CEGUI Unified Editor Developer Manual

Martin Preisler

July 13, 2014

This document has been laid out for a computer screen viewing and thus may be unsuitable for printing. LyX sources are available in *doc/developer-manual-src* in the source tarball if you wish to relayout.

Contents

Chapter 1

Prerequisites

1.1 Knowledge requirements

Because of size constrains, I will not cover Python, PySide, Qt and CEGUI API.

1.2 Getting the source code

```
$ hg clone https://bitbucket.org/cegui/ceed
```

1.2.1 Branches and Tags

- *default* - unstable forward development, likely to be based on unstable CEGUI
- *snapshotX* - development snapshots, based on unstable CEGUI, should be considered tech previews
- **-devel* - feature branches, are expected to be closed and merged into default at some point

1.3 Starting without installation

This section is UNIX only!

It is extremely valuable to start the editor without installing it. You can do so by using the *runwrapper.sh* script in the repository. This script will spawn a new shell that will have environment set so that CEED finds its own modules and PyCEGUI. By default it assumes the following directory structure:

```
$prefix/CEED/bin/runwrapper.sh
$prefix/cegui/build/lib/PyCEGUI.so
```

If your directory structure looks differently you need to alter the script.

Chapter 2

Directory structure

2.1 Top directory

2.1.1 maintenance script

Provides means to compile Qt .ui files, build documentation, fetch newest CEGUI datafiles and make a tarball for CEED releases.

maintenance-temp is a directory with various temporary data that maintenance script needs to run.

2.1.2 perform-pylint

Runs *pylint* over the codebase, results will be stored in pylint-output. It is imperative to run this script, especially before releases, it often uncovers nasty bugs. Even though *pyflakes* has no helper script to run it, you can run it as well, there are no configuration or such files required.

2.1.3 setup.py

Used to install CEED system-wide. Running python setup.py install as root will get the job done. Make sure you already have all the dependencies installed.

Can also be used to create tarballs, the maintenance script may be better for that though, see Section 2.1.1.

2.1.4 cx_Freezer.py

This is a setup.py script that is adapted for freezing the application into a bundle using cx_Freeze. The resulting bundle does not need any dependencies, not even *Python*. Tested on Windows 7 and GNU/Linux distros, both 32bit and 64bit.

Might need copying of some dependencies the script fails to pick up!

Please see the cx_Freeze documentation [4] for more information.

2.1.5 copyright related

Also includes the *AUTHORS* file with CEED contributors and several *COPYING* files of libraries we bundle in Windows and MacOS X builds.

2.2 bin directory

All contents are executable, these are entry points to various functionality of CEED.

2.2.1 ceed-gui

Starts the CEED interface. Provides several CLI options that may be very useful for development, especially auto opening of projects and files after start, see ./ceed-gui –help.

2.2.2 ceed-mic

This is the CLI metaimageset compiler, see the *User manual* for more info.

2.2.3 ceed-migrate

CLI interface to the compatibility machinery in CEED, can be useful for testing newly developed layers, see ./ceed-migrate –help for more info.

2.2.4 runwrapper.sh

Can be used to start CEED without having to install it, see Section 1.3 for more info.

2.3 build directory

Contains results of cx_Freeze build process, see Section 2.1.4 for more info.

2.4 ceed directory

This is where the bulk of the codebase resides. The directory is a *Python package* and none of its files should be executable.

2.4.1 action subpackage

Implements the Action API and defines basic global actions.

2.4.2 cegui subpackage

Wraps Embedded CEGUI (see Section 3.6 for more details). Also provides base classes for CEGUI widget manipulators and all the machinery that they require - GraphicsScene, GraphicsView, ...

2.4.3 compatibility subpackage

Implements the Compatibility API, contains implementations of all the stock *Type Detectors* and *Compatibility Layers*.

2.4.4 editors subpackage

This subpackage encapsulates all editing functionality within CEED. All classes that inherit from TabbedEditor except the convenience wrapper classes should be implemented inside this subpackage.

You can find implementation of imageset editing in the *imageset* subpackage, layout editing in the *layout* subpackage, ...

2.4.5 metaimageset subpackage

Classes required for metaimageset parsing, saving and compiling are implemented in this subpackage. This is what ceed-mic (see Section 2.2.2) uses internally to compile a metaimageset.

2.4.6 propertytree subpackage

UI to inspect and change properties of any class inheriting *CEGUI::PropertySet*.

2.4.7 settings subpackage

Implements the Settings API, defines basic global settings entries.

2.4.8 ui subpackage

Contains .ui files created using *Qt Designer*. The maintenance script is used to compile these into *Python modules*. See Section 3.9 for more info.

2.5 data directory

Contains icons, the splashcreen, stock property mappings, sample CEGUI datafiles and sample project files.

2.6 doc directory

Contains LyX source code for developer manual, quickstart guide and user manual. Also contains the PDF versions after ./maintenance build-docs has been executed (see Section 2.1.1).

Chapter 3

Core API

The whole code is divided into folders where the root folder provides basic reusable functionality (project management, undo view, tab management, ...) and the editors themselves are providing editing facilities for various file types.

3.1 TabbedEditor

A base class for editors hosted in a tab. If you are writing new editing functionality for CEED you definitely need to inherit from this class.

3.1.1 Responsibilities

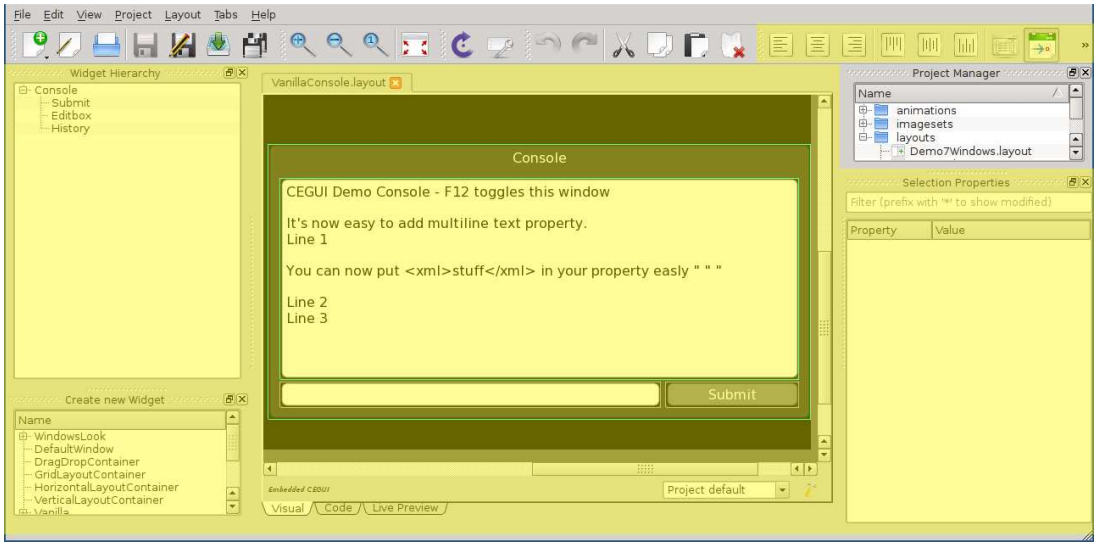

Figure 3.1: tabbed editor responsibilities are highlighted in yellow

The most important part of a TabbedEditor is its widget. The widget represents the central part in Figure 3.1. TabbedEditors also often add toolbars, dock widgets and other elements.

3.1.2 Life cycle

Each tabbed editor goes through the following cycle:

1. Construction of the class
2. Initialisation
   (a) all the supporting widgets get created
   (b) the file is loaded and processed
3. Activation
   (a) this puts the tabbed editor "on stage"

4. User interaction
5. Deactivation
6. Finalisation
   (a) the editor is no longer shown in the interface
7. Destruction
   (a) all held data and widgets are destructed

3.1.3 Derived classes

To avoid repeating code and adhere to the DRY principle [1], there are 2 very important classes that add functionality to TabbedEditor that you want to inherit if applicable to avoid reinventing.

UndoStackTabbedEditor

Very useful in case you are already using the Qt's UndoStack. This connects all the necessary calls and exposes undo and redo of the undo stack to the rest of the application.

MultiModeTabbedEditor

Useful when you want multiple editing modes. As an example, let us take the layout editor. It has three modes - visual, code and live preview. You can freely switch between them and they each offer a different look at the same data. At any point in time you are viewing/editing in one mode only. Please note that you must be using UndoStack in this situation as switching modes is an undo action.

Each mode has its own life cycle and depends on the life cycle of its host tabbed editor. First the tabbed editor gets on "the stage" and then the editor's mode is asked to activate itself.

```
# the host tabbed editor gets constructed and activated
A.deactivate()
B.activate()
# the user merrily edits in the B edit mode
```

Figure 3.2: process of switching from edit mode A to B

The actual mode switch process is a bit more involved because of the necessity to make mode switch an undoable action. You can see the full implementation of it in *ceed.editors.multi.MultiModeTabbedEditor.slot_currentChanged*.

3.2 Undo / Redo

One of the cornerstones of CEED is the ability to undo everything. This is implemented using Qt's QUndoCommand class. Each TabbedEditor has its own independent undo stack, undo commands are never shared across editors.

3.2.1 Principles

- everything that changes data has to be an UndoCommand
- all data that undo command stores in itself must be "independent", storing references to widgets would not work if there is a DestroyCommand that invalidates them
- state switching that would make some undo commands not applicable have to be undo commands themselves

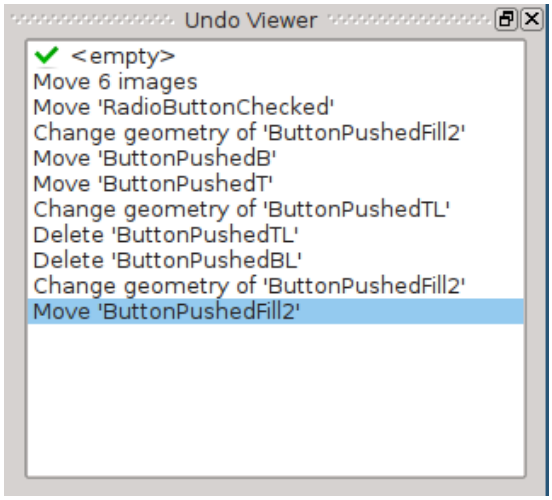

Figure 3.3: example of an undo stack

3.2.2 Moving in the undo stack

Let us consider the undo stack shown in Figure 3.3. If user clicks the <*empty*> line, all the undo commands will get .undo() called in the bottom-up order. If now the user clicks the *Move 'ButtonPushedFill2'* line again, the commands will get .redo() called in the top-down order. It is important to notice that the undo commands are always acted upon sequentially and that order of the calls matter! Some of the commands might not even make any sense if they are called out of order. Consider a *Create Image 'XYZ'* command followed by *Move 'XYZ'*. They need to be acted upon in the right order otherwise the Move command is asked to move a non-existent image.

3.3 Property editing

A lot of CEGUI classes provide basic introspection via property strings. CEED has a set of classes to reuse when you want to edit properties of widgets or any other classes that inherit from PropertySet.

3.3.1 Usage

Even though the *propertytree* subpackage (see Section 2.4.6) gives you access to its very internals and allows very advanced uses, including using it on classes that do not even inherit from the CEGUI::PropertySet, only the basic usage scenarios will be discussed in this document.

```
from ceed import propertysetinspector
from ceed import mainwindow

# parent is a QWidget and can be None
inspector = propertysetinspector.PropertyInspectorWidget(parent)
self.inspector.ptree.setupRegistry(propertytree.editors.PropertyEditorRegistry(True)
pmap = mainwindow.MainWindow.instance.project.propertyMap
self.inspector.setPropertyManager(propertysetinspector.CEGUIPropertyManager(pmap))
```

Figure 3.4: creating a property inspector widget

```
# inspector is a property inspector as created previously

inspector.setPropertySets([propertySetToInspect])
```

Figure 3.5: inspecting a PropertySet using a property inspector

3.4 Settings API

Whenever you want users to be able to change some value to affect behavior of the application, consider using the Settings API. You only need to define the *settings entry* and the UI that allows changing it will be auto-generated for you.

```
category = settings.createCategory(name = "layout", label = "Layout editing")

visual = category.createSection(name = "visual", label = "Visual editing")

visual.createEntry(name = "continuous_rendering",
                   type = bool,
                   label = "Continuous rendering",
                   help = "Check this if you are experiencing redraw issues...",
                   defaultValue = False, widgetHint = "checkbox",
                   sortingWeight = -1
```

Figure 3.6: defining a settings entry

It is recommended to query the settings entry once and keep the reference stored to avoid having to look it up frequently.

```
entry = settings.getEntry("layout/visual/continuous_rendering")
# entry is a reference to SettingsEntry class
# we get the fresh value whenever we use entry.value later in the code
print("Continuous rendering is %s" % ("on" if entry.value else "off"))
```

Figure 3.7: using a settings entry

3.5 Action API

Whenever there is an action needed you are advised to use the action API, see ceed.action module. The actions inherit from QAction and offer the same functionality but shortcuts are handled automatically for the developer, including UI for the user to remap them.

To use the Action API you have to define your actions first, this is usually done in a separate file to keep things clean. See *editors/imageset/action_decl.py* and *editors/layout/action_decl.py*. Then you query for this action in your code and connect your signals to it. You can use the convenience ConnectionMap to ease mass connects and disconnects.

```
cat.createAction(
             name = "align_hleft",
             label = "Align &Left (horizontally)",
             help = "Sets horizontal alignment of all selected widgets to left.",
             icon = QtGui.QIcon("icons/layout_editing/align_hleft.png"))
```

```
cat.createAction(
             name = "snap_grid",
             label = "Snap to &Grid",
             help = "When resizing and moving widgets, if checked this makes sure..."
                ,
             icon = QtGui.QIcon("icons/layout_editing/snap_grid.png"),
             defaultShortcut = QtGui.QKeySequence(QtCore.Qt.Key_Space)).setCheckable(
                True)
```

Figure 3.8: defining new actions

You can check the shortcut remap UI generated for you in *Settings » Shortcuts*.

3.6 Embedded CEGUI

To make sure everything is rendered exactly as it will appear in CEGUI it is used in the editor. This also ensures that whatever custom assets you have, they will be usable in the editor exactly as they are in CEGUI itself.

3.6.1 PyCEGUI bindings

As CEGUI is a C++ library, making it accessible from Python is not trivial. I have written python bindings for CEGUI called PyCEGUI using *py++* and *boost::python* for this purpose. It is important to realise though that even though I tried to make it pythonic and reasonably safe, mistreating PyCEGUI can still cause segfaults and other phenomena usually prevented by using a scripting language.

3.6.2 Shared CEGUI instance

There is only one CEGUI instance in CEED. This makes tabbed editor switches slightly slower but CEED uses less memory. The main reason for this design decision is that CEGUI did not have multiple GUI contexts at the time CEED was being designed.

Furthermore, the shared instance is wrapped in a "container widget" which provides convenience wrappers. That way developer can avoid dealing with *OpenGL* and *QGLWidget* directly.

```
ceguiContainerWidget = mainwindow.MainWindow.instance.ceguiContainerWidget

# parentWidget is the widget that will host the CEGUI rendering, it cannot be None!
ceguiContainerWidget.activate(parentWidget, self.scene)
ceguiContainerWidget.setViewFeatures(wheelZoom = True, continuousRendering = True)

# you can then use CEGUI directly through PyCEGUI, the result will be rendered
# to the host widget specified previously
PyCEGUI.System.getSingleton().getDefaultGUIContext().setRootWindow(self.
    rootPreviewWidget)

# ... rendering, interaction, etc.

# after your work is done, deactivate the container widget
ceguiContainerWidget.deactivate(self.ceguiPreview)
```

Figure 3.9: accessing and using the CEGUI instance

Always clean up!

The *CEGUI* container widget is shared, therefore the whole *CEGUI* instance and the default *GUIContext* are shared. *CEGUI* resources are not garbage collected, they are created in the C++ world and have to have their life cycles managed manually. Make sure you always destroy all your widgets and other resources after use. They will not get cleaned up until the whole editor is closed!

Beware of name clashes!

Becuase the CEGUI instance is shared there can be name clashes for many resources - images, animation definitions, ... A good way to circumvent this is to generate unique games with an integer suffix and hide the fact from the user.

This is what the *Animation list editor* does internally, for more details see *ceed.editors.animation_list*.

3.7 Compatibility layers

Compatibility is only dealt with on data level. The editor itself only supports one version of each format and layers allow to convert this raw data to other formats. Here is an example of how to do that:

```
# we want to migrate and imageset from data format "foo" to "bar"
# data is a string containing imageset in "foo" format

from ceed.compatibility import imageset as compat
convertedData = compat.manager.transform("foo", "bar", data)
```

There are also facilities to guess types of arbitrary data. See API reference of CompatibilityManager for more info.

3.7.1 Testing compatibility layers

Running the GUI and loading files manually by clicking is not practical for compatibility layer development and testing. Use the *ceed-migrate* executable instead. See Section 2.2.3.

3.8 Model View (Controller)

As most editing applications we have the MVC paradigm [3]. When I say something is the *model* I mean that it encapsulates and contains the data we are editing. The *view* on the other hand encapsulates the facility to view the data we are editing in their current state. The *controller* allows the user to interact with the data. Most of the time *view* meshes with *controller* as it does in the Qt world so we are using one class instance for both *view* and *control*.

Separating model from view helps make the code more maintainable and cleaner. It also makes undo command implementation easier.

3.9 Qt designer .ui files

Qt designer allows RAD so it pays off to keep as much GUI layout in .ui files as possible. Whenever you are creating a new interface, consider creating it with the Qt designer instead of coding it manually.

3.9.1 Compiling

The files have to be compiled into *Python modules*.

Development mode

The preferred method if you want to continuously develop CEED. Allows automatic recompilation of all ui files.

```
$ vim ceed/version.py
# make sure the DEVELOPER_MODE line is set to True
```

Figure 3.10: turning the developer mode on

maintenance script

If you only want to compile the ui files rarely you are better off with the maintenance script. See Section 2.1.1.

```
./maintenance compile-ui-files
```

Figure 3.11: recompiling ui files via the maintenance script

Chapter 4

Editing implementation

4.1 Imageset editing

Lives in the *ceed.editors.imageset* package. Provides editing functionality for CEGUI imagesets. Please see the CEGUI imageset format documentation [2] for more details about the format.

4.1.1 Data model

Classes from the *ceed.editors.imageset.elements* package are used to model the data instead of using CEGUI in this editor. The reason is relative simplicity of the data and big changes to the image API between CEGUI 0.7 and 0.8. Compatibility layers are used to convert given data to the native format before they are loaded into the data model. See Section 3.7 for more details.

4.1.2 Undo data

Undo data are implemented using string for image definition reference and Python's builtin types to remember geometry.

4.1.3 Multiple modes

It is a multi-mode editor with visual and code modes. The code mode always uses and displays native CEGUI data.

4.1.4 Copy / Paste

Copy paste is implemented using custom MIME type and bytestreams. It is even possible to copy image definitions across editor instances.

4.2 Layout editing

Located in the *ceed.editors.layout* package. CEGUI Window is used to model the entire layout hierarchy. We use WidgetManipulator class to add serialisation (for undo/redo), resizing handles and more to windows. It is a multimode editor with visual, code and live preview modes. The live preview mode does no editing, instead it just views the current layout and allows user to interact with it to test it.

4.2.1 Data model

Layout editing operates of widget hiearchies, a data model natively implemented in CEGUI that we use directly. Since CEGUI does not have global window names since version 0.8 we do not even have to worry about name clashes.

4.2.2 Undo data

Undo data are implemented using strings for widget path reference and widget properties are serialised using Python's builtin types.

LookNFeel property caveat

When you change the LookNFeel property the auto child widgets get destroyed and constructed anew. This breaks undo history and is not allowed at the moment. I don't it is worth the effort to support this. Either way we would have to "alter history" in some cases. Changing it in code mode will of course work because the entire hierarchy will be reconstructed from scratch.

WindowRenderer property caveat

Similar to the LookNFeel case it makes changes to the window that break undo history. Right now it is disallowed to change it from the editor. Changing it in code mode will of course work because the entire hierarchy will be reconstructed from scratch.

4.2.3 Multiple modes

Visual, Code and *Live preview* modes are provided. Code is a simple XML editing mode but the other two are implemented using embedded CEGUI.

4.2.4 Copy / Paste

Copy paste is implemented using custom MIME type and bytestreams. It is even possible to copy widget hierarchies across editor instances.

4.3 Animation editing

Located in *ceed.editors.animation_list* package. We use wrappers to deal with the fact that CEGUI has no model for a list of animations.

KeyFrames had to have indices added because comparing floats for equality is unreliable. So in the end we sort all keyframes by position and figure out their indices from that. To avoid placing two keyframes at the exact same position we add a small epsilon until we have no clashes whenever we encounter this possibility.

Chapter 5

Contributing

5.1 Coding style

CEED does not follow the *PEP8* style recommendation when it comes to method and variable naming. The reason I chose to use camelCase for methods and variables is that PySide and CEGUI both use that and CEED calls a lot of methods form these 2 APIs. The code looked much better with camelCase naming.

Use the following rules for all contributed code to CEED:

- use 4 spaces for indentation
- use CamelCase for class naming
- do not use wildcard imports [1]
- use camelCase for method and variable naming
- document methods and classes with the triple quote docstyle syntax
- comment all other things with # prefix only

5.2 Communication channels

You can reach the CEGUI team using:

- IRC: #cegui on irc.freenode.net[2]
- email: team@cegui.org.uk

5.3 DVCS - forking

Create a fork of `https://bitbucket.org/cegui/ceed` on `http://bitbucket.org` or elsewhere. Start each feature or substantial fix in a separate branch, this makes it easy to review and possibly reject some parts without rejecting everything. When you are finished with your branch make sure you merge all upstream changes if any. Having to deal with merge conflicts makes the reviewers more likely to postpone integration. After all of this is done, simply contact upstream developer to merge your changes into the main repository. You can usually reach someone through IRC (freenode/#cegui), mantis bug tracker or email (*team@cegui.org.uk*).

5.4 The old fashioned way - patches

You can alternatively just send unified diff patches by email if you so desire. Use the *team@cegui.org.uk* email address. Make sure you state what the patchset is based on.

[1] `from package import *` cannot appear anywhere in the code.
[2] See `http://freenode.net` for more information about the network.

Bibliography

[1] Andrei Alexandrescu and Herb Sutter. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. 2004.

[2] CEGUI development team. Imageset xml format specification. `http://www.cegui.org.uk/docs/current/xml_imageset.html`.

[3] Alan Ezust. *An Introduction to Design Patterns in C++ with Qt 4*. 2006.

[4] Anthony Tuininga. cxFreeze documentation. `http://cx_freeze.readthedocs.org/en/latest/index.html`.