Kakadu Survey Documentation (last updated for Version 7.5)

David Taubman, UNSW

August 25, 2014

Note: Kakadu comes with an extensive, integrated documentation system which is automatically generated from the structure and extensive comments embedded in the publically includable header files. This documentation consists of almost 1000 HTML pages and other documents, accessed through the *"documentation/index.html"* file. You may find it best to read the present document from within the integrated documentation system; you will need a license to the Kakadu software to be able to generate and use this documentation system though.

Especially if you do not have a license to the full Kakadu toolkit, you may find it helpful to refer to the additional publically available documents, *"Overview.txt", "Usage-Examples.txt", "java-interfaces.pdf"* and *"jpip-kakadu.pdf"*.

1 Introduction

Kakadu is an implementation of Part 1 of the JPEG2000 standard. It should be fully conformant with "Profile-1", "Class-2", as defined in Part 4 of the standard, which describes compliance. It conforms at "Profile-2" (unrestricted profile), except in a number of minor (quite outlandish) respects, where oversights in the original standard had to be corrected through minor restrictions in Profile-0 and Profile-1.

Kakadu also implements many features from other parts of the JPEG2000 standard, including:

- Virtually all features of the JPX file format, described by Part 2 of the JPEG2000 standard, including rich colour spaces, multiple compositing layers, animation and rich metadata support.
- All aspects of the Motion JPEG2000 standard (Part 3) which apply to video.
- Some of the most useful technology enhancements from JPEG2000 Part 2, including multi-component transforms (IS 15444-2 Annex J), arbitrary

transform kernels (IS 15444-2 Annexes G and H), and arbitrary decomposition structures plus arbitrary downsampling factor styles (IS 15444-2 Annex F).

- Client and server components for a comprehensive implementation of the JPIP (IS 15444-9) standard for interactive image communications.

Unfortunately, it is difficult to interact with the JPEG2000 standard without some appreciation for the technology involved. This document does not claim to provide such an appreciation, although certainly it should prove helpful. The reader is referred to the book, "JPEG2000: image compression fundamentals, standards and practice" by Taubman and Marcellin, published by Kluwer Academic Publishers in November 2001.

Nevertheless, developers should find that the example programs provide sufficient insight to start working with Kakadu and hence with JPEG2000 right away. **If your objective is to get up and running with sophisticated decompression and rendering functions as soon as possible, the best place to start would be the *"kdu-render"* example, which contains four short demonstration code fragments based around the powerful *kdu-region-decompressor* and *kdu-region-compositor* objects. The last of these requires only a few lines of code to render just about any JPEG2000 source (raw codestream, JP2 file, JPX file, JPX animation or MJ2 video track) to a memory buffer.**

If, on the other hand, your objective is to understand the Kakadu framework better, a recommended plan is to start with the "*simple-example-c*" and "*simple-example-d*" examples. After understanding these, new developers are encouraged to look at the "*kdu-buffered-compress*", "*kdu-buffered-expand*" and "*kdu-render*" examples, which provide an introduction to some extremely useful high level objects. These examples still fail to cover more than a small fraction of the capabilities offered by Kakadu. For more advanced applications, quite a number of other demonstration applications are provided.

1.1 Features

The core system is intended to be fully platform independent and to be well-suited to both embedded systems and higher level operating systems. The target platform must at least support 32-bit integers. Various processing enhancements are currently available for x86 family processors, for ARM processors, for Power PC processors, and for SPARC and UltraSPARC processors.

Due to the complexity of the JPEG2000 standard, a key focus in the implementation is on memory efficiency and execution speed. Another key focus is the provision of a highly flexible architecture, which is able to meet the demands of a large diversity of applications from simple compression or decompression filters to sophisticated interactive applications, client-server systems and network transcoders.

Kakadu provides a very sophisticated multi-threaded core sub-system that can utilize all processing resources on most modern platforms, almost transparently.

Kakadu provides fault tolerant handling of virtually all error conditions, through a fully customizable error and warning handling mechanism.

Many of the capabilities of the system are demonstrated by a number of example applications. These range from a highly sophisticated interactive viewer and remote image browsing utility, "*kdu-show*", to very simple (and quite naive) examples involving buffer-oriented compression and decompression.

Kakadu's automatic documentation building utility can also build a full set of Java native interfaces, capable of exposing virtually every aspect of the Kakadu system to Java applications. The same system also builds managed native interfaces for the Microsoft^{TM} managed languages such as C# and Visual Basic. Java, C# and VB interfaces are automatically documented along with the native C++ functions in the HTML docs built by the "*kdu-hyperdoc*" utility. For instructions on building and using Java interfaces and managed native interfaces, see the separate "*java-and-managed-interfaces.pdf*" document.

Kakadu provides rich support for 3 of the 4 wrapping file formats which have been developed by the JPEG committee: the baseline JP2 file format; the extended JPX file format; and the motion MJ2 file format. The offered support includes rich colour space conversion capabilities, compositing, animataion, metadata management, and metadata-driven visual overlays, amongst other things.

1.2 Typesetting Matters

In the ensuing text, names found in the Kakadu software appear in italics. Moreover, underscore characters appearing in the actual system names are typeset instead as dashes, for improved readability.

2 Overview

2.1 Core Sub-Systems

Figure 1 illustrates the most significant sub-systems of the core Kakadu architecture. Dashed lines in the figure identify interface objects which are used to interact with internal implementations. In most cases, the interface objects have no storage of their own, serving primarily to isolate the sub-systems from one another and from the application. The directions of the arrows on these lines are not intended to suggest data flow, but rather the provision of services. Only a small subset of the interactions are actually shown in the figure. For convenience, we have omitted the ROI (Region of Interest) processing sub-system from this figure. ROI processing is discussed later in Section 6. We have also omitted the interfaces relevant to directly accessing JPEG2000 precinct data.

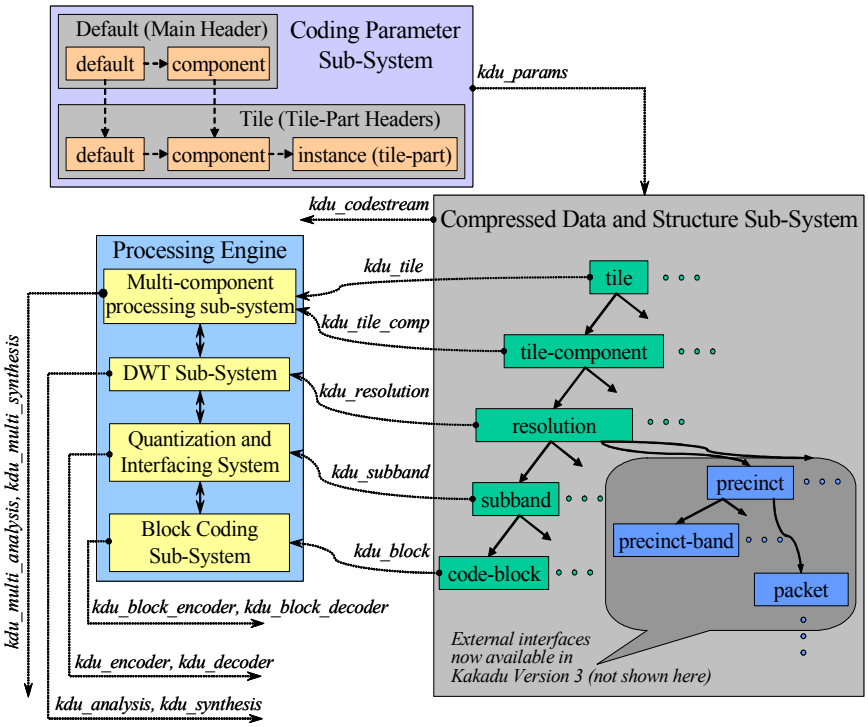

Figure 1: Kakadu core system architecture, showing the most significant sub-systems. Access to the internal implementation of each sub-system is obtained via interface objects whose class names are indicated in italics. Not all classes or interfaces are shown here.

Before proceeding with a description of the core sub-systems, we note that Kakadu provides many other objects and sub-systems which build upon the core being discussed here. These provide support for JPEG2000 file formats and interactive client/server systems, as well as providing support specifically targeted at helping developers to easily interface their applications to Kakadu. Not shown in the figure is the core multi-threading sub-system; however, this would add unnecessary complication to the current introductory description.

2.1.1 Coding Parameter Sub-System

In JPEG2000, coding parameters are described through a hierarchy of specifications with successively narrower scope. At the top level, most coding parameters are usually described through main header defaults, which apply to all image components of all tiles, unless explicitly overridden. The main header of a JPEG2000 code-stream may also contain component-specific overrides for the default coding parameters. Tile headers may override the default specifications

for an individual tile or just an individual tile-component. Finally, coding parameter information may be updated incrementally over multiple tile-parts within a tile. This structure and the appropriate parameter inheritance mechanisms are reflected in the coding parameter sub-system. It is worth noting that the JPEG2000 syntax contains a number of irregularities, which Kakadu conceals from the user (or application developer). In this way, Kakadu presents a more consistent interface to the coding parameters.

The coding parameter sub-system consists of a collection of *kdu-params* objects (actually, *kdu-params* is an abstract base class for various derived coding parameter objects), organized internally through multi-dimensional lists. Most interaction with the parameter sub-system is through the single *kdu-params* object at the root of the multi-dimensional structure. This root object is invariably an instance (there is only one) of the derived *siz-params* class. Thus, the *kdu-codestream::access-siz* member function returns a reference to the root *kdu-params* object for the entire collection of coding parameters.

There are many advantages to maintaining the coding parameters in a completely separate sub-system from that used to manage compressed data (see below). It is sufficient here to note that interaction between individual parameter specifications is of a quite different nature to the interaction between the various structural elements associated with the compressed data. Although both involve tiles and tile-components, the concepts of main header defaults and tile-parts are anathema to the compressed data.

In addition to providing a uniform interface to the coding parameters which might be embodied in a JPEG2000 code-stream, the parameter sub-system provides quite a number of other useful services. It is able to establish appropriate default coding parameters, taking into account the parameters which the user or the application is willing to specify explicitly. The parameter sub-system is able to generate a copy of itself which reflects the effects of various useful transcoding operations, such as resolution scaling, rotation and restrictions on the number of image components. It is able to accept and externalize coding parameter specifications in a variety of different forms, including:

- parsing or generating code-stream marker segments;
- parsing or generating textual descriptions;
- explicit calls to a rich set of strongly typed binary function interfaces.

Although not strictly necessary for Part 1 of the standard, the parameter sub-system is designed to be easily extended to incorporate new coding parameters without impacting applications. The parameter sub-system is also a useful repository of some coding parameters which are not actually recorded in the code-stream. These include rate control optimization hints and encoder-side ROI (Region of Interest) specifications.

It is worth noting that the parameter sub-system supports incremental updating of the coding parameter information while compression or decompression is in progress. This is useful for decompression, since it means that a

JPEG2000 code-stream (e.g., a file) can be processed incrementally. It is also useful for transcoding operations, where one code-stream must be generated from an existing one and possibly for compression applications, where the coding parameters for some tiles might be deferred until the tile is encountered. For more information on all these topics, consult the interface function descriptions in *"kdu-params.h"*.

2.1.2 Compressed Image Management Sub-System

Perhaps the most substantial core sub-system in Kakadu is that which manages compressed data and the structure of the compressed image representation. This sub-system involves a family of hierarchically related interface objects, which interface with a substantial internal machinery for interacting with the compressed image representation. At the top of the interface hierarchy is the *kdu-codestream* object. Applications usually create only one of these objects, configuring it either for output (generating a JPEG2000 code-stream) or input (expanding a JPEG2000 code-stream). It is also possible to configure *kdu-codestream* objects for a different type of behaviour known as "interchange," which is particularly useful for interactive server applications. Input, output and interchange codestream objects are discussed further in Section 3.

From the *kdu-codestream* object, it is possible to gain access to subordinate structural elements through the *kdu-tile*, *kdu-tile-component*, *kdu-resolution* and *kdu-subband* interface objects. When created for "interchange" (rather than input or output), a *kdu-precinct* interface object is also available. Like *kdu-codestream* itself, each of these objects is only an interface, containing an appropriate pointer into the actual internal machinery. Interfaces have no storage of their own and may come and go without any impact on the internal machinery. Copying interface objects simply duplicates the reference, but again has no impact on the internal machinery.

Apart from isolating the application and other sub-systems from the internal implementation, these interfaces implement appearance transformations which alter the way in which the compressed image representation appears from the outside. The *kdu-codestream::change-appearance* and *kdu-codestream::apply-input-restrictions* member functions may be used to alter this appearance. In particular, the compressed image representation presented by the interfaces may contain a restricted set of image components, a reduced resolution view, a restricted spatial region (viewing portal) or a geometrically transformed view (rotated or flipped). The behaviour of the appearance transformations is such that clients of the interfaces see the compressed data as though the original image had been subjected to these appearance transformations prior to compression. This is extremely handy in implementing interactive applications, which often maintain only a restricted view into a larger compressed representation. It is also convenient for applications which need to consume the image samples in a different order (e.g., left to right) to that in which they were originally compressed (e.g., top to bottom).

It is important to bear in mind that appearance transformations are not re-

flected in the information managed by the coding parameter sub-system (see above) associated with the compressed data[1]. For this reason, most or all interaction with the compressed data structure should be through the *kdu-codestream*, *kdu-tile*, *kdu-tile-component*, *kdu-resolution* and *kdu-subband* interfaces.

At the bottom of the compressed data interface hierarchy is the *kdu-block* class, which provides direct access to the embedded bit-stream representing an individual code-block, along with other information which may be used to identify the relative importance of the embedded pieces (coding passes) in relation to those found in other code-blocks. Such information is available for both input and output codestream objects and is particularly useful for transcoding applications.

2.1.3 Data Processing Sub-Systems

In order to actually compress or decompress image sample data, it is necessary to perform wavelet transform (DWT), quantization, block coding and any required multi-component and colour transformation operations. These are managed by data processing systems which may be "bolted" onto the compressed data sub-system, as suggested by Figure 1. The relevant interface descriptions may be found in the core header files, *"kdu-sample-processing.h"* and *"kdu-block-coding.h"*.

Data processing sub-systems are discussed further in Section 5. For the moment, however, it is helpful to make several observations. Firstly, the data processing objects interact with the compressed data through the *kdu-resolution*, *kdu-subband* and *kdu-block* interfaces mentioned above, which implement appearance transformations. For this reason, the processing and its results are interpreted with respect to the transformed image appearance (resolution, orientation and spatial viewing portal). This means that applications are released from the burden of performing such manipulations themselves.

Secondly, the data processing sub-systems are designed to put themselves together into complete processing engines. For example, constructing a *kdu-analysis* object automatically constructs all appropriate *kdu-encoder* and *kdu-block-encoder* objects required to completely transform image samples into compressed code-block bit-streams, shipping them off to the compressed data sub-system as they become available. Similarly, *kdu-synthesis* objects automatically construct all appropriate *kdu-decoder* and *kdu-block-decoder* objects and attach them to the compressed data sub-system.

Processing engines constructed for forward processing (compression) of individual image components have a common abstract base class, *kdu-push-ifc*, which may be used to interact with the processing engine without regard for the elements which it embodies. Similarly, processing engines constructed for

[1] It is, however, possible to get the parameter sub-system to copy itself into another suitably transformed parameter sub-system, which can reflect appearance transformations other those associated with restricted spatial regions.

reverse processing (decompression) of individual components have a common abstract base class, *kdu-pull-ifc*.

From version 5.0, there are two new core processing objects, *kdu-multi-analysis* and *kdu-multi-synthesis*, which build upon the *kdu-push-ifc* and *kdu-pull-ifc* derived objects to provide complete multi-component processing solutions. Prior to the introduction of Part-2 multi-component transform support in version 5.0, the only required inter-component processing was the conversion of RGB components to and from a luminance-chrominance representation, as performed by the unbound functions, *kdu-convert-rgb-to-ycc* and *kdu-convert-ycc-to-rgb*. With multi-component transforms, however, the variety of operations which may be required can become highly complex. For this reason, the *kdu-multi-analysis* and *kdu-multi-synthesis* objects have been added. These objects allow applications to push and pull image component lines without regard for what inter-component transformations might be involved.

2.2 Embedded Documentation

The Kakadu software is supposed to be largely self-documenting. High level access to the system is conducted entirely through interface functions declared in one or another of the header files in the *"coresys/common"* directory. In particular, coding parameters, compressed data and data processing capabilities are accessed via the functions declared in *"kdu-params.h"*, *"kdu-compressed.h"* and *"kdu-sample-processing.h,"* respectively. Each function is carefully documented in its respective header file and these comments should be understood and used as the principle source of documentation for the system.

It is now possible (and encouraged) to access virtually all documentation through the extensively hyperlinked, integrated documentation system constructed using the *"kdu-hyperdoc"* utility. The root of this documentation system is at *"documentation/index.html"*, which is constructed automatically when you compile the *"kdu-hyperdoc"* utility. If you cannot find this file, consult the instructions appearing in *"documentation/README.TXT"*.

If some ambiguity remains in the descriptions of the various high level interface functions, it may often be resolved by examining the implementation of the relevant function, although we would appreciate your informing us of this ambiguity (email: d.taubman@unsw.edu.au) so that the descriptions can be clarified in future releases.

The Kakadu software comes with a number of useful examples. The *"simple-example-c.cpp"* and *"simple-example-d.cpp"* files contain very simple examples of the use of Kakadu for image compression and decompression, respectively. These examples do not use very many of the interesting features of the Kakadu system and do not represent the most efficient solution to many applications which developers may encounter. Nevertheless, they embody the type of processing paradigms which most developers may initially expect and they also provide an easy introduction to the system. The files contain comments suggesting how the developer might exploit more of the features of Kakadu. It is best to work through these examples while keeping the header files *"kdu-compressed.h"*,

"kdu-stripe-compressor" and *"kdu-stripe-decompressor"* open, since these contain the complete descriptions of the various interface functions which are being used. Alternatively, use the HTML documentation system to read about the various object's and their interface functions, as you encounter them in the example applications.

For a richer demonstration of compression in Kakadu, you may find it useful to build upon the *"kdu-buffered-compress"* example application. This application can readily be extended to include all the functionality of *"kdu-compress"* but we deliberately stop short of that so as to leave it as accessible as possible. Many developers wish to use Kakadu to compress images they have either partially or fully in internal memory buffers, which is exactly what *"kdu-buffered-compress"* is designed to illustrate.

For non-interactive decompression applications, you may well wish to build on top of the *"kdu-buffered-expand"* example application. Again, this application can easily be extended to encompass the full functionality of *"kdu-expand"*, but we stop short of that mark so as to make it more accessible as an example.

The high level objects mentioned above are concerned directly with JPEG2000 codestreams, as exposed via the *kdu-codestream* interface. This is sufficient when working with raw codestreams and often sufficient when working only with JP2 files, since they are only able to embody a single codestream. Even with JP2 files, however, many applications will be interested in colour conversion and/or image component resampling, based on the descriptions embodied in the JP2 header. These operations can be performed explicitly by the application, particularly with the aid of the *jp2-colour-convert* object. However, for application developers seeking one-stop solutions to all of the colour conversion and resampling problems which might arise in a typical application, the *kdu-region-decompressor* and *kdu-region-compositor* objects should prove immensely valuable. The latter object builds upon *kdu-region-decompressor* to add all the functionality required for rendering compositing layers and animated compositions from JPX files, frames from MJ2 (Motion JPEG2000) files, and much more.

For a dead-easy introduction to the *kdu-region-decompressor* and *kdu-region-compositor* objects, developers are strongly encouraged to review the *"kdu-render"* application, which contains four demo functions: two based upon *kdu-region-decompressor*; and two based upon *kdu-region-compositor*. The most sophisticated of these will correctly render a frame from any raw codestream, JP2 file, JPX compositing layer, JPX animation or MJ2 video track – it is only 50 lines long.

While simple code fragments which can decompress any JPEG2000 source into a memory buffer are nice to have around, JPEG2000 actually comes into its own in applications where images are so big that rendering the whole thing is unnecessary and possibly altogether infeasible – consider a 100 Tera-pixel image, for example. This is also where Kakadu truly excels. For this reason, you should view these early examples as ways to lead you into more sophisticated use of the Kakadu interfaces.

For interactive decompression (e.g., decompression for an interactive viewer,

or an image browser), you may like to start with introductory examples of the *kdu-region-decompressor* or *kdu-region-compositor* interfaces, via the *"kdu-render"* application. For serious applications of the interactive functionality offered by Kakadu, however, you should review the extensive documentation accompanying these objects (I would suggest the higher level one, *kdu-region-compositor*, since it builds on *kdu-region-decompressor*) and/or look at how they are used within *"kdu-show"*. All of these objects, on which the most interesting Kakadu demo applications are built, are fully platform independent in their implementation; however, some of the actual applications do contain some platform dependent code. This is true of *"kdu-show"* and *"kdu-server"*, which require networking and multi-threading.

Other demonstration applications which you may find yourself referencing include *"kdu-compress"*, *"kdu-expand"* and *"kdu-transcode"*. Many developers may find it useful to borrow significant code fragments from these applications. On the other hand, for obvious reasons the source code for these applications must be more complex than that encountered in the simpler compression/decompression examples.

For developers who are intending to get deeply involved with the Kakadu implementation, adding new capabilities or accessing internal properties which are not currently exported, the following pointers may prove helpful:

- As a rule, header files are the richest source of comments, since good programming practice usually dictates that individual objects be easily understood from their class or structure declarations, rather than the implementation of member functions.

- Class, structure, function or other names which begin with the prefix *kdu-*, *jp2-*, *jpx* and so forth, are either high level exported entities, or may potentially become high level exported entities. Class, structure, function or other entities which might be shared amongst multiple source files, but are best understood as contributing to the internal implementation of a single sub-system, generally have names beginning with *kd-*, *j2-*, *jx-* and so forth.

2.3 Programming Conventions

In addition to the directory structure and naming conventions mentioned above, here is a list of some of the programming conventions which have been adopted in implementing Kakadu.

- All source files have line lengths of 79 characters or less and tab characters are not used. This improves readability across a variety of different platforms and editors.

- We have tried to stick entirely to ANSI C++ for the core system, i.e., you should not find any platform specific services used there.

- The core system does not rely in any way upon the C++ I/O system, since this has not universally supported on different platforms.
- Source files are organized in a bottom-up fashion (some programmers prefer top-down, but this leads to more forward declarations). Internal functions are declared *static* and appear at the top of the file; class implementations follow; and global external functions (there are very few of these) usually appear last in the file.
- To ensure re-entrant code, static local variables are never used and global variables are used only to hold invariant constants. For example, a lookup table may be declared globally, but only if its contents are guaranteed to be the same in every circumstance where the system is used.
- The *"coresys/common"* directory is not polluted with internal implementation details. Instead, header files specific to the implementation of any sub-system are generally included in the same directory as the corresponding source files and assigned names concluding with the suffix, *"-local.h"*.
- Several techniques are employed to gain strong isolation between the appearance (or interface) to a sub-system and its actual implementation.
  - Of course, the usual C++ encapsulation paradigms are employed throughout.
  - A second level of protection is afforded by the use of interface classes, which embed references to the actual internal implementation objects. The interface functions provide means of interacting with the internal object, while not exposing its details to the user. This approach is employed extensively to reinforce the distinction between the internal implementation of the core system and its interface to applications. Multiple interfaces can generally be maintained to the same internal object and the interface objects do not generally have destructors, since they do not have any resources of their own. When appropriate, destruction of the internal object through an interface is usually conducted through an interface function named *destroy*.
  - The directory structure itself represents a form of encapsulation, whose boundaries are carefully respected. The local header files appearing in any given directory (other than "core/common") are not included by source files outside that directory, helping to isolate the various sub-systems.
  - Finally, C++ namespaces (see below) are used to minimize the risk of name collisions, especially if you choose to statically link some or all of the kakadu SDK into your application.

2.4 Native C++ Namespaces

All the Kakadu names (functions, classes, types, etc.) are defined within one of a number of C++ namespaces, as follows:

- The *kdu-core* namespace contains all public names that are intended to be exposed by the Kakadu core system.

  The *kd-core-local* namespace contains all private implementation names that are found in the core system; these are generally defined in the *"-local.h"* headers mentioned above.

  The *kd-core-simd* namespace contains all SIMD (i.e., vector processing) accelerator names that are used in the implementation of the core system – these are mostly names of architecture-specific functions that begin with a prefix that suggests the instruction set being used (e.g., *ssse3-*, *avx-*, *avx2-*, *neon-*, etc.).

- The *kdu-supp* namespace contains all public names that are intended to be exposed by highly re-usable interfaces found outside the core system – these interfaces are found within the "apps" sub-directory, rather than the "coresys" sub-directory of the Kakadu source tree. The most important of these interfaces are: the file format API's such as *jp2-source*, *jp2-target*, *jpx-source*, *jpx-target*, *mj2-source*, *mj2-target*, etc.; the data processing support API's *kdu-stripe-compressor*, *kdu-stripe-decompressor*, *kdu-region-decompressor* and *kdu-region-compositor*; and the higher level interactive communication and animation API's such as *kdu-cache*, *kdu-client*, *kdu-serve* and *kdu-region-animator*. There are many additional important classes defined within the *kdu-supp* namespace, so that almost all applications that build upon Kakadu will need to use this namespace.

- The *kd-supp-local* namespace mirrors the *kd-core-local* namespace, holding local names that are used in the implementation of the interface exported publically by the *kdu-supp* namespace.

- The *kd-supp-simd* namespace mirrors the *kd-core-simd* namespace, holding the names of architecture-specific vector processing accelerator functions that are used in the implementations of the *kdu-supp* API's.

At the application level, the only two namespaces that are important are kdu-core and kdu-supp; however, kdu-supp uses kdu-core, so it is usually sufficient to include a statement of the form

```
"using namespace kdu-supp;"
```

in source files that reference Kakadu API's, while it may sometimes be necessary to add

```
"using namespace kdu-core;"
```

You should never need to use any of the other (internal) namespaces mentioned above.

In addition to the above, there are some particularly useful API's that are defined within the implementation of specific demonstration applications that are expected to have widespread use. Where this is the case, these API's have been given their own namespace in the relevant header file. Two of particular interest are the *kdu-supp-vex* and *kdu-supp-vcom* namespaces that contain re-usuable API's defined by the *"kdu-vex.h"* and *"kdu-vcom.h"* header files that form part of the implementation of the *"kdu-vex-fast"* and *"kdu-vcom-fast"* video decompression/compression demo apps.

3 The "kdu-codestream" Interface

The material presented here is only of a summary nature. For more detailed information and information regarding the many capabilities not described here, the developer is expected to work with the interface descriptions appearing in the core header file, *"kdu-compressed.h"*.

3.1 The Canvas Coordinate System

JPEG2000 compressed images must be understood in terms of a family of spatial partitions, all of which are defined in a common coordinate space known as the "canvas coordinate system". Individual image components (usually colour components) may have different dimensions, but they are all related through sub-sampling factors to a single consistent "image region", defined on the canvas.

The upper left hand corner of the image region on the canvas has coordinates $[E_1, E_2]$, where the first coordinate refers to the vertical location (displacement below the canvas origin) and the second refers to the horizontal location (displacement to the right of the canvas origin) of the region. The lower right hand corner of the image region has coordinates $[F_1 - 1, F_2 - 1]$, so that the image region has height $F_1 - E_1$ and width $F_2 - E_2$. In JPEG2000, all coordinates must be non-negative. However, the coordinate system works perfectly well with signed coordinates and Kakadu exploits this fact to implement appearance transformations. As a result, the coordinates returned by the various interface objects of the compressed data sub-system (*kdu-codestream*, *kdu-tile*, *kdu-tile-component*, ...) are all signed quantities which could be negative.

Every sample of every image component has a notional location on the canvas. In particular, the samples of image component c have notional locations

$$[n_1 S_1^c, n_2 S_2^c]$$

where n_1 and n_2 are integers and S_1^c and S_2^c are vertical and horizontal sub-sampling factors. As a result, the image region may be mapped to a component region for any given component, c, through the following equations:

$$E_i^c = \left\lceil \frac{E_i}{S_i^c} \right\rceil, \quad F_i^c = \left\lceil \frac{F_i}{S_i^c} \right\rceil, \quad i = 1, 2$$

This is also illustrated in Figure 2.

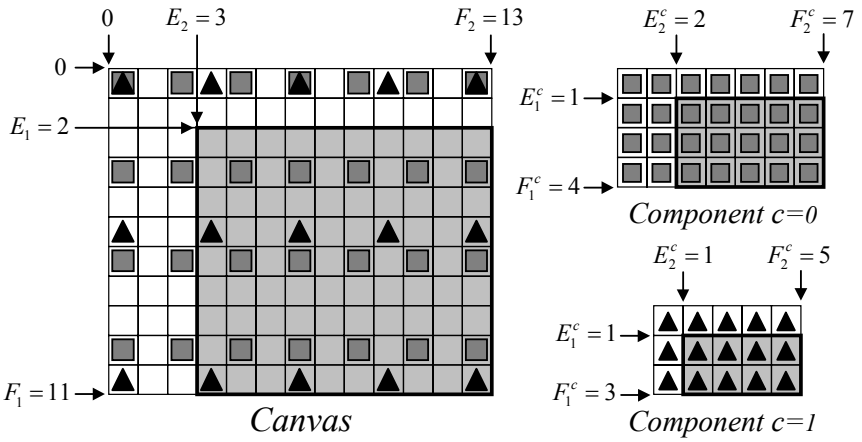

Figure 2: Notional placement of component samples on the canvas.

Tiles are also defined on the canvas as partitioning the image region. In many cases there may be only one tile for the entire image, no matter how large it is, and this is often desirable since tiling may lead to compression artefacts. Nevertheless, decompressors at least must be prepared for arbitrary tilings. The tile partition is described in terms of an anchor point, $\left[\Omega_1^{\mathrm{T}}, \Omega_2^{\mathrm{T}}\right]$, and tile dimensions, $T_1 \times T_2$, leading to the partition shown in Figure 3.

Individual tiles may be mapped into each image component region through the notional placement of component samples, leading to the following tile-component regions:

$$E_i^{\mathbf{t},c} = \left\lceil \frac{E_i^{\mathbf{t}}}{S_i^c} \right\rceil, \quad F_i^{\mathbf{t},c} = \left\lceil \frac{F_i^{\mathbf{t}}}{S_i^c} \right\rceil, \quad i = 1, 2$$

where $\left[E_1^{\mathbf{t}}, E_2^{\mathbf{t}}\right]$ and $\left[F_1^{\mathbf{t}} - 1, F_2^{\mathbf{t}} - 1\right]$ are the coordinates of the tile's upper-left and lower-right corner on the canvas.

These mapping rules are extended to reduced resolution subsets of the compressed images and also to a variety of other partitions which form the basis of efficient coding. Application developers are not expected to implement the region mapping rules and need not thoroughly understand them, since the Kakadu system implements all necessary mappings and also subjects them to the necessary appearance transformations. Nevertheless, a number of cautionary observations are in order:

- Even though the elements of the tile partition all have the same dimensions (except at boundaries) on the canvas, they might have different dimensions when mapped into the domain of any given image component. The dimensions might differ not only from component to component, but also from tile to tile within a component. This happens whenever the tile di-

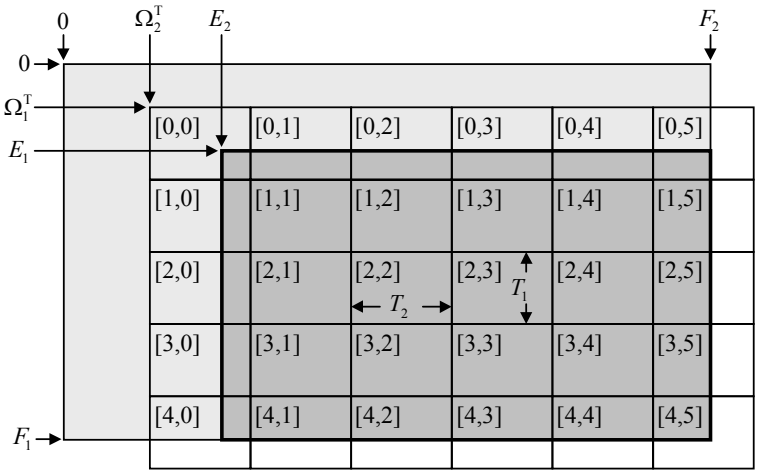

Figure 3: Tile partition on the canvas.

mensions, T_1 and T_2, are not divisible by the component sub-sampling factors.

- There is no guarantee that any image component will have sub-sampling factors of 1. As a result, it is generally a mistake to determine image dimensions from the dimensions of the image region on the canvas. Instead, the *kdu-codestream::get-dims* member function may be used to determine the dimensions of individual image component regions.

- In order to restrict the appearance of the compressed image to a limited spatial region (viewing portal), the relevant region must be provided in canvas coordinates. It is important to realize that this may be related only indirectly to the region occupied by a particular image component. In interactive applications, it often happens that there is a need to determine the region on the canvas which corresponds to a particular region within an image component. The mapping of image component regions to the full high resolution canvas coordinates is not unique, but a suitable mapping is provided by *kdu-codestream::map-region* for the benefit of these applications.

- It can happen that a tile contains no samples from some particular image component. In fact, it can happen that no component contributes any samples at all to a particular tile. This is true, even though tiles are required to have a non-empty intersection with the image region on the high resolution canvas. Developers must be prepared for this possibility. In general, the dimensions of a tile-component, at any of the resolutions which it offers, may be determined by using the *kdu-resolution::get-dims* member function and this is the recommended method.

For a more comprehensive treatment of JPEG2000 partitions, the reader is referred to Chapter 11 of the book by Taubman and Marcellin.

3.2 Code-Stream Input

To process an existing JPEG2000 code-stream, the input form of the *kdu-codestream* object's *create* function should be used. This form accepts objects derived from the abstract base class, *kdu-compressed-source*. Compressed data sources may include anything from simple file readers, to custom file format parsers and even special network management systems. They may represent streaming sources (read once), seekable devices or random access devices with caching capabilities.

Several types of sources are currently offered (these belong to the application-specific part of the Kakadu package, rather than the core system). The simplest of these data sources is represented by the derived class, *kdu-simple-file-source*, which does nothing other than provide reading and seeking interfaces to an open file stream. Developers interested in creating their own sources (e.g., memory buffered data sources, network connected data sources and so forth) would do well to build upon this very simple derived class.

Figure 4 provides an overview of the various compressed source objects offered by Kakadu and some of the ways in which they may be used. JPEG2000 sources are either raw code-streams or else code-streams which are wrapped in one of the family of file formats described by the various parts of the JPEG2000 standard. Kakadu provides a generic mechanism which can be used to build support for any member of the JPEG2000 family of file formats. All of these formats share a common box construct, which we identify as a JP2 box. Kakadu's *jp2-input-box* class is actually derived from *kdu-compressed-source*, meaning that any box which contains an embedded code-stream can be passed straight into *kdu-codestream::create*. Kakadu provides a generic set of primitives for navigating around arbitrary JP2 family file formats, including the ability to manage multiple open boxes within a single file at the same time.

To open a *jp2-input-box* or a *jp2-source* object, you first need to construct and open a *jp2-family-src* object, which provides a common mechanism for interfacing JP2 family file format parsers with a variety of data sources which may provide the ultimate content of the file (including content which is dynamically served by a JPIP server). The jp2-family-src object is then used to open the *jp2-input-box* or *jp2-source* object. The *jp2-source::read-header* function reads a JP2 file's header, after which the application can access and exploit all the rendering information contained in the JP2 file. The rendering information offered by *jp2-source* objects may be directly imported by the high-level support object *kdu-region-decompressor*, which then does all the rendering work for you.

Kakadu contains extensive support for interactive client-server applications, based on the forthcoming JPIP standard (JPEG2000 Part 9). At the lowest level, this is achieved by supplying a *kdu-compressed-source* object which advertises the *KDU-SOURCE-CAP-CACHED* capability. Caching data sources may be accessed by the code-stream parsing machinery in any order at all, either in

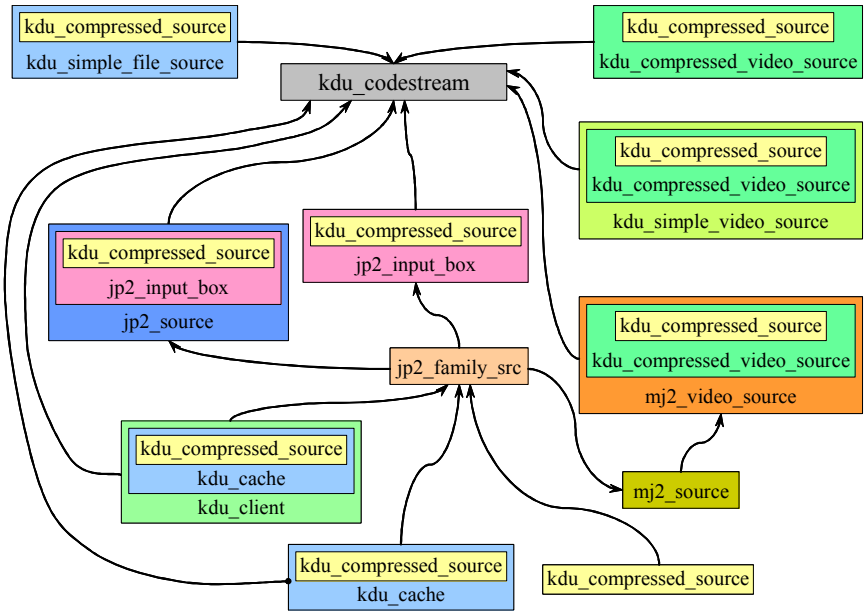

Figure 4: Objects which may be passed as compressed data sources to the input form of *kdu-codestream::create*. [Note: this is an old figure that does not show any of the infrastructure defined to support JPX files.]

part or in full. The contents of a caching data source are parsed on demand by Kakadu's code-stream management machinery, on a precinct-by-precinct[2] basis. Moreover, when the relevant data is required a second time (e.g., by a decompression/rendering engine), the data is reloaded from the cache and reparsed; in this way, the most recent contents of the cache can be utilized where the cache is being incrementally updated by an external agent.

The *kdu-cache* object provides a highly efficient, platform independent implementation of essential and desirable capabilities for typical caching applications. It supports the caching of information from multiple code-streams, as well as the caching of meta-data which may be meaningful to a file format parser. The *kdu-cache* object (or any object derived from it) may be passed directly into *kdu-codestream*'s *create* function, but this allows access only to code-stream data from a single code-stream. Alternatively, the cache may be used to open a *jp2-family-src* object, which is subsequently used to parse boxes in JP2 family file formats. In this capacity, the full power of the cache can be realized. The relevant JP2 family file can be parsed dynamically by any suitable reader, as file format meta-data becomes available in the cache. Whenever the reader opens a JP2 box which represents a code-stream, that *jp2-input-box* object can be passed directly to *kdu-codestream* and the code-stream management machinery will automatically inherit all the properties of the dynamic caching data source, as described above.

The JPIP standard describes a clever mechanism for sending "placeholder" boxes in place of the original boxes from a JPEG2000 family file. The placeholder boxes allow files to be hierarchically restructured by the JPIP server for more efficient incremental delivery of both meta-data and code-stream data. Kakadu's *jp2-input-box* object provides all the machinery to exploit and conceal the existence of placeholder boxes from a file format parser which uses its interfaces, thereby making remotely served files appear exactly as if they were local. All that is needed is to use the server supplied data to dynamically update the *kdu-cache* object (usually this happens in a different thread of execution to that which is actually parsing the file and/or decompressing code-stream data).

To complete the client side of a comprehensive implementation of the features offered by JPIP, Kakadu provides a *kdu-client* object, derived from *kdu-cache*. This object manages all client-side communication required for JPIP client-server interaction, including connection establishment, channel transport negotiation, processing of client-application windows of interest, generation of requests to the server, and processing of asynchronous response data from the server. The kdu-client object is remarkably easy to use, as demonstrated by the *"kdu-show"* example application.

3.2.1 Persistence for Input Objects

In order to minimize memory consumption, the Kakadu system creates resources required to buffer compressed data and maintain its structural integrity as late

[2] Precincts are spatially defined regions within a given resolution level and image component. They provide an excellent basis for random access into a JPEG2000 code-stream.

as possible. By default, these resources are also destroyed (or recycled) as soon as the data has been consumed. As each tile is closed, its resources are cleaned up. Within each tile, as each code-block is closed its resources are also cleaned up. Resources for precincts and other internal entities are also cleaned up as soon as possible and recycled for use in future processing steps.

For some applications, this behaviour is inappropriate. In particular, interactive applications are likely to want to preserve the compressed image representation so that the same region of the image can be decompressed again at a later time, often with a different appearance transformation (different resolution, number of components, spatial view port, or orientation). To support such applications, the compressed data sub-system offers a persistent mode, which may be established by invoking the *kdu-codestream::set-persistent* member function. The *"kdu-show"* application makes extensive use of this capability.

Persistence does not necessarily imply that the entire compressed code-stream must be fully buffered in memory. If the compressed data source is a caching source, data is not buffered in the compressed data sub-system at all; instead, it is reloaded on demand from the caching source. If the compressed data source is seekable and the code-stream contains sufficient information to enable random access, parsed segments of the code-stream are unloaded from memory (and reloaded on demand) as soon as an internal cache threshold is reached. This threshold may be set using the *kdu-codestream::augment-cache* function. Note that the internal caching policy for seekable sources with random access hints is quite independent from the type of caching which is implemented by a caching compressed data source (a *kdu-compressed-source* object which advertises the *KDU-SOURCE-CAP-CACHED* capability).

3.2.2 When Random Access Hints are Unusable

We take the opportunity here to point out that the compressed data sub-system is not always able to exploit random access hints offered by the code-stream. Such hints come in the form of TLM (Tile-part Length, Main header), PLM (Packet Length, Main header) and PLT (Packet Length, Tile-part header) marker segments, all of which are optional in the sense that a valid code-stream can be correctly decompressed without paying any attention to them.

Currently, PLM marker segments are discarded summarily. The reason for this is that it is not possible to interpret packet length information which appears in the main header until the relevant tile headers have been loaded.

The information in TLM and PLT marker segments is clearly useless unless the compressed data source supports seeking (advertises the *KDU-SOURCE-CAP-SEEKABLE* capbility). Additionally, the compressed data sub-system is able to utilize PLT marker segments only under the following conditions, all of which encourage efficient use of the mechanism:

1. All packets of any given precinct must appear consecutively within the code-stream, not broken by packets from other precincts or by tile-part headers. This implies that one of the progression orders RPCL, PCRL

or CPRL must be used and that tile-part boundaries not appear in silly places.

2. POC marker segments must not be used; otherwise, the system has no idea of whether or not the above condition will be satisfied for the entire tile.

3. PLT marker segments must follow any COD or COC marker segments for the tile in which they appear.

3.2.3 Error Handling for Input Objects

Errors may be encountered while parsing a JPEG2000 code-stream for various reasons. Errors may be due to corruption in a lossy communication channel, but they may also arise from incorrect implementation of the compressor or some transcoder (quite a few mistakes are easy to make). To cover a variety of applications, input codestream objects offer three different policies for dealing with these errors. These policies may be set by invoking one or another of the *kdu-codestream::set-fussy*, *kdu-codestream::set-fast* and *kdu-codestream::set-resilient* member functions.

The fussy mode generates terminal error messages in response to many error conditions, going out of its way to check for these conditions. The fussiness extends to both code-stream parsing and also decoding of the embedded code-block bit-streams. For more information on how terminal errors are handled in a manner which should suit most or all applications, see Section 7.

The fast mode generates terminal error messages in response to many of the same error conditions which are checked by the fussy mode, but deliberately does not implement some time consuming error checks. This is the default mode.

The resilient mode attempts to recover from errors encountered while parsing the code-stream and attempts to conceal errors detected while decoding the embedded code-block bit-streams. Terminal errors conditions should not arise in this mode unless the code-stream contains multiple tiles or tile-parts or an error occurs in the main header[3].

3.3 Code-Stream Output

To generate a JPEG2000 code-stream, either from scratch or by transcoding from an existing one, the output form of the *kdu-codestream* object's *create* function should be used. This form accepts a single *siz-params* object identifying the image dimensions, and an object derived from the abstract base class, *kdu-compressed-target*. Compressed data targets may include anything from simple file writers, to custom file format creators and even special network

[3] Under these conditions, errors can create ambiguous states from which robust recovery is very difficult. As a rule, the resilient mode is only intended for applications where the code-stream may have been corrupted in transit. Sources targeting such environments should avoid tiling and tile-parts if possible.

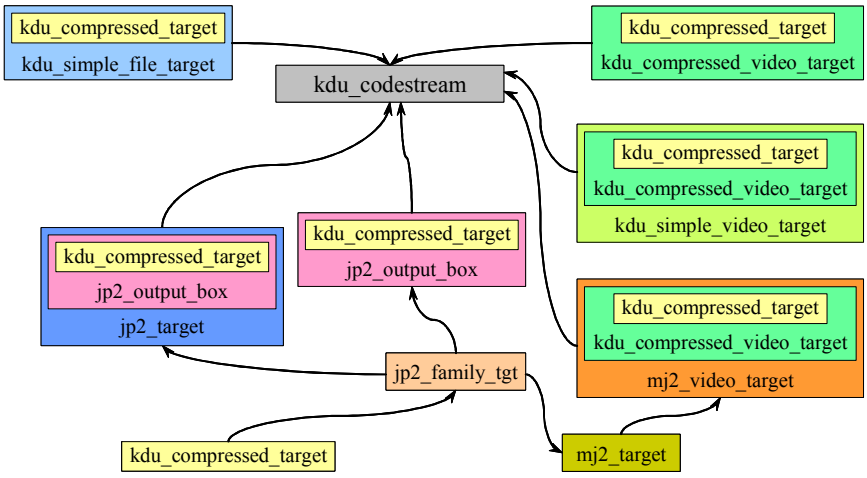

Figure 5: Objects which may be passed as compressed data targets to the output form of *kdu-codestream::create*.

management systems. The Kakadu software tools include a variety of derived compressed target classes, as illustrated in Figure 5. The architecture for compressed targets is similar to that for compressed sources. For more information, the reader is referred to the detailed interface descriptions or the various example applications *"simple-example-c"*, *"kdu-buffered-compress"* and *"kdu-compress"*, in that order.

3.3.1 Rate Control

The EBCOT paradigm (Embedded Block Coding with Optimized Truncation) adopted by JPEG2000 allows one or more target compressed bit-rates to be achieved without compressing the image multiple times, as in JPEG. As image samples are processed, embedded code-block bit-streams are generated and passed to the compressed data sub-system via the *kdu-block* object. Once all samples have been encoded, the *kdu-codestream::flush* member function must be called to generate the final code-stream. This function accepts an array of target compressed sizes (measured in bytes) for each of a number of quality layers. It is also possible to directly control or receive feedback concerning the rate-distortion slope thresholds associated with each quality layer.

Although not strictly required, the array of target sizes should contain one entry for each quality layer. The actual number of quality layers which you choose to build into the code-stream is set up through the coding parameter sub-system (see the *"Clayers"* attribute of the *cod-params* parameter class). To obtain a distortion scalable code-stream (quality/SNR progressive unless you specify a different progression order for the information), you should use more than one quality layer (the default). A typical distortion scalable code-stream

might have anywhere from 6 to 60 layers.

The *kdu-codestream::flush* implements a useful heuristic for filling in missing entries (entries whose value is 0) in the array of quality layer sizes which it receives. In fact, every entry in the array may be 0, in which case a reasonable layering policy is selected for you. For more information, consult the extensive description of this function in the *"kdu-compressed.h"* header file or (preferrably) via the integrated documentation system.

3.3.2 Incremental Rate Control

The EBCOT rate control paradigm may be implemented incrementally as the compressed code-block bit-streams become available. In particular, it is possible to make judgements concerning which coding passes (pieces of the embedded bit-streams) are likely to be discarded during rate control and to discard them earlier. In this way, the total amount of memory required to buffer the compressed data prior to rate control need not substantially exceed the maximum target compressed size. This functionality may be enabled by invoking the *kdu-codestream::set-max-bytes* member function, although close attention should be paid to the advice offered in the description of that function in *"kdu-compressed.h"*.

In addition to reducing memory consumption, incremental rate prediction is used to terminate the block coding process for any given code-block, once it is considered to have produced all the compressed bits which are likely to survive rate control. This can result in substantial processing speedup during compression. Again, this capability is enabled through the *kdu-codestream::set-max-bytes* member function and the advice offered in its description should be carefully heeded.

An alternative and even more efficient means of limiting the amount of data which must be encoded and stored prior to flushing the code-stream is to specify the minimum rate-distortion slope threshold directly via *kdu-codestream::set-min-slope-threshold*. This approach is particularly attractive when slope thresholds are determined in a rate control feedback loop (e.g., for video compression) or a collection of images is to be compressed with similar quality (rather than similar bit-rate).

3.3.3 Incremental Flushing

Kakadu offers substantial support for incrementally flushing the code-stream. To use this feature, you need simply call *kdu-codestream::flush* whenever you would like the code-stream management machinery to generate the next portion of the code-stream; however, before plunging in, you should understand some of the constraints associated with code-stream generation.

There is no longer any need to wait until the entire image has been processed before calling *kdu-codestream::flush*. Each time you call this function, it will write as much as possible of the compressed code-stream, given the amount of the image which has been processed, and the constraints associated with the

order in which information must appear within the code-stream. This avoids the need to store the entire compressed image in memory, which could render compression of huge images infeasible.

The constraints associated with incremental flushing are imposed by the packet progression sequence which is in force for each tile in the image. By and large, if any tile uses a layer or resolution dominant packet progression, little or not code-stream data can be written for that tile until all relevant image samples have been compressed. Consequently, incremental flushing is beneficial only if the image has been tiled, or if an untiled image uses a spatially progressive packet sequence. In the latter case, you must also be careful in selecting the precinct dimensions, since spatially progressive sequencing is useful only with small precincts. In particular, you should endeavour to select precincts with roughly hierarchically descending dimensions, at least in the vertical direction. For examples of this, see the "*Usage-Examples.txt*" file or the extensive documentation accompanying the *kdu-codestream::flush* function.

The PCRL (precinct-component-resolution-layer) sequence is generally best for most incremental flushing applications, since it allows incremental flushing while processing the image components in an interleaved fashion from top to bottom. The interleaved processing of image components ensures that incremental rate control mechanisms will be exposed to the diversity of statistics from the different image components (luminance components, for example, generally have very different statistics to chrominance). This is the policy followed by the "*kdu-compress*" demo application. You may care to read the usage statement printed by "*kdu-compress*" in connection with its *-flush-period* option. It is also instructive to look at the way in which this application calls *kdu-codestream::flush* to implement incremental flushing – it is not complicated.

You should be aware of the fact that each time the *kdu-codestream::flush* function processes an active tile, a new tile-part will be created for that tile. JPEG2000 Part 1 imposes a limit of 255 tile-parts per tile, so you should try to flush code-stream data as infrequently as possible. To help you avoid needless calls to the *flush* function, it is highly recommended that you call *kdu-codestream::ready-for-flush* first.

If you build your compression application on top of the high-level *kdu-stripe-compressor* object, incremental flushing can be achieved with negligible effort, simply by providing a non-zero value for the *flush-period* argument to *kdu-stripe-compressor::push-stripe*.

As a final note, we point out that incremental flushing supports the specification of quality layers in terms of their final size (target sizes or bit-rates) or in terms of their distortion-length slope thresholds. In fact, the latter approach may be preferred if frequent flushing is required. We also point out that if the *ORGgen-plt* attribute is set to *yes*, PLT (packet length) marker segments will be entered into every tile-part generated, whether by incremental flushing or otherwise, allowing efficient random access into the code-stream by viewers and servers such as "*kdu-show*" and "*kdu-server*".

3.3.4 Fragmented Compression

Kakadu allows you to compress a tiled image in pieces, while preserving all the functionality of one-shot compression, so long as the image is fragmented on tile boundaries. In fact, with careful synchronization at the compressed data target, it should be possible to use this feature to compress multiple fragments concurrently, in a multi-threaded application (e.g., to exploit the availability of multiple physical processors). To understand fragmented compression, consult the interface description for the first form of the *kdu-codestream::create* function and/or see how it is implemented in the *"kdu-compress"* application.

3.4 Code-Stream Interchange

A third form of the *kdu-codestream::create* accepts neither a compressed data source, nor a compressed data target. We refer to the objects created using this function as "interchange" objects. In many ways, interchange objects are best interpreted as output objects. Code-block data must be written to the object and this may be obtained by processing wavelet coefficients or by transcoding another code-stream (this is the principle source envisaged). However, instead of constructing a sequential JPEG2000 code-stream, the role of an interchange object is to allow applications to directly construct JPEG2000 packets in any order whatsoever. This is particularly important for server applications, enabling them to deliver compressed JPEG2000 packets to clients in the order which best matches the client's current spatial region and resolution of interest. The *kdu-serve* object, on top of which the *"kdu-server"* application is built, makes extensive use of *kdu-codestream* objects created for interchange; this allows the server to transcode local code-streams on the fly, so as to make them more amenable to efficient and responsive interactive delivery over low bandwidth networks.

4 Image Components and Multi-Component Transforms

This section is devoted to the discussion of image components, which are somewhat complicated by the introduction of Part-2 multi-component transform technology. We must now identify two distinct types of image components, for which we use the terms *"codestream image component"* and *"output image component"* consistently throughout the Kakadu system. The relationship between codestream and output image components is depicted in Figure 6. The codestream image components are the ones which are subjected to spatial wavelet transformation (DWT) and block coding processes. Equivalently, during decompression, the code-stream is decoded into spatial wavelet coefficients which are subsequently subjected to inverse DWT transformation, yielding the codestream image components. Each codestream image component is thus an independently compressed grey-scale image.

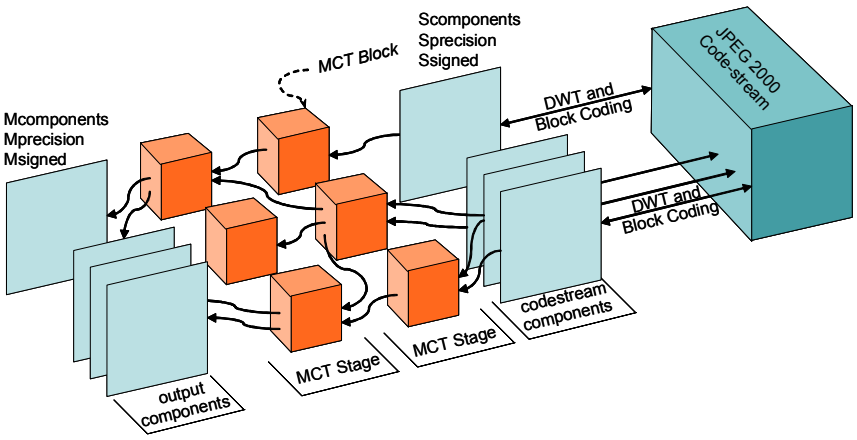

Figure 6: Relationship between *codestream image components* and *output image componetns*.

The term *output image component* comes from taking the perspective of the decompressor. After the codestream image components have been reconstructed, they may be subjected to one or more stages of multi-component transformation. Each multi-component transformation stage may itself consist of one or more transform blocks. Each transform block takes its inputs from image components produced by the previous stage (or from the codestream components, in the case of the first stage). Each transform block may implement a variety of inter-component transformations, which are classified as follows:

- Null transform blocks – these are used to reorder components, create new empty components, or discard components which you don't wish to pass through to the output.
- Irreversible matrix decorrelation transforms – here, corresponding points from each input component to the transform are treated as a vector which is multiplied by an $M \times N$ matrix, with M the number of output components and N the number of input components to the block.
- Reversible matrix decorrelation transforms – here, a sequence of reversible combinatorial steps progressively transforms the N block input components into N output components with integer values. These transforms may be exactly inverted.
- Reversible and irreversible dependency transforms – these involve the prediction of successive image components based upon previous input image components, in a recursive fashion.
- Reversible and irreversible wavelet transformations – here, corresponding points from each of the block input components are treated as subband

samples from a DWT with a specified depth and canvas origin (in the component direction). Arbitrary wavelet kernels are supported.

Needless to say, multi-component transforms can become highly complex. For a thorough discussion of the possible transforms, it is best to refer to Part-2 of the JPEG2000 itself (IS15444-2). However, you may well be able to glean enough information to create your own useful multi-component transforms by doing the following:

1. Read the interface descriptions for functions *kdu-tile::get-mct-block-info*, *kdu-tile::get-mct-matrix-info*, *kdu-tile::get-mct-rxform-info*, *kdu-tile::get-mct-dependency-info* and *kdu-tile::get-mct-dwt-info*.
2. Go through the examples labeled (Af), (Ag), (Ah), (Ai) and (Ak), in the *"Usage-Examples.txt"* file; these examples demonstrate a wide variety of useful transform networks and contain a significant number of helpful explanatory notes.
3. Read the usage statements for all the parameter attributes whose name commences with the letter *"M"*, as printed by *"kdu-compress"* with the *"-usage"* option.

If you do not wish to create your own multi-component transforms, you may still wish to be able to decompress and render images which do use the multi-component transform features from JPEG2000 Part-2. For this, you will find that the *kdu-multi-synthesis* object does everything you need already. However, there is one significant point that you should take on board. Specifically,

> *The number of output components may differ from the number of codestream image components. Moreover, the bit-depths and signed/unsigned properties of output image components need not correspond to those of the codestream image components.*

For this reason, Kakadu's *siz-params* object manages parameter attributes with the names *Mcomponents*, *Mprecision* and *Msigned*, in addition to the *Scomponents*, *Sprecision* and *Ssigned* attributes. The former describe output image components, while the latter describe codestream image components. In the event that no Part-2 multi-component transform is in use, the *Mcomponents*, *Mprecision* and *Msigned* attributes will not be present, since then the codestream and output image components are in one-to-one correspondence.

4.1 Image Components and the *kdu-codestream* Interface

The presence of two different types of image components means that funtions provided by *kdu-codestream* and *kdu-tile* must be careful to distinguish between the type of component for which information is being requested. This is done through an optional final argument, *want-output-comps*, which appears at the end of the argument list of every function which is used to request component-specific information. Examples of these are

- *kdu-codestream::get-num-components* – returns either the number of codestream image components or the number of output image components, currently visible to the application. Note that the visible components are affected by calls to *kdu-codestream::apply-input-restrictions*.
- *kdu-codestream::get-dims* – returns the size and location on the canvas of a visible codestream image component or a visible output image component.
- *kdu-codestream::get-tile-dims* – returns the size and location of a visible codestream or output image component, as it appears within a tile of interest.
- and many others.

To ensure backward compatibility with all applications built upon early versions of Kakadu (prior to the introduction of support for Part-2 multi-component transforms), the default value of the optional *want-output-comps* argument has to be false, meaning that if you are not careful, you will retrieve the number, dimensions, precision, etc., of the codestream image components. For most applications, however, it is the output image components that you are likely to be interested in when querying the *kdu-codestream* interface. Thus, for full compatibility with Part-1 and Part-2 codestreams, you will have to take some care.

The *kdu-codestream::apply-input-restrictions* function plays a central role in determining how image components appear to the application. This function now has two versions, where the second version allows components to be arbitrarily remapped, leaving an arbitrary set of components visible to the application. Both functions also allow you to render all multi-component transforms invisible. In particular, the *kdu-codestream* machinery maintains the notion of a current "component access mode". This may be set to one of *KDU-WANT-OUTPUT-COMPONENTS* or *KDU-WANT-CODESTREAM-COMPONENTS*. In the second case, multi-component transforms appear not to be present, so you will only ever see codestream image components. Equivalently, the output image components will appear to be identical to the codestream image components, whenever you access component-specific information via the *kdu-codestream* or *kdu-tile* interfaces. In the case of the *KDU-WANT-OUTPUT-COMPONENTS* mode, however, the difference between output and codestream image components will appear to be apparent (if there is one). Primarily, this is the mode of interest, and so this is the default.

For a thorough discussion of these issues, you are referred to the interface descriptions for the *kdu-codestream::apply-input-restrictions* functions.

4.2 Don't Stress!

At this point, you may be starting to become concerned that this matter of codestream and output image components will be very confusing and make it hard for you to work with Kakadu. This should not be a real concern, firstly

since the distinction is only important if you wish to operate successfully with Part-2 codestreams and there are few implementations of the JPEG2000 standard which can even produce them. Secondly, so long as you stick to working with the high level objects, such as kdu-multi-analysis, kdu-multi-synthesis, kdu-region-decompressor, kdu-region-compositor, and so forth, you can remain largely ignorant of the fine points of codestream vs. output image components. The only point at which you should be conscious of the distinction is if you wish to directly access image component information, or if you wish to directly change the codestream appearance via a call to *kdu-codestream::apply-input-restrictions*. The high level *kdu-region-compositor* object, for example, makes all these calls for you, so you shouldn't ever need to think about it.

5 Data Processing

5.1 Sample Value Representations

JPEG2000 defines two quite different processing paths. The reversible processing path works with integers, while the irreversible path works with notionally floating point quantities. Accordingly, sample values are represented in Kakadu either as signed integer values or as normalized real-valued quantities with a nominal range of $-\frac{1}{2}$ to $\frac{1}{2}$. In the case of the reversible path, the dynamic range of the integers is determined by the sample bit-depth, which is returned by various convenient interface functions.

In both cases, the nominal dynamic range may be exceeded during decompression as a result of quantization errors. None of the core system services clips the data back to its nominal dynamic range. This is deliberate, since clipping is application dependent and should generally be deferred until the point at which the samples must be stored to disk or rendered.

The Kakadu data processing objects provide support for both 16-bit and 32-bit precision data representations. 32-bit irreversibly processed quantities use a single precision floating representation, while 16-bit irreversibly processed quantities use a fixed point representation having 13 fraction bits and 3 integer bits (including the sign bit). Internal machinery directly calculates the worst case dynamic range expansion properties of the various numerical transformations and uses this information to make optimal use of the limited precision representations.

Although the application developer is free to select either 16 or 32 bit representations, Kakadu provides a service for determining the most appropriate selection. The reader is referred to the *kdu-tile-comp::get-bit-depth* member function and its description in *"kdu-compressed.h"* for more information regarding this service.

Importantly, however, from v5.0 you no longer need to concern yourself with determining the most appropriate data representation precision, so long as you are using the *kdu-multi-analysis* or *kdu-multi-synthesis* objects. These objects can be forced to use *32*-bit representations, but otherwise make the

most appropriate decision for you, based upon the transforms involved. These objects allocate line buffers for you, so all you have to do is be prepared to write or read data to the line buffers using their declared precision.

It is worth pointing out that the original sample bit-depth and unsigned/signed properties recorded in the code-stream's *SIZ* marker segment (set through appropriate calls into the coding parameter sub-system, as demonstrated in the simple examples) has relatively little to do with the representation selected for processing data in Kakadu. The JPEG2000 standard specifies that all data be processed as signed quantities, so that original sample values which were unsigned must be level shifted into a signed representation for processing. Kakadu regards level shifting as the responsibility of the application, which has a number of advantages. For example, most rendering applications will want to level shift the decompressed sample values regardless of whether the original data were signed or unsigned, since the best way to display signed quantities is generally to level shift them. Thus, building level shifting of originally unsigned data into the core Kakadu processing system would actually complicate the application.

For data compressed in the irreversible path, the bit-depth information recorded in the *SIZ* marker segment plays a purely informative role in JPEG2000, having no impact on processing or appropriate data representations whatsoever. This is because the irreversible path involves notional floating point quantities, with unbounded precision, regardless of the original sample bit-depth. The JPEG2000 algorithm is essentially bit-depth blind when operating in the irreversible processing path and most applications will not need to know what the original bit-depth was. The same is not true for the reversible path, where the bit-depth information is required to determine where the most significant bits of the decompressed integer sample values are so that they can be appropriately rendered.

5.2 Component Blending and Channels

As mentioned in Section 2.1 and illustrated in Figure 1, the Kakadu software framework provides services for processing the sample values within any given tile-component. In most cases, a high level processing engine is constructed from the derived *kdu-encoder*, *kdu-analysis*, *kdu-decoder* or *kdu-synthesis* classes, and the abstract base class's *push* and *pull* interfaces are then used to exchange tile-component lines between the application and the system. These interfaces work with the *kdu-line-buf* class. Ever since version 5.0, additional high level objects, *kdu-multi-analysis* and *kdu-multi-synthesis* have been provided to accommodate inter-component transformations which are defined at the code-stream level. Inter-component transforms include the simple RGB to YCC conversions associated with the JPEG2000 Part-1 ICT (Irreversible Colour Transform) and RCT (Reversible Colour Transform). However, they also include the vastly more flexible Multi-Component Transform features from JPEG2000 Part-2. These objects operate on whole tiles. As might be expected, their behaviour depends on whether multi-component transformations are currently

apparent, based upon the component access mode supplied in any previous calls to *kdu-codestream::apply-input-restrictions*.

In the simplest case, the component samples input to *kdu-multi-analysis* or recovered from *kdu-multi-synthesis* correspond to actual colour channels: e.g., red, green and blue channels. More generally, however, some combination of colour transformation and/or interpolation operations may be required to convert image components into a true colour image which can be displayed or printed. These depend upon information which is embedded in a wrapping file format, such as JP2, JPX or MJ2. In each case, the colour representation itself is obtained via a common *jp2-colour* interface, while the particular components which contribute to the colour representation in question are identified via a *jp2-channels* interface.

Kakadu provides various tools to simplify these tasks. In particular, *kdu-region-decompressor* combines the information found in file format colour descriptions, with the component sub-sampling and registration information found in the relevant code-stream, to correctly render RGB imagery. It also recovers alpha channel information where appropriate and provides more generic image rendering services, including support for rendering into a wide variety of buffer organizations with a variety of output sample bit-depths.

The much higher level *kdu-region-compositor* object builds (a great deal) on the functionality offered by *kdu-region-decompressor*, providing a complete set of services for managing an interactive imaging application, including multiple image composition, customizable metadata-driven overlays, buffer management, scaling, reorientation, and animation.

Although these functionalities are wrapped up for you, the next sub-section provides some useful background information to the way in which colour and channels are represented in the JP2 family of file formats.

5.2.1 Colour Channels in JP2

One of the most important responsibilities of any file format designed to embed a JPEG2000 code-stream is to provide an interpretation of the image component samples. The JP2, JPX and MJ2 file formats provides such an interpretation, which is best understood in two parts. The first part of the description identifies a mapping between image components and colour channels or other reproduction functions. The second part of the description identifies the colour space in a manner which permits appropriate rendering by a range of different applications.

As mentioned, the first part of the description involves a mapping between the image component samples and colour channels, or other reproduction functions. The image components which are referenced by these channel mapping rules are actually the *output image components* described in Section 4, as distinct from *codestream image components*. This distinction is important only in the case where Part-2 multi-component transforms have been used, which is currently allowed only within JPX files; in all other cases, there is no distinction between the two types of image components. Nevertheless, it is best that you are aware of the distinction. There are up to 9 different image reproduction

functions defined, which may be classified as follows:

Colour Only three colour channels, R (red), G (green) and B (blue), are supported by the JP2 file format. Alternatively, a luminance image may have exactly one colour channel, Y (luminance). The colour channels must be mapped to the code-stream. The JPX file format supports an arbitrary number of colour channels, however, so that colour spaces such as CMYK may be described.

Opacity Exactly one opacity channel is defined for each colour channel. Any or all of the opacity channels may be unmapped, meaning that the relevant colour data is to be rendered directly without multiplication by an opacity mask. The opacity information can be ignored by applications which do not need to blend the image into a background or another image.

Pre-Multiplied Opacity Exactly one pre-multiplied opacity channel is defined for each colour channel. Any or all of the pre-multiplied opacity channels may be unmapped, meaning that the relevant component has not been pre-multiplied by an opacity mask.

Each of these reproduction functions which is mapped to the code-stream (certainly all of the colour channels) is associated with exactly one image component and possibly a palette transformation. When a palette transformation is used, the image component's samples are interpreted as the indices of a lookup table. Each mapped reproduction function may use any of the available image components and any of the available palette lookup tables, with the following restrictions (these are imposed by the standard).

1. Image components which are used as indices of a palette lookup table must only be used in this way.

2. No image component which is used for one class of reproduction function (colour, opacity or pre-multiplied opacity) may be used for any of the other classes of reproduction function, except if it is used to index a palette lookup table. In this case, the same combination of source image component and palette lookup table must not be used for different classes of reproduction function (colour, opacity or pre-multiplied opacity). There is a way of working around this reasonable constraint without violating the specifications of the standard, but this was not the intent, so Kakadu explicitly prohibits writers from doing it.

Kakadu simplifies the task of working through the tangled interactions described by the JP2 file format, providing a clean set of interfaces through its *jp2-channels* and *jp2-palette* objects. Since illegal files might be encountered, which do not conform exactly to the above constraints, the Kakadu JP2 reader does not insist that the constraints be satisfied and applications should also try to avoid relying on them.

The second part of the colour reproduction description provided by the JP2, JPX and MJ2 file formats involves the specification of a colour space which is to be associated with the colour channels (one channel if the image carries luminance only). Kakadu facilitates interaction with these colour descriptions through the *jp2-colour* object. This object allows file writers to construct a variety of colour descriptions, including embedded ICC profiles. It allows file readers to recover the descriptions. A separate *jp2-colour-converter* object provides the services required to efficiently convert the colour channel data to the sRGB colour space. These capabilities are of great assistance in the construction of conforming JP2, JPX and MJ2 rendering applications, such as *"kdu-show"*.

The above functionality is extended and generalized to fully support JPX files, in which channels may be derived from the output image components associated with different codestreams within the file (or even codestreams found in other files). JPX also allows for a much richer set of colour representations and associated colour conversion capabilities. These features are all fully implemented by Kakadu.

Thorny Issues

The JPEG2000 standard insists that the Part-2 codestream colour transforms (ICT or RCT) should only be used in connection with RGB colour data. In particular, they should not be used to compress image components which are then to be used as palette indices, or which already represent YCbCr data. Kakadu does not explicitly impose this restriction.

It is possible to specify the use of a luminance-chrominance representation in two different ways using JP2. One way to do this is to have the JP2 file identify an RGB colour space and the code-stream specify the use of one of the colour transforms, ICT or RCT. Alternatively, the JP2 file may identify the code-stream image components as representing luminance-chrominance data. In the latter case, the *jp2-colour::get-space* returns *JP2-sYCC-SPACE*. The same thing may be done in JPX and MJ2 files.

These two different ways of specifying luminance-chrominance compressed imagery have different implications. Specifying a luminance-chrominance space in the JP2 colour box means that the image's colour channels are to be interpreted as luminance and chrominance samples. If writing the decompressed output to a file, this representation should probably be preserved, since it is understood as the original image representation and has a different gamut to that obtained after conversion to RGB. By contrast, the code-stream's colour transform is a mandatory part of the decompression pipeline. Of course, when rendering to a display, the data should always be converted to an appropriate RGB format. This is handled automatically by the *jp2-colour-converter* object and the higher level *kdu-region-decompressor* and *kdu-region-compositor* objects which use it.

5.2.2 Interpolation and Registration

Whereas the ICT and RCT transforms defined by Part-1 code-streams, as well as the more general multi-component transforms defined by Part-2, al-

ways operate on image components with identical dimensions, JP2, JPX and MJ2 files describes colour channels which may have different sub-sampling factors. This means that conversion to an appropriate rendering space must be preceded by interpolation to a common grid. It is also important to correctly register the channels against one another on the interpolation grid. The *kdu-codestream::get-registration* function provides convenient access to the relevant registration information, while *kdu-codestream::get-subsampling* or *kdu-tile-comp::get-subsampling* may be used to recover sub-sampling information. Together, these two pieces of information are sufficient to correctly implement the relevant interpolation operations.

Since interpolation, registration and the other channel mapping and rendering operations can become rather complex, developers may find the *kdu-region-decompressor* object particularly helpful. This powerful object encompasses all of the functionality required to robustly decompress an arbitrary image region, applying all necessary channel mapping, interpolation and colour transformation operations. The implementation is platform independent, although there are some platform dependent speedups implemented for specific processor families. The *kdu-region-decompressor* object correctly and transparently manages the complexities associated with the presence of tiles, strange registration offsets and weird sub-sampling factors. The *kdu-region-decompressor* object supports practically any application-defined memory-buffer structure, both 8- and 16-bit buffer sample precisions, decompression of additional channels representing opacity, and so forth.

The *kdu-region-compositor* object carries the above level of support several steps further, wrapping up all the intelligence required to composit multiple images onto a composition canvas, taking into account whatever cropping, scaling and re-registration parameters might be specified by a JPX file's composition box.

5.3 Flow Control Considerations

The JPEG2000 code-stream describes images as a collection of tile-components, which may be compressed or decompressed independently. Thus, in principle there is no reason why an implementation cannot process tile-components one at a time, using the capabilities offered by Kakadu. While this is simple, there are many reasons why it is inadvisable. Firstly, whenever colour transformation is required, the information from multiple image components must be brought together. This is usually best done prior to rounding and clipping data to fit into say an 8-bit per sample buffer and so memory efficiency and locality dictates that most applications should attempt to process image components together, rather than sequentially.

Secondly, compressors which choose to process image components one after another will not be able to take advantage of the incremental rate prediction capabilities offered by Kakadu to skip coding passes whose contents are likely to be discarded by the rate control system. This prediction system relies upon the incremental estimation of image statistics, which is not possible

if the image components are processed one by one. For more information on this, consult the description of the *kdu-codestream::set-max-bytes* function in *"kdu-compressed.h"*.

Kakadu provides several "off-the-shelf" approaches for concurrently processing image component samples. The *kdu-stripe-compressor* and *kdu-stripe-decompressor* objects usually provide the most efficient interfaces to Kakadu for applications that compress or decompress whole images, allowing you to push imagery in (or pull it out) in stripes that can be as small as one image row at a time or as large as the entire image. Internally, the *kdu-stripe-compressor* and *kdu-stripe-decompressor* interfaces open and close tile interfaces in a manner that is as efficient as possible, based on the geometry of the stripe data that is being pushed into or pulled out of the high level interfaces.

The *kdu-region-decompressor* object is designed with image rendering applications in mind. In this case, the rendering device manages a view port into the image and that view port has a random access buffer large enough to accommodate decompressed colour data for the viewing port. The *kdu-region-decompressor* object processes the relevant tiles (if the image is tiled) one by one, writing them into the relevant region of the buffer.

6 ROI Processing

JPEG2000 Part 1 provides two mechanisms for assigning higher priority to a region of interest (ROI) and encode time. One mechanism is known as the "max-shift" method. In this case, subband samples which are involved in the reconstruction of the region of interest are scaled up prior to quantization. The decompressor undoes this scaling, so that the net effect is equivalent to reducing the quantization step size used for these higher priority subband samples. The second method is to adjust the cost function which drives the PCRD-opt rate allocation algorithm, so that code-blocks whose subband samples contribute to the region of interest are assigned a higher priority.

ROI encoded code-streams usually have many quality layers. The initial quality layers contain information only from the region of interest, whose quality improves rapidly as the number of layers used in the reconstruction increases. At some point, the quality of the background region (the rest of the image) begins to increase. For losslessly encoded images, the region of interest (foreground) becomes lossless first, but once all layers have become available for reconstruction, the entire image can be reconstructed losslessly.

Both methods for ROI encoding which are mentioned above are able to support the ROI quality progressions described above, but they each have their own advantages and disadvantages. The chief disadvantage of the "max-shift" method is that the scaling factor, which determines the amount by which the foreground quantization step sizes are reduced relative to the background, must be very large. In fact, it must be so large that the foreground and background regions can be distinguished by the decoder, simply by inspecting their subband sample magnitudes. Typical minimum scale factors are in the thousands.

You scale factor is actually expressed in terms of a shift via the coding parameter attribute, *Rshift*. As an example, reversible compression of 8-bit colour images requires a shift value of at least 12. The corresponding scale factor is $2^{12} = 4096$. The problem with such large scale factors is that the foreground improves all the way to lossless (or the maximum possible quality if compression is irreversible) before any information is received for the background. A second problem with the "max-shift" method is that the block decoder implementation must be capable of decoding a large number of magnitude bit-planes. As a result, decoder implementations conforming to Compliance Class-0 may not recover any information at all for the background.

These problems may both be avoided by using the second approach described above, in which the cost function used to allocate code-block contributions to quality layers is modified in accordance with the region of interest. This allows a user selectable scaling factor (supplied through the coding parameter attribute, *Rweight*) and hence gives the user full control over the transition between foreground and background quality improvements. On the other hand, the region definition is significantly poorer with this method, since adjustments can only be made on a code-block by code-block basis. When using this method, you are recommended to use smaller code-blocks; in particular, set the *Cblk* coding parameter attribute to *Cblk={32,32}*.

Kakadu supports both of the ROI encoding methods described above. For details, we suggest that you review the *"kdu-compress"* examples appearing in the file, *"Usage-Examples.txt"*. In the remainder of this section, we direct the reader's attention to the services offered by Kakadu for constructing region processing engines. The core Kakadu system provides low level ROI services through the interfaces defined in "*core/common/kdu-roi-processing.h*". These services are designed to run in tandem with the sample data processing engines, converting image level descriptions of the region of interest into information concerning the foreground/background identify of every subband sample. The system supports arbitrary region geometries and performs the region mapping steps incrementally, using as little memory as possible. To take advantage of the memory efficiency it is generally preferred that you employ good flow control principles when compressing the image, as explained in Section 5.3.

Applications should supply an appropriate derivation of the abstract base class, *kdu-roi-image*, whose chief role is to hand out suitably derived versions of the abstract base class, *kdu-roi-node*, which manage ROI information at the image level. The "*kdu-roi-sources.h*" and "*kdu-roi-sources.cpp*" files in the "*apps/kdu-compress*" directory provide two useful examples of such derived classes. The derivation represented by "*kdu-roi-rect*" is a particularly simple example, which manages only a single rectangular foreground region. This is interesting mainly for didactic purposes. The much more sophisticated "*kdu-roi-graphics*" derivation sources arbitrary region shapes from an auxiliary graphics file. It automatically scales the auxiliary image to match the dimensions of each image component for which ROI region processing is requested. Application developers may find that this second example is a good starting point for developing interactive region specification tools (e.g., tools for radiologists to mark

up patient imagery with regions of interest).

7 Custom Messaging Services

Kakadu aims to retain complete platform independence, at least in the core system. One potential complication in this regard is the delivery of error and warning messages. Some platforms may support popup windows; others may support only terminal I/O; while others might only support message logging. A similar diversity exists amongst different types of applications on any given platform. For some applications, it may be appropriate to issue an error message on a terminal's standard error stream and exit the program in response to a fatal error. In other cases, error conditions might expected to produce interactive messages in popup windows and the containing program should continue to execute.

Kakadu supports this diversity of scenarios by providing customizable error and warning message services. All messages generated from within the core system are delivered via local instances of one of the classes *kdu-error* or *kdu-warning*. Applications may use the same handy services. The following code fragment illustrates the use of these services:

```
if (error)
    { kdu-error e; e <<"An error has occurred!"; }
```

The messaging services have two important properties: 1) they format the text to introduce line breaks and keep track of paragraph indentation; and 2) they do not rely upon the existence of a particular delivery mechanism for the message text. By default, the messages will disappear into thin air. For this reason, you should generally invoke the core functions, *kdu-customize-errors* and *kdu-customize-warnings*. These functions allow the application to supply an appropriate handler for the message and to specify the formatting which is to be applied to it. In many cases, these handlers can be very simple, since most of the work is done by the core system. Custom handlers may also be supplied to throw exceptions or provide alternate exit processing when a *kdu-error* object goes out of scope. For more information regarding these capabilities, consult the detailed descriptions in *"kdu-messaging.h"*.

The core system only uses these messaging services in "safe contexts," by which we mean that it is possible to throw an exception from the message handler and then safely close down the system from the exception handler. This is particularly interesting for interactive applications, such as that demonstrated by *"kdu-show"*. One consequence of this safe error handling policy is that the core system steers away from generating errors within the constructor of any object which consumes resources. This is why many key objects have their key initialization steps performed in a post-construction function call, usually with some name like *create* or *init*. As a general rule, constructors are the least

robust place to perform substantial initialization in C++. Java does not have this weakness, of course.

For the purpose of debugging, you may find it very useful to provide a custom error message handler into which a breakpoint can be inserted. Alternatively, you may choose to insert breakpoints directly into the constructor or destructor function of the *kdu-error* class; these are implemented in *"core/messaging.cpp"*.

7.1 Internationalization

Kakadu provides all the support required to internationalize and/or customize all error/warning messages produced by the Kakadu system or derivative applications. This is done in an almost seamless manner, which remains backward compatible with the simple uses of *kdu-error* and *"kdu-warning"* described above; moreover, the internationalization strategy is entirely platform independent. Original error/warning text remains in the source files where they are easiest to follow and edit. However, alternate constructors are provided for the *kdu-error* and *kdu-warning* objects, which allow text to be registered with unique identifiers. This is done using macros, so that very little editing of the simpler, direct error/warning calls is required to prepare them for internationalization. The same macros are used by a new tool, *"kdu-text-extractor"*, to collect all the registerable text into separate language files.

You can create as many versions of these as you like, translating text into new languages (and even using unicode for languages with large alphabets). To use any of these external language files, you compile the original code (core system, applications, etc.) with the *KDU-CUSTOM-TEXT* macro defined. You then simply include the language files of interest into your end application and the translation process is complete.

If you like, you can construct separate language-specific DLL's (Windows) or shared librares (Unix) containing the language files. Your application then just needs to load the language DLL or shared library of interest at run time.

By default, *KDU-CUSTOM-TEXT* is not defined, and everything behaves exactly as it did in earlier versions of Kakadu. In this case, there is no need (indeed no point) including the language files. For more information, see the *"Compilation-Instructions.txt"* file.