boost::gil

concept based image processing

context

- created by adobe >10y ago alongside adam&eve
  - adam: property dependency trees
  - eve: property<->view binding
  - gil: concept based image processing
  - accepted in boost in 2006
  - mostly dormant, “done”?

gil motivation

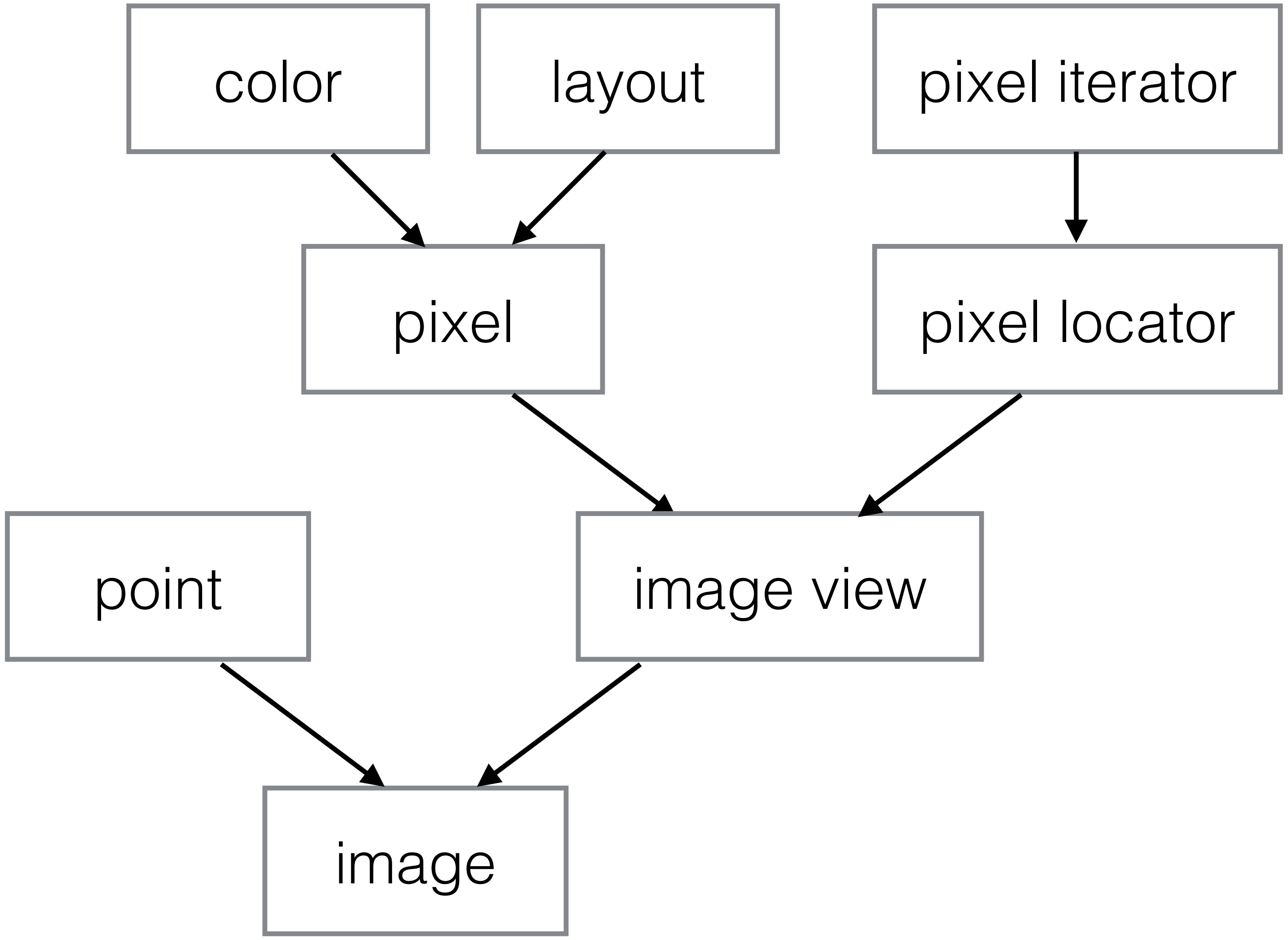
- define algorithms based on concepts
- freedom of implementation/extensibility
  - storage
  - layout
  - color space
- provides models&algorithms

concepts

- type interfaces (“duck typing”)
- not in c++ (yet)

```
auto concept DefaultConstructible<typename T> {
    T::T();
};

auto concept CopyConstructible<typename T> {
    T::T(T);
    T::~T();
};

auto concept Assignable<typename T, typename U = T> {
    typename result_type;
    result_type operator=(T&, U);
};

auto concept EqualityComparable<typename T, typename U = T> {
    bool operator==(T x, T y);
    bool operator!=(T x, T y) { return !(x==y); }
};

concept SameType<typename T, typename U> { /* unspecified */ };
template<typename T> concept_map SameType<T, T> { /* unspecified */ };

auto concept Swappable<typename T> {
    void swap(T& t, T& u);
};

auto concept Regular<typename T> : DefaultConstructible<T>, CopyConstructible<T>,
                                   EqualityComparable<T>, Assignable<T>, Swappable<T>
{};

auto concept Metafunction<typename T> {
    typename type;
};
```

gil concepts

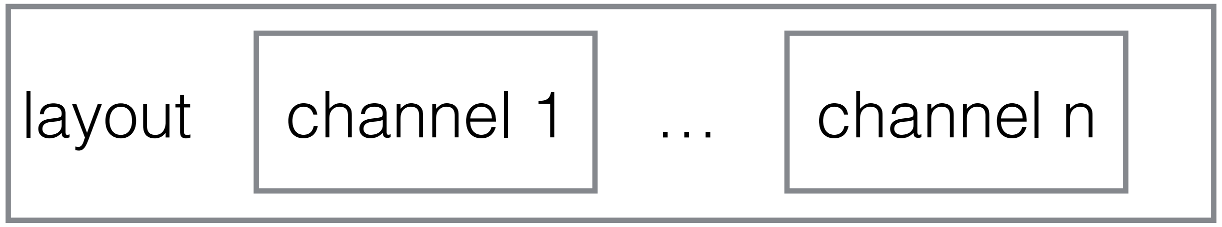


Point

```
concept PointNDConcept<typename T> : Regular<T> {
    // the type of a coordinate along each axis
    template <size_t K> struct axis; where Metafunction<axis>;

    const size_t num_dimensions;

    // accessor/modifier of the value of each axis.
    template <size_t K> const typename axis<K>::type& T::axis_value() const;
    template <size_t K>       typename axis<K>::type& T::axis_value();
};


concept Point2DConcept<typename T> : PointNDConcept<T> {
    where num_dimensions == 2;
    where SameType<axis<0>::type, axis<1>::type>;

    typename value_type = axis<0>::type;

    const value_type& operator[](const T&, size_t i);
          value_type& operator[](      T&, size_t i);

    value_type x,y;
};
```

Pixel

- layout of named channels with given precisions



- rgba8 {{red : 8bits}, {green : 8bits}, {blue : 8bits}}
- bgr232 {{blue: 2bits}, {green: 3bits}, {red: 2bits}}
- gil provides swizzling assignment between same precision types, conversions for different precision types

Channel

```
concept ChannelConcept<typename T> : EqualityComparable<T> {
    typename value_type      = T;        // use channel_traits<T>::value_type to access it
       where ChannelValueConcept<value_type>;
    typename reference       = T&;       // use channel_traits<T>::reference to access it
    typename pointer         = T*;       // use channel_traits<T>::pointer to access it
    typename const_reference = const T&; // use channel_traits<T>::const_reference to access it
    typename const_pointer   = const T*; // use channel_traits<T>::const_pointer to access it
    static const bool is_mutable;        // use channel_traits<T>::is_mutable to access it

    static T min_value();                // use channel_traits<T>::min_value to access it
    static T max_value();                // use channel_traits<T>::min_value to access it
};

concept MutableChannelConcept<ChannelConcept T> : Swappable<T>, Assignable<T> {};

concept ChannelValueConcept<ChannelConcept T> : Regular<T> {};
```

Color

```
concept ColorBaseConcept<typename T> : CopyConstructible<T>, EqualityComparable<T> {
    // a GIL layout (the color space and element permutation)
    typename layout_t;

    // The type of K-th element
    template <int K> struct kth_element_type;
        where Metafunction<kth_element_type>;

    // The result of at_c
    template <int K> struct kth_element_const_reference_type;
        where Metafunction<kth_element_const_reference_type>;

    template <int K> kth_element_const_reference_type<T,K>::type at_c(T);

    template <ColorBaseConcept T2> where { ColorBasesCompatibleConcept<T,T2> }
        T::T(T2);
    template <ColorBaseConcept T2> where { ColorBasesCompatibleConcept<T,T2> }
        bool operator==(const T&, const T2&);
    template <ColorBaseConcept T2> where { ColorBasesCompatibleConcept<T,T2> }
        bool operator!=(const T&, const T2&);

};
```

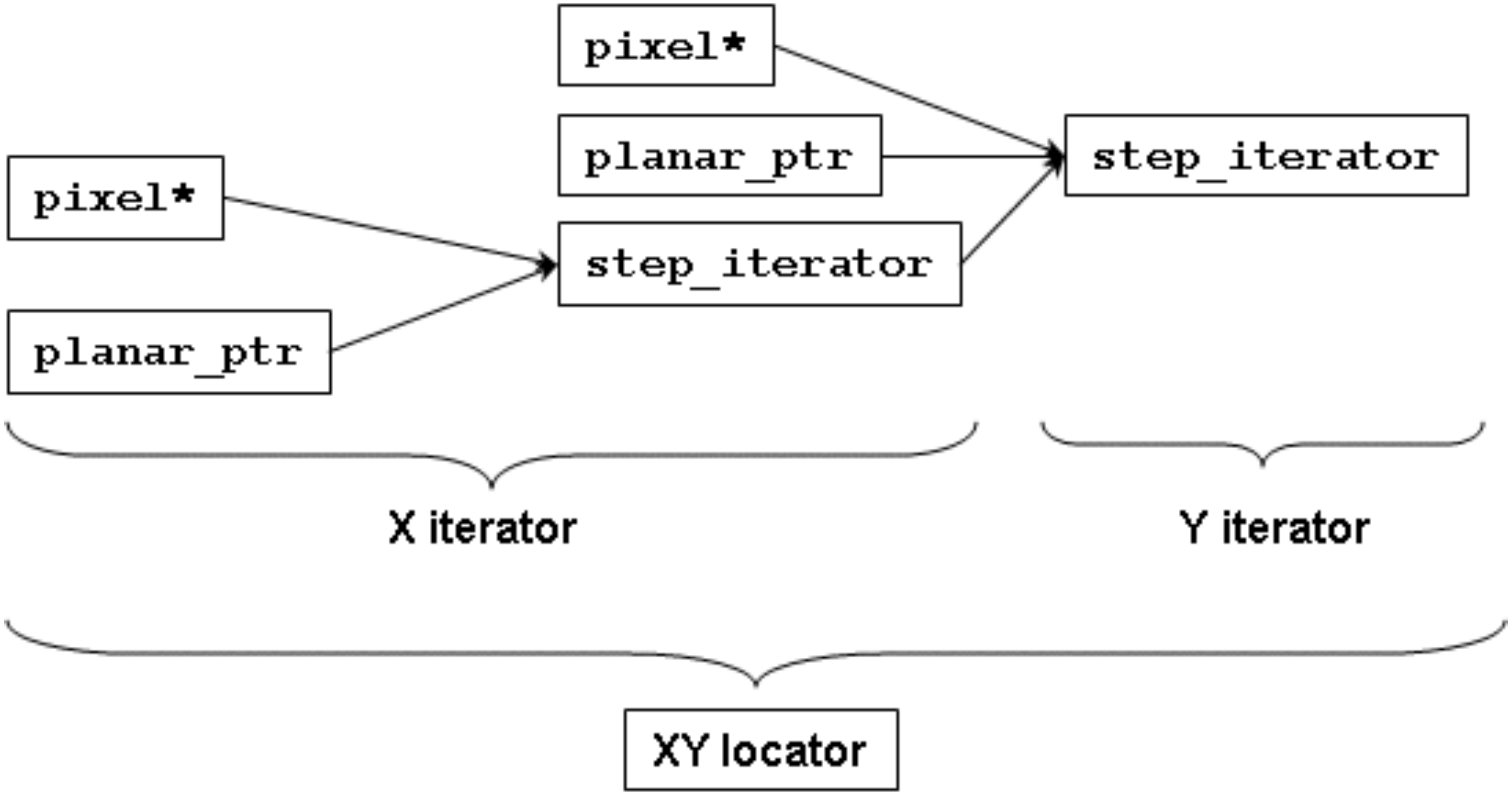
Pixel

```
concept PixelBasedConcept<typename T> {
    typename color_space_type<T>;
        where Metafunction<color_space_type<T> >;
        where ColorSpaceConcept<color_space_type<T>::type>;
    typename channel_mapping_type<T>;
        where Metafunction<channel_mapping_type<T> >;
        where ChannelMappingConcept<channel_mapping_type<T>::type>;
    typename is_planar<T>;
        where Metafunction<is_planar<T> >;
        where SameType<is_planar<T>::type, bool>;
};

concept PixelConcept<typename P> : ColorBaseConcept<P>, PixelBasedConcept<P> {
    where is_pixel<P>::type::value==true;

    typename value_type;       where PixelValueConcept<value_type>;
    typename reference;        where PixelConcept<reference>;
    typename const_reference;  where PixelConcept<const_reference>;
    static const bool P::is_mutable;

    template <PixelConcept P2> where { PixelConcept<P,P2> } P::P(P2);
    template <PixelConcept P2> where { PixelConcept<P,P2> }
        bool operator==(const P&, const P2&);
    template <PixelConcept P2> where { PixelConcept<P,P2> }
        bool operator!=(const P&, const P2&);
};
```

Color Algorithms

```
// This is how to access the first semantic channel (red)
assert(semantic_at_c<0>(rgb8) == semantic_at_c<0>(bgr8));

// This is how to access the red channel by name
assert(get_color<red_t>(rgb8) == get_color<red_t>(bgr8));

// This is another way of doing it (some compilers don't like the first one)
assert(get_color(rgb8,red_t()) == get_color(bgr8,red_t()));

get_color<red_t>(ref) = 10;     // assignment is ok because the reference is mutable

rgb8_pixel_t red_in_rgb8(255,0,0);
cmyk16_pixel_t red_in_cmyk16;
color_convert(red_in_rgb8,red_in_cmyk16);
```

iterators&locators



Pixel Iterator

```
concept PixelIteratorConcept<RandomAccessTraversalIteratorConcept Iterator>
  : PixelBasedConcept<Iterator>
{
    where PixelValueConcept<value_type>;
    typename const_iterator_type<It>::type;
        where PixelIteratorConcept<const_iterator_type<It>::type>;
    static const bool  iterator_is_mutable<It>::type::value;
    static const bool  is_iterator_adaptor<It>::type::value;
};
```

Pixel Locator

```
concept RandomAccessNDLocatorConcept<Regular Loc>
{
    typename value_type;        // value over which the locator navigates
    typename reference;         // result of dereferencing
    typename difference_type; where PointNDConcept<difference_type>; // return value of operator-.
    typename const_t;           // same as Loc, but operating over immutable values
    typename cached_location_t; // type to store relative location (for efficient repeated access)
    typename point_t  = difference_type;

    static const size_t num_dimensions; where num_dimensions = point_t::num_dimensions;

    Loc& operator+=(Loc&, const difference_type&);
    Loc& operator-=(Loc&, const difference_type&);
    Loc operator+(const Loc&, const difference_type&);
    Loc operator-(const Loc&, const difference_type&);

    reference operator*(const Loc&);
    reference operator[](const Loc&, const difference_type&);

    // Storing relative location for faster repeated access and accessing it
    cached_location_t Loc::cache_location(const difference_type&) const;
    reference operator[](const Loc&,const cached_location_t&);

    // Accessing iterators along a given dimension at the current location or at a given offset
    template <size_t D> axis<D>::iterator&       Loc::axis_iterator();
    template <size_t D> axis<D>::iterator const& Loc::axis_iterator() const;
    template <size_t D> axis<D>::iterator        Loc::axis_iterator(const difference_type&) const;
};
```

images

- planar vs. interleaved
- different sources
  - owned memory
  - file
  - foreign memory

Image

```
concept RandomAccessNDImageConcept<typename Img> : Regular<Img>
{
    typename view_t; where MutableRandomAccessNDImageViewConcept<view_t>;
    typename const_view_t = view_t::const_t;
    typename point_t      = view_t::point_t;
    typename value_type   = view_t::value_type;
    typename allocator_type;

    Img::Img(point_t dims, std::size_t alignment=0);
    Img::Img(point_t dims, value_type fill_value, std::size_t alignment);

    void Img::recreate(point_t new_dims, std::size_t alignment=0);
    void Img::recreate(point_t new_dims, value_type fill_value, std::size_t alignment);

    const point_t&        Img::dimensions() const;
    const const_view_t&   const_view(const Img&);
    const view_t&         view(Img&);
};
```

views

- const/mutable
- subregions
- lazy transformations
    - rotation
    - scale
    - color conversion
    - …

view example

```
jpeg_read_image("monkey.jpg", img);
step1=view(img);
step2=subimage_view(step1, 200,300, 150,150);
step3=color_converted_view<rgb8_view_t,gray8_pixel_t>(step2);
step4=rotated180_view(step3);
step5=subsampled_view(step4, 2,1);
jpeg_write_view("monkey_transform.jpg", step5);
```

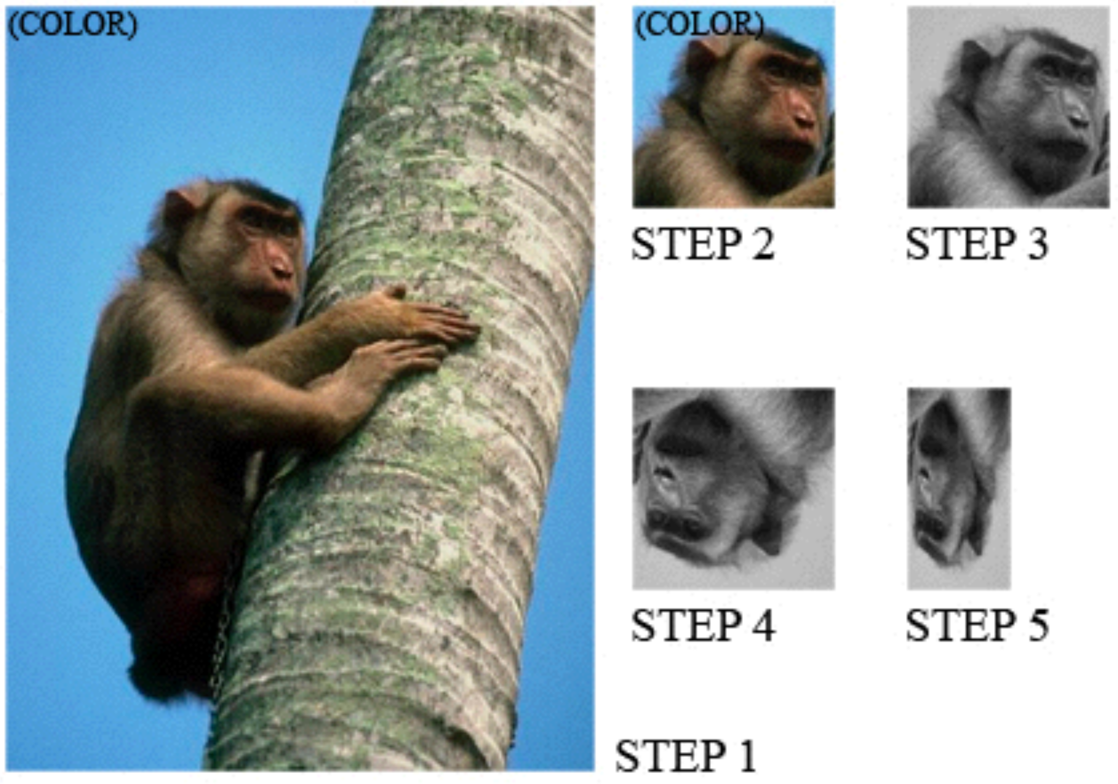


Image View

```
concept RandomAccessNDImageViewConcept<Regular View> {
    typename value_type;      // for pixel-based views, the pixel type
    typename reference;       // result of dereferencing
    typename difference_type; // result of operator-(iterator,iterator) (1-dimensional!)
    typename const_t;  where RandomAccessNDImageViewConcept<View>; // same as View, but over immutable values
    typename point_t;  where PointNDConcept<point_t>; // N-dimensional point
    typename locator;  where RandomAccessNDLocatorConcept<locator>; // N-dimensional locator.
    typename iterator; where RandomAccessTraversalConcept<iterator>; // 1-dimensional iterator over all values
    typename reverse_iterator; where RandomAccessTraversalConcept<reverse_iterator>;
    typename size_type;       // the return value of size()

    static const size_t num_dimensions = point_t::num_dimensions;

    // Create from a locator at the top-left corner and dimensions
    View::View(const locator&, const point_type&);

    size_type        View::size()       const; // total number of elements
    reference        operator[](View, const difference_type&) const; // 1-dimensional reference
    iterator         View::begin()      const;
    iterator         View::end()        const;
    reverse_iterator View::rbegin()     const;
    reverse_iterator View::rend()       const;
    iterator         View::at(const point_t&);
    point_t          View::dimensions() const; // number of elements along each dimension
    bool             View::is_1d_traversable() const;   // Does an iterator over the first dimension visit each value?

    // iterator along a given dimension starting at a given point
    template <size_t D> View::axis<D>::iterator View::axis_iterator(const point_t&) const;

    reference operator()(View,const point_t&) const;
};
```

models

- many color types/layouts
- dense and stepped iterators
- planar and interleaved images
- const&mutable views from whole image, subregions, transformations
- polymorphic images&view

algorithms

- very few
  - fill
  - copy/assignment
  - color conversion
  - generic image transform
- examples reference numeric extension which seems not to be part of boost (yet?)
  - at http://gil-contributions.googlecode.com/svn/trunk/
  - convolution
  - resampling
  - …

relevance

- nowadays loses relevance because GPUs are much faster
- for complex image processing opencv is much more potent (and contains GPU implementations)
- still useful for some quick image processing hacks and file io
    - cleaner and less heavy than opencv
    - part of boost
    - some algorithms are better on cpu

example

- switch to editor…