Stefan Kohl, BSc

Design and Development of a Modular Widget Toolkit

Master's Thesis

Graz University of Technology

Institute for Computer Graphics and Vision
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Graz, August 2014

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

The purpose of this work is to design and implement a GUI toolkit for the application framework "Murl Engine", a 3D and multimedia engine that focuses on cross-platform application development for mobile devices and desktops. A GUI toolkit provides common widgets and utilities for developers to implement graphical user interfaces in applications. Most of the available GUI toolkits are based on desktop environments and do not run in 3D context. They further require certain operating systems or third-party libraries. Therefore, a solution is required that is only based on the Murl Engine itself and that can be integrated into its scene graph structure. The first chapter introduces the engine and states the requirements for the toolkit. An evaluation of existing GUI toolkits is given in the second chapter. The results will be used to deduct a concept for the toolkit to implement, which will be introduced in the third chapter. The main aspects of implementing a GUI toolkit (which traditionally lies in the 2D domain) in a scene graph oriented 3D framework is the topic of chapter four. Finally, chapter five demonstrates example applications written with the toolkit, before chapter six concludes this thesis. The realization of this work has shown that the capabilities of the Murl Engine and its scene graph system provides great effort in implementing a state-of-the-art 2D GUI toolkit, with some limitations though. Another challenge was to bridge the gap between the event-based paradigm of common toolkits with tick-based polling used in 3D engines. The result covers most common features evaluated in other toolkit and provides a strong foundation for future extensions.

Contents

List of Figures

List of Tables

1 Introduction

The main objective of this thesis is the engineering of a graphical user interface (GUI) toolkit for an existing application framework and 3D engine. A GUI toolkit is a library for composing GUIs in applications. This first chapter explains the initial situation of the framework to extend and the company behind it, before giving a definition of what a GUI toolkit is (and what it is not) and which requirement it must fulfill. After that, the methods applied in this work will be described. Finally, an introduction to the framework is given to learn more about the basic concepts and ideas behind it.

1.1 Initial Situation

The *Murl*[1] *Engine* (ME) is a cross-platform framework for native applications driven by 3D graphics and multimedia content. Supported desktop platforms are Windows, Linux, and OS X. On mobile devices, Android and iOS are targeted, Windows Phone support is in development. Although running on both device classes, it is mainly optimized for mobile platforms. The engine is a product of the Austrian startup company Spraylight GmbH in Graz and is in development since 2011. The project itself is closed source, although many parts of the engine (e.g., the platform code, see section 1.4) are open. Source code licenses can be negotiated, otherwise free licenses with a fading banner and premium licenses are available for any developer. [Spraylight GmbH 2014f]

The goals of the ME are to provide a framework with high flexibility and transparency for developers to create high-performance programs for multiple platforms. The official website lists the following fields of application [Spraylight GmbH 2014a]:

- Indie Game Developers
- App Developers
- Visualization Components in the field of Medicine, Architecture, Simulation, Advertising etc.
- Research and Teaching
- Presentation Layer on Embedded Systems

A huge set of features is already covered by the engine, including 3D graphics, animation, physics, audio, networking, and others [Spraylight GmbH 2014b]. However, it completely lacks functionality for easily creating simple or complex GUIs,

[1] The word "Murl" is a colloquialism in Austrian slang, literally meaning "engine" or "motor". Source: `http://www.ostarrichi.org/buch-3274-15194-Mearle.html`

allowing users to interact with the application by sending the input to *widgets*. Widget is short for "window gadget" and is a general term for many kinds of interactive visible elements in an user interface.[2] GUIs are composed of widgets like buttons, sliders, checkboxes, menus, tables, lists, and many others. It is possible to handle user input from keyboards, mice, or touch devices with the ME. However, it requires some effort to build a complete slider or checkbox, and even more, if a whole form needs to be written. GUI toolkits aim to support developers with this task by providing a set of customizable widgets that work out of the box. GUI elements are nearly indispensable for all kinds of applications, even for games or scientific visualization programs. Therefore, the goal of this thesis is to provide a GUI toolkit for the ME, from here on called *Murl GUI Toolkit* (MGT). The term "GUI toolkit" is used ambiguously on the Internet. Therefore, the next section will clarify what the concrete tasks of a GUI toolkit are. Although this definition may not be universally applicable, it helps to set focus for the main topics covered in the following chapters.

1.2 Definition and Requirements

When evaluating existing GUI toolkits available on the internet, one might learn that many popular products not only contain GUI functionality, but also many other libraries used for software development. Those examples are more like application frameworks rather than GUI toolkits. Some toolkits do not contain certain features that may be seen as essential for GUI programming. Since there is no definition of official characteristics of a GUI toolkit, the following requirements and demarcations were determined for the MGT. A GUI toolkit, with special consideration of the ME, must meet these demands:

- The toolkit is an extension to an existing application framework (here: the ME). It is neither an application framework itself nor does the targeted engine depend on it.
- All common widgets shall be available for developers to build modern GUIs.
- The toolkit shall be capable of handling modern user input patterns, e.g., drag and drop, or multitouch.
- A transparent event handling system shall assist the toolkit users in handling input events or other actions.
- A flexible layout system shall be optionally available to automatically layout widgets in consideration of space and a given scheme.
- The appearance of both single widgets and complete GUIs shall be customizable. The process of defining the appearance is called "skinning" or "theming".
- An extension concept is required to allow programmers to easily extend the toolkit in case of missing features in particular situations.

[2] The word "widget" as it is used in this thesis will always refer to the term "GUI widget", unless otherwise noted. In general, "widget" may also refer to tiny pieces of software, that is driven by a widget environment, often included in operating systems. Other names for the latter are "gadget", "applet", or "desk accessory".

The following points are not part of the requirements, since there is no direct relation to GUI development, but rather to particular applications or special tasks. Furthermore, many of these features are provided by the ME, so there is another reason not to implement them:

- Providing a framework for an application,
- providing libraries for special tasks (e.g., scientific, productivity),
- handling and conversion of specific file formats, and
- granting access to platform or hardware features (e.g., threading, networking, file access).

1.3 Methods

The concept of the MGT is the result of evaluating existing GUI toolkits, analyzing their features, finding similarities and differences, both in implementation strategies and range of functionality. The first step was to create a list of toolkits that shall be considered for evaluation. As many possible candidates exist out there, some criterions were needed to reduce the samples to a small set of toolkits relevant to the MGT. The criterions are related to the characteristics of the ME and have been defined as follows:

- Based on OpenGL or any other 3D API or (game) engine,
- platform agnostic,
- modular and easy to use,
- consideration of mobile devices,
- being renowned for a particular concept, and
- support for declarative GUI definitions (e.g., via XML).

At least two of these criterions had to be fulfilled by a toolkit to be evaluated. Since there were still too many candidates left, toolkits without enough relevance (presence, popularity, and market share) were dropped, as well as candidates with an unacceptable lack of documentation or information available (e.g., “libnui”). As mentioned before, the actual set of features between the evaluated toolkits differ significantly in some aspects, but the requirements identified in the previous section can be seen as a common subset, as they are implemented (with some variations) in most examples. The final candidates have therefore been evaluated one after another regarding the following list of aspects:

- History and general information,
- supported platforms,
- programming language and used libraries,
- included widgets,
- event handling techniques,
- layout management, and
- skinning support.

The results will be presented in the next chapter. The following section will now give an introduction to the ME and its basic ideas. While the concept of the MGT (see chapter 3) was deducted from the evaluation results, the actual implementation of the toolkit (see chapter 4) strongly depends on the concepts and technologies behind the ME.

1.4 Murl Engine Architecture

Applications based on the ME are written in C++ and run therefore natively (i.e., executed by the CPU without any intermediate stages like an interpreter) on the supported platforms. This brings greater performance by avoiding the overhead produced by the primary programming languages and frameworks on each platform. Android apps are usually written in Java and need to be interpreted on runtime. Less but still noticeable overhead is also caused by Objective-C programs as they are used in iOS: Class methods are called by sending "messages" to objects, with the actual targeted method being unknown during compile time. This requires a dynamic lookup process on runtime.

To understand how the engine works, the different layers of its architecture will be introduced in the next section. The following sections will give an overview of some main concepts realized by the engine, particularly with regard to the features important for the MGT.

1.4.1 Engine Architecture Layers

Figure 1.1 depicts the layers of an application based on the Murl Engine. An explanation on each layer will be given below, going from bottom to top.

Native Applications

As the name already suggests, every platform has its own (native) application format, which bundles the app's final resources and binaries ready for execution. A cross-platform application is not a single executable that runs on multiple platforms, moreover it uses cross-platform sources and resources which are compiled and packed into the respective executable file format for each targeted platform. The main idea behind this solution is to write source code only once and just build it for the desired platform(s), whenever a debug or a release build is wanted. This requires the engine to abstract the common features available on all platforms.

Platform Abstraction

The platform code is unique for each target platform and must be included in applications that are contemplated to be built for that platform. The ME framework is shipped with project files for the supported platforms, which are set up to automatically include the appropriate platform code during the build process. Platform code serves two purposes:

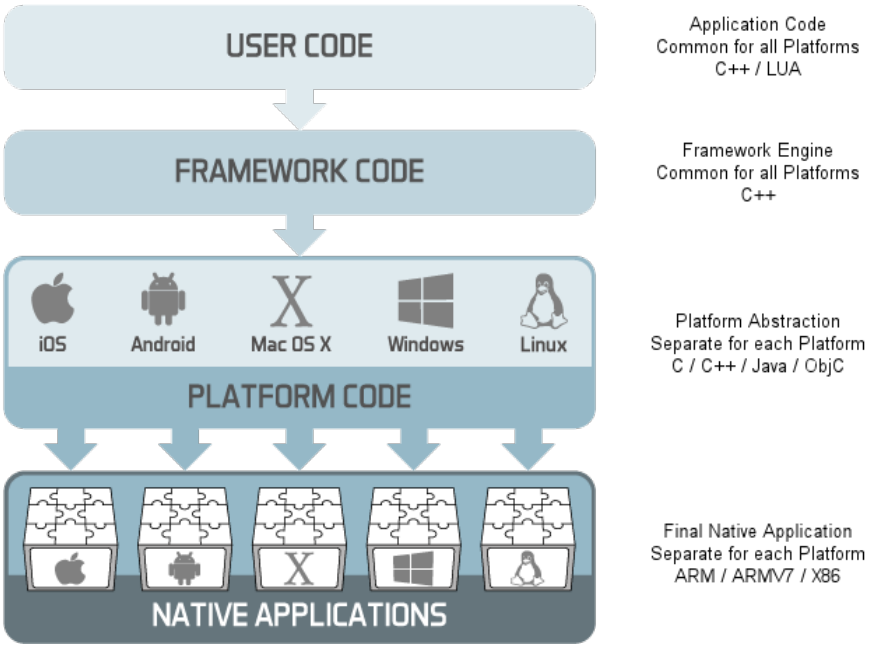

Figure 1.1: The layers of an application based on the Murl Engine

The user code written for an application resides on the top layer. The ME comprises the second and third layers. Building the project for a specific platform generates the native application bundle.

Source: `http://murlengine.com/usersguide/en/_user_guide_short_introduction.php`

- It wraps the user-level C++ code into the proprietary bundle format used on each platform to integrate the application into its app ecosystem. Hence, the platform code acts as entry point during the app launching process. This process differs from one operating system to another. After launch, the platform code yields execution to the engine. On Android, this is done by the Native Development Kit (NDK). For iOS builts, the Objective-C++ extension of the compiler front-end is used to bridge Objective-C and C++.
- The platform code is further responsible for abstracting the platform-dependent APIs, acting as a gateway between the engine and the operating system. A function provided by the engine (e.g., sending TCP packages) will probably use an API call different on each target. The user, however, just needs to call the engine function without any concern on which platform this line of code will be executed.

The platform code for every supported target is included in the framework and can be adopted by the developer to implement additional platform-level features. Nevertheless, this layer of the framework can be ignored, if no adaption is needed.

Framework Engine

The framework code is the core of the ME and built upon the platform code. It facilitates the main engine features for the app developer to use in her application.

It is entirely written in C++ and contains many functional packages (organized in namespaces). The classes of the following namespaces are of special interest for the implementation of the MGT:

- `Graph`: scene graph nodes and scene graph related classes
- `Resource`: classes for managing package resources (assets)
- `Logic`: classes for implementing the (tick-based) application logic
- `Input`: classes for handling input devices (including gyroscopes and location services)
- `Output`: classes for accessing device and system features (e.g., telephone, vibration motor, E-mail client, ...)
- `Util`: supporting classes and templates for easily realizing common programming tasks

Others are `App`, `Math`, `Audio`, `Core`, `Debug`, `Physics`, `Video`, `Net`, `System`, and `Platform`. The `App` namespace includes classes representing the app and are the entry point for the engine to execute user code. It is therefore necessary for the developer to attach her user code to this namespace by implementing a designated app class.

Application Code

The user code is written by the developer(s) of the application. They are responsible for providing an entry point for the engine to invoke logic execution after launch as well as for setting up the engine depending on their needs. Since the ME is tick-based, at least one *engine processor* (sometimes simply called *logic*) is required. The engine processor's interface defines an obligatory callback method that is invoked on each tick to update the logic state according to input events or other parameters. The app developer is responsible for writing the user code by relying on the features provided by the framework. There is no need to care about the platforms at this layer since the abstraction is done by the framework code. It is strongly advised against including and utilizing any other libraries than the framework itself.

This concludes the section about how source code is organized in the engine architecture. Beside the source files, most multimedia applications require additional assets to be presented as content or as interface. The way they are managed in the ME will be explained in the next section.

1.4.2 Resource Files and Packages

Assets are called *resources* in the terminology of the ME and refer to all kinds of files loaded and processed by the application on runtime. Resource files are extensively used in modern computer games, as most contents (images, sounds, models, etc.) created by artists require a huge amount of memory and are therefore loaded and freed on demand. Resources are bundled into *packages*. A package (identified by its file name extension `.murlpkg`) is a binary file which contains custom resources, optimized for efficiently loading them at runtime. Packages are generated by the

utility program `resource_packer` shipped with the ME. The program requires an input folder which contains a file `package.xml` for specifying all resource files (images, sounds, graphs, etc.) that are part of the package. An example is shown by listing 1.1. In the C++ code and the scene graph XML code resources are referred to by a package name prefix, followed by a colon and a custom identifier string (e.g., `assets:box_texture`).

```
1  <?xml version="1.0"?>
2
3  <Package id="assets">
4
5    <!-- Graph resources -->
6    <Resource id="main" fileName="graphs/main.xml"/>
7
8    <!-- Image resources -->
9    <Resource id="box_texture" fileName="textures/box.png"/>
10
11   <!-- Graph instances -->
12   <Instance graphResourceId="main"/>
13
14 </Package>
```

Listing 1.1: Definition of resource packages

Resources must be put in action somehow to appear on the output device. An image, for example, needs to be assigned to a texture that will be rendered to a geometry. Both texture and geometry are part of a *scene*, a term used in 3D modeling to address the union of all perceptible objects, the lighting setup, and the camera of a 3D world. Scenes are organized as scene graphs, which is a common concept used in 3D applications just as in the ME.

1.4.3 Scene Graphs

Scene graphs are directed acyclic graphs (DAG) used in object-oriented programming languages to organize objects of a 3D scene as *nodes* [Miaoulis and Plemenos 2009, p. 13]. Scene graphs first occurred 1989 as structure in Open Inventor (Silicon Graphics) to facilitate programming of OpenGL scenes by using object-oriented programming paradigms. In 1991, Silicon Graphics released IRIS Performer, another scene graph library, that focuses on performance rather than on ease of use [DeLoura 2001, p. 201]. Today, scene graphs are supported by many game engines (e.g. Panda3D[3], Unity[4], Ogre3D[5]), as well as in dedicated scene graph libraries (like OpenSceneGraph[6], Coin3D[7], OpenSG[8]).

One main feature of scene graphs is the inheritance of a node's transformation to its children, thus allowing geometric objects that belong together in a spatial sense

[3] `http://www.panda3d.org/`
[4] `http://unity3d.com/unity`
[5] `http://www.ogre3d.org/`
[6] `http://www.openscenegraph.org/`
[7] `http://www.coin3d.org/`
[8] `http://www.opensg.org/`

to be aligned with each other. For example, consider the 3D model of a motorbike to be translated to a specific position in the world. When attaching a character model to the vehicle, the character node will be inserted as child node of the object to inherit the transformation from it. The child node may now be transformed relatively to the parent's transform. In some engines, like in Unity, the scene graph structure does not serve any other purposes. However, scene graphs allow many more possibilities of scene manipulation to be realized. The extent of how much can be achieved solely by defining a scene graph structure depends on the actual framework that is used.

The ME offers a versatile collection of nodes which can be used to setup even complex scenes entirely in the scene graph XML resource file. This allows logic classes to be free from any lines of codes that creates or configures scene objects. On initialization, only references to nodes need to be set up. Figure 1.2 shows an example of how a scene graph may be set up in the ME.

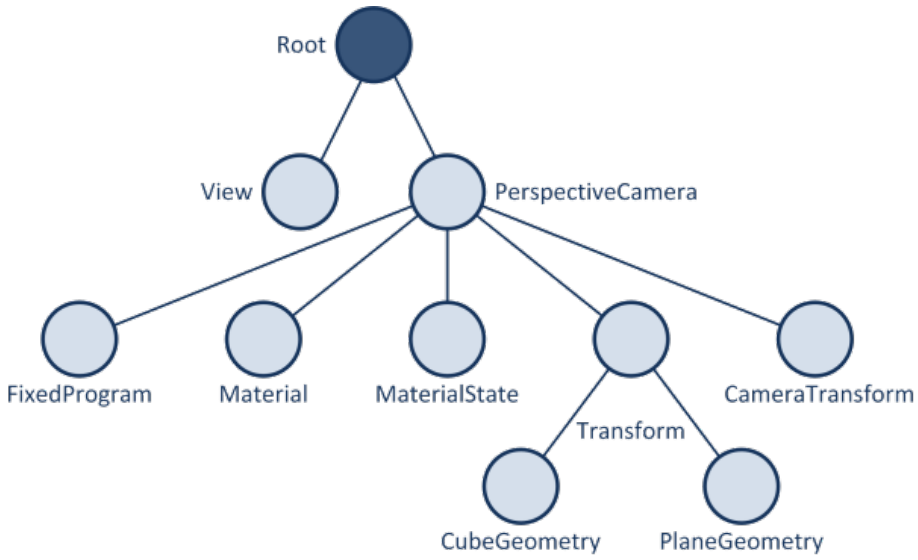

Figure 1.2: An example of a scene graph in the Murl Engine

Source: `http://murlengine.com/usersguide/en/_user_guide_short_introduction.php`

On each tick, the scene graph is entirely traversed by two different threads, *logic* and *output*, using depth-first traversal. The logic thread updates the internal state of all nodes while the output thread processes the presentation of the scene for the output devices. Output processing is done for the state one tick behind. So while one state will be drawn, the subsequent state will be processed. The ME provides some node types to control the traversal process in a smart way. The following list describes some of them, especially those interesting for the toolkit:

Instance: An Instance creates a parametrized copy of an dedicated sub-graph (defined in a separate file). It allows complex sub-graphs to be reused over and over again.

Reference: A Reference is a node referring to another already traversed target node to traverse the target again. This can be used to render geometry nodes more than once or to cause a state change triggered by the target.

Switch: A Switch has an index property to set exactly one (or none) immediate child node as active and visible, thus causing the other children to be ignored by the traverser.

Materials, parameters[9], and textures are implemented as nodes and are used for rendering geometry nodes. There are two ways to define the appearance of a geometry through materials, parameters, and texture nodes:

1. *Structural* by parenting the geometry nodes with one or more texture/parameters node and (at least) the material node.[10] This affects all descendant nodes until another texture/parameters node or material node gets inserted into the hierarchy.

2. *Temporal* by using material states and texture/parameters states. Those nodes refer to previously[11] defined material or texture nodes and activate them for the following nodes regardless even if there is no parental relationship. States can be overwritten by other states or by using strategy 1.

The example in figure 1.2 uses a `MaterialState` to render the `CubeGeometry` and the `PlaneGeometry`. The material state refers to the sibling `Material` node one index before. Neither of the two geometries is a descendant of the material state, but since the state node was traversed before and no other node did overwrite the material state, both geometries will be rendered according to the material definition. The same effect can be produced by attaching the geometries as children of `Material`.

1.4.4 Processors

Processors are classes that run the application logic. Similar to the scene graph, they can be structured hierarchically. On each tick, the `OnProcessTick()` method is called to execute custom application logic. Every portion of code that is part of the application logic (so virtually everything except the initial setup functions) will be held by a processor. Therefore, they play a vital role during runtime, as there needs to be at least one processor for applications to do something interesting. Processors are allowed to retrieve references ("node observers") to scene graph nodes, allowing them to query or modify referenced nodes.

[9] Parameters represent the OpenGL material lighting properties as described in Shreiner and Group 2009.

[10] Note that textures and parameters are not necessary at all to render geometry. For example, materials can be built from shaders without input data. However, the common practice followed by the MGT always uses textures or parameters in conjunction with materials, thus there will not be paid attention to this special case in this chapter.

[11] In the context of scene graphs, "previous" refers to a node that will be traversed chronologically before another node according to depth-first search.

2 Evaluation Of Existing Toolkits

2.1 Overview

An overview of all evaluated toolkits shall be given in this section. I will outline the major characteristics and state the reason for choosing a particular toolkit.

2.1.1 Android SDK

By the time of writing, Android reached a market share of about 80 percent in the mobile phone sector [Heisler 2013], making it obvious to take a closer look on the standard GUI toolkit of Android. GUIs are developed with the aid of the Android SDK, which is the foundation for Android app development. Android uses a proprietary GUI API, optimized for mobile devices, which is part of Dalvik VM, the Java implementation used on Android machines. Common Java GUI APIs like AWT, Swing, or SWT (see below) are not available in Dalvik VM. Furthermore, the Android GUI is only available on Android platforms and not suitable for cross-platform applications. Figure 2.1 shows some GUI-based applications typical for Android.

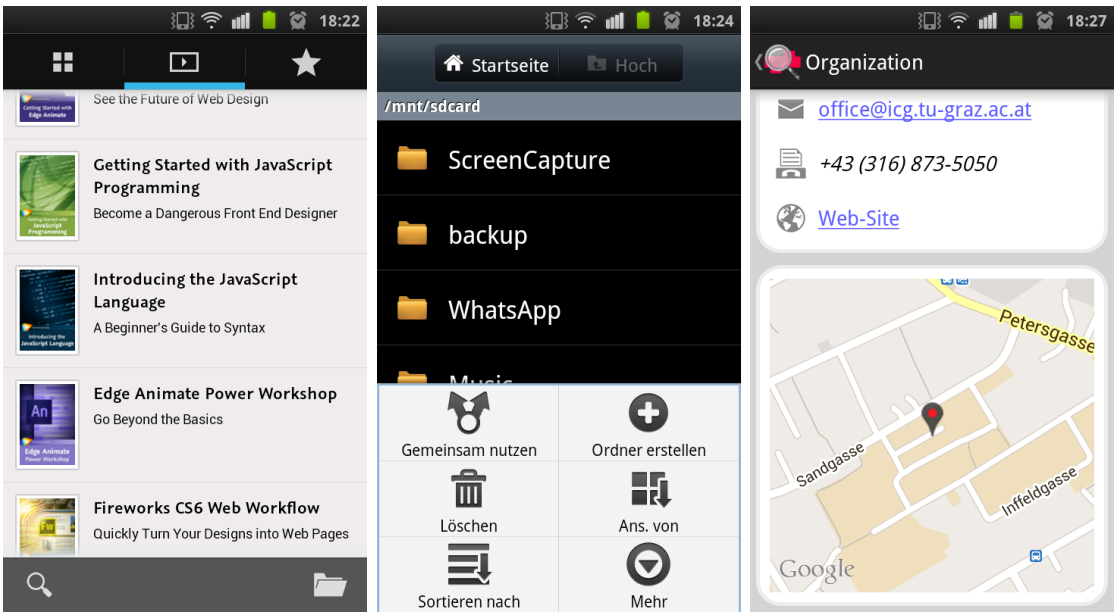

Figure 2.1: GUI of three Android apps

From left to right: "video2brain"[1], "Eigene Dateien", and "TU Graz Suche"[2].

2.1.2 Apache Pivot

Apache Pivot is self-classified as an "open-source platform for building installable Internet applications" [Apache Software Foundation 2014a]. Installable Internet applications (IIAs) are cross-platform Java-based applications deployed via web browsers. They differ from applets in how they are executed – IIAs can both run standalone and embedded in browsers. Based on Java, Pivot applications are platform-independent and just require a web browser with Java support. However, due to the lack of JRE support on the dominant mobile operating systems Android and iOS, Pivot is primarily used for desktop applications. Figure 2.2 shows an example of a GUI application embedded in a browser.

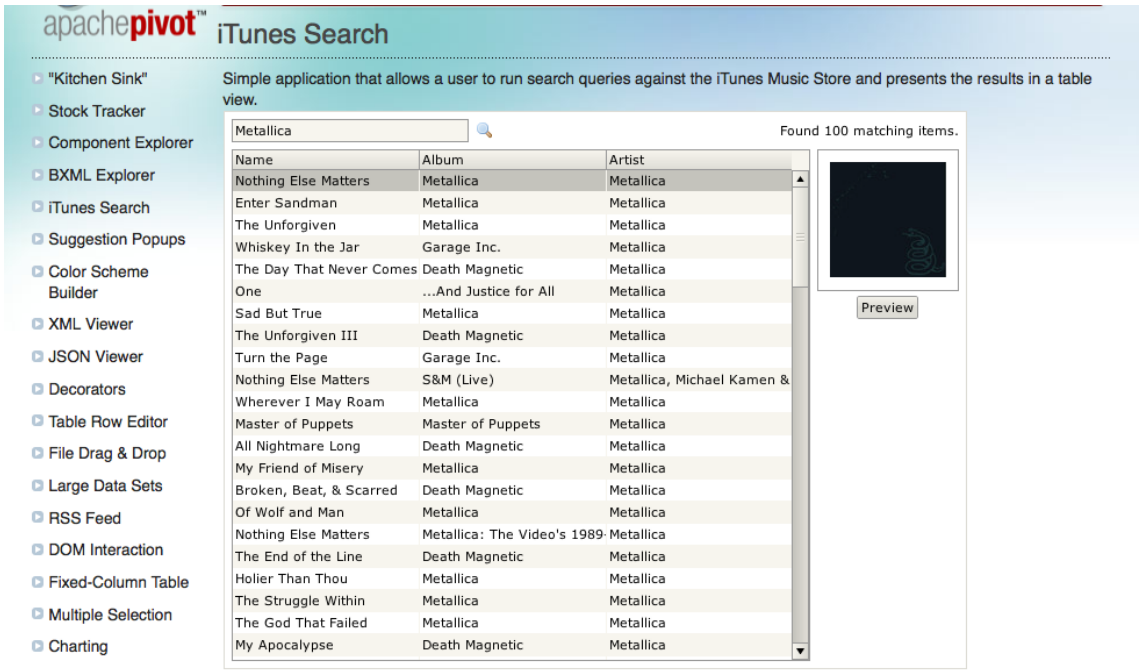

Figure 2.2: Apache Pivot Example

Source: `http://pivot.apache.org/demos/itunes-search.html`

2.1.3 Abstract Window Toolkit

The Abstract Window Toolkit (AWT) is part of the Java Foundation Classes (JFC) and abstracts the native GUI API of the respective operating system. It does not define or render widgets itself. This result is only a small set of features. AWT is almost outdated by now and is not supported on mobile platforms. However, it is still relevant as foundation for Swing (see below).

2.1.4 DirectGUI

DirectGUI is the name of GUI toolkit integrated in open-source game engine Panda-3D. Panda3D is a cross-platform framework written in C++, but with Python as primary language for developing applications. GUIs therefore need to be scripted like the whole application logic as well. The reason for evaluating DirectGUI is not the toolkit itself, but Panda3D. The framework, like the Murl Engine, is a platform-agnostic and scene graph oriented 3D game and multimedia engines based on OpenGL. Panda3D, however, is rather optimized for rapid application development (RAD) than for mobile platform distribution. Code is not compiled into an executable binary, but runs as script that is parsed on runtime. Figure 2.3 shows DirectGUI used in a game called "Epoch".

Figure 2.3: DirectGUI Example (Epoch)

Source: `http://www.panda3d.org`

2.1.5 Fast, Light Toolkit

Fast, Light Toolkit (FLTK) is a backronym that has its origins in Forms and the Forms Library (FL). Forms was a GUI toolkit for SGI machines that inspired the founder of FLTK to rewrite his own toolkit, later called FL, to base it on Forms. FL was then mainly used in Linux applications, which lead to the decision to drop the rendering code written for OpenGL and replace it with X. Though, support for rendering OpenGL scenes is still an included and promoted feature. FLTK runs on Windows, (Mac) OS X, and on common Unix and Linux derivates. There is no support for mobile devices. [Spitzak 2012b] The audio software "Giada" (see figure 2.4) is one example of software based on FLTK.

Figure 2.4: FLTK Example (Giada)

Source: `http://www.giadamusic.com`

2.1.6 FOX Toolkit

The *FOX Toolkit* (FOX) is a window-based widget toolkit written in C++ and available for desktop operating systems, especially Windows and many Unix/Linux derivates. FOX stands for "Free Objects for X" and uses the X Window System to render GUIs rather than wrapping native widgets. [Zijp 2013b] FOX also follows a similar philosophy as the Murl Engine in trying to "[e]liminate all platform specific header files" for easier portability. [Zijp 2013c] However, there is neither support for mobile platforms nor for OS X. An example application that uses FOX is depicted in figure 2.5.

2.1.7 GIMP Toolkit+

The *GIMP Toolkit+* (GTK+) was originally intended as GUI library for the GNU Image Manipulation Program (GIMP). It was then released in the late 90's as free widget library and became very popular on Linux platforms. Beside the GIMP (see figure 2.6), another famous example of software that is based on GTK+ is the Gnome desktop environment. GTK+ requires the GIMP Drawing Kit (GDK), an abstraction layer between GTK+ and the low-level windowing and drawing API of the operating system. [Clasen 2004] It has been ported for X Window System, Windows API, and Quartz, making it available on Linux, Windows, and OS X.

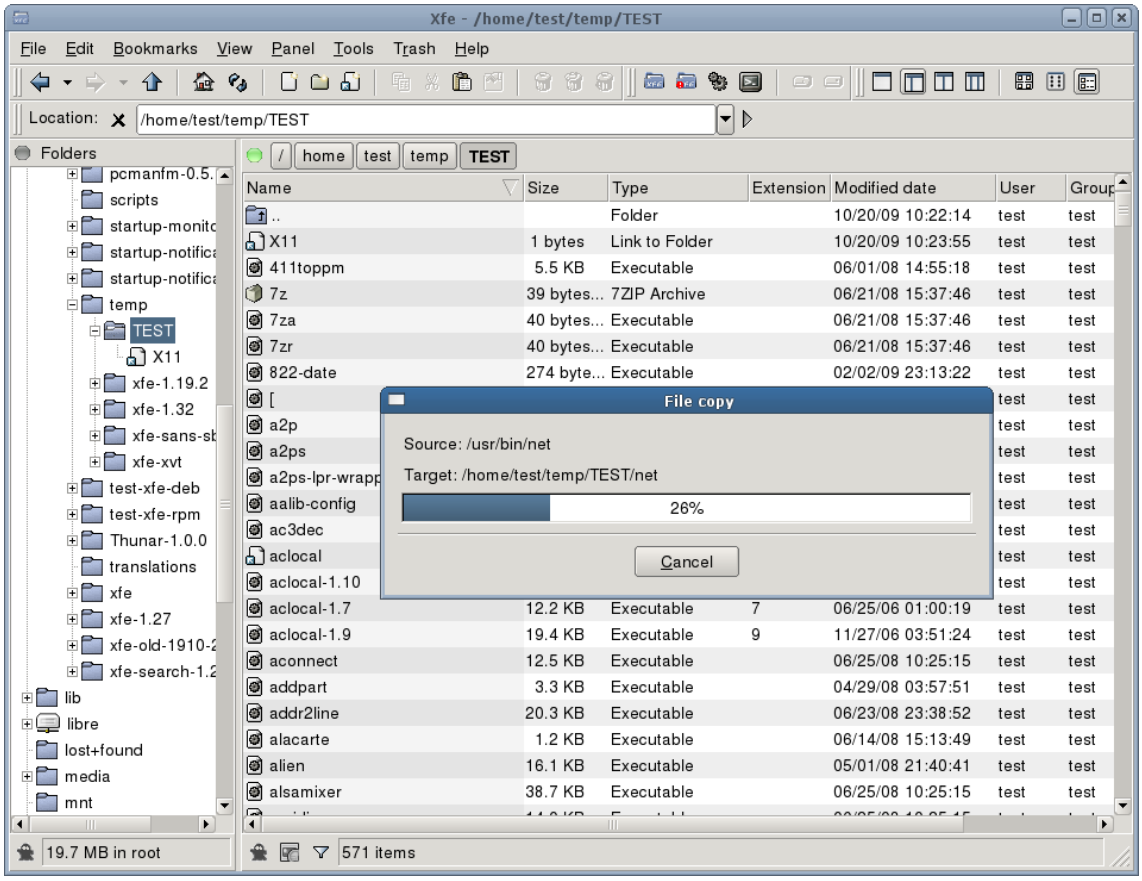

Figure 2.5: FOX Toolkit Example (Xfe)

Source: `http://roland65.free.fr/xfe`

2.1.8 JUCE

Jules' Utility Class Extensions (JUCE) is a cross-platform toolkit written in C++ that was first released in 2003. Beside GUI functionality and an OpenGL-based 2D engine, many other modules are included or optionally available. Ports exist for all major desktop platforms (Windows, Linux, OS X) and mobile operating systems (Android, iOS) and is thus one of the few evaluated toolkits that work fully on both device classes. [Raw Material Software Ltd. 2014a] Originally used for audio software, its predominant field of application still lies within this domain and is also commercially used by many notable manufacturers.

2.1.9 OpenGL User Interface Library

The *OpenGL User Interface Library* (GLUI) is light-weight GUI library based on OpenGL Utility Toolkit (GLUT) and therefore platform-independent, since the widgets are entirely rendered in OpenGL. GLUI only features some basic widgets and is not intended to use for complex GUI-based applications. The latest stable release has been on July 2006, so it is presumably not working on mobile devices.

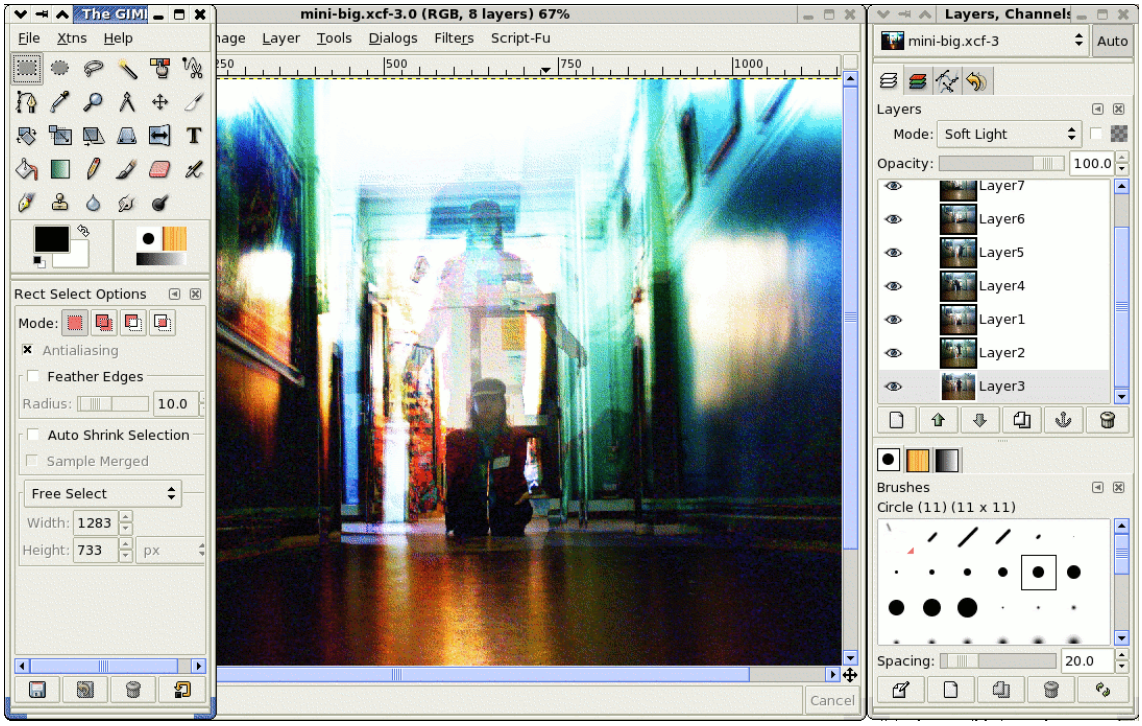

Figure 2.6: GTK+ Example (GIMP) [GPLv3]

Source: `http://commons.wikimedia.org`

[Rademacher, Stewart, and Baxter 2006] There is also no evidence that GLUI is compatible with OpenGL ES. However, it is used in Box2D, a cross-platform 2D game engine, for in-game physics testing during development. An example application is shown in figure reffig:evaluation-overview-glui. Because GLUI uses OpenGL for rendering, just like the ME, it will evaluated though.

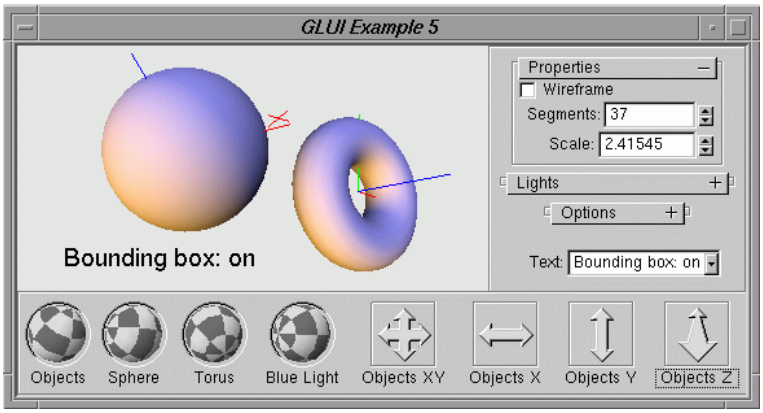

Figure 2.7: GLUI Example

Source: `http://glui.sourceforge.net/`

2.1.10 Qt

Qt is a widely used cross-platform library featuring both a GUI toolkit and libraries for other tasks. Its origins are in the early 1990s when it was initiated by a company that was later named Trolltech. Qt became the primary toolkit used in KDE (now KDE Software Compilation, see figure 2.8). This is comparable to the relation between GTK+ and Gnome, which is an important factor to the library's popularity. If available, Qt uses the native widget set of the platform, or its own rendering engine otherwise. One unique feature of Qt is its Meta Object Compiler (MOC), that is executed before compilation to expand some macros that are not supported by common C++ compilers. The macros add some new constructs to C++, e.g., the "signals and slots" concept, which is an easy-to-implement observer pattern. A newer feature of Qt is the declarative Qt Modeling Language (QML) for designing GUIs as hierarchical element trees. QML is the default modeling format on the upcoming Ubuntu Phone OS. On desktop environments, Qt has been ported for Windows, OS X, and all Linux/Unix variants that run the X Window System. It is also available in Maemo and S60, two mobile operating systems by Nokia. Support for other mobile platforms – iOS and Android – is in development and still experimental by now. [Qt Project Hosting 2014] Qt has been chosen for evaluation because of its wide cross-platform support (on both desktop and mobile devices) and its sophisticated features.

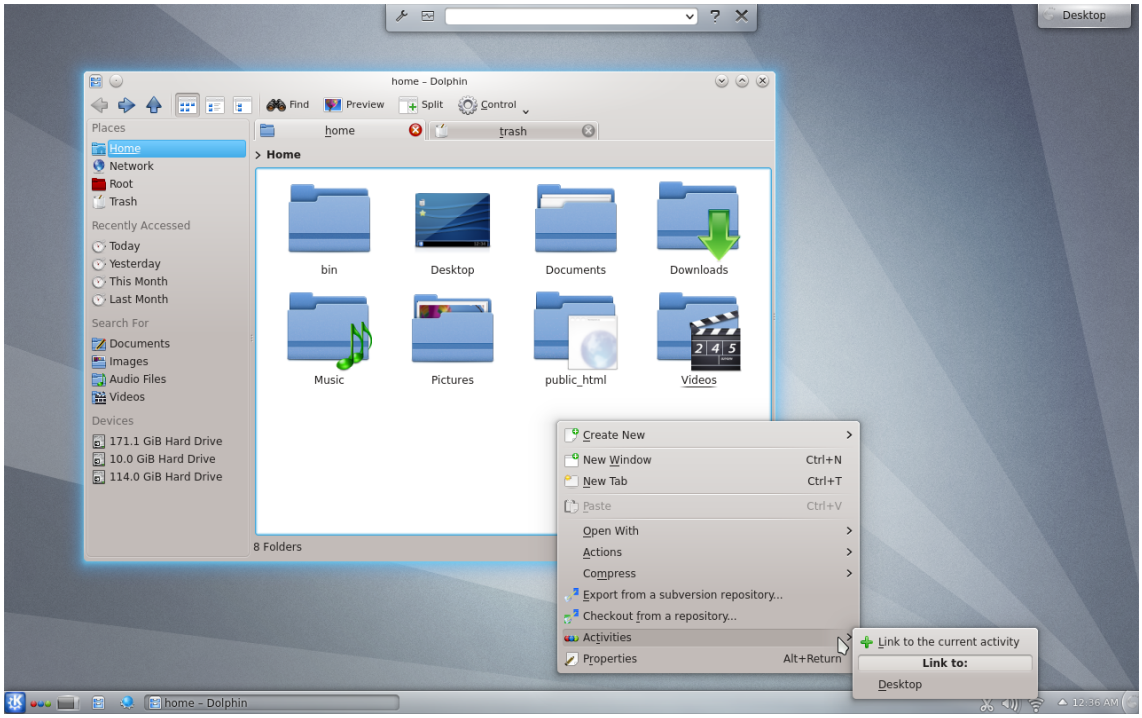

Figure 2.8: SWT Example (TuxGuitar) [GPLv3]

Source: `http://commons.wikimedia.org`

2.1.11 Standard Widget Toolkit

The *Standard Widget Toolkit* (SWT) is, similar to AWT, an abstraction of the native widget API on the respective platform. Nevertheless, SWT has always been seen as a competitor to Swing (see below). Its development was initiated in 2001 by IBM to provide a cross-platform widget toolkit for Eclipse, an IDE with a GUI also based on SWT. Widgets that are not supported by the target platform's API will be emulated, which improves portability, but reduces the overall performance on some systems. [Stanchfield 2012] On desktop machines, SWT is available for Windows (Windows API), Linux/Unix derivates (GTK+), and OS X (Cocoa), using the respective native GUI APIs. The only mobile platform supported is the already outdated Windows CE. See figure 2.9 for an example of SWT.

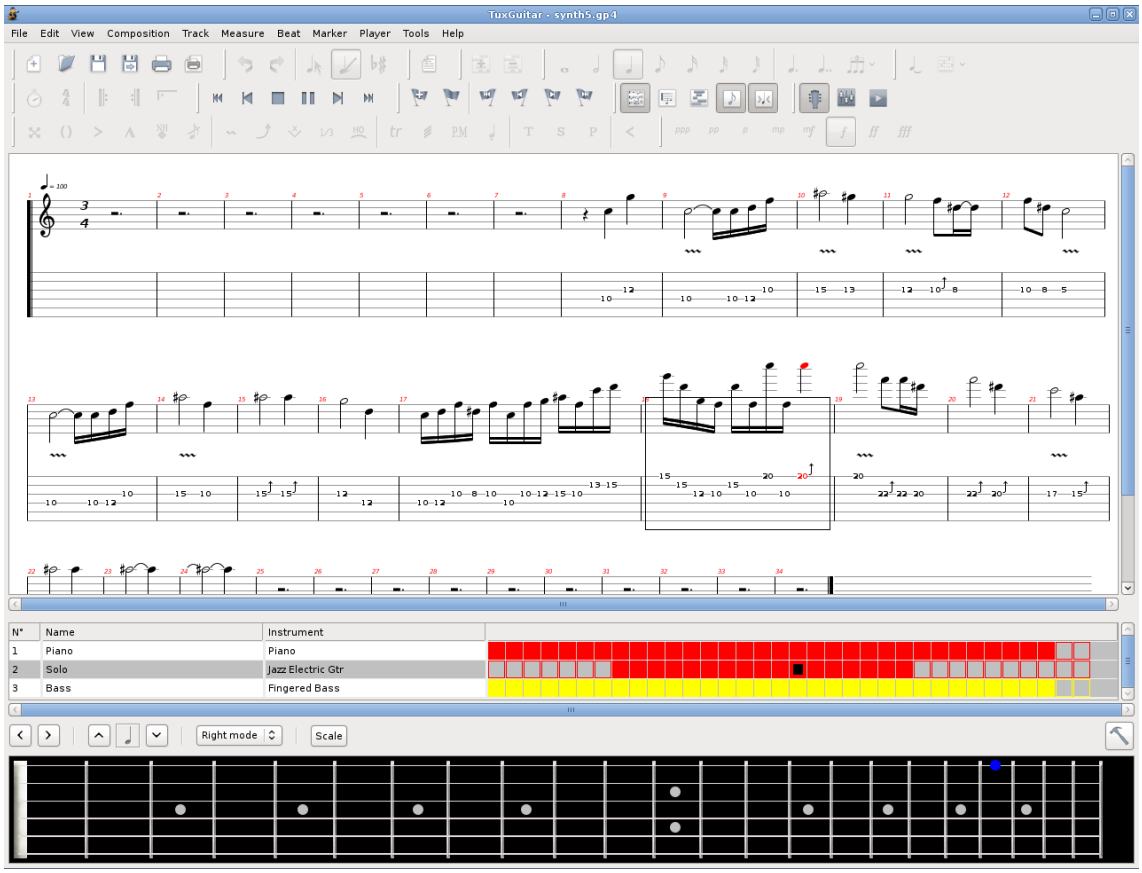

Figure 2.9: SWT Example (TuxGuitar) [FDLv12]

Source: `http://commons.wikimedia.org`

2.1.12 Swing

Swing is part of Java Standard and Enterprise Edition (J2SE and J2EE) since version 1.2 and is an alternative to AWT, but also based on it. Swing uses the AWT window class which wraps a native window object generated from the primary platform API.

All other Swing widgets, however, are drawn by the library itself into the window using Java 2D ("lightweight UI"). This results in better portability compared to AWT. [Niemeyer and Leuck 2013, S. 589–626] There is no official support of Swing in Java 2 Mobile Edition (J2ME), but a third-party library called Swing ME is available. Swing ME is also ported to Android (Android ME) and BlackBerry (BlackBerry ME). Layout managers are often associated with Swing, which is the reason why it is part of the evaluation. An example application is shown in figure 2.10.

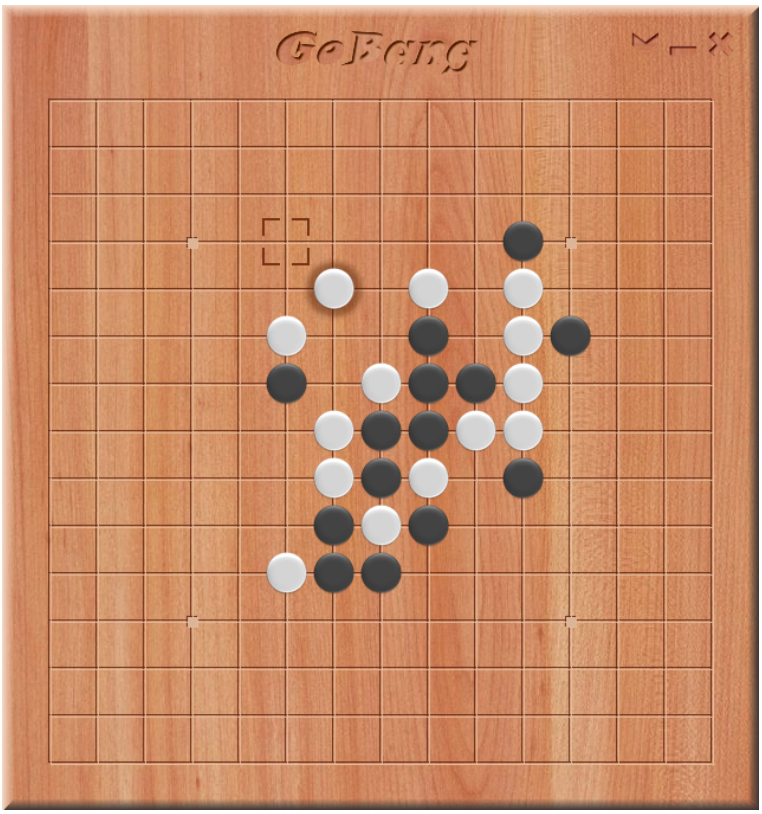

Figure 2.10: Swing Example [FDLv12]

Source: `http://commons.wikimedia.org`

2.1.13 UIKit

The main frameworks available for developers writing iOS applications is the fourth layer on Apple's iOS platform architecture – Cocoa Touch (see figure 2.11). Cocoa Touch is based on Cocoa, the related framework for Apple's desktop operating system OS X, and adopted to fulfill the requirements of mobile devices. The primary framework for GUI development upon Cocoa Touch is called *UIKit*. UIKit has been chosen for evaluation because the UI idioms brought by UIKit had an huge impact on the design principles of mobile GUIs for the last few years, for example by omit-

ting the touch screen stylus in favor of fingers and the reduction and enlargement of interface elements. [Apple Inc. 2013b] Example applications are shown figure 2.12.

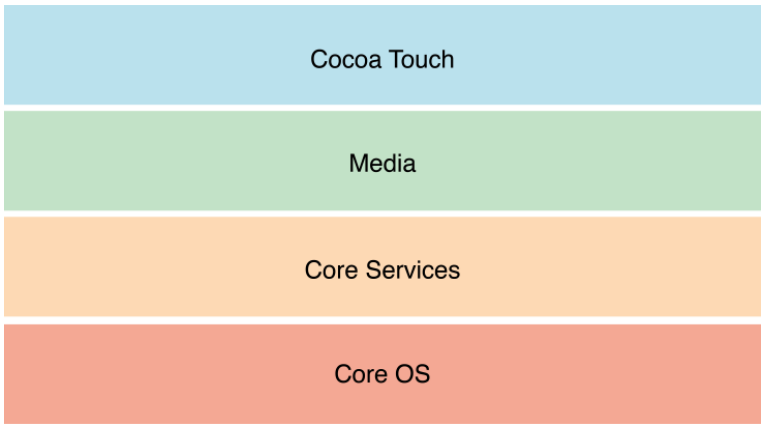

Figure 2.11: iOS Architecture Layers

Source: `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html`

Figure 2.12: iOS/UIKit Example Applications

2.1.14 UnityGUI

Unity3D is a widely used game engine and comes in conjunction with a full-scale project/asset management tool, a scene (graph) editor, and an IDE. Products can be built for all common desktop and mobile target platforms. The ME aims for a similar goal, as well as the Panda3D engine introduced above. Having a closer look on UnityGUI, the GUI toolkit of Unity3D, seems therefore reasonable. The

best known software realized with UnityGUI is probably the cross-platform Unity3D editor itself, which is shown in figure 2.13.

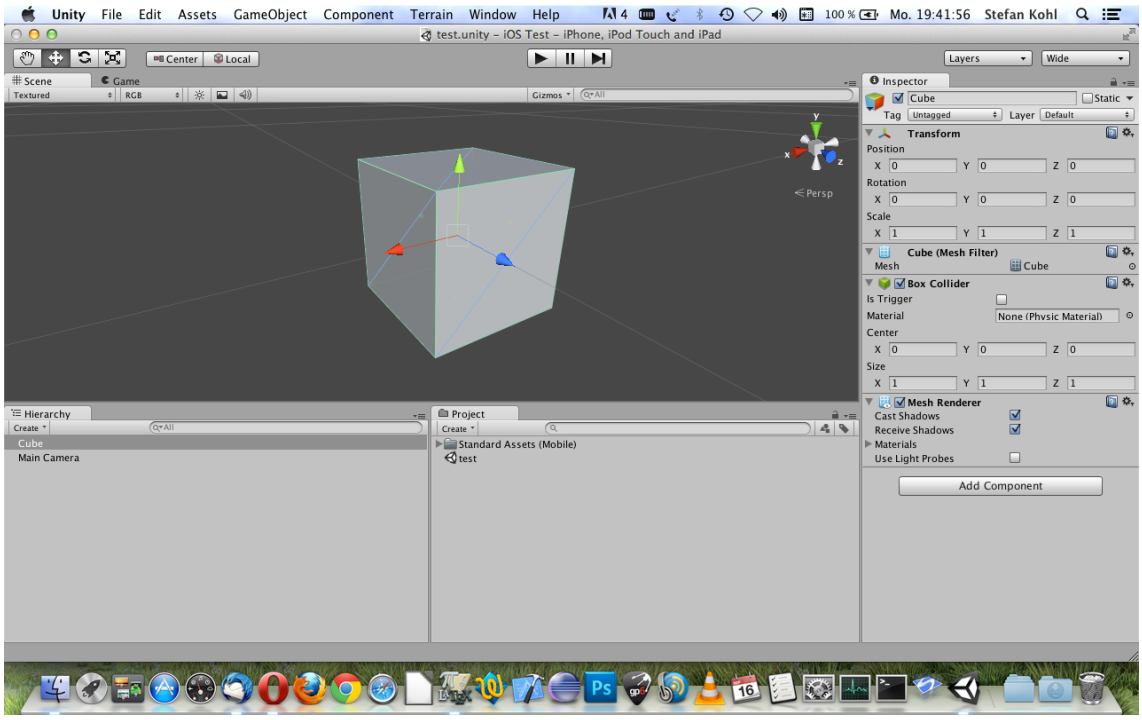

Figure 2.13: UnityGUI Example (Unity3D Editor)

2.1.15 Windows Presentation Foundation

The *Windows Presentation Foundation* (WPF) is a framework for both standalone and web-based applications depending on Microsoft .NET 3.0 or higher. The framework not only covers a widget toolkit, but several APIs for realizing multimedia, 2D and 3D graphics, text effects and data handling. If 3D-accelerated hardware is available, WPF uses DirectX for rendering the output of an application. Composing GUIs can be done either programmatically (as in most toolkits) or by describing them through an XML format called *Extensible Application Markup Language* (XAML). [Microsoft Corporation 2014a] Microsoft deploys WPF only for Windows and there are currently no intentions to implement WPF support in Mono. [Mono 2014]

2.1.16 wxWidgets

wxWidgets is a framework written in C++ that wraps around the native GUI API and makes code cross-platform ready. Ports are officially available for Windows, Linux, Unix derivates, OS X, and some mobile platforms like iOS, but not for Windows Phone and Android. The project was started in 1992 to support cross-platform development between Windows and Unix. [wxWidgets 2014b] According to the datasheets, wxWidgets also has an own set of widgets (“wxUniversal”) that

does not depend on an underlaying GUI API, e.g. for use in X11 [wxWidgets 2014d]. Many bindings exist for wxWidgets, making it possible to access the API with scripting language. Figure 2.14 shows an application that has been realized using wxWidgets. This toolkit has been evaluated because of its popularity and cross-platform support for both mobile and desktop platforms.

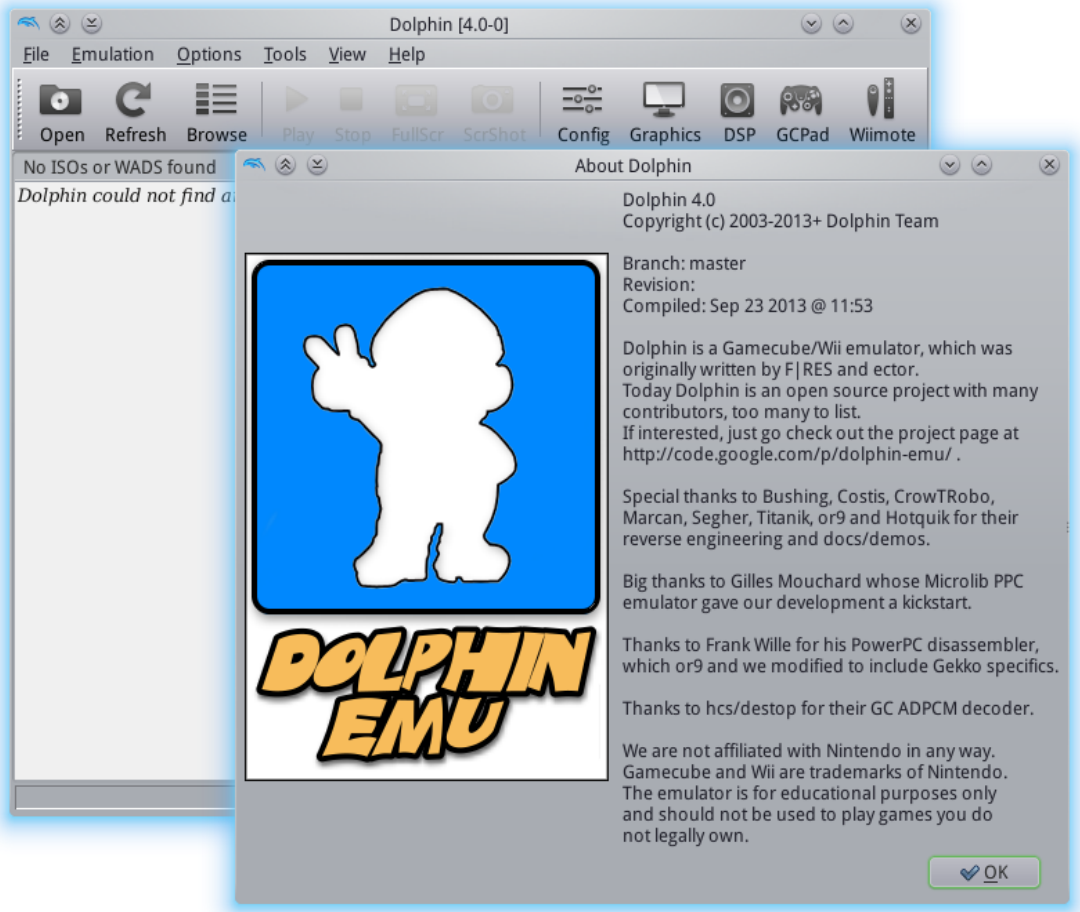

Figure 2.14: wxWidgets Example (Dolphin) [GPLv2]

Source: `http://commons.wikimedia.org`

2.2 Featured Widgets

Table 2.1 sums up the survey on available widgets among the evaluated toolkits. This chart is neither complete nor completely accurate. First of all, some exotic widgets only exist in single toolkits and do usually not occur in daily experience. Those have been dropped in order to provide a more general list. Furthermore, additional simplifications were necessary because there is no direct 1:1 correspondence between the implementations of particular widgets among the different toolkits. For

example, some toolkits provide simple one-line text field widgets and additional multi-line, scrollable text area implementations, while in other toolkits both are just configurable variations of one and the same widget type. Another example is the radio button, which may either occur as dedicated widget or as the result of configuring a more general option button widget. During evaluation, further problems occurred because of different terminology, inadequate documentation, or the lack of a comprehensive widget list like the "Component Explorer" of Apache Pivot (see figure 2.15).

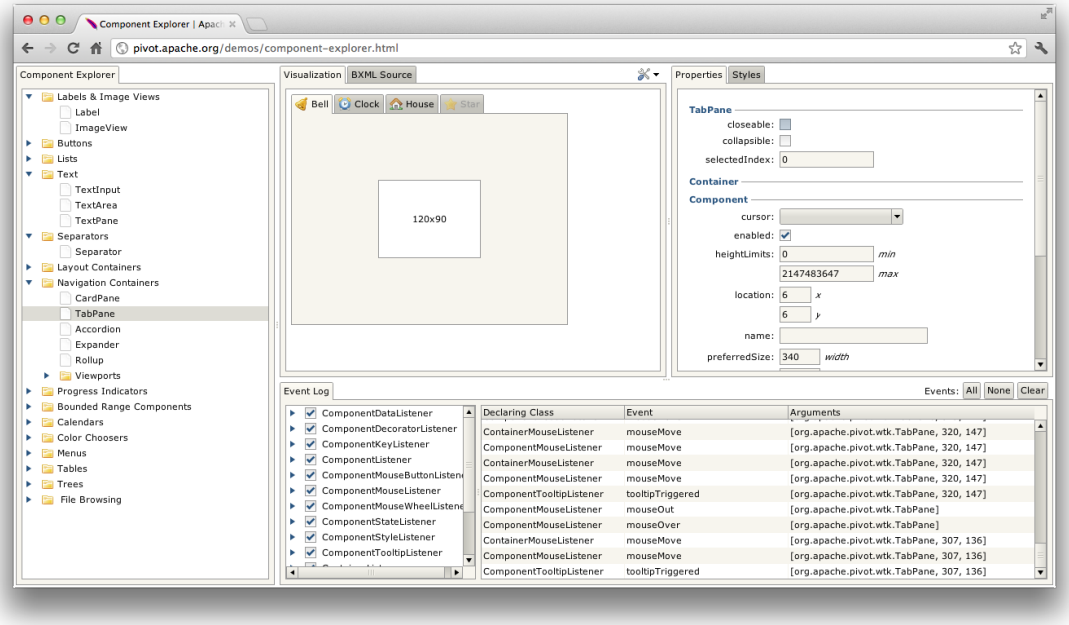

Figure 2.15: Apache Pivot demo: "Component Explorer"

Interpreting of the presented results led to the following assumptions:

- Buttons, radio/option buttons, textfields, labels, list views, and checkboxes (or its variation – the toggle switch) occur in all toolkits. They can be seen as the most basic widgets of a GUI. As as side node, they are also part of the HTML specification, which only features a few widgets.
- Another formerly basic widget, the combo box (also known as drop-down field), seemingly disappears on newer toolkits (e.g. UIKit) and on those which are part of a 3D engine (e.g. Unity3D). One reason might be the growing popularity of list views, driven by the design paradigm shift initiated by Apple.
- Sliders and progress bars are included in all recent toolkits (toolkits that have been introduced or updated during the last five years). Spinners are found in mobile GUI toolkits (where they make most sense) and in a few up to date examples for desktops.

		Android SDK	Apache Pivot	AWT	DirectGUI	FLTK	FOX	GTK+	JUCE	GLUI	Qt	Swing	SWT	UIKit	UnityGUI	WPF	wxWidgets
Input	Button	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Checkbox	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔
	Combo box	✔	✘	✘	✘	✔	✔	✔	✔	✘	✔	✔	✔	✘	✘	✔	✔
	Radio/Opt.	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Slider	✔	✔	✘	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
	Spinner	✔	✔	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✔	✘	✘	✘
	Stepper	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✔
	Switch	✔	✘	✘	✘	✔	✔	✔	✔	✘	✘	✔	✔	✔	✘	✘	✔
	Textfield	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Output	Image	✔	✔	✘	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
	Label	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Progressb.	✔	✔	✘	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✘	✔	✔
	Separator	✘	✔	✘	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	✘	✔	✔
	Statusbar	✔	✘	✘	✘	✔	✔	✔	✘	✘	✔	✘	✔	✔	✘	✔	✔
	Tooltip	✘	✔	✘	✘	✔	✔	✔	✔	✘	✔	✔	✔	✘	✘	✔	✔
Cmds.	Hyperlink	✔	✔	✘	✘	✘	✘	✔	✔	✘	✔	✘	✔	✔	✘	✔	✔
	Menubar	✔	✔	✔	✘	✔	✔	✔	✔	✘	✔	✔	✔	✔	✘	✔	✔
	Popupmenu	✘	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✘	✘	✔	✔
	Toolbar	✘	✔	✘	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
Data	List View	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Table View	✔	✔	✘	✘	✔	✔	✘	✔	✘	✔	✔	✔	✔	✘	✔	✔
	Tree View	✘	✔	✘	✘	✔	✔	✔	✔	✘	✔	✔	✔	✘	✔	✔	✔
Containers	Dialog	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
	Folding	✔	✔	✘	✘	✘	✔	✔	✘	✔	✘	✘	✔	✘	✘	✔	✔
	Grouping	✔	✘	✘	✘	✘	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✔
	Scrolling	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
	Splitting	✘	✔	✔	✘	✔	✔	✔	✔	✘	✔	✔	✘	✘	✘	✔	✔
	Tabbing	✔	✔	✘	✘	✔	✔	✔	✔	✘	✔	✔	✔	✔	✘	✔	✔
Special	Calendar	✔	✔	✘	✘	✘	✔	✔	✘	✘	✔	✔	✔	✔	✘	✔	✔
	Colorwheel	✘	✔	✘	✘	✔	✔	✔	✔	✘	✔	✔	✔	✘	✘	✘	✔
	Drag/Drop	✘	✔	✘	✘	✔	✔	✔	✔	✘	✔	✘	✘	✘	✘	✔	✔
	File dlg.	✘	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔

Table 2.1: Comparison of featured widgets

- Scrollable containers are supported by all toolkits except GLUI. In contrast to containers with folding, grouping, splitting, and tabbing accessories or capabilities, scrolling appears to be a quite common feature.
- Drag-and-Drop does not seem to be a must-have feature. Even Android SDK and UIKit do not have built-in functions for this task.

2.3 Rendering

There are several different ways on how to cast a rendered GUI onto the screen. The methods described in this section are a summary of those identified in the evaluated toolkits.

2.3.1 Using Native Widgets

When using the set of widgets provided by a native API, i.e., the default GUI API shipped with the operating system (e.g., Cocoa on OS X), toolkits provide an own library of widgets and features, usually represented as set of classes. However, theses classes are some kind of "wrappers", which produce native widgets by calling the interfaces of the provided API. This part of code is usually referred to as "platform code", and each port of a toolkit must define its own portion of platform code that instantiates and manages the native widgets. In the SWT, for example, the platform code consists of a set of Java classes doing JNI calls. These classes are used by the actual feature implementation classes and have the same signature on each platform, but different method bodies. [Northover 2001]

The method described here is primary used by the AWT, the SWT, wxWidgets, and by newer versions of Qt. wxWidgets uses 2D rendering as fallback, if the native GUI framework does not support a particular widget [wxWidgets 2014c]. The opposite is done by WPF, which usually orders DirectX to draw its own set of widgets: Some particular widgets, like the file dialog, are wrappers around Win32 API components.

2.3.2 Using 2D and 3D APIs

The most common method is drawing widgets with 2D or 3D graphic APIs. This leads to greater flexibility and easier portability, as only the engine that communicates with the graphics APIs needs to be ported. There are also drawbacks: Much effort must be put into the creation of resources and routines because all widgets must be built from scratch, normally by using graphic primitives. If someone also intents to provide the native look and feel of the targeted platform, even more work needs to be done. However, preferring the native look and feel as opposed to an universal GUI style is a matter of taste and in discussions it is mentioned both as advantage and disadvantage, depending on the point of view.

Different graphics APIs are used among the toolkits, strongly depending on the platform they are ported for. The Android SDK GUI uses a service called SurfaceFlinger for drawing. SurfaceFlinger, a display service developed by Google, internally calls native libraries based on OpenGL ES to fulfill its tasks. Swing and Apache Pivot render widgets with Java 2D. [Apache Software Foundation 2014d] Java 2D is part of the AWT, which has already been introduced as a toolkit that abstracts native widgets. DirectGUI and UnityGUI are using the graphic libraries of the game engines they belong to for rendering. GLUI is based on the OpenGL Utility Toolkit (GLUT). GTK+ has its own rendering framework, GDK, which abstracts the native 2D rendering API on the appropriate platform. Qt uses both 2D rendering and native widgets. When rendering the widgets by itself, Qt imitates the style of the operating system it runs on. This was once the only method for Qt to render widgets, while in newer versions, wrapping the native API is also supported. UIKit is built upon other Cocoa Touch frameworks provided on a lower level, most notably the 2D rendering framework Core Graphics, which uses OpenGL on a lower level. WPF widgets are rendered by a vector-based engine using DirectX [Microsoft Corporation 2014a].

2.3.3 Using X Window System

X Window System is an open specification of a "distributed, network-transparent, device independent, multitasking windowing and graphics system" [Pountain 1989]. The most popular implementation of this standard is X.Org (also called X11) and is available on all common Unix derivates and Linux distributions. X Window uses a client-server structure. The X server is the foundation of many GUI-based desktop environments and can be seen as a layer between the GUI (software) and the graphics hardware. The main tasks GUI clients are delegating to X server are windowing and 2D drawing. Windowing describes the whole pattern of having concurrent programs writing their output to distinct "windows", which are dedicated rectangle areas on the screen. X Window is also responsible for well-known idioms such as layering or mouse pointing.

The X Window System is also preferred by many GUI toolkits for its 2D capabilities and windowing features, most notably the FLTK and the FOX toolkit. GTK+, which already has been mentioned in the previous section, also uses the X Window System on its Unix/Linux ports, while in Windows and OS X drawing commands are put to the native drawing libraries. There also exists a port of wxWidgets called "wxX11" that uses the wxUniversal widget set for rendering GUIs with X11 [Smart et al. 2011].

2.4 Skinning

Skinning (also called "theming") is the process of customizing the appearance of single widgets or whole GUIs. There might be other definitions of when to talk about skinning or not, but for this thesis, skinning begins when changing the text color of a widget. Among all toolkits, most widgets, when represented as objects,

have properties that can be manipulated in order to produce a different appearance of the rendered widget. Some toolkits provide more flexible and cleaner tools for skinning a GUI, with a central skin definition that usually affects all widgets without explicitly assign properties to single objects. This section will sum up some common practices on skinning that were detected on the evaluated toolkits.

2.4.1 Skin Properties

Manipulating object properties is the easiest way to grant developers the opportunity for visual tuning. The number of available options varies, and there are different ways on how this is done. Assigning values to object properties is often used in terms of colors, fonts, or sizing. Manipulating the background plane of a widget is often more complicated, because of pre-rendered borders or other kind of ornaments. In the FOX toolkit, for example, skinning the entire theme requires subclassing of all needed widgets and overriding their `onPaint()` methods. This can also be done in UIKit, if manipulating the public properties is not sufficient, by overriding the `drawRect:` message of an `UIView` class. Other toolkits, like FLTK[3] or DirectGUI, try to provide as much flexibility as possible by exposing many properties. Due to their nature, both AWT and SWT are limited to the properties which were abstracted from the widget set of the low-level API.

2.4.2 Skin Classes

Skin classes are a centralized approach. There is usually one class or global object that is queried by widgets for skin properties. The properties are set by code statically or dynamically. In some cases, skin classes even play a greater role in rendering. In Pivot, for example, skinning is done by subclassing abstract skin classes and implementing the skin's rendering routine with Java 2D [Apache Software Foundation 2014d]. This is comparable to Swing, where skins are called *look and feel*s (LAF). Although there are usually more LAFs available to choose from, writing a custom LAF is a quite complex tasks, as there are many callbacks to implement. Another more restrictive possibility is the configuration of existing LAFs. [Oracle Corporation 2014a] Similar, but more transparent, is the skinning strategy in JUCE. A class called `LookAndFeel` exists, which can be instantiated and configured before being propagated globally as skin to use. It is further possible to extend this class and override it with custom drawing methods. [Raw Material Software Ltd. 2014b] In iOS 5, Apple introduced a complete set of features around their `UIAppearance` class to define so called "appearance proxies". Developers use them to configure the appearance of widgets programmatically once for the whole app. [Apple Inc. 2013c]

2.4.3 Skin Description Files

Some toolkits include resource files defined by the developer to gain information about the theme. This is comparable to the HTML/CSS technology used in web

[3]Beside property-based appearance configuration, it is also possible to switch between a selection of hardcoded skins in the current stable version 1.3.x. A new skinning system is planned for FLTK 2.0.

development, with content and structure being separated from style. In WPF, for example, a widget's visual property is defined either individually as XML attribute in XAML or as object property in source code or as dedicated style entity in XAML. Styles define the appearance of the widgets they get attached to. Furthermore, inheritance is also possible to extend existing styles. [Microsoft Corporation 2014c] The Android SDK follows a similar approach, but is even more inspired by HTML/CSS. The style (as it is called there) is defined as an XML resource file separated from the layout file. The Android API Guides show an example of how this may look like. [Android Developers 2014c] Since version 3 of GTK+, a new skinning engine is available that allows actual CSS files to be loaded as skins [Garnacho 2011]. CSS is also supported by GTK+'s competitor Qt [Qt Project Hosting 2013b]. Also Pivot, which supports skin classes as mentioned above, additionally has its own file format, which resembles CSS in many aspects, for defining the style of GUI [Apache Software Foundation 2014b]. Finally, in Unity, so called "GUI Skin" assets can be created to configure the skin of the GUI using the Unity editor [Unity Technologies 2014b].

None of these strategies were found in GLUI and wxWidgets, despite the fact that there may exist some external tools, which are not part of the framework.

2.5 Event Handling

Event handling is mandatory for all GUI toolkits. It enables the developer to specify the behavior of a GUI on user interaction. As an example, consider a piece of code that should be executed after the user clicked on a certain button. The assignment of the code to the button and the invocation of the code by the button are actually the tasks an event handling system is responsible for. Event handling is realized in many different way, and this section will sum up the most common practices discovered.

2.5.1 Polling

Polling is both used in hardware and software design and is defined by Gehani 1991, p. 94 as "repeated checking, to determine the occurrence of an event or to wait until a condition becomes true". In GUI programming, polling was superseded by message-based event handling systems.[4] Nonetheless, it still makes sense to use polling in particular cases, most likely in frameworks with tick-based execution callbacks. Unity is one example, and the UnityGUI is affected by it [Unity Technologies 2014a]. In Unity, a list of methods is invoked in a particular order within each frame cycle to handle the game's physics, logics, etc. A method called `OnGUI()` is both called for rendering and polling widgets.

[4] In literature, polling is not seen as an event handling strategy, but opposed to it. Since polling is used in one of the evaluated toolkits, it is mentioned here to compare it with others.

2.5.2 Callbacks

Callbacks are the simplest way of event handling: A widget provides methods to register function pointers or lambda functions (depending on the programming language and the toolkit) for being called by the widget itself after it discovered the occurrence of certain events. DirectGUI, for example, defines one primary "command" related to the basic usage pattern behind a particular widget (e.g., clicking a button, or pressing a key on text fields). When this certain event occurs, the callback is invoked. [Carnegie Mellon University 2010] The same is done in GLUI, where a single function can be registered to receive events from multiple sources, distinguished by an integral ID value [University of Alaska Fairbanks 2006]. FLTK goes one step further by allowing any number of callbacks to be added to (or removed from) a list, supporting more than one event handler for a single event [Spitzak 2012a]. UIKit uses an Objective-C construct called "selectors"[5] to also allow multiple methods being called for event handling [Apple Inc. 2013a].

2.5.3 Observers and Messages

Toolkits preferring this approach vary in their terminology and in many implementation details, but in fact it is derived from the *observer pattern* described by the "Gang of Four" (GoF) in Gamma et al. 1995. Another term for observer is "listener" or "target", and messages are also called "notifications" or just "events". Message-based event handling is an extended version of the callback strategy described above. Most notably, callbacks are replaced by objects of classes which implement dedicated interfaces. Different methods (specified by the interface) can be called on different event types (e.g., mouse click and move move). There is also an $n : m$ relation possible between observers and observables. This pattern is prevalent on Java-based GUI toolkits, where observers are called "event listeners" and are used in an event handling concept that shares many similarities among the evaluated examples, viz. Apache Pivot [Apache Software Foundation 2014c], the Android SDK [Android Developers 2014a], AWT/Swing[6], and the SWT [Eclipse contributors and others 2011] [Oracle Corporation 2014c]. Other toolkits using an object-oriented event handling system based on the observer pattern are FOX [Zijp 2013a], JUCE [Raw Material Software Ltd. 2014c], WPF [Microsoft Corporation 2014b], and wxWidgets [wxWidgets 2014a].

2.5.4 Signals and Slots

Signals and slots are found in GTK+ [The GNOME Project 2014a] and Qt [Qt Project Hosting 2013c] and are comparable to callbacks and observers. What makes signals and slots special is their realization by meta-programming C++. The commands used to connect signals with slots are not C++ compliant but macros that need to be resolved by a *meta object compiler* (MOC). Roughly speaking, connecting signals with slots (both are function callbacks) results in the slot callback being invoked after the signal function has been called. Signals and slots are said to be

[5] Although technically not the same as callbacks, selectors are used similarly.
[6] Note that Swing uses the event handling architecture of AWT.

more flexible and safer than callbacks and require less code than observer-related approaches, resulting in an improved code readability. [Thompson 2013]

2.6 Layout Control

This section will cover a general description on common layout managers that appear in most toolkits. They often differ in name and in how many aspects a layout may be configured. Before summarizing on the common subset identified, another question has to be answered: How are layouts actually implemented? The following approaches have been detected.

2.6.1 Layouts as Widgets

Layout managers are derived from widget base classes or vice versa. The widgets to layout are attached as child components to the layout manager. This done by FOX. In Pivot, layout managers are container subclasses and thus part of Pivot's component-container hierarchy [Apache Software Foundation 2014e]. Layouts as subclasses of container widgets are also found in the Android SDK [Android Developers 2014b], JUCE, GTK+ [The GNOME Project 2014b], and WPF [Microsoft Corporation 2014a].

2.6.2 Layouts as Controllers

Layouts are instances of layout classes (based on a common interface) attached to widgets. In Qt the layout handling interface is part of the base widget class [Qt Project Hosting 2013a]. In other cases, especially if the toolkit defines explicit container classes, layouts are most likely passed to them. This is done in AWT, Swing (both using the same layout managers) [Oracle Corporation 2014b], SWT [MacLeod 2009], wxWidgets [Victor 2014], and also in the UIKit, which allows custom and built-in layout managers to be assigned to its `UICollectionView` [Apple Inc. 2014] widget. The layout system of UnityGUI belongs to this category too, although it works a bit different due to the procedural GUI setup on each tick [Unity Technologies 2014c].

2.6.3 Layouts as Macros

This strategy, used by FLTK, is unusual and depends on the graphical GUI editor tool "Fluid" [Spitzak 2012c]. With Fluid, the developer can compose the GUI and set up constraints for widgets on how to behave on container size updates. The GUI is then saved in a proprietary file format and converted to C++ source files for compilation. On runtime, the generated code will manage the layout according to the properties set in Fluid.

No strategies for layout control were found in DirectGUI and GLUI.

2.7 Summary

Many toolkits exceed their main purpose as a GUI library and offer additional components for frequent tasks. This includes file handling, database access, XML parsing, scripting support, etc. However, the MGT is an extension of an existing multimedia engine, which is supposed to already have most of these components on board. The toolkit will therefore focus on the main tasks of GUI development. Their details will be discussed in the next chapter. A final discussion on the most important aspects and strategies learned will now conclude the evaluation.

Featured Widgets: There exist many different type of widgets and even more ways how they have been realized. However, the importance of a particular widget and its "mechanics" can be estimated by finding commonalities among all candidates. For the MGT, it makes sense to implement basic widgets that are assumed to be provided by a toolkit. It is also not the goal of the MGT to compete with the market leaders in terms of feature richness, but to demonstrate on how to realize a GUI toolkit with a scene graph based graphic engine.

Rendering: The evaluated toolkits revealed three approaches of how to present widgets. Although all three can be used for cross-platform development, the highest grade of portability is facilitated by using 2D/3D rendering APIs or libraries. X Window Server is not available on all platforms and particular native widgets differ too much from platform to platform. Sticking with graphics APIs is even more emphasized by the fact that the ME uses OpenGL to display content. Additionally, by using the ME preferably for mobile games and multimedia apps, native widgets with their default look and feel are even less attractive for artists who are responsible for the GUI design.

Skinning: Three skinning philosophies were identified during evaluation. Decentralized skinning and skin classes are both done in user code, where the first approach only affects single widgets and the second one sets global rules. Alternatively, skin description files also have global effect, but in comparison to skin classes, they are defined by special file formats, which makes them easier to maintain and to exchange. The third method is most preferable for the MGT, since the ME has a very flexible resource handling system and a versatile scene graph XML format.

Event Handling: Event handling has been identified as a variant of the observer pattern. Depending on the language, the notification of the observer is done by callbacks (function pointers) or interface implementations (methods). This also seems reasonable for the MGT. The signal-slot concept operates on a higher level and requires the integration of a MOC into the toolchain of the ME. This would violate the requirement that the toolkit may depend on the engine, but not vice versa. Furthermore, the efforts of writing clean and easily readable standardized C++ code (as proposed by the ME) would also be affected.

Layout Control: Similar to widgets, layout managers exist in many varieties, sharing some resemblances across different toolkits. Some samples only differ in name or customization properties. The more interesting part, however, is how layouts are integrated in an existing hierarchy of widgets. Extending the container class is one option, but writing separate layout controllers is a more lightweight solution, since layouts primarily perform some spacing arithmetics on widgets. Another objection to layout widgets is the fact that containers are also base classes of richer containers with particular functions or accessories (e.g. scroll containers). A controller can just be attached to any container subclass, while layout containers must be embedded into extended containers.

The next chapter will present the concepts of the MGT that are based on the knowledge gained during the evaluation phase and adopted to the demands of the ME.

3 Design

This chapter will cover the main concepts of MGT. Based on the evaluation results from the previous chapter, the MGT attempts to cover the basic features and subsystems of modern GUI toolkits. The strategies behind the implementations are discussed here. Which strategy actually was preferred for a feature strongly depends on the usefulness of the feature within the primary field of application (i.e., multimedia applications on mobile platforms) and on the capabilities of the ME. As said before, the MGT is built upon the ME and the third-party libraries included in the engine. Hence the toolkit cannot be used with any other engine or standalone.

3.1 Toolkit Architecture

The implementation of the MGT entirely resides on the level of the user code layer introduced in section 1.4.1. It is therefore connected to the engine only through the public interfaces. This is sufficient for implementing all features covered in this chapter. Moreover, a toolkit written on the user code layer is more modular, since it does neither penetrate the underlaying engine nor does it create unnecessary dependencies. This is also one of the requirements the MGT must fulfill.

3.1.1 Basic Structure

Figure 3.1 shows a component diagram representing the distinct parts and their connection. The architecture is mainly flat, even if the diagram may suggest the opposite. However, the outlined, third level is the actual implementation of the toolkit, while the upper level consists of public interfaces, which are derived from some base interface classes. Those make up the basic entity types that are part of the toolkit. The user of the toolkit only accesses the instantiated entities (i.e., objects) through their public interfaces, after they have been created by a factory. The source code of the implementation will not be shipped by the distribution of the engine to developers with basic licenses. This strategy is also used by the ME itself, following the *abstract factory* design pattern drafted by the Gang of Four. The next section will give an overview of the different kinds of classes implemented in the MGT. A detailed explanation of the concept behind will be given in the subsequent sections. There are, of course, some more classes and functions available in the toolkit, which were not taken into account by the component diagram in figure 3.1 because they are merely used as utilities to loosely support other classes. A closer look on the utilities will be given in chapter 4, Implementation.

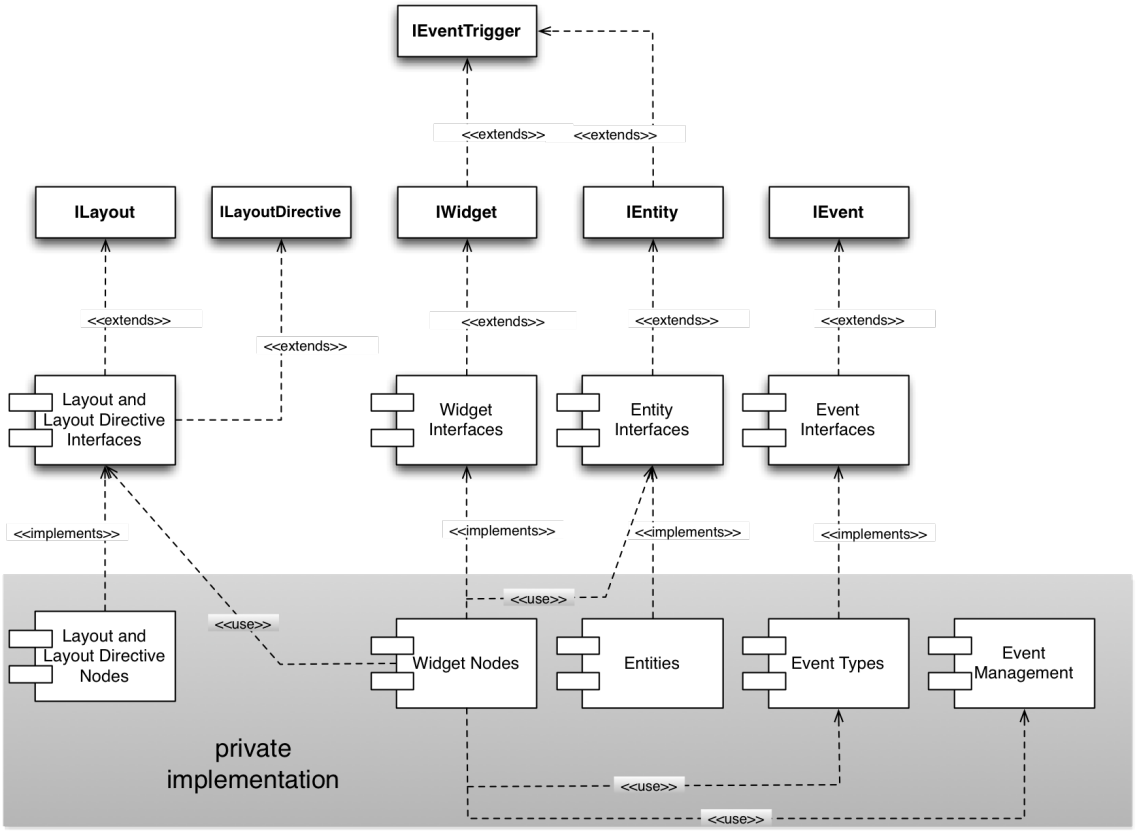

Figure 3.1: Component diagram of the MGT

3.1.2 Toolkit Object Types

Events

The *Event* types are implementations of `IEvent` and describe objects holding data about events occurred on *Event Triggers* (i.e., instances of `IEventTrigger`), at which one object represents one event. Events[1] are created by Event Triggers and dispatched to Event Handlers by a singleton Event Pipeline, which realizes an observer pattern. This process will be described in section 3.2.4. For now, it is sufficient to know that there is an Event class for each type of event the MGT is able to handle. This is shown in the class diagram of figure 3.2. The abstract base class `Event` implements the interface `IEvent` and provides access to the object that triggered the Event. All derived classes add further information depending on the kind of the event occurred. There is no other purpose of Events beyond their informational use case. Event class implementations are not available directly. Instead, the interfaces of the various Event types can be queried. Details on the different Event types will be provided in section 3.2.2.

[1]When using the term *Event* (uppercase), I refer to the implementation of event description objects in the MGT. The term *event* (lowercase) is used when discussing events in a more general (toolkit-independent) context.

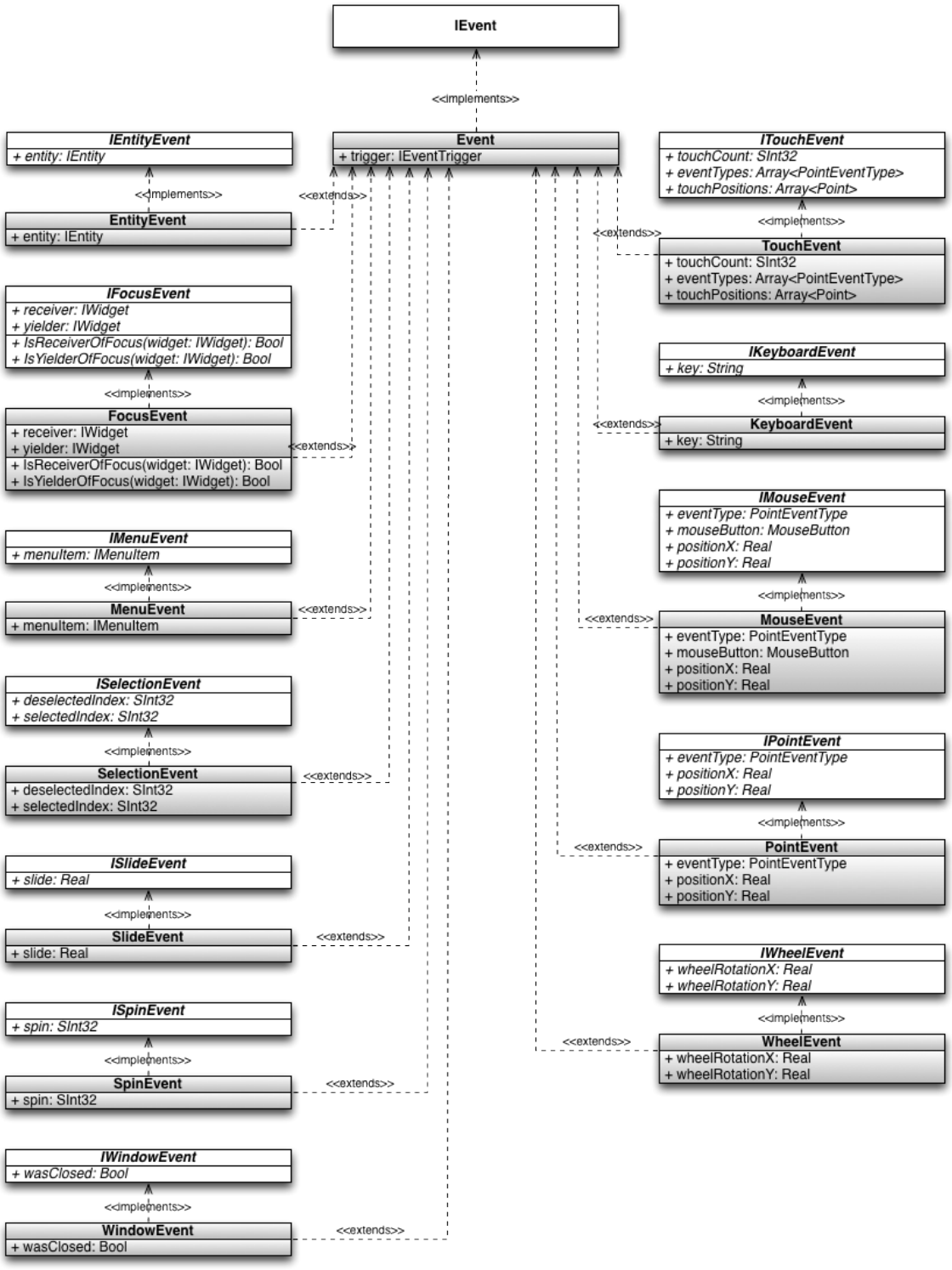

Figure 3.2: Event classes and interfaces

`Event` is an abstract class that provides access to the Event Trigger. This class as well as any other implementation of the Event type is hidden and not accessible from user code (gray area).

Entities

An *Entity* is an object for wrapping and managing data that is processed by GUIs. The base class `Entity` implements the `IEntity` interface which is a subclass of `IEventTrigger`. Hence, Entities can be identified as the origin of certain types of Events, namely `EntityEvent`s. An Entity fires an Event, whenever the data it holds has been changed. In general, there is no specification about the data structure below an Entity, so an object conforming to `IEntity` can hold any possible data. However, there are some Entity subclasses available to satisfy basic data handling for some controls included in the toolkit. The Entity inheritance tree is depicted in figure 3.3. The abstract factory pattern is utilized here. Accessing an Entity is therefore only possible through its public interface. A further explanation on Entities will follow in section 3.3.1. There will also be an outline on the purpose of some special Entity classes.

Widgets

The base interface of all widgets of the MGT is called `IWidget`. Like the `IEntity` interface, `IWidget` inherits from `IEventTrigger`. A Widget[2] is a renderable geometry that is able to receive and handle user input events as well as to create events itself. Figure 3.4 shows a class diagram which depicts the immediate inheritance of the `IWidget` interface.[3] Subclassing `IEventTrigger` allows a Widget to create an Event, or – more precisely – allowing a Widget to identify itself as the Event Trigger of an Event.

Most of the Widget interfaces are subclasses of `IComponent` (as seen in figures 3.5 and 3.7), whereas the other Widget interfaces `IMenu` and `IMenuStrip` are on their own. A *Component* simply refers to a Widget that is described as rectangle by its position and dimension. Moreover, Components can be put into a *Container*, that is another Component which implements the `IContainer` interface. A Container inherits the properties of a Component and can therefore be used to build an UI with hierarchically structured Components. It may be clear now why `IMenu` and `IMenuStrip` are not Components at all: Their intent as menus in a GUI (main menus, context menus, popup menus[4], etc.) is not compatible with the idea of laying out Widgets hierarchically. Figure 3.5 shows Containers and simple render-only Components.

The MGT realization of the Drag-and-Drop concept is linked to Components and Containers, so the involved classes shall be introduced here, although they are no Widgets. Components can be dropped into Containers, when using both in

[2] When using the term *Widget* (uppercase), I refer to the implementation of widgets in the MGT. The term *widget* (lowercase) is used when discussing widgets in a more general (toolkit-independent) context.

[3] Note: For reasons of brevity, the class diagram does not include the getter and setter methods. They are just listed as public properties. Since there are no actual public properties used in the classes and interfaces, all public properties in the class diagram may be interpreted as property accessors/mutators. There are also no protected and private properties since their use is implementation specific and not relevant for the concept itself.

[4] One popular example of a popup menu is the `UIActionSheet` included in Cocoa Touch.

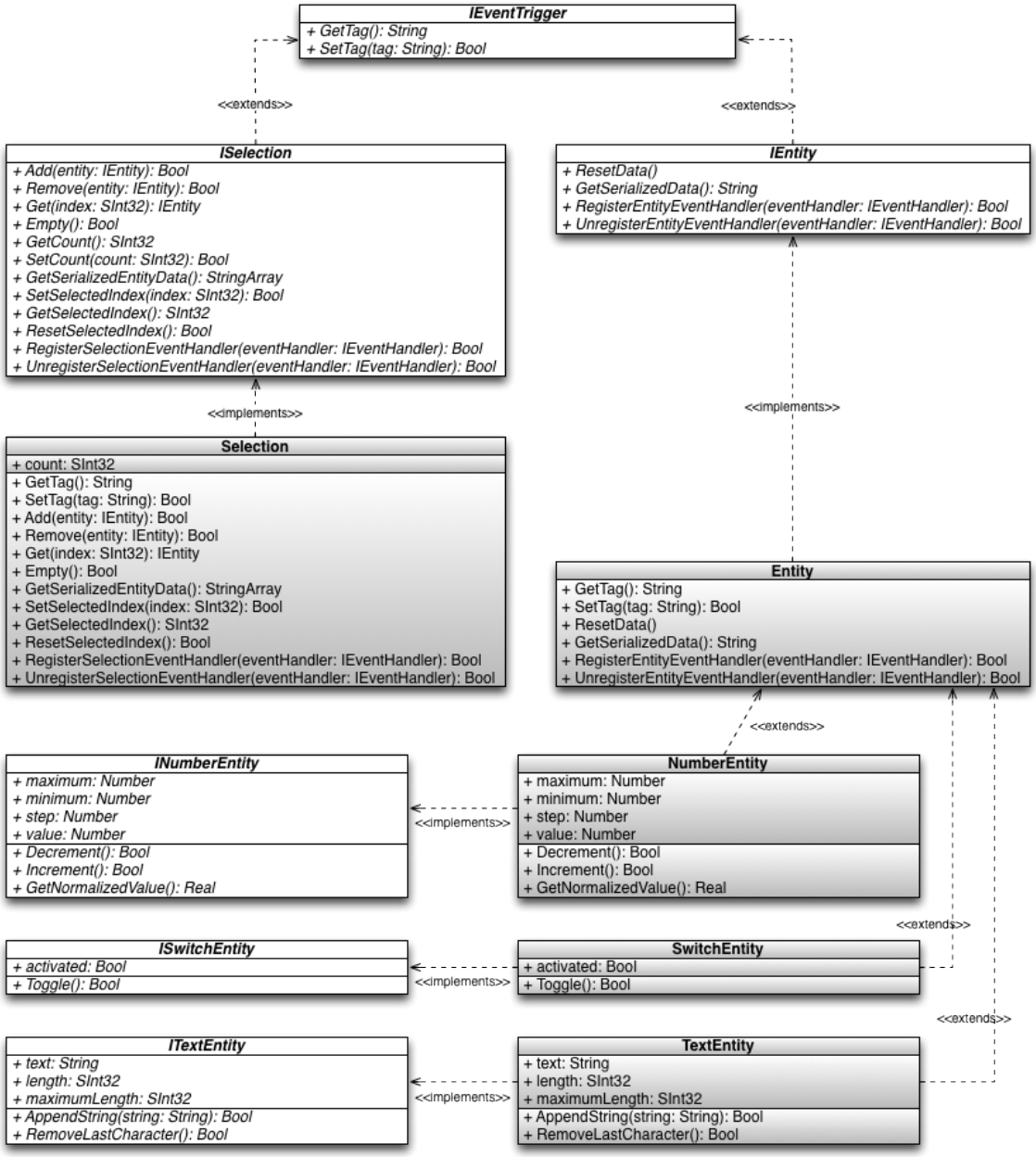

Figure 3.3: Entity classes and interfaces

The gray area marks the classes (all implementations) that are not visible to the developer. Due to the abstract factory pattern, objects can only be accessed through their public interfaces.

combination with the same *Drag Drop Family* Node instance, which handles the logic behind Drag-and-Drop. A Drag Drop Family is an extension of `Node` and has its separate class hierarchy as shown in figure 3.6

There exists a special group of Widgets within the Component inheritance tree. They are called *Controls* and are vital elements to make a GUI capable of receiving user interaction. A button is the most important control of a toolkit, since trig-

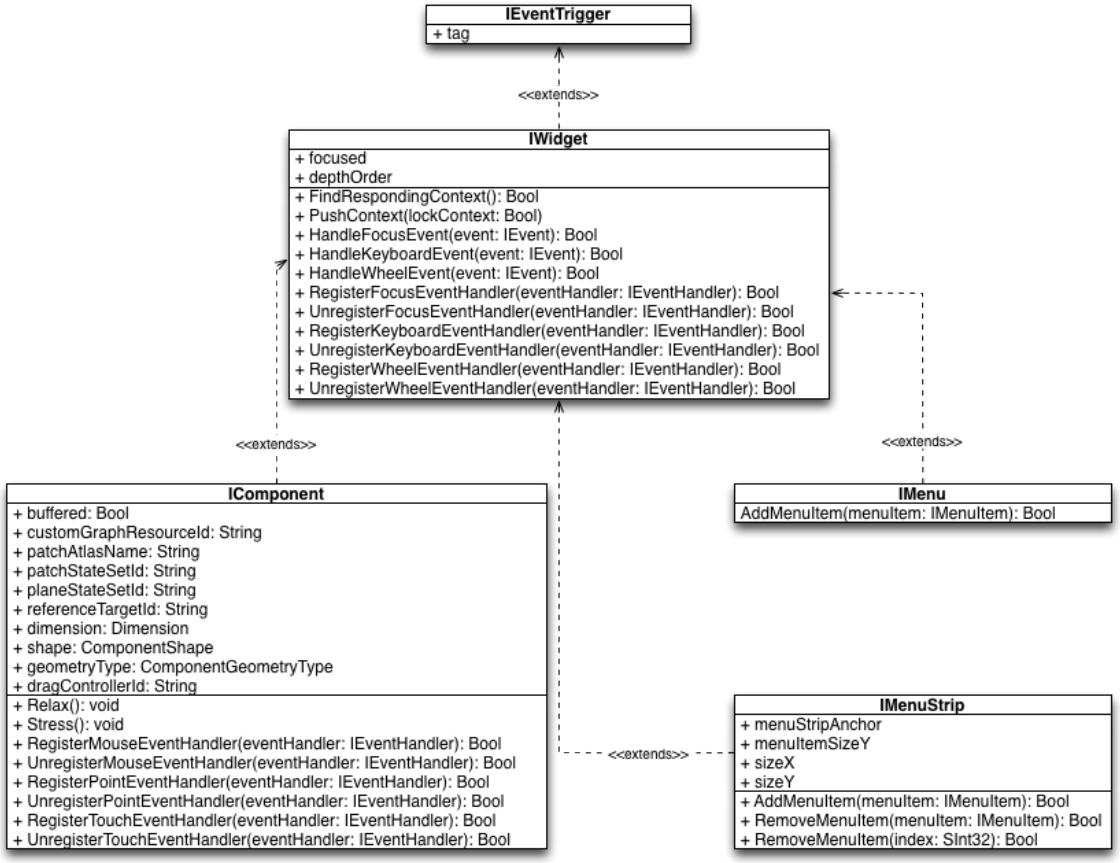

Figure 3.4: Widget classes and interfaces

gering actions by clicking or touching UI elements is the primary idiom of GUIs. The button of the MGT implements the `IButton` interface and is a subclass of `StatefulComponent`. All other Controls are subclasses of `Control`, which implements the `IControl` interface. Figure 3.7 shows the Control inheritance class diagram. Section 3.4 will cover each Widget in greater detail.

Layouts and Layout Directives

Layouts[5] are attached to Container widgets to control size and position of contained Component widgets. The base interface is called `ILayout` and is implemented by the abstract class `BaseLayout`, which serves as base class to all concrete Layout implementations. A developer may specify *Layout Directives* for each Component to explicitly control the layout process run by the parented Container. A Layout Directive is an implementation of an interface subclassed from `ILayoutDirective`, providing one interface type for each Layout type. Layouts and Layout Directives are both subclasses of `Node`, allowing them to be instantiated in a scene graph XML resource. Figures 3.8 and 3.9 show class diagrams revealing the relations between

[5]When using the term *Layout* (uppercase), I refer to the implementation of layouts in the MGT. The term *layout* (lowercase) is used when discussing layouts in a more general (toolkit-independent) context.

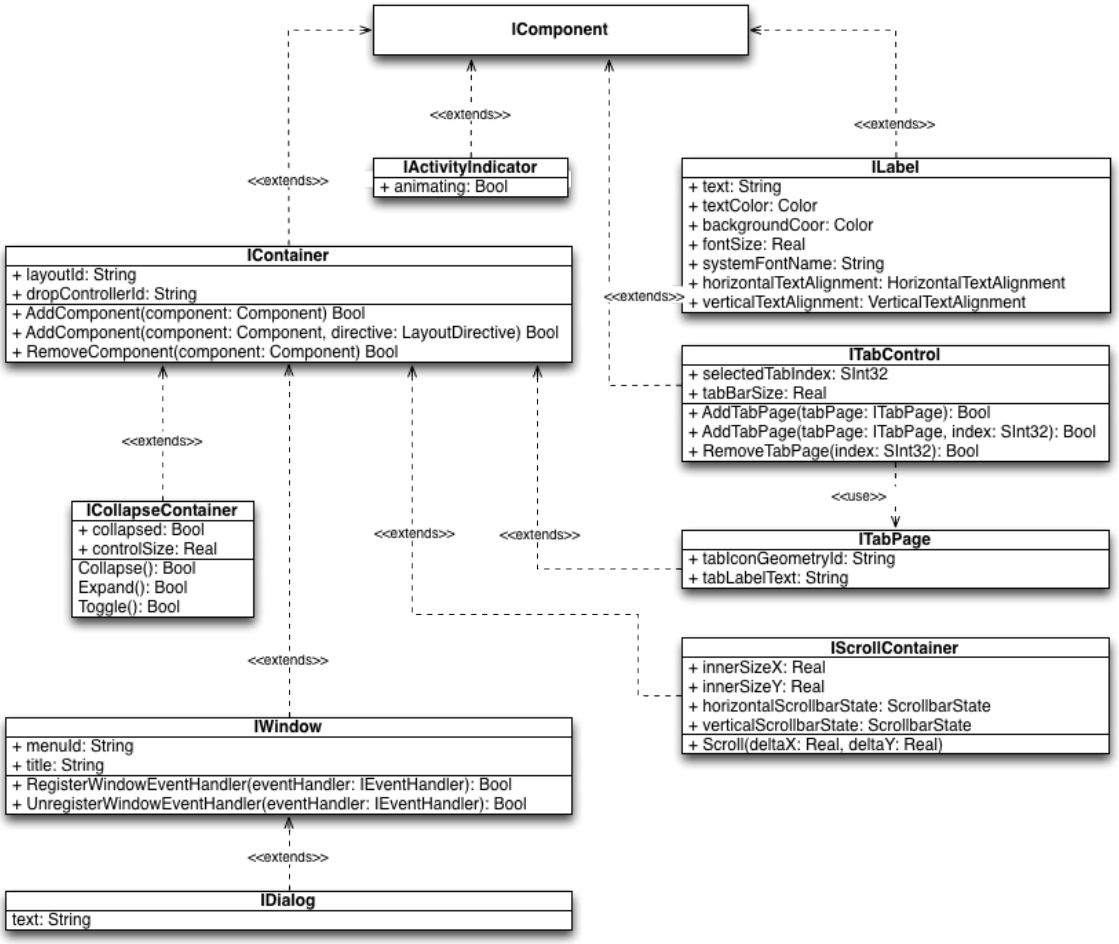

Figure 3.5: Component classes and interfaces

all Layout and Layout Directive types. Information on the Layout handling process itself will be covered in section 3.5.

The next two sections will illustrate the concept of two important MGT features, event handling and data management. An adequate understanding of those concepts is necessary, before having a closer look on the node types provided by the MGT for implementing GUIs in a scene graph.

3.2 Event Handling

3.2.1 Polling vs. Dispatching

A GUI is generally event-driven, meaning only user input events cause functions or methods to invoke. The event handling system of a GUI toolkit is generally a realization of the observer pattern. The ME, however, uses polling to query user interaction, device states, and logic states. To provide a convenient and transparent high-level API for developers familiar with event-driven programming, a message-based event handling system has been implemented as part of the MGT. Widgets and

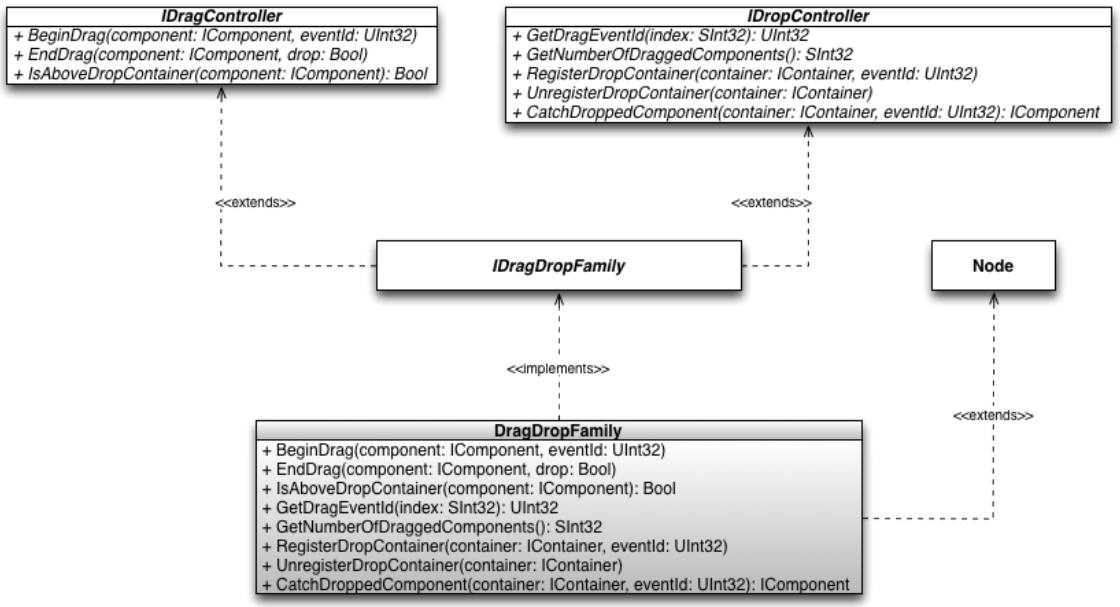

Figure 3.6: Drag-and-Drop control classes and interfaces

logics poll input information via the methods provided by the ME and encapsulate it into Event objects. Afterwards, these objects are passed to the event handlers through an Event Pipeline. Similar concepts are found in Java's AWT event handling or in the Core Foundation's `NSNotificationCenter`. This process will be described in detail in the next few sections.

However, the low-level alternative to this message-based approach, polling, is also available and equally informative. This means that everything that could be learned from dispatched Event objects can also be queried from Widgets or other Event Triggers directly. Which strategy to use depends on the application logic: Is it tick-based or event-based? Computer games are usually tick-based, so to keep things consistent, input actions on a game HUD shall be polled on each tick. As many game logics are implemented as finite state machines (FSM), a state transition shall be started from a state logic (e.g., by polling button presses) rather than being initiated externally (without knowledge of the state) at an undefined time. Other apps, such as image viewers or news readers, are idle until user interaction happens. In this case, event handling will just be fine for this job.

3.2.2 Events

Before explaining the dispatching system of the MGT, this section will describe the purpose of all available Event types already introduced in section 3.1.2. Events (uppercase!) are instances of `Event` and `IEvent` describing an event. They are dispatched to Event Handlers, which will then process the information according to the developer's needs. Events can be grouped into three categories, depending on how they were created. Table 3.1 classifies the Event types introduced below. The three categories are:

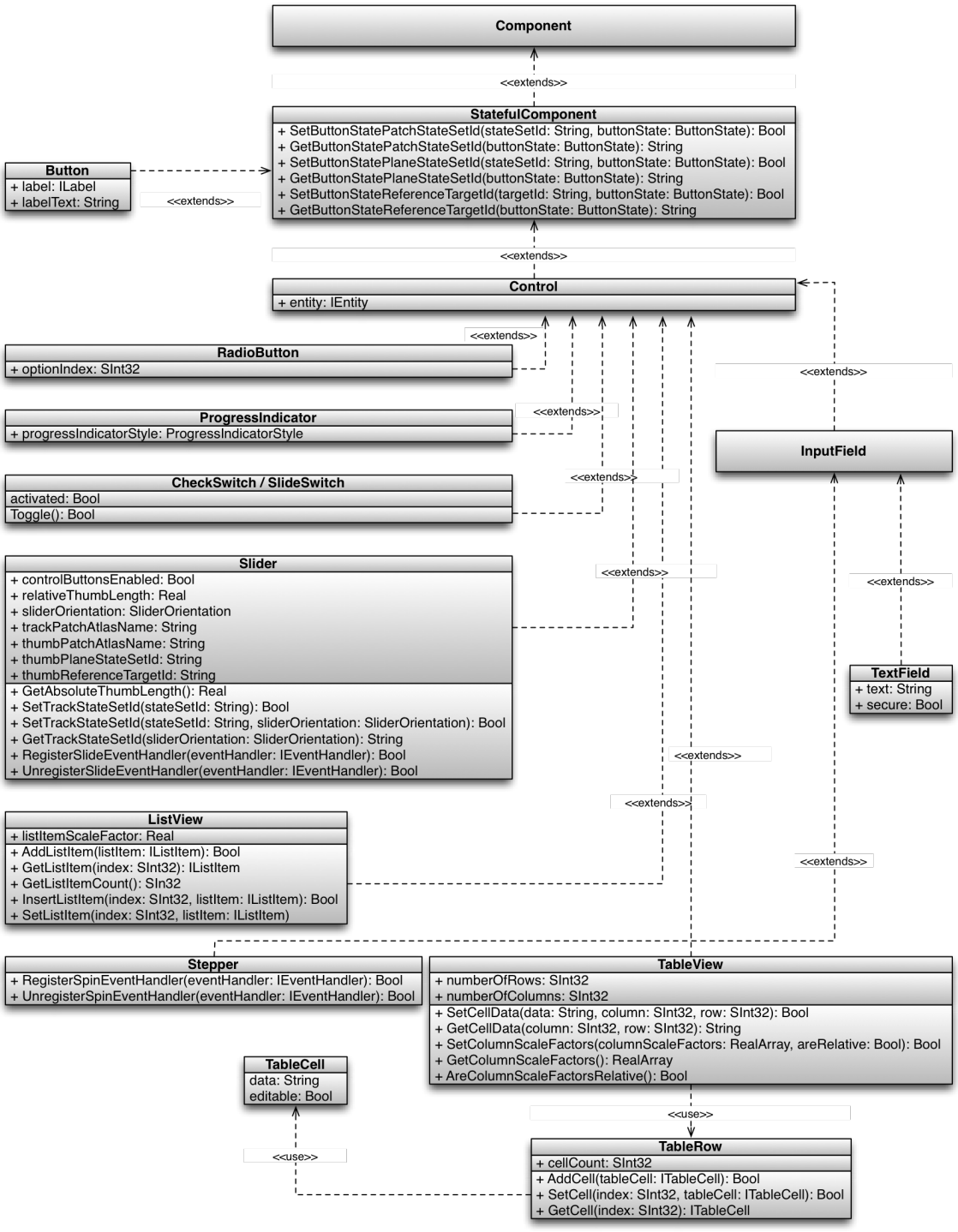

Figure 3.7: Control classes and interfaces

Controls can be seen as an own category within the Component inheritance tree. They are subclasses of `StatefulComponent`, which is an indirect subclass of `Node`. All non-abstract Control classes shown in the diagram can therefore be instantiated as Node in the scene graph XML resource. Note: For reasons of brevity, this class diagram only shows the class extensions of the Control classes. The implemented interfaces declare the same methods as their implementations and are named after the related classes, preceded by an `I`.

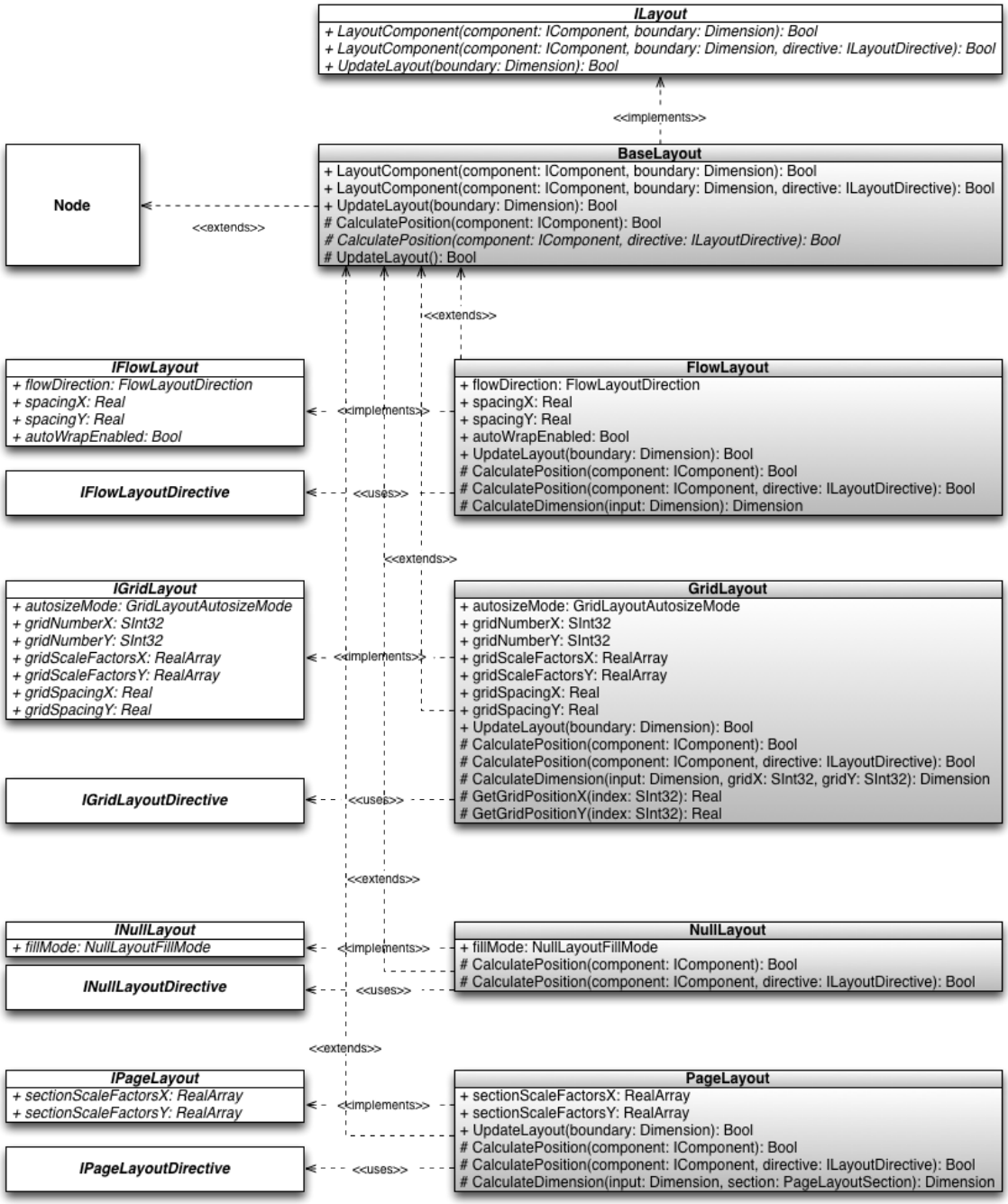

Figure 3.8: Layout classes and interfaces

A Layout is a subclass of `BaseLayout`, which is an implementation of `ILayout`, and a subclass of the ME base node class `Node`. Additionally, there are specific interfaces for each Layout to implement, as well as interfaces to access the respective Layout Directive information. The gray-shaded classes are private implementations and hidden to the user.

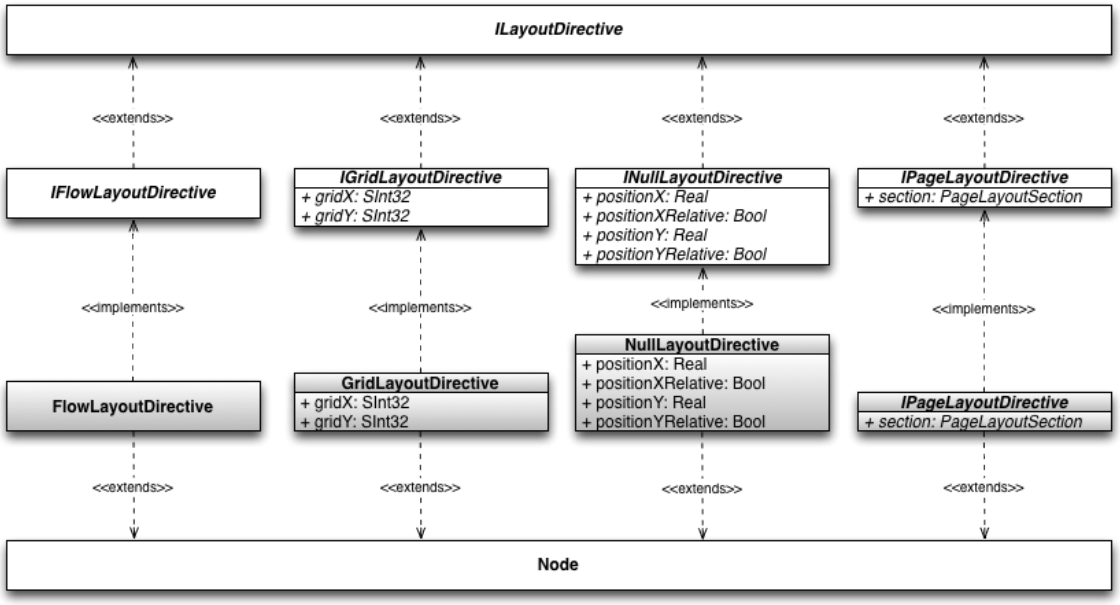

Figure 3.9: Layout Directive classes and interfaces

Layout Directives are subclasses of `Node` and implementations of `ILayoutDirective` or rather of one of its subclassed interfaces. They provide access to properties used for parameterizing the layout process. The gray-shaded classes are private implementations and hidden to the user.

- *Local input events* are generated from pointing input devices (e.g., mice, touch screens). The screen coordinate hit by the input device will be transformed to the local model coordinates of the Widget after applying reverse Model-View-Projection transformation and ray casting to find the objects intersected by the ray extruded from the input point. The resulting coordinates are always part of the information gained from local input events.
- *Global input events* are generated from non-pointing input devices like keyboards, gyroscopes, or mouse wheels. In contrast to input generated from pointing devices, there is no spatial context when using non-pointing devices, as they do not point to a specific position on the screen. The target Widget therefore needs to be determined by the Device Polling Logic (see below) included in the MGT after a global input event has been detected.
- *High-level events* can be generated from local or global input events or programmatically. Rather than describing the input event itself, they are focusing on the effects of the input event. For example, a Slide Event does not describe the involved touch gesture of the pointing device, but the thumb movement relative to the track.

Entity Event: Entity Events are fired after the data of an Entity has been changed. The Entity involved in the update can be retrieved from the `IEntityEvent` interface.

Event type	Event category
Entity Event	high-level
Focus Event	high-level
Keyboard Event	global input
Menu Event	high-level
Mouse Event	local input
Point Event	local input
Selection Event	high-level
Slide Event	high-level
Step Event	high-level
Touch Event	local input
Wheel Event	global input
Window Event	high-level

Table 3.1: Event type categorization

Focus Event: Focus Events are invoked whenever a Widget receives or yields the focus of the input devices. No more than one Widget may have the focus in the same time. Thus, two Focus Events will be fired if the focus moves from one Widget to another. The Event Handler is able to query the receiver and the yielder of the focus. The `IFocusEvent` interface is used to get information about the Widgets involved in passing the focus.

Keyboard Event: Keyboard Events are detected by the Device Polling Logic and invoked by the focused Widget after a key was pressed on the keyboard. The `IKeyboardEvent` interface provides information on the keyboard hit, e.g., which key was pressed.

Menu Event: Menu Events are fired by `IMenuBar` and `IMenuStrip` instances after a contained `IMenuItem` was selected. A pointer to the selected Menu Item can be retrieved from the Menu Event object, accessed by the `IMenuEvent` interface.

Mouse Event: Mouse Events are invoked on pressing or releasing a mouse button on a Component. The `IMouseEvent` provides access on information of this Event about the click position relative to the Component boundaries, the mouse button which performed the clicked, and the phase ("event type") of the click. The following event types are handled and propagated by the toolkit for all pointing devices:

- `PRESS`: The user presses down the pointing instrument (e.g., mouse button, finger) inside the boundaries of the area defined for point input.
- `RELEASE_INSIDE`: The user releases the previously pressed down pointing instrument inside the boundaries of the area defined for point input.
- `RELEASE_OUTSIDE`: The user releases the previously pressed down pointing instrument outside the boundaries of the area defined for point input.

Point Event: Point Events are an abstraction of Mouse Events and Touch Events, limited to simple, common properties. There will be no information about mouse buttons and only the touch event with the lowest tracked event ID[6] will be considered. Point Events are preferred in situations where the dedicated device properties do not affect the input idiom, i.e., the user points to an object and selects it. More comfort in event handling is therefore assured for GUIs, which are designed for both mouse and touch input devices, as only one type of event needs to be observed. This tackles a common problem on platform-agnostic application frameworks, which require developers to write dedicated code for both mouse and touch input handling. However, Point Events will only abstract mouse inputs, if the left mouse button was involved. Point Events invoked by multi-touch inputs will only handle the touch with the lowest tracked event ID. Information is accessible via the `IPointEvent` interface.

Selection Event: Selection Events are fired by instances of `ISelection`, which contains a list of items (e.g., Entities, see section 3.1.2) and an index pointing to a selected item, marking it as "selected". Widgets related to Selections usually provide options to choose from. The item related to the chosen option is then marked as selected by assigning the item index to the selection index. The previously selected index will be overwritten. A Selection Event provides a pointer to the `ISelection` instance and both indices: the index of the selected item and the index of the item that has been deselected in favor of the new selection. Information is accessible via the `ISelectionEvent` interface.

Slide Event: Slide Events are fired by Slider widgets (see section 3.4.6), if the thumb of the Slider is moved by the user or programmatically. The `ISlideEvent` interface allows to query the slide value, i.e., the relative thumb movement on the track. This value is normalized to $[-1, 1]$, where a negative sign means a decremental move (left or up). Slide Events do not propagate information about the current Entity value the Slider represents. This information can be retrieved from the Entity itself, which will fire a separate Entity Event.

Step Event: Step Events are fired by a Stepper (instances of `IStepper`) after its step up/down button has been pressed, but before the Entity updated the internal, numerical state representation. The `IStepEvent` will inform the Event Handler about the step direction, whereas a negative sign means a negative step. This information is available through the `IStepEvent` interface. Step Events do not propagate information about the current Entity value the Stepper represents. This information can be retrieved from the Entity itself, which will fire a separate Entity Event.

Touch Event: Touch Events occur on devices with touch screens or track pads when the user performs press or release actions on the input sensor. Compared to

[6] A tracked event is an input event of a pointing device that was started by pressing down within a given boundary. Examples are touch screen gestures and mouse clicks. On multi-touch input, each single touch will be identified by an unique tracked event ID.

Mouse Events and Point Events, Touch Events are able to track an infinite number of simultaneous inputs, only limited to the capabilities of the device. Coordinates and event types (same as on Mouse Event, see above) can be queried from the `ITouchEvent` interface.

Wheel Event: Wheel Events occur on any Widget that is focused in the current Context after a mouse wheel input happened. They are detected by the Device Polling Logic. Information is available via the `IWheelEvent` interface.

Window Event: Window Events are fired by Window widgets after a Window has been opened or closed. This information is provided by the `IWindowEvent` interface.

3.2.3 Device Polling Logic

As mentioned before, global input events must be polled by a Widget-independent logic. Local polling is done by the Widget logic itself for mouse or touch events and any other events that originate from the Widget itself. Polling mouse and touch events is possible because the ME provides nodes to define clickable/touchable areas within a scene graph. This node is called `Button` (not to confuse with the MGT Widget with the same name) and allows to query common pointing device input actions that happened during the most recent tick. Furthermore, other local events like stepping, sliding, or focusing are generated from pointing device inputs on child node button areas. Global polling is necessary for all kinds of events that do neither happen on a local button area (local input events) nor originate from the logic of the Widget (high-level events).

The `EventPollingLogic` class is a logic processor for polling global input events and delegating them to the active Context. A *Context* is a node implementing the `IContext` interface that manages the Focus of descendant Widget nodes. A *Focus* is a pointer to a Widget that will be notified on global input events and will then create the appropriate Event objects. It can be moved from one Widget to another by either selecting it with the pointing input device or by pressing TAB (forward) or SHIFT+TAB (backward) on the keyboard. The following constraints are defined for this concept of Context and Focus[7]:

1. No more than one Widget per Context can be focused at a time.
2. No more than one Context per app session can be active at a time.
3. Inactive Contexts do not have any Widget focused.

A fourth constraint can be deducted from the first three: No more than one Widget per app session can be focused at a time. This abstract explanation just describes a common GUI behavior every user is familiar with. One can navigate through the

[7]Be aware not to confuse with the focus-plus-context technology, allowing both an overview (the context, e.g. a map) and a detail (the focus, e.g. a building) to be displayed simultaneously on a screen.

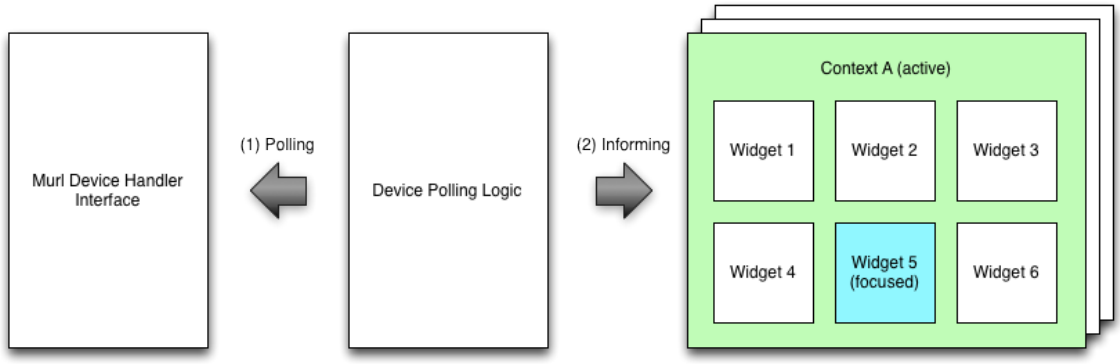

Figure 3.10: Device Polling Logic, Contexts, and Focus

The Device Polling Logic polls the device handler for input events on each tick. If there has been input on the most recent tick, the gathered information will be passed to the active Context. The active Context then creates the Event object by declaring the focused Widget as Event trigger. It will then be passed to the pipeline.

GUI widgets with the TAB key and focus a particular widget by clicking or touching it. But navigating only works for widgets within the same window or tab page and only if the window or tab page "is in front". In the MGT, the Context is a realization of this constraint, though it is not restricted to windows or tab pages. Contexts have implicit sort orders for rendering GUI layers, whereas the Context with a higher sort order will occlude the Context with a lower sort order. The active Context is always on the top layer of stacked Contexts and thus not occluded by the Widgets of any other Contexts. The effect can be seen in figure 3.10. Activating another Context pushes it onto the top of the stack and moves down the previous active Context for one layer. This implies that Contexts with lower sort orders have been inactive for a longer time than Contexts with a higher sort order (despite the possibility that they might have been initialized with a lower sort order). Figure 3.11 illustrates how these push operations affect the Context ordering on the stack. Note that the terminology of stacks has been chosen for reasons of comprehensibility, unaware of the actual algorithm implemented.

Finally, it is also possible to lock the active Context. No other Context can then be pushed active unless the lock is undone by the logic. A common use case for this feature is showing a dialog window, which blocks all other user interaction apart from pressing one of the dialog buttons.

3.2.4 Event Dispatch Table, Event Pipeline, and Event Channels

The `EventPipeline` singleton is in charge for dispatching Events to Event Handlers. Event objects are passed to the pipeline together with a reference to an instance of `EventChannel`. An *Event Channel* is used to identify the origin of an Event and can be mapped to an array of targeted Event Handlers. Each Event Trigger has one Event Channel for each type of Event that can be created by the Event Trigger. For example, a mere `Component` can trigger six different types of Events, thus every single

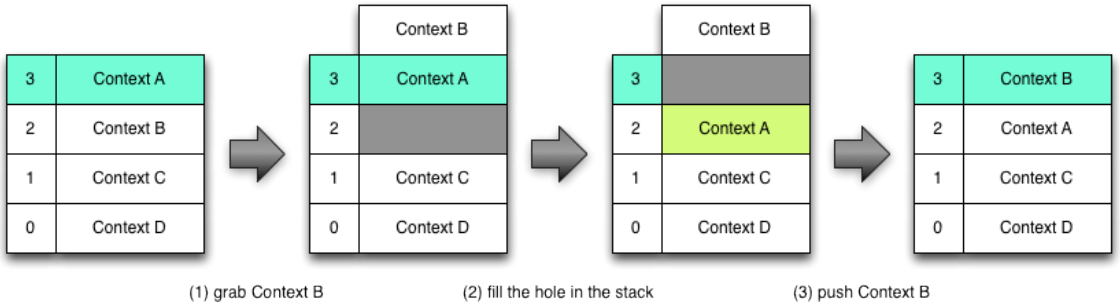

Figure 3.11: Pushing Contexts active

The active Context always has the highest order on the stack (marked sea green). Pushing Context B will affect the order of all Contexts in the stack above B (marked light lime green) as they need to fill up the gap left by B (marked dark gray). B will be pushed on the top of the stack afterwards.

instance will have six Event Channels in use. The `EventDispatchTable` is another singleton that manages the assignment of Event Handlers to Event Channels. The cardinality of their relationship is $m : n$. One Event Channel can be used to dispatch an Event to one or more Event Handlers, one Event Handler may receive Events from one or more Event Channels. The complete process of Event dispatching to a single Event Handler, depicted in figure 3.12, can be described as follows:

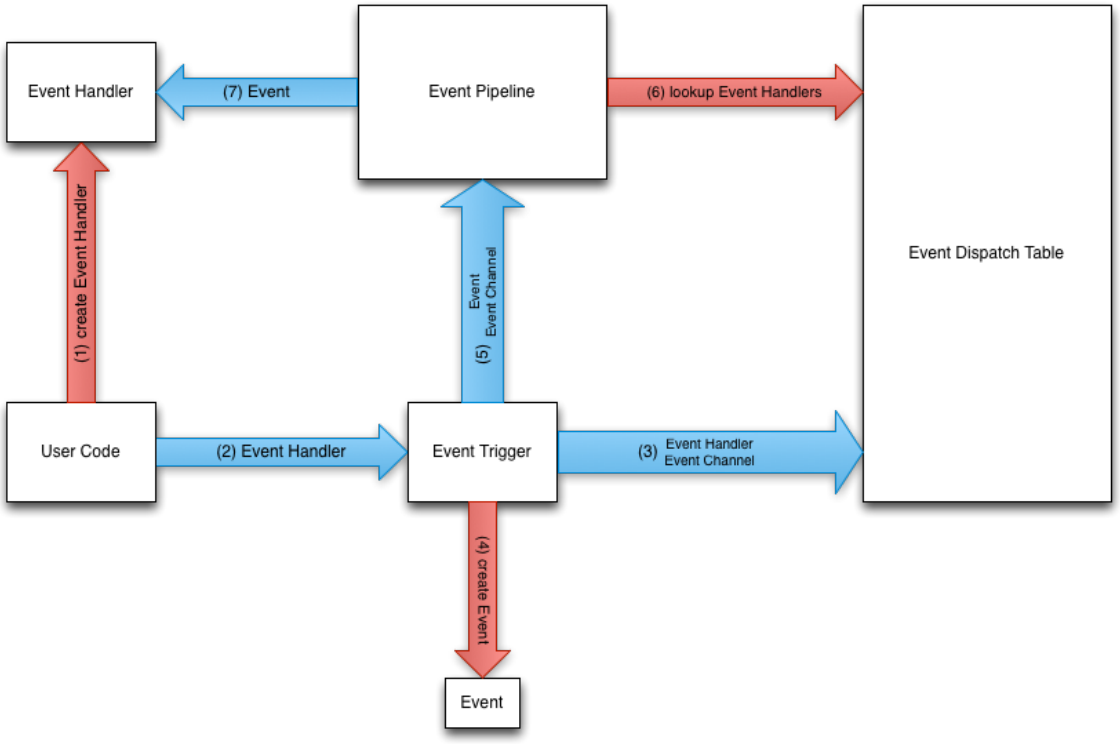

Figure 3.12: Event dispatching

The blue arrows symbolize parameter passing by calling a method. The red arrows mark the retrieving of an object, i.e., by creating one.

1. An instance of `IEventHandler`, the Event Handler, is created by user code.
2. The Event Handler is passed to the appropriate registration method of the Event Trigger.
3. The Event Trigger passes its appropriate Event Channel and the Event Handler to the Event Dispatch Table, which maps both objects. This is called registration.
4. The Event Trigger discovers the occurrence of an event and creates an Event object.
5. The Event Trigger passes the Event and the corresponding Event Channel to the Event Pipeline.
6. The Event Pipeline retrieves a list of all Event Handlers mapped to the given Event Channel from the Event Dispatch Table.
7. The Event Pipeline iterates over all Event Handlers and calls their `Perform()` method using the Event object as parameter.

3.2.5 Event Handlers

An *Event Handler* is an implementation of the `IEventHandler` interface, which declares the abstract method `Perform()` to implement (see figure 3.13). This method is called to handle the Event, passed as parameter, by a custom implementation. The interface is unified for all kind of Events, thus the developer must take care to handle the correct type (e.g., by applying dynamic casting). There are two different ways to implement an Event Handler:

- Event Handlers can be written by implementing the `IEventHandler` interface either as a dedicated class or as an arbitrary class. This is similar to implementing the `EventListener` interface in Java AWT.
- Event Handlers can be written as custom methods with the same parameters and return values as the `Perform()` method declared in the `IEventHandler` interface. An instance of the `CallbackEventHandler` template class will then wrap the function pointer to this method and the object the method shall be called on (see figure 3.13). The `CallbackEventHandler` instance will be treated as the Event Handler since it is an implementation of `IEventHandler`. Calling `Perform()` on this instance will simply invoke the method the function pointer points to.

Event Handlers must be registered to receive Events. The registration is done by calling dedicated registration methods of `IEventTrigger` instances, which require the Event Handler to be passed as argument. Event Triggers are Widgets, Entities, or Selections. The class diagrams in figures 3.3 and 3.4 give information about which interfaces provide registration methods (`Register*EventHandler()`) for what Event types.

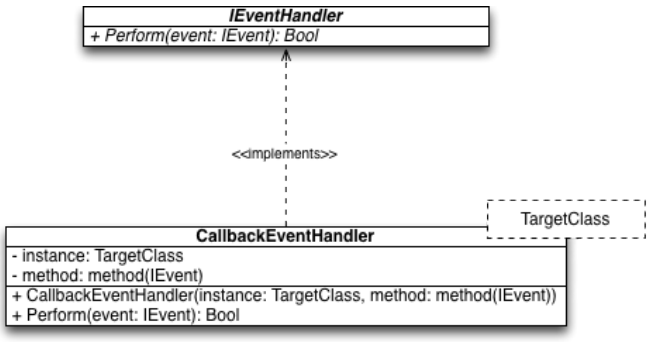

Figure 3.13: Event Handler classes and interfaces

3.2.6 Event Handler Table

The `EventHandlerTable` is not an integral part of the event handling subsystem but offers a method for global mapping of identifier strings to `IEventHandler` instances. This can be very useful for GUIs defined entirely in XML resources, allowing Widget nodes to specify which Event Handler will handle certain kind of Events. The Event Handler to use must first be registered by the Event Handler Table on application startup by assigning an unique identifier string. On deserialization, the identifier string gained from an attribute of the Widget's XML node will be used to lookup the Event Handler. If found, the Widget will take care of the registration, sparing the developer from doing this in a logic processor. Table 3.2 shows which attribute is defined for what Widget.

This concludes the explanation on a basic feature of the MGT. The next section will describe the data handling concept, which relies on the event handling subsystem introduced above.

3.3 Data Management

The extent of data management features provided by a GUI toolkit depends on the primary fields of application. This varies from simple widget object properties to full support of relational databases. The next two sections explain which data management concepts are provided by the MGT.

3.3.1 Entities

Some widgets shipped with the MGT are called *Controls* (after their common base class), see section 3.1.2 for more details. Their main purpose is to present data, while processing output, and to manipulate data, while processing logic (this is when user interaction is parsed). Text fields are the most obvious examples for this kind of widgets. They are able to process user input and update an internal string value as well as display the current string on screen. The evaluation has shown that data handled by control widgets can be categorized the following way:

	focusEventHandler	keyboardEventHandler	wheelEventHandler	mouseEventHandler	pointEventHandler	touchEventHandler	slideEventHandler	stepEventHandler	menuEventHandler	windowEventHandler	entityEventHandler	selectionEventHandler
ActivityIndicator	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
Button	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
CollapseContainer	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
Component	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
Container	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
Dialog	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘	✘
Label	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
ListItem	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✔	✘
ListView	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✔
Menu	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘
MenuItem	✔	✔	✔	✔	✔	✔	✘	✘	✔	✘	✘	✘
MenuStrip	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘
OptionButton	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✔
ProgressIndicator	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✔	✘
ScrollContainer	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
Slider	✔	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
Stepper	✔	✔	✔	✔	✔	✔	✘	✔	✘	✘	✔	✘
Switch	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✔	✘
TabControl	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
TabPage	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘
TableView	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✔
TextField	✔	✔	✔	✔	✔	✔	✘	✘	✘	✘	✔	✘
Window	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘	✘

Table 3.2: XML attributes available for XML tags

This table shows which XML tag (rows) supports which XML attribute (column). The tags represent an instance of Widget while the attributes refer to Event Handler instances that must be registered by their identifier on application startup.

- *Switch:* A boolean value representing an on/off state of an arbitrary option.
- *Number:* A fixed or floating point number value within given limits.
- *Text:* An arbitrary string value.
- *List:* A list of values of any type, including those described in a category above.

Instead of immediately storing their raw data as object properties of a Widget, a proxy object that wraps the data will be used. Those objects are called *Entities* and are subclasses of the abstract type `Entity` and implementations of the `IEntity` interface. The class hierarchy of the Entity type has already been outlined in figure 3.3. The role of the Entity types are best described in the following paragraphs by having a closer look on the provided methods.

Entity: The abstract superclass defines methods for registering and unregistering Event Handlers on Entity Events. Events are fired by the protected method `Notify`, which is called by all subclasses on data updates.

Number Entity: The *Number Entity* (class `NumberEntity`, an implementation of `INumberEntity`) wraps a raw number stored as fixed point or floating point value. Beside changing the number itself, methods for incrementing, decrementing, and altering the step size (which defines the incremental/decremental step) are available for manipulating the value.

Switch Entity: The *Switch Entity* (class `SwitchEntity`, an implementation of `ISwitchEntity`) holds a boolean value and provides methods to read, write, or toggle the value.

Text Entity: The *Text Entity* (class `TextEntity`, an implementation of `ITextEntity`) contains a string and provides some basic methods for string manipulation.

Selection: A *Selection* (class `Selection`, an implementation of `ISelection`) is a common view on a set of indexed items, where an arbitrary number of items (identified by their indices) can be marked as selected. The interpretation of the Selection depends on how this Entity is used. The MGT uses Selections for Option Buttons and List Views (see section 3.4.6) to track which option or List Item is currently selected. At this level of abstraction, the Selection does not define which kind of data is handled or stored. Selections may refer to arrays, but their common description only consists of the number of contained items and the selection indices.

Although the provided implementation of the Entities just wraps raw data types, the interfaces themselves do neither define where to store data nor from where to retrieve it. Thus, custom implementations of Entities can wrap around any kind of raw data (as an example, consider the system audio volume level) as long as their usage conforms to the implemented interface.

3.3.2 Entity Events and Selection Events

The communication between Widgets and Entities is primary done by Events. The Widget parses user interaction and forwards the update directly to the Entity rather than updating its own appearance. After the update, the Entity fires an Entity Event. Controls handle Entity Events of their Entities by default, so the fired event will be handled by the Widget which has been in charge for the update. Figure 3.14 shows this behavior as sequence diagram. Selection Events work similar, but are used for Option Buttons and List Views.

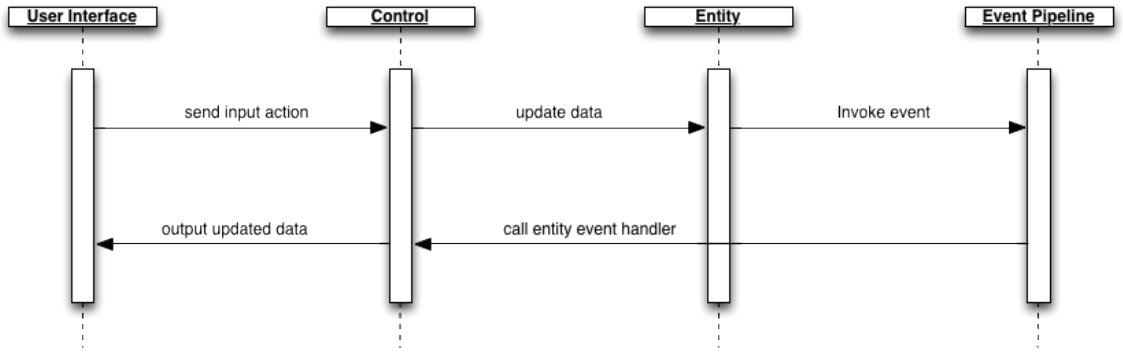

Figure 3.14: Control updating through Entity Events (sequence diagram)

This cyclic process might seem strange. However, on closer examination, it is a clean and efficient way to keep data and appearance consistent and transparent. The one-way synchronization from Entity to Widget ensures consistency. There is no way to update the appearance independently from the Entity, and every Entity update (caused by user interaction or programmatically) yields an Event that is handled by the Widget. Transparency is achieved by keeping the number of LOCs[8] low. An Entity Event will be fired by any means, so if the Widget updates its appearance before it updates the Entity, it will finally receive an Entity Event after the Entity update. This causes the Widget to refresh its appearance again. This can be prevented by unregistering the Event handler before updating the Entity and registering it again afterwards, which is, in fact, not a perfect solution either.

3.4 Widgets

Widgets are subclasses of the engine's basic node class `Node`, making a GUI created by the MGT part of the scene graph. This allows the Widgets to be affected by most scene graph features of the ME, for example, key-frame animations, transformations, rendering effects, and traversal manipulators (see section 1.4.3). Furthermore, as a consequence of Widgets being Nodes, Widgets must be designed as a composition of other Nodes. To accomplish this, the complete palette of Nodes available in the ME can be used for this purpose, with transformations, geometries, and surface rendering nodes being the most important ones. The following sections will describe the properties of each Widget and their internal composition.

[8]LOC, lines of code

3.4.1 The Widget Node

The abstract base class of any Widget available in the MGT is called `Widget`. It extends the `Node` class and implements the `IWidget` interface, which is an subclass of `IEventTrigger`. It does not handle any geometry, as its subclasses differ too much in this aspect. However, common to all Widgets is the ability to push the Context they belong to (see section 3.2) and to manage registration of Event Handlers for the following Event types: Focus Event, Keyboard Event, and Wheel Event. These Events are not generated by the Widget itself, but by the Context it belongs to. However, they will be informed by the Context, if a global input Event or a Focus Event occurs. By default, the Widget just updates its internal focus flag to whether it received or yielded the Focus. Table 3.3 contains a list of all properties common to all Widget subclasses. The property names refer to the XML attribute names of the corresponding Node. More XML attributes are specified in table 3.2. Properties may also be accessed via the appropriate getter/setter methods defined in the appropriate class. Please note that all property tables in this section only describe properties which are defined by the corresponding Widget itself.

Name	Type	Description
`depthOrder`	integer	The depth sorting order used to control the layering, while rendering the Widget geometry.
`tag`	string	The custom (and preferably unique) tag of the Event Trigger, which may be used in custom Event Handlers to distinguish between Widgets.

Table 3.3: Widget properties

3.4.2 Menu Bars, Menu Strips, and Menu Items

Menu Bars and *Menu Strips* are both implementations of the `IMenu` interface or, more precisely, of its descendant interfaces `IMenuBar` and `IMenuStrip`. Their common property abstracted by `IMenu` is the capability of containing an arbitrary number of *Menu Items* (`IMenuItem`). Menu Bars and Menu Strips differ in presentation and in field of usage. Further, Menu Strips may be attached to Menu Items to pop up as submenu. With these node types, custom hierarchical menus can be defined for many purposes. Figure 3.15 is a mockup illustrating the usage of Menu Bars, Menu Strips, and Menu Items.

Menu Bars are main menus displayed below the title bar of a window. This is a widely used pattern in desktop GUI applications, but not so in mobile applications. The output generated from Menu Bars is a flat bar with horizontally arranged Menu Items going from left to right. Selecting a Menu Item will highlight the label with a different background and open a submenu, if a Menu Strip is attached.

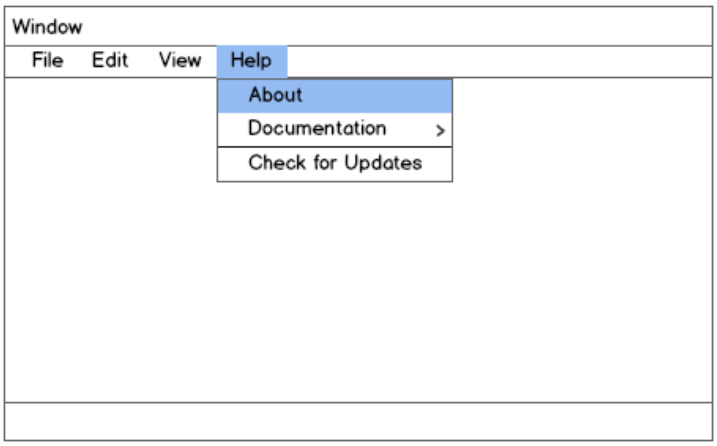

Figure 3.15: Menus, Menu Items, and Menu Strips

A Menu is a menu bar below the window title bar. A Menu Item represents a single item within a Menu or a Menu Strip. A Menu Strip is a vertically arranged list of Menu Items that is either attached to a Menu Item within a Menu or a Menu Item within another Menu Strip.

Menu Strips serve multiple purposes. They can be used as

- submenus of Menu Items,
- context menus of Components, and as
- popup menus on the bottom of App Windows (used in mobile GUIs, see figure 3.16 for an example).

Menu Strips can be attached to Menu Items, Components, and App Windows by their Menu (Strip) Node Target property. Table 3.4 sums up all properties available to Menu Strips.

Name	Type	Description
`menuItemSizeX`	real	The width of a contained Menu Item, equal to the width of the Menu Strip.
`menuItemSizeY`	real	The height of a contained Menu Item. The total height of the Menu Strip is calculated from the Menu Item height times the number of Menu Items.

Table 3.4: Menu Strip properties

Menu Items represent selectable items in both Menu Bars and Menu Strips. On selection, they fire a Menu Event. This requires the Menu Item to be in an enabled state. It is also possible to use a Menu Item as parent of a submenu created from a Menu Strip. A graphical icon (e.g., an arrow) will then indicate the existence

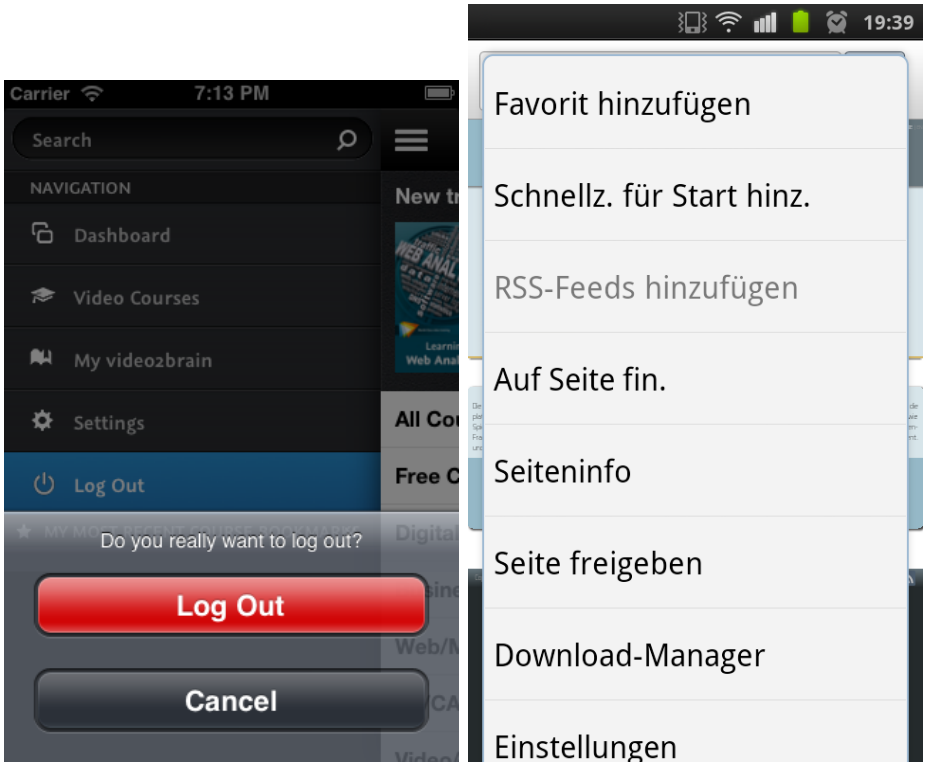

Figure 3.16: Menus, Menu Items, and Menu Strips

Popup menus are widely used in mobile apps. Cocoa Touch offers the class `UIActionSheet` to create popup menus that appear at the bottom of the screen (left). On Android, this is done by the class `android.view.Menu` (right).

of a submenu, which will open on hover or touch. Menu Items are not subclassed from Widget. The node itself does not present anything, it just provides information for the containing Menu Bar or Menu Strip. The rendering of Menu Items is actually done by Menu Bars and Menu Strips. Menu Strips will also consider the type property of Menu Items, which defines their presentation either as `TEXT` or as `SEPARATOR_LINE`. Separator lines are not selectable though. The full list of properties of a Menu Item is shown in table 3.5.

Name	Type	Description
`title`	string	The title to be displayed in the Menu containing the Menu Item.
`enabled`	boolean	If true, the Menu Item can be selected by the user.
`menuItemType`	enum	The type of the Menu Item (ignored by Menu Bars).

Table 3.5: Menu Item properties

3.4.3 Components and Containers

The key to compose complex GUIs is the Component/Container concept adopted from AWT and Swing. In the MGT, Components are rectangular Widgets with custom size, origin, and appearance. Containers are Components capable of containing other Components. Furthermore, Containers can use Layouts (see section 3.5) for arranging child Components automatically by controlling their size and origin.

Component: *Components* are instances of the `Component` class, subclass of `Widget` and implementation of `IComponent`. Mere Components are already aware of some Event types (see table 3.2) and can be customized in means of size, origin, and appearance. They also support drag and drop and provide access to the dimension property, which is used by Layouts as input/output parameter on layout computation. However, there is no further logic behind them. A probable field of use may be the presentation of flat images, e.g. icons in a GUI. A Component is described by the properties listed in table 3.6.

All other Widgets, except those introduced in section 3.4.2, are descendants of `Component` and will therefore inherit its properties. This is essential for a unified Component/Container concept that supports layout management and composing.

Name	Type	Description
`customGraphResourceId`	string	The ID of the graph to instantiate instead of the default subgraph.
`dragControllerId`	string	The Node ID of the drag controller (see below).
`buffered`	boolean	If true, the Component and all its children are rendered into a frame buffer and displayed onto a plane geometry.
`dimension`	vector	Size (width, height) and origin (x, y) of the Component.
`buttonShape`	enum	The shape of the click/touch area surrounding the Component.
`geometryType`	enum	The geometry type of the Component (see section 3.6).
`patchAtlasName`	string	The ID of the atlas used to create a generic geometry (see section 3.6.2).
`patchStateSetId`	string	The state set to use for rendering a generic geometry (see section 3.6.2).
`planeStateSetId`	string	The state set to use for rendering a plane geometry (see section 3.6.2).
`referenceTargetId`	string	The Node ID to refer to for rendering a predefined geometry (see section 3.6.2).

Table 3.6: Component properties

Container: *Containers* are instances of the `Container` class, subclass of `Component` and implementation of `IContainer`. Containers extend Components by the capability of containing other Components as child nodes. Furthermore, Layouts can be used by Containers for automatically manipulating size and position of contained Components according to the Layout parameters. Layouts are covered in more detail in section 3.5. Containers have a property to refer to a Layout node (specified by the ID) to use for the layout process. Therefore, a Layout node must be created before the Container node in the scene graph. Layout nodes usually are not part of the Component-Container hierarchy. Since Layouts keep track of the Components laid out, having two or more Containers referring to the same Layout will lead to an undefined behavior. Table 3.7 contains a list of dedicated properties used by Containers.

Name	Type	Description
`layoutId`	string	The Node ID of the Layout to use for arranging contained Components.
`dropControllerId`	string	The Node ID of the drop controller (see below).

Table 3.7: Container properties

Collapse Container: *Collapse Containers* are instances of the `CollapseContainer` class, subclass of `Container` and implementation of `ICollapseContainer`. In addition to the default Container behavior and properties, Collapse Containers come along with a control bar on the top, containing a title label and a button to toggle between the collapse/expand state. The expand state is the default state and indicates that the Container itself and its content is fully visible. The collapsed state, however, hides the Container and its content, while the control bar remains visible. This will also cause the Widget dimension to collapse to the size of the control bar. A Collapse Container is described by the properties listed in table 3.8.

Name	Type	Description
`collapsed`	boolean	If true, the Collapse Container is collapsed, otherwise expanded.
`controlSize`	real	The vertical size of the control bar. The effective Container size will be reduced by this value to fit into the Widget size.

Table 3.8: Collapse Container properties

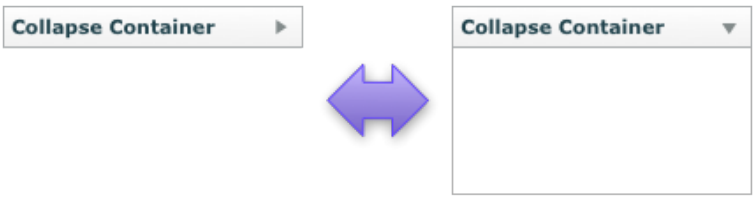

Figure 3.17: Collapse Container

A Collapse Container includes a control bar above the container area. The control bar contains a customizable title label and an expand/collapse button to show or hide the container area. The button icon indicates the current expand/collapse state.

Scroll Container: *Scroll Containers* are instances of the `ScrollContainer` class, subclass of `Container` and implementation of `IScrollContainer`. In contrast to other Container types, the area containing the child Components, the *inner container*, is not constrained to the size of the Scroll Container widget. The Scroll Container acts as "window" that only shows a segment of the inner container. Scrollbars are available on the right (vertical) and bottom (horizontal) edge of the Scroll Container, allowing the user to move the viewing window continuously over the inner container. This process is known as *scrolling*. In some situations, the inner container may be smaller than or equal to the size of its window. Scrolling does then have no effect as the entire content is fully visible. The visibility of the scrollbars may be customized, especially with regard to such situations. Table 3.9 mentions all dedicated properties of Scroll Containers. A schematic of a Scroll Container is shown in figure 3.18.

Name	Type	Description
`innerSizeX`	real	The total width of the inner container.
`innerSizeY`	real	The total height of the inner container.
`horizontalScrollbarVisibility`	enum	Determines when the horizontal scrollbar shall be displayed: `ALWAYS`, `NEVER`, or `ON_DEMAND`.
`verticalScrollbarVisibility`	enum	Determines when the vertical scrollbar shall be displayed: `ALWAYS`, `NEVER`, or `ON_DEMAND`.

Table 3.9: Scroll Container properties

Drag Drop Family: *Drag Drop Families*[9] are instances of the `DragDropFamily` class, subclass of `Node` and implementation of `IDragDropFamily`, which provides

[9]The term "family" was preferred over "group" because the latter term probably implies a structural purpose of the node.

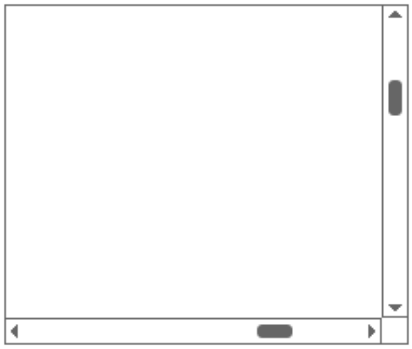

Figure 3.18: Scroll Container

A Scroll Container consists of one or two optional scrollbars, an inner container area of any size, and a plane ("window") that shows a segment of the inner container.

access to the `IDragController` and `IDropController` interfaces. They are no Widgets themselves, but an extension of the Component/Container model introduced above. Drag-and-drop support is available through the GUI scene graph XML resource to minimize programming effort. To use drag-and-drop, the developer must at least set up the following nodes:

- A Component with a Drag Controller Node Target.
- A Container with a Drop Controller Node Target.
- A Drag Drop Family which serves as Drag/Drop Controller.

All Components are draggable, if they refer to a Drag Controller, which is an instance of `IDragController`. Similarly, all Containers can "catch" Components dropped "above" them and add those as a child, if they refer to a Drop Controller, which is an instance of `IDropController`. The Drag Drop Family is a special Node, which implements both the `IDragController` and the `IDropController` interfaces, so it will act as node target for Components and Containers. Components can only be dropped into Containers, which belong to the same Drag Drop Family, hence both Components and Containers must refer to the same Drag Drop Family Node Target.

Multiple Drag Drop Families can be created to define which Components are "compatible" with which Container. As an example, consider an inventory/equipment menu of a role-playing game (RPG). Different types of equipment may be provided to move objects (Components) from the inventory (a grid-structured Container) to the equipment slots (single-cell Containers). An existing example is shown in figure 3.19.

3.4.4 Tab Controls and Tab Pages

Tab-based GUIs can be created by using Tab Controls and Tab Pages. In desktop applications, tabs are often embedded in GUIs to switch between forms focusing on

Figure 3.19: An inventory menu of an iOS RPG.

(c) 2009 Pixel Mine, Inc.

different aspects. Tabs behave the same in mobile applications, but in contrast to desktop applications, they are even more used for navigating through the app on the top navigation level. Nevertheless, Tab Controls and Tab Pages are used in both cases. Figure 3.20 shows a possible result generated by a Tab Control with some Tab Pages.

Tab Control: *Tab Controls* are instances of the `TabControl` class, subclass of `Component` and implementation of `ITabControl`. They are responsible for grouping Tab Pages that belong together and generating the tab bar according to the icon and label properties of the Tab Pages. In the scene graph, Tab Controls are parent nodes of Tab Pages. Tab Pages can also be added or removed programmatically during the initialization process. Table 3.10 lists all dedicated properties of Tab Controls.

Name	Type	Description
`selectedTabIndex`	integer	The zero-based index of the tab button currently selected.
`tabBarSize`	real	The height of the tab bar. The effective Tab Page size will be reduced by this value to fit into the Widget size.

Table 3.10: Tab Control properties

Tab Page: *Tab Pages* are instances of the `TabPage` class, subclass of `Container` and implementation of `ITabPage`. The content of a tab is held by a Tab Page, which is similar to the Container Widget. Additionally, there are two properties describing the appearance of the tab button on the tab bar, as explained in 3.11.

Name	Type	Description
`tabIconTargetId`	string	The Node ID of the plane geometry (possibly generated by the texture atlas generator) that presents the tab icon (optional).
`tabLabelText`	string	The text to be displayed as label in the tab button.

Table 3.11: Tab Page properties

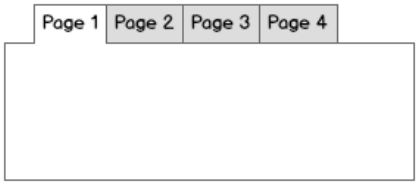

Figure 3.20: Tab Control with Tab Pages

The Tab Control generates the tab buttons from information provided by the Tab Pages and switches between them whenever the user presses a tab button. Tab Pages are just Containers with tab information.

3.4.5 Windows, Dialogs, and the App Window

Widgets of type `IWindow` inherit from `Container` and come along with a title bar and a close button in it. A *Window* can be moved around by dragging the title bar. Dragging the lower right corner allows the user to resize the Window. Windows are provided by most toolkits in some variations and are addressed by the term *modal* (e.g., modal view, modal form, modal window, etc.). *Dialogs* are special purpose derivates of Windows with some additional restrictions. Another window-related kind of object only included in the MGT is the *App Window* proxy, which is actually not a Widget. Windows, Dialogs, and App Windows define separate Contexts (see section 3.2) for each instance, which groups the contained Widgets in consideration of event handling for user interaction. See below for more details.

Window: *Windows* are instances of the `Window` class, subclass of `Container` and implementation of `IWindow`. Figure 3.21 shows the elements of a Window widget which were added to the default Container:

- A title bar for dragging the Window,
- a label for custom title strings,
- a close button to close the Window,
- and a drag button to change the Window's size.

A Window, in contrast to the base Container, provides a Context for all contained Components. Multiple Windows can overlap each other, having the Window of the active Context (i.e., the window which is receiving user input) above all others. As a subclass of Container, a Window can use Layouts to automatically arrange its child Components. Additionally, the *Menu Node Target* of the Window may refer to a previously defined Menu to be included as main menu of the Window. The MGT includes some default animations for opening and closing a Window, but the developer is free to assign any animation for this purpose. Proprietary ME animations are defined as keyframes in an animation XML file. As a side note, opening and closing a Window also yields a Window Event. Table 3.12 lists all supported properties of a Window.

Dialog: *Dialogs* are instances of the `Dialog` class, subclass of `Window` and implementation of `IDialog`. Dialogs are supplemented by some other Components, intended to act as a message box with a Label and one or more Buttons. This spares some effort to the developer, since there is no need to build it from the ground up. The most important feature of a Dialog is the behavior of "locking" the Context when showing up. This prevents all other Widgets outside the Dialog to receive events, which is an expected behavior. Dismissing the Dialog can be achieved by hitting one of its buttons. After dismissal, the Context will unlock. Dialogs do not have draggable corners to resize them. Their size depends on the content or on customized settings. A schematic Dialog is illustrated in figure 3.21. The dedicated Dialog properties are listed in table 3.13.

AppWindow: *App Windows* are instances of the `AppWindow` class, subclass of `Context` and implementation of `IAppWindow`. An App Window is a proxy to the actual application window and can be used for assigning a main menu (Menu Bar or Menu Strip) and Layouts. It also defines a Context for all contained Widgets, except Window and Dialog instances. This class shares similarities with the Window class, although there is no inheritance or direct relation between these two. The App Window is technically no Widget at all, but it offers a handy way for customizing the app (seen as windowed application) similarly to modal windows (i.e., Window instances). The most common use case for App Windows are GUI-based applications rather than games or other kinds of 3D-driven applications. The properties defined by App Window are listed in table 3.14.

Name	Type	Description
`menuBarId`	string	The Node ID of the Menu Bar to use as main menu. After initialization, the Menu Bar can be set or queried via the Menu Bar Node Target of the Window.
`title`	string	The text to be displayed as title in the title bar.
`animationResourceId`	string	The name of the animation resource to use for both opening and closing the Window.
`openAnimationResourceId`	string	The name of the animation resource to use for opening the Window. If set, this property is prioritized over `animationResourceId`.
`openingAnimationStartTime`	real	The start time in seconds of the Window opening animation defined in the animation resource.
`openingAnimationEndTime`	real	The end time in seconds of the Window opening animation defined in the animation resource.
`closingAnimationResourceId`	string	The name of the animation resource to use for closing the Window. If set, this property is prioritized over `animationResourceId`.
`closingAnimationStartTime`	real	The start time in seconds of the Window closing animation defined in the animation resource.
`closingAnimationEndTime`	real	The end time in seconds of the Window closing animation defined in the animation resource.

Table 3.12: Window properties

Name	Type	Description
`text`	string	The text to be displayed as Dialog message.

Table 3.13: Dialog properties

3.4.6 Buttons And Controls

This section will cover the most important kind of Widgets regarding user interaction: *Buttons* and *Controls*. Buttons and Controls are derived from *Stateful Component*, which itself is a Component enriched by four states of user interaction. The states are defined by the ME itself and are part of the native Button node, which is a vital element of the Component node composed by the MGT. The following states are available according to the API documentation of the ME [Spraylight GmbH 2014d]:

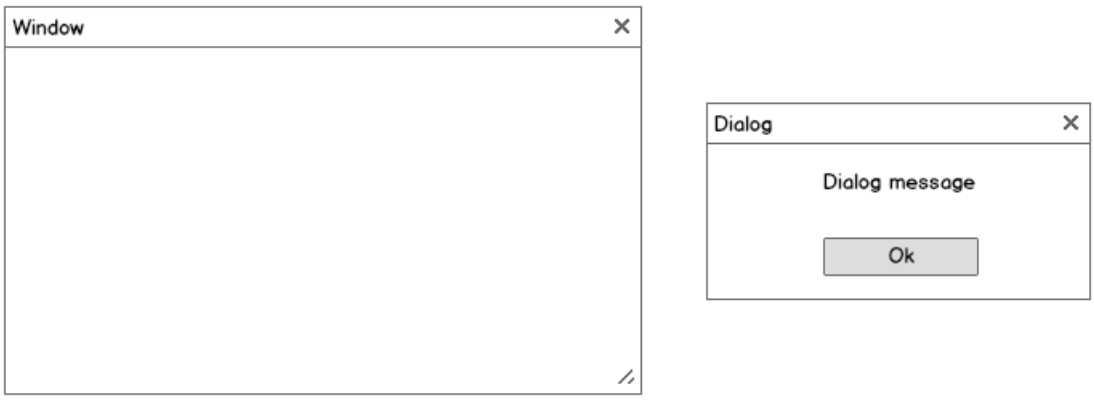

Figure 3.21: Window and Dialog

Windows

Name	Type	Description
`backgroundAtlasName`	string	The name of the atlas resource used for building a generic, patched background geometry. If not set, a plane geometry will be generated instead.
`backgroundStateSetId`	string	The Node ID of the state set to use for rendering the background plane.
`layoutId`	string	The Node ID of the Layout to use for arranging contained Components. After initialization, the Layout can be set or queried via the Layout Node Target of the App Window.
`menuId`	string	The Node ID of the Menu (Menu Bar or Menu Strip) to use as main or popup menu. After initialization, the Menu can be set or queried via the Menu Node Target of the App Window.

Table 3.14: App Window properties

- A `Button` is in *Up* state, if it is not disabled and no event is tracked within the Button's boundaries.
- A `Button` is in *Down* state, if it is not disabled and there is a tracked event within the Button's boundaries.
- A `Button` is in *Hover* state, if it is not disabled and there is an active event but no tracked event within the boundaries, or if a tracked event moved outside the boundaries.
- A `Button` is in *Disabled* state, if this state was set explicitly.

States are primarily used for customizing the appearance or behavior of the geometry within the Button area. More sophisticated manipulations are also possible. From a more general point of view, the Button acts as a switch, which activates sub-graphs depending on the active state. The MGT utilizes this mechanism to activate certain states for widget rendering according to the Button state.

A closer look has to be taken on the Control class, since it acts as superclass for many basic interactive widgets. A Control is a Stateful Component that manages an Entity. The type of the Entity and the modality of how to present and manipulate data are specified by the subclasses. A Control sends an update to its Entity whenever the Control node detects user interaction (e.g., dragging a slider thumb), while processing logic. Subsequently, after the Entity has updated the raw data it holds, an event will be sent back to the Control node. The Control node receives the event and updates its display representation according to the Entity state.

Buttons: *Buttons* are instances of the `Button` class, subclass of `StatefulComponent` and implementation of `IButton`. A Button extends the Stateful Component by a Label and a suitable skin setup. The required event handling for Buttons is already inherited from Component and Widget, so this class adds only little functionality to the base classes. The supported properties of Buttons are listed in table 3.15. Figure 3.22 schematically shows the four Button states.

Name	Type	Description
`text`	string	The initial value of the nested Label.

Table 3.15: Button properties

Figure 3.22: Button states

A Button is a rectangular plane with a Label in it. The presentation is different for each state of user interaction.

List Views and List Items: *List Views* are instances of the `ListView` class, subclass of `Control` and implementation of `IListView`. *List Items* are instances of the `ListItem` class, subclass of `Control` and implementation of `IListItem`. List Items are Controls for displaying the serialized Entity data as text. The Entity type itself does not matter. List Views make use of Scroll Containers to provide a vertically arranged, scrollable list of List Items. The List View refers to a Selection instance to handle the selection of List Items. List Views do not have any specific properties. The List Item properties are outlined in table 3.16. Figure 3.23 schematically illustrates List Views and List Items.

Name	Type	Description
`title`	string	The optional title to display instead of the serialized Entity data.

Table 3.16: List Item properties

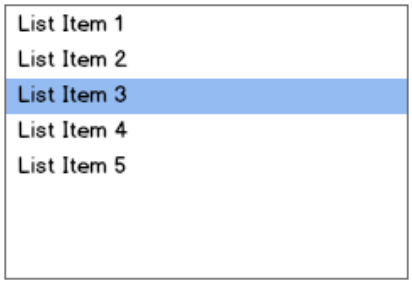

Figure 3.23: List View with List Items

List Views vertically arrange List Items in a Scroll Container.

Option Buttons: An implementation of the `IOptionButton` interface is provided by the `OptionButton` class, which is a subclass of `Control`. An Option Button allows the user to select exactly one option from a list of options by clicking/touching the Widget. Therefore, an Option Button always needs to be used together with other instances of the same type, forming a group that has at most one Option Button selected. In most widget toolkits, there is a dedicated widget for grouping radio buttons (see chapter 2). The MGT, however, uses a different approach by utilizing Entities for this purpose. The Entity type used for Option Buttons is `ISelection`, similar to List Views. Each Option Button belonging to the same option group must refer to the same Selection, which is usually created by the `OptionButton` Node instance defined first, $\mathbf{O}_0$. Subsequent Option Buttons $\mathbf{O}_{1...n}$ just refer to $\mathbf{O}_0$ to get access to the Selection. The according property is described in table 3.17, which provides an overview of all available Option Button properties. Figure 3.24 illustrates Option Buttons with the appearance of radio buttons.

Name	Type	Description
`sharedSelectionControlId`	string	The Node ID of the Control that will share its Selection with the current Option Button, instead of creating a new one. Only available as XML attribute. Once initialized, no other Selection can be assigned.

Table 3.17: Option Button properties

Figure 3.24: Option Buttons

Option Buttons without customization look like radio buttons known from other toolkits.

Progress Indicators: An implementation of the `IProgressIndicator` interface is provided by the `ProgressBar` class, which is a subclass of `Control`. *Progress Indicators* are Widgets visualizing the relative value of a Number Entity within its limits normalized to $[0, 1]$. This interface does not specify how the output of the implementation should look like. The *Progress Bar* Widget of the MGT follows the idiom of figure 3.25. The supported properties are listed in table 3.18.

Name	Type	Description
`value`	real	The initial value of the Number Entity. After initialization, the value is only accessible through the Number Entity.
`sharedEntityControlId`	string	The Node ID of the Control that will share its Number Entity with the current Progress Indicator, instead of creating a new one. Only available as XML attribute. Once initialized, no other Entity can be assigned.

Table 3.18: Progress Indicator properties

Figure 3.25: Progress Indicator (Progress Bar)

A Progress Bar is just an example implementation of a Progress Indicator, displaying a horizontal bar with progress indication. Other implementations with different output results (e.g., a round progress meter) are possible and independent from logic.

Sliders: *Sliders* are instances of the `Slider` class, subclass of `Control` and implementation of `ISlider`. They provide an continuous input method for numbers, managed by a Number Entity. A Slider consists of a track, a thumb, and two optional step buttons, one on each side (left and right on horizontal Sliders, top and bottom on vertical Sliders) of the track, to increment or decrement the number.

There are many possibilities to customize the appearance of a Slider, even track and thumb can both be styled independently. The MGT itself takes advantage of this: The scrollbars of the Scroll Containers are customized Sliders with step buttons and an adopted appearance. Table 3.19 shows a full list of properties made available by the Slider widget. Please note that the properties regarding the internal value (`value`, `maximumValue`, and `minimumValue`) are not values stored by the Slider object itself, but are passed to the Number Entity during Node initialization. Possible results can be seen in figure 3.26.

Name	Type	Description
`value`	real	The initial value of the Number Entity.
`maximumValue`	real	The maximum value of the Number Entity.
`minimumValue`	real	The minimum value of the Number Entity.
`sharedEntityControlId`	string	The Node ID of the Control that will share its Number Entity with the current Slider, instead of creating a new one.
`orientation`	enum	The orientation of the Slider, either `HORIZONTAL` (left-to-right) or `VERTICAL` (top-down).
`enableStepButtons`	boolean	If true, the Slider will have a decrement button on the left and an increment on the right side.
`relativeThumbLength`	real	The length of the thumb geometry relative to the track.
`thumbAtlasName`	string	The name of the atlas resource used to create a generic thumb geometry.
`thumbStateSetId`	string	The Node Id of the state set used for skinning the thumb geometry.
`trackAtlasName`	string	The name of the atlas resource used to create a generic track geometry.
`trackStateSetId`	string	The Node Id of the state set used for skinning the track geometry.

Table 3.19: Slider properties

Steppers and Text Fields: *Steppers* are instances of the `Stepper` class, subclass of `InputField` and implementation of `IStepper`. *Text Fields* are instances of the `TextField` class, subclass of `InputField` and implementation of `ITextField`. Both Controls are keyboard input fields, abstracted to `InputField`. While Text Fields can receive any input, Steppers will only accept numeric characters. A Stepper handles discrete numeric values, in contrast to the continuous method of the Slider. By default, Steppers use floating point numbers. Forcing integer values can be done by setting the integral property. Additionally, Steppers have two buttons to increment or decrement the number by the step defined in the Number Entity (default is 1).

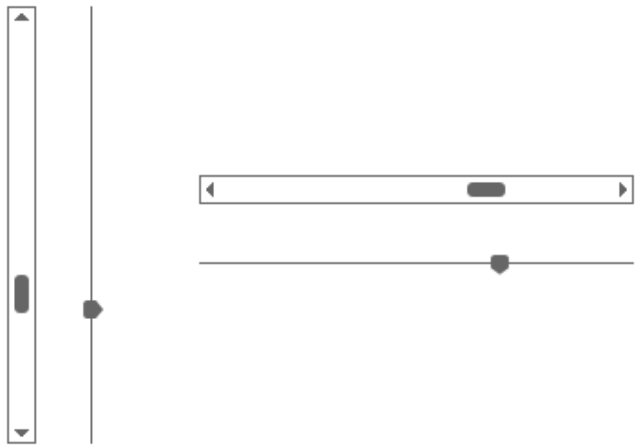

Figure 3.26: Sliders

Sending input characters or pressing the step buttons will immediately update the Control's Entity. Text Fields support secure flags to prevent the current value to be revealed on display. This is commonly used in password input fields. A scheme of how both Widgets look like is shown in figure 3.27. Tables 3.20 and 3.21 outline their supported properties.

Name	Type	Description
`value`	real	The initial value of the Number Entity.
`integral`	boolean	If true, the number is represented by an integer type, thus not supporting floating point values.
`sharedEntityControlId`	string	The Node ID of the Control that will share its Number Entity with the current Stepper, instead of creating a new one.

Table 3.20: Stepper properties

Name	Type	Description
`text`	string	The initial text of the Text Entity.
`secure`	boolean	If true, the characters of the input text are replaced by a placeholder.
`sharedEntityControlId`	string	The Node ID of the Control that will share its Text Entity with the current Text Field, instead of creating a new one.

Table 3.21: Text Field properties

Figure 3.27: Input Fields (Text Field and Stepper)

Switches: A *Switch* is a Widget that represents a boolean value which is toggled on user interaction. Implementations of the associated `ISwitch` interface are provided by the `CheckSwitch` and the `SlideSwitch` classes, both are subclasses of `Control`. A switch can be presented by different metaphors, each depending on the device class and the application itself. Traditionally, switches are realized by checkboxes. The advent of touch devices brought the slideable switch, with `UISwitch` of Cocoa Touch being the most famous example, later followed by the `ToggleButton` of Android. Another metaphor, used in many software audio mixers, is the toggle switch. Nevertheless, the paradigm of the switch remains the same in all three examples. This nature is abstracted by the `ISwitch` interface. The MGT includes two different implementations of `ISwitch` to demonstrate how a single UI paradigm (and a single interface) can be reused for different widgets. A `CheckSwitch` imitates the checkbox widget, while a `SlideSwitch` is composed by a thumb that moves along a track, inspired by the switches used on touch devices. Figure 3.28 shows schematics of both Widgets. The list of properties required by the interface is outlined in table 3.22.

Name	Type	Description
`activated`	boolean	If true, the initial state of the Switch Entity is set to “on”.
`sharedEntityControlId`	string	The Node ID of the Control that will share its Switch Entity with the current Switch, instead of creating a new one.

Table 3.22: Switch properties

Figure 3.28: Switch (Check Switch and Slide Switch)

On the left side is an example of both active and inactive Check Switches. The Slide Switches on the right side have the same internal state on their Switch Entity as their Check Switch counterpart.

Table Views, Table Rows, and Table Cells: *Table Views* are instances of the `TableView` class, subclass of `Component` and implementation of `ITableView`. Table Views are a rudimentary implementation of editable data matrices. Be aware not to confuse them with the sophisticated list views provided by Cocoa Touch, called `UITableView`. The MGT Table Views are two-dimensional, so they require rows and cells to be defined. The immediate child nodes of Table View are Table Rows, at which one instance represents one row.

Name	Type	Description
`numberOfRows`	integer	The number of Table Rows. The array of Table Row instances will be expanded or trimmed according to the difference between its size and this parameter.
`numberOfColumns`	integer	The number of Table Cells per Table Row. The array of Table Cell instances will be expanded or trimmed according to the difference between its size and this parameter.
`columnWidth`	real	The uniform width of all columns in the Table View.
`columnWidths`	array	The individual widths for each column in the Table View. If set, the uniform width is ignored.
`columnWidthsAreRelative`	boolean	If true, the uniform or individual width values are interpreted as width relative to the Table View width.

Table 3.23: Table View properties

Table Rows are instances of the `TableRow` class, subclass of `Node` and implementation of `ITableRow`. They serve a structural purpose as parent nodes of Table Cells.

Name	Type	Description
`data`	string	A string representing the data held by the Table Cell.
`isEditable`	boolean	If true, the data string may be edited by the user.

Table 3.24: Table Cell properties

Table Cells are instances of the `TableCell` class, subclass of `Node` and implementation of `ITableCell`. One Table Cell represents one concrete cell in the Table View, identified by its row and column indices. Table Cells provide access to the data string stored by them.

Neither Table Rows nor Table Cells are rendered directly. Instead, this is done by the Table View, which creates all the geometry necessary to render all cells including their content. If the Table Cell has the editable flag set, an Input Field Widget will be created automatically inside the cell to manipulate its data. Besides, Table Views make use of Scroll Containers to allow an arbitrary number of rows or columns to be defined independently of the Table View size. The available properties of Table Views and Table Cells are listed in tables 3.23 and 3.24.[10] Figure 3.29 illustrates Table Views schematically.

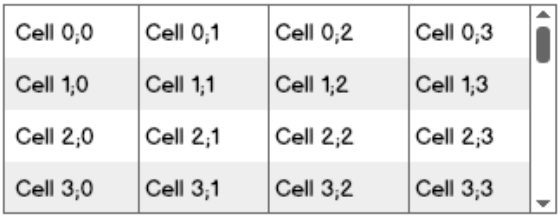

Figure 3.29: Table View

3.4.7 Other Widgets

Activity Indicators: *Activity Indicators* are instances of the `ActivityIndicator` class, subclass of `Component` and implementation of `IActivityIndicator`. Their main purpose is to tell the user to wait until an ongoing, time-consuming process is finished. There is no logic behind this Widget. The icon and the animation are customizable resources of the Widget. The default skin includes the image of a circle and an animation that rotates the object for an infinite number of times, or until the user code explicitly stops the animation. Table 3.25 outlines the available properties of this Widget.

Name	Type	Description
`geometryId`	string	The Node ID of the geometry to display as indicator icon.
`animationResourceName`	string	The name of the animation resource to animate the indicator icon.
`animating`	boolean	If true, the Activity Indicator is animating.

Table 3.25: Activity Indicator properties

[10] Table Rows do not have any properties.

Labels: *Labels* are instances of the `Label` class, subclass of `Component` and implementation of `ILabel`. They are used to display custom texts. Labels are just wrappers of a Node type already included in the ME – `TextGeometry` [Spraylight GmbH 2014e, see] – allowing text-textured plane geometries to be treated as Components, making them accessible for Layouts. The available properties for this Widget are outlined in table 3.26.

Name	Type	Description
`text`	string	The text to render.
`systemFontName`	string	The name of the system font to use for rendering the text.
`fontSize`	real	The point size of the rendered text.
`textColor`	vector	The color of the rendered text.
`backgroundColor`	vector	The background color of the rendered text.
`horizontalAlignment`	enum	The horizontal alignment of the text within the label boundaries.
`verticalAlignment`	enum	The vertical alignment of the text within the label boundaries.

Table 3.26: Label properties

3.5 Layouts

3.5.1 Basic Idea

Layout management is done by Layout nodes. These are dedicated classes derived from the common scene graph Node. Like the Layout Managers available in the Java GUI toolkits, different kinds of layouts are implemented in different classes.

3.5.2 Null Layout

Null Layouts are used by Containers on default, thus, if no other Layout is targeted. A Null Layout simply grabs the intrinsic position properties of a Component to use them as local position within the Container. There is no further logic when using them in this way. This may seem quite useless, but some more advanced options are available due the Null Layout Directives.

Using Relative Positions: Positions defined in Directives will overwrite the Component's intrinsic position properties. This makes sense, if a given position shall be interpreted as relative position, which can be done by setting an appropriate flag. A *relative position* $P_R(\mathbf{C}, \mathbf{P}) = (x_R, y_R) \in \mathbb{R}^2$ of a Component $\mathbf{C}$ with respect to its parent Container $\mathbf{P}$ maps to an absolute position $P_A(\mathbf{C})$ using the following computation:

Figure 3.30: Absolute and relative coordinates with Null Layout.

Container A uses a Null Layout with absolute coordinates $(-160, 160)$ for Component 1 and $(160, -160)$ for Component 2. Container B uses a Null Layout with relative coordinates $(-1, 1)$ for Component 1 and $(1, -1)$ for Component 2.

$$P_A(\mathbf{C}) = \left(\frac{x_R}{2} \cdot (w(\mathbf{P}) - w(\mathbf{C})), \frac{y_R}{2} \cdot (h(\mathbf{P}) - h(\mathbf{C}))\right)$$

Since $(0, 0)$ is the center of a Container, relative positions within the vertical and horizontal boundaries must be in $[-1, 1]$, with $(-1, -1)$ being the lower-left corner and $(1, 1)$, the upper-right. An absolute position of a Component is the local position of its center point in the Container. Thus, as in the formula above, we also must consider the size of the Component itself to prevent it from being placed (fully or partially) outside the Container boundaries. By subtracting the Component size from the Container size, the Null Layout achieves a corner-by-corner or edge-by-edge arrangement, if the relative position is -1 or 1. Someone who wants to place Components outside the boundaries by intention may accomplish this by using relative positions beyond $[-1, 1]$. Figure 3.30 illustrates the usage of both absolute and relative coordinates.

Keep in mind that although the relative positions of two Components in the same Container might be equal ($P_R(\mathbf{C_1}, \mathbf{P}) = P_R(\mathbf{C_2}, \mathbf{P})$), the absolute positions $P_A(\mathbf{C_1})$ and $P_A(\mathbf{C_2})$ are not equal if their sizes $S(C_1) \neq S(C_2)$.

Resize Components to Fill the Container: Another property of the Null Layout Directive is the *fill mode*. The fill mode specifies which dimension of the Component will be resized to exactly fit the Container's dimension. The available options are *horizontal*, *vertical*, *both*, and *none*. The default value is none, meaning that a Component will not be resized. Figure 3.31 shows the different results when varying the parameter.

3.5.3 Flow Layout

A simple but popular layout mechanism is provided by *Flow Layouts*. The Flow Layout of the MGT is capable of both arranging Components in a horizontal and in a vertical flow direction. Depending on the flow direction, Components are placed one after another beneath or below each other.

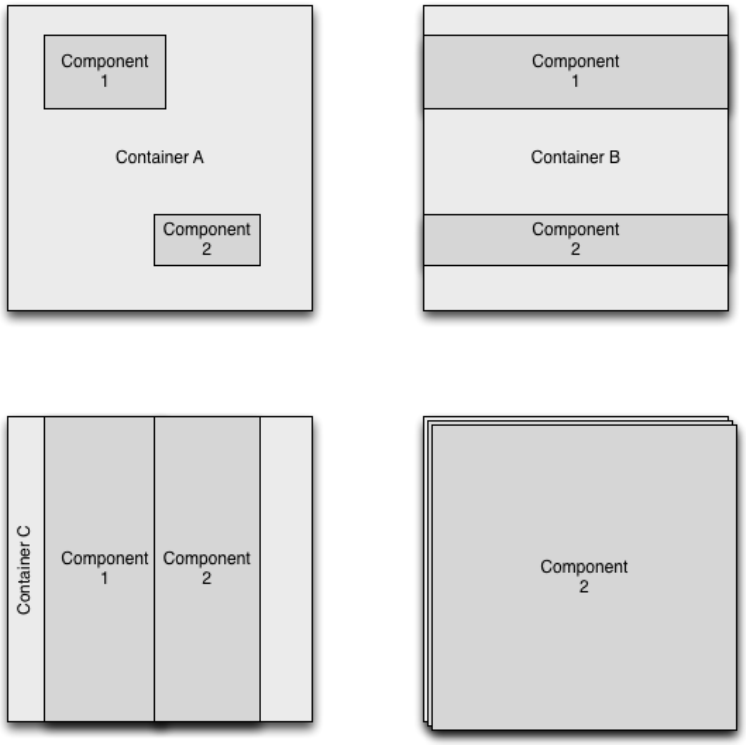

Figure 3.31: Fill modes supported by the Null Layout

The effect of each fill mode supported the Null Layout can be seen in the following examples: none (Container A), horizontal (Container B), vertical (Container C), and both (Container D, bottom right).

Customizing Flow Layouts: The vertical and horizontal gap spacings between Components are customizable properties of a Flow Layout. Figure 3.32 shows the result of both Flow Layouts with and without spacing.

Controlling the Flow Behavior: As mentioned before, the flow direction is a property of the Flow Layout and within the developer's control. Another option is called *auto-wrapping* which is enabled by default. If the Container's boundary will be exceeded by a Component to layout, auto-wrapping causes the flow to continue arranging Components on the next "row" (horizontal flow) or "column" (vertical flow). Each flow is then bounded by one dimension (e.g., the horizontal flow is bounded to the Container's width), so the Flow Layout will arrange Components towards infinity of the contrary dimension as shown in figure 3.33. If auto-wrapping is disabled, the flow arranges Components towards the infinity of the corresponding dimension.

Despite to any other Layout, there is no actual use for Flow Layout Directives. The node implementation is provided for reasons of completeness, though.

3.5.4 Grid Layout

Another atomic layout is the well-known *Grid Layout*. It defines a grid which splits the container into a two-dimensional field of cells addressed by column and row

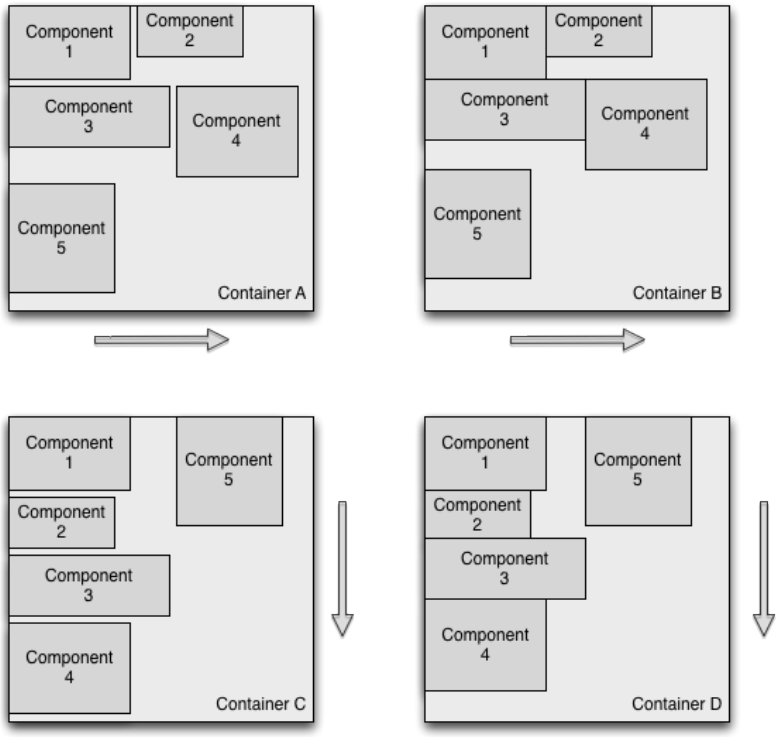

Figure 3.32: Flow Layout spacing

The flow of Container A and B is horizontal, Container C and D use vertical flowing. The effects of spacing a spacing value greater than zero can be seen in Container A and C, while Container B and D show the results of zero-spacing.

indices. The Layout fits each Components into a dedicated cell. If the number of Components exceeds the number of cells (which is columns times rows), the Components carried over will not be considered. The number of columns and rows may be changed any time, the whole layout will be recomputed, though.

Customizing Grid Layouts: Similar to Null Layouts, *fill modes* are also available for Grid Layouts. Again, the available modes are *horizontal*, *vertical*, *both*, or *none*, but for Grid Layouts they indicate which dimension(s) will be resized to fit the Component's cell, rather than fitting the whole container. Component dimension which will not be resized may cause the Component to overlap with Components of neighbor cells. Figure 3.34 illustrates the effects of all available fill modes.

Controlling the Cell Assignment: Components are assigned to cells from top to bottom and from left to right in the sequence they appear as child nodes of the Container in the scene graph. The developer can assign Components to cells manually by using *Grid Layout Directives*. These Directives offer properties for setting the column and row index of the Component that needs to be laid out.

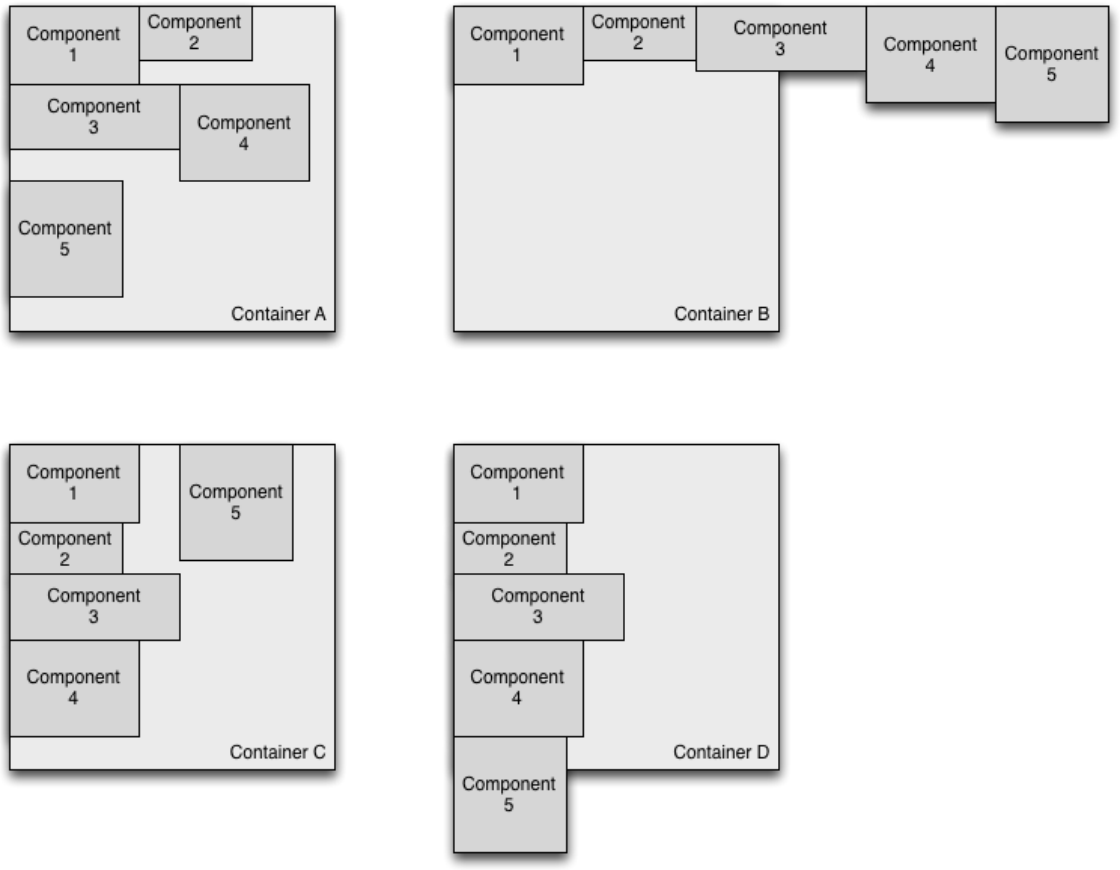

Figure 3.33: Flow Layouts in both directions with and without auto-wrapping.

The flow of Container A and B is horizontal, Container C and D use vertical flowing. The effects of auto-wrapping can be seen in Container A and C, while Container B and D show the results of auto-wrapping turned off.

3.5.5 Page Layout

Page Layouts follow a more practical approach and are essential for designing GUIs with a more sophisticated layout. They are similar to Border Layouts, which are available in Swing or many other toolkits. A Page Layout splits the Container into up to five distinct sections, which are similar to the main sections of web page. The sectioning may also be compared to the sections of any other kind of document. Figure 3.35 outlines the five pre-defined sections of a Page Layout.

Each section is optional and may be omitted. However, it is not possible to layout more than five Widgets in total or more than one Widget for the same section. This restriction keeps the logic of the Page Layout simple and efficient. Putting more than one Component into a single section can be achieved by using a Container as sectioned Component. Using this strategy is the first step in composing complex GUIs with sophisticated layouts. In many cases, top-level containers use Page Layouts to layout sub-containers. The sub-containers themselves use other layouts depending on their needs.

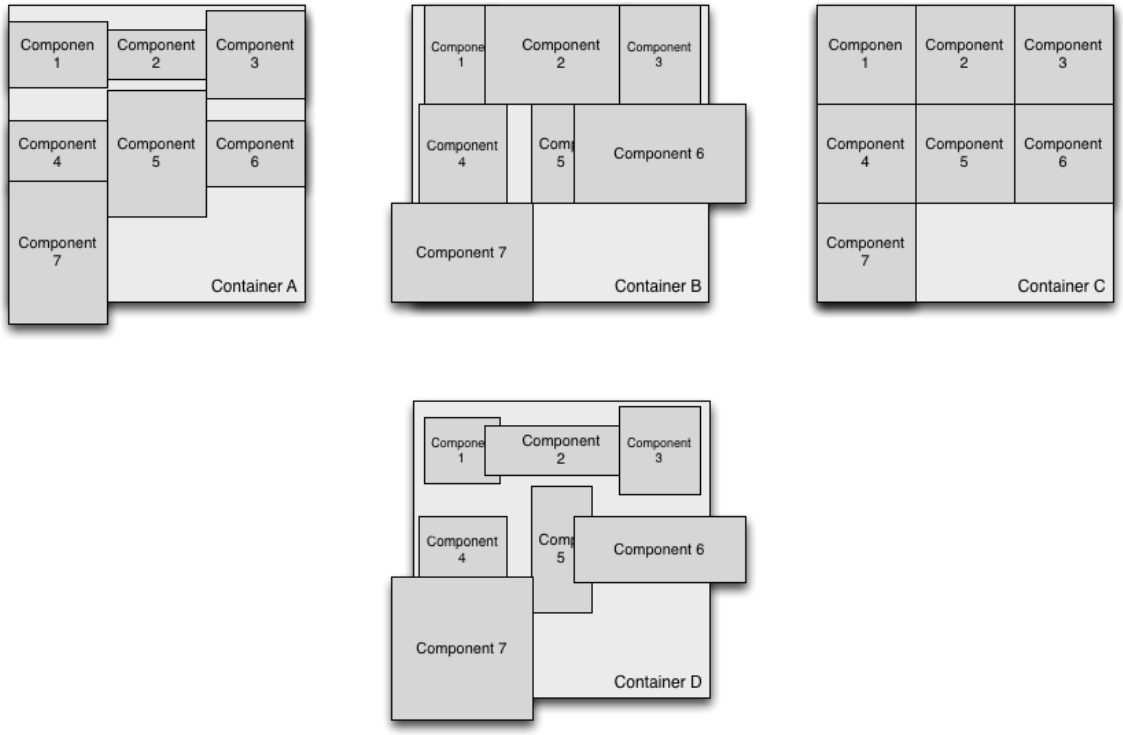

Figure 3.34: Grid Layout fill modes

Horizontal filling sets each Component's width to the width of the respective cell (Container A). Vertical filling sets each Component's height to the height of the respective cell (Container B). It is also possible to combine both filling modes (Container C) or use neither of them (Container D).

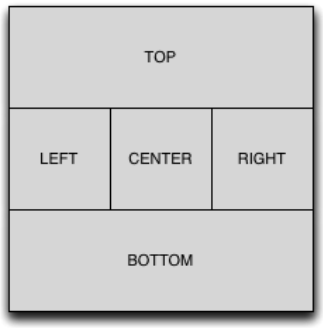

Figure 3.35: Page Layout sections

Customizing Page Layouts: The scale factors of both dimensions of each section can be customized by setting each scale factor s_C relative to the Container size s, whereas $s_C \in [0, 1]$ and $s \in \mathbb{R}$. Scale factors do not refer to explicit sections but to a grid-like scheme which defines 3 rows and 3 columns, as shown in figure 3.36. The actual size for each section is computed by the according scale factors of each dimension and the absolute Container size.

If a section will not be used for any Component, a gap that fits the section space will be visible. Gaps can be closed by setting the proper scale factor to 0. The

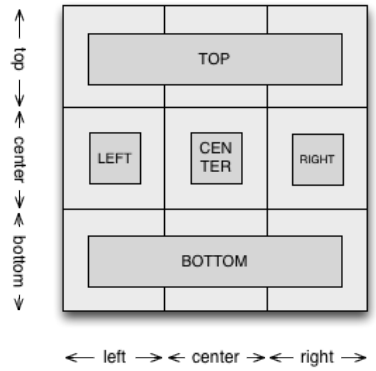

Figure 3.36: Page Layout scale factors

The scale factors of the Page Layout refer to a virtual grid scheme. The relative width of the top and the bottom sections is the sum over all horizontal scale factors (left, center, right).

remaining scale factor(s), however, shall sum up to 1, otherwise the Layout will not fill the whole container. The following setup defines a Page Layout which only consists of a left and a right section:

$$V = \{0, 1, 0\}$$

$$H = \{1, 0, 1\}$$

More examples are depicted in figure 3.37. From here on, many variations of complex layouts can be generated by using this technique in combination with nested, laid out Containers (e.g., a Container with a Page Layout inside another Container with another Page Layout).

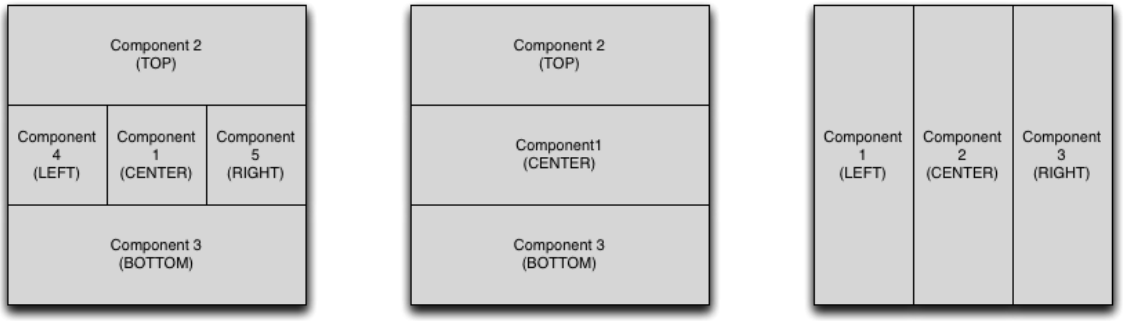

Figure 3.37: Page Layout scaled sections

The first example Container (left) uses a Page Layout with default configuration. The second Container is the result of setting the horizontal scale factors to $H = \{0, 1, 0\}$, causing the left and right sections to disappear and giving the center section full width. The third Container is generated from the vertical scale factors $V = \{0, 1, 0\}$, which hides the top and bottom sections and stretches the others to full height.

Controlling the Section Assignment: To assign an arbitrary Component to a specific section, a developer can use *Page Layout Directives* to do so by setting their section property. The Page Layout will consider this property, when computing the layout of a Component. If no Directive was defined, the Components will be sectioned depending on their child index within the Container graph in the following order: center, top, bottom, left, right. As an example, the third Component defined will therefore be arranged in the bottom section.

3.6 Skinning

Skinning is a key feature for applications to customize the GUI elements according to the designer's specifications and can be found in nearly every GUI toolkit introduced in the previous chapter. This section introduces the skinning concept of the MGT.

3.6.1 Basic idea

Implementing skinning in an 3D engine based on OpenGL simply means providing an abstraction of setting up materials and textures used for rendering the Widget geometry (usually a plane). There are many ways for building a scene graph that applies materials and textures on geometries. The ME defines some node classes for activating materials, parameters, and textures defined previously somewhere in the scene graph: `MaterialState`, `ParametersState`, and `TextureState`. As the name already suggests, these node types overwrite the current OpenGL state, causing following-up geometry to be rendered with the properties these nodes refer to. The recommended texturing practice in scene graph engines is to define materials and textures once as loose nodes that are visited during an early traversal state. Whenever they will be used, a `Reference` Node must be set before rendering the designated geometry (see section 1.4.3 for an explanation on References).

The MGT follows this approach, but goes one step further: It bundles the state nodes into a new empty node that entirely describes the appearance of a certain widget. This nodes are called *State Sets* and are defined in a dedicated *Skin package* that needs to be included by the application on the initialization process. State Sets are just instances of the `Node` class and only contain state manipulation nodes. The identifier of a State Set (needed for References to find the targeted Node) is similar to the class name of the Widget and automatically referenced within the Widget's subgraph. Widgets with more than one geometry (like a `Slider`, which consists of a track and a thumb) often use more than one State Set. A dot-notation will then be used to group and specify the State Sets of the individual elements a Widget consists of. For example, a `Slider` uses the Skins named `/Gui/Skin/Slider.Horizontal.Thumb` and `/Gui/Skin/Slider.Horizontal.Track`. Table 3.27 shows a list of all Nodes that need to be supplied by a Skin package.

State Set Node ID	Description
ActivityIndicator	Activity Indicator icon
AppWindow	App background
Button.Up	Up-state Button
Button.Down	Down-state Button
Button.Hover	Hover-state Button
Button.Disabled	Disabled-state Button
CheckSwitch.Up	Up-state Check Switch
CheckSwitch.Down	Down-state Check Switch
CheckSwitch.Hover	Hover-state Check Switch
CheckSwitch.Disabled	Disabled-state Check Switch
CollapseContainer	Collapse Cont. content background
CollapseContainer.Control	Collapse Cont. control bar backgr.
Container.Patch	Container patched geom. backgr.
InputField	Text Field / Stepper background
ListItem	List Item background
ListItem.Selection	Selected List Item background
ListView	List View background
MenuBar.Background	Menu Bar background
MenuBar.SelectedItem	Selected Menu Item background
MenuStrip.Background	Menu Strip background
MenuStrip.Foreground.Up	Up-state Menu Item in Menu Strip
MenuStrip.Foreground.Hover	Hover-state Menu Item in M.Strip
MenuStrip.Foreground.Separator	Separator Menu Item in M.Strip
OptionButton.Up	Up-state Option Button
OptionButton.Down	Down-state Option Button
OptionButton.Hover	Hover-state Option Button
OptionButton.Disabled	Disabled-state Option Button
ProgressBar.Background	Progress Bar background
ProgressBar.Gauge	Progress Bar gauge
ScrollContainer.Horizontal.Thumb	Horizontal scrollbar thumb
ScrollContainer.Horizontal.Track	Horizontal scrollbar track
ScrollContainer.Vertical.Thumb	Vertical scrollbar thumb
ScrollContainer.Vertical.Track	Vertical scrollbar track
Slider.Horizontal.Thumb	Horizontal Slider thumb
Slider.Horizontal.Track	Horizontal Slider track
Slider.Vertical.Thumb	Horizontal Slider thumb
Slider.Vertical.Track	Horizontal Slider track
Stepper.DecrementButton	Stepper decrement button
Stepper.IncrementButton	Stepper increment button
TabControl.TabButton.Background	Not-selected tab button
TabControl.TabButton.Foreground.Inner	Selected inner tab button
TabControl.TabButton.Foreground.Left	Selected left-most tab button
TabControl.TabButton.Foreground.Right	Selected right-most tab button
TabPage	Tab Page background
Window.Frame	Window frame with title bar
Window.Buttons	Top-right window control buttons
Window.ResizeButton	Bottom-right resize dragger

Table 3.27: State Set Nodes required in the /Gui/Skin namespace

Applying a State Set to a Widget can be done the following ways:

1. Defining a complete Skin package which contains all Nodes using the identifiers of table 3.27.

2. Defining a single State Set and assign its ID to the Widget's State Set ID property.

(1) is used whenever the designer creates a whole new theme for the app that differs from the default skin in a large part. There is no need to update attributes of Widgets in the scene graph XML or in the C++ code. They "know" the appropriate State Sets by the given identifiers as mentioned above. However, sometimes instances of the same kind of Components may require different skins (where it does not matter which Skin package is actually used). This can be fulfilled by method (2). Components define a State Set ID attribute to overwrite the default State Set an instance refers to. For example, a designer may want to use three buttons, each in a different color.

3.6.2 Elements of a Skin

Geometries

For greater flexibility in sizing Widgets, we must also consider their geometry, even if they appear to be flat. It is easy to create materials and textures for fixed-size Widgets such as `OptionButton`s or `CheckSwitch`es by just preparing an image texture that fits exactly into the geometry. But using the same approach in Widgets with flexible geometry (e.g., `Button`s) will result in an image scaled by OpenGL to fit into the plane. Widgets larger than the texture will have their borders and other features blurred, while in smaller Widgets the borders and features get squeezed. Depending on the design that will actually be used for skinning, the MGT provides three different kind of geometry types to set up a Component with.

Plane Geometry: A `PlaneGeometry` is a node of the ME for rendering double-faced rectangular planes. They are described by their position and extent.

Plane geometries are suitable for fixed-size Widgets with an arbitrary skin. For example, if the designer creates a fancy button image with 300×120 pixels, a plane geometry of the same size is sufficient.[11] But plane geometries can also be used with any materials insensitive to scaling, like plain diffuse colors or gradients. Plain colors without texture are becoming popular GUI styles on mobile devices (and maybe also on desktops) nowadays due to the new design guidelines of the main manufacturers.

Generic Geometry: If plane geometries do not satisfy the designer's needs, so-called *generic geometries* can help out. A `GenericGeometry` is a node class part of the ME that manages a custom geometry defined by the developer. The developer

[11] Keep in mind that the texture must be padded to fit a power of two dimension. Therefore, it is also necessary to set the texture coordinates of the rectangle vertices properly, which is done by the MGT.

must setup a *vertex buffer* and an *index buffer* explicitly. This kind of geometry was used in the MGT to implement a *patched geometry*. The idea of patched geometry was inspired by the `NinePatch` class which is part of the Android SDK. A nine-patch divides a texture into a 3×3 grid. Each sub-image created this way (nine in total) is defined by the texture coordinates of its vertices. A nine-patched geometry is a rectangular plane but subdivided by additional vertices and edges within the plane, also resulting in a 3×3 grid of cells. On creation or size update, each vertex position is calculated accordingly to the features the adjacent cells represents (e.g., the left bottom corner). This can be used to prevent blurring or squeezing of features represented by single pixels (like one-pixel-borders). Particular cells have one or two fixed dimensions equal to the cell dimensions of the texture image. Figure 3.38 illustrates this idea. The corner cells cannot be scaled in any direction, while edge cells can scale either horizontally or vertically. The center cell may be scaled both horizontally and vertically.

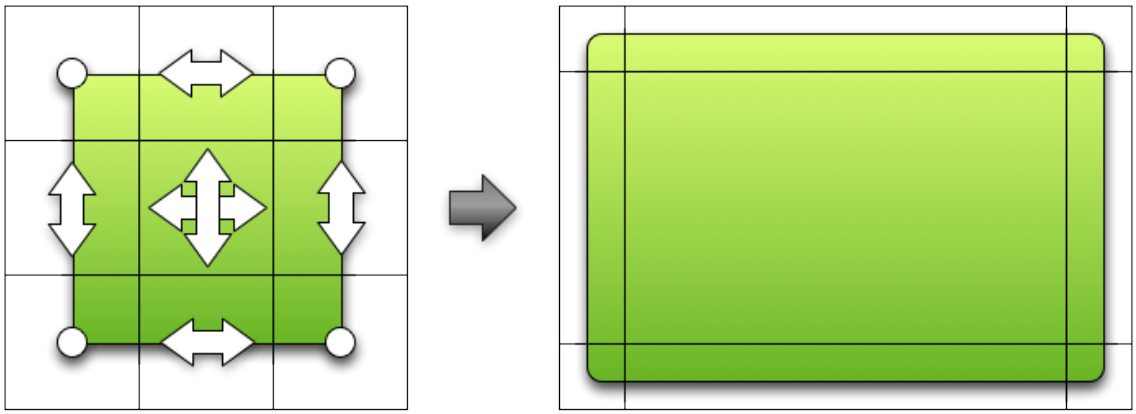

Figure 3.38: Nine Patch

The left plane is the input image divided into patches. The arrows within the cells indicate directions the cell may be stretched along. A dot means that the cell can not be stretched in any direction. The right plane represents a possible result.

Some variants of the nine-patch pattern exist in the MGT. The `Slider` widget for example uses a 3×1 patched geometry, with the first and the third cell defining both (fixed-size) ends of the track. Only the second cell will be scaled to the `Slider` dimension minus the size of the first and the third cell.

Referenced Plane Geometry: In many GUI themes, the same kind of widgets have similar appearance both in size and surface. This fact applies especially to checkboxes and radio buttons. There is simply no need for the designer to create multiple varieties, since there only exists a small and limited number of representations (e.g., the states on, off, disabled). In contrast to widgets like progress bars, which may differ in length, or buttons with different captions, a single pre-rendered image can be used for example to display all active checkboxes. The ME can utilize this circumstance to (1) reduce scene graph complexity and (2) to speedup asset creation. (1) is due to the fact that the same textured plane geometry (i.e., a flat image) used in different Widgets only needs to be defined once as a Node. The engine's

`Reference` or `ReferenceTransform` nodes can then be used to include[12] the image anywhere in the scene graph. In the previous example, instead of defining State Sets in the Skin package, like it needs to be done for the two geometry types introduced before, the Skin package only needs to provide a plane geometry node with material slot, texture slots, and texture coordinates already set. (2) is because the ME comes along with an atlas generator tool. The atlas generator creates a single texture image from a bunch of input images. The atlas corresponding to the compiled image defines regions within the texture to outline every single source image for texture mapping. Additionally, a graph containing plane geometry definitions, one for each input image, will be created. Reducing the number of textures by using atlases provides great performance improvements, especially on devices without NPOT[13] texture support in situations with many small textures. Texture atlases are a widely used technique, especially in two-dimensional tile-based rendering.

Materials

Material nodes are vital not only for skinning, but also for making Widgets visible, as they instruct the underlaying OpenGL API how to render the geometry. Because materials are a huge topic in OpenGL rendering, I will just focus on the most basic ways on how to create skins with materials, parameters, and textures.

A material in the ME uses a *Program* (i.e., an instance of the abstract `Program` node class), which is either a predefined program (`FixedProgram`, comparable to the almost outdated fixed function pipeline) or a program of vertex shaders and fragment shaders (`ShaderProgram`) [The Khronos Group 2014]. Fixed function programs are sufficient for rendering common GUI skins. However, many GUI effects are possible with shaders, although they might cause problems on mobile devices due to restrictions of OpenGL ES.

Plain materials: Skins made with plain materials are the simplest to create and to render. The most basic installment just uses a fixed program with coloring enabled and assigned to a simple material (see listing 3.1). The outcome is a white surface without any effects. As minimalistic designs become increasingly popular, this might already be sufficient for, e.g., the background of a Container.

```
1 <FixedProgram id="prg_white"/>
2 <Material id="mat_white" programId="prg_white"/>
```

Listing 3.1: A plain material for rendering white, unlit surfaces.

`FixedParameters` can be used in conjunction with a `FixedProgram` to set up custom ambient, diffuse, specular, or emissive color as well as shininess [Spraylight GmbH 2014c]. See listing 3.2 for an example:

[12] There is no actual inclusion of the referenced geometry though. References are resolved during graph traversal, causing the visitor to "jump" to the targeted node, traversing it again, and finally returning to the Reference node, continuing traversal straightforward.

[13] NPOT, non power of two

```
<FixedProgram id="prg_color" coloringEnabled="yes"/>
<Material id="mat_color" programId="prg_color"/>
<FixedParameters id="par_green" diffuseColor="0.0f, 1.0f, 0.0f, 1.0f"/>
```

Listing 3.2: A plain material for rendering colored, unlit surfaces. A `FixedParameters` node is used to specify the rendering parameters (i.e. surface colors).

Textured materials: Materials for texturing need a `FixedProgram` with at least one texture unit enabled. The texture itself is defined by a `FlatTexture` node which refers to an image resource included by the package. Blending can be configured by the attributes/properties of the `Material` node as shown in example 3.3.

```
<FixedProgram id="prg_texture0" textureUnit0Enabled="yes"/>
<Material id="mat_texture0+alpha" programId="prg_texture0" visibleFaces="
    FRONT_AND_BACK" blendMode="ALPHA" depthBufferMode="NONE"/>
```

Listing 3.3: A textured material with alpha blending and rendering enabled for both sides of the geometry.

Custom shaders: As mentioned before, the ME allows custom GLSL programs to be incorporated into a `ShaderProgram`, which requires a vertex shader and fragment shader. Parameterization of single Skin nodes can be done by preparing `Parameters` nodes and refer to them using `ParametersState`. If the `Material` uses the `ShaderProgram`, the developer is free to create any effect on the Widget through the shader programs. This can be used for more advanced rendering effects with or without textures and/or parameters. Some examples will be provided in the results chapter.

3.6.3 Definition and Integration of a Skin

State Set nodes are bundled into a dedicated Murl package – the Skin package.[14] State Sets can be used interchangeably, as long as they follow the conventions of the MGT. This includes the availability of a set of Nodes with certain identifiers. The required Node identifiers and their usage can be looked up in table 3.27. One or more graph resources are needed to define the Nodes, with each Node having the appropriate state nodes attached. Listing 3.4 shows an example of how to define a State Set node.

```
<Node id="Button.Up">
  <MaterialState materialId="TextureAlpha" slot="0"/>
  <TextureState textureId="Textures/Button.Up" slot="0" unit="0"/>
</Node>
```

Listing 3.4: Definition of Skin nodes in a custom package.

[14]Note that there are many ways to build a scene graph with a certain structure from XML resources. The way described here is considered to be the most modular and intuitive one.

The Nodes must not be part of any Namespace, otherwise the toolkit will not be able to find them. To hook the State Sets into the MGT, a special attribute, of which the ME is aware, is needed in the package definition file within the graph instantiation tag (see 3.5 for an example). This directive tells the deserializer not to instantiate the graph in-place, but to attach it to another (already existing) Node. That Node is a Namespace called /Gui/Skin and is defined by the MGT common package. After loading the Skin package, all Skin nodes are available within this namespace. Using this strategy, Skins can easily be exchanged by just including another Skin package. Whenever a Skin package is properly included, a Widget node will find its Skin nodes in /Gui/Skin. Including more than one Skin package will likely lead to an error, as the ME does not accept duplicate Node identifiers.

```
1 <Instance graphResourceId="skin_nodes" parentNodeId="/Gui/Skin"/>
```

Listing 3.5: Integration of Skin nodes into the toolkit.

3.7 Extension Concept

Extending the MGT is strongly tied to the possibilities of object-oriented inheritance. Thus, this section focuses on groups of classes which are easily extendible and modular in a sense of functional independence. The following types are meant to be extended:

- Widgets (IWidget) and Components (IComponent)
- Layouts (ILayout) and Layout Directives (ILayoutDirective)
- Entities (IEntity)

In fact, adding a new Widget, Layout, Layout Directive, or Entity can easily be done by just implementing the base interface. However, for productional use, the extension of existing implementations is only possible by obtaining a source code license. An implementation solely based on interfaces will suffer from the lack of access to many hidden implementation details. This section ignores those restrictions and assumes that the developer has full access to the source code, since the conception of license models is not part of the thesis.

3.7.1 Extending Widgets and Components

Component is a subclass of Widget as shown in figure 3.4. Due to the fact that a developer most likely wants to extend a Component class rather than Widget, this section focuses especially on Components. Adding a custom Component definition requires two steps:

1. Implementing a node class based on Component and
2. registering the node class through the GraphFactoryRegistry of the ME.

The `Component` class defines some protected methods that are called in certain situations during the Component's life-time. The Widget generated by the `Component` class itself is just a rectangular, interaction-aware area with a configurable appearance. Extending these properties requires the developer to override some of the protected methods of `Component` as they are introduced in section 4.2.3.

3.7.2 Extending Layouts

Layouts are nodes, thus adding new layouts consists of

1. implementing a node class based on `ILayout` and
2. registering the node class through the `GraphFactoryRegistry` of the ME.

The same steps must be done for Layout Directives by implementing the interface `ILayoutDirective` if the developer intents to provide such nodes.

The developer who implements a Layout must be aware of how they work internally. The following methods must be implemented and behave as described:

+ `LayoutComponent(component: IComponent, boundary: Dimension): Bool`
  This method layouts the Component within the given boundary by considering the internal layout state. It also needs to track the Component and update the internal layout state.
+ `LayoutComponent(component: IComponent, boundary: Dimension, directive: ILayoutDirective): Bool`
  This method layouts the Component within the given boundary by considering the given Layout Directive. It also needs to track the Component.
+ `UpdateLayout(boundary: Dimension): Bool`
  This method re-layouts all previously tracked Components within the given boundary. It is called automatically after the Container has been resized.

Layouts are expected to keep track of all Components that had been laid out so far in order to recalculate their new dimensions iteratively, after the Container size did change.

3.7.3 Extending Entities

New Entity types can be added by subclassing the public `Entity` class. If a source code license is available, extension can also be done by implementing the `IEntity` interface. The developer must then assure that the Entity Event is dispatched to the Event Pipeline, which cannot be done without access to the full source code. The `Entity` class has one pure virtual method named `GetSerializedData()`. The developer must implement this method by returning a string representation of the current Entity state. There are no additional requirements. The new Entity can

be used as handler for any kind of data. This may also include representations of database entities including full model management. The Core Data framework follows a similar strategy below its main concept, the object graph.

Extending concrete built-in Entity like `SwitchEntity` or `NumberEntity` is only possible with a source code license since their implementation is hidden.

4 Implementation

The previous chapter described the entire set of features and functionalities of the toolkit, deducted from the requirements the MGT must fulfill. The following sections will now explain some implementation strategies that were important to achieve the goal. This chapter focuses on major challenges in the context of writing a GUI toolkit for 3D engines, especially for the Murl Engine, rather than providing a complete documentation of the source code, which contains about 220 files and approximately 18.000 source lines of code (SLOC). The ME encourages a clean and consistent style of coding and problem solving. The MGT tries to adopt this style, thus many patterns were inspired directly by the engine itself.

4.1 Project Structure

The organization of the source files is kept flat and follows the conventions of the ME. The public interfaces are kept separated from the implementation code and their header files in the dedicated directory `./base/include` relative to the root directory of the engine. Implementation files can be found in `./base/source`. Therefore, the public interfaces are kept in `./base/include/gui`, while their implementation resides in `./base/source/gui`. In order to keep the header path configuration simple and transparent, no other subfolders were introduced.

This flat hierarchy has also been considered in the namespace policy. The ME uses the top-level namespace `Murl` which contains the common classes and interfaces. Submodules of the engine are grouped into second-level namespaces, for example scene graph related source code is found in `Murl::Graph`. The MGT defines its own namespace `Murl::Gui` where the complete implementation is found. During this chapter, all mentioned class types are prefixed by their second-level namespace to identify their affiliation. This is consistent to the names used in the implementation which omit the top-level namespace `Murl` as a result of the `using namespace` directive at the beginning of each C++ file. As an example, the Button widget of the MGT is implemented by `Gui::Button` (`Gui::IButton`), while the Button node of the engine is called `Graph::Button` (`Graph::IButton`).

The MGT requires two Murl packages to be loaded on startup: `gui_base` and `gui_skin`. The base package defines some common resources and scene graph nodes that are mandatory for the toolkit to run. The Skin package contains an example/template Skin that can be modified or replaced by another package that conforms to the requirements defined in section 3.6. Package contents are stored in folders named after the package and suffixed by the extension `.murlres`. For productional use, the packages must be compiled to the proprietary `.murlpkg` format.

Both resource folders and packages are stored in `./common/data/packages`. Thus, the following files/folders can be found there:

- `./common/data/packages/gui_base.murlres`
- `./common/data/packages/gui_base.murlpkg`
- `./common/data/packages/gui_skin.murlres`
- `./common/data/packages/gui_skin.murlpkg`

4.2 GUI Graph Nodes

The MGT takes advantage of the engine features regarding scene graph definition with XML resource files. This is why most features are encapsulated in graph nodes. As outlined in section 3.1, most classes provided by the toolkit are subclasses of the graph node base class `Graph::Node`. Exceptions are Entity classes and classes related to event handling. Before having a closer look on the Node implementations, some general annotations on the Node life-cycle of the ME have to be made.

4.2.1 Graph Node Implementation

Adding a graph node type based on `Graph::Node` to the repertory of the engines requires a few steps to be done. Although scene graphs can be built programmatically, the default way of doing this in the ME is defining scene graphs as an XML resource. Consequently, the registration of custom Node types for the XML parser is also a necessary step explained here. Furthermore, understanding the Node life-cycle is vital for adding custom behavior. The full process is exemplified by the `Gui::Widget` and `Gui::Button` nodes.

Extending Graph::Node: Custom Nodes must inherit from `Graph::Node`, a class that is publicly available in the ME. As already mentioned before, the implementation of classes in the MGT is hidden, therefore a toolkit node class must also implement a public interface that grants the developer adequate access to the object. All Node classes provided by the MGT will hence follow the declaration scheme of listing 4.1. Of course, in many cases they inherit from `Gui::Widget` (or its subclasses) instead directly from `Graph::Node`.

```
1  namespace Murl
2  {
3      namespace Gui
4      {
5          class Widget : public IWidget, public Graph::Node
6          {
7              static INode* Create(const Graph::IFactory* factory);
8
9              // member declaration
10         };
11     }
12 }
```

Listing 4.1: Example declaration of a node class.

Providing Factory Information: Creation of graph nodes is done by a factory object called Graph Factory (`Graph::IFactory`), which needs to know some common information about the Node it creates, especially when deserializing a graph from an XML resource. Declaration and definition of these information access methods is done by some macros provided by the ME. Listing 4.2 shows the declarational usage in the header file, embedded in the code listed in 4.1.

```
1 MURL_DECLARE_FACTORY_OBJECT_BEGIN(Gui::Widget)
2     MURL_DECLARE_FACTORY_OBJECT_PROPERTY(PROPERTY_DEPTH_ORDER)
3     MURL_DECLARE_FACTORY_OBJECT_PROPERTY(PROPERTY_TAG)
4 MURL_DECLARE_FACTORY_OBJECT_END(Gui::Widget)
```

Listing 4.2: Declaration of class information and properties.

Both the full, namespace'd class name and the bare class name will later be available as XML tag identifier.[1] The properties in listing 4.2 will be resolved as enum constants and correspond to attributes available in XML. Listing 4.3, located in the CPP file, shows the appropriate definition, which also includes the the mapping from XML attribute name strings to enum values. The factory uses them to deserialize the property that belongs to the XML attribute (see below).

```
1 MURL_DEFINE_FACTORY_OBJECT_BEGIN(Gui::Widget)
2     MURL_DEFINE_FACTORY_OBJECT_PROPERTY(PROPERTY_DEPTH_ORDER, "depthOrder")
3     MURL_DEFINE_FACTORY_OBJECT_PROPERTY(PROPERTY_TAG, "tag")
4 MURL_DEFINE_FACTORY_OBJECT_END(Gui::Widget)
```

Listing 4.3: Definition of XML attributes.

Registering Node Class: The `App::AppBase` class (i.e., the base class of the app's custom main class) defines two methods that are called on startup/termination to register/unregister custom node classes: `RegisterCustomFactoryClasses()` and `UnregisterCustomFactoryClasses()`. The developer who wants to add her own node types must override these methods and call the registration methods of the factory registry (`IAppFactoryRegistry`). This must be done for every node type the developer wants to register as shown in listing 4.4. `GetClassInfo()` was defined by the macros introduced in the previous paragraph.

```
1 Bool App::GuiTestApp::RegisterCustomFactoryClasses(IAppFactoryRegistry*
      factoryRegistry)
2 {
3     Graph::IFactoryRegistry* graphRegistry = factoryRegistry->
          GetGraphFactoryRegistry();
4
5     graphRegistry->RegisterNodeClass(Gui::ActivityIndicator::GetClassInfo());
6     graphRegistry->RegisterNodeClass(Gui::Button::GetClassInfo());
7     graphRegistry->RegisterNodeClass(Gui::CheckSwitch::GetClassInfo());
8     graphRegistry->RegisterNodeClass(Gui::CollapseContainer::GetClassInfo());
9
```

[1] As you may have noticed, `Gui::Widget` is not a valid XML tag name. However, the engine XML format is not compliant to the W3C XML standard. Instead the format has been slightly adopted in favor of the engine's features.

```
10      // [...]
11
12      return true;
13  }
```

Listing 4.4: Implementation of `RegisterCustomFactoryClasses()`

This, however, is not possible unless the developer owns the source code of the MGT. Furthermore, doing this for every single app is unreasonable overhead. Therefore, the class `Gui::Main` can be instantiated to handle registration/unregistration in a single call (see listing 4.5). The declaration of this class is available in a public header file, the implementation is hidden, allowing the node types to remain opaque.

```
1  Bool App::GuiTestApp::RegisterCustomFactoryClasses(IAppFactoryRegistry*
       factoryRegistry)
2  {
3      mGuiMain->RegisterGuiGraphClasses(factoryRegistry);
4      return true;
5  }
```

Listing 4.5: Implementation of `RegisterCustomFactoryClasses()` with GUI main class.

Deserializing Attributes: Listing 4.6 shows how XML attribute values are deserialized into Node object properties. This method is called for each attribute detected while parsing the markup. The tracker already contains methods for parsing string representations of common types, so only a reference to the targeted member variable needs to be passed.

```
1  Bool Gui::Widget::DeserializeBaseAttribute(Graph::IDeserializeAttributeTracker*
       tracker)
2  {
3      switch(tracker->GetBaseAttributeProperty(GetProperties()))
4      {
5          case PROPERTY_DEPTH_ORDER:
6              tracker->GetAttributeValue(mWidgetDepthOrder);
7              return true;
8
9          case PROPERTY_TAG:
10             tracker->GetAttributeValue(mEventTriggerTag);
11             return true;
12
13         default:
14             return Graph::Node::DeserializeBaseAttribute(tracker);
15     }
16 }
```

Listing 4.6: Implementation of `DeserializeBaseAttribute()`

Initializing the Node: The `OnInit()` method is a callback declared by the `Graph::INode` interface that gets called once for each node while initializing the graph. Initialization happens after deserialization (if any). The engine recursively initializes the graph nodes by calling `OnInit()`, which delegates this responsibility to

`OnInitSelf()` and on `OnInitChildren()`. The latter method carries on the recursion while `OnInitSelf()` is reserved for setting up the current node. Because nearly all Widgets are dynamically composed of other nodes to provide greater flexibility for a wide range of customization, initialization plays a major role in the toolkit. Widgets of type `Gui::Component` split up this phase into a few virtual methods, e.g., for referencing State Sets or for geometry creation. A Component subclass then overrides these methods rather than `InitSelf()` to run its specific setup. In many cases, the appearance setup of `Gui::Component` is sufficient and already customized by the properties, but there may still be some extra node required to add. Listing 4.7 shows the initialization of `Gui::Button`, which inherits the appearance and behavior of `Gui::Component`, but additionally needs to create a Label node to display text on a Button. Nodes can be dynamically created by calling `CreateNode()` on the `Graph::IRoot` instance provided during initialization. After configuration, the node must be added to the existing hierarchy by calling `AddChild()` on a node that is a qualified parent. When layering plane geometries, the geometry that lies "below" another is suitable. Here, the Label is put upon the *Component Geometry* node (the background plane of the Button). To ensure proper layering, the depth order of the geometry "above" another must be greater than zero.[2]

```
1  void Gui::Button::InitSelf(Graph::IInitTracker* tracker)
2  {
3      Gui::Component::InitSelf(tracker);
4
5      Graph::IRoot* root = tracker->GetRoot();
6
7      mButtonLabel = dynamic_cast<Gui::ILabel*>(root->CreateNode("Gui::Label"));
8
9      if(mButtonLabel != 0)
10     {
11         mButtonLabel->SetText(mButtonText);
12         mButtonLabel->SetHorizontalTextAlignment(::IEnums::
               HORIZONTAL_TEXT_ALIGNMENT_CENTER);
13         mButtonLabel->SetVerticalTextAlignment(::IEnums::
               VERTICAL_TEXT_ALIGNMENT_CENTER);
14         mButtonLabel->GetComponentInterface()->GetWidgetInterface()->
               SetDepthOrder(1);
15         mButtonLabel->GetComponentInterface()->SetPosition(Real(0), Real(0));
16         mButtonLabel->GetComponentInterface()->SetSize(mComponentInnerDimension.
               mSizeX, mComponentInnerDimension.mSizeY);
17
18         if(mComponentUsesFrameBuffer)
19         {
20             mButtonLabel->GetComponentInterface()->SetBuffered(false);
21         }
22
23         mComponentGeometry->GetNodeInterface()->AddChild(mButtonLabel->
               GetNodeInterface());
24     }
```

[2] In 2D rendering, depth ordering is a replacement for depth buffering. Explicit integer values are used to define the order of rendering, where a lower value means an earlier draw call. Objects drawn later will overlay the existing output and result in a layer effect. A depth order of zero means that the transformation has the same depth order as its parent node. Like other transformation properties, depth orders also affect child nodes.

```
25 }
```

Listing 4.7: Example of a virtual `SetupGeometry()` implementation, called from `Gui::Component::InitSelf()`

Processing Node Logic: The method `ProcessLogicSelf()` is declared as part of the `Graph::INode` interface and is called on each tick to update the internal state of the node. Widgets that are aware of user interaction are likely to override this method, at least Components and all its subclasses do so to handle Drag and Drop (if enabled). There are some user input events though that are easier to handle in a later period of the frame (see next paragraph). In most cases, `ProcessLogicSelf()` is used to handle drag gestures like in `Gui::Slider` or `Gui::Window`. More details will follow in the upcoming sections.

Finishing Node Logic: The finish logic phase happens after traversing the entire scene graph for logic processing. At this point, all nodes have been updated. The `FinishLogic()` method will then be called for nodes which explicitly signed up for this step in the preceded traversal. This is where the Event objects are created and dispatched, regardless of when the actual event has been detected. For example, the flags indicating press or release events during the most recent tick on a `Graph::Button` instance are not set until the logic traversal is finished, other than the drag movement of a Slider that will be fully parsed during the Node's logic process.

De-initializing the Node: Destruction of a graph leads to the destruction of all attached Nodes. This phase is called "de-initializing" and is handled in `DeInitSelf()`. Since a Node of the MGT is a composition of other Nodes that were created during initialization, the created child Nodes must finally get destroyed again as a consequence of proper memory management. This is done bottom up by detaching them one by one from their parent before calling the `DestroyNode()` method as it is done in listing 4.8.

```
1 Bool Gui::Button::DeInitSelf(Graph::IDeInitTracker* tracker)
2 {
3     mComponentGeometry->GetNodeInterface()->RemoveChild(mButtonLabel->
          GetNodeInterface());
4
5     tracker->GetRoot()->DestroyNode(mButtonLabel->GetNodeInterface());
6
7     return Gui::Component::DeInitSelf(tracker);
8 }
```

Listing 4.8: Implementation of `DeInitSelf()`

This concludes the description on how custom node types are integrated in the scene graph system of the ME. The points mentioned here were all considered during

the implementation of the classes covered in the following sections, although it might not be mentioned explicitly.

4.2.2 Widget Nodes

Widget nodes are derived from their base class `Gui::Widget`, as shown in figure 3.4. This means that all properties and behaviors described here will also apply to all other Widget types among this chapter. `Gui::Widget` is subclass of `Graph::Node`, which makes all Widgets of the MGT to node types available for building scene graphs in the ME. Like other nodes, the Widget's active (for logic traversal) and visible (for rendering traversal) flags can be manipulated. They are also fully affected by the transformations and state passes through the graph. In regard to the MGT functionality, `Gui::Widget` provides registration methods for Event Handlers of the global input events that were delegated to the Widget by its Context (see section 3.1.2). Widgets usually do not only dispatch Events on global input events to Event Handlers, but also handle these events by themselves. The focus flag, for example, which is defined by `Gui::Widget`, will be set on reaction to a Focus Event, as shown in listing 4.9. `HandleKeyboardEvent()` and `HandleWheelEvent()` are also implemented, but with an empty body, since there is no default behavior among all Widgets.

```
1  Bool Gui::Widget::HandleFocusEvent(Gui::IEvent::ConstSharedPtr event)
2  {
3      Gui::IFocusEvent::ConstSharedPtr focusEvent = Gui::IFocusEvent::
           ConstSharedPtr::DynamicCast(event);
4      mWidgetIsFocused = focusEvent->IsReceiverOfFocus(this);
5      return true;
6  }
```

Listing 4.9: Implementation of `Gui::Widget::HandleFocusEvent()`

Another important feature implemented by this base class is the Context handling. The method `FindRespondingContext()` can be used to traverse the graph bottom up (beginning from the current Widget node) to find the first implementation of `Gui::IContext`. Alternatively, if the Widget is not within the subgraph of a `Gui::Context` node, the *Widget Context Node Target* can be set explicitly. The full implementation is available in listing 4.10. Through dynamic casting, parent Widget nodes are detected to invoke a recursive call on them. If a parent is of type `Gui::IContext`, it will be returned as result, otherwise it will be traversed at the end of the ongoing iterations. 0 will be returned if no Context has been found.

```
1  Gui::IContext* Gui::Widget::FindRespondingContext()
2  {
3      if(mWidgetContextNodeTarget.GetNumberOfNodes() != 0)
4      {
5          return mWidgetContextNodeTarget.GetNode(0);
6      }
7      else
8      {
9          Graph::INodeArray parents = GetParents();
10         for(SInt32 index = 0; index < parents.GetCount(); index++)
11         {
```

```
12              Graph::INode* parent = parents[index];
13              Gui::IWidget* widget = dynamic_cast<Gui::IWidget*>(parent);
14              if(widget != 0)
15              {
16                  return widget->FindRespondingContext();
17              }
18              else
19              {
20                  Gui::IContext* context = dynamic_cast<Gui::IContext*>(parent);
21                  if(context != 0)
22                  {
23                      return context;
24                  }
25                  else
26                  {
27                      parents.Add(parent->GetParents());
28                  }
29              }
30          }
31          return 0;
32      }
33  }
```

Listing 4.10: Implementation of `Gui::Widget::FindRespondingContext()`

The responding Context is primarily used in another important method. `PushContext()` tells the responding Context to activate itself and "push" itself in front of all other Contexts. `PushContext()` is called whenever an unfocused Widget receives an user input event (most probably local input events). As a result, it will also receive the focus. This is a well-known behavior, e.g., from input fields: the user first clicks into an input field to focus and then uses the keyboard to write characters into it. If the window of the input field was in the background, it will also have been pushed to front. Listing 4.11 contains the MGT implementation of this pattern. Dialogs can set the `lockContext` flag to prevent other Contexts from pushing until the Dialog is closed (see below).

```
1  void Gui::Widget::PushContext(Bool lockContext)
2  {
3      Gui::IContext* context = FindRespondingContext();
4      if (context != 0)
5      {
6          if (!context->IsActive())
7          {
8              context->PushActive();
9              context->SetContextLocked(lockContext);
10         }
11         if (!mWidgetIsFocused && IsFocusable())
12         {
13             context->Focus(this);
14         }
15     }
16 }
```

Listing 4.11: Implementation of `Gui::Widget::PushContext()`

`Gui::Widget` does not contain any renderable geometry or material definitions. This is part of the responsibility of the Widgets derived from this class.

Menu Bars and Menu Items

A Menu Bar acts as container to Menu Items. Menu Items are pure informational nodes that do not contain any geometries to render. An instance of `Gui::MenuBar` therefore takes this information from the `Gui::MenuItem` and generates the appropriate node to create the desired outcome. In the scene graph XML resource, Menu Items are simply defined as child elements of Menu Bars. The first step on initialization is the registration of these nodes to track them as Menu Items. This is done by iterating over all immediate child nodes and check their type by doing dynamic casts as shown in listing 4.12.

```
1 Graph::INodeArray children = GetChildren();
2 for(SInt32 index = 0; index < children.GetCount(); index++)
3 {
4     Gui::IMenuItem* menuItem = dynamic_cast<Gui::IMenuItem*>(children[index]);
5     if(menuItem != 0)
6     {
7         mMenuItems.Add(menuItem);
8     }
9 }
```

Listing 4.12: Registration of all `Gui::MenuItem` nodes attached as child nodes.

Later on, the geometry for the Menu Bar background will be created, as well as the nodes for rendering every single Menu Item. Figure 4.1 shows an example subgraph of an initialized Menu Bar with two Menu Items.

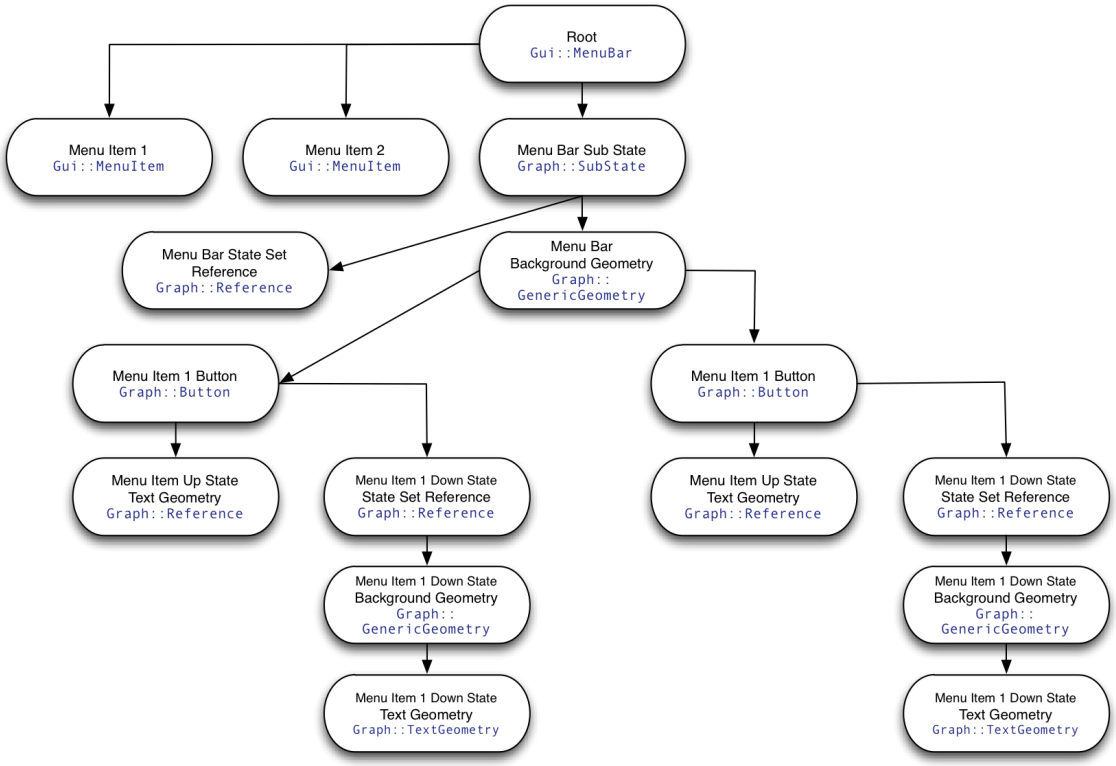

Figure 4.1: Menu Bar Subgraph

Menu Bars are also responsible for parsing pointing device input by querying the button nodes (e.g. *Menu Item 1 Button* in the figure above) iteratively on each tick. Menu Events will then be created. If the Menu Strip Node Target is set for a particular Menu Item, the referred Menu Strip will be displayed beneath the Menu Item button.

One limitation of Menu Bars is the fact that there is currently no possibility to find out the width of a rendered text through the engine. Therefore, all Menu Items are assumed to be equally wide (60 virtual pixels). Menu Items with longer label texts will be trimmed, smaller texts will look like padded text. This issue will also affect other Widgets, as it will be learned in the following sections of this chapter. A feature to tackle this problem has been staged after finalizing the implementation though.

Menu Strips

Like `Gui::MenuBar`, `Gui::MenuStrip` is also an implementation of `Gui::IMenu`, but in contrast to Menu Bars, Menu Strips arrange Menu Items in a vertical manner. Though, Menu Strips have much in common with Menu Bars in terms of event handling. They mostly differ in the way they are used. Menu Strips replace Menu Bars on mobile devices. Additionally, they can be hierarchically structured by attaching other Menu Strips to contained Menu Items. For the proper effect, Menu Strips must be defined as child nodes of an inactive/invisible node in the scene graph. Menu Item use `Graph::ReferenceTransform` nodes to present Menu Strips as submenus at the correct position. If the Menu Strip shall be rendered and processed, the reference node will be set active and visible, otherwise inactive and invisible. This is a common practice propagated by the ME: A complex node is defined within an inactive and invisible branch of the scene graph before its first use. Reference nodes are then used to render the complex node whenever and wherever they are needed.

4.2.3 Component Nodes

`Gui::Component` extends `Gui::Widget` by a specific scheme of subgraphs. As most Widgets are Component derivates, this class defines the common subset of features used by them. As stated in the previous chapter, (1) a Component is a Widget with rectangular dimensions and a position. (2) They have a custom appearance rendered onto their geometry. And finally, (3) Components are aware of local input events. The base class `Gui::Component` provides default setup and event parsing methods to assist derived Widgets in these three aspects. Component nodes themselves are already very versatile Widgets for rendering custom output and doing basic event handling. Before explaining the special adaptions of Components in their subclasses, a closer look on the base class shall be given here. Note that the most substantial topic regarding Components – Skinning – is covered in section 4.5 due to the fact that skinning was formulated as one of the major requirements the MGT must satisfy and therefore deserves its own section.

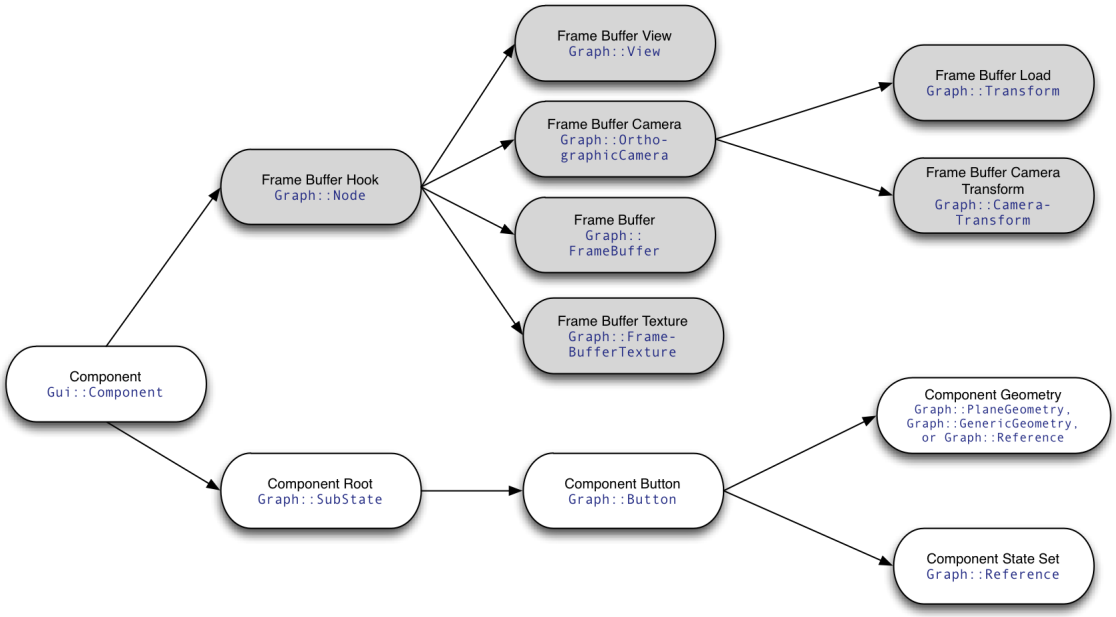

Figure 4.2: Component Subgraph

The subgraph shown in figure 4.2 is generated by `InitSelf()` on initialization of the Component node. The gray elements are optional and depend on the frame buffer settings (enabled or disabled) of the Component. Using frame buffers for Component means that the output geometry is not rendered directly to screen but into a frame buffer texture that is put on a simple plane geometry. This produces additional cost in time and memory performance for Components that only consist of a texture and a simple plane geometry. However, performance improvement can be achieved on Widgets with many sub-elements, e.g., composed Widgets. Instead of drawing all planes on each tick, the output is once written into the frame buffer for one tick before the entire subgraph will be set to invisible until the next update of the appearance. Depending on the complexity of the GUI and the Widgets, this may reduce the number of processed vertices from dozens, hundreds, or thousands down to four. The developer is responsible for proper handling this feature. It can be turned on or off by the method `SetFrameBufferEnabled()`.

The other nodes of the Component subgraph make up the default setup. Since many derived Widgets refer to those in order to query or manipulating certain properties, their usage shall be described as follows:

- *Component Root* is a `Graph::SubState` node for sandboxing all state manipulations (probably quite a lot) within the Component subgraph. This ensures that the nodes outside the Component are not affected by any states of the subgraph. It must therefore be used as the root node of all other nodes.
- *Component Button* defines the input area of pointing devices. It has usually the same height and width as the bounding box of the contained geometry (see below). If the geometry is a plane geometry or a multi-patch geometry, the dimensions of button, bounding box, and geometry are equal and thus always updated together. `Graph::Button` nodes are subclassed from

`Graph::Transform`, which gives them a second important role: The *Component Button* holds the transformation of the Component, that depends on the X and Y position properties manipulated by users or by layouts.

- *Component State Set* is a reference to a State Set node defined in the skin package.
- *Component Geometry* refers to the node that holds the Widget geometry to render. Its node type is not specified, as there are three possible candidates per default (from the three geometry types as introduced in section 3.6.2) and any number of arbitrary geometries theoretically. The surface of the geometry is determined by the State Set (see above). Plane and generic geometries are affected by size updates. Furthermore, if Components contain any decorative child geometries or child Widgets, they are attached to this node.

The basic structure is adopted by child classes according to their needs. To provide a consistent interface for all Component derivates in order to built a proper subgraph, some callback methods are defined that may be partially or fully overridden by child classes.

\# `ApplyPositionUpdate()`
A callback method that is called after the position properties of a Component changed. If the Component has already been initialized, this method propagates the updated values to the transform node of the Component (*Component Button*).

\# `ApplySizeUpdate()`
A callback method that is called after the size properties of a Component changed. If the Component has already been initialized, this method propagates the updated size values to the *Component Geometry* (as long as it is not an instance of `Graph::Reference`). Finally, `ReCalculateInnerDimension()` and `ReLayoutChildren()` will be executed, allowing subclasses to update their custom geometries.

\# `GetDefaultComponentGeometryType(): ComponentGeometryType`
Returns the default/desired geometry type (see section 3.6.2) of the Component.

\# `GetEffectiveComponentGeometryType(): ComponentGeometryType`
Returns the actual geometry type to use for the Component. This allows the Component subclass to force a certain type by ignoring the default or custom value. By default, this method just returns the current value of the related property.

\# `ReCalculateInnerDimension()`
A callback method that is called after the size of a Container changed. This method can be used to calculate a new inner dimension according to the updated outer dimension and the Component-specific insets. If not overridden, this method just assigns the outer dimension to the inner dimension.

`ReLayoutChildren()`
A callback method that is called after the size of a Component changed. This method can be used to update additional geometry or Widgets attached to the Component. For example, a Window uses this method to update the position of the title bar buttons.

`SetupDimension(tracker: Graph::IInitTracker): Bool`
A callback method that is called once on initialization to setup the dimension of the Component geometry. This method can be used to ensure proper size values and to calculate size and position of child geometries.

`SetupStateSet(tracker: Graph::IInitTracker): Bool`
A callback method that is called once on initialization to setup the Component State Set. This method can be used by subclasses to create even more complex State Sets or materials if references to skin packages are not sufficient.

`SetupFrameBuffer(tracker: Graph::IInitTracker): Bool`
A callback method that is called once on initialization to setup the Component frame buffer. If frame buffering is enabled for the Component, this method can be used to customize the frame buffer nodes. However, there is hardly any use for this opportunity.

`SetupGeometry(tracker: Graph::IInitTracker): Bool`
A callback method that is called once on initialization to setup the Component geometry. This method can be used to add additional geometries or to setup the base geometry apart from the default rectangle.

Another important task of `Gui::Component` is handling local input events and dispatching Event objects. This is done in `FinishLogic()`, as otherwise the input events would not be parsed yet by `Graph::Button`. If there is only a single tracked event on the *Component Button*, the algorithm reads the events flag from the button (e.g., `WasPressed()`). It is also possible that no certain point event type (press, release inside, or release outside) was detected, which is the case if the pointing instrument is currently dragging. Even then, Event objects must be dispatched – as someone might want to know the current drag position. There is no distinction between device types yet. If there is more than one tracked event (multitouch), the algorithm must keep track of every single "tracked event ID" and event position. The ID list of the previous tick will then be compared with the current one. If there are new entries, the event type of the Events to dispatch must be set to `PRESS`. On the other side, if event IDs are missing in the newer list, `RELEASE_INSIDE` or `RELEASE_OUTSIDE` (depending on the positions) must be assigned instead. Again, there will probably be events without any of these types. They will just propagate their drag positions. Finally, if all tracked events have been parsed, the collected data will be put into an instance of `Gui::TouchEvent` and additionally abstracted to `Gui::PointEvent`. All created Event objects will then be dispatched iteratively.

As a last point, Components are involved in the Drag-and-Drop logic of the MGT. Their role is to handle dragging, while their counterparts, Containers, are watching

for “dropped” Components. Details on this feature will be covered in a separate section at 4.2.7.

Activity Indicators

The Component subclass `Gui::ActivityIndicator` restricts the geometry type to `COMPONENT_GEOMETRY_TYPE_REFERENCE` and extends the `SetupGeometry()` method by interposing a `Graph::Timeline` and `Graph::Transform` node between the *Component Geometry* and its parent. The result of this configuration is an animating activity icon. The icon is obtained from the skin atlas, where it is defined as textured `Graph::PlaneGeometry` node, so only a reference has to be created. Listing 4.13 furthermore demonstrates the dynamic configuration of keyframe animation on nodes.

```
1  void Gui::ActivityIndicator::SetupGeometry(Graph::IInitTracker* tracker)
2  {
3      Gui::Component::SetupGeometry(tracker);
4
5      Graph::IRoot* root = tracker->GetRoot();
6      Graph::INode* hook = mComponentGeometry->GetParent(0);
7
8      // Rebuild graph after default geometry setup.
9
10     hook->RemoveChild(mComponentGeometry);
11
12     mIndicatorTimeline = dynamic_cast<Graph::ITimeline*>(root->CreateNode("
           Timeline"));
13     mIndicatorTimeline->SetStartTime(Real(0));
14     mIndicatorTimeline->SetEndTime(Real(1));
15     mIndicatorTimeline->SetStartOnActivateEnabled(true);
16     mIndicatorTimeline->SetNumberOfLoops(-1);
17     mIndicatorTimeline->GetNodeInterface()->SetActive(mIndicatorAnimating);
18     hook->AddChild(mIndicatorTimeline->GetNodeInterface());
19
20     mIndicatorTransform = dynamic_cast<Graph::ITransform*>(root->CreateNode("
           Transform"));
21     mIndicatorTransform->GetNodeInterface()->AddChild(mComponentGeometry);
22     mIndicatorTimeline->GetNodeInterface()->AddChild(mIndicatorTransform->
           GetNodeInterface());
23
24     Graph::IController* controller = tracker->GetGraphFactory()->CreateController
           ("AnimationController", mIndicatorTransform->GetNodeInterface());
25     mIndicatorAnimationController = dynamic_cast<Graph::IAnimationController*>(
           controller);
26     mIndicatorAnimationController->GetAnimationResourceTarget()->AddResourceId(
           MURL_GUI_ACTIVITY_INDICATOR_ANIMATION_RESOURCE_ID);
27     mIndicatorTransform->GetNodeInterface()->AddController(
           mIndicatorAnimationController->GetControllerInterface());
28 }
```

Listing 4.13: Implementation of `Gui::ActivityIndicator::SetupGeometry()`

A few words on how to animate scenes in the ME need to be given now. Animations are stored as XML files in the proprietary format of the ME. An example is available in listing 4.14. This file contains a sequence of animation keys, each with an exact keyframe time and the value(s) of the key at the given time. The values between two consecutive keyframes are interpolated linearly by default (like in the example), but it is also possible to explicitly set the interpolation function. The

Activity Indicator wants a counter-clockwise rotation around the z-axis, which is the default viewing direction used in the MGT.

```
<?xml version="1.0" ?>

<Animation>

    <RotationKey time="0.0" axisX="0" axisY="0" axisZ="-1" angle="0deg"/>
    <RotationKey time="1.0" axisX="0" axisY="0" axisZ="-1" angle="360deg"/>

</Animation>
```

Listing 4.14: Animation resource of the Activity Indicator.

To make animations come to life, a `Gui::Timeline` node must be set as parent of the subgraph that shall be animated. Timelines have a definite start and end time and further properties like time scale and loop configuration. After starting, the timeline updates its time value every frame by the time passed since the most recent tick. Afterwards, it updates all descendant nodes with instances of `Graph::AnimationController` assigned. Nodes with controllers must also specify the animation resource to use. The controller then updates the properties of the node according to the keys of the animation file and the current time of the timeline. The example above affects the rotation of transformable nodes. The Activity Indicator therefore needs to create a `Graph::Transform` node with an animation controller as parent of the icon geometry and attach it to the timeline to achieve the desired effect. By setting the number of loops to -1, the timeline will loop infinitely or until it becomes inactive.

Labels

Labels are rendered as plane geometries textured by `Graph::FlatTextTexture` nodes. No other geometry types are allowed or possible, since `Gui::Label` overrides the effective geometry type getter of `Gui::Component`. `Graph::FlatTextTexture` instances are dynamically generated texture nodes which contain the rendering output of customizable text. The surface is created by the operating system on the CPU, before being copied to the video memory. This has to be considered when creating many Labels or long text paragraphs. Another drawback will probably occur on devices without NPOT texture support because text textures must be padded from the actual text size to the next power of two size in both dimensions.

As extension of `Gui::Component`, the setup of Labels is mainly done in the virtual method `SetupStateSet()`. This is where the text texture is created and configured. Listing 4.15 reveals that most properties are taken from `Gui::Label` which can be seen as MGT compliant wrapper class for text textures. The reason for a dedicated Label node was the compatibility with the Layout handlers and the Component/-Container concept of the MGT.

```
void Gui::Label::SetupStateSet(Graph::IInitTracker *tracker)
{
    Graph::IRoot* root = tracker->GetRoot();
```

```
4
5       mLabelTextureSizeX = Gui::CalculateNextPowerOfTwo(mComponentOuterDimension.
            mSizeX);
6       mLabelTextureSizeY = Gui::CalculateNextPowerOfTwo(mComponentOuterDimension.
            mSizeY);
7
8       // create texture
9
10      mLabelTextTexture = dynamic_cast<Graph::ITextTexture*>(root->CreateNode("
            FlatTextTexture"));
11      mLabelTextTexture->GetTextureInterface()->SetPixelFormat(::IEnums::
            PIXEL_FORMAT_R8_G8_B8_A8);
12      mLabelTextTexture->GetTextureInterface()->SetAlphaEnabled(true);
13      mLabelTextTexture->GetTextureInterface()->SetSlot(0);
14      mLabelTextTexture->GetTextureInterface()->SetUnit(0);
15      mLabelTextTexture->GetTextureInterface()->SetHorizontalWrapMode(::IEnums::
            TEXTURE_WRAP_MODE_CLAMP_TO_EDGE);
16      mLabelTextTexture->GetTextureInterface()->SetVerticalWrapMode(::IEnums::
            TEXTURE_WRAP_MODE_CLAMP_TO_EDGE);
17      mLabelTextTexture->SetText(mText);
18      mLabelTextTexture->SetTextColor(mTextColor);
19      mLabelTextTexture->SetBackgroundColor(mBackgroundColor);
20      mLabelTextTexture->SetFontId(mSystemFontName);
21      mLabelTextTexture->SetFontSize(mFontSize);
22      mLabelTextTexture->SetHorizontalTextAlignment(mHorizontalTextAlignment);
23      mLabelTextTexture->SetVerticalTextAlignment(mVerticalTextAlignment);
24      AddChild(mLabelTextTexture->GetNodeInterface());
25
26      // Add texture to widget graph.
27
28      mComponentStateSet = Gui::CreateTextureMaterialState(root, mLabelTextTexture
            ->GetTextureInterface());
29
30      Component::SetupStateSet(tracker);
31  }
```

Listing 4.15: Implementation of `Gui::Label::SetupStateSet()`

Also note the private utility functions `Gui::CalculateNextPowerOfTwo()` and `Gui::CreateTextureMaterialState()` used here and in some other parts of the toolkit. `Gui::CalculateNextPowerOfTwo()` is a one-liner that rounds up an arbitrary number k to the smallest power of two number $\geq k$. Example: $f(75) = 128$

```
1   UInt32 Gui::CalculateNextPowerOfTwo(Real number)
2   {
3       return UInt32(Math::Pow(2.0f, Math::Ceil(Math::Log(number) / MURL_MATH_LN_2))
            );
4   }
```

Listing 4.16: Implementation of `Gui::CalculateNextPowerOfTwo()`

`Gui::CreateTextureMaterialState()` is used to create a graph that instantiates a fixed program, a material, a material state, and a texture state that refers to the texture passed as parameter. The resulting graph can then be used as State Set for the Widget and is independent from skin package definitions. However, even though the overhead is negligible, the same function and material nodes will be instantiated over and over again.

Finally, some calculations on texture coordinates are necessary for rendering correct text output. Depending on the horizontal and vertical text alignment, the plane

size, and the texture size, the texture coordinates of the plane geometry must be calculated by taking these factors into account. The method `ReCalculateTextureCoordinates()` will be called on initialization and whenever the Label properties did change. u_1, v_1, u_2, and v_2 are calculated using the following formulas, where w_t and h_t denote the width and height of the texture, and $w_\mathbf{L}$ and $h_\mathbf{L}$ the width and height of the Label:

$$u_1 = \begin{cases} \frac{w_t - w_\mathbf{L}}{2} & \text{if horizontal alignment is } \textit{center} \\ 0 & \text{if horizontal alignment is } \textit{left} \\ w_t - w_\mathbf{L} & \text{if horizontal alignment is } \textit{right} \end{cases}$$

$$u_2 = \begin{cases} w_t - u_1 & \text{if horizontal alignment is } \textit{center} \\ w_\mathbf{L} & \text{if horizontal alignment is } \textit{left} \\ w_t & \text{if horizontal alignment is } \textit{right} \end{cases}$$

$$v_1 = \begin{cases} \frac{h_t - h_\mathbf{L}}{2} & \text{if vertical alignment is } \textit{center} \\ 0 & \text{if vertical alignment is } \textit{top} \\ h_t - h_\mathbf{L} & \text{if vertical alignment is } \textit{bottom} \end{cases}$$

$$v_2 = \begin{cases} h_t - v_1 & \text{if vertical alignment is } \textit{center} \\ h_\mathbf{L} & \text{if vertical alignment is } \textit{top} \\ h_t & \text{if vertical alignment is } \textit{bottom} \end{cases}$$

The version of the engine used to implement the MGT did not support querying the GPU driver for the availability of NPOT textures, so texture mapping is treated equally on all platforms by using the calculations above. Another issue that has already been mentioned before is the missing feature for querying the size of a string by using certain fonts and point sizes. The developer is therefore responsible for manually setting a Label size that fits the assigned text.

Tab Controls

Tab Controls contain a number of Tab Pages and a control bar to switch between them. They are extensions of `Gui::Component`, because they are no Containers at all, but a utility to control Containers. The rendering of the Tab Pages and their content is handled by `Gui::TabPage`, so the Tab Control is responsible for a correct transformation of the Containers held. This means that the size of the Tab Page ($w_\mathbf{P} \times h_\mathbf{P}$) depends on the size of the Tab Control ($w_\mathbf{C} \times h_\mathbf{C}$) and on the size of the tab bar ($w_\mathbf{B} \times h_\mathbf{B}$):

$$w_\mathbf{P} = w_B = w_\mathbf{C}$$

$$h_\mathbf{P} = h_\mathbf{C} - h_B$$

This holds for horizontal tab bars. No other formats are supported yet, but it is possible to add this feature at low cost. The relative position of the Tab Pages must also be moved down by half of the tab bar's height. The more complicated part is to handle the toggling between Tab Pages and the visual adoption of tab buttons to indicate which tab is active. As learned in the previous chapter, the ME provides a node type called `Graph::Switch` for switching between child nodes, i.e., setting all

nodes deactivated and invisible unless the one that is considered to be selected by the Switch. At most one child node will be active and visible at a time, or none, if the Switch switches to an invalid child node index. Grouping all Tab Pages in a Switch node will yield the desired result. The index of the Switch will then be set on initialization and whenever the user presses a tab button.

4.2.4 Container Nodes

Containers

Containers are extensions of `Gui::Component` and inherit the property of being configurable in size and position. The Components added to the Container are attached to a dedicated Node within the Container subgraph that is called *Container Node Parent* internally. This could be any Node generated for this purpose and attached to the *Component Button* node to inherit the transformation. Nonetheless, the *Component Geometry* will be referenced to use for this purpose. All contained Components will therefore be children of the geometry node. Adding Components is only possible before the Container has been initialized, since the engine does not allow to restructure the graph after the initialization phase. There are also some other things that need to be considered.

First of all, the whole subgraph of a Widget will be created during initialization. This means that nothing more than the bare Node will exist immediately after creation. A Container deserialized from an XML file has the contained Components attached as immediate children.[3] As explained before, the Components are required to be children of the *Container Node Parent* though, otherwise they will not be affected by the transformation of the *Component Button*. The reorganization is done in `InitSelf()` as shown in listing 4.17

```
1  Graph::INodeArray children = GetChildren();
2  for(SInt32 index = 0; index < children.GetCount(); index++)
3  {
4      Gui::IComponent* component = dynamic_cast<Gui::IComponent*>(children[index]);
5      if(component != 0)
6      {
7          AddComponent(component);
8          RemoveChild(component->GetNodeInterface());
9      }
10     else
11     {
12         Gui::ILayoutDirective* directive = dynamic_cast<Gui::ILayoutDirective*>(
               children[index]);
13         if(directive != 0)
14         {
15             Graph::INodeArray layoutedChildren = directive->GetNodeInterface()->
                   GetChildren();
16             for(SInt32 jndex = 0; jndex < layoutedChildren.GetCount(); jndex++)
17             {
```

[3] It is also possible to attach any Node regardless of its type to Containers or even Widgets. Parent-child relationships are a basic feature of the engine's `Graph::Node` type and can not be turned off. The MGT ignores this fact and delegates the awareness to the developer. There might indeed be some situations for exploiting this relation usefully, e.g. by appending particle systems to Widgets.

```
18                  Gui::IComponent* layoutedComponent = dynamic_cast<Gui::IComponent
                        *>(layoutedChildren[jndex]);
19                  if(layoutedComponent != 0)
20                  {
21                      AddComponent(layoutedComponent, directive);
22                      directive->GetNodeInterface()->RemoveChild(layoutedComponent
                            ->GetNodeInterface());
23                  }
24              }
25          }
26      }
27  }
```

Listing 4.17: Reorganization of Container child nodes in `Gui::Container::InitSelf()`

This algorithm iterates over the immediate child nodes and checks if they are Components by dynamic casting, which will return 0 if not. The Component is then removed from its position in the graph and re-added with `AddComponent()`. This method does a proper registration of the Component for internal purposes and adds it as a child of *Container Node Parent*. If the node is not a Component, it may be a Layout Directive that is wrapped around one or more Components. If so, an overloaded version of `AddComponent()` is called that accepts both Component and Layout Directive. This method puts the Component to its right place, passes it to the Layout instance to fit it into the layout and also maps the Component to the Layout Directive for prospective re-computations of the Layout (see next paragraph). `AddComponent()` can also be called for adding Components programmatically instead of using XML. The Components are stored in an array and will become added to the graph while `InitSelf()` is executed. This will fail, however, if the Container has already been initialized (see above).

If the size of a Container changes, the Layout has to be refreshed. A Component will automatically invoke its virtual callback method `ApplySizeUpdate()`, which is overridden by `Gui::Container` (see listing 4.18). At first, the new inner dimension (outer dimension minus borders) will be calculated by the superclass. After that, the result will be passed to the Layout using the `UpdateLayout()` method, which hints the new boundaries and updates the size and/or position of all previously laid out Components.

```
1  void Gui::Container::ApplySizeUpdate()
2  {
3      Gui::Component::ApplySizeUpdate();
4
5      if (IsInitialized())
6      {
7          mContainerLayoutNodeTarget.GetNode(0)->UpdateLayout(
               mComponentInnerDimension);
8      }
9  }
```

Listing 4.18: Implementation of `Gui::Container::ApplySizeUpdate()`

Collapse Containers

`Gui::CollapseContainer` is a subclass of `Gui::Component` and has a compositional relation to `Gui::Container`. It consists of a background plane (*Component Geometry*), a `Gui::Container` node that holds the content and a control bar that is made of a multi-patch `Graph::GenericGeometry`, a State Set reference, a Label, and a `Graph::Switch` with two `Graph::Reference` nodes, all sandboxed into a `Graph::SubState` node. Figure 4.3 illustrates the subgraph of a Collapse Container.

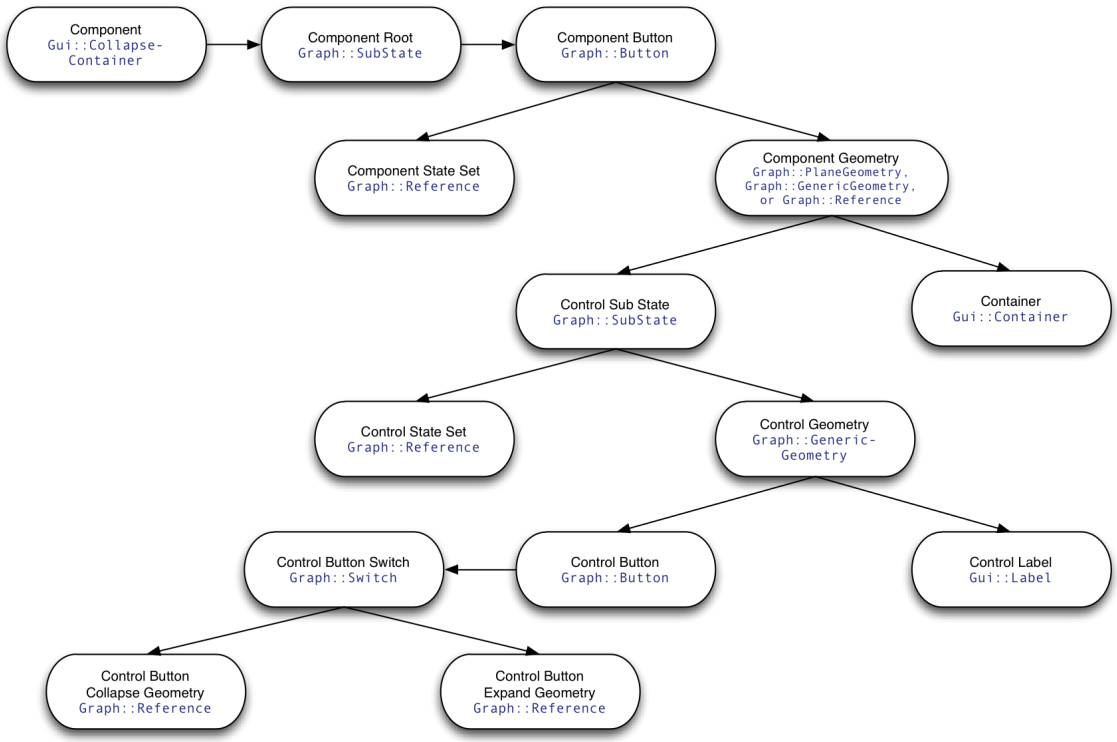

Figure 4.3: Collapse Container Subgraph

Collapse Containers use additional nodes to generate a control bar. Child Components are put into the nested *Container* node.

The public methods `Collapse()`, `Expand()` are used to collapse (hide) and expand (show) the nested Container. Listing 4.19 shows the implementation of `Collapse()` as an example. When collapsing the Widget, the size of the Component has to set to the size of the control bar as it is the only visible element. Because of the origin being centered in planes, the Y-position has to be moved up by the half of the nested Container height. The Container itself is turned off and the control button switch activates the expand button. The counterpart method, `Expand()`, undoes these operations. A third method, `Toggle()`, executes the right method to invert the current collapse state and is either called programmatically or in `FinishLogic()`, if the collapse/expand button of the control bar has been pressed.

```
1 Bool Gui::CollapseContainer::Collapse()
2 {
3     if (!mIsCollapsed)
```

```
4       {
5           mIsCollapsed = true;
6
7           if (IsInitialized())
8           {
9               Gui::Component::SetSizeY(mControlSize);
10              SetPositionY(mComponentOuterDimension.mPositionY + (mExpandedSize -
                    mControlSize) / Real(2));
11
12              mControlGeometry->GetTransformInterface()->SetPositionY(Real(0));
13              mContainer->GetNodeInterface()->SetActiveAndVisible(false);
14              mControlButtonSwitch->SetIndex(0);
15          }
16
17          return true;
18      }
19      else
20      {
21          return false;
22      }
23  }
```

Listing 4.19: Implementation of `Gui::CollapseContainer::Collapse()`

Collapse Containers are responsible for propagating size changes to its child Components. The control bar width depends on the inner width of the Collapse Container, which is equal to the outer width, if no multi-patch geometry is used to render the background plane. The Container dimension is both dependent from the width of the Collapse Container and the heights of the control bar and the Collapse Container. The Label within the control bar is stretched to the full width, and the control button is always arranged on the right side of the bar. All of these constraints are handled on initialization and on size updates in the Component callback method `ApplySizeUpdate()` (see listing 4.20).

```
1   void Gui::CollapseContainer::ApplySizeUpdate()
2   {
3       Gui::Component::ApplySizeUpdate();
4
5       if (IsInitialized())
6       {
7           mControlGeometry->GetTransformInterface()->SetPositionY(Gui::
                CalculateTopAlignmentPosition(mComponentOuterDimension.mSizeY,
                mControlSize));
8           mMultipatchFactory.SetSize(mComponentOuterDimension.mSizeX, mControlSize)
                ;
9
10          mControlLabel->GetComponentInterface()->SetSize(mMultipatchFactory.
                GetInnerDimension().mSizeX, mControlSize);
11
12          mControlButton->GetTransformInterface()->SetPositionX(
                CalculateRightAlignmentPosition(mMultipatchFactory.GetInnerDimension
                ().mSizeX, mControlButtonSizeX));
13
14          mContainer->GetComponentInterface()->SetPositionY(mControlSize / Real(-2)
                );
15          mContainer->GetComponentInterface()->SetSize(mComponentOuterDimension.
                mSizeX, mExpandedSize - mControlSize);
16      }
17  }
```

Listing 4.20: Implementation of `Gui::CollapseContainer::ApplySizeUpdate()`

The implementation of `Gui::CollapseContainer` has a serious drawback. Although the `Gui::IContainer` interface is accessible through `Gui::ICollapseContainer`, the accessor method will return 0, if the node has not been initialized, since this class *has a* Container but not *is a* Container. As said before, child nodes can not be created by the node itself before invoking `InitSelf()` (or, more precisely: `Init()`). However, adding Components to Containers is not possible any longer after that step, as mentioned above. Developers are therefore forced to define Collapse Containers in XML resources. There are ways to tackle this issue. One possibility would be the implementation of a proxy class that implements `Gui::IContainer` for private use only. When creating the actual Container, the properties will then be copied from the proxy to the Widget. However, since no flawless solution has been found yet, the issue is still open.

Scroll Containers

The `Gui::ScrollContainer` class has the most complex implementation among all Widgets. Scroll Containers display a rectangular aperture of a Container (and its Components) that has usually a bigger area than the "window" of the Scroll Container. The aperture is moved along the Container surface in both directions according to the scrollbar movements. A simple translation of the Container content will not be sufficient, since there is no way to clip the Components outside the borders of the Container. Therefore, a solution based on frame buffers has been implemented. Figure 4.4 illustrates the node hierarchy of a single Scroll Container. Scroll Containers consist of two subgraphs. The first is similar to most other Components, the second one represents a scene of contained Widgets that are rendered into a frame buffer texture (*Frame Buffer Texture*). This texture is referenced by the *Scroll Container Texture State* node, which causes the frame buffer to be drawn onto the *Component Geometry*.

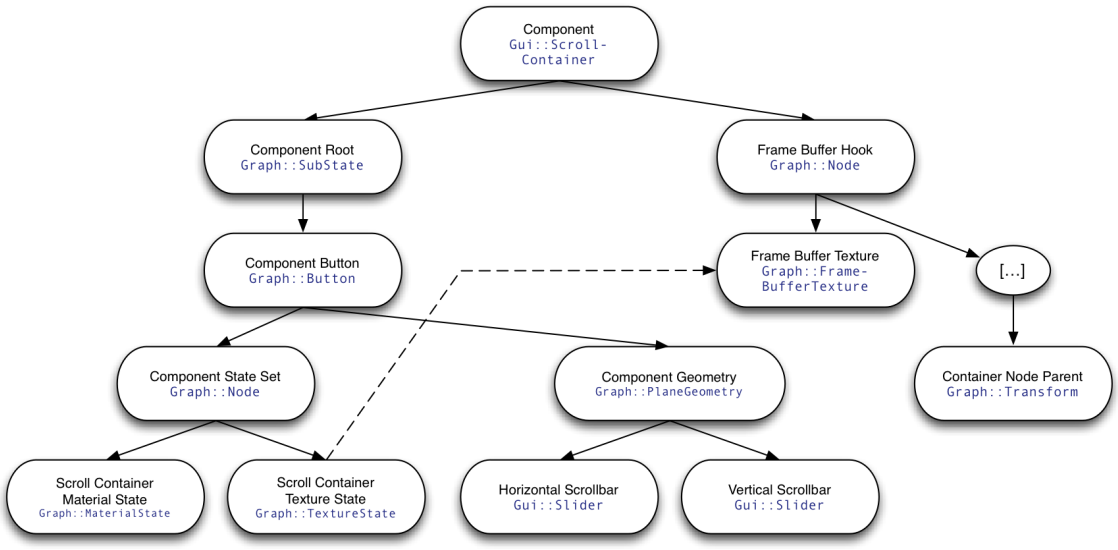

Figure 4.4: Scroll Container Subgraph

The structure of the frame buffered scene is similar to the one that is optionally made available by `Gui::Component`. In this case, however, the frame buffer subgraph is mandatory for the Scroll Container to work. The following list outlines the purpose of the required nodes that are needed for a proper use of frame buffers in the ME:

- The `Graph::Camera` (or rather one of its concrete subclasses) is used to capture the scene according to its transformation. It is aligned with the Z-axis and looks into the positive direction. One virtual coordinate unit maps to one rasterized pixel.
- The `Graph::View` writes the output pixels produced by a camera to a specified buffer, i.e., the frame buffer in this case.
- The `Graph::FrameBuffer` node manages an OpenGL frame buffer object (FBO) and is used to define the render texture target.
- The `Graph::FrameBufferTexture` stores the output of the frame buffer in a two-dimensional texture.

The nodes must be defined in reversed order, as they are listed here because of reference dependencies. The camera has to be the parent node of scene itself (i.e., the aligned Components of the Container), otherwise the scene will be rendered by the main camera. The output of the rendered sub-scene is finally available via the texture. By using `Graph::TextureState` nodes, a reference to the texture will be set within the visible sub-graph of the Scroll Container to map it onto the window plane.

The scrollbars are customized `Gui::Slider` objects. Their Number Entity will generate Entity Events, whenever the user drags the thumb. In other words, scrolling is performed by handling Entity Events of two Sliders within Scroll Containers. The "horizontal" Slider interpolates its values between $[-\frac{w_{\mathbf{C}}-w'_{\mathbf{C}}}{2}, \frac{w_{\mathbf{C}}-w'_{\mathbf{C}}}{2}]$, the "vertical" between $[-\frac{h_{\mathbf{C}}-h'_{\mathbf{C}}}{2}, \frac{h_{\mathbf{C}}-h'_{\mathbf{C}}}{2}]$, with w and h denoting width and height of the Scroll Container $\mathbf{C}$ and w' and h' the width and height of its aperture. To achieve a scroll effect, the frame buffer camera that records the content must now be moved to a position derived from the Number Entities of the Sliders after an Event has been emitted. The virtual X-coordinate is hereby equal to the value of the horizontal Slider. The virtual Y-coordinate is the negative value of the vertical Slider. The sign inversion has to be done in order to accommodate to the OpenGL coordinate system, with a positive Y moving upwards. Vertical Sliders will increment their value when moving the thumb downwards, because they are 90 degree clockwise rotated horizontal Sliders. Also note that the Slider interval does not extend to the full dimensions of the Container surface, but is cut off by the half width of the aperture on both ends. Again, the reason for this is OpenGL: The camera position in a 2D projection is the center of the view. So moving the frame buffer camera to $\pm\frac{w_{\mathbf{C}}}{2}$, for example, will align the camera upright to the left or right edge of the surface. One half of the view will then capture the void outside the Container, or – even worse – overflowing Components that were intended to be clipped.

Setting up the `Gui::Slider` instances to work as scrollbars requires some effort. In the default skin of the MGT, they differ in appearance and look more like the classical scrollbars of Microsoft Windows. The settings are taken from the skin package (which defines dedicated State Sets and atlases for scrollbar Sliders) and passed to the Sliders during Scroll Container initialization. The scrollbars are aligned on the right and bottom edge of the Container and have a higher depth order than the *Component Geometry* to generate an overlay effect upon the rendered content. Their visibility either depends on the size of the inner container (that means a scrollbar must be visible if the inner container is bigger than the aperture) or can be manually set on or off. If both scrollbars are visible, each one's length has to be reduced by the width of the other, preventing them to overlap on the bottom-right corner. The size of the scrollbars has to be updated whenever the size of the aperture (which is the Component's size property of the actual Scroll Container) changes. Additionally, the upper and lower limits of the Number Entities of both Sliders must be updated if the inner size changes.[4] This is both done in a protected method called `UpdateScrollbar()`, which is invoked during initialization and after the mentioned updates occur.

There is an important aspect that has to be considered in order to make all Widgets in the frame buffer scene ready for user interaction. As they are not visible to the main camera, interaction with `Graph::Button` nodes does not work. To solve this problem, the ME introduced a feature of the button node type called "Frame Buffer Node Target" (FBNT). If the FBNT of a button is set to a frame buffer node, the button forwards the input to the targeted frame buffered scene and also does a correct view transformation of the ray. For this purpose, the *Component Button* is configured with a FBNT that refers to the frame buffer node of the second sub-graph.

As a final note, areas that are not occluded by Components will occur as transparent pixel in the texture. So, depending on the content, the rendered Scroll Container might not be opaque. The default and effective geometry type of the Scroll Container (seen as Component) is set to `COMPONENT_GEOMETRY_TYPE_PLANE`, so there is no way to customize the appearance of the background immediately. To add a background anyway, the recommended solution is to put the Scroll Container into an ordinary Container with equal dimensions and a custom background.

Tab Pages

The implementation of `Gui::TabPage` is derived from `Gui::Container` and just adds two additional properties to set the tab label text and the tab icon. The rendering of the tab button is done by `Gui::TabControl`, which reads the properties from the Tab Page. The properties (both strings) can be set by calling the mutators

[4] Since Entity updates will fire Entity Events that cause the Slider to update the thumb position automatically, no extra adjustment needs to be done in order to synchronize the inner container size with the scrollbar thumbs.

or by deserializing the corresponding XML attributes. The default geometry type of the Component is set to `COMPONENT_GEOMETRY_TYPE_MULTIPATCH`.

Windows and Dialogs

`Gui::Window` is an extension of `Gui::Container` that adds some accessories and the ability to resize and drag. The effective geometry type is locked to `COMPONENT_GEOMETRY_TYPE_MULTIPATCH`. Windows therefore require a nine-patch texture that contains a title bar represented by the first three patches. The height of the first patch row defines the height of the title bar, while the height of the bottom row and the width of the middle left and middle right patches define the border insets of the Window. The inner dimension of the Window will therefore be equal to the dimension of the vertically and horizontally centered patch that is fully stretchable.

The right top corner contains a close button. This button is composed by a `Graph::Button` that holds a Reference to a textured plane geometry generated by the atlas generator. The plane geometry has the appearance of a typical closing icon. Pressing it will start the Window Closing Animation. After the animation has finished, the Window subgraph turns itself inactive and invisible. A user can change the Window's position by dragging its title bar. The implementation of this behavior is covered in section 4.2.7 about the Drag-and-Drop feature.

`Gui::Dialog` is an extension of `Gui::Window` that overwrites the resizable flag to `false` and defines some additional properties for setting up a Label and an Ok Button. The resulting Dialog subgraph is a composition of Widgets and Layouts. Figure 4.5 shows the graph structure that is attached to the *Container Node Parent* of `Gui::Container` as explained above. Since Dialogs are also Containers, a Page Layout is assigned automatically to align the Label and the Button.

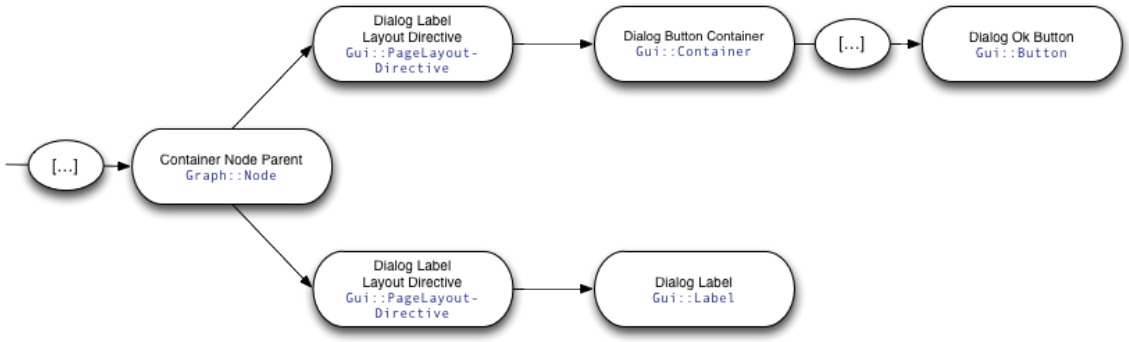

Figure 4.5: Dialog Widget Subgraph

A Dialog, which is also a Container, uses the Page Layout to align a predefined Label and a Button. Layout Directives are used to put the Label into the center section and the Button to the bottom.

4.2.5 Control Nodes

The abstract base class `Gui::Control` is a subclass of `Gui::Component` and adds the Entity property as well as some accessor methods that were declared by its interface `Gui::IControl`:

```
1 virtual Bool SetEntity(IEntity::SharedPtr Entity);
2 virtual IEntity::SharedPtr GetEntity();
3 virtual IEntity::ConstSharedPtr GetEntity() const;
```

Listing 4.21: Getter and setter methods of the Entity.

The Entity is not accessed directly, instead, a shared pointer is passed. Shared pointers are so called smart pointers for managing allocation and deallocation of objects. They are implemented as templates to wrap pointer types. A shared pointer holds the pointer and an internal reference counter to keep track of how many pointers to the object exist. If the counter reaches zero by destructing the last existing shared pointer, the object will be deallocated automatically. The ME provides implementations for each smart pointer type. Shared pointers are very important for Entities, because they are not bound to a specific Control and can be shared amongst many. This, of course, relieves the developer from the responsibility to destruct objects. Even if manually allocated, an explicit deallocation will most probably lead to an illegal memory access later on.

Beside the Entity property, `Gui::Control` also declares two pure virtual methods to be implemented by its subclasses:

```
1 public:
2     virtual Bool HandleEntityEvent(IEvent::ConstSharedPtr event) = 0;
3
4 protected:
5     virtual void SetupDefaultEntity() = 0;
```

Listing 4.22: Pure virtual methods of Control.

`HandleEntityEvent()` is a Callback Event Handler (see section 4.3.2) for Entity Events. A Control must handle them in order to update the appearance of the Widget according to the Entity state. The second method `SetupDefaultEntity()` is called on initialization, if neither an explicit Entity has been set nor another Control has been referred to in order to obtain its (shared) Entity. The upcoming sections will now provide information on the implementation of specific Control subclasses.

Buttons

`Gui::Button` is a subclass of `Gui::Component`, but not of `Gui::Control`, since Buttons do not make use of Entities. However, they are generally seen as controls, which is the reason for covering them in this section. The event handling done in `Gui::Component` is sufficient for Buttons to work properly, so the subclass does not

override its parent in context of this aspect. The difference is found in the setup of the State Sets, because a Button must react on user interaction related to the four button events defined by the engine. Instead of having a single node assigned to *Component State Set* (which are `Graph::Reference` nodes in most cases), the Button instantiates four references, each pointing to a certain State Set node defined by the skin. They will be attached to the *Component Button* and referred to as state-dependent nodes from the parent. To accomplish this, `Gui::Button` overrides `SetupStateSet()` without calling the base method. Note that the `mComponentStateSet` pointer will remain uninitialized, pointing to `NULL`. This will cause the base class to ignore the node, especially on de-initialization. Listing 4.23 contains an excerpt of the `Gui::Button::SetupStateSet()` implementation. For reasons of brevity, only the configuration of the up state is shown in the code, but the other three states work similar.

```
1  void Gui::Button::SetupStateSet(Graph::IInitTracker* tracker)
2  {
3      Graph::IRoot* root = tracker->GetRoot();
4      ComponentGeometryType geometryType = GetEffectiveComponentGeometryType();
5
6      if (geometryType == IEnums::COMPONENT_GEOMETRY_TYPE_REFERENCE)
7      {
8          return;
9      }
10     else if (geometryType == IEnums::COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
11     {
12         if (mButtonUpStateSetId.IsEmpty())
13         {
14             mButtonUpStateSetId = MURL_GUI_BUTTON_MULTIPATCH_UP_STATE_SET_ID;
15         }
16         [...]
17     }
18     else if (geometryType == IEnums::COMPONENT_GEOMETRY_TYPE_PLANE)
19     {
20         if (mButtonUpStateSetId.IsEmpty())
21         {
22             mButtonUpStateSetId = MURL_GUI_BUTTON_PLANE_UP_STATE_SET_ID;
23         }
24         [...]
25     }
26
27     mButtonUpStateSetReference = dynamic_cast<Graph::IReference*>(root->
           CreateNode("Reference"));
28     [...]
29
30     mButtonUpStateSetReference->GetNodeTarget()->SetNode(root->FindNode(
           mButtonUpStateSetId));
31     [...]
32
33     mComponentButton->GetNodeInterface()->AddChild(mButtonUpStateSetReference->
           GetNodeInterface());
34     [...]
35
36     mComponentButton->SetStateChildIndex(::IEnums::BUTTON_STATE_UP, 0);
37     [...]
38 }
```

Listing 4.23: Implementation of `Gui::Button::SetupStateSet()`

Another extension to the Component class provided by `Gui::Button` is the nested Label instance. Buttons have an own text property that is forwarded to the Label's

text property on initialization or when the property is modified. The Label is created in `InitSelf()` and attached to the *Component Geometry*, with a relative depth order of 1.

Check Switches

Class `Gui::CheckSwitch` provides an implementation of a checkbox, based on a Switch Entity. A Check Switch always uses a predefined plane geometry for rendering, since there are usually no variations within the same skin. Therefore, the `GetEffectiveComponentGeometryType()` of `Gui::Component` has been overridden to return constantly `COMPONENT_GEOMETRY_TYPE_REFERENCE`. Similar to Buttons (see previous section), Check Switches must consider up to four button states that are handled by the ME. Multiplied by the two switch states on and off, eight possible graphics must be ready for rendering. This differs from the situation of Buttons, which only have one background geometry to consider. Furthermore, Check Switches refer to plane geometries instead of State Sets, so no *Component State Set* node is not needed. But, instead of using a single node that represents the geometry, a hierarchy of two `Graph::Switch` nodes is required to switch between the button states and the switch states. As already mentioned, `Graph::Button` internally works like a switch node, if child nodes are explicitly associated with button states. Hence, `Gui::CheckSwitch` must only define four `Graph::Switch` nodes and two child nodes for each. This is done in listing 4.24. Figure 4.6 illustrates the resulting subgraph.

```
1  void Gui::CheckSwitch::SetupGeometry(Graph::IInitTracker* tracker)
2  {
3      Graph::IRoot* root = tracker->GetRoot();
4
5      SInt32 childCounter = 0;
6
7      for(SInt32 state = 0; state < ::IEnums::NUM_BUTTON_STATES_VISIBLE; state++)
8      {
9          if(mCheckSwitchOnReferenceTargetIds[state].GetLength() > 0 &&
               mCheckSwitchOffReferenceTargetIds[state].GetLength() > 0)
10         {
11             Graph::INode* offStateTarget = root->FindNode(
                   mCheckSwitchOffReferenceTargetIds[state]);
12             Graph::INode* onStateTarget = root->FindNode(
                   mCheckSwitchOnReferenceTargetIds[state]);
13
14             if (offStateTarget != 0 && onStateTarget != 0)
15             {
16                 Graph::ISwitch* toggleSwitch = dynamic_cast<Graph::ISwitch*>(root
                       ->CreateNode("Switch"));
17                 Graph::IReference* offStateReference = dynamic_cast<Graph::
                       IReference*>(root->CreateNode("Reference"));
18                 Graph::IReference* onStateReference = dynamic_cast<Graph::
                       IReference*>(root->CreateNode("Reference"));
19
20                 toggleSwitch->GetNodeInterface()->AddChild(offStateReference->
                       GetNodeInterface());
21                 toggleSwitch->GetNodeInterface()->AddChild(onStateReference->
                       GetNodeInterface());
22                 offStateReference->GetNodeTarget()->SetNode(offStateTarget);
23                 onStateReference->GetNodeTarget()->SetNode(onStateTarget);
24
25                 mComponentButton->GetNodeInterface()->AddChild(toggleSwitch->
                       GetNodeInterface());
26                 mComponentButton->SetStateChildIndex(::IEnums::ButtonState(state)
                       , childCounter);
```

```
27                  childCounter++;
28
29                  mCheckSwitchToggleSwitchNodes.Add(toggleSwitch);
30              }
31          }
32      }
33  }
```

Listing 4.24: Implementation of `Gui::CheckSwitch::SetupGeometry()`

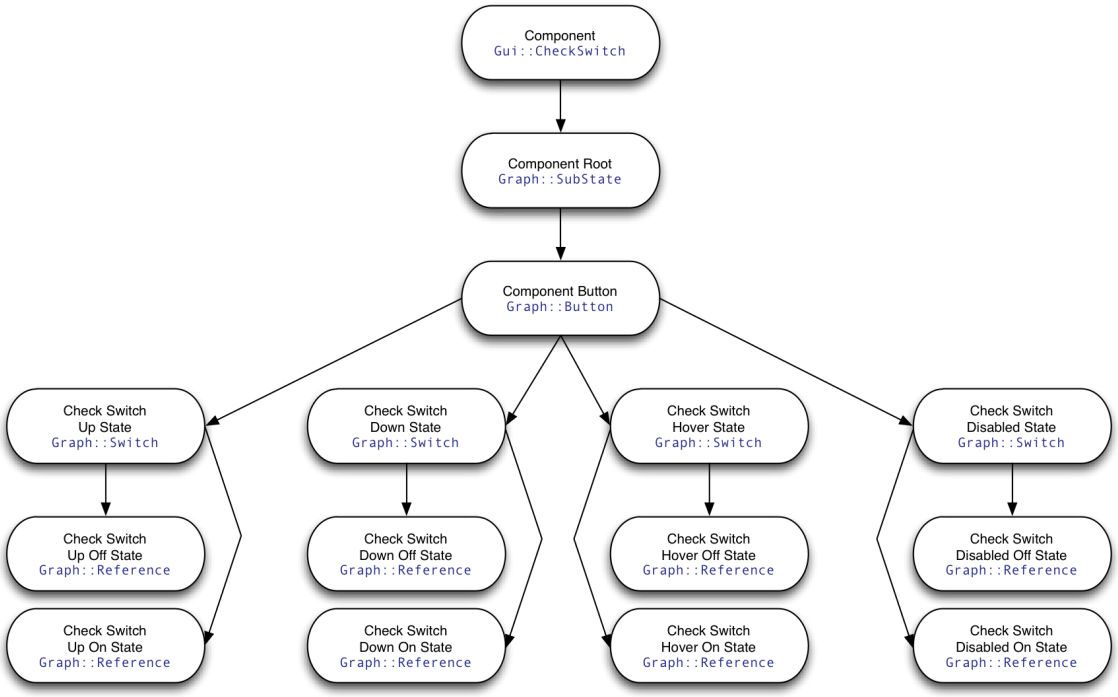

Figure 4.6: Check Switch Subgraph

The *Component Button* handles the switching between referenced geometries with respect to user interaction. The switching between the representation of the on and off state is done in the Entity Event Handler of the Check Switch, which is required by `Gui::IControl`. `HandleEntityEvent()` calls another method called `UpdateSwitchAppearance()` (see listing 4.25). This method simply determines the desired `Graph::Switch` index by converting the `Gui::SwitchEntity` state to an integer (0 or 1) and propagates it to all switch nodes created by the code shown above. Since those switches also need to be set up on initialization of the Check Switch, this step is encapsulated in an own protected method, rather than in `HandleEntityEvent()`.

```
1  void Gui::CheckSwitch::UpdateSwitchAppearance()
2  {
3      SInt32 switchIndex = SInt32(IsActivated());
4
5      for (SInt32 index = 0; index < mCheckSwitchToggleSwitchNodes.GetCount();
           index++)
6      {
7          mCheckSwitchToggleSwitchNodes[index]->SetIndex(switchIndex);
```

```
8       }
9   }
```

Listing 4.25: Implementation of `Gui::CheckSwitch::UpdateSwitchAppearance()`

The toggling of the Check Switch is implemented in the `FinishLogic()` callback. If a button release on the *Component Button* has been detected, the `Toggle()` method of the Switch Entity will be called. It performs the boolean inversion and emits an Entity Event that causes the Check Switch to update its appearance.

List Views and List Items

`Gui::ListView` is a Control that is composed of other Widgets. It does neither use an own *Component Geometry* nor a State Set. Instead, the rendering is done by a wrapped `Gui::ScrollContainer` instance that is configured for List Views. The Scroll Container uses a Grid Layout with $1 \times n$ grids, with n representing the number of contained List Items. `Gui::ListItem` is a subclass of `Gui::Control` and consists of a background plane (the *Component Geometry*) and a `Gui::Label`. List Items are vertically arranged top down by the Grid Layout. Their width is equal to the width of the Scroll Container, so a horizontal scrollbar is not needed and therefore turned off. Their height is a custom value and equal among all List Items.

The Entity instantiated by List Views is a `Gui::EntitySelection` that contains `Gui::TextEntity` objects. List Items, which are Controls just like List Views and thus require Entities, will be connected to the Text Entities of the Entity Selection. In this way, the relation between List Views and List Items will be the same as the relation between the assigned Entity Selection and its Text Entities. The string held by Text Entities will be passed to the Label of a single List Item. Other Entity types are also possible, because the string is retrieved by calling the `GetSerializedData()` method of the `Gui::IEntity` interface.

The List View is responsible for handling Selection Events of the `Gui::EntitySelection` object. It uses the `SetSelected()` method declared by the `Gui::IListItem` interface to set the selection state of the selected and the deselected List Items (if available). This is merely done for updating the appearance of the List Items, e.g. by setting a proper highlighting of the background that indicates selection. The implementation is shown in listing 4.26. This subgraph of the List View contains a `Graph::Switch` node to switch between two State Sets for rendering its background plane (*Component Geometry*) according to the selection state. If an Entity of the Entity Selection is updated, the List Item is responsible for handling the Entity Event by simply updating the text of its Label child node to the string gained with `GetSerializedData()`.

```
1 Bool Gui::ListView::HandleEntityEvent(Gui::IEvent::ConstSharedPtr event)
2 {
3     Gui::ISelectionEvent::ConstSharedPtr selectionEvent = Gui::ISelectionEvent::
          ConstSharedPtr::DynamicCast(event);
4     SInt32 deselectedIndex = selectionEvent->GetDeselectedIndex();
5     SInt32 selectedIndex = selectionEvent->GetSelectedIndex();
```

```
6
7       if (deselectedIndex != -1 && mListItems.GetCount() > deselectedIndex)
8       {
9           mListItems[deselectedIndex]->SetSelected(false);
10      }
11      if (selectedIndex != -1 && mListItems.GetCount() > selectedIndex)
12      {
13          mListItems[selectedIndex]->SetSelected(true);
14      }
15
16      return true;
17  }
```

Listing 4.26: Implementation of `Gui::ListView::HandleEntityEvent()`

The developer that uses List Views must consider that the set of List Items assigned to the List View and the items of the Selection are handled separately and follow an adopted version of the MVC pattern. List Items (views) are used to present the content of a Selection item (model), e.g., another Entity. They are tied together by a List View (controller). As instances of `Gui::ListItem` are nodes, all required objects must be allocated not later than the initialization of the List View. `Gui::ListView::OnInit()` synchronizes the contained List Items (added either programmatically or in XML) with the Selection. This means that each List Item gets mapped to a Selection item in the order they are stored in the corresponding arrays. If there are more Selection items than List Items, the lacking List Items will be automatically created. If there are more List Items than Selection items, the spare List Items will be set to inactive and invisible and may be used after initialization when adding more Entities to the Entity Selection. This is the only way to add items even if the List View has already been initialized. However, the maximum number of presented List Items can not be changed after initialization, because no extra nodes can be added to the subgraph of the List View. Adding further Entities will only expand the Entity Selection.

Although List Views generally involve much complexity, this implementation is rather simple, as many functional requirements have been delegated to other Widgets or classes of the toolkit.

Option Buttons

In the default skin, Option Buttons are resembling radio buttons as they are provided by most toolkits. In the MGT, the rendered texture may be replaced by any other image. That is why Option Buttons are not called radio buttons in the MGT. The appearance configuration is similar to Check Switches. First, the only geometry type supported is `COMPONENT_GEOMETRY_TYPE_REFERENCE`, i.e., a fixed-size, pre-defined image plane, preferably from the atlas generator. Further, the *Component Button* has four children, one for each button state. Each child is a `Graph::Switch` node that switches between two images (“on” and “off”) according to the selection state of the Option Button. Please refer to listing 4.25 for an exemplification of how the subgraph is built. The update of the appearance is shown in listing 4.27.

```
1  void Gui::OptionButton::UpdateOptionButtonAppearance()
2  {
3      Gui::ISelection::SharedPtr selection = Gui::ISelection::SharedPtr::
           DynamicCast(GetEntity());
4      SInt32 switchIndex = (selection->GetSelectedIndex() == mOptionButtonIndex) ?
           1 : 0;
5
6      for (SInt32 index = 0; index < mOptionButtonToggleSwitchNodes.GetCount();
           index++)
7      {
8          mOptionButtonToggleSwitchNodes[index]->SetIndex(switchIndex);
9      }
10 }
```

Listing 4.27: Implementation of `Gui::OptionButton::UpdateOptionButtonAppearance()`

This implementation resembles the one used by Check Switches, but handles another Entity type: Selections (`Gui::ISelection`). Each Selection has a certain number of items associated with it and each Option Button has an option index. An Option Button is selected, if the selection index of the Selection is equal to the option index of the Option Button. If there is a shared Entity assigned, the Option Button adds an item to the Selection and retrieves its option index from the number of currently registered items. Otherwise, a new Selection will be created with a single item that represents the Option Button. Like in any other Control, this happens during `InitSelf()` execution. Note that per default the `Gui::Selection` Entity type is used, which does not contain actual items. It is rather a counter with a selected index property that points to a virtual or exterior item. However, the developer is free to replace this by any other implementation of `Gui::ISelection` programmatically, e.g., Selections that wrap array data.

In `Gui::OptionButton::FinishLogic()`, the *Component Button* is checked. If it has been released in the most recent tick, the selection index of the Selection is set to current option index. This will fire an Entity Event (and a Selection Event), which is handled by Option Buttons by default, resulting in an invocation of `UpdateOptionButtonAppearance()` for all Options Buttons within the same Selection.

The usage of Selections as Entity type for this Control has a great advantage over many other radio button solutions in other toolkits. Obviously, the amount of code is very small since there is no extra logic required for handling the interconnection between different Widgets with the same Entity. Many toolkits use additional objects (radio button groups) instead. Here, each Option Button is still on its own, it just queries its Entity. This leads to another point, the conceptual consistency with other Controls, since the Entity concept fully satisfies the Option Buttons' requirements. Selections and shared Entities are no particular feature of Option Buttons, but occur in many other Widgets also.

Progress Indicators

The default implementation of Progress Indicators provided by the MGT is the `Gui::ProgressBar` widget. It consists of a background plane and a foreground plane that covers the background from left to right to a degree related to the normalized value of the used Number Entity (`GetNormalizedValue()`). The background geometry is the default *Component Geometry* and automatically configured by the base classes. An additional geometry needs to be created and added to the existing one as a child node for rendering the foreground. This is done in listing 4.28, which also adds the proper State Set of the foreground bar. If the geometry is a multi-patch type, a factory is used to create the geometry. Otherwise, it will be a plane geometry (no references are allowed here).

```
1  void Gui::ProgressBar::SetupGeometry(Graph::IInitTracker* tracker)
2  {
3      Gui::Control::SetupGeometry(tracker);
4
5      Graph::IRoot* root = tracker->GetRoot();
6
7      mProgressBarStateSet = dynamic_cast<Graph::IReference*>(root->CreateNode("
           Reference"));
8      mProgressBarStateSet->GetNodeTarget()->SetNode(root->FindNode(
           mProgressBarStateSetId));
9      mComponentGeometry->AddChild(mProgressBarStateSet->GetNodeInterface());
10
11     if (GetEffectiveComponentGeometryType() == Gui::IEnums::
           COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
12     {
13         mProgressBarGeometryFactory.Init(root, mComponentMultipatchRectangles,
               mComponentOuterDimension.mSizeX, mComponentOuterDimension.mSizeY, 3,
               4);
14         mProgressBarGeometry = mProgressBarGeometryFactory.GetMultipatchPlane()->
               GetNodeInterface();
15         mProgressBarGeometryFactory.GetMultipatchPlane()->GetTransformInterface()
               ->SetDepthOrder(1);
16     }
17     else
18     {
19         Graph::IPlaneGeometry* planeGeometry = dynamic_cast<Graph::IPlaneGeometry
               *>(root->CreateNode("PlaneGeometry"));
20         planeGeometry->GetTransformInterface()->SetDepthOrder(1);
21         mProgressBarGeometry = planeGeometry->GetNodeInterface();
22     }
23
24     mComponentGeometry->AddChild(mProgressBarGeometry->GetNodeInterface());
25 }
```

Listing 4.28: Implementation of `Gui::ProgressBar::SetupGeometry()`

During initialization and after each Entity Event, the private `ApplyProgressUpdate()` method of listing 4.29 is called to update the appearance of the Progress Bar. The foreground bar width is calculated from the background width times the normalized Number Entity value. After resizing, the geometry must be realigned to match the left edge of the background using the `Gui::CalculateLeftAlignmentPosition()` utility function. Both plane geometry and multi-patch geometry types must be handled separately due to their different interface.

```
1  void Gui::ProgressBar::ApplyProgressUpdate()
```

```
2  {
3      Graph::ITransform* transform = 0;
4      Gui::ComponentGeometryType geometryType = GetEffectiveComponentGeometryType()
           ;
5      Gui::INumberEntity::SharedPtr numberEntity = Gui::INumberEntity::SharedPtr::
           DynamicCast(GetEntity());
6      Real normalizedValue = numberEntity->GetNormalizedValue();
7      Real backgroundSizeX = mComponentOuterDimension.mSizeX;
8      Real foregroundSizeX = normalizedValue * backgroundSizeX;
9
10     mProgressBarGeometry->SetActiveAndVisible(normalizedValue > Math::Limits<Real
           >::Epsilon());
11
12     if(geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
13     {
14         transform = dynamic_cast<Graph::IGenericGeometry*>(mProgressBarGeometry)
               ->GetTransformInterface();
15         mProgressBarGeometryFactory.SetSize(foregroundSizeX,
               mComponentOuterDimension.mSizeY);
16     }
17     else if(geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_PLANE)
18     {
19         transform = dynamic_cast<Graph::IPlaneGeometry*>(mProgressBarGeometry)->
               GetTransformInterface();
20         dynamic_cast<Graph::IPlaneGeometry*>(mProgressBarGeometry)->
               SetScaleFactor(foregroundSizeX, mComponentOuterDimension.mSizeY, Real
               (1));
21     }
22
23     if(transform != 0)
24     {
25         Real positionX = Gui::CalculateLeftAlignmentPosition(backgroundSizeX,
               foregroundSizeX);
26         transform->SetPositionX(Util::Max(positionX, mComponentInnerDimension.
               mSizeX / Real(-2)));
27     }
28 }
```

Listing 4.29: Implementation of `Gui::ProgressBar::ApplyProgressUpdate()`

Slide Switches

`Gui::SlideSwitch` provides an alternative implementation for `Gui::ISwitch`, resulting in a switch that is much more associated with mobile devices. Due to the same interface, there is no difference for the developer to handle this Widget's logic. There are, however, differences in the internal setup of the subgraph. Most significantly, no button states are considered here. The current state of the switch is indicated by the position of the thumb and the visible part of the track. The thumb's movement is animated by a controller and triggered on pressing the Widget. Again, only referenced geometries are supported – Slide Switches do usually not vary in appearance within the same skin.

Sliders

A Slider consists of a track geometry (background), a thumb geometry (foreground), and two optional increment/decrement buttons (aside). `Gui::Slider` uses the *Component Geometry* node of its parent class as track. It therefore needs to add a `Graph::Button` that contains a plane or generic geometry for the thumb object as well as for each of the two optional buttons, if activated. The generated subgraph

is sketched in figure 4.7. As mentioned above, Sliders are also used as scrollbars for Scroll Containers. This is the reason why some more flexibility has to be implemented, for example the increment/decrement buttons and the variable geometry type of the thumb. The whole configuration is done in `SetupGeometry()`.

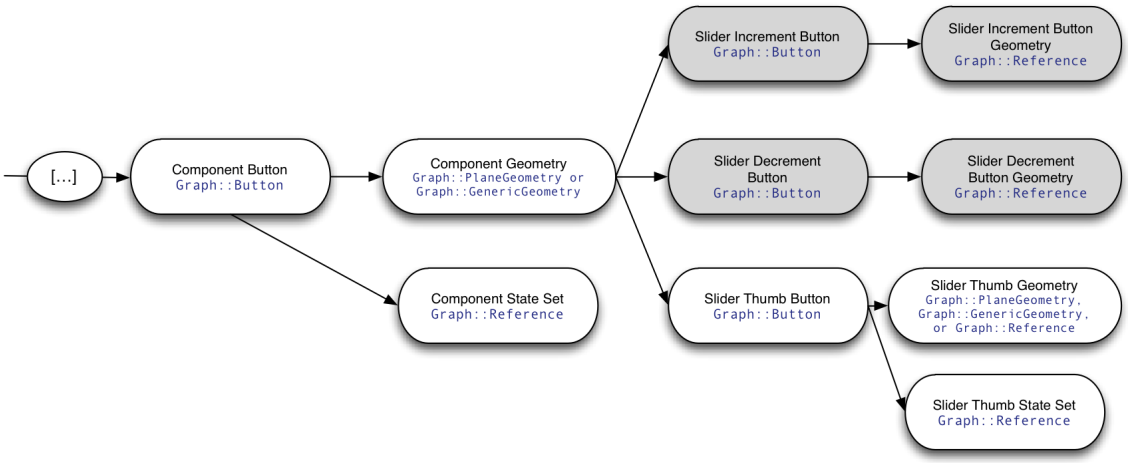

Figure 4.7: Slider Subgraph

Sliders use additional buttons and geometries to add a draggable thumb and optional increment/decrement buttons (here: gray).

Listing 4.30 contains the `ProcessLogicSelf()` method that is primarily used to handle the thumb dragging. The dragging algorithm is an adopted version of the one used by `Gui::Window` and by the Drag-and-Drop implementation (see below). A flag is used to remember if a drag input has been detected during the last tick, and the *Slider Thumb Button* is queried for the number of tracked events. Both parameters may create four different situations:

1. A tracked event has been detected and the flag is set to false: This indicates the beginning of the drag movement. The flag is set to true and the position of the tracked event is stored for later use. This is the origin position of the drag.

2. A tracked event has been detected and the flag is set to true: The user has already started the interaction and now proceeds with the drag movement. The current drag distance is the difference between the origin position and the current position of the tracked event. The value of the Number Entity is interpolated between minimum and maximum at the relative position of the dragged thumb. Finally, a Slide Event is emitted.

3. No tracked event has been detected, but the flag is set to true: The user interacted with the Slider until the previous tick and then released the thumb. Reset the flag and the origin position.

4. No tracked event has been detected and the flag is set to false: The user is not and was not interacting with the Slider. Do nothing.

```
1  Bool Gui::Slider::ProcessLogicSelf(Graph::IProcessLogicTracker* tracker)
2  {
3      if (!Gui::Control::ProcessLogicSelf(tracker))
4      {
5          return false;
6      }
7
8      UInt32 numberOfTrackedEvents = mSliderThumbButton->GetNumberOfTrackedEvents()
           ;
9
10     if (numberOfTrackedEvents > 0)
11     {
12         UInt32 eventID = mSliderThumbButton->GetTrackedEventId(0);
13
14         if (mSliderThumbIsDragged)
15         {
16             Gui::INumberEntity::SharedPtr model = Gui::INumberEntity::SharedPtr::
                   DynamicCast(GetEntity());
17             Gui::Number maximum = model->GetMaximum();
18             Gui::Number minimum = model->GetMinimum();
19             Real eventPosition = mSliderThumbButton->GetLocalEventPosition(
                   eventID).x;
20             Real slidePosition = mSliderThumbButton->GetTransformInterface()->
                   GetPositionX() + eventPosition - mSliderThumbDragOriginPosition;
21
22             slidePosition = Util::Min(Util::Max(slidePosition, -mSliderConstraint
                   ), mSliderConstraint);
23
24             Real slide = slidePosition / (2.0f * mSliderConstraint) + 0.5f;
25
26             if (model->GetValue().IsReal())
27             {
28                 model->SetValue(slide * (Real(maximum) - Real(minimum)) + Real(
                       minimum));
29             }
30             else
31             {
32                 model->SetValue(SInt64(slide * ((SInt64(maximum) - SInt64(minimum
                       )) + SInt64(minimum))));
33             }
34
35             mSliderThumbButton->GetTransformInterface()->SetPositionX(
                   slidePosition);
36
37             SKIP_ONCE_IF(mIsRelaxed)
38             {
39                 Gui::ISlideEvent* slideEvent = new Gui::SlideEvent(this, slide);
40                 Gui::IEvent::ConstSharedPtr pointer = Gui::IEvent::ConstSharedPtr
                       (slideEvent->GetEventInterface());
41                 Gui::EventPipeline::GetInstance()->DispatchEvent(pointer,
                       mSlideEventChannel);
42             }
43
44             PushContext(false);
45         }
46         else
47         {
48             mSliderThumbIsDragged = true;
49             mSliderThumbDragOriginPosition = mSliderThumbButton->
                   GetLocalEventPosition(eventID).x;
50         }
51     }
52     else if(mSliderThumbIsDragged)
53     {
54         mSliderThumbIsDragged = false;
55         mSliderThumbDragOriginPosition = 0.0f;
56     }
57
58     return true;
```

```
59 }
```

Listing 4.30: Implementation of `Gui::Slider::ProcessLogicSelf()`

In `FinishLogic()`, the increment and decrement buttons are checked, if available. If a button event has been detected, the `Increment()` or `Decrement()` method of the Slider's Number Entity will be called, resulting in an Entity Event. Similar to the drag actions, `UpdateThumbGeometry()` will be called in the Entity Event Handler of the Slider. `Gui::Slider` also provides an overridden `HandleWheelEvent()` method that reads the Y-axis of the wheel rotation and calls `Increment()` on the Number Entity if the rotation is greater than zero or `Decrement()` otherwise. This is just a demonstration on how the Slider can be improved considering alternative input sources. Similar extensions are possible, for example by overriding the Keyboard Event Handler and parsing the arrow buttons.

Table Views, Table Rows, and Table Cells

The `Gui::TableView` node is a subclass of `Gui::Component` and, similar to List Views, a composition of other Widgets and Layouts. Its *Component Geometry* is used as background plane while the foreground content is put into a Scroll Container that uses a Grid Layout. The cells are represented by Text Fields for both manipulating and presenting the two-dimensional data array. An array of column widths can be passed to the Table View object. They will be forwarded to `Gui::GridLayout::SetGridScaleFactorsX()`, which will tell the Layout to set the Text Fields' width to the corresponding column values.

The hierarchy of Table Row nodes and Table Cell nodes exists beneath the Component hierarchy established during the Table View initialization. They are not rendered, but serve a structural purpose when deserializing scene graph XML files and store the data strings. A Table Row simply keeps track of its Table Cells and provides this information to the Table View. A Table Cell only holds the properties for the data itself and its editable flag. The `InitSelf()` method of `Gui::TableView` iterates over the Table Views and Table Cells to build the subgraph of this structure's visual counterpart. Further, a Callback Event Handler will be prepared and assigned to each created Text Field in order to react on Entity updates. On Entity Event occurrence, the Event Handler synchronizes the looked up Table Cell node with the updated data string to keep data consistent (see listing 4.31).

```
1 Bool Gui::TableView::HandleEntityEvent(Gui::IEvent::ConstSharedPtr event)
2 {
3     Gui::IEntityEvent::ConstSharedPtr entityEvent = Gui::IEntityEvent::
          ConstSharedPtr::DynamicCast(event);
4     const Gui::ITextField* textField = dynamic_cast<const Gui::ITextField*>(
          entityEvent->GetEventInterface()->GetTrigger());
5
6     SInt32 index = mTableViewTextFields.Find(textField);
7
8     if (index != -1)
9     {
```

```
10          SInt32 row = index / mTableNumberOfColumns;
11          SInt32 column = index % mTableNumberOfColumns;
12
13          SetCellData(textField->GetText(), column, row);
14      }
15  }
```

Listing 4.31: Implementation of `Gui::TableView::HandleEntityEvent()`

Input Fields, Text Fields and Steppers

`Gui::InputField` is the base class for alphanumeric user input Controls, allowing character input from hard or virtual keyboards. The base class itself configures and manages the subgraph that is common to both derived Widgets, `Gui::TextField` and `Gui::Stepper`. Both Widgets consist of a background plane (in most toolkits, also here, input fields are rendered as white rectangles with a one-pixel-border) and a text-textured geometry on the front. The background is set up by `Gui::Component` as *Component Geometry*. The overridden `SetupGeometry()` method additionally generates a `Graph::FlatTextTexture` and a `Graph::PlaneGeometry` node to put it on the top of the background geometry. The depth ordering of the plane is set to 1 (relative to the background) to ensure proper overlaying. As a result, the text texture is rendered to the plane geometry that is in front of a background geometry. The subclasses are responsible for the logic and will modify a dedicated protected string property and set a modify flag to update the rendered text according to the processed input during the next logic traversal (see listing 4.32).

```
1  Bool Gui::InputField::ProcessLogicSelf(Graph::IProcessLogicTracker* tracker)
2  {
3      if (Control::ProcessLogicSelf(tracker))
4      {
5          if (mInputFieldTextWasUpdated)
6          {
7              mInputFieldTextWasUpdated = false;
8              mInputFieldTextTexture->SetText(mInputFieldText);
9          }
10         return true;
11     }
12     else
13     {
14         return false;
15     }
16 }
```

Listing 4.32: Implementation of `Gui::InputField::ProcessLogicSelf()`

Text Fields: Text Fields store their content in a Text Entity. The subgraph setup of its base class is sufficient for presenting a qualified Text Field. The required behavior is achieved by implementing some Event Handlers and logic as described here. `Gui::TextField` overrides the `HandleKeyboardEvent()` method inherited from `Gui::Widget`. Its implementation (see listing 4.33) provides a minimalistic keyboard input handling. The input keys can be read from `Gui::IKeyboardEvent` and appended to the existing string in the Entity by using the `AppendString()`

method. If a backspace input was detected, the `RemoveLastCharacter()` method of the Entity is called instead. Both methods are wrappers to the string types native manipulation operations, but in addition, they also invoke an Entity Event to imply an update of the string. Note that tab keys are already parsed by `Gui::Context`.

```
1  Bool Gui::TextField::HandleKeyboardEvent(Gui::IEvent::ConstSharedPtr event)
2  {
3      Gui::IKeyboardEvent::ConstSharedPtr keyboardEvent = Gui::IKeyboardEvent::
           ConstSharedPtr::DynamicCast(event);
4      Gui::ITextEntity::SharedPtr textEntity = Gui::ITextEntity::SharedPtr::
           DynamicCast(GetEntity());
5      String key = keyboardEvent->GetKey();
6
7      if(key[0] == ::IEnums::KEYCODE_BACKSPACE && textEntity->GetText().GetLength()
           > 0)
8      {
9          textEntity->RemoveLastCharacter();
10     }
11     else
12     {
13         textEntity->AppendString(key);
14     }
15     return true;
16 }
```

Listing 4.33: Implementation of `Gui::TextField::HandleKeyboardEvent()`

Like other Controls, the Text Field must handle Entity Events to react on interior and exterior Entity manipulation. Since rendering of the text is done by the base class, the Text Field must set an update flag and overwrite the string property provided by `Gui::InputField`, as it is done in listing 4.34.

```
1  Bool Gui::TextField::HandleEntityEvent(Gui::IEvent::ConstSharedPtr event)
2  {
3      Gui::IEntityEvent::ConstSharedPtr entityEvent = Gui::IEntityEvent::
           ConstSharedPtr::DynamicCast(event);
4      const Gui::ITextEntity* textEntity = dynamic_cast<const Gui::ITextEntity*>(
           entityEvent->GetEntity());
5
6      mInputFieldText = textEntity->GetText();
7      mInputFieldTextWasUpdated = true;
8
9      return true;
10 }
```

Listing 4.34: Implementation of `Gui::TextField::HandleEntityEvent()`

The blinking cursor of the Text Field is implemented as underscore that oscillates its visibility every second. Thus, when the tick time is right, the underscore is simply appended to the string property that is used as input for text rendering. The update flag also needs to be set here. Listing 4.35 contains the cursor code. Also the implementation of the security feature is shown here, which simply generates a string of asterisks with the same length of the actual string.

```
1  Bool Gui::TextField::ProcessLogicSelf(Graph::IProcessLogicTracker* tracker)
2  {
3      if(InputField::ProcessLogicSelf(tracker))
4      {
```

```
5           UInt32 time = Math::Round(tracker->GetCurrentLogicTickTime() * Real(2));
6           if(mWidgetIsFocused && mTextFieldTickTime != time)
7           {
8               if(!mTextFieldSecure)
9               {
10                  mInputFieldText = GetText();
11              }
12              else
13              {
14                  mInputFieldText = String('*', GetText().GetLength());
15              }
16
17              if(time % 2 == 0)
18              {
19                  mInputFieldText.Cat("_");
20              }
21              mInputFieldTextWasUpdated = true;
22              mTextFieldTickTime = time;
23          }
24          return true;
25      }
26      else
27      {
28          return false;
29      }
30  }
```

Listing 4.35: Implementation of `Gui::TextField::ProcessLogicSelf()`

Additional consideration must be made in consequence of the cursor implementation. If the Component loses focus, while the text cursor is in displaying phase, it will remain visible until the next text update. Therefore, Text Field must also override `HandleFocusEvent()` from `Gui::Widget` to handle this case. When the Widget loses focus, the text will be set to the string without cursor and marked as updated (see listing 4.36).

```
1   Bool Gui::TextField::HandleFocusEvent(Gui::IEvent::ConstSharedPtr event)
2   {
3       Gui::InputField::HandleFocusEvent(event);
4
5       if (!mWidgetIsFocused)
6       {
7           mInputFieldText = GetText();
8           mInputFieldTextWasUpdated = true;
9       }
10      return true;
11  }
```

Listing 4.36: Implementation of `Gui::TextField::HandleFocusEvent()`

The solution explained here is neither sophisticated nor aesthetically appealing. Moreover, it is an provisional solution since the engine did not allow to query the size of the rendered text (not the size of the texture!) at the time of implementation. The desired solution would be the usage of an cursor image or geometry overlay with an animation controller to oscillate visibility. The position of the cursor would simply be deducted from the rendered text size. Another important feature that has been omitted due to the lack of engine features is the marking of text or parts of it.

Steppers: Steppers are numerical input fields with additional increment/decrement buttons. The data is held by a Number Entity. The existing subgraph of `Gui::InputField` will be extended by two `Graph::Button` nodes, each with a `Graph::PlaneGeometry` for rendering the increment and decrement buttons. Both buttons are placed one upon the other on the inner right side of the Widget using the `Gui::CalculateRightAlignmentPosition()` utility method. The buttons are queried in the `FinishLogic()` method. If they were pressed, the increment/decrement method of the Number Entity is called, which will lead to an Entity Event that is handled by the Stepper as shown in listing 4.37. Again, the update flag will be set, notifying the base class that the text for rendering has been changed.

```
1 Bool Gui::Stepper::HandleEntityEvent(Gui::IEvent::ConstSharedPtr event)
2 {
3     mInputFieldText = GetEntity()->GetSerializedData();
4     mInputFieldTextWasUpdated = true;
5     return true;
6 }
```

Listing 4.37: Implementation of `Gui::Stepper::HandleEntityEvent()`

This concludes the discussion about the implementation of Widget nodes. The second collection of GUI node types are Layouts and Layout Directives, covered in the next section.

4.2.6 Layout and Layout Directive Nodes

The MGT introduces eight Layout-related node types, four Layouts and one Layout Directive for each. Figure 3.8 shows the four Layout implementations `Gui::FlowLayout`, `Gui::GridLayout`, `Gui::NullLayout`, and `Gui::PageLayout` being subclassed from the abstract `Gui::BaseLayout`. `Gui::BaseLayout` implements the common `Gui::ILayout` interface, which declares methods for registering Components and updating the Layout. It unifies the handling of Components and Layout Directives and reduces the subclasses to pure algorithm implementations. The subclasses just need to implement their dedicated interfaces and the following pure virtual methods declared by `Gui::BaseLayout`:

```
1 virtual Bool CalculatePosition(IComponent* component) = 0;
2 virtual Bool CalculatePosition(IComponent* component, const ILayoutDirective*
      directive) = 0;
```

Listing 4.38: Pure virtual methods declared by `Gui::BaseLayout`.

`CalculatePosition()` requires the Layout to update the position of the Component according to a Layout Directive or an internal state. This method is called within the implementation of the public `LayoutComponent()` methods, which handles registration of Components and Layout Directives, but also updates the already laid out Components, if the boundary size has changed:

```
1 Bool Gui::BaseLayout::LayoutComponent(Gui::IComponent* component,
```

```
2       const Gui::Dimension& boundary,
3       const Gui::ILayoutDirective* directive)
4   {
5       Real epsilon = Math::Limits<Real>::Epsilon();
6       Real deltaSizeX = Math::Abs(mBoundary.mSizeX - boundary.mSizeX);
7       Real deltaSizeY = Math::Abs(mBoundary.mSizeY - boundary.mSizeY);
8
9       if(deltaSizeX > epsilon || deltaSizeY > epsilon)
10      {
11          UpdateLayout(boundary);
12      }
13      mComponentList.Add(component);
14      mDirectiveTable.Add(component, directive);
15
16      return CalculatePosition(component, directive);
17  }
```

Listing 4.39: The `LayoutComponent()` method. The overloaded version without the third argument works similar, but ignores Layout Directives.

As said above, the subclasses mainly focus on the implementation of the layout schemes described in section 3.5, which consists of some straight-forward calculations with no need for describing them here. A look at figure 3.9 also reveals that Layout Directives do nothing more than holding information for the Layout to process, thus, their implementation mainly consists of property getters and the `DeserializeBaseAttribute()` method, which does its work simply as described in section 4.2.1.

Note that a concrete Layout class implements its respective interface, but not the common `Gui::ILayout`. This interface is implemented by their common base class `Gui::BaseLayout` and made accessible by the `GetLayoutInterface()` methods, as the implementation hidden by the abstract factory pattern obscures the actual inheritance. This spares the developer from doing explicit dynamic casts on objects that do not provide any hint on their implementation details. This pattern is found in many classes of the ME and the MGT. It is a fail-safe solution to the diamond problem illustrated in figure 4.8.

4.2.7 Drag-and-Drop Related Nodes

Drag-and-Drop is a feature of Component and Container instances, allowing Components to be dragged into Containers. Section 3.4.3 explains this in greater detail. The involved classes and interfaces are outlined in figure 3.6. The role of Components is described by the `Gui::IDragController` interface, while Containers obtain their behavior from `Gui::IDropController`. Both interfaces are accessible via the `Gui::IDragDropFamily` interface, which is implemented by the node class `Gui::DragDropFamily`. As stated in the previous chapter, the unification of both controllers has been done in order to group Components and Containers within the same Drag-and-Drop context. Implementing the Drag Drop Family as Node was not necessary, but makes it easier for the developer to implement Drag-and-Drop in the XML scene graph description. In this way, Drag-and-Drop works out of the box without further configuration.

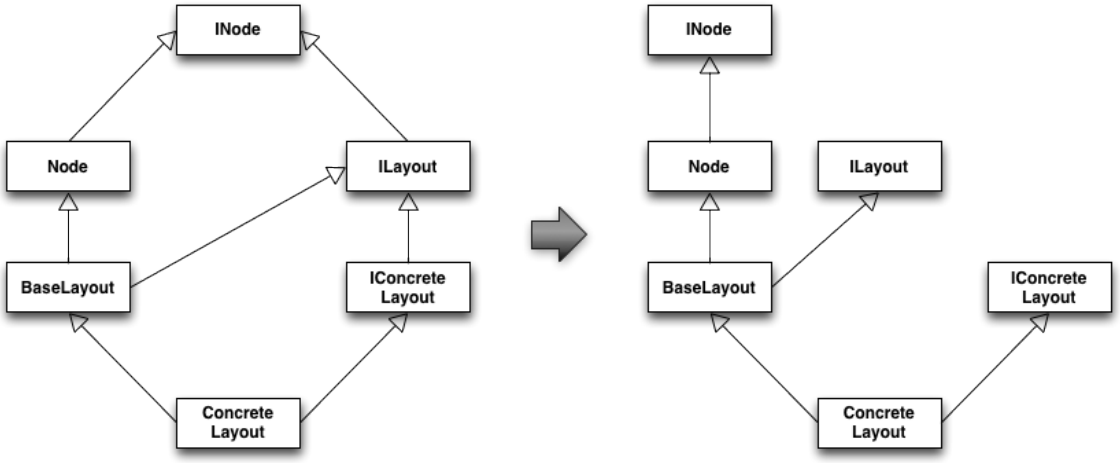

Figure 4.8: The Diamond Problem and its Solution

The classes on the left are involved in a double diamond problem. The problem was resolved by cutting off two direct interface implementations. Instead of explicitly represent the private inheritance within the public interfaces, only the base classes will implement the common interface. To drop the necessity of dynamic casting (which would require knowledge on the hierarchy of hidden classes), interface accessor methods are declared by the former child interface.

Drag-and-Drop must be handled by both Component and Container involved simultaneously. The Drag Drop Family acts as a communication channel, allowing Components to query available drop targets and Containers to check if a Component has been dragged/dropped recently. This requires both sides, Component and Containers, to notify the controller about current events. Components talk to Drag Drop Families, seeing them as Drag Controllers, by calling the methods declared in `Gui::IDragController` as it is done in listing 4.40. Handling drag input is the main purpose of `Gui::Component::ProcessLogicSelf()` by considering the following situations and reacting on them as follows:

- No tracked events are detected on the *Component Button*, and the `mComponentIsBeingDragged` flag is not set: There is no user interaction, so do nothing.
- Tracked events are detected on the *Component Button*, and the `mComponentIsBeingDragged` flag is not set: This is possibly the beginning of a drag movement. The Component must propagate its pointer and the tracked event ID to the Drag Controller by calling `BeginDrag()`. The local event position must be stored in order to calculate the relative movement later on. Furthermore, the drag flag is set and the *Component Button* is permitted to pass input events onto buttons that lie "behind" it, i.e., have a lower depth order. This is necessary for the targeted Container to receive input events, allowing it to handle drop actions.[5]

[5] `PushNodeToFinishLogic()` causes the traverser to call `FinishLogic()` after traversal, which does the event handling for the current Component. This has nothing to do with Drag-and-Drop at all, but it is necessary to push the node at each tick, otherwise it will be ignored and no common event handling would be possible. However, to reduce overhead, this will only be done if tracked events are available.

- Tracked events are detected on the *Component Button*, and the `mComponentIsBeingDragged` flag is already set: Dragging has been started and is going on. The event position will be used to move the Component to the position of the pointing device. The new position is calculated by adding the current event position relative to its original event position to the current Component position.
- No tracked events are detected on the *Component Button*, and the `mComponentIsBeingDragged` flag is set: The Component has been released ("dropped"). It now has to ask the Drag Controller if there is a Container in the same family that is ready to "catch" the Component. If so, it tells the Drag Controller to mark the Component as dropped, otherwise it will be moved back to its original position.

```
1  Bool Gui::Component::ProcessLogicSelf(Graph::IProcessLogicTracker* tracker)
2  {
3      if (!Gui::Widget::ProcessLogicSelf(tracker))
4      {
5          return false;
6      }
7
8      if (mComponentButton->GetNumberOfTrackedEvents() > 0)
9      {
10         if (mComponentDragControllerNodeTarget.GetNumberOfNodes() > 0)
11         {
12             UInt32 eventId = mComponentButton->GetTrackedEventId(0);
13             Math::Vector<Real> position = mComponentButton->GetLocalEventPosition
                   (eventId);
14
15             if (!mComponentIsBeingDragged)
16             {
17                 mComponentDragControllerNodeTarget.GetNode(0)->BeginDrag(this,
                       eventId);
18                 mComponentDragOriginPositionX = position.x;
19                 mComponentDragOriginPositionY = position.y;
20                 mComponentBeforeDragPositionX = mComponentOuterDimension.
                       mPositionX;
21                 mComponentBeforeDragPositionY = mComponentOuterDimension.
                       mPositionY;
22                 mComponentIsBeingDragged = true;
23
24                 mComponentButton->SetPassEventsEnabled(true);
25             }
26             else
27             {
28                 Real positionX = mComponentOuterDimension.mPositionX + position.x
                       - mComponentDragOriginPositionX;
29                 Real positionY = mComponentOuterDimension.mPositionY + position.y
                       - mComponentDragOriginPositionY;
30
31                 SetPosition(positionX, positionY);
32             }
33         }
34
35         tracker->PushNodeToFinishLogic(this);
36     }
37     else if (mComponentIsBeingDragged)
38     {
39         Gui::IDragController* dragController = mComponentDragControllerNodeTarget
               .GetNode(0);
40
41         if (!dragController->IsAboveDropContainer(this))
42         {
```

```
43              dragController->EndDrag(this, false);
44              SetPosition(mComponentBeforeDragPositionX,
                    mComponentBeforeDragPositionY);
45          }
46          else
47          {
48              dragController->EndDrag(this, true);
49          }
50
51          mComponentButton->SetPassEventsEnabled(true);
52
53          mComponentDragOriginPositionX = Real(0);
54          mComponentDragOriginPositionY = Real(0);
55          mComponentIsBeingDragged = false;
56      }
57
58      return true;
59  }
```

Listing 4.40: Implementation of `Gui::Component::ProcessLogicSelf()`

Similar to Components, the `ProcessLogicSelf()` method is also used in Containers for Drag-and-Drop handling. It asks its Drop Controller (if any) how many dragged Component are registered. The Drag Controller and the Drop Controller are the same object, so it keeps track of all Components that were registered through `BeginDrag()`, but not yet unregistered with `EndDrag()`. If there are dragged Components, the Container queries every single event ID that has been propagated and checks if the appropriate event is also inside its own boundary. This is done by the method `IsEventInside()` invoked on the *Component Button* of the Container. If the event is inside (and thus the Component "above" the Container), the Container registers itself as drop target and sets an internal flag to remember its registration state. `RegisterDropContainer()` will cause a positive answer when the dragged Component asks if it `IsAboveDropContainer()`. Finally, if no Components are dragged, but the flag has been set, indicating a released drag, the Container retrieves a pointer to the dropped Component from its Drop Controller (`CatchDroppedComponent()`) and puts it into the right place before unregistering itself as drop target. The whole process is listed in 4.41.

```
1   Bool Gui::Container::ProcessLogicSelf(Graph::IProcessLogicTracker* tracker)
2   {
3       if (!Component::ProcessLogicSelf(tracker))
4       {
5           return false;
6       }
7
8       if (mContainerDropControllerNodeTarget.GetNumberOfNodes() > 0)
9       {
10          Gui::IDropController* dropController = mContainerDropControllerNodeTarget
                .GetNode(0);
11          SInt32 numberOfDraggedComponents = dropController->
                GetNumberOfDraggedComponents();
12
13          if (numberOfDraggedComponents > 0)
14          {
15              for (SInt32 index = 0; index < numberOfDraggedComponents; index++)
16              {
17                  UInt32 eventId = dropController->GetDragEventId(index);
18
19                  if (mComponentButton->IsEventInside(eventId))
20                  {
```

```
21                      if (!mContainerIsTrackingDragComponent)
22                      {
23                          dropController->RegisterDropContainer(this, eventId);
24                          mContainerIsTrackingDragComponent = true;
25                      }
26                  }
27                  else
28                  {
29                      if (mContainerIsTrackingDragComponent)
30                      {
31                          dropController->UnregisterDropContainer(this);
32                          mContainerIsTrackingDragComponent = false;
33                      }
34                  }
35              }
36          }
37          else if (mContainerIsTrackingDragComponent)
38          {
39              UInt32 eventId = 0;
40              Gui::IComponent* component = dropController->CatchDroppedComponent(
                    this, &eventId);
41              Math::Vector<Real> position = mComponentButton->GetLocalEventPosition
                    (eventId);
42
43              component->GetNodeInterface()->GetParent(0)->RemoveChild(component->
                    GetNodeInterface());
44              component->SetPosition(mComponentOuterDimension.mPositionX,
                    mComponentOuterDimension.mPositionY);
45
46              dropController->UnregisterDropContainer(this);
47              mContainerIsTrackingDragComponent = false;
48          }
49      }
50
51      return true;
52  }
```

Listing 4.41: Implementation of `Gui::Container::ProcessLogicSelf()`

Unfortunately, the Drag-and-Drop implementation is not working correctly due to framework restrictions: It is not possible to change the hierarchy of graph nodes after initialization. Other strategies have been tested, but led to an incomplete support of Drag-and-Drop. For example, updating the transformation of the Component according to the targeted Container (for demonstration purposes, this workaround has been implemented in the code listed above) only works on GUIs without any animation. Since there will be no direct parent-child relationship between Container and Component, scene graph transformations of the Container will not be passed on to the Component, or at least not the way the user expects. An update of the ME regarding this behavior is required to make Drag-and-Drop work.

4.3 Event Handling

This section covers implementation details of the event handling subsystem. It is the only self-contained subsystem of the MGT and does not rely on other component, while the Widgets and Entities rely on this messaging approach.

4.3.1 Event Triggers

Event Triggers are implementations of the `Gui::IEventTrigger` interface and are used to identify the origin of an Event. The only property described by the interface is the tag property, a custom string that helps identifying Event Trigger instances. In the MGT, `Gui::IEventTrigger` is extended by the `Gui::IEntity` and `Gui::IWidget` base interfaces. Therefore, all Entities and Widgets are Event Triggers, which allows them to dispatch Events as described in section 3.2.4. The getter and setter methods for the tag property as well as the property itself are implemented by the respective base classes `Gui::Event` and `Gui::Widget`. Furthermore, Widget node types are able to parse the `tag` attribute that may be specified in XML resources.

4.3.2 Events and Event Handlers

Events are concrete implementations of the public interface `Gui::IEvent` and are informative objects carrying data about certain events that occurred during the most recent tick. The whole process of dispatching is unaware of the actual Event type, even the `Perform()` method of an Event Handler only receives a pointer to the `Gui::IEvent` interface. The developer is therefore responsible for casting and validating Event objects properly. The `Gui::IEventHandler` interface only has one concrete implementation (see below) and is mainly provided for developers by allowing them to wrap custom methods into Event Handlers.

This section will not focus on the implementation of concrete Event types, since they only consist of properties and accessor methods. The class implementations are not part of the public interface, as it is not intended for developers to create Event objects at will. However, their properties can be read by casting a `Gui::IEvent` pointer to the public interface of the respective Event type. Note that shared pointers are used instead of raw pointers, thus Events must be cast by calling the `DynamicCast()` method of the pointer instance. An example is shown in listing 4.42.

```
1 Bool EntityEventHandler::Perform(Gui::IEvent::ConstSharedPtr event)
2 {
3     Gui::IEntityEvent::ConstSharedPtr entityEvent = Gui::IEntityEvent::
          ConstSharedPtr::DynamicCast(event);
4     const Gui::IEntity* Entity = entityEvent->GetEntity();
5
6     // perform logic here
7
8     return true;
9 }
```

Listing 4.42: Example of how to use Event objects.

Event objects are directly created by Event Triggers through the constructor of the Event type. Since there are no setter methods defined, the complete Event description must be passed as constructor arguments. Event Triggers are also responsible for wrapping the objects into shared pointers and pass them to the pipeline for further processing.

Callback Event Handlers: Writing classes for each Event Handler required by an application may lead to an unhandy number of small classes. The alternative is writing a single Event Handler for an arbitrary number of Events and distinguish between Event Triggers through their tag property and between Event types through dynamic casting. Despite the probably undesired growth of line numbers, string comparison may become rather expensive if many Events and Event Triggers are involved. Both strategies are commonly used in Java. However, the MGT makes use of function pointers to provide a more flexible way of defining Event Handlers. Any method can be used as Event Handler as long as it has the same parameters and return type as the `Gui::IEventHandler::Perform()` method. A pointer to the method and to an instance the method shall be called on will be passed to the constructor of the template class `Gui::CallbackEventHandler`. Callback Event Handlers are in fact Event Handlers that only invoke method calls on objects in their implementation of `Perform()`, forwarding the Event object to the actual event handling method. The full implementation is available in listing 4.43.

```
1  template<class TargetClass>
2  class CallbackEventHandler : public IEventHandler
3  {
4  public:
5      CallbackEventHandler(TargetClass* instance, Bool (TargetClass::*method)(
           IEvent::ConstSharedPtr))
6      : mInstance(instance)
7      , mMethod(method)
8      {
9      }
10
11     virtual Bool Perform(IEvent::ConstSharedPtr event)
12     {
13         return (mInstance->*mMethod)(event);
14     }
15
16 protected:
17     TargetClass* mInstance;
18     Bool (TargetClass::*mMethod)(IEvent::ConstSharedPtr);
19 };
```

Listing 4.43: Implementation of the `CallbackEventHandler` template class

4.3.3 Event Channels

`Gui::EventChannel` is used to create unique identifiers for each Event type that may occur on a certain Event Trigger. It only stores a consecutive identification number and the number of registered Event Handlers. The Event Dispatch Table (see below) is responsible for managing the assignment of Event Handlers to Event Channels and takes care of the subscriber count, which represents the total number of Event Handlers registered for a certain Event Channel. The subscriber count is a protected member, but it is possible to query if at least one Event Handler is registered (`Gui::EventChannel::HasSubscribers()`). This is a performance measure for Event Triggers, allowing them to ignore events if the Event Channel has no subscribers. Listing 4.44 contains the declaration of `Gui::EventChannel`.

```
class EventChannel
{
public:
    EventChannel();
    virtual ~EventChannel() {}

    virtual UInt64 GetId() const;

    virtual Bool IncrementSubscriberCount();
    virtual Bool DecrementSubscriberCount();
    virtual Bool ResetSubscriberCount();

    virtual Bool HasSubscribers() const;

protected:
    static UInt64 mNextId;
    UInt64 mId;
    UInt32 mSubscriberCount;
};
```

Listing 4.44: Declaration of the `EventChannel` class

Event Channels are usually instantiated as member variables of Event Triggers, so no explicit initialization is necessary. As a consequence, destroying the Event Trigger will also result in destroying the Event Channel. So, the only thing an Event Trigger must be aware of is to pass the appropriate Event Channel reference when registering or unregistering Event Handlers or dispatching Events. This process is described in more detail in the following section.

4.3.4 Event Dispatch Table

The singleton instance of `Gui::EventDispatchTable` is accessible through `Gui::EventDispatchTable::GetInstance()` and manages the mapping from Event Channels to arrays of Event Handlers. The Event Dispatch Table is used by Event Triggers to forward the registration of Event Handlers by calling `Gui::EventDispatchTable::RegisterEventHandler()` and passing the appropriate Event Channel. This is exemplified in listing 4.45, which shows the registration of Event Handlers for Point Events on Components. Every registration method works the same way.

```
Bool Gui::Component::RegisterPointEventHandler(const Gui::IEventHandler*
    eventHandler)
{
    Gui::EventDispatchTable* eventDispatchTable = Gui::EventDispatchTable::
        GetInstance();

    if (eventDispatchTable != 0 && eventHandler != 0)
    {
        return eventDispatchTable->RegisterEventHandler(*eventHandler,
            mPointEventChannel);
    }

    return false;
}
```

Listing 4.45: Implementation of `Gui::Component::RegisterPointEventHandler()`

The `Gui::EventDispatchTable::RegisterEventHandler()` method is shown in listing 4.46. Each Event Channel is mapped to an array of Event Handlers. The registration method creates the array if not available and pushes the Event Handler on top. Finally, the subscriber count of the Event Channel is incremented. The unregistration method works similar and reverts the steps done on registration, which includes decrementing the subscriber count.

```
1  Bool Gui::EventDispatchTable::RegisterEventHandler(const IEventHandler&
       eventHandler, EventChannel& eventChannel)
2  {
3      SInt32 index = mEventHandlerMap.Find(eventChannel.GetId());
4      EventHandlerArray* eventHandlers = 0;
5      if (index == -1)
6      {
7          eventHandlers = new EventHandlerArray();
8          mEventHandlerMap.Add(eventChannel.GetId(), eventHandlers);
9      }
10     else
11     {
12         eventHandlers = mEventHandlerMap.Get(mEventHandlerMap.GetKey(index));
13     }
14
15     if(eventHandlers->Find(const_cast<IEventHandler*>(&eventHandler)) == -1)
16     {
17         eventHandlers->Add(const_cast<IEventHandler*>(&eventHandler));
18         eventChannel.IncrementSubscriberCount();
19     }
20
21     return true;
22 }
```

Listing 4.46: Implementation of `Gui::EventDispatchTable::RegisterEventHandler()`

Following up the registration, Event Triggers will dispatch Events to the Event Pipeline, which also uses the Event Dispatch Table to lookup the associated Event Handlers. This process is described in the next section.

4.3.5 Event Pipeline

The `Gui::EventPipeline` class manages a singleton that is accessible through `Gui::EventPipeline::GetInstance()` for Event Triggers to dispatch Event objects by passing them to the method `Gui::EventPipeline::DispatchEvent()`. An Event Pipeline holds a pointer to the Event Dispatch Table singleton, which will be used to get an array of Event Handlers that are associated with the Event Channel passed together with the Event object to dispatch. The dispatch logic iterates over the Event Handlers and calls their `Perform()` method, passing the Event object as parameter. The code is straight forward and shown in listing 4.47. Note that the Event object is passed as constant shared pointer. It is constant because Event Handlers are not allowed to change the information supplied by the Event Trigger. The decision of using shared pointers is based upon the fact that it is not known what happens to the Event objects after performing the unknown Event Handler logic. It is possible, though not likely, that they will keep track of the object beyond the current tick.

```
1  void Gui::EventPipeline::DispatchEvent(Gui::IEvent::ConstSharedPtr event, const
       Gui::EventChannel& eventChannel)
2  {
3      const EventHandlerArray* eventHandlers = mEventDispatchTable->
           GetEventHandlersOfEventChannel(eventChannel);
4      if (eventHandlers != 0)
5      {
6          for (SInt32 index = 0; index < eventHandlers->GetCount(); index++)
7          {
8              IEventHandler* eventHandler = (*eventHandlers)[index];
9              eventHandler->Perform(event);
10         }
11     }
12 }
```

Listing 4.47: Implementation of`Gui::EventPipeline::DispatchEvent()`

This class has no public interface and is therefore not available in user code.

4.3.6 Event Handler Table

The class `Gui::EventHandlerTable` is public and can be used by developers to reduce effort in connecting Widgets from XML resources with Event Handlers. In contrast to the Event Dispatch Table, which maps Event Channels to Event Handlers, an Event Handler Table maps a global string identifier to an Event Handler. Finding the corresponding Event Handler to an event does not happen during process time (i.e., after the initialization and before the termination of the application logic) but on initialization of the Widget. If the Widget has an Event Handler property (or XML attribute) set, the Widget tries to get a pointer to the Event Handler from the table and, in case of success, associates this Event Handler with its corresponding Event Channel. The code in listing 4.48 shows this process for `Gui::Component`s and Point Events. The steps are the same for all combinations of Widget types and Event types as summarized in table 3.2.

```
1  if (mComponentPointEventHandlerIdentifier.GetLength() > 0)
2  {
3      Gui::EventHandlerTable* eventHandlerTable = Gui::EventHandlerTable::
           GetInstance();
4      const Gui::IEventHandler* eventHandler = eventHandlerTable->GetEventHandler(
           mComponentPointEventHandlerIdentifier);
5
6      eventHandlerTable->RegisterEventHandler(*eventHandler, mPointEventChannel);
7  }
```

Listing 4.48: Example of Event Handler registration.

The implementation of `Gui::EventHandlerTable` is nothing more than a wrapper for the internally used map object that is of type `Map<String, const Gui::IEventHandler*>`. However, the actual comfort is brought by a macro defined in the header file:

```
1  #define MURL_GUI_REGISTER_EVENT_HANDLER(object, className, methodName) \
2  { \
3      Gui::CallbackEventHandler<className>* eventHandler = new Gui::
           CallbackEventHandler<className>(object, &className::methodName); \
```

```
4     Gui::EventHandlerTable::GetInstance()->RegisterEventHandler(#className "::" #
          methodName, eventHandler); \
5 }
```

Listing 4.49: Definition of macro to register an Event Handler.

This macro requires a method name of a method that may act as a callback of a Callback Event Handler, the class name the method belongs to, and a concrete object that is an instance of that class. An instance of `Gui::CallbackEventHandler` is then created and registered as Event Handler identified by a string composed from the class name, a double colon, and the method name, similar to the C++ notation of function references. A Widget defined the way as shown in listing 4.50 will be properly set up by using the code in listing 4.51 on app initialization.

```
1 <Gui::Button id="myButton" pointEventHandler="MyLogic::MyPointEventHandler"/>
```

Listing 4.50: Example of how to assign global Event Handler identifiers in XML resource files.

```
1 MURL_GUI_REGISTER_EVENT_HANDLER(mMyLogic, MyLogic, MyPointEventHandler);
```

Listing 4.51: Example of how to register Callback Event Handlers with an automatically composed global identifier.

`mMyLogic` is an instance of the example logic class `MyLogic`, which has a method named `MyPointEventHandler()` that will act as a callback on Point Events for the Button named `myButton`. The Event Handler will be identified by the string `MyLogic::MyPointEventHandler`, which will be looked up by the Button as consideration of its Point Event Handler XML attribute. For reasons of proper object deallocation, the macro's counterpart needs to be called on termination of the app and is defined as follows:

```
1 #define MURL_GUI_UNREGISTER_EVENT_HANDLER(className, methodName)\
2 { \
3     Gui::IEventHandler* eventHandler = Gui::EventHandlerTable::GetInstance()->
          GetEventHandler(#className "::" #methodName); \
4     if (eventHandler != 0) \
5     { \
6         Gui::EventHandlerTable::GetInstance()->UnregisterEventHandler(#className
              "::" #methodName); \
7         delete eventHandler;\
8     } \
9 }
```

Listing 4.52: Definition of macro to unregister an Event Handlers.

The curly brackets are used to restrict the scope of the local variable `eventHandler`.

4.4 Entities

The abstract Entity base class `Gui::Entity` is a public implementation of `Gui::IEntity` that has already implemented some pure virtual methods common to all Entity types. The class declaration is as follows:

```
1  class Entity : public IEntity
2  {
3      public:
4          Entity();
5          virtual ~Entity();
6
7          virtual IEntity* GetEntityInterface();
8          virtual const IEntity* GetEntityInterface() const;
9
10         virtual void ResetData();
11
12         virtual Bool RegisterEntityEventHandler(const IEventHandler* eventHandler
               );
13         virtual Bool UnregisterEntityEventHandler(const IEventHandler*
               eventHandler);
14
15         virtual Bool SetTag(const String& tag);
16         virtual const String& GetTag() const;
17
18     protected:
19         virtual void Notify() const;
20
21     private:
22         class Private;
23
24         Private* mPrivateThis;
25 };
```

Listing 4.53: Declaration of the `Gui::Entity` class

There is not much logic behind these methods. `ResetData()` is supposed to be called by overriding methods to unify the effect of resetting Entity data. The effect is the invocation of an Entity Event by calling `Notify()` in the base class implementation, while subclasses are responsible to reset data to a proper initial state. The `Notify()` method can be used by subclasses, whenever it is needed to send Entity Events, especially after data has been modified. The following piece of code shows how Entity Events are created and dispatched by `Notify()`:

```
1  void Gui::Entity::Notify() const
2  {
3      const Gui::IEntityEvent* event = new Gui::EntityEvent::EntityEvent(this, this
           );
4      Gui::IEvent::ConstSharedPtr pointer(event->GetEventInterface());
5      Gui::EventPipeline::GetInstance()->DispatchEvent(pointer, mPrivateThis->
           mEntityEventChannel);
6  }
```

Listing 4.54: Implementation of `Gui::Entity::Notify()`

Additionally, trivial setter and getter methods for the tag property specified by the `Gui::IEventTrigger` interface and Event Handler registration methods are implemented by `Gui::Entity`. The method `GetSerializedData()` is abstract and needs

to be implemented by the subclasses to generate a suitable string representation of the encapsulated data.

The declaration of this class is considered to be public to allow Entity subclassing for anyone. However, some implementation details must remain private, because of its connection to the private event handling system. While the access to the dispatcher happens in the C++ file that will be compiled into the MGT binary, the declaration of an Event Channel property would require `Gui::EventChannel` to be public. This has been solved by a forward declaration of the private class `Gui::Entity::Private` and the declaration of a pointer to an instance of this class. Declaring pointers to forward declared types is a legal technique in C++. Listing 4.53 shows this at the declaration of `mPrivateThis`. Listing 4.55 defines the private class and allocates/deallocates the object, which has already been used in listing 4.54.

```
1  class Gui::Entity::Private
2  {
3  public:
4      EventChannel mEntityEventChannel;
5
6      String mEventTriggerTag;
7  };
8
9  Gui::Entity::Entity()
10 : mPrivateThis(new Gui::Entity::Private())
11 {
12 }
13
14 Gui::Entity::~Entity()
15 {
16     delete mPrivateThis;
17 }
```

Listing 4.55: Definition and instantiation of the `Gui::Entity::Private` class

4.4.1 Primitive Entities

The Control Widgets of the MGT require the following primitive Entity types (i.e. Entities that wrap primitive data types): `Gui::NumberEntity`, `Gui::SwitchEntity`, and `Gui::TextEntity`. A Switch Entity just wraps a boolean value and provides convenient property accessors and mutators, for example `Toggle()` to invert the value. Text Entities also provide some string manipulation functions like `AppendString()` and allows to set or get the maximum string length. As noted before, Entity types must also implement `GetSerializedData()` declared by `Gui::IEntity`. While Text Entities simply pass through their wrapped string, a Switch Entity returns the string `true` or `false` according to the value. Both Entity types will emit Entity Events after manipulating the boolean or string value by calling the parent's `Notify()` method.

Number Entities are a bit more complicated as they must handle both integer and floating point types. Beside the number itself, they also store the boundaries (min-

imum and maximum values) and the step size for incrementing and decrementing the value. These additional properties are a consideration of the Controls which use Number Entities as their default Entity, namely `Gui::Slider` and `Gui::Stepper`, and can be accessed or modified by the following methods:

```
virtual Bool Decrement();
virtual Bool Increment();
virtual Number GetMaximum() const;
virtual Number GetMinimum() const;
virtual Number GetStep() const;
virtual Number GetValue() const;
virtual Real GetNormalizedValue() const;
virtual Bool SetMaximum(Number maximum);
virtual Bool SetMinimum(Number minimum);
virtual Bool SetStep(Number step);
virtual Bool SetValue(Real value);
virtual Bool SetValue(SInt64 value);
```

Listing 4.56: Public methods of `Gui::NumberEntity`

Through a Stepper, a user is able to increment and decrement the value as well as directly typing it into the number field. Both increment and decrement are limited by the maximum and the minimum value, moving toward the limits by the step size on each button press. A Slider, however, is not interested in absolute values, but in relative ones, since the thumb is located on a fixed-size track that is unaware of the actual Entity boundaries. Thus, `GetNormalizedValue()` will return a value between 0 and 1, where the minimum value is mapped to 0 and the maximum to 1. The implementation of the Number Entity is straight forward and done in a few lines of code, since the utility functions of the ME help to reduce common coding tasks. As an example, see the implementation of the required `GetSerializedData()` method:

```
String Gui::NumberEntity::GetSerializedData() const
{
    if(mValue.IsReal())
    {
        return Util::DoubleToString(static_cast<const Double>(mValue));
    }
    else
    {
        return Util::SInt64ToString(static_cast<const SInt64>(mValue));
    }
}
```

Listing 4.57: Implementation of `Gui::NumberEntity::GetSerializedData()`

The `SetValue()` methods will call `Notify()` to fire an Entity Event, but only if the previous value differs from the new one. This also happens on `Increment()` and `Decrement()`, as both methods internally call `SetValue()` to add or subtract the step size.

Note the `Gui::Number` type that occurs in listing 4.56. This class was introduced by the MGT to simplify the handling of numbers that are not specified as integers or floats explicitly. This approach tries to compensate the possible inadequacy of integers and the precision noise of floating point values when obtaining the value

from the Entity. The casting operators for SInt64 and Real are overridden, allowing the developer to query the value as a type that matches her needs. The implementation is not complete in regard to operator overriding but is a foundation to more advanced use cases.

4.4.2 Selections

The Selection classes Gui::Selection and Gui::EntitySelection are subclassed from Gui::Entity. The general implementation Gui::Selection does not describe a selection of concrete objects, but a selection of arbitrary items, which are neither defined nor accessible. The set of items is described by its count value, with zero indicating the empty set, and a virtual selection index, with -1 meaning no item is selected. Although this class seems to be trivial, all pure virtual methods of the Gui::ISelection are implemented with a proper behavior (see figure 3.3 for an overview). Beside common getter, setter, and Event Handler registration methods, Add() increments the counter and ResetData() calls Empty(), which sets the counter to 0 and the selected index to -1. This lightweight Selection type is suitable for selection Widgets like Option Buttons, if the developer is only interested in the selected index rather than any data. In general, the index of an Option Button within a Selection is the actual data.

It is possible to associate selection indices with concrete data objects. This will be useful for more sophisticated selection Widgets like List Views, which also present data. In such situations, developers are encouraged to use more sophisticated subclasses instead of Gui::Selection. The MGT comes along with an example called Gui::EntitySelection. Keep in mind that Selections are Entities themselves, so an Entity Selection is an Entity that maintains a list of other Entities. The following methods were implemented for Gui::EntitySelection as an extension to Gui::Selection:

```
1 virtual Bool Add(IEntity* entity);
2 virtual Bool Remove(IEntity* entity);
3 virtual Bool Set(SInt32 index, IEntity* entity);
4 virtual IEntity* Get(SInt32 index);
5 virtual const IEntity* Get(SInt32 index) const;
6 virtual SInt32 Find(const IEntity* entity) const;
```

Listing 4.58: Public methods of Gui::EntitySelection

These methods are simple wrappers for the underlaying array (which is of type Murl::Array) that contains the pointers to the Entities known to the Selection. The developer can access the listed Entities for example in Event Handlers by casting the Event to a Selection Event and read both the selected index and the Gui::IEntity object stored at the index.

4.5 Skinning

The skinning concept of the MGT was introduced in section 3.6. As already mentioned, the process of skin creation is kept compliant to asset handling, as it is done by the ME. The regular package format of the engine will be used to define skin packages, which is completely sufficient for this purpose. This section now focuses on how the graph nodes of the skin packages are used by Widgets. Applying a skin to a Widget is done by setting up the geometry and the State Sets as described below.

4.5.1 Loading the Skin Package

Loading a Skin package is similar to loading any other package in the ME. The example below includes a skin called `gui_skin`. Despite this custom package, an additional mandatory package called `gui_base` must be loaded before. This package contains basic resources (e.g., animations, or materials) independent from the skin.

```
1 Bool App::SomeApp::Init(const IAppState* appState)
2 {
3     ILoader* loader = appState->GetLoader();
4     loader->AddPackage("gui_base", ILoader::LOAD_MODE_LOAD_MODE_STARTUP, 0);
5     loader->AddPackage("gui_skin", ILoader::LOAD_MODE_LOAD_MODE_STARTUP, 0);
6
7     return true;
8 }
```

Listing 4.59: Code for loading Skin and GUI package.

4.5.2 Configure Geometries

The following piece of code defines three Buttons with different geometry types:

```
1 <Gui::Button id="button1" geometryType="MULTIPATCH" multipatchAtlasResourceId="
      gui_skin:button_atlas" stateSetId="/Gui/Skin/Button"/>
2 <Gui::Button id="button2" geometryType="PLANE" stateSetId="/Gui/Skin/
      CustomButtonStateSet"/>
3 <Gui::Button id="button3" geometryType="REFERENCE" referenceTargetId="/Gui/Skin/
      CustomButtonPlane"/>
```

Listing 4.60: Example of how to define Widgets (here: Buttons) with different geometry types in XML.

Depending on the attribute `geometryType`, further attributes are required. They are parsed to properties defined and handled by `Gui::Component`, allowing their use for all Widgets that are Component types. However, even if a certain property is set, it only has an effect on the rendering if the respective geometry type is used. Their correct usage is demonstrated in the listing above. The following list provides a summary on them:

1. `multipatchAtlasResourceId` refers to an atlas resource that defines the layout of a multi-patch plane. The atlas rectangles are processed by `Gui::MultipatchFactory` as described below.

2. `stateSetId` contains the node ID of the State Set container node that shall be referred to in order to configure the rendering process of the Widget's geometry. This property is both used for multipatch and plane geometries.

3. `referenceTargetId` contains the node ID of a prepared geometry node that shall be referred to in order to generate output for the Widget.

These properties can also be mutated by calling the appropriate setter methods on Components. This only works with instances that have not been initialized yet, because it depends on the configuration which nodes will be dynamically created for the Widget subgraph. The ME does not allow attaching or detaching nodes to an initialized graph, thus no modifications are possible by the engine.

4.5.3 Generate Geometries

Generating the geometry (or the reference) is done in the virtual protected method `Gui::Component::SetupGeometry()`, which is called by `Gui::Component` during initialization. Subclassed Widgets may override this method. However, most Widgets that do so just execute custom code to create further geometries (e.g., the thumb of a Slider) and invoke the parent method additionally. If they also need to prepare the main geometry (*Component Geometry*, referenced by `mComponentGeometry`), the default geometry creation must be omitted. In all other cases, the following code will be executed:

```
1  void Gui::Component::SetupGeometry(Graph::IInitTracker* tracker)
2  {
3      if (mComponentGeometry == 0)
4      {
5          Gui::ComponentGeometryType geometryType =
               GetEffectiveComponentGeometryType();
6
7          if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
8          {
9              [...]
10         }
11         else if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_PLANE)
12         {
13             [...]
14         }
15         else if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_REFERENCE)
16         {
17             [...]
18         }
19     }
20
21     mComponentContentHook->AddChild(mComponentGeometry);
22 }
```

Listing 4.61: Structure of `Gui::Component::SetupGeometry()`

The most complicated geometry type is the patched geometry. The complexity has been hidden from the Component and outsourced to `Gui::MultipatchFactory` which will be described later on. The Component is still responsible for loading the atlas resource and passing the rectangles and the Component dimension to the factory before retrieving the generic geometry. The factory manages the geometry for the whole life-time of the Component, but the Component must propagate size updates for the changes to be applied on the geometry.

```
1  if(mComponentStylePatchedRectangles == 0 && mComponentMultipatchAtlasResourceId.
       GetLength() != 0)
2  {
3      const Resource::ICollection* collection = tracker->GetResourceCollection();
4      mComponentStylePatchedAtlas = collection->GetAtlas(
           mComponentMultipatchAtlasResourceId);
5      mComponentStylePatchedRectangles = mComponentStylePatchedAtlas->GetRectangles
           ();
6
7      if(mComponentStylePatchedRectanglesCount == 0)
8      {
9          mComponentStylePatchedRectanglesCount = mComponentStylePatchedAtlas->
               GetNumberOfRectangles();
10     }
11 }
12
13 mComponentMultipatchFactory.Init(tracker->GetRoot(),
       mComponentStylePatchedRectangles,
14     mComponentOuterDimension.mSizeX, mComponentOuterDimension.mSizeY,
15     mComponentStylePatchedRectanglesCount, mComponentStylePatchedRectangleOffset)
           ;
16
17 mComponentGeometry = mComponentMultipatchFactory.GetMultipatchPlane()->
       GetNodeInterface();
18 mComponentInnerDimension = mComponentMultipatchFactory.GetInnerDimension();
```

Listing 4.62: Implementation of the multi-patch geometry generation in `Gui::Component::SetupGeometry()`

In contrast to multi-patch geometries, planes and references are set up pretty straight forward. The ME node types `Graph::PlaneGeometry` and `Graph::Reference` are used for this purpose:

```
1  Graph::IPlaneGeometry* planeGeometry = dynamic_cast<Graph::IPlaneGeometry*>(
       tracker->GetRoot()->CreateNode("PlaneGeometry"));
2
3  planeGeometry->SetScaleFactor(mComponentOuterDimension.mSizeX,
       mComponentOuterDimension.mSizeY, Real(1));
4
5  mComponentGeometry = planeGeometry->GetNodeInterface();
6  mComponentInnerDimension = mComponentOuterDimension;
```

Listing 4.63: Implementation of the plane geometry generation in `Gui::Component::SetupGeometry()`

```
1  Graph::IReference* reference = dynamic_cast<Graph::IReference*>(tracker->GetRoot
       ()->CreateNode("Reference"));
2  Graph::INode* target = tracker->GetRoot()->FindNode(mComponentReferenceTargetId);
3
4  reference->GetNodeTarget()->SetNode(target);
5
6  mComponentGeometry = reference->GetNodeInterface();
```

```
7 mComponentInnerDimension = mComponentOuterDimension;
```

Listing 4.64: Implementation of the geometry reference generation in `Gui::Component::SetupGeometry()`

An appropriate node object is created and configured according to the passed attributes. The node is then assigned to `mComponentGeometry`, which will later be used to build the subgraph of the Widget. Finally, the inner dimension is set to the outer dimension, since planes and referenced geometries do not specify any insets or borders. The pointer type of `mComponentGeometry` is `Graph::INode`, the common base interface of all nodes. Whenever there is mutating access to the geometry, a distinction of cases has to be done to accommodate the different mutation patterns of the three geometry types. In most cases, the reference type will be ignored, since the target node is not intended to be modified. The following listing, for example, is an excerpt from `Gui::Component::ApplySizeUpdate()` and demonstrates the consideration of different geometry types in case of size updates. The *Component Button* defines the input area and does not depend on the geometry. However, generic geometries are modified through the multi-patch factory they belong to, while the size of plane geometries is set directly.

```
1  ComponentGeometryType geometryType = GetEffectiveComponentGeometryType();
2
3  mComponentButton->SetScaleFactor(mComponentOuterDimension.mSizeX,
       mComponentOuterDimension.mSizeY, 1.0f);
4
5  if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
6  {
7      mComponentMultipatchFactory.SetSize(mComponentOuterDimension.mSizeX,
           mComponentOuterDimension.mSizeY);
8  }
9  else if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_PLANE)
10 {
11     Graph::IPlaneGeometry* planeGeometry = dynamic_cast<Graph::IPlaneGeometry*>(
           mComponentGeometry);
12     planeGeometry->SetScaleFactor(mComponentOuterDimension.mSizeX,
           mComponentOuterDimension.mSizeY, Real(1));
13 }
```

Listing 4.65: Example of handling different geometry types in `Gui::Component::ApplySizeUpdate()`

The `Gui::Component::SetupGeometry()` method introduced above is sufficient for most Component types. However, some overriding methods exist that do not even call the method of the base class because of a completely different appearance behavior. As an example, consider `Gui::CheckSwitch`, the checkbox implementation of the MGT. A checkbox is represented by an image that does not vary in size or content among its instances within the same application. But, usually, checkboxes react to mouse events, e.g., by highlighting on hover. Also there are two possible states to represent: on and off. To sum up, the implementation must handle the following requirements:

1. Only use References to image geometries defined in the Skin package since there is no need for resizable planes.

2. Switch between different References according to the button state.

3. Switch between the images representing a boolean state for each button state.

This is why Widgets like Check Switch or Option Button differ from most other Component types and require different setup routines (an example has already been shown in listing 4.24) to generate a subgraph like the one illustrated in figure 4.6.

Multipatch Factory

Patched geometries are instances of the `Graph::GenericGeometry` Node. This class allows customized composition of vertex data and access to the geometry's vertex buffer and index buffer. The MGT uses a factory class for creating and managing generic geometries according to the needs of Widgets which use a patched geometry: `Gui::MultipatchFactory`. This is a pure helper class for implementing the toolkit and not available via the public interfaces. Its purpose is to hide the complexity of creating vertex and index buffer objects representing patched planes that were built from rectangle data provided in an atlas XML resource file. Further, it handles the manipulation of vertex data (most likely position attributes) on size updates after creation. From outside, the program simply needs to call `SetSize()`, as it is done in listing 4.65. The instantiation of the generic geometry node happens once for each multi-patch plane by calling the `Gui::MultipatchFactory::Init()` method, which is declared as follows:

```
1 virtual void Init(Graph::IRoot* root, const Resource::IAtlas::Rectangle*
      rectangles, Real sizeX, Real sizeY, UInt32 numberOfRectangles, UInt32
      rectangleOffset = 0);
```

Listing 4.66: Declaration of `Gui::MultipatchFactory::Init()`

The `root` points to the graph root and is used to instantiate nodes dynamically. `rectangle` points to an array of atlas rectangles. A rectangle is a section of a texture specified by the top-left and bottom right corner. `sizeX` and `sizeY` are the initial dimensions. `numberOfRectangles` tells how many rectangles shall be processed and `rectangleOffset` is used when multiple definitions of patched planes occur in the same atlas and texture. After configuring the internal used parameters, the `Init()` method instantiates a `Graph::GenericGeometry` node, allowing the program to create and manage VBOs and IBOs comfortably. The following piece of code gives an idea of how this works:

```
1 UInt32 byteOffset = 0;
2 mGeometry = dynamic_cast<Graph::IGenericGeometry*>(root->CreateNode("
      GenericGeometry"));
3 mGeometry->AddAttribute(IEnums::ATTRIBUTE_ITEM_COORD, IEnums::
      ATTRIBUTE_TYPE_FLOAT_VECTOR_3, byteOffset);
4 mGeometry->AddAttribute(IEnums::ATTRIBUTE_ITEM_TEXCOORD_0, IEnums::
      ATTRIBUTE_TYPE_FLOAT_VECTOR_2, byteOffset);
5 mGeometry->SetVertexByteSize(byteOffset);
6 mGeometry->SetIndexType(IEnums::INDEX_TYPE_UINT16);
7 mGeometry->SetPrimitiveType(IEnums::PRIMITIVE_TYPE_TRIANGLES);
8
9 mGeometry->SetMaxNumberOfVertices(mVertexCount, false);
```

```
10 mGeometry->SetNumberOfVertices(mVertexCount);
11 SetupVertexBuffer();
12
13 mGeometry->SetMaxNumberOfIndices(mIndexCount, false);
14 mGeometry->SetNumberOfIndices(mIndexCount);
15 SetupIndexBuffer();
```

Listing 4.67: Creating and configuring a generic geometry node in `Gui::MultipatchFactory::Init()`

The vertex attributes are added subsequently to the node by specifying a predefined variable binding and the data type. The byte size of the attribute will be added to an offset that contains the final vertex byte size after all attributes have been set. In this case, the vertices need a position and texture coordinates. The geometry will be composed of triangles, thus a triple of three indices will refer to a single polygon of the geometry. Note that an index must be a 16 bit unsigned integer to work correctly on with OpenGL ES on mobile devices. After setting the number of vertices and indices, a pointer to the reserved memory can be retrieved and filled with data. This is done in separate methods, `SetupVertexBuffer()` and `SetupIndexBuffer()`. The latter one is just a straight-forward implementation of iteratively adding six indices per patch, each with a certain offset:

```
1  void Gui::MultipatchFactory::SetupIndexBuffer()
2  {
3      UInt16* indexBuffer = static_cast<UInt16*>(mGeometry->GetIndexData());
4      UInt32 nextRow = mPatchesX + 1;
5      UInt32 index = 0;
6      UInt32 offset = 0;
7
8      for(UInt32 indexY = 0; indexY < mPatchesY; indexY++)
9      {
10         for(UInt32 indexX = 0; indexX < mPatchesX; indexX++)
11         {
12             indexBuffer[index++] = offset;
13             indexBuffer[index++] = offset + nextRow;
14             indexBuffer[index++] = offset + 1;
15             indexBuffer[index++] = offset + nextRow;
16             indexBuffer[index++] = offset + 1;
17             indexBuffer[index++] = offset + nextRow + 1;
18             offset++;
19         }
20         // Skip last vertex in row.
21         offset++;
22     }
23
24     mGeometry->SetIndicesModified();
25 }
```

Listing 4.68: Implementation of `Gui::MultipatchFactory::SetupIndexBuffer()`

For each row of patches and each patch in that row, six indices will be created, forming a rectangle patch composed of two triangles. With the top left vertex index of a patch given, the indices of the other three vertices can simply by calculated, since all vertices are indexed from left to right and from top to bottom as shown in figure 4.9.

A bit more logic is involved when generating vertex data. The first step is to calculate the horizontal and vertical vertex positions. The patches form a regular

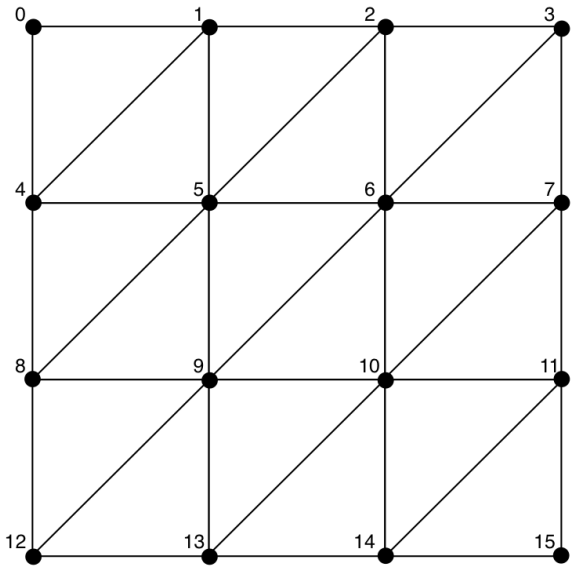

Figure 4.9: Vertex indices of a nine-patch geometry

grid consisting of columns and rows, so it is not necessary to calculate X and Y separately for each vertex. Z is always 1. The texture coordinates are directly taken from the rectangles passed initially. The vertex data is put into the array iteratively similar to feeding the IBO as it was done above. The most critical part here is the consideration of the aspect that makes patched planes so useful: the ability to stretch certain regions while keeping others constant. The flexible patches are determined implicitly for both directions, horizontal and vertical: Assuming the columns and rows of patches are numbered from 1 to the number of patches in one dimension. Patches with even column numbers are flexible on the horizontal axis, patches with even row numbers are flexible on the vertical axis. If the plane only consists of one patch per row, this patch will be stretched to the full width. The same holds for one patch per column, which will then stretch to the full height. For a better understanding, only the horizontal axis shall now be considered (e.g., 1×1, 3×1, or 5×1 planes). The reason for the flexibility alternating between neighbored patches comes from the insight that it does not make sense having two constant patches among each other, as they could simply be combined into one. A 3×1-patch, for example, will have a flexible second patch, while patch 1 and 3 have a constant width, making them borders. A 5×1-patch consists of a stretchable second and fourth patch, the first and fifth patches are now the borders and the third element is some kind of dividing space between the two flexible patches. Somebody might complain that an opposite configuration (e.g., patch 1 and 3 being flexible, patch 2 constant) is not possible. However, since the MGT does not need this variant and the class is private anyway, this possibility has been dropped. When now building the patches, the fixed amount of space is calculated for each dimension at first. This is done by summing up the distances between the u_1 and u_2 texture coordinates of odd numbered rectangles and subtracting it from the total width of the geometry. The remaining space is then equally divided up into the flexible patches. Optionally, the developer may pass an array of custom weighting values (which sum up to 1).

After initialization, the generic geometry may be obtained from the factory. Whenever a size update is necessary, the factory calls `SetupVertexBuffer()` again, overwriting the existing vertex data with new values.

4.5.4 Setup State Sets

Similar to geometry initialization, State Sets are set up during the initialization phase of the Widget node when `Gui::Component::SetupStateSet()` (or an overriding method) is called. The following implementation checks if the State Set has already been set by a subclass:

```
1  void Gui::Component::SetupStateSet(Graph::IInitTracker* tracker)
2  {
3      if (mComponentStateSet == 0)
4      {
5          Graph::IRoot* root = tracker->GetRoot();
6          Gui::ComponentGeometryType geometryType =
               GetEffectiveComponentGeometryType();
7
8          if (geometryType != Gui::IEnums::COMPONENT_GEOMETRY_TYPE_REFERENCE)
9          {
10             if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_MULTIPATCH)
11             {
12                 mComponentStateSetId = mComponentMultipatchStateSetId;
13             }
14             else if (geometryType == Gui::IEnums::COMPONENT_GEOMETRY_TYPE_PLANE)
15             {
16                 mComponentStateSetId = mComponentPlaneStateSetId;
17             }
18
19             if (mComponentStateSetId.GetLength() > 0)
20             {
21                 mComponentStateSet = root->CreateNode("Reference");
22                 dynamic_cast<Graph::IReference*>(mComponentStateSet)->
                       GetNodeTarget()->SetNode(root->FindNode(mComponentStateSetId)
                       );
23             }
24             else
25             {
26                 mComponentStateSet = CreateSimpleStateSet(root);
27             }
28         }
29     }
30
31     if (mComponentStateSet != 0)
32     {
33         mComponentButton->GetNodeInterface()->AddChild(mComponentStateSet);
34     }
35 }
```

Listing 4.69: Implementation of `Gui::Component::SetupStateSet()`

If it has not been set and the geometry type is no reference, the Node ID of the State Set according to the geometry type will be chosen to build a Reference to the Node. If no ID is set, a trivial State Set (for rendering a white plane) will be created in-place. Finally, the the created Node will be added to the Component subgraph. There are two different State Set node ID properties, one for multi-patch planes and one for simple planes. This is because of the default values defined for a certain Component. A Component that can handle both geometry types will have different

State Sets for each, e.g., a non-bordered texture for the plane type. This way, only the geometry type attribute has to be set when working with default State Sets.

This concludes the chapter about implementation details of the MGT. In the next chapter, some sample applications will be shown in order to demonstrate the final results of the toolkit.

5 Results

This chapter will demonstrate four sample applications using the ME and the MGT, that has been introduced and discussed in the previous two chapters.

5.1 Widget Showcase

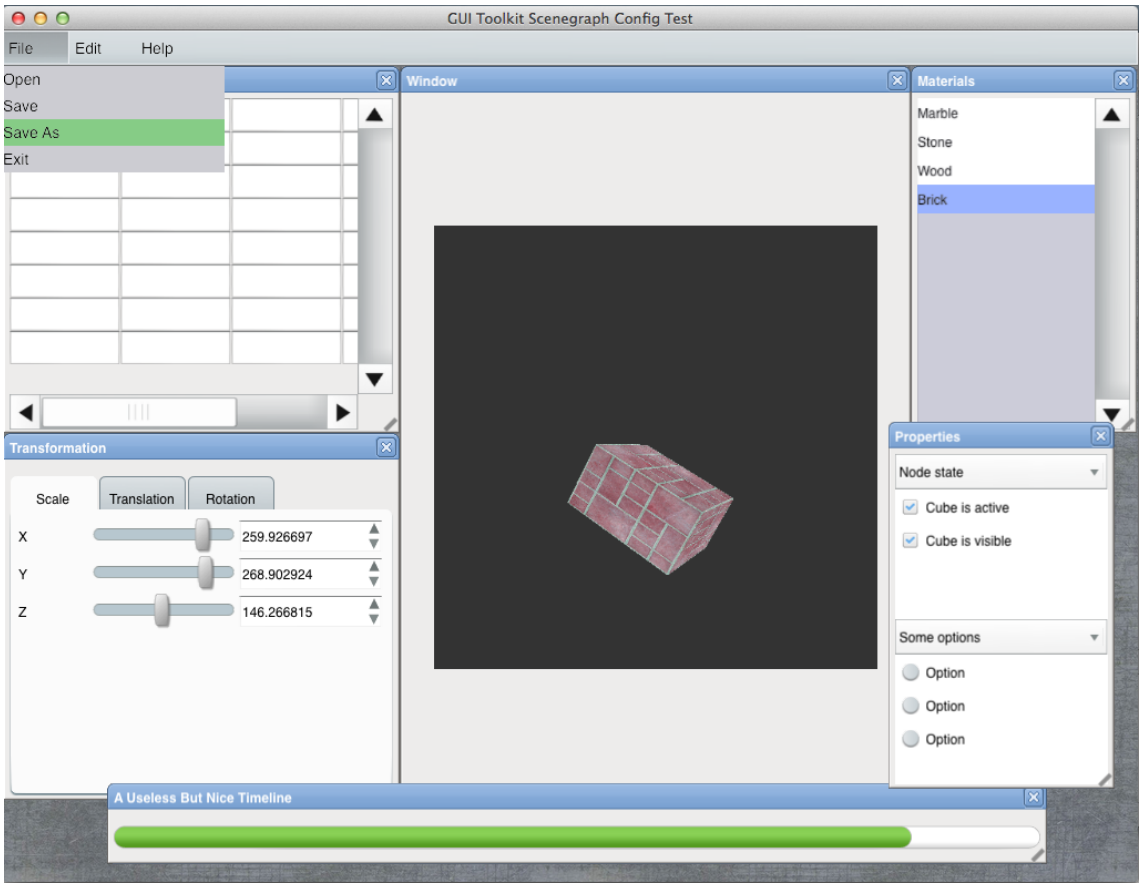

Figure 5.1: Demo: Widget Showcase

The application captured for figure 5.1 is a demonstration of the available widgets and their interaction. This is a typical example of desktop applications featuring so called modal windows. Below the title bar is a main menu, an instance of `Gui::MenuBar`, which contains menu items and menu strips. The six modal windows are initially arranged by a `Gui::PageLayout` that is assigned to the `Gui::AppWindow` proxy. The upper left window contains a table view with scroll bars. Below is a window with a tab control and three tab pages, each contains sliders and steppers to

manipulate the transformation of a 3D object. The labels and controls are arranged by a horizontal flow layout. Also note that each slider shares its entity with the stepper right of it, so each pair is simultaneously affected by user input. The resulting 3D object is shown in the center window, which contains a `Gui::Component` node skinned by a frame buffer texture. The upper right window has a list view for selecting the actual surface to render on the geometry. Selecting an item yields an event that is handled by an event handler. The event handler reads the selection index and forwards this value to a `Graph::Switch` node to activate the corresponding texture state affecting the cube geometry. The lower right window demonstrates collapse containers with check switches and options buttons. The check switches refer to the node flags "active" and "visible". For example, when unchecking the visibility box, the event handler will cause the geometry node to become invisible. The bottom section contains a window with a progress bar in it. The progress simply increments over time, the value is updated by the `OnProcessTick()` method of the engine processor.

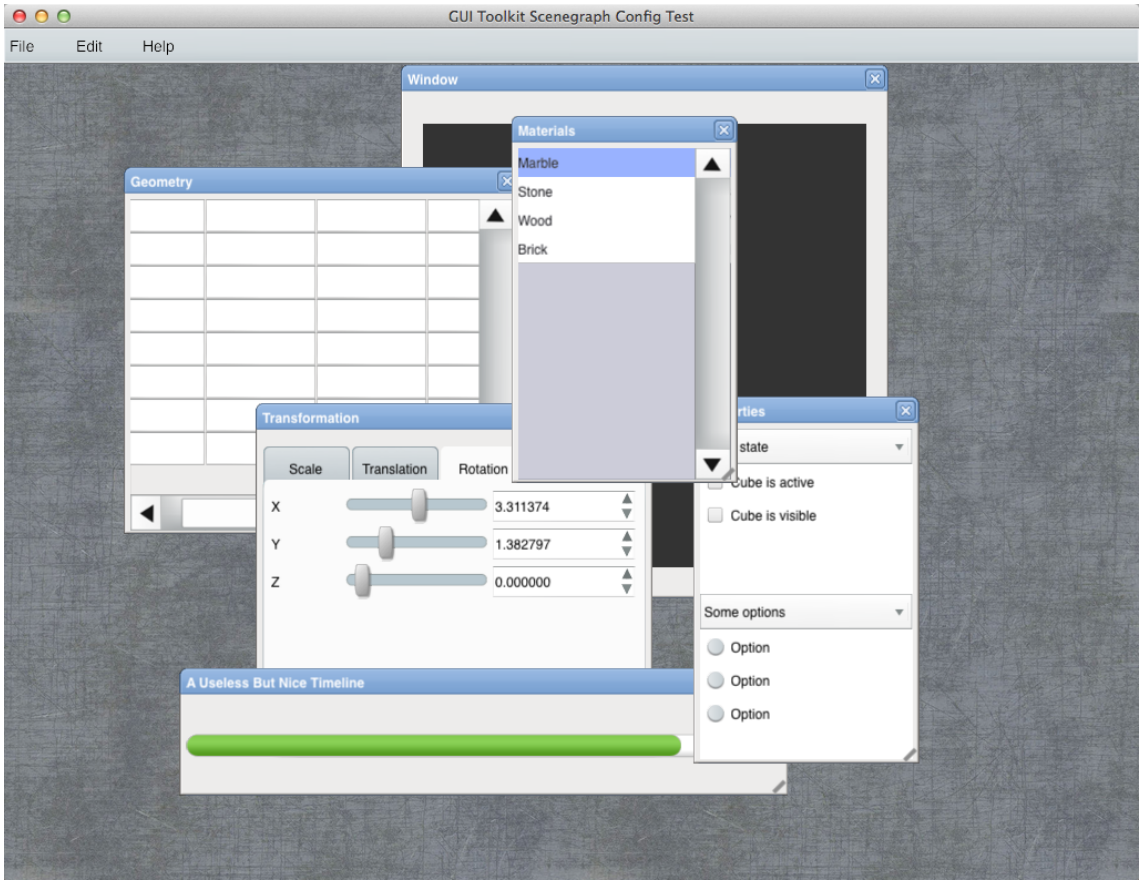

Figure 5.2: Demo: Widget Showcase (Window Layering)

This example on context layering reveals the inverse order of when a particular window (or most probably one of its widgets) has received input from the user: "Materials", "Properties", "A Useless But Nice Timeline", "Transformation", "Geometry", and "Window".

Figure 5.2 shows the similar application, but now demonstrates the overlay effect of window instances, handled by contexts. After sending an input event to a widget

via the pointing device, the widget finds its corresponding context (which is held by the respective window, in this case) and pushes it to the "front".

5.2 Layout Showcase

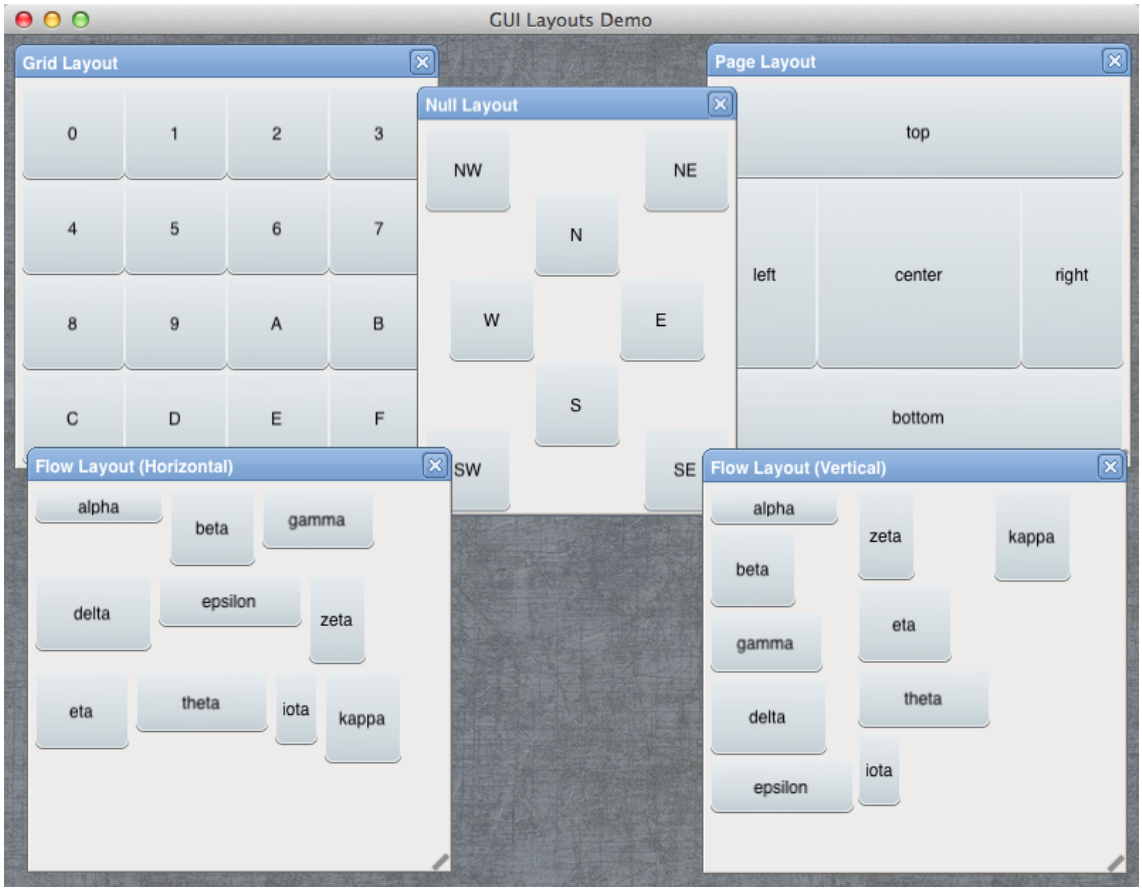

Figure 5.3: Demo: Layout Showcase

The "GUI Layouts Demo" application (see figure 5.3) uses windows, layouts, layout directives, and button nodes to demonstrate the effects of automatic layout management. The top left window uses a grid layout configured to produce four grid cells equal in width and height on both axes. The upper right window shows a page layout with a section space ratio of 1 : 2 : 1 for both width and height. Horizontal and vertical flow layouts are demonstrated in the lower windows, both with the same set of widgets. Finally, a null layout sample is available in the top center window. The corner elements use a relative position, causing them to "stick" beneath the corner even after resizing the window. The buttons in the center are positioned with absolute coordinates, so they will always be aligned by the same distance around the origin. Resizing a window will instantly lead to relayout the entire content according to the new boundaries in all five examples.

Figure 5.5 shows the results of nested layouts: The window itself uses a page layout that aligns five container widgets into sections. The containers in the top

and left sections use horizontal and vertical flow layouts. The elements in the center container are arranged by a grid layout. Null layouts are used in the right and bottom section, with the first using absolute and the latter relative positions. When resizing a container on a higher level (like the window in this example), the nested layouts will be updated iteratively to the size of the nested container they are assigned to. The result is shown in figure 5.6.

5.3 Drag-and-Drop Demo

Figure 5.4: Demo: Drag-and-Drop

Figure 5.4 shows an iOS application that demonstrates Drag-and-Drop, running on iPhone 4. This demo is an example for a traditional inventory menu of role-playing games (RPG). The bottle in the middle (a `Gui::Component` node with texture) can be dragged and dropped into one of the four containers on each side. The actual `Gui::Container` instances are invisible, but they have the same boundaries as the four darkened squares at the background image. The screenshots depict three phases of Drag-and-Drop. The component is placed at its origin position in the initial situation (left). On dragging, the user can move the component freely around. A drag can be completed if the drag input position (e.g., mouse or touch position) intersects both the component and the container (center). After releasing, the container obtains the component and passes it to the layout, which, in this case, places the object at the containers origin (right). As mentioned in the previous section, Drag-and-Drop support is incomplete due to engine restrictions. After a component has been dropped into a container, the position of the component is updated to *appear* at the origin of the container (or whatever the layout has calculated). However, this scene graph structure remains unchanged, with the component still being attached to the same parent as before. This might work in the example shown here, but flaws will occur when performing transformations or other manipulations on either the actual parent container or the targeted container.

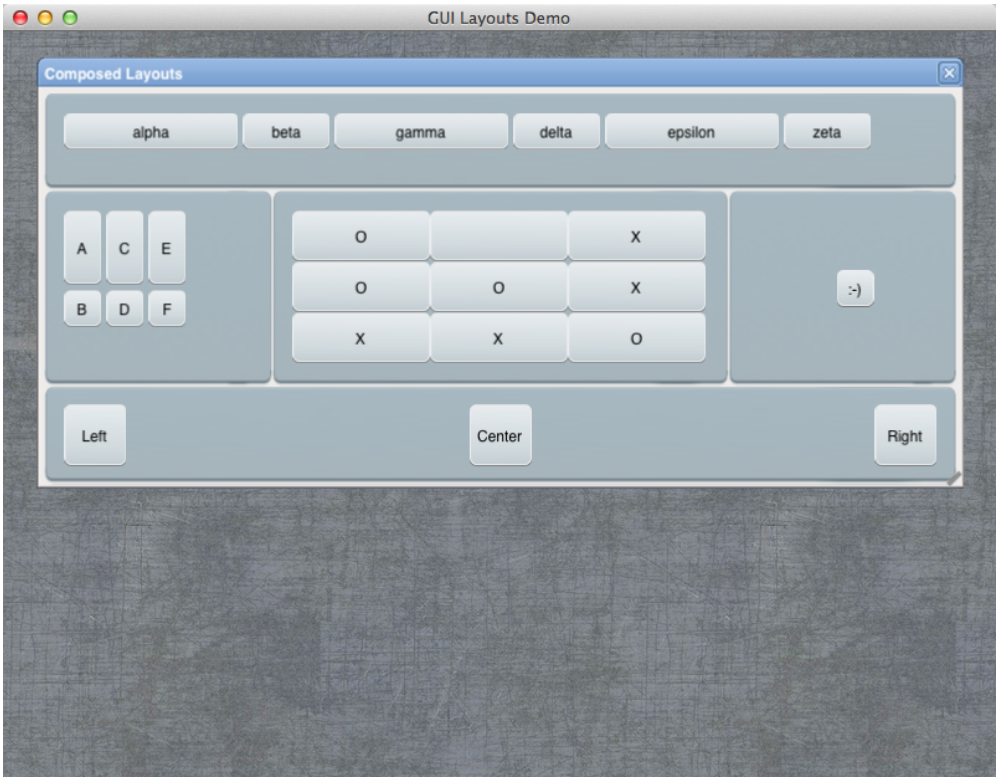

Figure 5.5: Demo: Layout Showcase (Layout Composition 1)

Figure 5.6: Demo: Layout Showcase (Layout Composition 2)

5.4 Shader Effects Demo

The final demo application shows one of many possibilities on how to apply standard 3D graphic manipulation on a GUI made by the MGT (see figure 5.7). Here, shaders are used to define materials for rendering the widget geometry. The example consists of two containers (left and right), with the first one holding four option buttons, and the second contains a regular button that uses a referenced geometry. The material used by the containers is generated from a default vertex shader and a fragment shader that produces a gradient of two colors among the geometry. When selecting one of the four options, the selection event handler sets the index of a `Graph::Switch` node placed in the state set node of the button to yield one of four available different materials – one default material and three shader programs (sparkles, color inversion, and grayscale).

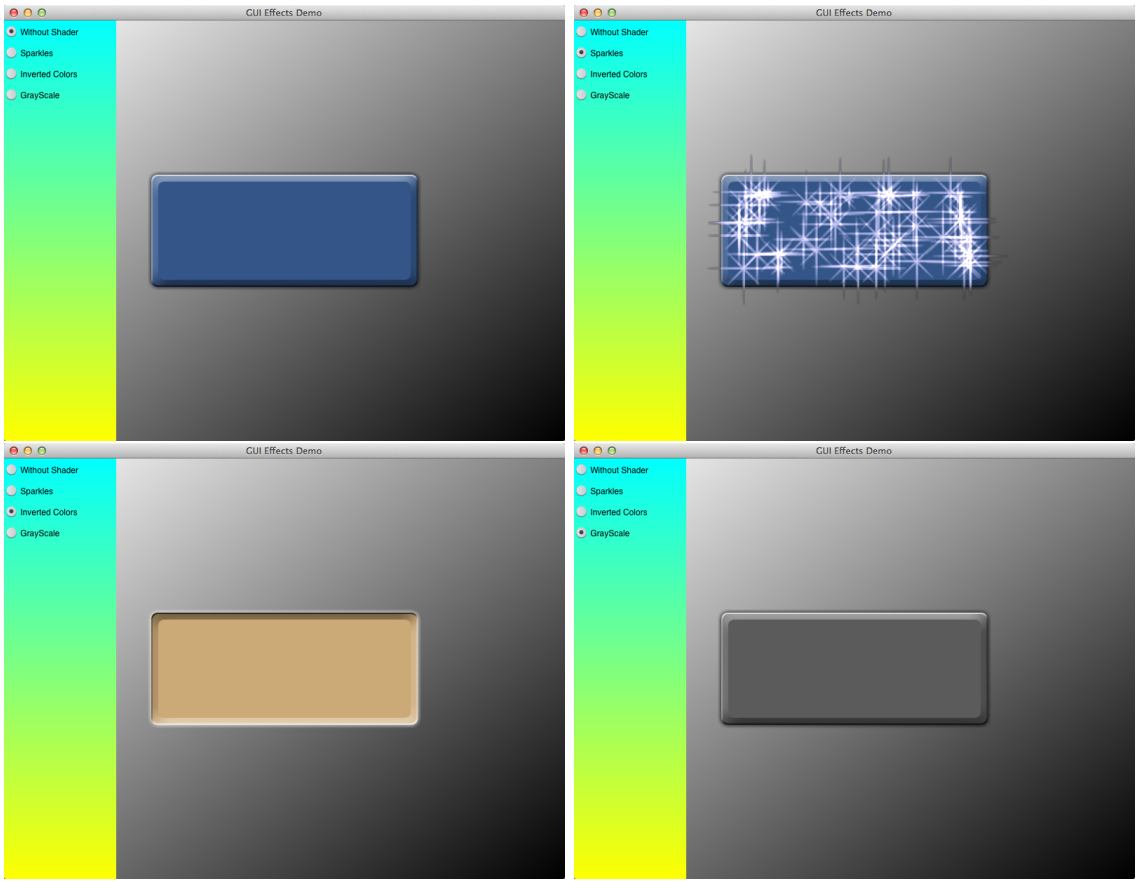

Figure 5.7: Demo: Shader Effects

6 Conclusion

This work has shown the development of the Murl GUI Toolkit, the GUI toolkit extension of the cross-platform application framework Murl Engine. The ME includes a scene graph based 3D engine on the top of OpenGL and focuses primary on the development of visualization, multimedia, and game applications for mobile devices. To provide a GUI toolkit that lives up to contemporary expectations in user interface design, a selection of existing GUI toolkits has been evaluated to identify common features and implementation strategies. This lead to the development of the following five concepts:

Although 3D applications are generally tick-based, the MGT includes (1) a lightweight event handling system based on the observer design pattern. This allows developer to do event-based programming, which is much more common in GUI-driven applications than polling. The most obvious aspect is (2) the set of widgets available. Not all widgets discovered during evaluation have also been implemented by the MGT. Especially complex high-level widgets like calendars or file dialogs have been dropped in favor of basic widgets and because of the fact that those widgets are unlikely to appear in heavily 3D-oriented mobile applications. However, basic widgets for both desktop and mobile platforms have been successfully integrated in the toolkit by using simple 3D primitives, textures and materials, and incorporate them into the scene graph. It is now possible to combine the flexibility of a high-performance scene graph engine with the needs of a 2D GUI toolkit. The declaration of even complex GUIs in an XML resource is fairly simple (comparable to writing HTML markup) and, since it is part of the scene graph, decorative 3D effects are also possible to attach. There are some limitations though: Manipulating and rendering of text is an important task in GUIs, but not so in 3D rendering. So, more work needs to be done in order to abstract native text rendering APIs on different platforms (e.g., Core Text). While the engine is still in beta stadium during the time of writing, particular challenges (like marking of text or getting a text's width and height) could not be solved. Another requirement has been (3) automatic layout management. Just like (2), layouts are completely integrated in the scene graph as container nodes that handle the transformation of their children in order to fit into a desired layout scheme. (4) Skinning also strongly uses the engine's graph capabilities by defining so called State Set nodes (nodes that activate materials, textures, lighting parameters, etc.) that are referred to by widgets in order to render their geometry surface. Finally, (5) the Entity concept has been implemented to unify the representation and handling of data used by widgets.

For productional use, some additions and improvements have to be made. First of all, the toolkits needs to be updated to a newer version of the ME, which con-

tains features that have not been available during the time of implementation (e.g., auto-alignment and text size queries). The next step would be the extension of existing widgets with more options for a greater flexibility and a better support of mobile UI guidelines. Also some new kind of widgets shall be considered, especially those known from iOS and Android, like the calendar spinner or the navigation bar. As a further suggestion, by optionally including the SQLite library "libsqlite3" into the ME and by providing wrapping interfaces on engine layer, the Entity system of the MGT can be extended to a database abstraction layer, serving as model logic between a database and particular widgets (e.g. List Views or Table View). The MGT is already a good foundation for GUIs in 3D applications. But further improvement of the current toolkit by including more features developers ask for when writing pure GUI-based mobile apps – like REST/JSON tools and the already mentioned database abstraction layer –, mobile developers outside the games industry will probably show their interest in the ME as an opportunity to develop cross-platform, native applications.

Bibliography

Android Developers (2014a). *Input Events (Android API Guides)*. URL: http://developer.android.com/guide/topics/ui/ui-events.html (visited on 08/01/2014).

– (2014b). *Layouts (Android API Guides)*. URL: http://developer.android.com/guide/topics/ui/declaring-layout.html (visited on 08/01/2014).

– (2014c). *Styles and Themes (Android API Guides)*. URL: http://developer.android.com/guide/topics/ui/themes.html (visited on 07/31/2014).

Apache Software Foundation (2014a). *Apache Pivot*. URL: http://pivot.apache.org (visited on 07/31/2014).

– (2014b). *Component & Container*. URL: http://pivot.apache.org/tutorials/component-and-container.html (visited on 07/31/2014).

– (2014c). *Event Handling*. URL: http://pivot.apache.org/tutorials/stock-tracker.events.html (visited on 08/01/2014).

– (2014d). *Frequently Asked Questions (FAQ)*. URL: http://pivot.apache.org/faq.html (visited on 07/31/2014).

– (2014e). *Layout Containers*. URL: http://pivot.apache.org/tutorials/layout-containers.html (visited on 08/01/2014).

Apple Inc. (2013a). *About Events in iOS*. URL: https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Introduction/Introduction.html (visited on 08/01/2014).

– (2013b). *About the iOS Technologies*. URL: https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html (visited on 07/31/2014).

– (2013c). *UIAppearance Protocol Reference*. URL: https://developer.apple.com/library/ios/documentation/uikit/reference/UIAppearance_Protocol/Reference/Reference.html (visited on 07/31/2014).

– (2014). *UICollectionView Class Reference*. URL: https://developer.apple.com/library/ios/documentation/UIKit/Reference/UICollectionView_class/Reference/Reference.html (visited on 08/01/2014).

Bishop, Judith (2004). "Developing Principles of GUI Programming Using Views". In: *35 th SIGCSE Technical Symposium on Computer Science Education*. ACM-SIGCSE, Wiley, pp. 373–377.

Carlisle, Martin C. (1999). "A Truly Implementation Independent GUI Development Tool". In: *SIGAda Letters, Vol XIX* 3, pp. 47–52.

Carnegie Mellon University (2010). *Panda3D Manual: DirectButton*. URL: https://www.panda3d.org/manual/index.php/DirectButton (visited on 08/01/2014).

Clasen, Matthias (2004). *GTK+ History*. URL: http://people.redhat.com/mclasen/Usenix04/notes/x29.html (visited on 07/31/2014).

DeLoura, Mark (2001). *Game Programming Gems 2*. Rockland, MA, USA: Charles River Media, Inc. ISBN: 1584500549.

Eclipse contributors and others (2011). *Interface Listener*. URL: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fwidgets%2FListener.html (visited on 08/01/2014).

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201633612.

Garnacho, Carlos (2011). *Styling GTK+ with CSS*. URL: http://thegnomejournal.wordpress.com/2011/03/15/styling-gtk-with-css/ (visited on 07/31/2014).

Gehani, Narain (1991). *Ada: Concurrent Programming*. Summit, NJ: Silicon Press. ISBN: 9780929306087.

FDLv12 (2002). *GNU Free Documentation License*. Version 1.2. Free Software Foundation. URL: http://www.gnu.org/licenses/fdl-1.2.

GPLv2 (1991). *GNU General Public License*. Version 2. Free Software Foundation. URL: http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

GPLv3 (2007). *GNU General Public License*. Version 3. Free Software Foundation. URL: http://www.gnu.org/licenses/gpl.html.

Heisler, Yoni (2013). *IDC report: iPhone lost marketshare to Android in Q2 2013*. URL: http://www.tuaw.com/2013/08/07/idc-report-iphone-lost-marketshare-to-android-in-q2-2013/ (visited on 07/31/2014).

Johnson, Daniel and Janet Wiles (2003). "Effective Affective User Interface Design in Games". In: *Ergonomics* 46, pp. 1332–1345.

Landay, James A., James A. L, and Todd R. Kaufmann (1993). *User Interface Issues in Mobile Computing*.

Leisegang, Christoph (2011). "Schlüsselfigur. Model View Presenter: Entwurfsmuster für Rich Clients". In: *iX* 1, pp. 128–133.

MacLeod, Carolyn (2009). *Understanding Layouts in SWT*. URL: http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html (visited on 08/01/2014).

Miaoulis, Georgios and Dimitri Plemenos (2009). *Intelligent Scene Modelling Information Systems*. 1st. Springer Publishing Company. ISBN: 9783540929017.

Microsoft Corporation (2014a). *Einführung in WPF*. URL: http://msdn.microsoft.com/de-de/library/aa970268.aspx (visited on 07/31/2014).

– (2014b). *Events (WPF)*. URL: http://msdn.microsoft.com/en-us/library/ms753115(v=vs.110).aspx (visited on 07/31/2014).

– (2014c). *Styling and Templating*. URL: http://msdn.microsoft.com/en-us/library/ms745683(v=vs.110).aspx (visited on 07/31/2014).

Mono (2014). *WPF*. URL: http://www.mono-project.com/WPF (visited on 07/31/2014).

Niemeyer, Patrick and Daniel Leuck (2013). *Learning Java*. O'Reilly Media. ISBN: 9781449372507.

Northover, Steve (2001). *SWT: The Standard Widget Toolkit. PART 1: Implementation Strategy for Java Natives*. URL: http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html (visited on 07/31/2014).

Oracle Corporation (2014a). *How to Set the Look and Feel*. URL: http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html (visited on 07/31/2014).

– (2014b). *Lesson: Laying Out Components Within a Container*. URL: http://docs.oracle.com/javase/tutorial/uiswing/layout/index.html (visited on 08/01/2014).

– (2014c). *Lesson: Writing Event Listeners*. URL: http://docs.oracle.com/javase/tutorial/uiswing/events (visited on 07/31/2014).

– (2014d). *Trail: Creating a GUI With JFC/Swing*. URL: http://chimera.labs.oreilly.com/books/1234000001805/ch16.html (visited on 07/31/2014).

Pausch, Randy et al. (1992). *Lessons Learned from SUIT, the Simple User Interface Toolkit*.

Pountain, Dick (1989). *The X Window System*. URL: http://www.guidebookgallery.org/articles/thexwindowsystem (visited on 07/31/2014).

Qt Project Hosting (2013a). *Layout Management (Qt Project Documentation)*. URL: http://qt-project.org/doc/qt-4.8/layout.html (visited on 08/01/2014).

– (2013b). *Qt Style Sheets (Qt Project Documentation)*. URL: http://qt-project.org/doc/qt-4.8/stylesheet.html (visited on 07/31/2014).

– (2013c). *Signals & Slots (Qt Project Documentation)*. URL: http://qt-project.org/doc/qt-4.8/stylesheet.html (visited on 08/01/2014).

– (2014). *Qt Project Documentation*. URL: http://qt-project.org/doc/ (visited on 07/31/2014).

Rademacher, Paul, Nigel Stewart, and Bill Baxter (2006). *GLUI User Interface Library*. URL: http://http://glui.sourceforge.net (visited on 07/31/2014).

Raw Material Software Ltd. (2014a). *About JUCE*. URL: http://www.juce.com/about-juce (visited on 07/31/2014).

– (2014b). *LookAndFeel Class Reference*. URL: http://www.juce.com/api/classLookAndFeel.html (visited on 07/31/2014).

– (2014c). *MessageManager Class Reference*. URL: http://www.juce.com/api/classMessageManager.html (visited on 07/31/2014).

Schmidt, Vincent A. (2010). *User interface Design Patterns*. Tech. rep. Warfighter Interface Division.

Shreiner, Dave and The Khronos OpenGL ARB Working Group (2009). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. 7th. Addison-Wesley Professional. ISBN: 0321552628, 9780321552624.

Smart, Julian et al. (2011). *wxX11 port*. URL: http://docs.wxwidgets.org/2.8/wx_wxx11port.html (visited on 07/31/2014).

Spitzak, Bill (2012a). *Common Widgets and Attributes*. URL: http://www.fltk.org/doc-1.3/common.html (visited on 08/01/2014).

– (2012b). *Introduction to FLTK*. URL: http://www.fltk.org/documentation.php/doc-1.1/intro.html (visited on 07/31/2014).

– (2012c). *Programming with FLUID*. URL: http://www.fltk.org/doc-1.3/fluid.html (visited on 08/01/2014).

Spraylight GmbH (2014a). *About the Murl Engine*. URL: http://murlengine.com/?murlpage=about&murllang=en (visited on 07/31/2014).

– (2014b). *Murl Engine Feature List*. URL: http://murlengine.com/?murlpage=features&murllang=en (visited on 07/31/2014).

– (2014c). *Murl::Graph::FixedParameters Class Reference*. URL: http://murlengine.com/api/en/class_murl_1_1_graph_1_1_fixed_parameters.php (visited on 08/01/2014).

– (2014d). *Murl::Graph::IButton Interface Reference*. URL: http://murlengine.com/api/en/class_murl_1_1_graph_1_1_i_button.php (visited on 08/01/2014).

– (2014e). *Murl::Graph::TextGeometry Class Reference*. URL: http://murlengine.com/api/en/class_murl_1_1_graph_1_1_text_geometry.php (visited on 08/01/2014).

– (2014f). *Price of the Murl Engine*. URL: http://murlengine.com/?murlpage=free&murllang=en (visited on 07/31/2014).

Stanchfield, Scott (2012). *What is the difference between AWT and SWT?* URL: http://www.jguru.com/faq/view.jsp?EID=507891 (visited on 07/31/2014).

The GNOME Project (2014a). *Events*. URL: https://developer.gnome.org/gtk-tutorial/2.90/x182.html (visited on 08/01/2014).

– (2014b). *Layout Containers*. URL: https://developer.gnome.org/gtk3/stable/LayoutContainers.html (visited on 08/01/2014).

The Khronos Group (2014). *OpenGL Shading Language*. URL: http://www.opengl.org/documentation/glsl (visited on 08/01/2014).

Thompson, Sarah (2013). *sigslot - C++ Signal/Slot Library*. URL: http://sigslot.sourceforge.net/ (visited on 08/01/2014).

Unity Technologies (2014a). *GUI Basics (Unity Documentation)*. URL: http://docs.unity3d.com/Documentation/Components/gui-Basics.html (visited on 07/31/2014).

– (2014b). *GUI Skin (Unity Documentation)*. URL: http://docs.unity3d.com/Manual/class-GUISkin.html (visited on 07/31/2014).

– (2014c). *Layout Modes (Unity Documentation)*. URL: http://docs.unity3d.com/Manual/gui-Layout.html (visited on 08/01/2014).

University of Alaska Fairbanks (2006). $GLUI_ControlClassReference$. URL: https://www.cs.uaf.edu/2006/fall/cs381/ref/glui/classGLUI__Control.html (visited on 08/01/2014).

Victor, Brian (2014). *What Do These Sizer Things Do?* URL: http://neume.sourceforge.net/sizerdemo/ (visited on 08/01/2014).

Walsh, Rory (2008). "Cabbage, a new GUI framework for Csound". In: *Proceedings of the Linux Audio Developers Conference KHM*.

Wang, L. J. and A. S M Sajeev (2006). "Abstract interface specification languages for device-independent interface design: classification, analysis and challenges". In: *Pervasive Computing and Applications, 2006 1st International Symposium on*, pp. 241–246.

wxWidgets (2014a). *Events and Event Handling*. URL: http://docs.wxwidgets.org/trunk/overview_events.html (visited on 11/30/2013).

– (2014b). *What is wxWidgets?* URL: http://docs.wxwidgets.org/3.0/page_introduction.html (visited on 07/31/2014).

– (2014c). *wxRendererNative Class Reference*. URL: http://docs.wxwidgets.org/trunk/classwx_renderer_native.html (visited on 07/31/2014).

– (2014d). *wxWidgets Datasheet*. URL: http://www.wxwidgets.org/about/datasheets.html (visited on 11/30/2013).

Zijp, Jeroen van der (2013a). *Fox Toolkit: Documentation: Messages*. URL: http://www.fox-toolkit.org/messages.html (visited on 07/31/2014).

– (2013b). *Fox Toolkit: Foreword*. URL: http://www.fox-toolkit.org/foreword.html (visited on 07/31/2014).

Zijp, Jeroen van der (2013c). *Fox Toolkit: Goals and Approach*. URL: `http://www.fox-toolkit.org/goals.html` (visited on 07/31/2014).