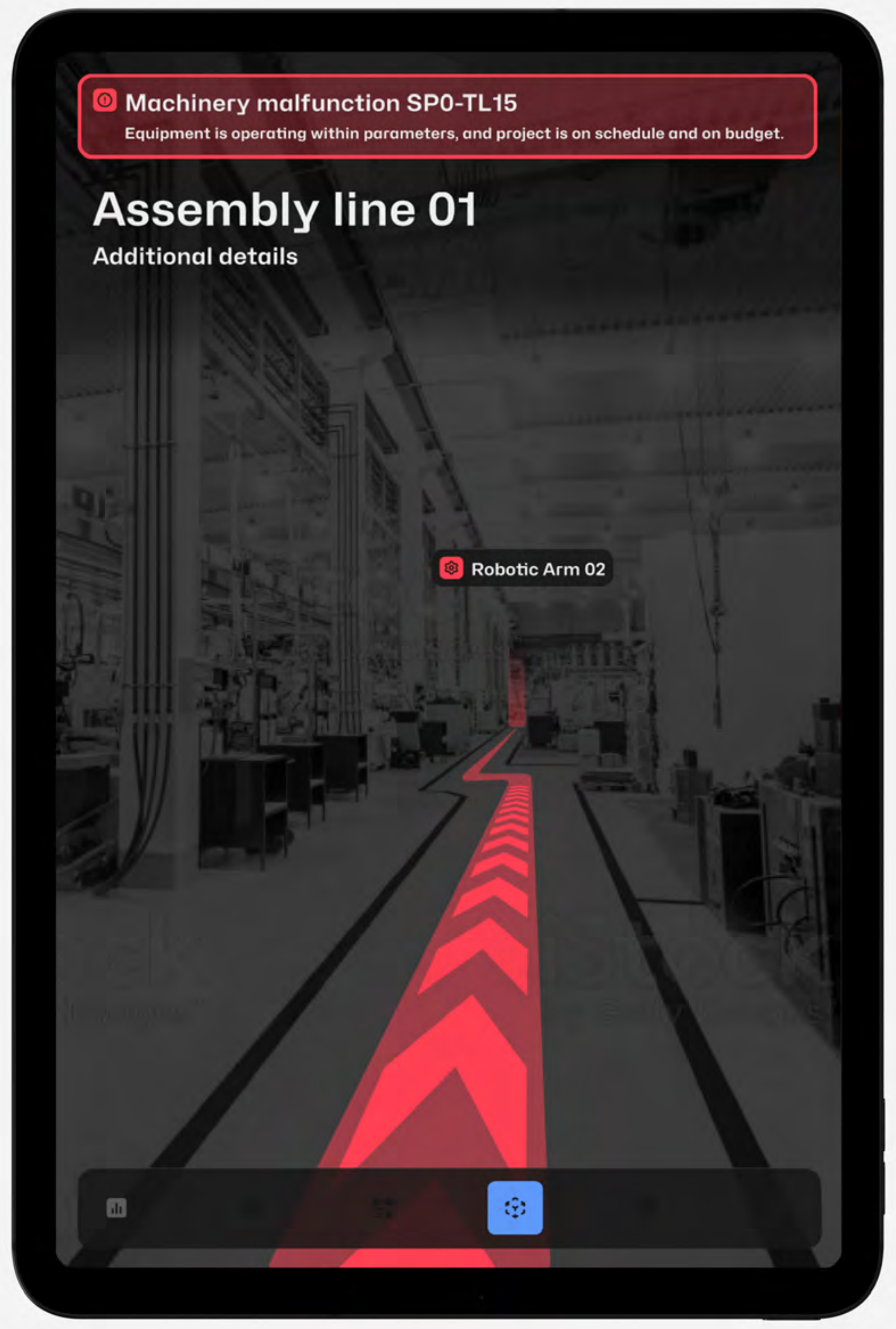


Embedding Game-Engine Graphics

A state-of-the-art UI framework typically supports different types of graphics APIs for the creation of advanced 3D visuals, such as OpenGL, OpenGL ES, Vulkan, Direct3D, and Metal. In addition, it can easily leverage content produced by game engines by either embedding such content within a UI window or by acting as middleware for the integration of the game engine's advanced visual and physics features with standard UI services like input management, networking, media, and even certified safe-rendering solutions that may be otherwise lacking.

Integrating the game engine's graphics within the UI framework may be beneficial at various levels when graphics quality, performance, safety, and cross-platform capabilities are at stake. Thanks to its optimized resource management system, the UI framework enables porting game-like experiences within resource-constrained devices and even their certification for safety-critical applications.

Instead of rendering visual content constantly at the highest frame rate—which is the standard for standalone gaming on PCs—the UI engine's event-driven architecture minimizes the CPU, GPU, and memory needed for each process. In other words, where the game engine demands unconstrained resource usage for high-quality rendering and smooth gameplay of a single application, the UI engine's on-demand behavior ensures that resources are used sparingly and dynamically allocated across multiple processes based on real-time needs. While keeping the number of UI elements being loaded, drawn, and held in memory minimal, the platform-agnostic nature of the UI engine also enables to scale the same user experience across hardware types, with all the benefits that have been discussed. Among these, not least, delivering compelling visuals also on low-end hardware as a cost-effective solution for high-quality software.


Machinery malfunction SP0-TL15
Equipment is operating within parameters, and project is on schedule and on budget.
Assembly line 01
Additional details
Robotic Arm 02

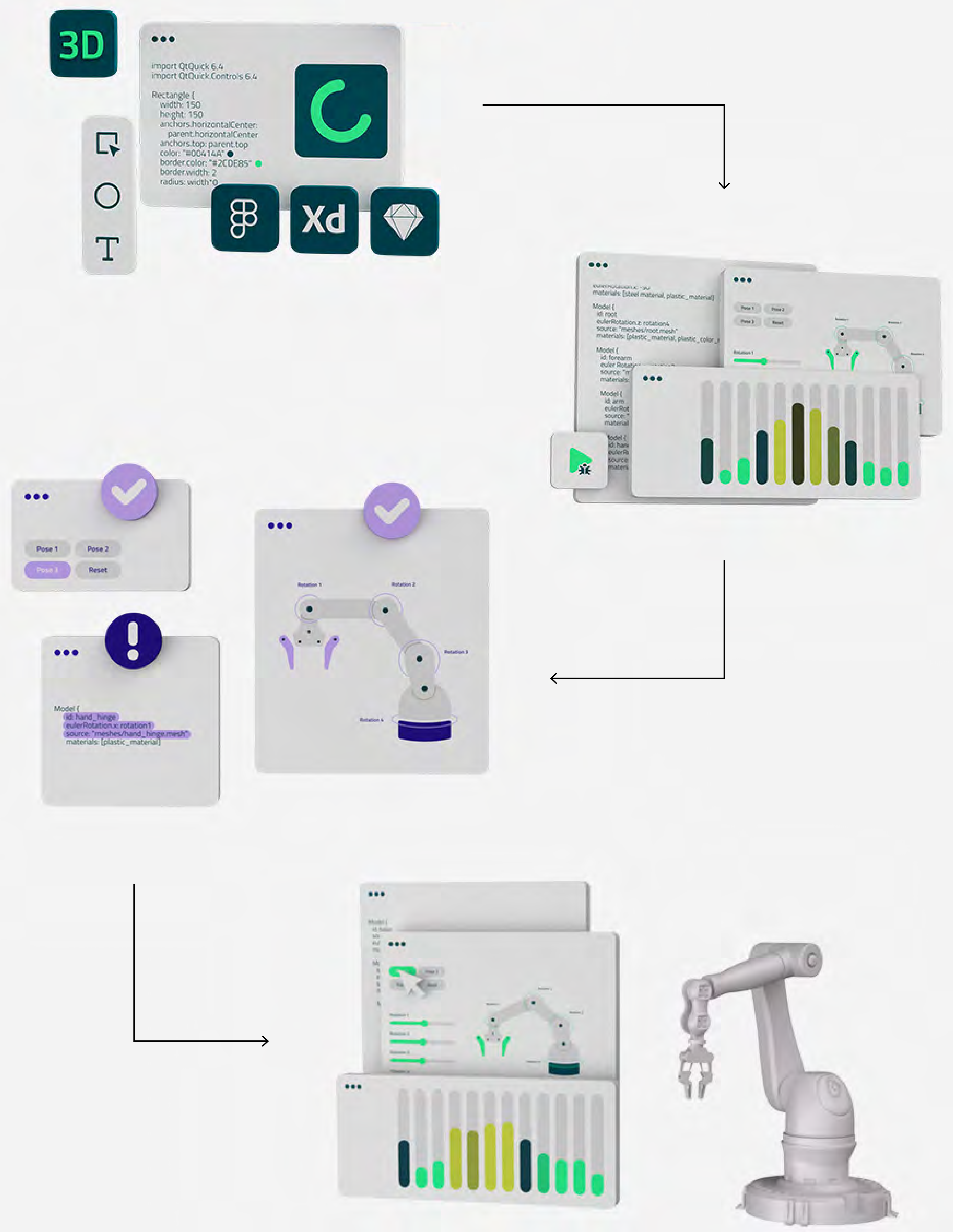
4. Workflow

There can be large variations in the result and how efficiently this is achieved depending on how the various software *tools interact* and *enable teams* to collaborate iteratively.

In the highly competitive modern market, a fragmented workflow, where tools don't naturally interface, and teams are confined in silos focusing each on one piece of code at a time is bound to fail due to its inefficiency.

When the production process spans from UI/UX design to deployment on a range of targets, through various iterations of testing and updates, the success of the enterprise—in terms of product quality, time to market, and cost—can be significantly enhanced by adopting tools that promote a **cohesive working methodology.**

User Interface Design: Functionality, Tooling, and Workflow



```
import QtQuick 6.4
import QtQuick.Controls 6.4

Rectangle {
   width: 150
   height: 150
   anchors.horizontalCenter:
      parent.horizontalCenter
   anchors.top: parent.top
   color: "#00414A"
   border.color: "#2CDE85"
   border.width: 2
   radius: width*0
```

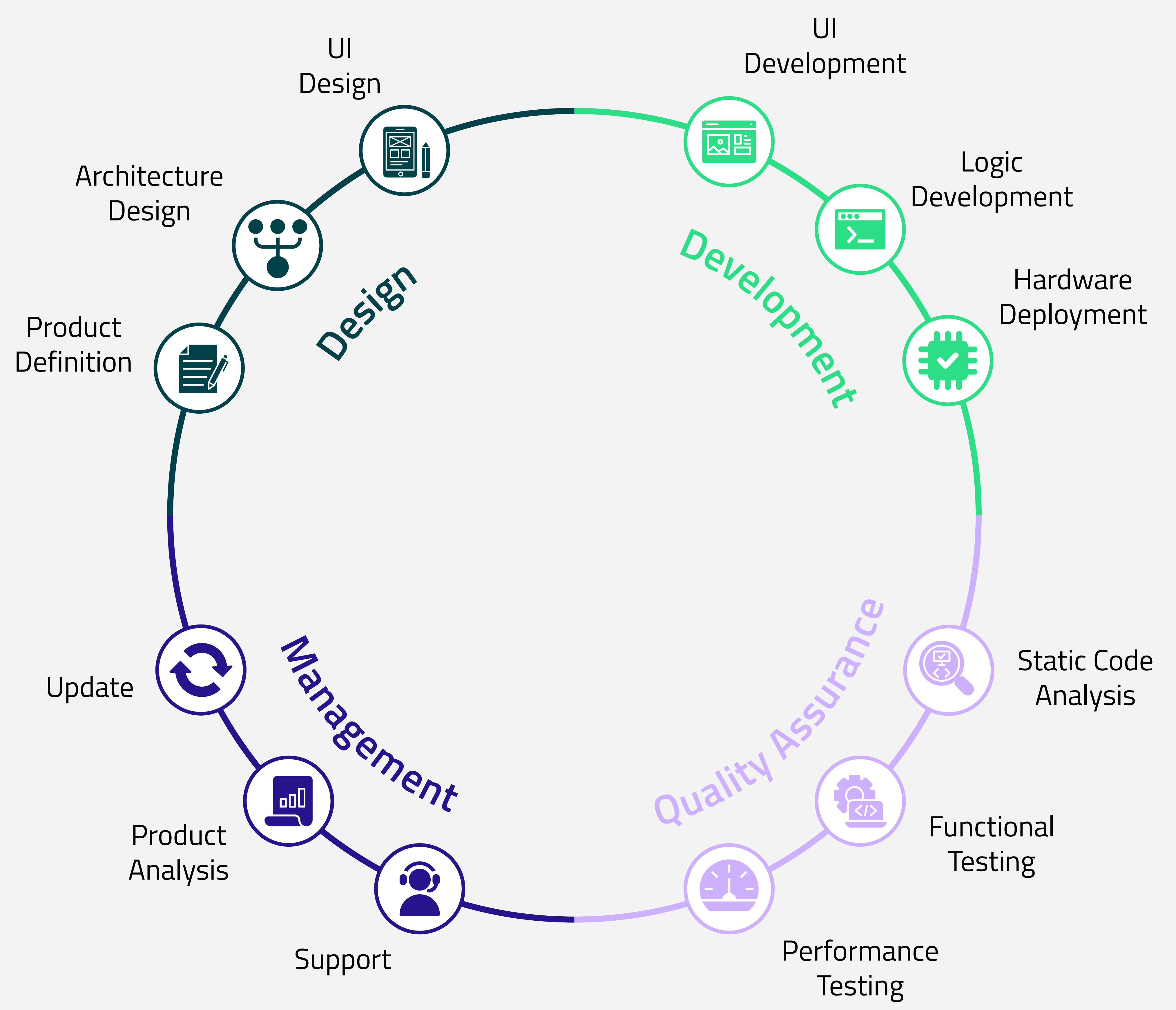
```
Model {
   id: hand_hinge
   eulerRotation.x: rotation1
   source: "meshes/hand_hinge.mesh"
   materials: [plastic_material]
```

An End-To-End Solution

The UI software creation process involves multiple stages, encompassing UI/UX design, hardware deployment, and iterative development and testing. Various teams with very different skills and competencies take part in the endeavor, and their interaction is not always easy—let alone efficient.

Without adequate tools, the design and experience ported to the app by the engineers may differ from the one originally devised by the designers. In turn, software testing should be executed at every iteration, at every update, and without **integrated testing automation tools,** manual execution is time-consuming, error-prone, and limited in scope and accuracy.

Rather than relying on fragmented tools to perform one task at a time on a need-by-need basis, the UI framework provides dedicated solutions for each stage of the software development process that build upon a **single codebase.** While providing a cohesive and consistent environment, in terms of workflow this *breaks down silos*, as UI/UX designers, 2D/3D technical artists, software architects, developers, and testing engineers can collaborate on the same repository through the UI framework's unified tooling. This enables OEMs to establish an effective *software-first* approach in which the UX is created in close connection with the software and hardware specifications.

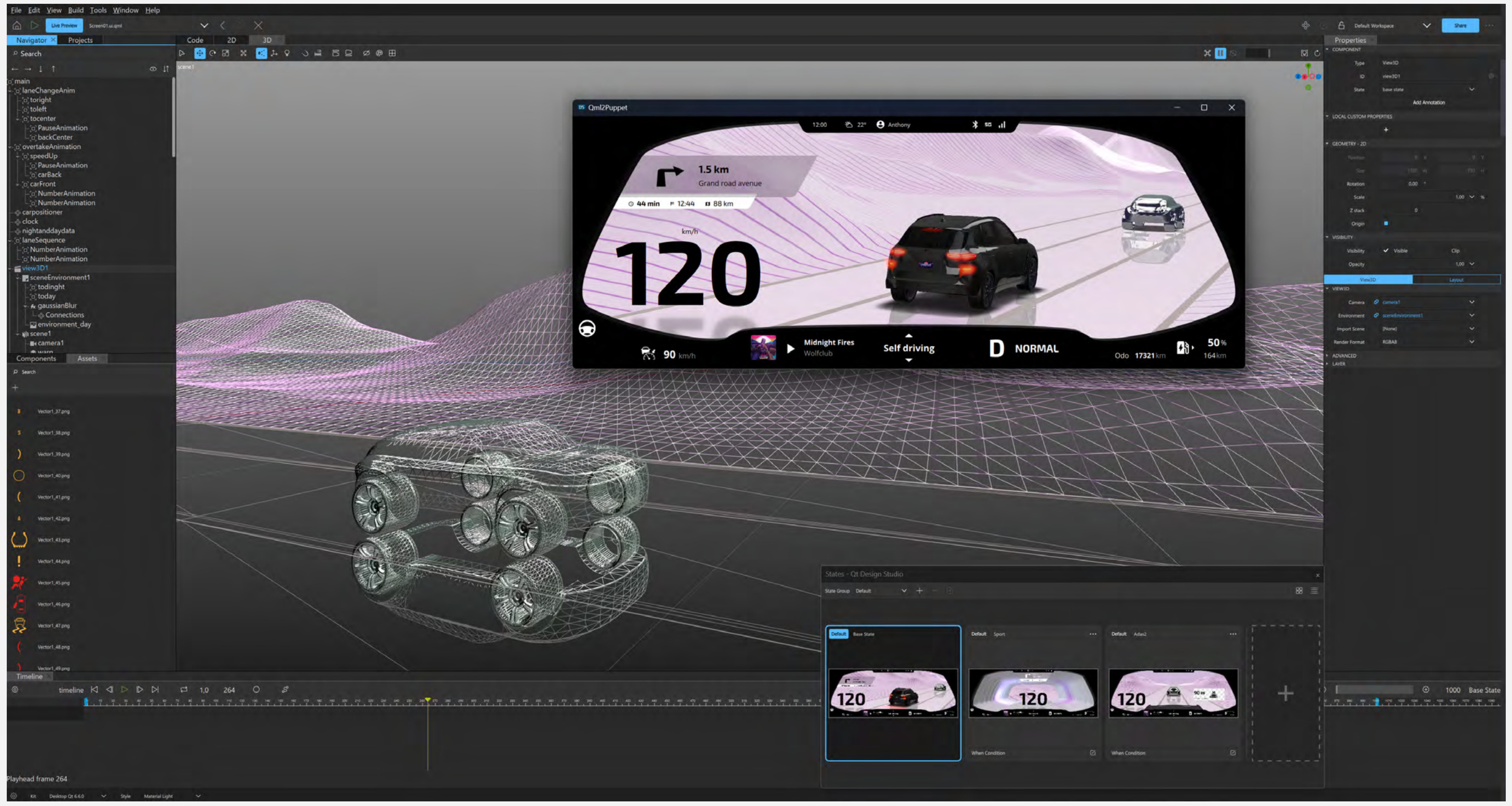


Design and Development

By sharing a common UI creation tool where the visual component of the app is built side by side with its code, technical artists and engineers can better understand each other's aims and constraints. UI designs are automatically converted into code, and code changes are immediately reflected on the UI's appearance and behavior—enabling *fast error detection* and *rapid iterations*, while leaving no margin for misunderstanding.

The possibility to *share over the web* not only the design concept but the fully working UI application allows all stakeholders to review the appearance, test the functionality, and share feedback on the overall user experience.

The UI framework's end-to-end tooling enables an integrated working methodology that drastically reduces complexity and efforts, with huge benefits on time to market and cost.



Automated Testing

When, in addition to design and development, the UI framework's tooling includes an *automated testing suite,* the highest quality standards, up to certification for safety-critical use cases, can be attained with smaller effort and cost.

Test automation increases **coverage** and **accuracy** while decreasing execution time and the risk of manual error. As frequent testing is standard in the current agile software development discipline, the possibility of scheduling extensive parallel tests on multiple platforms quickly pays off the effort of creating automated test scripts.

Standardization and Customization

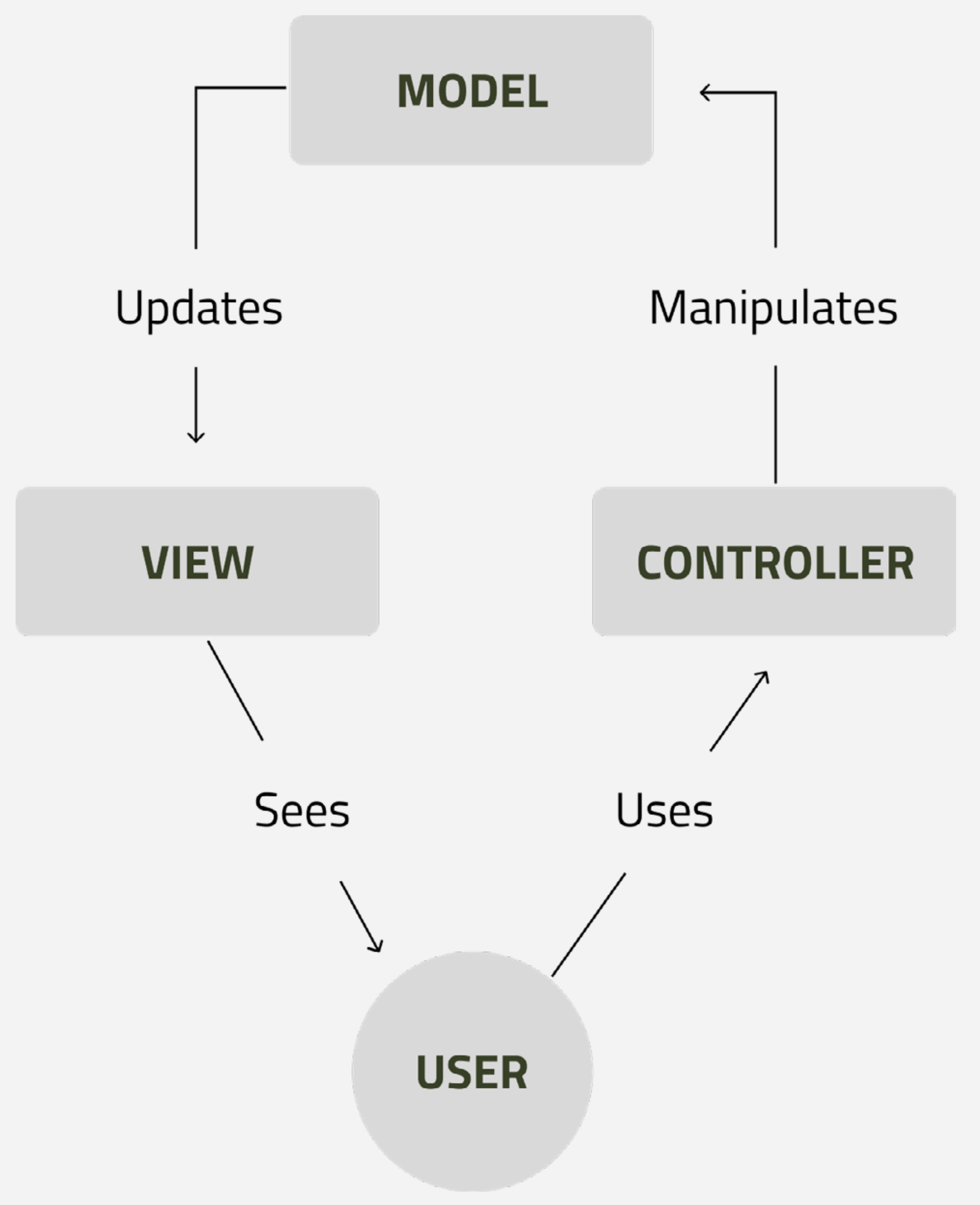
One key factor, when the product offering encompasses various devices, models, and markets is that of **standardization**: creating software for new models cannot be a re-creation from scratch and defining standardized reusable components is essential to port core functionalities across the full product portfolio.

The other key factor for multi-product companies is **differentiation,** that concerns the variation of functionality and look-and-feel across models, markets, regions, and demographic groups.

For a successful end-to-end development at scale, it is essential to harmonize two rather opposite requirements: that of *standardization* or *reusability* on the one side—namely, avoid doing the same thing twice—and *differentiation* on the other—that is, easily customize the appearance and functionality to fit different models, regions, locales, etc.

Such balance can be easily reached when the UI framework components are created according to the **model-view-controller** design pattern, namely by decoupling the appearance of the UI from the underlying logic. While the **logic** defining the applications' behavior and functionality remains, to a large extent, invariant, the **appearance** of the UI elements can be easily customized based on the specific requirements. For instance, the connectivity and intercommunication functions can be defined once and for all across devices. The layout and appearance of buttons and control elements can be easily varied across models, new themes can be added at any point, and so forth.

A UI framework offering standardized building blocks for all core components of the UI application and easy integration with the pre-existing software stack enhances compatibility and interoperability.

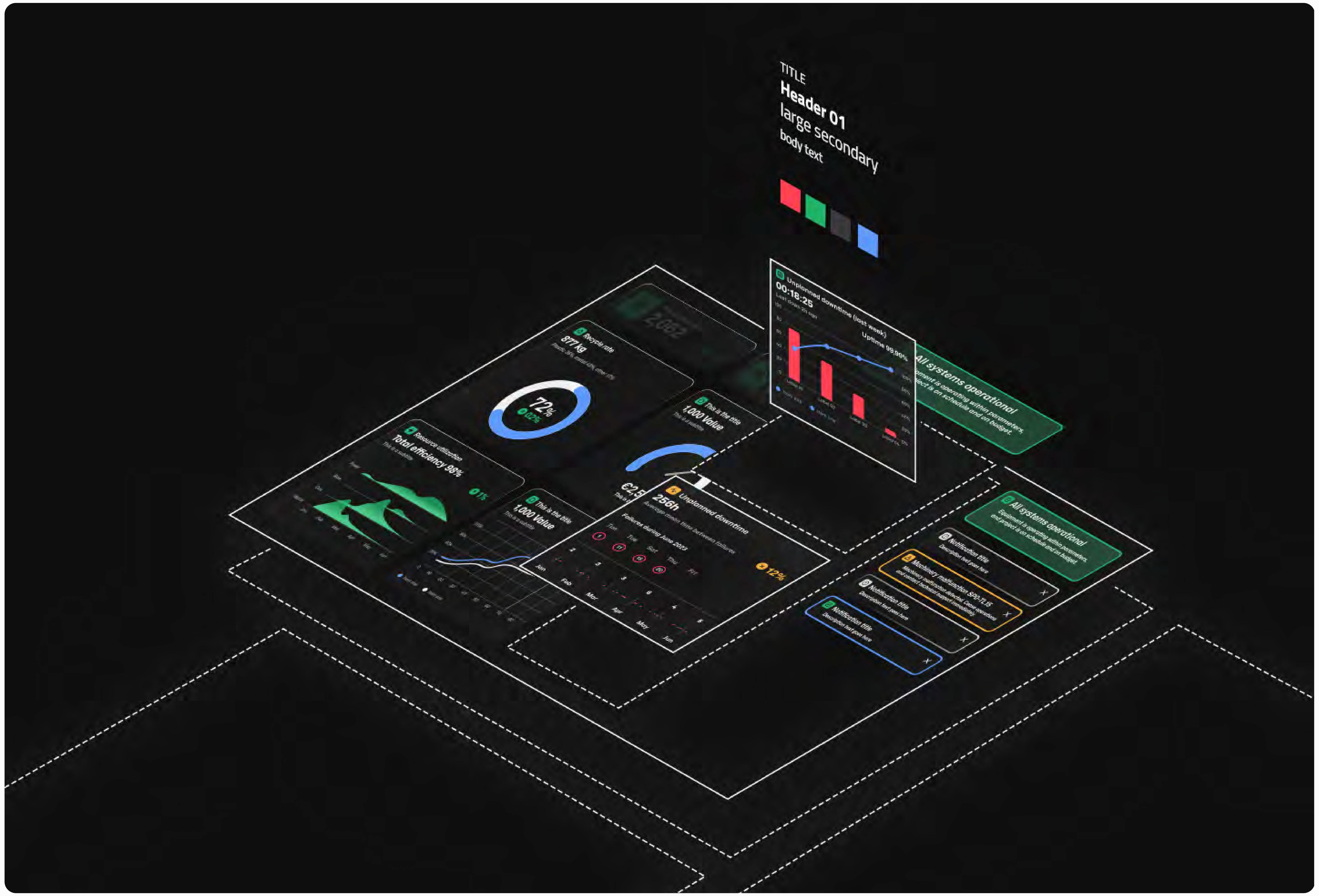


Platform Engineering

On a more general level, the UI framework enables the creation of **atomic UI primitives** that abstract over their context of use and can operate across different logic workflows (and, of course, platforms). Being each atom defined by a specific portion of code, it is simple to compose them into larger *molecules*, or **templates**, defining more complex UI elements and functions that can be applied to various use cases and products to provide *consistent functionality*. At the same time, the UI framework includes the tools to easily customize and differentiate the look-and-feel of the components according to the **design system** and UX requirements of the different models.

A UI framework enabling the creation and easy customization of such templates allows hardcore software developers and technical artists/front-end developers to work side-by-side on the same codebase, with major benefits in quality, efficiency, and costs. But there is more.

Building on the idea of standardization and re-usability, platform engineering aims at providing developers with a self-service infrastructure of highly optimized ready-made components that can be easily fetched, customized, and deployed to different devices. As a way to reduce the developers' cognitive load (increased substantially with the increase in software complex-



ity), such production strategy has recently been popularized for its benefits on *productivity, efficiency, quality* and overall **developer satisfaction.** In this setup, while *senior developers* maintain the self-service platform, ensuring the availability of robust functionality and compatibility with other components, it's easier even for less experienced developers to build and ship high quality products with greater efficiency.

In the platform engineering setting, the UI framework acts not only as the source of UI content and functionality to build a UI app, but more substantially as **middleware** providing the level of abstraction and standardization needed to integrate different systems together, for different modules to talk to each other, for different software stacks to live on the same codebase.

Ownership

As user experience in digital devices is aimed toward a smartphone-like experience, also the technological trend is shifting toward smartphone-like solutions: on the *hardware side*, technology is migrating toward a *single board* with multiple processors, while on the *software side*, toward a single OS covering all use cases. This applies across industries, from consumer electronics and health care to the automotive sector, as part of the **software-first** strategy being adopted by most OEMs.

This trend is motivated by the opportunity it offers for OEMs to reduce their bill of materials, the amount of code to be maintained, and the dependency on third-party suppliers. The possibility to use standardized software components and the availability of tools for their easy customization in a single UI framework is a key enabler for succeeding in the modern, competitive market.

The key takeaway from the trend toward a **platform strategy** is that by increasing ownership, OEMs can simplify their process, reduce costs, and reinforce their brand. The adoption of a unified UI framework whose tools encompass the end-to-end product lifecycle enables OEMs' emancipation from third-party suppliers, with major potential benefits in terms of optimized productivity, efficiency in delivery, brand differentiation, and quality.

Qt Group

While our description of the UI framework's features and capabilities abstracts away from specific market products to focus on the must-have of the end-to-end tooling for an effective software development process, a quick glance at Qt offering will convince our readers that such tooling is indeed available and can serve you as a reliable guide in your software development journey.

Development

Qt Design Studio
Turning Visions into Functional Applications

Qt Framework
Comprehensive Libraries for Industry-grade Software

Qt Creator
Coding, Building, Testing, and Deploying on one IDE

Qt Insight
Real-time Usage Intelligence

Quality Assurance

Squish
GUI Test Automation

Coco
Code Coverage Analysis

Test Center
Test Results Management and Analysis

Axivion
Static Code Analysis and Architecture Verification

Qt Group

qt.io