

# Crazy Eddie's GUI System

## An Overview of Renderer Model 2

by

Paul D. Turner

*This is preliminary documentation for a system that is currently under continued development and refinement, as such all classes, interfaces, requirements and recommendations described within this document are subject to change without further notification.*

This document is an overview of the new rendering subsystem that will be used by Crazy Eddie's GUI System (CEGUI) for its 0.7.0 release. Along with the OpenGL based implementation of the various classes, this document forms a reference for people writing new CEGUI renderer modules and also for maintainers of existing CEGUI renderer modules wishing to update ready for 0.7.0.



# Table of Contents

Introduction and motive.....	5
Out with the old.....	5
Unnecessary conversions.....	5
The cache that didn't.....	5
Move a window, redraw everything.....	5
No imagery caching.....	6
Small batches.....	6
In with the new.....	6
High level view of the new system.....	6
Implementation of a Renderer module.....	7
Prerequisites.....	7
Required object implementations.....	7
Renderer.....	8
Texture.....	8
GeometryBuffer.....	8
RenderTarget.....	9
TextureTarget.....	9
Drawing 1: Basic Rendering Subsystem Classes.....	10



## Introduction and motive

Development on the Mk-2 version of Crazy Eddie's GUI System started in February of 2004 with the intention of providing a powerful and flexible GUI system via an abstracted rendering interface. How successful earlier releases were at fulfilling the first two criteria could be debated for an eternity, one thing that is certain, however, is that things have become more powerful and more flexible as development has progressed. That said, one point of contention has always existed with regards to performance.

During various discussions back in early 2006 it was decided that a new renderer model was desirable; something that would address the performance issues some users were experiencing and also provide other enhanced rendering facilities.

For reasons not relevant to this discussion, progress on this new renderer model was much delayed; with work eventually beginning in February 2008 on what was dubbed the “Render Target” systems. While this new development progressed, with new implementations for most of the renderer modules provided with CEGUI, it soon became apparent that even though substantial performance gains were achieved, this work suffered from some familiar issues and would almost certainly not satisfy what people were expecting of a modern GUI system (at least from a game development perspective).

## Out with the old

Before discussing the new rendering subsystem, it may be helpful to briefly discuss some of the issues identified with the old approach to rendering; this helps to rationalise many of the design decisions, as many of the issues below map directly to some class or implementation recommendation for the new renderers. Overall, the previous approach to rendering in CEGUI was less than optimal, although worse than this was the fact the *implementations*<sup>1</sup> were also so poorly done.

## Unnecessary conversions

In the renderer modules, we had a situation where queued quads were stored in a non API native format and converted to something API specific *every time the screen was redrawn*. When performing an initial analysis, this was perhaps the thing that struck me as being most wasteful<sup>2</sup>.

## The cache that didn't

The `RenderCache` was introduced with the Falagard skinning system, although this was just caching high-level drawing operations as opposed to any actual quads, geometry or imagery. There is no question that the use of the `RenderCache` saved much re-calculation with regards to skin definitions, there was minimal gain overall since we were still re-creating quad lists in full every time any part of any window changed – meaning that something simple like highlighting a button would cause a total redraw, including unrelated windows and content.

## Move a window, redraw everything

This is somewhat related to the above issue, although at a different level of interaction. When simply moving a window (such as dragging a `FrameWindow` of some kind) we would again regenerate and redraw *everything*. Not only was this full redraw unnecessary, it would also have

<sup>1</sup> At least those provided by us, though most other people followed our model pretty closely!

<sup>2</sup> As the first stage of implementing the new OpenGL renderer, this system was changed so that the added quads were queued as OpenGL vertex data that could be used directly, the performance gain was quite substantial. Theoretically, existing renderers for 0.6.2 and earlier versions of CEGUI could get a large performance boost by implementing this change.

been unnecessary to even regenerate the content of the window being moved – nothing had changed except it's position!

## No imagery caching

While not the result of any deep analysis, the lack of facility for caching of already rendered imagery was a glaring omission. This is basically the only issue that the original render targets solution addressed.

## Small batches

Due to the fact there is imagery from multiple different texture sources being layered to form the CEGUI output, being able to form large batches for the rendering pipeline was, and still is, something difficult to achieve – at least without making possibly unreasonable stipulations with regards to implementation. With the new renderer model, we have again resisted this temptation, so you should be aware that the “small batches” issue, such that it is, is still present, although it's effects should be mitigated to a large degree by improvements elsewhere.

## In with the new

For your main overview, please refer to the diagram “Basic Rendering Subsystem Classes”. This diagram is *reasonably* straightforward to follow, and although there are a couple of things which may appear slightly unusual<sup>3</sup>.

The thing that will immediately stand out is that there is no `CEGUI::Renderer` class on the diagram, this is not a mistake, rather it serves to highlight the minimalistic role that `Renderer` now has in comparison to other parts of the rendering subsystem; `Renderer` now *mainly* serves as a facility for creating other objects.

On the diagram, the only familiar class is `Window`. This illustrates that from the library users perspective everything is done via the main window class and that this is sitting on top of the rest of the rendering subsystem. This is obviously not 100% accurate; `Window` objects do not usually perform any direct drawing operations, rather it is `ImageSet` and `Font` objects that do the drawing. Since the outward operation of `ImageSet` and `Font` is generally unchanged, these classes have also been omitted from the diagram, and the discussion, so as not to confuse things.

## High level view of the new system

- A `Window` object has a `GeometryBuffer`. The `GeometryBuffer` is used to cache the actual geometry used for that window<sup>4</sup>.
- `Window` rendering, via geometry stored in the `GeometryBuffer`, is queued to a `RenderingSurface`<sup>5</sup>.
- A `RenderingSurface` has a collection of `RenderingQueue` objects. `RenderingQueue` objects provide a means to reliably layer rendering performed on a

---

<sup>3</sup> Basically the fact that `RenderingSurface` maintains a collection of `RenderingWindow` - one of it's own subclasses. And that `RenderingSurface` makes use of a `RenderTarget` while `RenderingWindow` (a subclass of `RenderingSurface`) makes use of a `TextureTarget` (a subclass of `RenderTarget`), the key here is to know that in this case the `RenderTarget` / `TextureTarget` are the *same object*.

<sup>4</sup> And only that window; it does not include child window geometry, since child window have their own `GeometryBuffer` objects.

<sup>5</sup> When a window's geometry is queued for drawing, if the window does not have a `RenderingSurface` assigned, the `RenderingSurface` from the closest window back up the hierarchy is used. If no such window has a `RenderingSurface`, the default `RenderingRoot` returned from the `Renderer` object is used.

RenderingSurface.

- The most basic RenderingSurface is one that represents the display window. This will usually be the default RenderingRoot object returned by the Renderer object.
- Any Window can have a RenderingSurface directly assigned to it.
- A RenderingSurface assigned to a Window will receive the rendered output for the Window it is assigned to and any attached child Window objects<sup>6</sup>.
- Unless a Window is at the root of a hierarchy, the RenderingSurface assigned will almost always be a RenderingWindow<sup>7</sup>.
- A RenderingWindow is backed by a TextureTarget and is therefore able to cache the actual imagery for the content rendered to it.
- A RenderingWindow has its own GeometryBuffer and when drawn back onto some other surface, is rendered using the same mechanisms as described above.
- A RenderingWindow can have a RenderEffect assigned<sup>8</sup> to perform various custom rendering and geometry manipulations when compositing the RenderingWindow back onto the RenderingSurface that owns it.

## Implementation of a Renderer module

### Prerequisites

This section lists the minimum recommended facilities that the underlying API (or engine) should support in order to implement a decent CEGUI rendering module. Having said this, it's actually possible to work around the lack of some of these facilities by other means.

- Scissor testing (used for window clipping).
- Render to texture (used for full imagery caching and also effect applications).
- Vertex based geometry (CEGUI now provides vertex data to the rendering systems).

### Required object implementations

Much of the new rendering subsystem is actually already provided via concrete classes in the system; there are only a handful of abstract classes that must be implemented for a new renderer module:

- `Renderer`
- `Texture`
- `GeometryBuffer`
- `RenderTarget`
- `TextureTarget`<sup>9</sup>

The rest of this document will briefly discuss these objects along with any implementation

---

<sup>6</sup> A child window might first render to its own RenderingWindow, but eventually that will get rendered back to the parent's RenderingSurface..

<sup>7</sup> RenderingWindow is a subclass of RenderingSurface.

<sup>8</sup> Technically the RenderEffect is assigned to the GeometryBuffer that holds geometry for the RenderingWindow.

<sup>9</sup> This can theoretically be omitted, though many new facilities rely upon TextureTarget objects being available.

concerns.

## Renderer

While the `Renderer` object is still likely to be the first CEGUI related object client code creates, it is no longer the central hub of the rendering process. In order to optimise rendering, and also make rendering more flexible, most of the jobs that the old renderer module did have either been eliminated entirely or re-factored into other parts of the rendering subsystem. This leaves `Renderer` serving two main functions; as a place to get or set system-wide graphics information and settings, and as a factory for other objects.

Implementation of `Renderer` is now largely trivial. One recommendation we are now making is for the use of static `'create'` and `'destroy'` functions that will call the object constructor(s) and destructor. The main reason for this is that it makes modification of the implementation without breaking binary compatibility easier to achieve<sup>10</sup>.

The main task of the constructor should be to set up a default `RenderingRoot` object that can be returned later. `RenderingRoot` is a `RenderingSurface` subclass<sup>11</sup>.

The `beginRendering` and `endRendering` functions are used to initialise and clean up any *global level* systems or states required. We emphasise *global level*, because individual `RenderTarget` objects also get a chance to perform additional initialisation and cleanup steps.

Aside from this, much of the interface is concerned with creating and destroying various other objects, such as `GeometryBuffer` objects, `Texture` objects and `TextureTarget` objects.

The informational part of the interface has some familiar functions, although generally these have changed the return types to something more appropriate.

## Texture

The `Texture` interface has survived relatively in tact. Changes have been made to offer a more minimalistic interface, although everything is pretty much still there.

We have added a `saveToMemory` function that dumps the texture content to a memory buffer.

Implementation is simple, and if you have existing `Texture` implementations these can be adapted with very little effort.

One thing that has changed and needs to be considered when updating `Texture` classes is that the image codec system, previously used with the OpenGL renderer, has been promoted to being a major component of the core system. In order to fully compliant with the CEGUI way of working, your image loading should now be done via whichever `ImageCodec` is active in the system (you can get the current `ImageCodec` from the `System` object via the `getImageCodec` member function. If the API or engine that you are implementing a renderer module for has it's own image loading facilities, for maximum flexibility you should consider interfacing with those facilities via the CEGUI image codec and resource provider systems rather than directly.

## GeometryBuffer

The concept of the `GeometryBuffer` is to actually represent some renderable object, by providing a means to cache geometry and other related state such as position, rotations and clipping.

---

<sup>10</sup> Other techniques obviously exist, and you are free to use those instead or as well as create and destroy functions.

<sup>11</sup> That currently does nothing in addition to `RenderingSurface` but is used to prevent a `RenderingWindow` being used as a root (which is not logical, since a `RenderingWindow` is owned by some other `RenderingSurface`).

A `GeometryBuffer` has an active `Texture`, and geometry added uses the active `Texture` until a new `Texture` is set. This means that internally the `GeometryBuffer` has to keep track of these changes. In the OpenGL implementation we create internal per-texture 'batches' that are sent to the API when the `GeometryBuffer` is drawn.

The `GeometryBuffer` is passed `Vertex` objects to be added to the internal buffer(s). The geometry that CEGUI supplies is position independent, with the origin being the top-left corner<sup>12</sup>.

It is *highly recommended* that the `Vertex` objects be converted and stored – at the point of them being added – in a format that is more easily used by the underlying API or engine; this is to minimise the subsequent work that needs to be done when the geometry buffer is eventually drawn. For example, the OpenGL implementation stores the added geometry in a way that is *directly* usable by the OpenGL implementation. When the time comes to draw the geometry there is virtually zero additional overhead incurred.

The implementation of the rest of the interface should be fairly straightforward, one thing to note is that if you want to have `RenderEffect` support, on any assigned `RenderEffect` object you should call `performPreRenderFunctions` before performing the actual `GeometryBuffer` drawing and `performPostRenderFunctions` after you're done drawing.

## RenderTarget

`RenderTarget` is really a group of class types, that also encapsulates much of `TextureTarget`. The main function of the `RenderTarget` is drawing. The draw functions take a `GeometryBuffer` or a `RenderQueue`, and basically render whatever is contained therein onto whatever the `RenderTarget` is representing (such as the screen or a texture).

The drawing for the OpenGL implementation is actually done by the `GeometryBuffer`, although depending upon the API or engine, the `RenderTarget` might need to take more of an active role.

Other things to note are the `activate` and `deactivate` functions, these are used to perform additional initialisation and clean up required for specific target types.

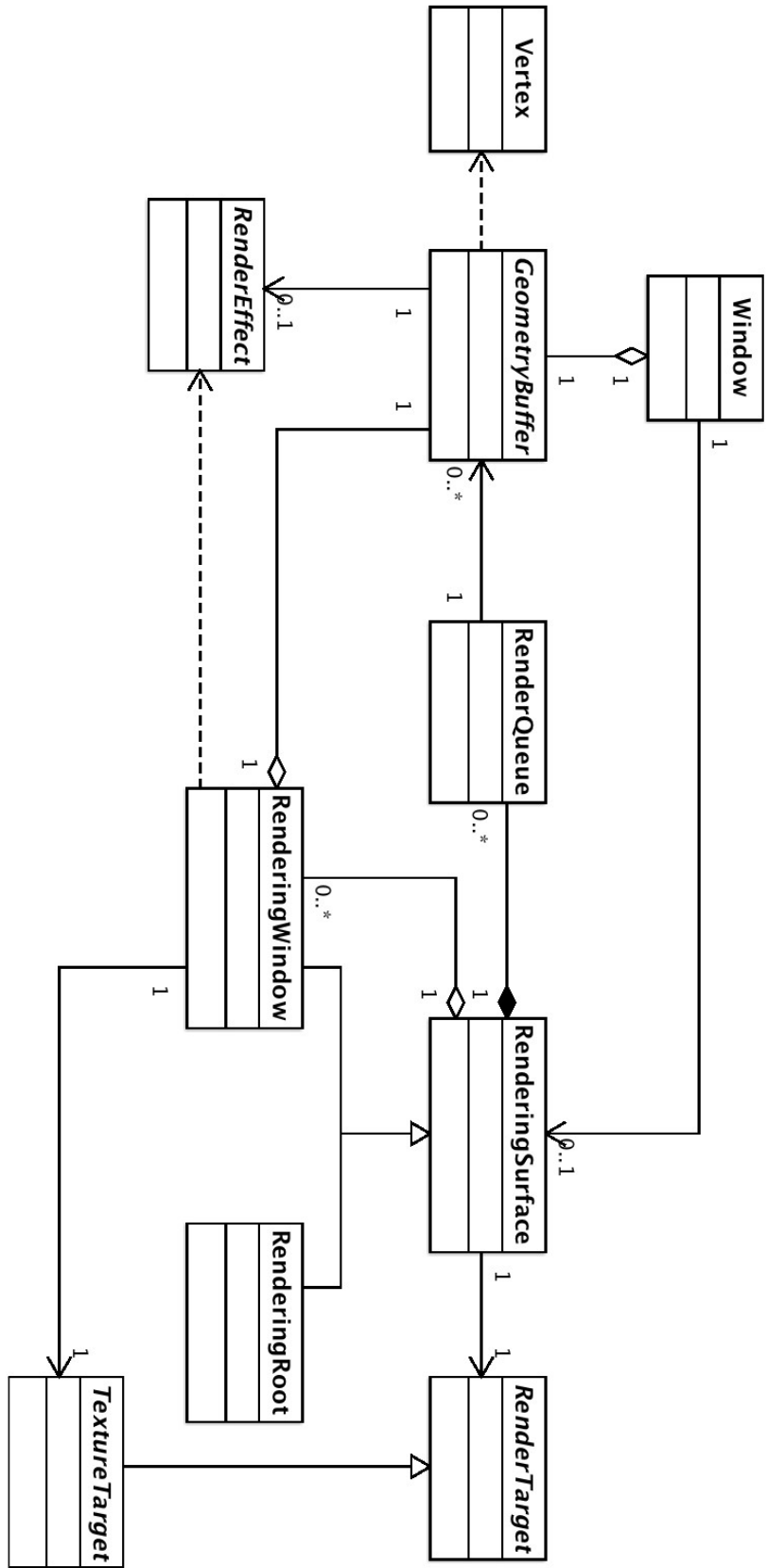
## TextureTarget

`TextureTarget` is, as mentioned, a specialisation of `RenderTarget` and all of what is said there applies here also.

In addition, `TextureTarget` provides a `clear` function that should clear the render texture, a `declareRenderSize` function that should be used to resize the render texture as needed, and finally a `getTexture` function to return the underlying texture as a `CEGUI::Texture` object (this object should usually remain owned by the `TextureTarget`).

---

<sup>12</sup> As you might expect in a system that's based largely on the concept of rectangles.



Drawing 1: Basic Rendering Subsystem Classes