Nucleus: A Design System for Animation and VFX Applications

Jon-Patrick Collins
jonc@al.com.au
Animal Logic
Sydney, NSW, Australia

Anno Schachner
annos@al.com.au
Animal Logic
Sydney, NSW, Australia

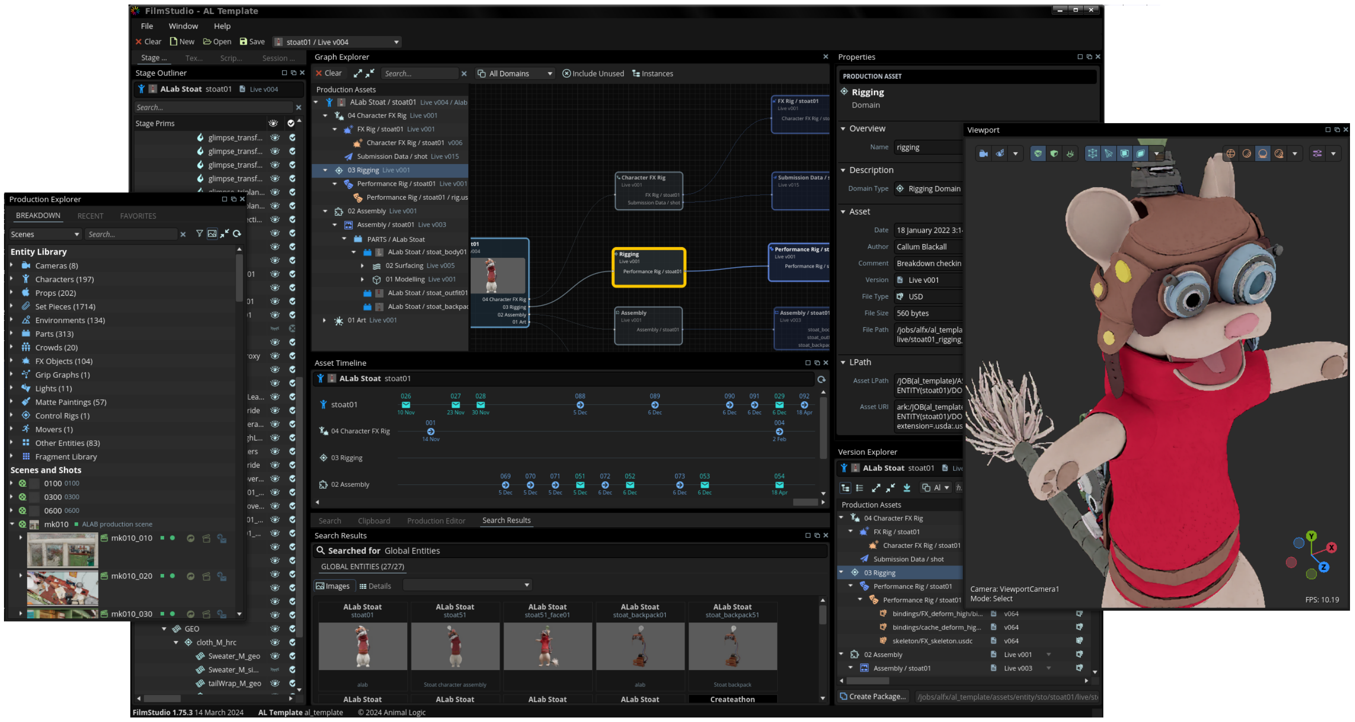


Figure 1: FilmStudio, an application for authoring USD-based assets, developed using the Nucleus Design System.

ABSTRACT

We describe a plugin-based architecture for developing component-based Qt applications for animation and visual effects, and discuss the benefits this approach offers in terms of code reuse, stability and consistency. We introduce an Application Maturity Model quality metric to characterize a set of best practices, design patterns and frameworks for developing complex interactive applications.

CCS CONCEPTS

• **Computing methodologies** → **Graphics systems and interfaces**.

KEYWORDS

Application, PyQt, Python, Qt, UI, UX, VFX, Workflow

ACM Reference Format:
Jon-Patrick Collins and Anno Schachner. 2024. Nucleus: A Design System for Animation and VFX Applications. In *Proceedings of The Digital Production Symposium (DigiPro '24)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Developing complex interactive software applications is non-trivial. In a typical animation or VFX studio context, this is even more so, due to factors that include heterogeneous craft groups working in a decentralized manner, tight production pressures with limited opportunities for developing polished products, and possibly fewer software architects available for such development.

Over the past two decades, the animation and VFX community has witnessed the emergence of Qt as the cross-platform user-interface application framework of choice, superseding the use of other frameworks such as wxWidgets, GTK and the older, proprietary UI toolkits shipped with common digital content creation (DCC) applications.

The emergence of Qt as an industry standard has provided an opportunity to move beyond the earlier fragmentation of UI tooling, allowing animation and VFX studios to concentrate efforts by targeting a single framework. Nevertheless, application development with Qt has its own challenges, requiring a degree of mastery of design patterns including model-view-controller (MVC), event broadcaster-subscriber and application orchestration. The Qt ecosystem includes over 2000 classes and is potentially daunting to developers without significant UI development experience, leading to poor choices of design patterns and limited use of the full Qt feature set.

Moreover, Qt applications developed by individual, autonomous craft groups (at least in our experience) typically follow **Conway's Law** [7]: collections of highly customized, tailor-made solutions that are difficult to reuse, leading to duplication of effort, a lack of design cohesion and reduced opportunities for collaboration.

At Animal Logic, we have experienced these challenges first-hand, and responded with the development of a range of best practices, design patterns and frameworks to successfully leverage the best of Qt application development in a fast-paced, distributed, multi-studio context. Ultimately, this journey led to the development of the Nucleus Design System, a framework widely used at Animal Logic, and the basis for many of our proprietary in-house USD-based systems, including **Forge**™ [2], our Maya-based animation system; **Filament**™ [1], our standalone lighting system; **FilmStudio**™ [4], our standalone USD asset authoring system; and **Plasma**™ [5], our Nuke-based compositing system.

In this paper we describe the Nucleus Design System in the context of an **Application Maturity Model** and key design decisions en route. This Maturity Model concept takes its inspiration from the Richardson Maturity Model [6] used to characterize Web services.

2 APPLICATION MATURITY MODEL

2.1 Level 0: Baseline Qt deployment

Early UI development at Animal Logic was largely ad hoc, piecemeal and disorganized. User interfaces, where they existed at all, were developed using either the host DCC toolkit, or wxWidgets. In 2010 we transitioned from Windows to Linux, and took this opportunity to migrate our (relatively small) UI codebase to Qt.

Prefer PyQt where possible. Given our wide use of Python in-house, we primarily adopted the Python PyQt bindings. This allowed the widest-possible audience of developers to contribute to our Qt-based applications. We also made selective use of the C++ Qt libraries, but mostly in contexts that already made heavy use of C++ or where performance was critical. We preferred PyQt over the alternative PySide for its more active ecosystem and product support (at the time). We also preferred PyQt to the alternative QML markup system for its more complete widget library, notably for data-intensive widgets such as tree, list and table widgets, where there were no full-featured QtQuick equivalents.

Prefer user-defined code to Qt Designer. We avoided the use of the Qt Designer tool for building user-interfaces, finding that the marginal short-term time gain in using rapid application development (RAD) tools during the visual design process was outweighed by the inability to maintain and refactor the resulting `.ui` files. Qt Designer also hindered our deployment of custom Qt widgets in place of default Qt widgets.

Following the deployment of Qt throughout a codebase, we can consider it to be at **Application Maturity Level 0**. We achieved this threshold around 2010.

2.2 Level 1: Unified visual design

With our applications migrated to Qt, an inevitable next step was to start leveraging this consistent baseline in various ways. In particular, we elected to ensure consistency in the look-and-feel across our various Qt-based tools.

Introduce UI-specific software package. At Animal Logic we use the Rez packaging system for software management. Early on, we introduced a new Rez package to house UI-related functionality. This was an important first step in making it easy for disparate teams to opt in to our shared UI library.

Define a shared Qt stylesheet. A powerful Qt mechanism is its stylesheet system, in which a QSS stylesheet file is used to define the look-and-feel of UI controls, in a similar manner to the CSS system in Web design. Over time, we fine-tuned a customized in-house

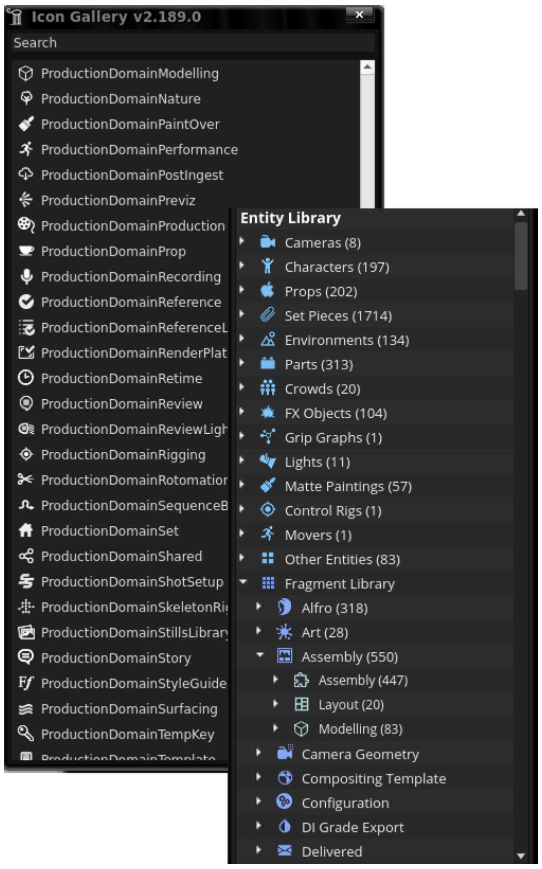


Figure 2: Icon Gallery browser highlighting a sample of the shared library of 500+ icons available. Icons may be programmatically colorized, highlighted and disabled, as illustrated with this ProductionExplorer widget example.

design, largely based on a need to move from the default dark-on-light Qt theme to a light-on-dark theme more suited to animation and VFX applications. Once complete, we encouraged the use of this stylesheet through all Qt-based tools to build up a recognizable Animal Logic brand by shipping this with the new UI package. This was a minimal-cost choice that nevertheless had a large benefit: the emergence of a visually cohesive software collection.

Automate QSS generation. QSS files make extensive use of hard-coded constants defining colors, fonts and many other attributes. We quickly determined that moving these constants to their own Python module and auto-generating QSS stylesheets from a baseline template reduced duplication, avoided inconsistencies and moreover allowed us to generate a wide family QSS variants. This allowed us to preserve a consistent in-house brand but still make concessions for certain contexts such as the specific design of a given host DCC application, or for different display resolutions.

Introduce common icon library. A key goal was to create a *strong visual design language* with a high degree of design consistency, where common studio-wide concepts such as "shot" or "character" have consistent visual representations across different UIs. Initially, our applications either shipped their own icon resource files (with typically inconsistent design), or simply did not use icons at all. We introduced a library of 16x16 PNG and SVG artworks with an associated Python API to allow these to be accessed as QImage, QPixmap or QIcon objects at runtime, with support for caching and various kinds of transformation (resizing, grayscale, colorization). We primarily adopted an iconographic design style in order to allow contributions from many developers, and defined these mostly as white-on-transparent PNG files, allowing for runtime colorization.

Following the visual design unification throughout a Qt codebase, we can consider it to be at **Application Maturity Level 1**. We achieved this threshold around 2012.

2.3 Level 2: Unified QWidget ecosystem

With the widespread adoption of the UI package throughout our codebase, it emerged as a natural home for custom widgets. Over time we added hundreds of custom widgets to this library. Furthermore, by utilizing the **Facade design pattern** [8], it became the primary conduit for access to *all* Qt classes.

Curate a shared library of custom widgets. Initially, our Qt applications that required custom widgets would define these directly within their own codebases. In some cases, these might be subclasses with additional or overridden methods, signals or slots; at other times these might be composite widgets comprising many individual widgets. In both cases, we made a concerted effort to migrate these into the UI package in order that they emerged as reusable widgets, with an easy instantiation pattern such as `myObj = ui.AdvancedSearchBar()`. By making these QWidget classes easy to find, easy to instantiate and easy to develop, our shared widget library built development momentum. We encouraged developers to contribute to this library, even in cases where their application may be the only consumer of a specific custom widget.

Unify built-in and custom widgets. With the success and widespread rollout of our UI package, we made the leap to import *all* built-in Qt classes via the same central module and remap these

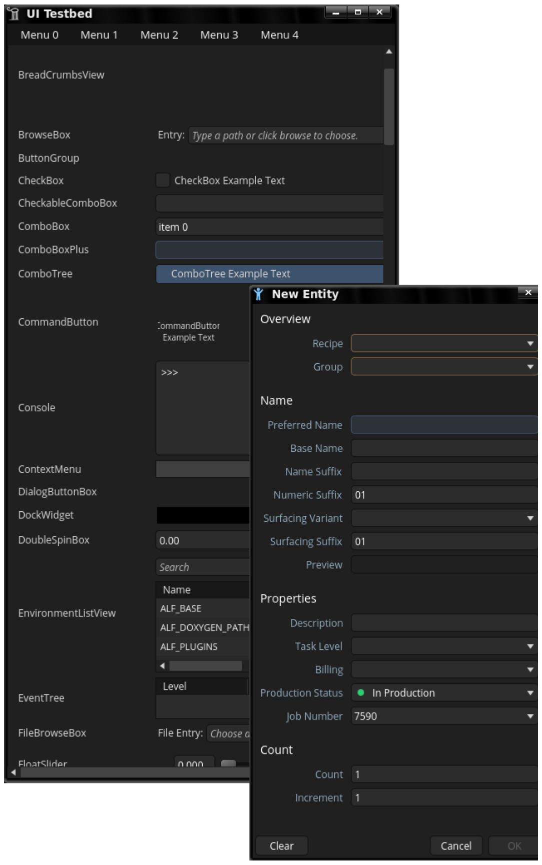


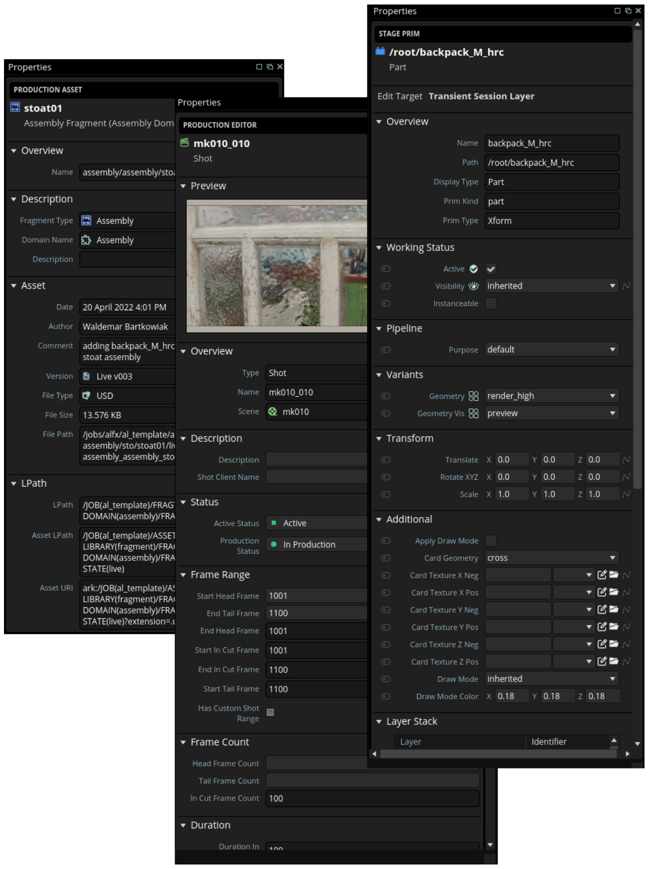
Figure 3: UI Testbed browser highlighting a sample of the shared library of 100+ widgets available, with sample usage illustrated with the New Entity dialog. Both user-interfaces utilize the shared QSS stylesheet, colors and fonts.

using our in-house convention of adopting the Qt name without the Q- prefix. Thus, `myObj = ui.ComboBox()` would be the preferred pattern to create our flavor of QComboBox. With this layer of indirection we were able to gradually introduce and modify subclasses of standard widgets at any point without disrupting client code (after the initial code conversion). Importantly, it erased the distinction between "standard Qt" widgets and "custom Qt" widgets by making the creation patterns identical. Furthermore, it allowed us to flatten out potentially deeply-nested Python package paths into a simple calling convention, with all widgets directly accessible via our UI facade module.

Following the deployment of a unified QWidget ecosystem and visual design language throughout a Qt codebase, we can consider it to be at **Application Maturity Level 2**. We achieved this threshold around 2014.

2.4 Level 3: Dependency inversion pattern

Widgets can be characterized as *data-light* or *data-heavy*: lightweight widgets such as QComboBox may be readily populated at creation time, whereas heavyweight widgets such as QTreeWidget


Properties

PRODUCTION ASSET

stoat01

Assembly Fragment (Assembly Dom

Overview

Name: assembly/assembly/stoa

Description

Fragment Type: Assembly

Domain Name: Assembly

Description:

Asset

Date: 20 April 2022 4:01 PM

Author: Waldemar Bartkowiak

Comment: adding backpack_M_hrc stoat assembly

Version: Live v003

File Type: USD

File Size: 13.576 KB

File Path: /jobs/alfx/al_template/a assembly/sto/stoat01/liv assembly_assembly_sto

LPath

LPath: /JOB(al_template)/FRAG DOMAIN(assembly)/FRA

Asset LPath: /JOB(al_template)/ASSET LIBRARY(fragment)/FRA DOMAIN(assembly)/FRA STATE(live)

Asset URI: ark:/JOB(al_template)/AS LIBRARY(fragment)/FRA DOMAIN(assembly)/FRA STATE(live)?extension=.u

Properties

PRODUCTION EDITOR

mk010_010

Shot

Preview

Overview

Type: Shot

Name: mk010_010

Scene: mk010

Description

Description:

Shot Client Name:

Status

Active Status: Active

Production Status: In Production

Frame Range

Start Head Frame: 1001

End Tail Frame: 1100

End Head Frame: 1001

Start In Cut Frame: 1001

End In Cut Frame: 1100

Start Tail Frame: 1100

Has Custom Shot Range: ☐

Frame Count

Head Frame Count:

Tail Frame Count:

In Cut Frame Count: 100

Duration

Duration In: 100

Properties

STAGE PRIM

/root/backpack_M_hrc

Part

Edit Target **Transient Session Layer**

Overview

Name: backpack_M_hrc

Path: /root/backpack_M_hrc

Display Type: Part

Prim Kind: part

Prim Type: Xform

Working Status

Active: ✓

Visibility: inherited

Instanceable: ☐

Pipeline

Purpose: default

Variants

Geometry: render_high

Geometry Vis: preview

Transform

Translate: X 0.0 Y 0.0 Z 0.0

Rotate XYZ: X 0.0 Y 0.0 Z 0.0

Scale: X 1.0 Y 1.0 Z 1.0

Additional

Apply Draw Mode: ☐

Card Geometry: cross

Card Texture X Neg:

Card Texture X Pos:

Card Texture Y Neg:

Card Texture Y Pos:

Card Texture Z Neg:

Card Texture Z Pos:

Draw Mode: inherited

Draw Mode Color: X 0.18 Y 0.18 Z 0.18

Layer Stack

Layer	Identifier

Figure 4: PropertiesView, a complex custom QWidget that supports inspection and modification of object attributes. Utilizing the data source design pattern, this widget is used in disparate contexts by injecting custom datasources as needed. Example illustrates integration with a suite of representative data sources, including USD and database objects.

have variable and potentially large datasets that can lead to significant slowdowns when attempting to pre-populate their entire datasets at creation time.

To ameliorate this, Qt provides a suite of the data-driven model-view widgets such as QTreeView and QTableView that utilize the **Dependency Inversion design pattern** [10]. This powerful pattern, colloquially thought of as "Don't call us—we'll call you", uses an intermediate data model that acts as a dynamic data accessor on a need-to-have basis. Whilst powerful, this category of widgets is exceptionally difficult to use, which in practical terms limits their real-world use. In our codebase, we found the correct use of these critically-important widgets to be vanishingly small.

Simplified programming model for model-view widgets. We identified QTreeView, QListView and QTableView as critical widgets for our in-house Qt applications, and created custom subclasses (for example `ui.TreeViewPlus` and `ui.ListViewPlus`) that ship with a powerful and simple pattern for registering row and column data sources that make it comparatively trivial for clients to utilize these widgets, with a much lowered barrier to entry, and avoiding the complexities of the older implementations using the QModelIndex class. This programming pattern has proved to be popular with our developers, with widespread adoption. In turn this has been instrumental in the development of our application ecosystem, where large hierarchical datasets are ubiquitous.

Widespread adoption of dependency inversion. We discovered that this pattern, and in particular the use of separate data source classes to manage the flow of data between specific business contexts and general-purpose QWidgets, is a very useful abstraction. We subsequently developed many other complex widgets, including comprehensive and reusable `ui.GraphView` and `ui.PropertiesView` widgets, with the data source pattern allowing these general-purpose widgets to be reused by many craft groups for a range of disparate purposes. We strongly encouraged developers building specific widgets for their needs to refactor these widgets using the data source pattern, to encourage their reuse, and to provide for a clear separation of concerns.

Following the deployment of dependency inversion, along with a unified widget library and visual design throughout a Qt codebase, we can consider it to be at **Application Maturity Level 3**. We achieved this threshold around 2016.

2.5 Level 4: Command pattern

Animation and VFX studios will typically accumulate a large body of disparate scripts, functions and systems. This functionality may be opaque to discover, non-trivial to use, and hard to easily integrate into other systems. We found that recasting such functionality using the **Command design pattern** [9] was a critical early decision that paved the way for the later Nucleus Design System.

Introduce commands. We introduced a Command base class with methods such as `doIt()` and `undoIt()`, with a unique Command ID per command. Over time, we recast all artist-facing tools and scripts by re-implementing these as commands, with their implementations moved into the `doIt()` methods. We extended the capabilities of commands to include optional typed arguments and argument presets, as well as pre-execution argument resolution. This process regularized and standardized hundreds of existing tools and scripts, which was a major boon to the codebase. Moreover, command execution became simplified; clients could now invoke commands with only knowledge of their relevant Command IDs.

Introduce discoverability and registration. We added automatic command discovery and registration to our command system, also adding build-time command manifest creation for optimized performance. Our Rez-based software packaging system was a good fit for this work, as Rez packages can extend environment variables (such as lists of system folders) such that the resolution of a given software package will automatically ensure its commands are injected into the command registry.

Introduce UI metadata and hinting. We added optional UI metadata to commands, including display names, tooltips and icons. Together with discoverability, this allowed us to create useful applications such as the **Artist Tools palette**, an auto-populated shelf of discovered commands. The Artist Tools palette has been highly successful, replacing many tediously maintained, inconsistent tool shelves. We also added optional `isVisible()` and `isEnabled()`

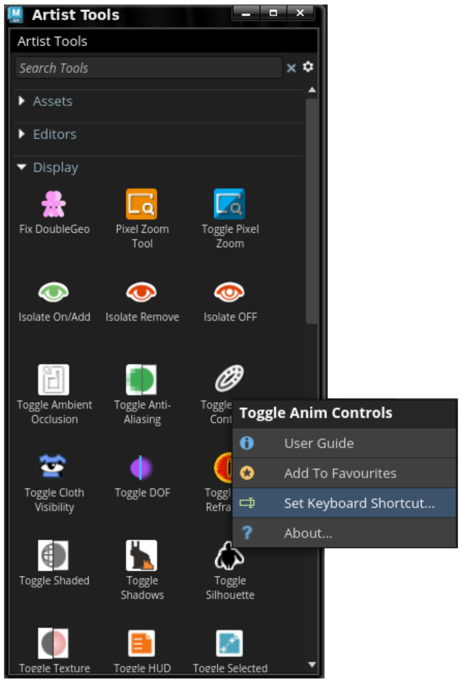


Figure 5: Artist Tools palette, an auto-populated collection based on the automatic discovery, registration and UI generation of command plugins discovered in the currently-resolved context. Note that the context menu actions illustrated are also implemented via the same command system.

command methods in order to allow commands to autonomously determine whether they are available and usable in a given context.

Replace QActions with Command IDs. Whilst the command pattern was initially motivated by seeking to develop an autonomous Artist Tools palette, it quickly became apparent that commands were exceptionally useful and clean ways to organize code. A key insight was that we could utilize commands as the bedrock for *all* our Qt applications: in place of hard-coded toolbars, menu bars and context menus with their associated static QActions, clients could instead nominate lists of Command IDs. At run-time we would find these commands in the command registry, and use their UI hints to determine settings for dynamically-created QActions.

The Command pattern creates an **elegant separation of concerns** between developers building applications and developers building their associated commands. As self-describing units of functionality, commands can be readily shared and reused across disparate applications. Calling code needs no explicit knowledge of module import paths, since commands are invoked simply via their Command IDs. Another useful side-effect is that if at runtime a given command is unavailable, the associated QAction is simply not created, making applications able to run in a wider range of environments with elegant degradation.

Following the deployment of commands, dependency inversion, a unified widget library and visual design throughout a Qt codebase, we can consider it to be at **Application Maturity Level 4**. We achieved this threshold around 2018.

3 NUCLEUS DESIGN SYSTEM

Despite a successful implementation of Maturity Level 4, **recurring issues emerged** due to the inevitable complexity arising from application size. In particular, building large interactive applications required an inordinate amount of duplicated logic for managing the creation and runtime state of user-interface elements; orchestrating complex signal-slot chains was error-prone and difficult to reason about; and the shared widget ecosystem itself did not prevent significant code duplication and splintering, since more business-oriented, higher-order controls would be regularly reinvented.

3.1 Level 5: Component plugin architecture

In 2018 we embarked on the next major phase of the Maturity Model with the introduction of the **Nucleus Design System**. In a nutshell, the Nucleus Design System reimagines applications using a **plugin-based architecture** with application-level configuration replacing application-level code.

Generalized MVC plugin system. Our key insight was to ask: *What makes one application distinct from another?* Our answer was: a specific collection of models, views and commands, and the way these are orchestrated. From this starting point, we determined that we could generalize our plugin-based command system and extend this design pattern to support the *full range of components* necessary for an entire model-view-controller (MVC) framework: Models, Views and Commands, along with additional plugins such as ToolbarControls and MenuBuilders. All such Nucleus plugins are characterized by a unique Nucleus ID, such as `Nucleus.Model.Maya.Selection` or `Nucleus.View.Properties`.

Single configurable application. We determined that in place of repeatedly developing separate Qt applications (with their inevitable code duplication, inconsistency and redundant effort) we could instead implement a *single canonical Qt application* (the "core Nucleus application") and handle distinct target application requirements by injecting different *application configuration* objects into the core application at startup. Application configurations are a set of mostly static key-value pairs that (amongst other things) nominate Model IDs and View IDs of interest. The core Nucleus application then takes responsibility for the correct discovery, instantiation and organization of these plugins. Other configuration settings similarly define toolbar controls, menus, splashscreens, window icons and related application infrastructure.

Centralized event system. Nucleus applications implement an *event system* that encapsulates and generalizes the lower-level Qt signal-slot system. Models and views can choose to *elevate* a QSignal by re-broadcasting it as a Nucleus event characterized by a unique Event ID with optional arguments. Nucleus application configurations then manage application control flow by nominating lists of Command IDs to be executed in response to specific Event IDs. For example, a data model could emit an event when its internal data is reloaded; this would trigger a sequence of commands that would reload application views that have some interest in the updated data. This essentially allows generic and modular, but unrelated, components to react to each other without having any knowledge of each other.

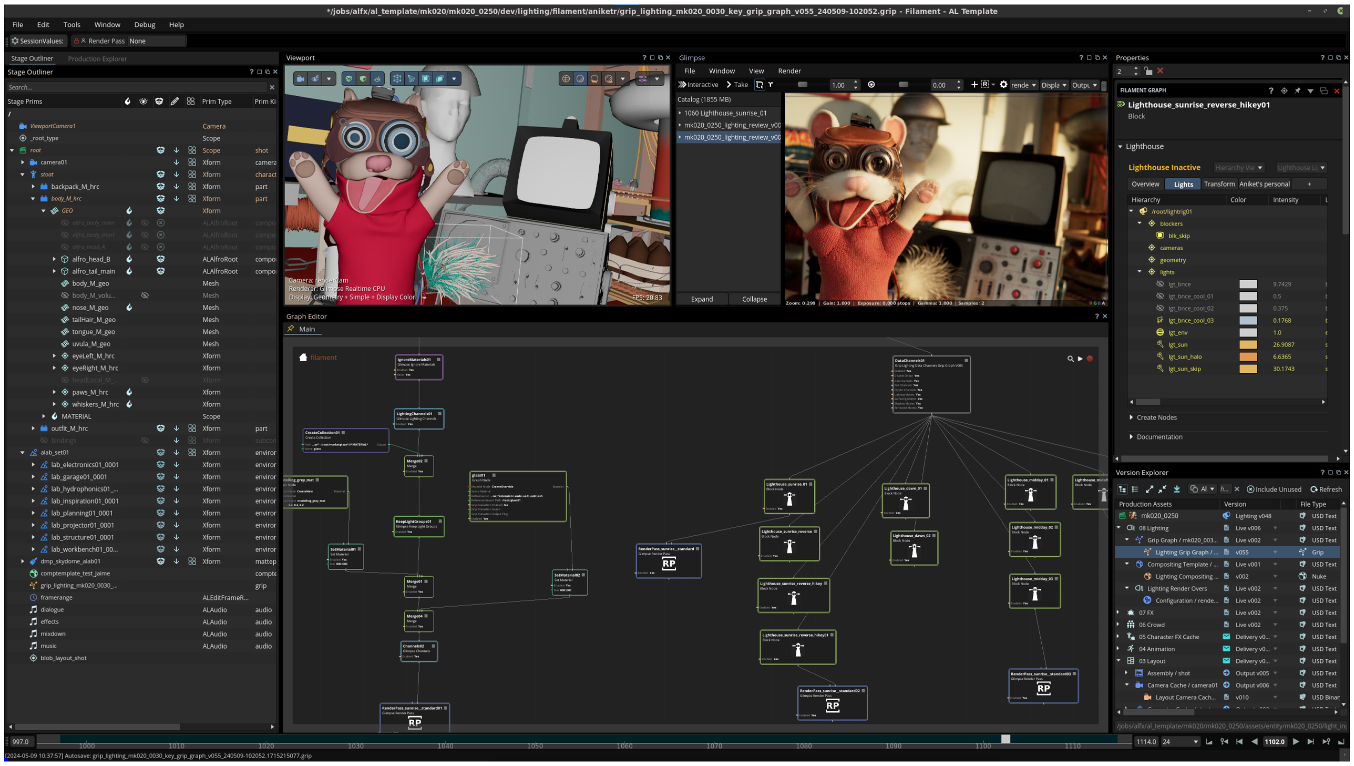

*/jobs/alfx/al_template/mk020/mk020_0250/dev/lighting/filament/aniketr/grip_lighting_mk020_0030_key_grip_graph_v055_240509-102052.grip - Filament - AL Template
File
Edit
Tools
Window
Debug
Help
Stage Outliner
Viewport
Glimpse
Properties
Graph Editor
Version Explorer
FILAMENT GRAPH
Lighthouse_sunrise_reverse_hikey01
Lighthouse
Create Nodes
Documentation

Figure 6: Filament, our flagship proprietary lighting system, fully developed using the Nucleus Design System. This complex application is built up from 17 Nucleus models and 25 Nucleus views, and observes 112 Nucleus events. The application configuration defining interactions between the models, views, and events, including keyboard shortcuts and context menus, is a single Python configuration module of just 1890 lines.

Following the deployment of a configuration-based MVC plugin architecture to a Qt codebase that already uses commands, dependency inversion, shared widgets and shared visual design, we can consider it to be at **Application Maturity Level 5**. We achieved this threshold around 2020.

3.2 Inspiration, results and discussion

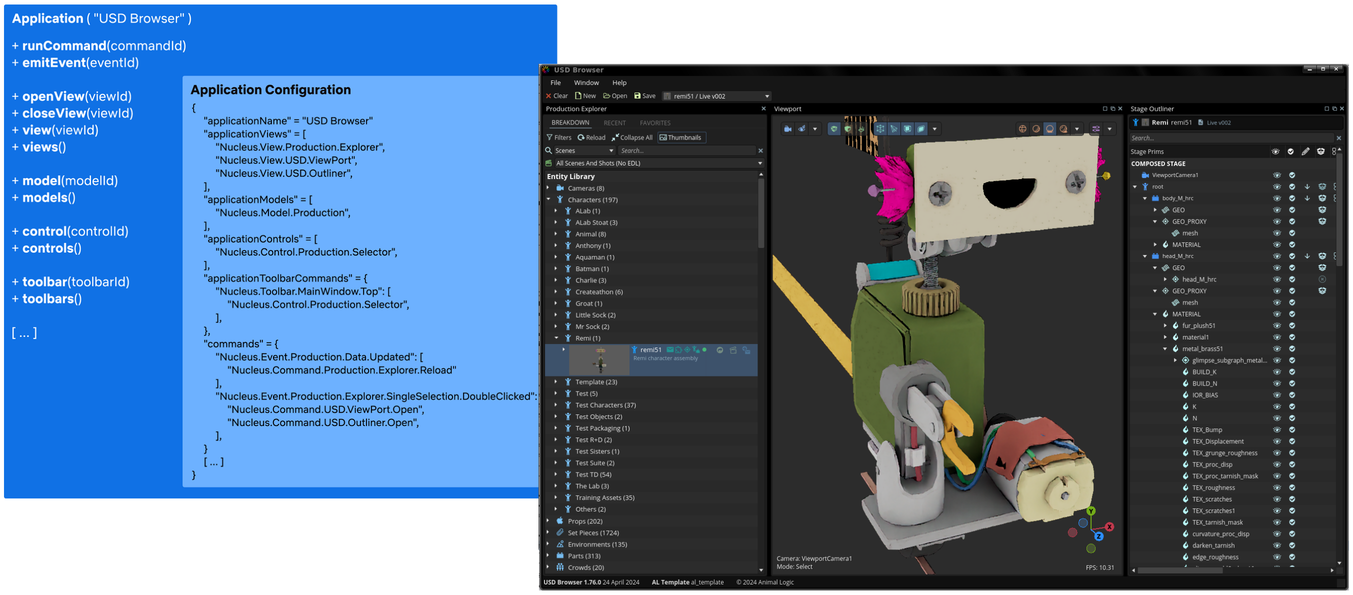
The Nucleus Design System was primarily motivated by the need to be able to design and develop useful functional components **independently of any applications** in which they might ultimately be used, and by the need for applications to be able to readily specify the components they seek to use with essentially zero effort. A key inspiration was the **Eclipse Integrated Development Environment** [3], which uses a similar architecture whereby all functionality is expressed via plugins. To our knowledge, Nucleus represents the first use of this pattern for Qt application creation.

Nucleus has proven to be a successful design pattern, and has allowed a small development team to build and maintain a large collection of complex, interactive applications. Indeed, benefits have accrued that were not immediately apparent when the design was initially considered.

Rich component ecosystem. A component-based approach to application design radically improves reusability, as the very act of creating a Nucleus component (such as a specific view) for one craft group makes it effortlessly reusable by other groups. Over time, this has created a rich ecosystem of reusable components. As one example, we now ship various views (such as the Script Editor view and Profiling view) as standard for all Nucleus applications.

Stability and consistency. With a single canonical Nucleus application, we are able to consolidate and enhance many otherwise disparate approaches to a wide range of Qt application implementation details. To take view management as one example, we can now handle layout persistence, user-defined named layouts and view menu synchronization in a single centralized Nucleus core codebase. Whereas previously, each application team was responsible for these measures on a per-application basis, in the new Nucleus-based paradigm a single team becomes responsible for the core application and its associated UI management. This has proven to be a major productivity boost and also led to significantly improved stability and consistency amongst individual products.

Event orchestration. The Nucleus event system is integral in tying together separate Nucleus components into a coherent application. It acts as a single point of contact between event broadcasting and handling. With all key events flowing through a single function in the configuration, development, testing and debugging is simplified. For our flagship applications, this centralized approach has been especially helpful when tracing control flow issues and understanding complex interdependencies between components.



Application ("USD Browser")

+ **runCommand**(commandId)
+ **emitEvent**(eventId)

+ **openView**(viewId)
+ **closeView**(viewId)
+ **view**(viewId)
+ **views**()

+ **model**(modelId)
+ **models**()

+ **control**(controlId)
+ **controls**()

+ **toolbar**(toolbarId)
+ **toolbars**()

[...]

Application Configuration

```
{
    "applicationName" = "USD Browser"
    "applicationViews" = [
        "Nucleus.View.Production.Explorer",
        "Nucleus.View.USD.ViewPort",
        "Nucleus.View.USD.Outliner",
    ],
    "applicationModels" = [
        "Nucleus.Model.Production",
    ],
    "applicationControls" = [
        "Nucleus.Control.Production.Selector",
    ],
    "applicationToolbarCommands" = {
        "Nucleus.Toolbar.MainWindow.Top": [
            "Nucleus.Control.Production.Selector",
        ],
    },
    "commands" = {
        "Nucleus.Event.Production.Data.Updated": [
            "Nucleus.Command.Production.Explorer.Reload"
        ],
        "Nucleus.Event.Production.Explorer.SingleSelection.DoubleClicked":
            "Nucleus.Command.USD.ViewPort.Open",
            "Nucleus.Command.USD.Outliner.Open",
        ],
    }
    [ ... ]
}
```

Figure 7: Application configuration (light blue) nominating certain models, views and event-command relationships of interest; Application instance (dark blue) initialized with this configuration; resulting Qt application comprising a main window with three dock widgets housing the three specified views.

Applications through configuration. By enforcing a pattern whereby applications are configured aggregations of plugin components, we essentially create "code-free applications". That is, all functionality is expressed by reusable components and there is a clear distinction between component authoring and application authoring. Where applications need a specific flavour or variant of an existing component, it becomes necessary to add this functionality to the shared components, as opposed to writing custom application-specific extensions. Over time, this allows the work of any given team to benefit all teams: a rising tide floats all boats.

Low barrier to entry. More generally, the centralized and string-based nature of application configurations make them very easy to read and edit. This allows even inexperienced users to readily create Nucleus applications.

3.3 Standalone and hosted modes

Initially, we developed two related Nucleus instantiation modes: standalone mode and hosted mode.

Standalone Mode. Our baseline Nucleus architecture was built for traditional, standalone applications that execute independently of any given DCC. Standalone mode is used by various in-house applications such as Filament, FilmStudio and RigStudio.

Hosted Mode. We also added support for hosted applications that run in the context of DCCs such as Maya or Houdini. These Nucleus applications utilize the same memory space as their host applications, but run with their own main windows separate from their host DCC main windows. Hosted mode is used by various in-house applications such as AnimationStudio, Artist Tools Palette, EnvironmentStudio, Forge, ModellingStudio, MultiShot and QCStation, all of which run hosted in Maya.

We introduced certain Nucleus models that **encapsulate aspects of host DCCs** such as their event and selection systems to support tight integration between the Nucleus application and the DCC application when running in hosted mode. Otherwise, the same Nucleus application configuration pattern is used in both modes.

4 NUCLEUS 2.0

Recurring feedback from our Nucleus 1.x clients was the desire to run Nucleus applications **more tightly integrated within their host DCCs**. Specifically, this would see Nucleus applications shipped *without* their own main windows, and instead with their views directly docked into the host DCC main window instead. Likewise, their other UI elements such as menus and toolbars would also be injected into the host DCC main window. The motivation was to create more unified UI/UX with tighter use of screen real estate, and workflows more coupled with the expectations of users.

4.1 Early work

We explored several prototypes of an embedding pattern in an attempt to address tighter host application integration, focusing on our key DCCs for modelling, layout and animation (Autodesk Maya™); surfacing and procedural effects (SideFX Houdini™); imaging and compositing (Foundry Nuke™); editorial reviews (Autodesk RV™) and USD asset introspection (Pixar Animation USDView™). A key challenge is that DCCs themselves have different levels of integration with Qt.

Native Qt. Some applications, such as RV and USDView, are natively developed in Qt. These are the easiest applications to target for a Nucleus embedded mode: we have direct programmatic access to their QMainWindow objects and can directly inject our QDockWidgets into these.

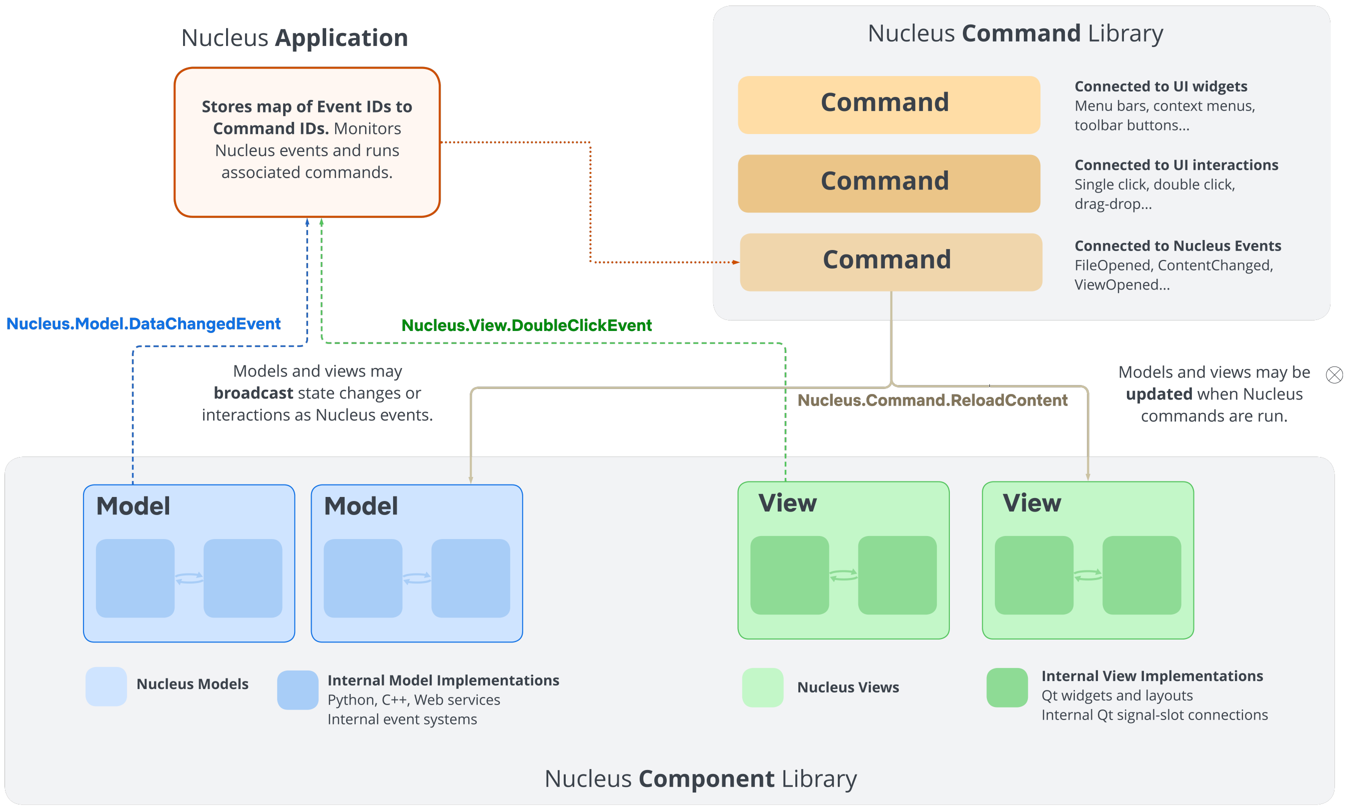

Nucleus Application
Stores map of Event IDs to Command IDs. Monitors Nucleus events and runs associated commands.
Nucleus Command Library
Command
Connected to UI widgets
Menu bars, context menus, toolbar buttons...
Command
Connected to UI interactions
Single click, double click, drag-drop...
Command
Connected to Nucleus Events
FileOpened, ContentChanged, ViewOpened...
Nucleus.Model.DataChangedEvent
Nucleus.View.DoubleClickEvent
Models and views may broadcast state changes or interactions as Nucleus events.
Nucleus.Command.ReloadContent
Models and views may be updated when Nucleus commands are run.
Model
Model
View
View
Nucleus Models
Internal Model Implementations
Python, C++, Web services
Internal event systems
Nucleus Views
Internal View Implementations
Qt widgets and layouts
Internal Qt signal-slot connections
Nucleus Component Library

Figure 8: Nucleus runtime ecosystem, comprising a plugin registry of Nucleus components representing models and views, another registry of Nucleus commands, and a central Nucleus application instance. Models and views broadcast Event IDs; these are intercepted by the Nucleus application, which maps these to a sequence of Command IDs and executes these commands synchronously or asynchronously, depending on the configuration. Commands in turn may then update model and view state.

Backports to Qt. Some applications, such as Maya and Nuke, were not originally developed in Qt, but in recent years have been rewritten for Qt. Nevertheless, they ship with proprietary layout management systems that add some complexity when integrating Nucleus components, as there is a layer of indirection between client code and the DCC user-interface Qt objects.

Non-Qt. Some applications, such as Houdini, are not written in Qt at all. They may still offer partial extensibility: for example, it is possible to register Houdini panes that encapsulate Nucleus views, but other aspects of the Houdini user-interface are less open to runtime modification.

In order to successfully target this wide range of DCCs, we first developed a **Layout Engine** system with DCC-specific extensions to gracefully handle the different capabilites of each DCC. For example, a toolbar command specified by a Nucleus application may be instantiated as an actual QToolButton in RV, but as a shelf script in Houdini. Likewise, a given Nucleus view may be docked in Maya using the `MayaQWidgetDockableMixin` class, whilst in Houdini it might use the equivalent `hou.Pane` class.

It is important to note that Nucleus offers many features that are not supported in *all* host applications; in select cases we are required to offer a reduced scope of functionality as needed to best suit the target, using a graceful degradation approach.

4.2 Level 6: Embedded mode

In early 2024, we released Nucleus 2.0 with full support for **Embedded Mode** applications in our target DCCs, using the Layout Engine system internally. A compelling feature of this update is that Nucleus configuration authors can work as before, **entirely agnostic to the specific mechanisms** needed to inject their applications into host DCCs. We treat these mechanisms as private implementation details of the framework.

Once again, such **separation of concerns** confers a range of benefits. Nucleus applications authored for standalone mode, for example, will work automatically in embedded mode without the need to rework these. Moreover, by centralizing the specific logic needed to integrate into a given DCC entirely with the Layout Engine system, we can readily upgrade this logic without needing to re-release any client code.

Following the deployment of a configuration-based MVC plugin architecture with support for standalone, hosted *and* embedded modes, along with commands, dependency inversion, shared widgets and shared visual design, we can consider a Qt codebase to be

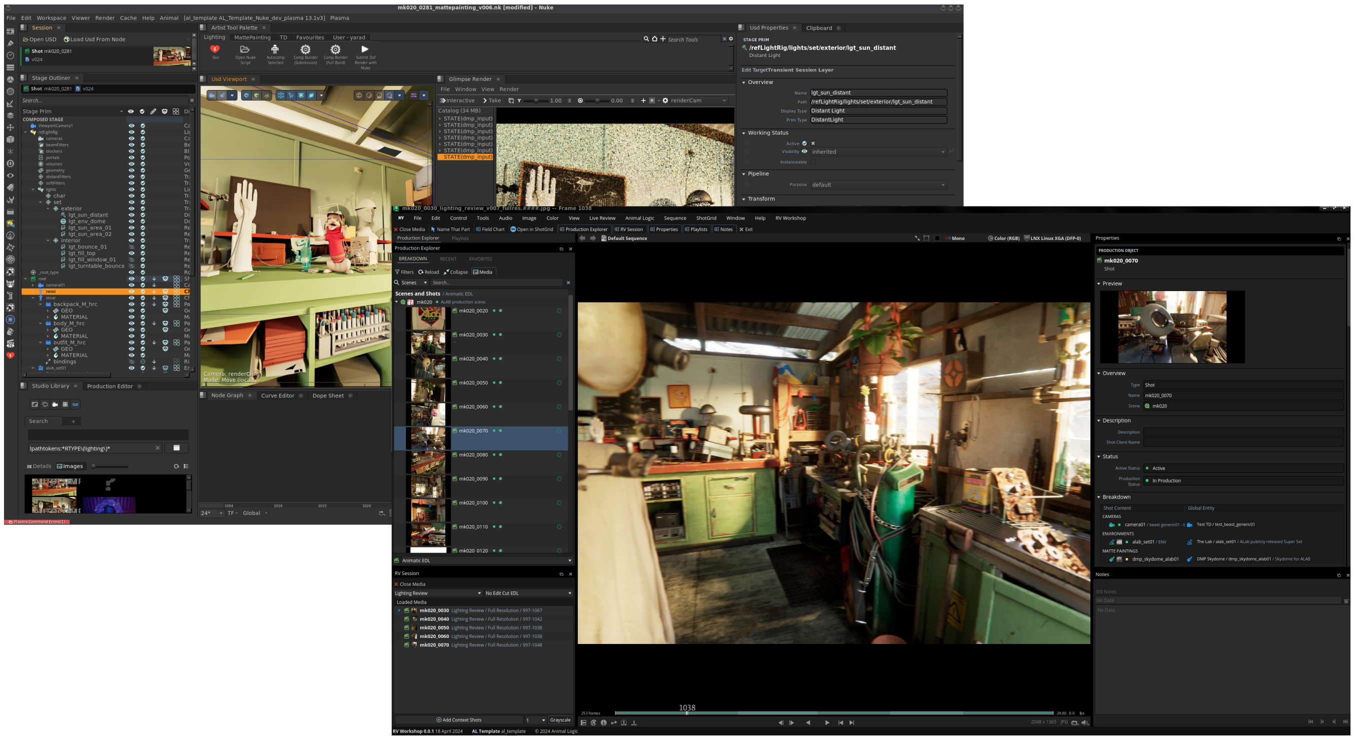


Figure 9: Various Nucleus 2.0 applications running embedded within host DCCs. Illustrated are Plasma, running within Nuke, and RVWorkshop, running within RV. We have fine-tuned our standard QSS stylesheets to incorporate some elements of host DCC look-and-feel to better integrate visually; a case in point is the orange selection color used by Plasma.

at the peak level of our model, **Application Maturity Level 6**. We have (provisionally) achieved this threshold as of 2024.

4.3 Results and discussion

The development of Nucleus 2.0 was not without its challenges: Nucleus had become a widely-used framework and it was critical to avoid regressions or interfering with production.

Initial work focused on treating embedded mode as a special case, leading to a large bifurcation in the Nucleus codebase with multiple parallel class hierarchies. After some trial-and-error we gained a **key insight**: redefine embedded mode as the standard case, and treat standalone and hosted modes as special cases in which the embedding target is simply a QMainWindow of our own creation. This elegant reimagining of the problem unblocked development and opened up a pathway towards a release of Nucleus 2.0.

Embedded mode has been **rapidly adopted** by a wide range of our craft groups. With the framework in place, we have worked with the Imaging team to develop the **Plasma** application for Nuke, as well as the **RVWorkshop** application for RV. In parallel, we have worked with the Procedural Assets and FX team to develop the **Alchemy** application for Houdini, and with the Assets team to develop **USDWorkshop** for USDView.

Migrating to embedded Nucleus can result in **simplification and modernization** of existing codebases, particularly those that rely on multiple DCC plugins as the basis for their functionality. For example, USDWorkshop replaces the use of twelve separate USDView plugins to achieve arguably a superior user experience; even more compellingly, RVWorkshop was able to replace no less than *forty* RV plugins with a single plugin.

5 FUTURE WORK

With the recent deployment of Nucleus 2.0, our current and future work involves working closely with craft groups to ensure successful embedded mode integrations with all our target platforms.

We continue to work on stability and performance, as well as additional features such as a command-line variant; improved support for asynchronous startup; migration to scalable SVG icons; improved end-user documentation; and also assisting in the development of a wider range of Nucleus components themselves.

We may potentially also transition to PySide, as this tends to be the preferred standard Qt Python binding. Currently, a lack of PySide Shiboken-related documentation is a limiting factor (relative to the more complete PyQt SIP binding reference guide), but this may change in future.

6 ACKNOWLEDGMENTS

The initial conception of a plugin-based Nucleus architecture was inspired by discussions with Aloys Baillet in 2018.

Many developers have contributed to the Nucleus framework and component ecosystem since then, with special thanks to Manuel Macha, Jonathan Penner and Cesar Saez for their work in taking this concept to a production footing. Special thanks also goes to

Oliver Dunn for earlier foundational work building up a common visual design language.

Finally, a special thanks to Romain Maurer for championing this initiative over many years.

REFERENCES

[1] Steve Agland, Jakub Jeziorski, Manuel Macha, Simon Bunker, Francesco Sansoni, and Eoin Murphy. 2020. Grip and Filament: A USD-Based Lighting Workflow. In *ACM SIGGRAPH 2020 Talks* (Virtual Event, USA) *(SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 33, 2 pages. https://doi.org/10.1145/3388767.3407350

[2] Aloys Baillet, Eoin Murphy, Oliver Dunn, and Miguel Gao. 2018. Forging a new animation pipeline with USD. In *ACM SIGGRAPH 2018 Talks* (Vancouver, British Columbia, Canada) *(SIGGRAPH '18)*. Association for Computing Machinery, New York, NY, USA, Article 54, 2 pages. https://doi.org/10.1145/3214745.3214779

[3] Azad Bolour. 2003. Notes on the Eclipse Plug-in Architecture. (2003). Retrieved May 7, 2024 from https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[4] Jon-Patrick Collins, Romain Maurer, Fabrice Macagno, and Christian Lopez Barron. 2022. USD at Scale. In *The Digital Production Symposium* (Vancouver, BC, Canada) *(DigiPro '22)*. Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/3543664.3543677

[5] Michael De Caria, Prethish Bhasuran, Mitja Müller-Jend, and Manuel Macha. 2023. Matte Painting a Brighter Future: A USD-Based Toolset in Nuke. In *Proceedings of the Digital Production Symposium* (Los Angeles, CA, USA) *(DigiPro '23)*. Association for Computing Machinery, New York, NY, USA, Article 10, 5 pages. https://doi.org/10.1145/3603521.3604294

[6] Martin Fowler. 2010. Richardson Maturity Model. (2010). Retrieved May 10, 2024 from https://martinfowler.com/articles/richardsonMaturityModel.html

[7] Martin Fowler. 2022. Conway's Law. (2022). Retrieved May 6, 2024 from https://martinfowler.com/bliki/ConwaysLaw.html

[8] Wikipedia. 2023. Facade pattern. (2023). Retrieved May 6, 2024 from https://en.wikipedia.org/wiki/Facade_pattern

[9] Wikipedia. 2024. Command pattern. (2024). Retrieved May 6, 2024 from https://en.wikipedia.org/wiki/Command_pattern

[10] Wikipedia. 2024. Dependency inversion principle. (2024). Retrieved May 6, 2024 from https://en.wikipedia.org/wiki/Dependency_inversion_principle