A Framework to Facilitate Reusable, Modular Widget Design for Real-Time Interactive Systems

Carmine Elvezio * Columbia University
Mengu Sukan* Columbia University
Steven Feiner* Columbia University

Abstract

Game engines have become popular development platforms for real-time interactive systems. Contemporary game engines, such as Unity and Unreal, feature component-based architectures, in which an object's appearance and behavior is determined by a collection of component scripts added to that object. This design pattern allows common functionality to be contained within component scripts and shared among different types of objects. In this paper, we describe a flexible framework that enables programmers to design modular, reusable widgets for real-time interactive systems using a collection of component scripts.

We provide a reference implementation written in C# for the Unity game engine. Making an object, or a group of objects, part of our managed widget framework can be accomplished with just a few drag-and-drop operations in the Unity Editor. While our framework provides hooks and default implementations for common widget behavior (e.g., initialization, refresh, and toggling visibility), programmers can also define custom behavior for a particular widget or combine simple widgets into a hierarchy and build arbitrarily rich ones. Finally, we provide an overview of an accompanying library of scripts that support functionality for testing and networking.

Keywords: Widget, Framework, Modular, Component-based Design, Augmented Reality, Virtual Reality, Unity

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; H.5.2 [Information Interfaces and Presentation]: User Interfaces—Input devices and strategies, Interaction styles; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques

1 Introduction

Frameworks that support the creation of rich *Realtime Interactive Systems* (RIS) often focus on integration and provide modules that each handle specific aspects of the system (e.g., tracking, input, networking, and rendering) and can be combined in different ways to provide custom-tailored solutions for heterogeneous computing environments (e.g., [2, 15, 13, 4, 17]). Using these frameworks, programmers can prototype and test UI elements. Additionally, many of these frameworks allow the programmer to extend their baseline functionality, following an object-oriented design pattern, often by subclassing (i.e., inheriting from) the base classes of the framework. Meanwhile, the entity–component–system architectural pattern has become popular in game development [11, 21]. This pattern promotes the principle of composition over inheritance. Every game entity (referred to as a *GameObject* in Unity [18], and as a *GO* in this paper) is added to the scene and placed in a scene graph. GOs get their behavior or functionality from one or more attached *components*. This allows the same functionality or behavior to be reused by different types of GOs. While this pattern provides a powerful framework for the creation of sophisticated behavior, it is often left to the programmer to manage the interaction between specific components.

Widgets (collections of GOs that are reusable, modular, and ready to communicate with each other) can require careful management by the programmer. For instance, imagine a scenario in which the programmer would like to implement a widget that instructs the user to move a tracked physical prop from one point to another. This could be communicated by using a *bundle* of simple widgets; for example, by rendering virtual highlights at the beginning and end points and a virtual arrow at the end point to indicate a motion vector. To make this bundle of widgets more interactive, the arrow could change size as the prop moves and the highlights could change color based on the location of the prop.

While in development, the programmer may want to test another bundle of widgets for the same task or swap one of the widgets in the bundle with another widget (e.g., a virtual copy of the prop that animates towards the end point instead of an arrow). There are many scenarios in which the complexity of managing these widgets and the bundles into which they are combined requires much attention and care by the programmer. To help reduce this overhead, we propose that programmers create modular, reusable widgets that share a common program interface (i.e., are interchangeable in software) and can be combined together to create arbitrarily complex widgets via combinations. We present *WF Toolkit*, a toolkit that centers around an extendible *Widget Framework*, which includes software components that facilitate management of widget-related components at runtime. In addition, our toolkit includes a user-study–management component that supports rapid user testing and assessment of widgets implemented with our framework.

2 Previous Work

There has been much effort devoted to developing fully integrated RIS application frameworks. Many of the early systems in this area, some of which we describe below, provided a complete engine, supporting rendering, networking, input, events, and integration of 3D tracking systems, along with an expandable framework running on that engine.

COTERIE [10], made it possible to develop distributed systems through object-oriented data distribution, including a distributed scene graph. Using COTERIE, MacIntyre and colleagues built a library to support a range of displays and trackers, and developed prototype augmented reality (AR) applications using head-worn displays for maintenance and outdoor mobile AR [6].

Studierstube [15] was designed to support programmers in creating collaborative virtual and augmented environments, implementing an architecture proposed earlier [8]. It emphasized annotations and interactions designed for a two-handed pen-and-pad interface. Studierstube was developed as a collection of C++ classes built on top of Open Inventor [16] and featured a distributed scene graph that enabled heterogeneous networked architectures including head-worn displays and projectors. Other versions provided support for mobile devices (backpack [14] and handheld [20]).

DWARF [2] was a modular, distributed, platform-independent framework that allowed programmers to quickly prototype AR ap-

*e-mail: {ce2236, m.sukan, skf1}@columbia.edu

plications by modifying XML files to customize services that communicated through CORBA. Built-in services included tracking, scene description, task flow, and UI.

MORGAN [13], introduced in 2004, was a distributed, modular C++ library for building heterogeneous AR and virtual reality (VR) applications. Its modules communicated through CORBA and included a platform-independent render engine and a device abstraction layer for various input/output devices. One of the unique features of MORGAN was its "External Scene Graph," which used plug-ins to map 3D scene information stored in various third-party file formats onto its own internal representation.

ARCS [4], a more recent modular framework, supports integration of exogenous modules with endogenous modules (e.g., tracking, rendering, and input systems). ARCS allows programmers to extend existing module types, define new module types by utilizing the abstract module design, and define relationships between modules through explicit (defined by the programmer) and implicit (defined by object initialization) communications. The framework also supports the creation of complex compound modules, which can then be integrated into larger workflows though macros.

GoblinXNA [12] and Bespoke [19] were similar frameworks built on the XNA platform [22], with support for integration of external tracking and input systems. In addition, both provided a scene graph that, in combination with XNA and C#'s general approachability, made it easy to create complex scenes. As both frameworks utilized object-oriented design patterns, expansion of the frameworks required that programmers derive the platform's scene graph nodes to introduce new functionality not supported by the base implementations.

As mentioned earlier, a common thread among these earlier systems is that their efforts are concentrated on bridging various low-level RIS components such as scene graph, rendering, tracking, I/O (i.e. displays, peripherals, etc.), and networking. More recently, Unity [18], a modern game engine with a large and active game developer community, has become popular among AR/VR researchers because (a) it supplies much of the core functionality provided by earlier toolkits and frameworks out of the box (e.g., platform independence, scene graph, rendering, physics, and networking) and (b) it makes it easy to interface with various AR/VR-related hardware and software (e.g., Oculus Rift, Leap Motion Controller, and PTC Vuforia) through vendor-provided plug-ins.

A few AR/VR frameworks that have been published in recent years (e.g., RUIS [17] and MiddleVR [9]) rely on Unity as a platform, yet still focus on integrating VR/AR hardware and software into a common framework. They provide an abstraction layer on which developers can build immersive VR applications with spatial interaction and stereoscopic 3D graphics, similar to some of the earlier systems mentioned above.

In addition to integration-focused systems, there has been some effort in providing a collection of reusable 3DUI elements. For example, Figueroa and Castro [7] developed a library of reusable, abstract, low granularity 3D selection and travel techniques that can be combined to prototype novel 3D interaction techniques. Their C++ implementation is built on top of VR Juggler [3], a modular graphics and devices abstraction library for VR applications.

Some of the work in related areas extend scene graphs to introduce logical connections between spatially disconnected scene elements. One example is the Virtual Manufacturing Lattice [1], which augmented a scene graph to support applications in virtual manufacturing. By including precedence relationships and object state at the node level, it enabled enforcing rules in virtual assembly planning and prohibiting execution of infeasible operations.

Our work on WF Toolkit builds on these concepts, yet differs in the following significant ways: We focus on providing programmers with a flexible yet powerful framework that allows them to define and implement a custom interface for modular, interchangeable widgets. Our framework's component-based design and Unity implementation allow easy drag-and-drop instantiation and flexibility for extension when required. A set of widgets that are implemented using WF Toolkit can easily be combined together to build full-featured user interfaces and can be administered through a common managing entity that is agnostic to the specific widgets it has to manage. Our framework is independent of the composition of the underlying hardware and software physical setup (e.g., displays and tracking). In fact, our Unity implementation can potentially be used in conjunction with other Unity-based frameworks that focus on integration. The background for our framework's evolution and the design decisions that underlie it are described in the next section.

3 WF Toolkit

To support the creation of modular, reusable widgets within a component-based game engine such as Unity, WF Toolkit provides:

1. A component-based framework to streamline the creation of modular widgets by implementing a simple interface and adding a drag-and-drop component for handling communication among widgets.

2. A management module for these widgets that can work on both local and networked configurations.

3. Tools for rapid prototyping and user testing of systems that include these modular widgets.

In the following section, we describe our implementation of this toolkit in the Unity game engine (Figure 1). In Unity, the scene graph or scene hierarchy is referred to as the *Hierarchy* and presented in the Unity Editor Hierarchy window. The components described in the following sections refer to Unity components, whose instances are attached to GOs. Unity components are classes that inherit from a base class, MonoBehaviour, which defines functions for hooking into the standard game initialization and update loop. As C# classes, components can implement interfaces and be subclassed (i.e., inherited from).

3.1 Design

WF Toolkit was designed during the development of software for a 3D rotation-guidance study [5] that compared different widgets for guiding a user in rotating a tracked object to a desired 3D orientation. Initially, a team of programmers developed several widgets in parallel across different Unity projects. When it was time to integrate those projects into a single user-study–management system, it proved to be a difficult engineering challenge. In Unity, access to GOs and their components is simplified through convenient utilities provided by Unity itself. However, managing which GOs to activate at a specific time and calling functions defined within their components is left to the programmer.

As our widgets started growing more complex, they began to span multiple component scripts across multiple GOs. Generally, these GOs were grouped together in the Unity scene hierarchy as children of the main widget GO, but we also encountered cases in which sub-GOs were spatially independent of the main GO and therefore could not be children of it. One example is when a 3D widget has a 2D UI element associated with it. Following the Unity UI guidelines, 2D UI elements should be children of a separate *canvas* GO. Such instances required each of our widgets to maintain a list of their associated GOs and call specific functions on them when they received function calls from an upstream managing entity. For example, we found ourselves replicating the functionality of the Unity *SetActive()* function. Normally, when *SetActive()* is invoked on a GO, it will automatically enable or disable all attached components and their children. However, since we had created widgets that span disparate hierarchies, a single invocation of Unity's *SetActive()* was not enough to activate or deactivate all related GOs. Thus, we needed to explicitly manage the *SetActive()* of each associated GO.

Additionally, we wanted to create compound widgets by reusing other simpler widgets as building blocks. Hooks to those lower-level widgets again needed to be explicitly managed by the higher-level widgets that included them as building blocks. For example, we wanted to share among different widgets a camera controller that accessed the main camera in the scene, transformed it to a new pose when each widget was activated, and then returned the camera to its original pose once that widget was deactivated. To follow Unity's component-based architecture, we initially created a separate camera controlling script that could be attached to multiple widgets as a component. However, even then we found ourselves placing repetitive code inside the widgets themselves to handle (e.g., initialize and activate) this camera-control behavior.

When we started considering how to encapsulate common features between widgets, we encountered an additional problem. Following a standard object-oriented model, we could have placed the needed behavior in a base class from which all widgets would derive. This would have solved the problems of providing common functionality, and of accessing behaviorally-related items through a single type. However, one of the common problems encountered in object-oriented patterns poses a difficult problem here. Widgets that are designed in different pipelines, and have undergone extensive development, required major refactoring to extend a common base class. Another possibility was to define an interface that associated components would implement. This allowed for common access (through an interface handle) but did not support an internal data structure that could be used to group components together. So we needed a way to allow programmers to develop widgets in their own pipelines, but take advantage of shared functionality and a common handle that could be used to get access to all related items.

To summarize, we needed a design that would allow us to:

1. Provide a common type that a managing entity could use to access all widgets (potentially hierarchically) without knowing the details of their startup, update, or other internal behavior.
2. Specify nonspatial hierarchies that a programmer could use to get access to GOs and component scripts comprising a widget.
3. Encapsulate behavior that some or all widgets share, without forcing the programmer to redesign their class structure.
4. Allow components to be reused by potentially diverse widgets.

To fulfill these requirements, we created the Widget Framework and developed *WF Node*, a component that can be added to GOs, providing a common hook to be used by a managing entity. We also developed the IWFBehavior interface, a simple interface that programmers can implement; this allows scripts to react to calls from WF Nodes, which pass along calls from the managing entity.

3.2 Widget Framework

The core of our toolkit is the Widget Framework. The *Widget Framework* is a collection of components (Figure 2) that a programmer can attach to GOs to create modular widgets that can be managed through a common interface.

3.2.1 IWFBehavior

The behavior of individual widgets is defined in components that implement the *IWFBehavior* interface. There are five core behavior functions defined in *IWFBehavior*: *Initialize()*, *Activate()*, *Update()*, *Reset()*, and *Destroy()*. *Initialize()* is run when the widget is

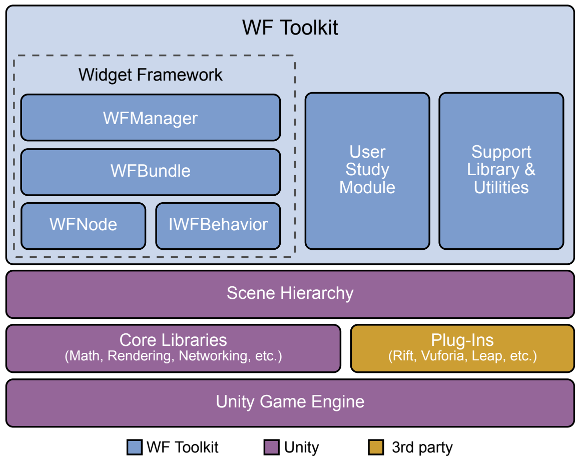


Figure 1: Widget Toolkit modules are layered on top of the Unity scene hierarchy. Interfacing with third-party hardware and software is established through plug-ins.

loaded. *Activate()* has a boolean toggle as an argument to turn the widget on or off. *Reset()* reinitializes the widget to its initial state and *Destroy()* removes the widget and its associated GO from the scene for garbage collection. *Update()* gets called every frame and is where the programmer specifies the widget behavior. Individual widgets are typically unaware of their siblings or children. The hierarchical setup is handled by the framework through the *WFNode* component.

3.2.2 WFNode

WFNode is a bridge between the main program, individual widgets, and their children. Each widget bundle has a *WFNode* component attached to its main GO. *WFNode*s can contain a list of children (i.e., references to other *WFNode*s), which allows for the creation of a *WFNode* hierarchy. As long as the main program has handles to each widget bundle in the scene (this can be managed through a manager component, *WFManager*), the framework does not require that a root *WFNode* be specified for the entire scene. *WFNode*s do not contain any widget-specific logic or behavior—all logic and behavior is contained in components that implement the *IWFBehavior* interface. *WFNode*s may contain multiple *IWFBehavior*s to achieve complex behavior.

To be able to function as a bridge, *WFNode* has implementations of the same behavior functions defined in *IWFBehavior*: *Initialize()*, *Activate()*, *Update()*, *Reset()*, and *Destroy()*. In a *WFNode*, each of these functions does two things: call the implementations of the function for each attached *IWFBehavior*, and pass the call on to its children (i.e., other *WFNode*s). *WFNode*s also contain utility functions to get a handle to the *WFNode* attached to a GO. These utility functions can streamline development, allowing the programmer to use the Unity Editor's drag-and-drop functionality to drop GOs with *IWFBehavior* components onto designated spots in the *WFNode*'s inspector window. To summarize, all a programmer needs to do to integrate with the widget framework is to add a *WFNode* to a GO in the scene hierarchy and implement the necessary specialty functions defined in the *IWFBehavior* interface.

To further streamline development, we provide a default implementation of the interface that includes basic behavior such as toggling the visibility of a GO or removing it from the scene. This allows the programmer to focus on core functionality and write code only when more advanced functionality is required.

The functions declared in the *IWFBehavior* interface are invoked by a managing entity through *WFNode*s and provide functionality that is not possible by relying only on Unity's built in functions. An example of this is initializing *IWFBehavior*s. A parameter to a *IWFBehavior* may be computed in the managing entity's *Awake()*, which may not have run by the time *IWFBehavior*'s own *Awake()* is called. Normally, the solution would be to move the initialization logic of the *IWFBehavior* from *Awake()* to *Start()*, so that it can safely be called after all *Awake()*s have been called. However, it is possible (which we encountered ourselves during the development of our rotation-guidance user study) that the managing entity's *Start()* may run first and try to reference a value that gets initialized in a *IWFBehavior*'s *Start()*.

Our solution is to utilize the *IWFBehavior Initialize()* function, which the managing entity can call explicitly at a specific time (e.g., after all *Awake()*s, but before any *Start()*s). While it is possible and recommended in some situations to explicitly set the script execution order in Unity directly, defining a common initialization point in the framework is helpful for promoting standardization and interchangeability between various *IWFBehavior* implementations.

The *Activate()* function serves a critical role in the base framework. *WFNode*'s *Activate()* function will iterate over all attached *IWFBehavior*s and all its own children. This allows the programmer to activate associated widgets (that may or may not be linked in the scene hierarchy) without any work in the *IWFBehavior* implementations. This both simplifies usage of the system and prevents errors when implementing the *IWFBehavior* interface. The programmer may simply place a Unity *SetActive()* in that function (allowing activation of all elements of the system), or may specify additional behavior that should occur before or after the *IWFBehavior* implementation is activated or deactivated. This becomes especially powerful in networked configurations in which synchronization and bandwidth conservation are important. The *Activate()* function can be triggered remotely, allowing for an extremely simple network synchronization of active states, without the need to turn all associated components into derivations of the Unity *NetworkBehaviour* class.

3.2.3 WFBundle

Together, a *WFNode* and the associated *IWFBehavior* implementations form a *WFBundle*. A *WFBundle* is a logical hierarchy that connects components that may be contained in multiple GOs across the scene hierarchy. To a managing entity, the *WFNode* is the entry point to a *WFBundle*. As a *WFNode* can point to multiple *IWFBehavior* implementations (within a single GO or across multiple GOs), as well as other *WFNode*s (which must be in different GOs), *WFBundle*s can grow wide and deep. This allows for complex, multi-tiered widgets and interactions.

In our Unity implementation, *WFBundle*s are prepared in one of three ways: completely in code, through the composition of GOs and component scripts at runtime, or through use of Unity Prefabs. *Prefabs* allow a programmer to save a *sub-hierarchy* (i.e., portion of the scene hierarchy), including component scripts attached to GOs and variable settings set through the Unity Inspector panel. While Prefabs facilitate the creation and deployment of *WFBundle*s, they do not allow for automatic connection to external components (i.e., components outside of the Prefab's sub-hierarchy). While it is possible to connect to certain component instances in script, this is usually only possible for Unity-provided components, globally visible and static components, and components attached to GOs known to the calling component. To connect to components outside of the sub-hierarchy, the programmer can use the *WFManager* to specify relationships formed at runtime.

3.2.4 WFManager

WFManager is not a required component; however, without a managing entity that can bridge and control multiple, unrelated *WFBundle*s, their utility is limited and the programmer must then manage

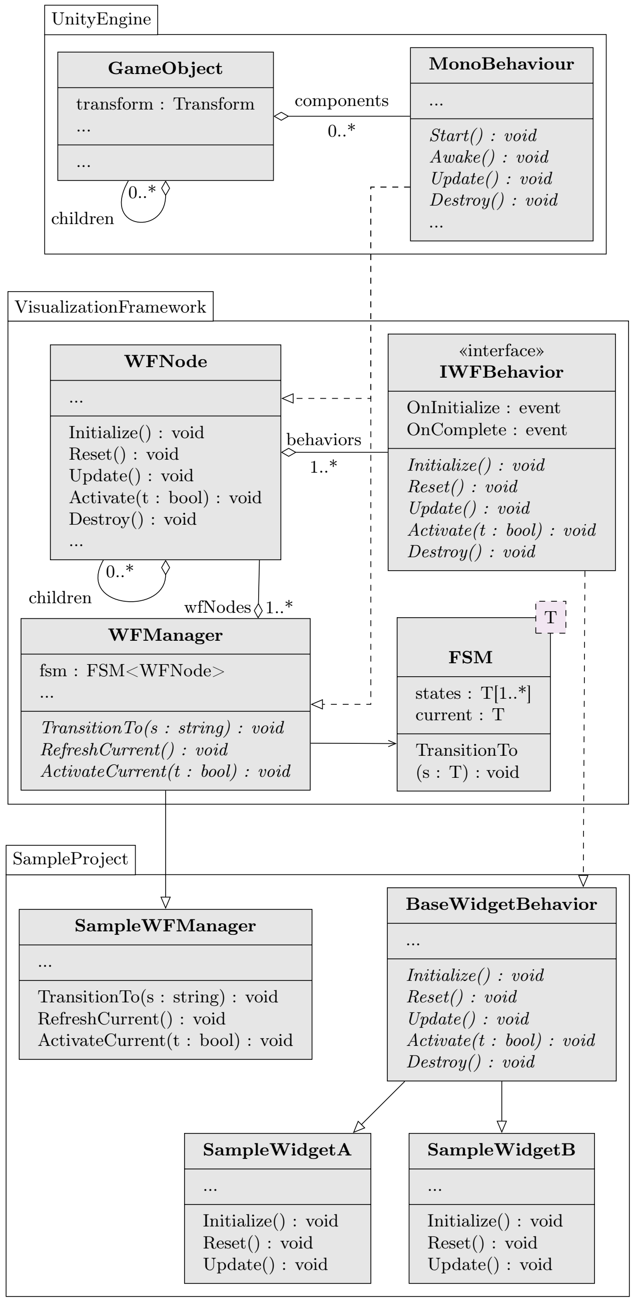


Figure 2: Simplified UML class diagram showing inheritance and containment relationships among Widget Framework items (*WFNode*, *IWFBehavior*, and *WFManager*), as well as downstream inheritance relationships for a sample project that uses the Widget Framework.

the *WFBundle*s manually. Thus, we provide an extendible managing component, the *WFManager*, that handles all registered *WFBundle*s in the scene and can call the *WFNode* specialty functions on active *WFNode*s. It is possible to have multiple *WFManager*s within a single application instance. Our toolkit also provides a networked *WFManager* module that makes it easy for programmers to synchronize *WFNode* hierarchies over a network. In our Unity implementation, networked *WFNode*s and *WFManager*s communicate through the Unity UNET system.

We represent the collection of *WFNode*s in a *WFManager* as a finite state machine (FSM), where each state is a *WFNode*. The entry and exit procedures for each state automatically handle each *WFNode*'s active state. Further, the entry and exit behaviors can be expanded to customize transitions between *WFNode*s. The data structure can also be changed to match the programmer's needs.

3.3 WUSM

WFNode and the *WFManager* are the foundation for an extendible and reusable user-study module, the Widget User-Study Manager (WUSM). Since the programmer's widget components are handled by *WFManager*s in the scene, the WUSM primarily supports loading and saving study data, study-trial management, and study flow. The WUSM references a user-study–state manager, which internally uses a FSM to represent the state of the user study. Like the *WFManager*'s state machine, the programmer can specify the state entrance and exit behaviors to customize transitions between user-study states.

In the WUSM, *WTrial*s are objects that contain data relevant to a particular study trial. *WTrial*s reference *WTask*s, which are objects that contain their own update loop, initialization function, and optionally, completion information. *WTask*s run independently of the calling widget (potentially across multiple application instances and network hosts), and can identify when a task has been completed. Further, each *WTask* object contains a list of *WTask* references, which can be useful when trying to create compound tasks, as in some assembly scenarios.

4 Use Case

As mentioned earlier, our toolkit was used to develop a system to guide users in rotating a tracked object to a destination 3DOF orientation in AR/VR, building on our earlier work [5]. For example, one widget we developed uses multiple 3D arrows to show the rotation (Figure 4a), while an alternative widget animates a semi-transparent replica of the tracked object (Figure 4b). Since these widgets did not share a base implementation, and the requirements for operating the widgets were very different, we needed a common interface for managing, testing, and deploying the interfaces.

In Unity, the *WFBundle*s that represent the arrow-based and animation widgets are created through a collection of Unity GO components. The components are brought together and managed through derivations of the base class that implements *IWFBehavior* as described above, which are then added to GOs in the Unity scene hierarchy. A *WFNode* is added to the same GO. All widgets to be tested are added to a list in the *WFManager*, also present in the scene hierarchy. The complexity of managing multiple behaviors is hidden from the programmer through use of a *WFManager* and its connections to *WFBundle*s through *WFNode*s, which automatically pass messages onto downstream *IWFBehavior*s.

In the example scene hierarchy shown in Figure 3, Arrow and Animation *WFBundle*s are controlled by a *WFManager*. When the *WFManager* makes *WFNode* function calls, those calls are passed on to the *IWFBehavior* components attached to each *WFBundle*. In the case of the Animation *WFBundle*, there are three *IWFBehavior*s: Animation, Camera, and Placement. The end effect of this set-up is that when the Animation *WFBundle* is activated, the camera is placed at a specific offset from a virtual object, a replica of the virtual object is placed next to it, and the replica animates along a motion path specified by the programmer inside the Animation behavior.

Without the Widget Framework, programmers would need to explicitly connect the widgets in code, or bundle them into prefabs using the Unity Editor. (Note that bundling GOs into prefabs does not work for GOs that are associated together but are spatially independent.) Further, if the Camera behavior and Placement behavior are added to higher-level widgets as sub-widgets, our Widget

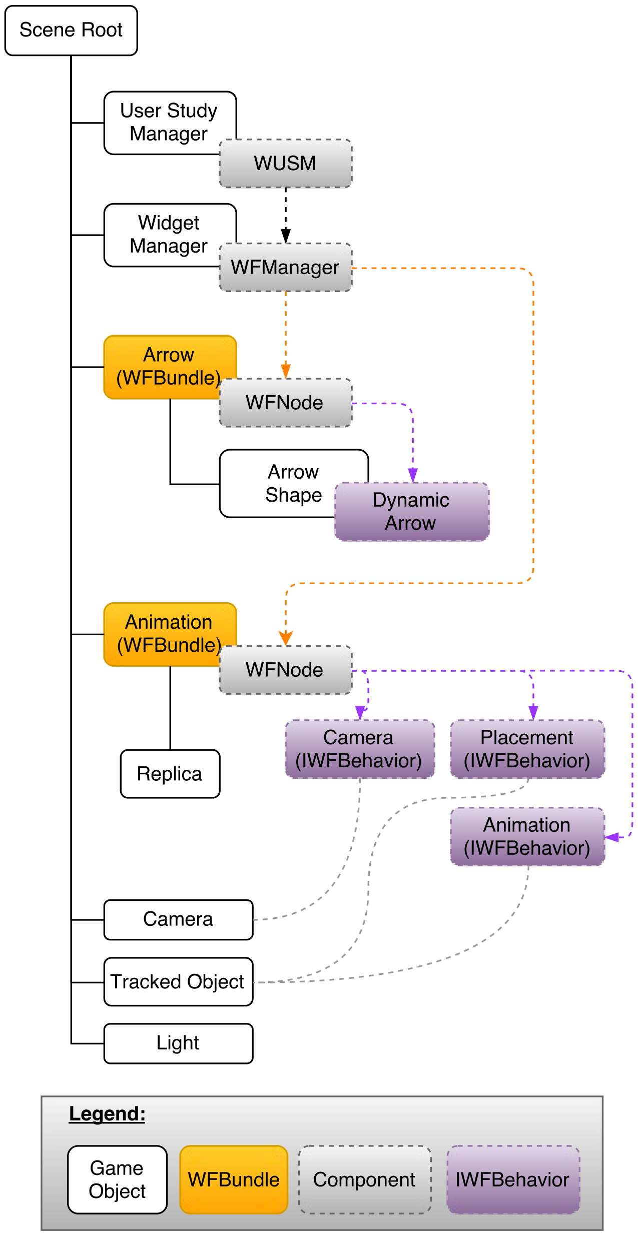
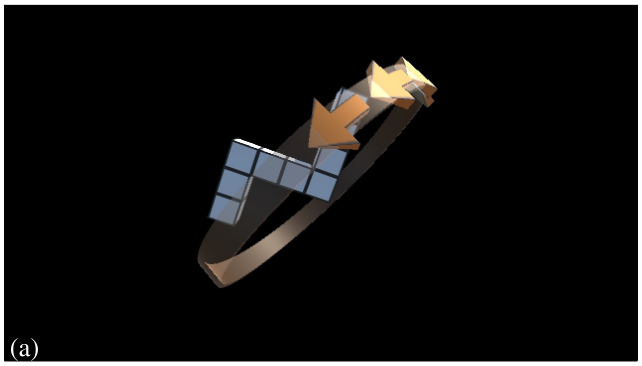


Figure 3: Simplified scene hierarchy from a sample project with two *WFBundles*: Arrow and Animation. *WFBundles* are controlled by the *WFManager*, which accesses the *WFBundle*s through their *WFNode*. Animation is a composite *WFBundle* and contains three *IWFBehavior*s: Animation, Camera, and Placement.

Framework not only allows defining those relationships by a single drag-and-drop operation, but it also automatically registers them to receive function calls from higher-level entities, which the programmer would need to carefully and tediously manage within each high-level widget without the help of our framework.

During the prototyping phase, widgets are managed through the Widget Test Manager, which references a networked *WFManager*. The Widget Test Manager handles scene initialization and widget iteration through the Unity Input module, since the *WFManager* manages the *WFBundle*s and directly instantiates *WFNode*s and *IWFBehavior* implementations through instantiation of GOs and/or Prefabs.



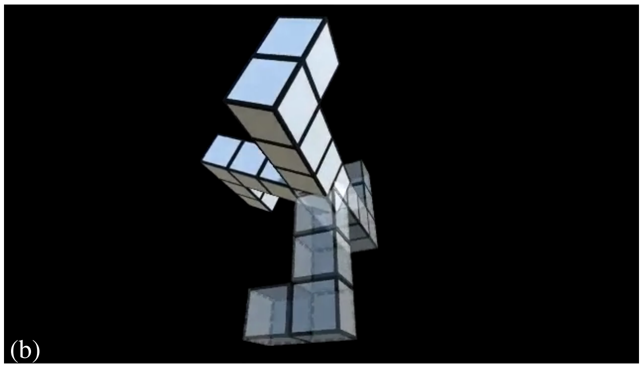


Figure 4: Sample widgets from a project that is built using WF Toolkit to guide users in rotating a shape by showing: (a) Circular arrows oriented about a single axis. (b) An animated representation of the rotation between the current and final poses overlaid on the tracked object.

When the *WFBundle*s are ready, the *WFManager* used during development is simply loaded into a WUSM in a GO in the scene hierarchy. In a user study, when a widget is needed for a trial, the WUSM requests a widget change from the *WFManager*. The *WFManager* uses the same *WFBundle*s regardless of whether the calling component is a WUSM or a Widget Test Manager. In the case of the Arrow *WFBundle*, arrow shape and material, and widget behavior are defined during the development phase, and saved to the *WFBundle* GO/Prefab, which is then directly used in the *WFManager* initialization process.

When the application starts, the *WFManager* will iterate through its list of registered widget GOs or Prefabs and instantiate new widgets in the scene. During initialization, the *WFManager* will create states for each *WFNode* to add to the widget FSM described above. The application-specific initial state will be loaded and then the *WFManager* will iterate over all other *WFNode*s to disable the registered widgets.

As the user study progresses between trials, or when a user manually triggers a change, the *WFManager* will invoke the *Deactivate()* function on the current *WFNode* (which will in turn call all associated *IWFBehavior Deactivate()* functions) to disable that widget, and will then invoke the *Activate()* function on the next *WFNode* to enable it. If the programmer has specified that the settings of a widget must be refreshed, the *WFManager* can invoke the *WFNode Refresh()* function, which will again call all associated *IWFBehavior Refresh()* functions. *Initialize()* and *Destroy()* work similarly. In each case, the programmer specifies how a widget will operate when the respective *IWFBehavior* function is invoked by the associated *WFNode*.

5 Conclusions and Future Work

We have presented the design and implementation of WF Toolkit, a flexible and extensible component-based framework that allows for the streamlined development and rapid deployment of modular, reusable widgets. Our framework is designed to connect widgets into bundles, all of which are administered by a single managing script. In addition, WF Toolkit includes a networked implementation of this manager for distributed and multi-user scenarios, as well as a module that facilitates user testing. We demonstrated the advantages of WF Toolkit through a use case, in which we implemented different widgets for rotation guidance in AR/VR scenarios. We intend to release the Unity implementation of the framework to the general public and to move the toolkit into an open-source development pipeline, to further expand its capabilities.

Going forward, we hope to specify broadcast and messaging protocols for use with *WFNode*s, to provide an alternative to inter-node communication. We would also like to introduce search and reorganization capabilities to *WFNode*s, to allow for simple reorganization of hierarchies. In addition, we plan on releasing a library of low-level widgets such as arrow, highlighter, and animator, that can be used as building blocks to design novel widgets. We envision that different widgets that serve the same purpose would implement a common interface. This would allow programmers to release novel alternative widgets, and other programmers to integrate those new widgets into their projects, swapping their existing widgets with new ones in a plug-and-play fashion.

Acknowledgements

This research was funded in part by National Science Foundation Grant IIS-1514429.

References

[1] A. Banerjee and P. Banerjee. A Behavioral Scene Graph for Rule Enforcement in Interactive Virtual Assembly Sequence Planning. *Comput Ind*, 42(2-3):147–157, 2000.

[2] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proc. IEEE ISAR*, pages 45–54, 2001.

[3] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proc. IEEE VR*, pages 89–96, 2001.

[4] J.-Y. Didier, M. Chouiten, M. Mallem, and S. Otmane. ARCS: A framework with extended software integration capabilities to build Augmented Reality applications. In *Proc. IEEE VR Workshop on SEARIS*, pages 60–67, 2012.

[5] C. Elvezio, M. Sukan, S. Feiner, and B. Tversky. [POSTER] Interactive Visualizations for Monoscopic Eyewear to Assist in Manually Orienting Objects in 3D. In *Proc. IEEE ISMAR*, pages 180–181, 2015.

[6] S. Feiner, B. MacIntyre, T. Hllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. In *Proc. IEEE ISWC*, pages 74–81, 1997.

[7] P. Figueroa and D. Castro. A reusable library of 3D interaction techniques. In *Proc. IEEE 3DUI*, pages 3–10, 2011.

[8] M. Gervautz, D. Schmalstieg, Z. Szalavri, K. Karner, F. Madritsch, and A. Pinz. Studierstube- A Multi-User Augmented Reality Environment for Visualization and Education. *Technical report TR-186-2-96-10, Institute of Computer Graphics, TU Vienna*, 1996.

[9] S. Kuntz. MiddleVR. `http://www.middlevr.com`.

[10] B. MacIntyre and S. Feiner. Language-level Support for Exploratory Programming of Distributed Virtual Environments. In *Proc. ACM UIST*, pages 83–94, 1996.

[11] R. Nystrom. *Game Programming Patterns*. Genever Benning, 1 edition edition, 2014.

[12] O. Oda and S. Feiner. Goblin XNA Framework. `http://goblinxna.codeplex.com/`.

[13] J. Ohlenburg, I. Herbst, I. Lindt, T. Frhlich, and W. Broll. The MORGAN framework: enabling dynamic multi-user AR and VR projects. In *Proc. ACM VRST*, pages 166–169, 2004.

[14] G. Reitmayr and D. Schmalstieg. A wearable 3D augmented reality workspace. In *Proc. IEEE ISWC*, pages 165–166, 2001.

[15] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavri, L. M. Encarnao, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. *Presence Teleoper Virtual Env.*, 11(1):33–54, 2002.

[16] P. S. Strauss and R. Carey. An Object-oriented 3D Graphics Toolkit. *Proc. ACM SIGGRAPH*, 26(2):341–349, 1992.

[17] T. M. Takala. RUIS: A Toolkit for Developing Virtual Reality Applications with Spatial Interaction. In *Proc. ACM SUI*, pages 94–103, 2014.

[18] Unity Technologies. Unity Game Engine. `http://unity3d.com`.

[19] P. D. Varcholik, J. J. LaViola , Jr., and C. Hughes. The Bespoke 3DUI XNA Framework: A Low-cost Platform for Prototyping 3D Spatial Interfaces in Video Games. In *Proc. ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 55–61, 2009.

[20] D. Wagner and D. Schmalstieg. First steps towards handheld augmented reality. In *Proc. IEEE ISWC*, pages 127–135, 2003.

[21] Wikipedia. Entity component system. `https://en.wikipedia.org/wiki/Entity_component_system`.

[22] Wikipedia. Microsoft XNA. `https://en.wikipedia.org/w/index.php?title=Microsoft_XNA`.