GIL 2.0 Changes

This document outlines the more significant changes in GIL introduced in version 2.0

Overall changes:

Added `deprecated.hpp` - a file that maps many of the deprecated names to current ones. Including it will help porting your code to GIL 2.0. After porting to GIL 2.0, however, make sure that your code works when this file is not included.

File structure:

- Directories follow the boost convention
- Removed (flattened) the 'core' directory
- Different models are usually now split in separate files
- Renamed some files to be more consistent
- Renamed classes, functions and template arguments with longer but clearer and more consistent names.
- Now compiles with GCC 4.1.1.

Changes in `image`, `image_view`, `any_image` and `any_image_view`: There are no longer global functions `get_width(), get_height(), get_dimensions(), num_channels()`. Use class methods `width(), height(), dimensions()` instead.

Changes in models of pixel, pixel iterator, pixel locator, image view and image:

There used to be different ways of getting to a pixel, channel, color space, etc. of an image view, pixel, locator, iterator and image. Some classes were using traits, other - member typedefs. In GIL 2.0 all pixel-based GIL constructs (pixels, pixel iterators, locators, image views and images) model `PixelBasedConcept`, which means they provide the following metafunctions:

```
color_space_type
channel_mapping_type
is_planar
num_channels
```

And for homogeneous constructs we also have:

```
channel_type
```

Example:

```
BOOST_STATIC_ASSERT((boost::is_same<
               color_space_type<rgb8_image_t>::type,rgb_t>::value));
BOOST_STATIC_ASSERT((is_planar<rgb8_planar_view_t>::value==true));
```

To get the pixel type or pixel reference/const reference type of an image, image view, locator, and pixel, use member typedefs `value_type`, `reference` and `const_reference`.

Changes in `locator`, `image`, `image_view`, `any_image` and `any_image_view`:

Removed `dynamic_x_step_t, dynamic_y_step_t, dynamic_xy_step_t`, and `dynamic_xy_step_transposed_t` as member typedefs of locators and image views. Instead, there are separate concepts `HasDynamicXStepTypeConcept, HasDynamicYStepTypeConcept, HasTransposedTypeConcept` which all GIL provided locators, views and images model. Those concepts require a metafunction to get the corresponding type. Analogously, all GIL pixel iterators model `HasDynamicXStepTypeConcept`.

- Fixed some bugs in defining reference proxies. Also added the required swap function for reference proxies, since the std::swap default does not do the right thing.

- Added metafunctions `iterator_type_from_pixel` and `view_type_from_pixel` to allow creating standard iterators and views associated with a pixel type.

Documentation:

- The design guide and tutorial have been updated with GIL 2.0 changes. The syntax used in concepts is updated with the latest concepts proposal (though there is still some syntax we are using that is probably not legal)
- The Doxygen documentation has been updated and restructured, so that concepts and models are closer in the browse tree.

Channel changes:

- The channel min/max value is now part of the channel traits. For all built-in types the channel range equals the physical range (as determined by `numeric_traits<T>::max()`).

- Added `scoped_channel_value`, a channel adaptor that changes the operational range of a channel. bits32f is defined as a float with range 0.0 to 1.0

- Added `packed_channel_value`, `packed_channel_reference` and `packed_dynamic_channel_reference` - models of channels operating on bit ranges. (see below for an example)

- Added support for `channel_convert` between any of the GIL provided channel types. The operation is also consistent - conversion is done as a linear mapping that maps the min/max to the min/max

- Added a comprehensive regression test for channels

Pixel changes:

- Major redesign of pixel-level constructs. Got rid of the channel accessors. `color_base` is renamed to `homogeneous_color_base` and is defined once, not for each color space. In general, the work needed to define a new color space is very minimal. Here is all you need to create an RGB color space with RGB and BGR ordering:

```
// create channel names
struct red_t {};
struct green_t {};
struct blue_t {};

// create a color space
```

```
typedef mpl::vector3<red_t,green_t,blue_t> rgb_t;

// create layouts (color space with associated channel ordering)
typedef layout<rgb_t> rgb_layout_t;
typedef layout<rgb_t, mpl::vector3_c<int,2,1,0> > bgr_layout_t;
```

- As the example shows, the color space now only specifies the set of channels. Their ordering in memory is specified by a layout. pixel is now templated over the channel value and layout:

```
typedef pixel<bits8, bgr_layout_t> bgr8_pixel_t;
```

- Color base is a first-class concept. Think of color base as a bundle of color elements. A pixel is a color base whose color elements are channels. A planar pixel iterator is a color base whose elements are channel iterators. A planar pixel reference proxy is a color base whose elements are channel references. A planar image can be represented as a color base whose elements are image planes, etc.

All former pixel-level algorithms and accessors now operate on color bases. The elements of a color base can be accessed by physical or semantic index or by name. Example:

```
rgb8_pixel_t rgb8(1,2,3);
bgr8_pixel_t bgr8(rgb8);

// Physical, semantic and named element accessors.
assert(at_c<0>(bgr8) != at_c<0>(rgb8));
assert(semantic_at_c<0>(bgr8) == semantic_at_c<0>(rgb8));
assert(get_color(bgr8,blue_t()) == get_color(rgb8,blue_t()));

// Physical element accessor whose index is specified at run time.
// Only works for homogeneous bases
assert(dynamic_at_c(bgr8,0) != dynamic_at_c(rgb8,0));
assert(bgr8[0] != rgb8[0]); // for pixels only, operator[] does the same
```

- channel names can no longer be accessed as members of the pixel (`my_pixel.gray = 0`). Use `get_color` instead, as shown above.

Renamed:

FROM	TO
equal_channels	static_equal
copy_channels	static_copy
fill_channels	static_fill
generate_channels	static_generate
for_each_channel	static_for_each
transform_channels	static_transform
min_channel	static_min
max_channel	static_max
channel	at_c
semantic_channel	semantic_at_c
get_nth_channel	dynamic_at_c
planar_ptr	planar_pixel_iterator
planar_ref	planar_pixel_reference
PixelConcept	HomogeneousPixelConcept
HeterogeneousPixelConcept	PixelConcept

- added metafunctions to get the k-th element of a color base (or its reference):

```
kth_semantic_element_type
kth_semantic_element_reference_type
kth_semantic_element_const_reference_type

color_element_type
color_element_reference_type
color_element_const_reference_type

element_type
element_reference_type
element_const_reference_type
```

Example:

```
kth_semantic_element_type<rgb8_pixel_t,1>::type green =
semantic_at_c<1>(my_rgb);
```

`my_pixel::num_channels` is no longer available. To get the number of elements of a color base use the metafunction `size`:

```
BOOST_STATIC_ASSERT(gil::size<rgb8_pixel_t>::value == 3);
```

- Added `heterogeneous_packed_pixel`, a model of a pixel whose channels are bit ranges. For example, here is how to define a 16-bit RGB pixel in the '565' format:

```
typedef const packed_channel_reference<uint16_t, 0,5,true> rgb565_channel0_t;
typedef const packed_channel_reference<uint16_t, 5,6,true> rgb565_channel1_t;
typedef const packed_channel_reference<uint16_t,11,5,true> rgb565_channel2_t;

typedef mpl::vector3<rgb565_channel0_t,rgb565_channel1_t,rgb565_channel2_t>
                                                          rgb565_channels_t;

typedef heterogeneous_packed_pixel<uint16_t,rgb565_channels_t,rgb_layout_t>
                                                             rgb565_pixel_t;
```

- most color base algorithms now can take heterogeneous pixels (i.e. pixels each channel of which may have a different type). `color_convert` can operate on heterogeneous pixels with the exception of to/from RGBA.

- got rid of `pixel_traits`. Use nested typedefs `value_type`, `reference` and `const_reference` or metafunctions implementing `PixelBasedConcept` (see below).

- No more LAB and HSB color space, because there is no color conversion support implemented for these. New color spaces can be added with just a few lines of code, as shown above.

- added a comprehensive regression test for pixels

Pixel iterator changes:

- got rid of `pixel_iterator_traits`. Use `std::iterator_traits`, `PixelBasedConcept` metafunctions or the following new metafunctions for pixel iterators:

```
const_iterator_type
iterator_is_mutable
is_iterator_adaptor
```

In addition, iterator adaptors have these new metafunctions:

```
iterator_adaptor_get_base
iterator_adaptor_rebind
```

- renamed `pixel_image_iterator` to `iterator_from_2d`

Pixel locator changes:

Renamed `is_contiguous` to `is_1d_traversable`.

Renamed `membased_2d_locator` to `byte_addressable_2d_locator.`

Image view changes:

- added algorithms `uninitialized_fill_pixels` and `uninitialized_copy_pixels.`

- added method `is_1d_traversable.`

Image changes:

- Images don't allow for getting access to the pixels - only through views. Got rid of the ability to directly navigate the pixels of an image. So image no longer models STL's random access container concept
- The class image is no longer templated over the image view. It is now templated over pixel value and a Boolean indicating if the image is planar:

```
typedef image<rgb8_pixel_t, true> rgb8_planar_image_t;
```

- Added support for creating images with a new value to fill.
- Images now invoke the default constructor of the pixels they allocate.
- Renamed `resize_clobber_image` to Image::`recreate`. Also allowed for optionally specifying the initial value.

Dynamic image changes:

- No `cross_vector_image_types` and `cross_vector_image_view_types`. Instead, just create a vector to explicitly enumerate your types. This resulted in removing a lot of MPL related code and simplified significantly the design.