

Brown University



# **VoxGIS Software Documentation**

J.L Mundy, Isabel Restrepo, Gamze Tunali

March 2008

# TABLE OF CONTENTS

- I. Introduction**
- II. Auxiliary Libraries**
  - i. Brown Relational Database: *brdb*
  - ii. Brown Process Basics Library: *bprb*
  - iii. Brown Statistics Library: *bst*
- III. BVXM: Brown Voxel –something– Modeling Library**
  - i. BVXM Basics
    - Memory Structure
    - Utility Classes
    - The voxel world
    - The appearance model processors
  - ii. BVXM Pro
- IV. BVXM\_BATCH**
- V. Additional Information**

## I. Introduction

This document describes in detail *bvxm*, Brown Volumetric Appearance Model library, and its dependency to existing VXL libraries.

## II. Auxiliary Libraries

### i. **Brown Relational Database Library: brdb** (*vxl/contrib/brl/bbas/brdb*)

The purpose of this library is to provide the necessary classes and the interface to generate an in memory SQL database, and to perform standard SQL operations.

One of the functionalities of the database is its ability to save itself. Therefore, any object that is stored in the database **must implement VXL I/O streaming methods; *vsl\_b\_write* and *vsl\_b\_read***.

Class Diagram:

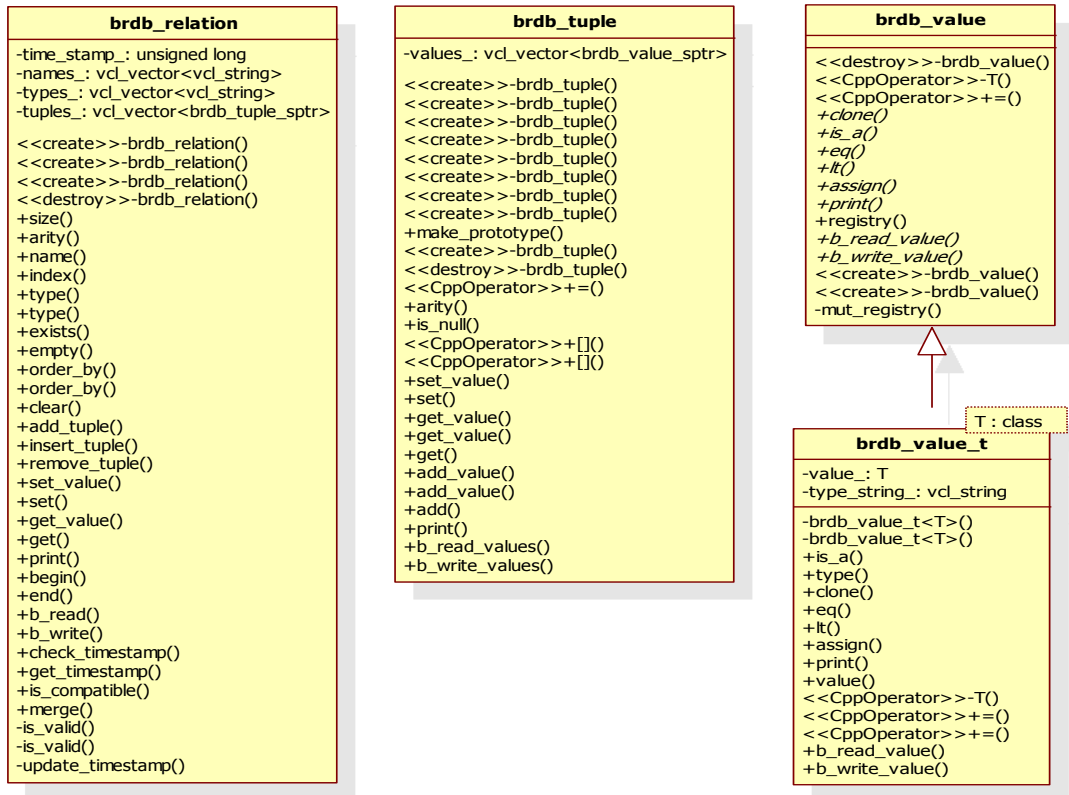


Figure 1. Database Objects :Relations, tuples and valued

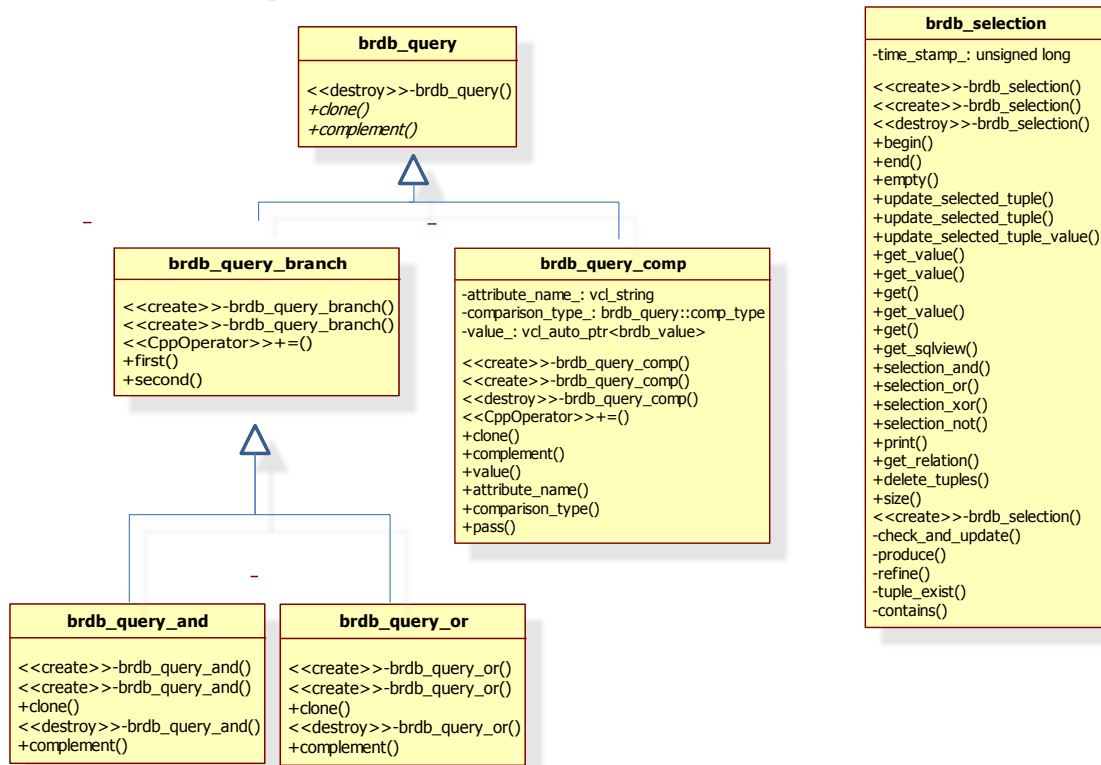


Figure 2. Data Manipulation Language (DML): Insert, select, update, delete.

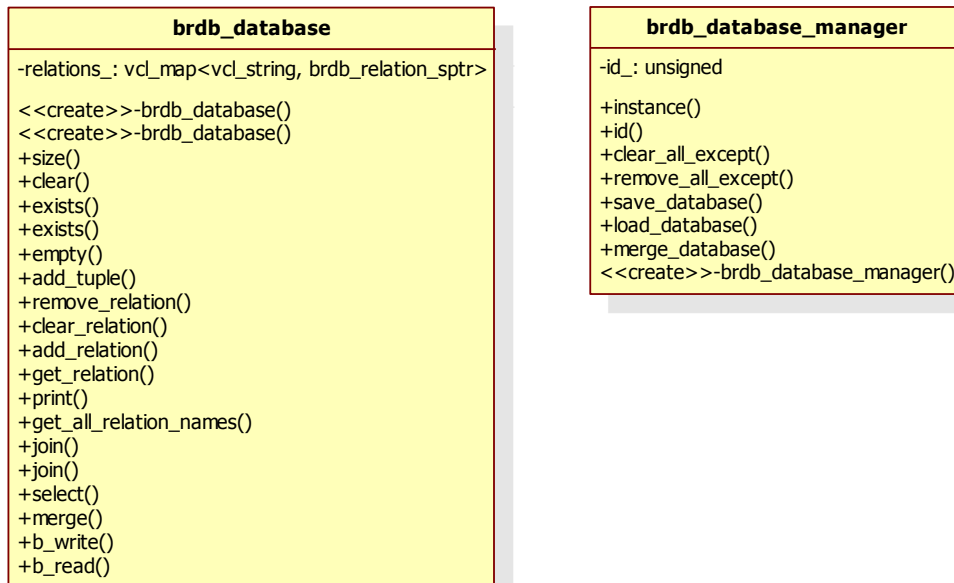


Figure 3: Data Definition Language (DDL): Create, save, load

## ii. Brown Process Basics Library: bprb (*vx1/contrib/brl/bpro/bprb*)

Brown Process Basics library provides a framework to create and execute processes. *bprb* consists of :

- Two process managers:
  - *bprb\_process\_manager*
  - *bprb\_batch\_process\_manager*
- A process
  - *bprb\_process*
- Process parameters related classes
  - *bprb\_parameter*
  - *bprb\_param*
  - *bprb\_param\_type*

### Class Diagram:

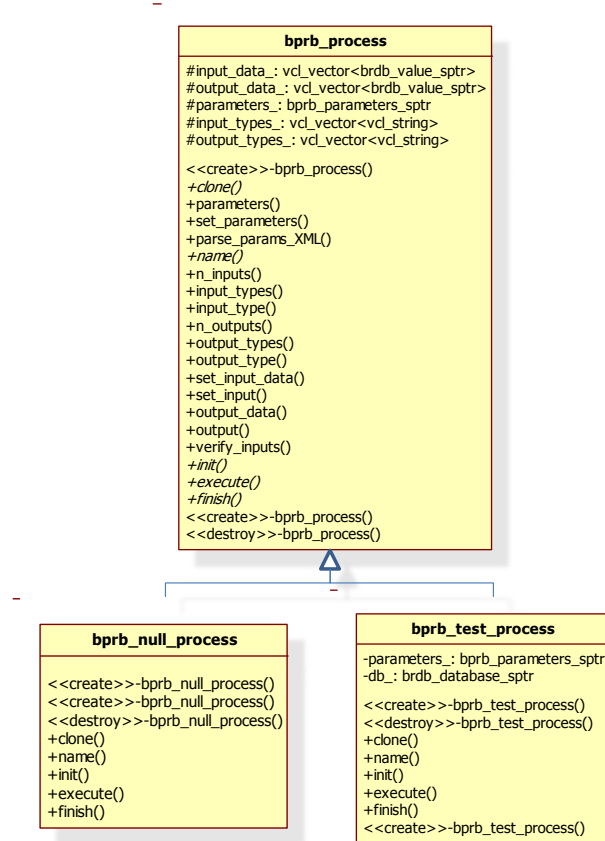
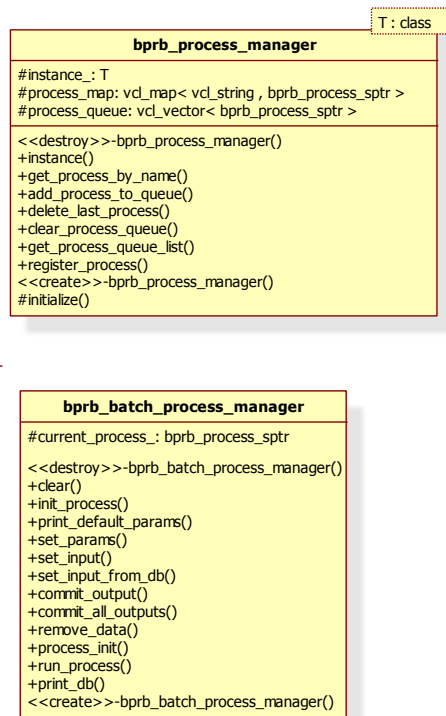


Figure 4: *bprb* Process Managers

Figure 5: *bprb* Processes

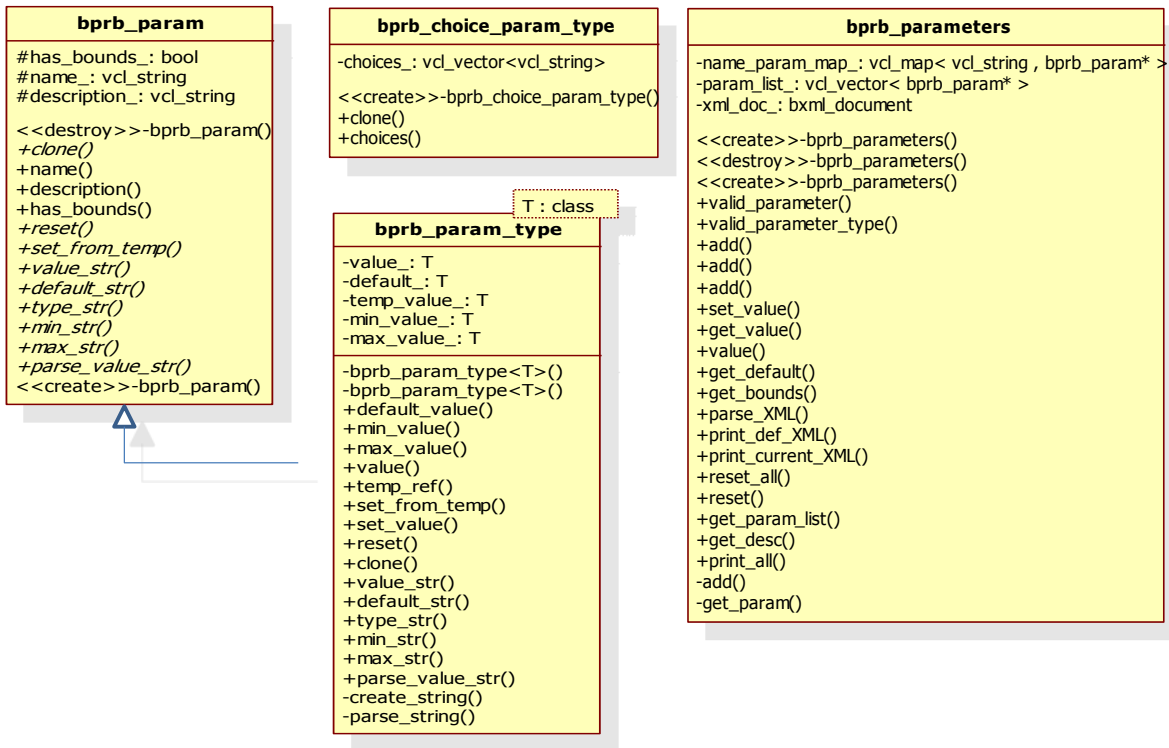


Figure 6: *bprb* Parameters Family

### Detailed Description

- bprb\_process:** Is an abstract class that manages its inputs and outputs by an in-memory database library: *brdb*, which keeps smart pointers to the data that the process uses/produces. Also, a process receives parameters which are defined in an XML file. The default values for the parameters are defined in the constructor of the process; and the current values are parsed from the XML file using XML parsing library *bxml*. The XML format expected for the parameters is the following:

```

<?xml version="1.0" encoding="utf-8" ?>
<ProcessName>
    <Param1 type="type1" desc="description of param1" value="10.0"/>
    <Param2 type="type2" desc="description of param2" value="x"/>
</ProcessName>

```

In addition to methods to set and get inputs, output and parameters, the process class has three fundamental methods: *init()*, *execute()* and *finish()*. These methods are virtual and they are expected to be implemented by the inheriting classes of *bprb\_process*. *init()* should do the necessary initializations before the process is ex-

ecuted. Finish() should do the clean up or final steps execution. The *execute()* method is the heart of the process where the main processing is done.

- **prb\_param**: Is an abstract class that defines the parameter's name, description and boundaries. It has methods to access and change those values.
- **bprb\_param\_type**: A templated class to accommodate any type of parameter. It inherits from *bprb\_param* and defines *value\_*, *default\_*, *temp\_value\_*, *min\_value\_* and *max\_value\_* members. Other than getters and setters, it converts the values to string and parses them from string.
- **bprb\_choice\_param\_type**: A parameter class for handling multiple choice parameters. It inherits from *bprb\_param\_type<unsigned>* and keeps a vector of strings as choices.
- **bprb\_parameters**: Keeps a list of type of parameters as *bprb\_param* pointers. Rapid access to these parameters is achieved by maintaining a map from parameter names to the parameters. It provides methods to add parameters, set and get their values, descriptions, defaults and currents. Using *parse\_XML*, the parameters can also be set from a given XML file whose format was defined earlier. It also provides methods to write the parameters (default and current) to XML files (*print\_def\_XML(...)* and *print\_current\_XML(...)*) and to the output stream.

### iii. **Brown Statistics Library: bsta**

This is a comprehensive and flexible library. It provides fundamental distributions and statistical methods. Most of its classes are templated, allowing probability distributions and methods to be as generic as possible. In VoxGIS software, this library is used to maintain the mixture of Gaussians appearance model. Since this library is far more extensive than what needs to be understood for VoxGIS software, only a brief description of its classes is presented below:

- **bsta\_gauss**: 1-d and 2-d Gaussian smoothing for Parzen window calculations
- **bsta\_histogram**: A simple histogram class. Supports entropy calculations
- **bsta\_joint\_histogram**: A simple joint\_histogram class
- **bsta\_k\_medoid**: Form k clusters using distance to representative objects (medoids)
- **bsta\_otsu\_threshold**: Implements Otsu's threshold method for 1D distribution and 2 classes
- **bsta\_int\_histogram\_1d**: 1D integer Histograms with bucket width = 1
- **bsta\_int\_histogram\_2d**: 1D and 2D integer Histograms with bucket width = 1
- **bsta\_distribution**: A base class for probability distributions
- **bsta\_gaussian** : A Gaussian distribution for use in a mixture model
- **bsta\_gaussian\_full**: A Gaussian distribution with a full covariance matrix
- **bsta\_gaussian\_indep**: A Gaussian distribution, independent in each dimension

- `bstgaussian_sphere`: A (hyper-)spherical Gaussian distribution (i.e. single variance)
- `bstmixture`: A mixture of distributions
- `bstmixture_fixed`: A mixture of distributions of fixed size
- `bstbasic_functors`:
  - `bstgaussian_prob_density_funct`: A functor to return the probability density at a sample
  - `bstgaussian_probability_funct`: A functor to return the probability integrated over a box
  - `bstgaussian_mean_funct`: A functor to return the mean of the Gaussian
  - `bstgaussian_var_funct`: A functor to return the variance of the Gaussian
  - `bstgaussian_diag_covar_funct`: A functor to return the variance of the Gaussian
  - `bstgaussian_det_covar_funct`: A functor to return the determinant of the covariance of the Gaussian
  - `bstmixture_weight_funct`: A functor to return the weight of the component with given index
  - `bstmixture_funct`: A functor to apply another functor to one distribution in the mixture
  - `bstmixture_data_funct`: A functor to apply another functor with data to one distribution in the mixture
  - `bstweighted_sum_funct`: A functor to apply another functor to each distribution and produce a weighted sum
  - `bstweighted_sum_data_funct`: A functor to apply another functor with data to each distribution and produce a weighted sum
  - `bstmixture_size_funct`: A functor to count the number of components in the mixture
- `bstdetector_gaussian`: Designed to include detectors applying to Gaussians. Currently it has one detector: `bstgaussian_md_dist_detector`, a simple Mahalanobis distance detector for a Gaussian
- `bstdetector_mixture`: Designed to include detectors applying to mixtures. Currently it has two detectors:
  - `bstmixture_top_weight_detector`: applies a detector to each component in order while the total weight is below a threshold
  - `bstmixture_mix_any_less_index_detector`: applies a detector to each component in order while the total weight is below a threshold



### III. BVXM: Brown Voxel -something- Modeling Library

Library architecture:

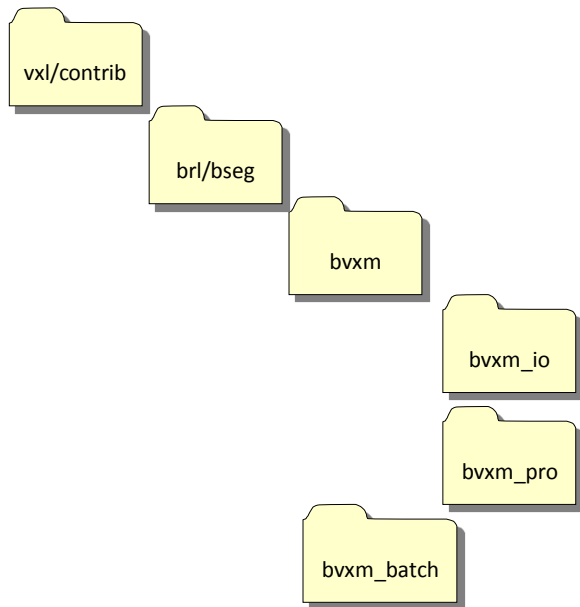


Figure 7: *bvxm* library architecture

- **bvxm:** This library contains basic classes to support volumetric data structures, processors and memory management.
  - **bvxm/pro:** A library of processes to be applied to *bvxm* structures.
  - **bvxm/io:** A library of classes managing I/O parsing of any *bvxm\_parameter* or I/O of any *bvxm* class that needs to be stored in the database.

#### i. BVXM Basics:

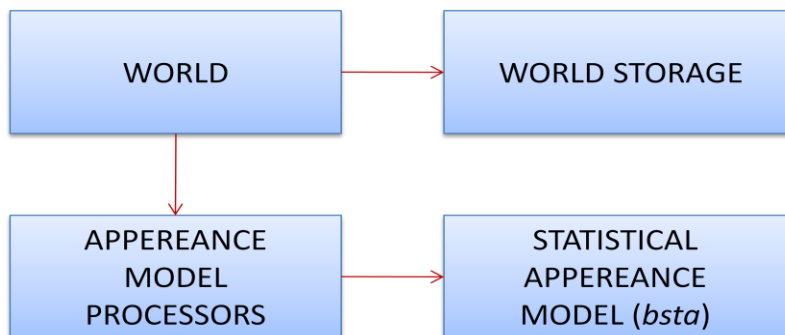


Figure 8: *BVXM Basics architecture diagram*

## World Storage: A memory family

- *bvxm\_memory\_chunk*: Block of data on the heap. It inherits from *vbl\_ref\_count* to keep a count of its references.
- *bvxm\_voxel\_slab*: A 3-D slab of data of any type, with data stored on the heap (via an associated *bvxm\_memory\_chunk*). This class is a generic template class that provides access to the data through *bvxm\_voxel\_slab\_iterator*<T> and “()” operators.
- *bvxm\_voxel\_grid*: A 3-D container for voxel data of generic type T. A voxel grid is meant to store an entire voxel grid, giving access to chunks via objects of type *bvxm\_voxel\_slab*<T>. Some careacteristics are
- *bvxm\_voxel\_storage*: An abstract class for storing and providing access to voxel data.
- *bvxm\_voxel\_storage\_disk*: Stores voxel data in a single file on disk, this file remains opened until the storage object goes out of scope
- *bvxm\_voxel\_storage\_mem*: Stores the voxel data on the heap. Basically, this class is a wrapper around a *bvxm\_memory\_chunk*

### Class Diagram



Figure 9: High-level *bvxm* storage

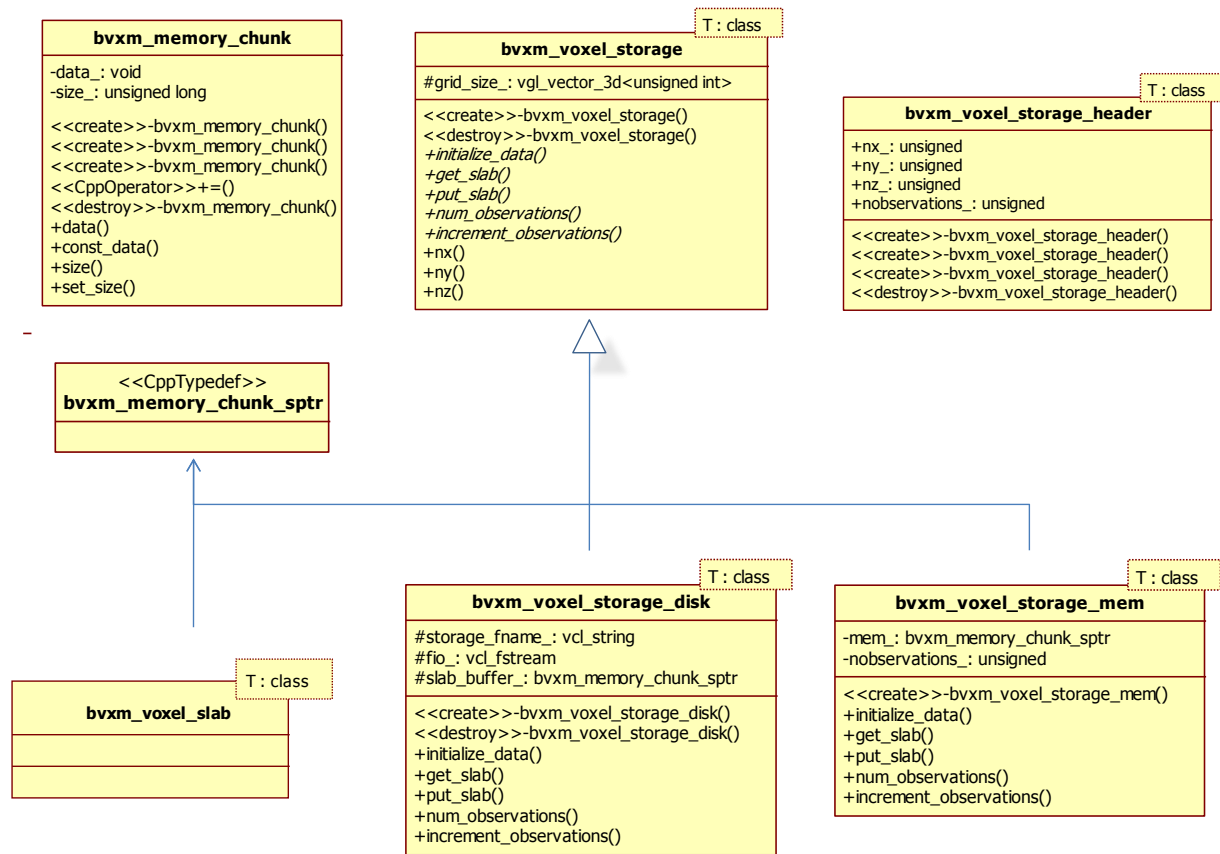


Figure 10: Low-level *bvxm* memory management

## Important Classes used by *bvxm\_voxel\_world*

Before describing the voxel world, it is important to understand the main classes that *bvxm\_world* uses to obtain parameters, data and data types

- **The World Parameters: *bvxm\_world\_params.h***

This class contains all the parameters necessary for the generation of *bvxm\_voxel\_world*.

### Class Diagram

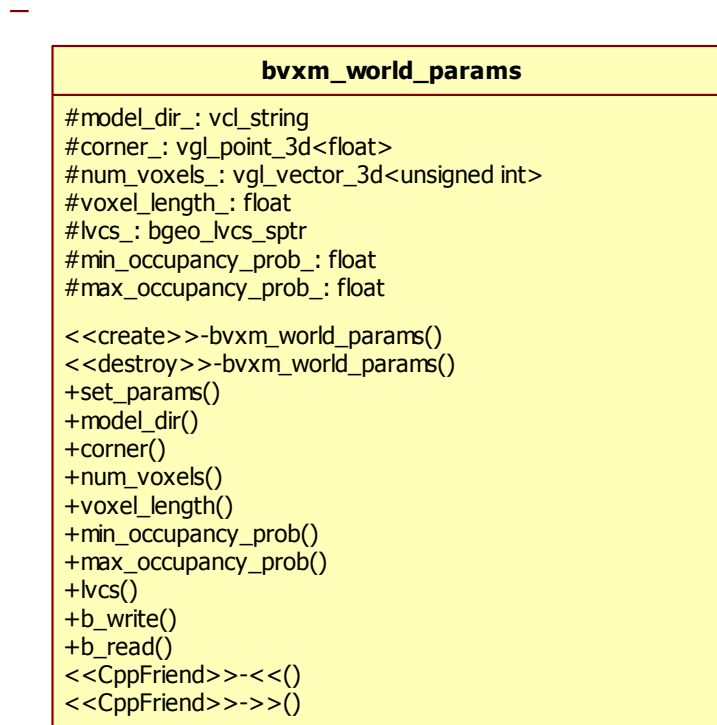


Figure 11: *bvxm\_world\_params* class

### Detailed Description:

- *model\_dir\_*: The directory where the model data, like images, resides. This directory is also used for temporary and output data.
- *corner\_*: The coordinates of the corner (minimum point) of the voxel world in local coordinate system.
- *num\_voxels\_*: Number of voxel in x, y and z directions

- *voxel\_length\_* : the dimensions of voxel in meters (uniform in each direction, x, y, and z)
- *lvcs\_*: the local vertical coordinate system that ties the voxel world positioning and measurements to the geographical coordinate system (please refer to the section on *bgeo\_lvcs*)
- *min\_occupancy\_probability\_*: lower boundary of the occupancy probabilities, the lower values than this will always be adjusted to accommodate this value
- *max\_occupancy\_probability\_*: upper boundary of the occupancy probabilities, the higher values than this will always be adjusted to accommodate this value

The following diagram illustrates the voxel world that is defined by the world parameters:

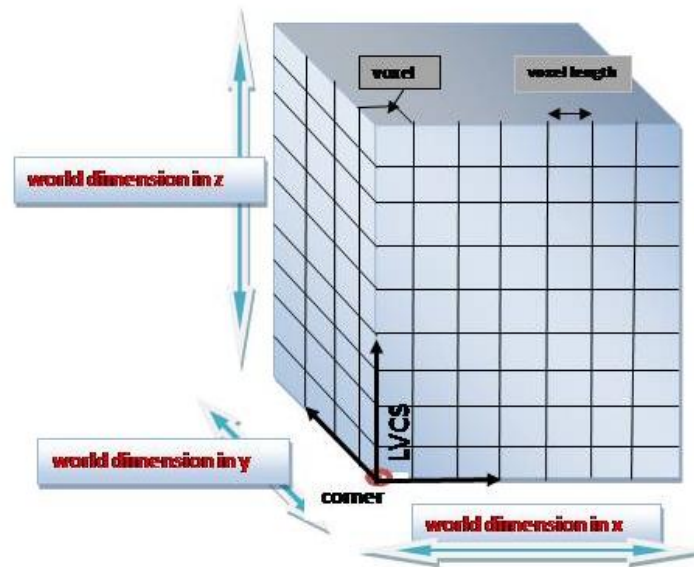


Figure 12: Voxel World

- **Pixel Traits: `bvxm_pixel_traits.h`**

This class contains important definitions that are unique to each voxel type. The first structure found in this class, enumerates the supported voxel types that can be stored in a voxel grid. This structure is shown below:

<<enumeration>> <b>bvxm_voxel_type</b>
+OCCUPANCY
+APM_MOG_GREY
+APM_MOG_RGB
+EDGES
+UNKNOWN

Following the voxel type definition, the class *bvxm\_voxel\_traits* is declared and implemented. *bvxm\_voxel\_traits* is templated over *bvxm\_voxel\_type*. Each specialization of this class contains appropriate data type definitions that are specific to a given *bvxm\_voxel\_type* e.g float, mixture of Gaussians.

The following piece of code shows *bvxm\_voxel\_traits* and some of its specializations.

```

//: Pixel properties for templates.
template <bvxm_voxel_type>
class bvxm_voxel_traits;

template<>
class bvxm_voxel_traits<OCCUPANCY>
{
public:
    //:Datatype of the occupancy probabilities
    typedef float voxel_datatype;

    static inline vcl_string filename_prefix(){ return "ocp"; }
    static inline bool is_multibin() { return false; }
    static inline voxel_datatype initial_val() { return 0.01f; }

};

template<>
class bvxm_voxel_traits<APM_MOG_RGB>
{
public:
    //:Datatype of the occupancy probabilities
    typedef bvxm_mog_rgb_processor appearance_processor;
    typedef bvxm_mog_rgb_processor::apm_datatype voxel_datatype;
    typedef bvxm_mog_rgb_processor::obs_datatype obs_datatype;
    typedef bvxm_mog_rgb_processor::obs_mathtype obs_mathtype;

    static inline vcl_string filename_prefix(){ return "apm_mog_rgb"; }
    static inline bool is_multibin() { return true; }
    static inline voxel_datatype initial_val()
    {
        voxel_datatype init_val;
        return init_val;
    }

};

...

```

- **BVXM utilities: `bvxm_util.h`**

When operating on the world, it is necessary to perform different types of operations on slabs and images. *bvxm\_util.h* provides such methods.

Class Diagram

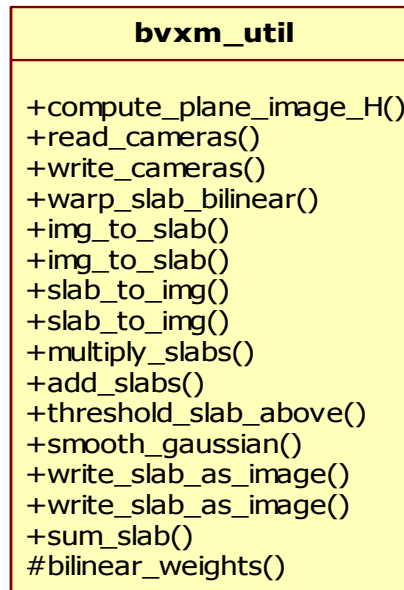


Figure 13: `bvxm_util` class

Detailed Description:

- *compute\_plane\_image\_H*: Computes homographies from voxel planes to image coordinates and vice-versa.
- *read\_cameras*: Reads *vpgl\_cameras* from a file.
- *write\_cameras*: Writes *vpgl\_cameras* to a file.
- *warp\_slab\_bilinear*: Backprojects the image onto a voxel plane given the plane-image homography. The warping is computed using bilinear interpolation.
- *image\_to\_slab*, *slab\_to\_image*: Convert images to slabs and vice-versa. This class is template over the dimension of the data. Currently gray scale and color are supported but extension to any other dimension should be trivial.
- *multiply\_slabs*: Multiplies two given slabs

- *smooth\_gaussian*: Applies a Gaussian smoothing.
- *write\_slab\_as\_image*: Saves a slab in an image file, mainly used for debugging purposes
- *sum\_slab*: Adds the elements of a slab.

### **The World: *bvxm\_voxel\_world***

*bvxm\_voxel\_world* is probably the most important class of voxGIS software. It has been implemented to be flexible, dynamic and efficient. *bvxm\_voxel\_world* can simultaneously maintain different types of voxel grids i.e. a mixture of Gaussian appearance model grid, a floating point occupancy grid or a grid containing edge information. It also provides the functions that operate on the grids i.e. update, obtain expected image, detect changes etcetera.

<b>bvxm_voxel_world</b>
<pre>#grid_map_: vcl_map&lt;bvxm_voxel_type, vcl_map&lt;unsigned int, bvxm_voxel_grid_base_sptr&gt; &gt; #params_: bvxm_world_params_sptr  &lt;&lt;create&gt;&gt;-bvxm_voxel_world() &lt;&lt;create&gt;&gt;-bvxm_voxel_world() &lt;&lt;destroy&gt;&gt;-bvxm_voxel_world() +update() +update() +expected_image() +update_edges() +expected_edge_image() +inv_pixel_range_probability() +pixel_probability_density() +mixture_of_gaussians_image() +virtual_view() +fit_plane() +get_params() +set_params() +set_grid() &lt;&lt;CppOperator&gt;&gt;+==() &lt;&lt;CppOperator&gt;&gt;+&lt;() +get_grid() +save_occupancy_raw() +clean_grids() +num_observations() +increment_observations() -update_impl()</pre>

**Figure 14: *bvxm\_voxel\_world* class**



## Detailed Description:

### Constructor & Destructor Documentation

- `bvxm_voxel_world::bvxm_voxel_world ()`
- `bvxm_voxel_world::bvxm_voxel_world (bvxm_world_params_sptr params):` Construct a voxel world give a set of parameters.
- `bvxm_voxel_world::~bvxm_voxel_world ()`

---

### Member Variables Documentation:

- **The Grid Map:**

`vcl_map<bvxm_voxel_type, vcl_map<unsigned int, bvxm_voxel_grid_base_sptr>> grid_map ;`

The voxel world can hold different types of grids at the same time. These grids are kept in a `vcl_map` where the key of the map must be one of the following `bvxm_voxel_type` :

```
enum bvxm_voxel_type
{
    OCCUPANCY = 0,
    APM_MOG_GREY,
    APM_MOG_RGB,
    EDGES,
    UNKNOWN,
};
```

The entry corresponding to each key is another `vcl_map` containing one or more `bvxm_voxel_grid_sptr`, each stored at a different bin i.e. if each bin contains a mixture of gaussian, the result is a multiple mixture of Gaussians model.

Any entry of the local map, is created the first time the user requestes it via the member function `get_grid`. This function is templeted over `bvxm_voxel_type` and it needs a bin numer (default 0) to acces the appropriate voxel grid.

```
//: Returns the voxel_grid that corresponds to a given bvxm_voxel_type and a bin number
template<bvxm_voxel_type VOX_T>
bvxm_voxel_grid_base_sptr bvxm_voxel_world::get_grid(unsigned bin_index)
{
    //retrieve map for current bvxm_voxel_type
    //if no map found create a new one
    if (grid_map_.find(VOX_T) == grid_map_.end())
    {
        //create map
        vcl_map<unsigned, bvxm_voxel_grid_base_sptr> bin_map;

        // look for existing appearance model grids in the directory
        vgl_vector_3d<unsigned int> grid_size = params_->num_voxels();
        vcl_string storage_directory = params_->model_dir();

        vcl_stringstream grid_glob;
        vcl_string fname_prefix = bvxm_voxel_traits<VOX_T>::filename_prefix();
```

```

grid_glob << storage_directory << "/" << fname_prefix << "*.vox";

//insert grids
for (vul_file_iterator file_it = grid_glob.str().c_str(); file_it; ++file_it) {
    vcl_string match_str = file_it.filename();
    unsigned idx_start = match_str.find_last_of('_') + 1;
    unsigned idx_end = match_str.find(".vox");
    vcl_stringstream idx_str;
    idx_str << match_str.substr(idx_start, idx_end - idx_start);
    int idx = -1;
    idx_str >> idx;
    if (idx < 0) {
        vcl_cerr << "error parsing filename " << file_it() << vcl_endl;
    } else {
        // create voxel grid and insert into map
        bvxm_voxel_grid_base_sptr grid =
            new bvxm_voxel_grid<bvxm_voxel_traits<VOX_T>::voxel_datatype>(file_it(), grid_size);
        bin_map.insert(vcl_make_pair((unsigned)idx, grid));
    }
}

grid_map_.insert(vcl_make_pair(VOX_T, bin_map));
}

//retrieve map containing voxel_grid
vcl_map<unsigned, bvxm_voxel_grid_base_sptr> voxel_map = grid_map_[VOX_T];

//retrieve voxel_grid for current bin
//if no grid exists at bin location create one filled with default values
if (voxel_map.find(bin_index) == voxel_map.end())
{
    vgl_vector_3d<unsigned int> grid_size = params_->num_voxels();
    vcl_string storage_directory = params_->model_dir();

    vcl_stringstream apm_fname;
    vcl_string fname_prefix = bvxm_voxel_traits<VOX_T>::filename_prefix();
    apm_fname << storage_directory << "/" << fname_prefix << "_" << bin_index << ".vox";

    typedef bvxm_voxel_traits<VOX_T>::voxel_datatype voxel_datatype;
    bvxm_voxel_grid<voxel_datatype> *grid =
        new bvxm_voxel_grid<voxel_datatype>(apm_fname.str(), grid_size);

    // fill grid with default value
    if (!grid->initialize_data(bvxm_voxel_traits<VOX_T>::initial_val())){
        vcl_cerr << "error initializing voxel grid" << vcl_endl;
        return bvxm_voxel_grid_base_sptr(0);
    }

    //Insert voxel grid into map
    bvxm_voxel_grid_base_sptr grid_sptr = grid;
    grid_map_[VOX_T].insert(vcl_make_pair(bin_index, grid_sptr));
}

return grid_map_[VOX_T][bin_index];
}

```

## Member Function Documentation:

### An example: pixel\_probability\_density

Many functions of the `voxel_world` are templated over the type of appearance model i.e. mixture of Gaussians grey scale, `APM_MOG_GREY`, or mixture of Gaussians color, `APM_MOG_RGB`. This makes the code generic and easy to expand to future appearance models. The following example illustrates this generic behavior.

```
template<bvxm_voxel_type APM_T>
bool bvxm_voxel_world::pixel_probability_density (bvxm_image_metadata const & observation,
vil_image_view< float > & pixel_probability, unsigned bin_index = 0)
```

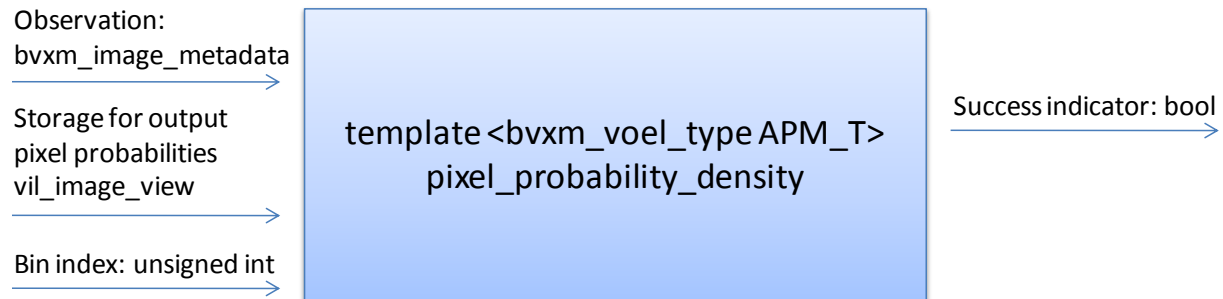


Figure 15: An example of member function of the `bvxm_voxel_world`

```
template<bvxm_voxel_type APM_T>
bool bvxm_voxel_world::pixel_probability_density(bvxm_image_metadata const& observation,
vil_image_view<float> &pixel_probability,
unsigned bin_index)
{
    // datatype for current appearance model
    typedef bvxm_voxel_traits<APM_T>::voxel_datatype apm_datatype;
    typedef bvxm_voxel_traits<OCCUPANCY>::voxel_datatype ocp_datatype;
    typedef bvxm_voxel_traits<APM_T>::obs_datatype obs_datatype;

    // the appearance model processor
    bvxm_voxel_traits<APM_T>::appearance_processor apm_processor;

    // check image sizes
    ...
    if ( (observation.img->ni() != pixel_probability.ni()) || (observation.img->nj() !=
pixel_probability.nj()) ) {
        vcl_cerr << "error: observation image size does not match input image size. " << vcl_endl;
    }

    vgl_vector_3d<unsigned int> grid_size = params_->num_voxels();
    ocp_datatype min_vox_prob = params_->min_occupancy_prob();
    ocp_datatype max_vox_prob = params_->max_occupancy_prob();

    // compute homographies from voxel planes to image coordinates and vise-versa.
    vcl_vector<vgl_h_matrix_2d<double> > H_plane_to_img;
    vcl_vector<vgl_h_matrix_2d<double> > H_img_to_plane;
    {
        vgl_h_matrix_2d<double> Hp2i, Hi2p;
```

```

    for (unsigned z=0; z < (unsigned)grid_size.z(); ++z)
    {
        bvxm_util::compute_plane_image_H(observation.camera, params_, z, Hp2i, Hi2p);
        H_plane_to_img.push_back(Hp2i);
        H_img_to_plane.push_back(Hi2p);
    }
}

// convert image to a voxel_slab
bvxm_voxel_slab<obs_datatype> image_slab(observation.img->ni(), observation.img->nj(), 1);
bvxm_util::img_to_slab(observation.img, image_slab);

...

// slabs for holding backprojections of visX
bvxm_voxel_slab<float> visX(grid_size.x(), grid_size.y(), 1);

bvxm_voxel_slab<obs_datatype> frame_backproj(grid_size.x(), grid_size.y(), 1);

// Pass 1: Get the color probabilities and compute total color and
// weight probabilities.
...

//get appearance model grid
bvxm_voxel_grid_base_sptr apm_grid_base = this->get_grid<APM_T>(bin_index);
bvxm_voxel_grid<apm_datatype> *apm_grid =
static_cast<bvxm_voxel_grid<apm_datatype>*>(apm_grid_base.ptr());

bvxm_voxel_grid<ocp_datatype>::const_iterator ocp_slab_it = ocp_grid->begin();
bvxm_voxel_grid<apm_datatype>::iterator apm_slab_it = apm_grid->begin();

for (unsigned z=0; z<(unsigned)grid_size.z(); ++z, ++ocp_slab_it, ++apm_slab_it)
{
    vcl_cout << ".";

    if ( (ocp_slab_it == ocp_grid->end()) || (apm_slab_it == apm_grid->end()) ) {
        vcl_cerr << "error: reached end of grid slabs at z = " << z << ". nz = " << grid_size.z()
<< vcl_endl;
        return false;
    }

    // backproject image onto voxel plane
    bvxm_util::warp_slab_bilinear(image_slab, H_plane_to_img[z], frame_backproj);

...

// calculate PI(X)
bvxm_voxel_slab<float> PI = apm_processor.prob_density(*apm_slab_it, frame_backproj);

...

// warp PIPX back to image domain
bvxm_util::warp_slab_bilinear(PIPX, H_img_to_plane[z], PIPX_img);

...

// fill pixel probabilities with preX_accum
vil_image_view<float>::iterator pix_prob_it = pixel_probability.begin();
bvxm_voxel_slab<float>::const_iterator preX_accum_it = preX_accum.begin();
for(; pix_prob_it != pixel_probability.end(); ++pix_prob_it, ++preX_accum_it) {
    *pix_prob_it = *preX_accum_it;
}

return true;
}

```

## Other Member Functions:

- *update*: This function is templated over appearance model type. It updates each voxel's Gaussian mixture and surface probability values with data from an image-camera pair.
- *update\_edges*: Updates the EDGE-type voxel grid with data from image/camera pair and return the edge probability density of pixel values.
- *expected\_image*: Calculates an expected image of the world from given viewing and illumination directions.
- *expected\_edge\_image*: Calculate the expected image of a world containing an EDGE voxel grid
- *inv\_pixel\_range\_probability*: Returns the probability that the observed pixels were **not** produced by a voxel in the grid.
- *pixel\_probability\_density*: Sum along the corresponding ray of voxels that the observation was produced by the voxel. This is based on algorithm published in Pollard + Mundy 06. The returned values are approximate samples of a probability density, with the pixel values being the independent value.
- *mixture\_of\_gaussians\_image*: Generates the mixture of Gaussians slab from the specified viewpoint. The slab should be allocated by the caller.
- *virtual\_view*: Produces a view of a video frame from a given viewpoint. Used for 3-D registration
- *fit\_plane*: Returns a planar approximation of the world.
- *get\_params*: Sets the world parameters equal to the given input.
- *set\_params*: Returns the world's parameters.
- *set\_grid*: Sets voxel grid equal to the given input.
- *get\_grid*: Returns the voxel grid
- *save\_occupancy\_raw*: Saves the occupancy grid in a ".raw" format readable by Drishti volume rendering software.
- *clean\_grids*: Removes all voxel data from disk
- *num\_observations*: Gets the observation count of a voxel type at a specific bin
- *increment\_observations*: Increments the observation count of a voxel type at a specific bin

## The Appearance Model Processors

There exist two processors in charge of performing probabilistic functions at the low level. The philosophy of the design is to have the processor traversing voxel slabs and calling a *bsta\_functor* at each data unit. At the functor level, is where probabilities are calculated, updated etcetera.

Two processors have been implemented

1. `bvxm_mog_grey_processor`: Manages a mixture of Gaussians appearance model for gray scale images.
2. `bvxm_mog_rgb_processor`: Manages for a mixture of Gaussians appearance model for color images.

Because processors are being used in a generic manner by `bvxm_voxel_world`, each processor must provide the following definitions:

```
typedef T apm_datatype; //e.g. bsta mixture of gaussian
typedef T obs_datatype; //e.g. float
typedef T obs_mathtype; //e.g. math type of the bsta mixture of gaussian
```

Where T is specific to each processor

Example of how *bsta\_functors* are being utilized:

```
//: Update with a new sample image
bool bvxm_mog_grey_processor::update( bvxm_voxel_slab<mix_gauss_type> &appear,
    bvxm_voxel_slab<float> const& obs,
    bvxm_voxel_slab<float> const& weight)
{
    // the model
    ...
    bsta_gauss_fl this_gauss(0.0f, init_variance);

    // the updater: bsta functor
    bsta_mg_grimson_weighted_updater<mix_gauss> updater(this_gauss,
        this->n_gaussian_modes_, g_thresh, min_variance);

    //check dimensions match
    assert(appear.nx() == obs.nx());
    assert(appear.ny() == obs.ny());
    assert(updater.data_dimension == appear.nz());
    //the iterators
    bvxm_voxel_slab<mix_gauss_type>::iterator appear_it;
    bvxm_voxel_slab<float>::const_iterator obs_it = obs.begin();
    bvxm_voxel_slab<float>::const_iterator weight_it = weight.begin();

    for(appear_it = appear.begin(); appear_it!= appear.end(); ++appear_it, ++obs_it, ++weight_it)
    {
        if (*weight_it > 0)
            updater(*appear_it, *obs_it, *weight_it);
    }
    return true
}
```

## Class Diagram

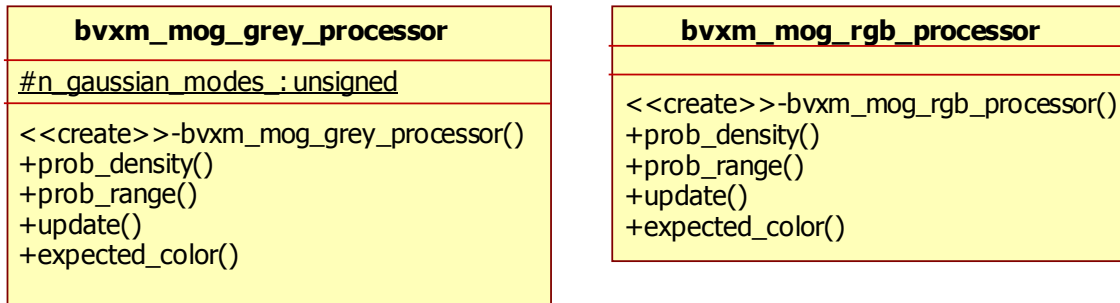


Figure 16: Appereance model procesors

## ii. BVXM Pro:

### The Process Architecture : bvxm\_pro

In this section, the high-level capabilities of the VoxGIS software are described. The modules presented in this section are processes managed by a singleton class: *bprb\_process\_manager*. Processes communicate via memory smart pointers that are managed by the in-memory data base: *brdb\_database*. The output of one process can be applied to the next process via retrieval from the database, where data items are tagged by a unique database id. The overall scheme is shown in the figure bellow.

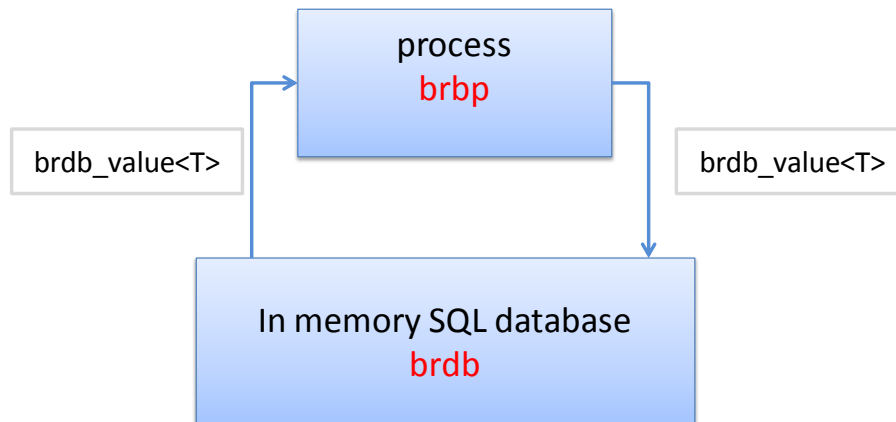


Figure 17: Process arquitecture

## Inputs and parameters:

Every process in *bvxml\_pro* inherits from *bprb\_process* (see *brpb* description in section II). Therefore, processes can have inputs and parameters. Inputs for a process reside in the database, whether parameters are specified in an xml file. The modules presented in the following section, show (on the left) what inputs **must** be passed to each process. Failure to do so will cause an error. On the other hand, if a parameter file is required, it is shown on the top of the module and in some cases an example of the parameter file will be given.

### BVXM processes

#### Create the 3-d voxel world

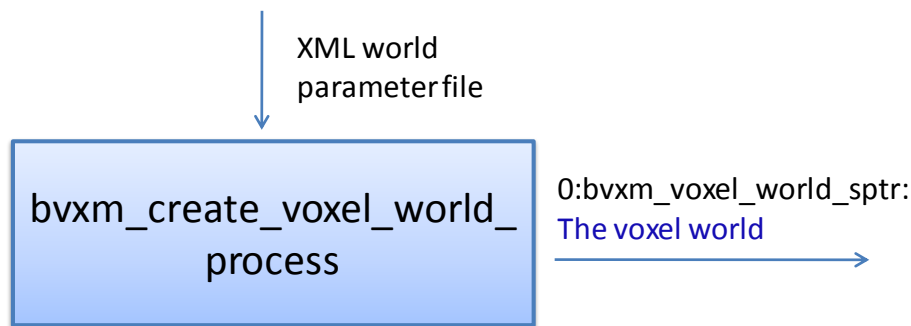


Figure 18: *bvxml\_create\_voxel\_world\_process*

This module takes no inputs. It creates the 3-d voxel world from scratch, based on a world parameter file. An example parameter file is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
- <CreateVoxelWorldProcess>
  <input_directory type="string" value="C:\ " />
  <corner_x type="float" value="0" />
  <corner_y type="float" value="0" />
  <corner_z type="float" value="0" />
  <voxel_dim_x type="unsigned" value="50" />
  <voxel_dim_y type="unsigned" value="50" />
  <voxel_dim_z type="unsigned" value="5" />
  <voxel_length type="float" value="10" />
  <lvcs type="string" value="world_model_params_lvcs.txt" />
  <apm_type type="unsigned" value="1" />
  <min_ocp_prob type="float" value="0.001" />
  <max_ocp_prob type="float" value="0.5" />
</CreateVoxelWorldProcess>
```

Figure 19: World Parameters example XML file



The voxel world is geographically defined by a local vertical coordinate frame (LVCS). The parameters XML file expects a text file path for the LVCS. LVCS class (bgeo\_lvcs) has a read (file) method that parses the parameters. A typical text file for the LVCS is shown below; please refer to the section on bgeo\_lvcs for class details and input file format:

<b>world_model_params_lvcs.txt</b>
<b>wgs84</b>
<b>meters</b>
<b>degrees</b>
<b>32.7242 -117.156 39.0</b>
<b>0 0</b>
<b>0 0 0</b>

The world parameters which are defined in the xml file (Figure 19) have the following defaults (the defaults are used if the current value is not defined in the parameter file):

<b>parameter</b>	<b>Default value</b>
<b>input_directory</b>	<b>“./”</b>
<b>corner_x</b>	<b>0.0</b>
<b>corner_y</b>	<b>0.0</b>
<b>corner_z</b>	<b>0.0</b>
<b>voxel_dim_x</b>	<b>10</b>
<b>voxel_dim_y</b>	<b>10</b>
<b>voxel_dim_z</b>	<b>10</b>
<b>voxel_length</b>	<b>1</b>
<b>lvcs</b>	<b>“./”</b>
<b>min_ocp_prob</b>	<b>1e-5f</b>
<b>max_ocp_prob</b>	<b>1-1e-5f</b>

### Initialize the incoming image region of interest

The main module in this step is *bvxm\_roi\_init\_process*. However, before defining the region of interest, the desired rational camera needs to be loaded into the database. Currently, there are two processes available for loading rational cameras: *vppl\_load\_rational\_camera\_process* and *vppl\_load\_rational\_camera\_nitf\_process*. The first one gives the user the flexibility to load a camera file (e.g a manually corrected camera); the latter extracts the rational camera from the NITF header. These two processes reside in *vxl/contrib/brl/bpro/core/vppl\_pro* and are illustrated bellow.

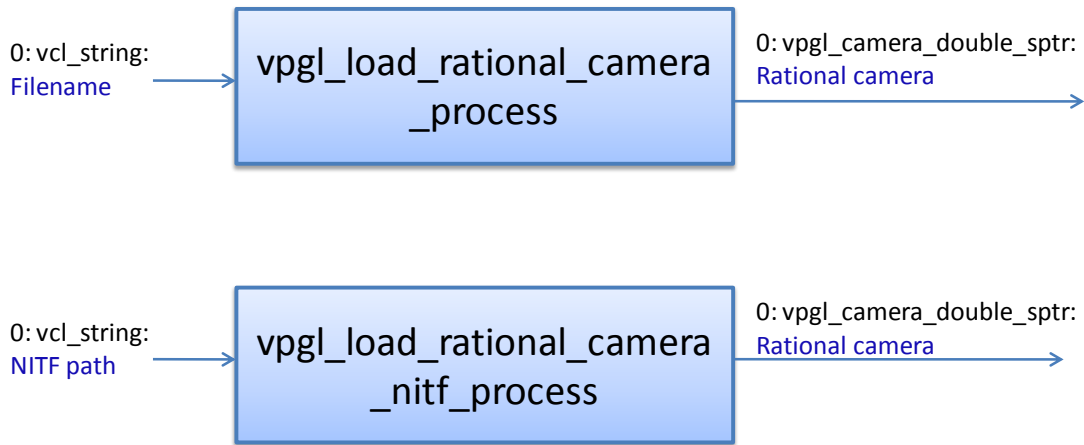


Figure 19: *vppl* processes to load rational cameras

Once the rational camera has been loaded into the database, it can be used as an input for the following process:

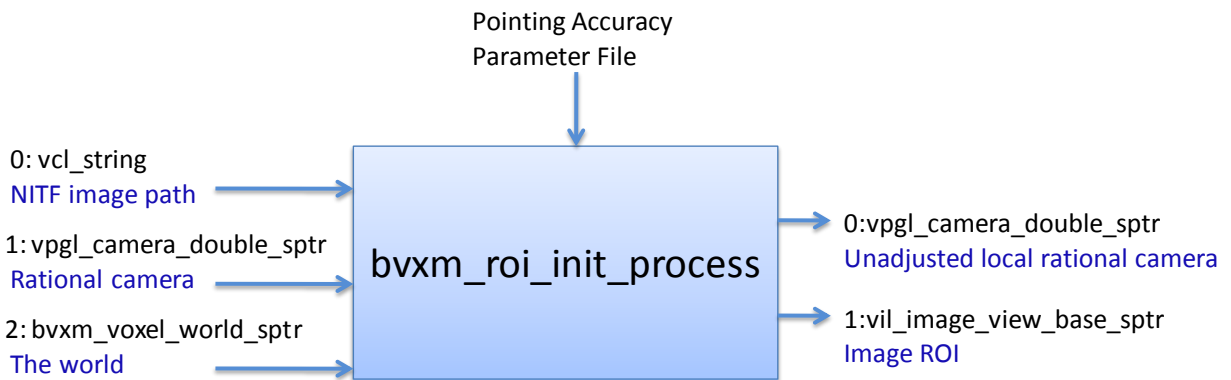


Figure 20: *bvxm\_roi\_init\_process*

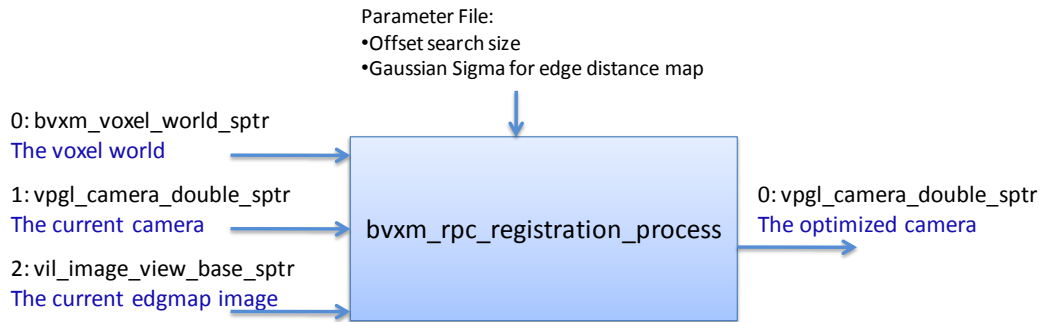
This module takes an existing voxel world, an NITF image path and the camera associated with the image. It produces the following two outputs:

1) A clipped image which is guaranteed to span the projection of the voxel world in the image. It is assumed that the input image will be significantly larger than the size used in the experiments.

2) A modified camera class instance (RPC) that reflects the offset due to clipping. The specialized type is *vpgl\_local\_rational\_camera<double>\** where it utilizes the local vertical coordinate system of the voxel world.

The execute method of *bvxm\_roi\_init\_process* creates rational cameras in the pointing accuracy span to project the voxel world onto the image plane. The pointing accuracy (e) span is defined as a box around the camera offset (c) where min point is (c.x-e, c.y-e, c.z-e) and max point is (c.x+e, c.y+e, c.z+e). Each camera, at the corners of the box, projects the voxel world and the end result is the bounding box of the union of these individual 2D regions. The clipped image is reduced to the pixel format byte with the same number of planes of the original image.

#### Register the new image by a translation of the image



**Figure 21: bvxm\_rpc\_registration\_process**

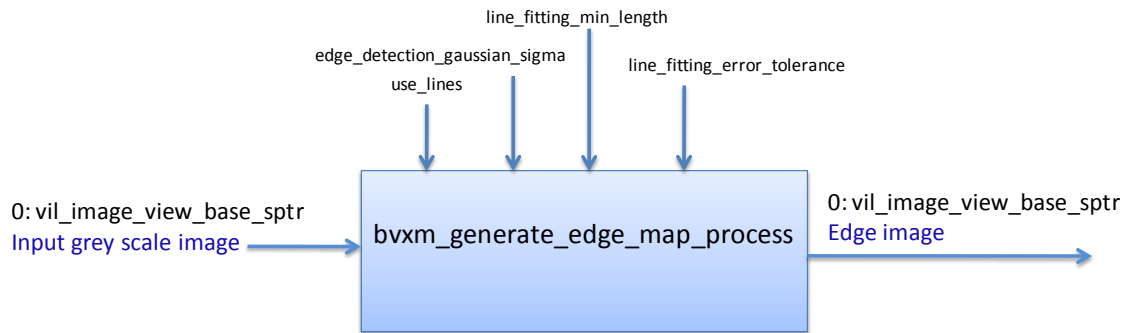
This process is designed to correct rational camera offsets by using a 3D voxel model and binary edge images created by “*bvxm\_generate\_edge\_map\_process*”. The voxel model used here keeps the probability of each voxel being occupied by a 3D edge. This process has two independent steps. The first step is the correction of rational camera offsets of the new image. An expected edge probability image is calculated by projecting the edge probability voxel model to the new image. This is followed by a simple optimization procedure to estimate the correct image offsets by maximizing the probability of observing the new edge image given the expected edge probability image. The second step is to update the voxel model using the new edge image

and the corrected rational camera. This is done by projecting each voxel onto the new image and updating the voxel edge probability using the pixel edge probability in the new image.

The process has 2 parameters:

- `cedt_image_gaussian_sigma` (2.0): gaussian sigma for the edge distance map
- `offset_search_size` (20): maximum expected error in the rpc image offset

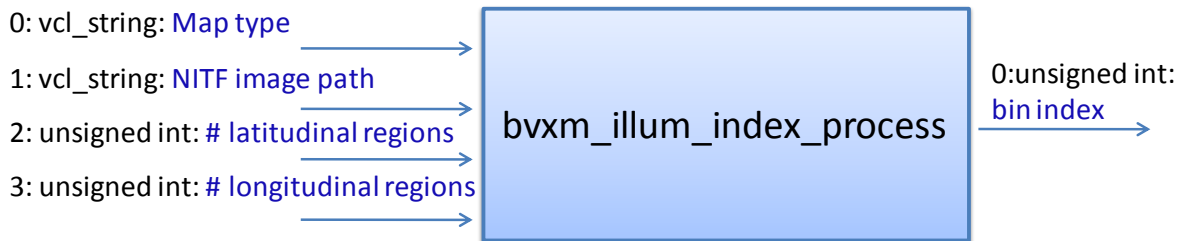
### Find the Edges in the Image



**Figure 22: bvxm\_generate\_edge\_map\_process**

This process is an interface for creating a binary edge image from an input grayscale image. There are two execution modes for this process. In the first execution mode, edges are detected using Van Duc's Canny edge detection method and then contours are formed (by edge following) in order to construct a topological network. The second mode detects the edges using the same approach, but it also detects 2D line segments in the edges using an incremental line fitting method.

### Determine appropriate Gaussian mixture for updating



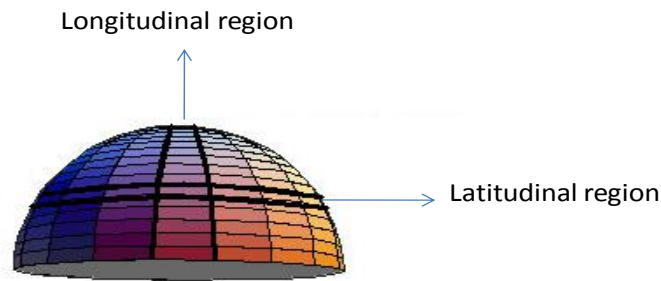
**Figure 23: bvxm\_illum\_index\_process**

Since image appearance in EO is strongly dependent on image illumination direction, the current update algorithm maintains a number of different Gaussian mixtures that are active for a

particular range of azimuth and elevation of the sun. The Gaussian mixtures are indexed by a bin number output by this process.

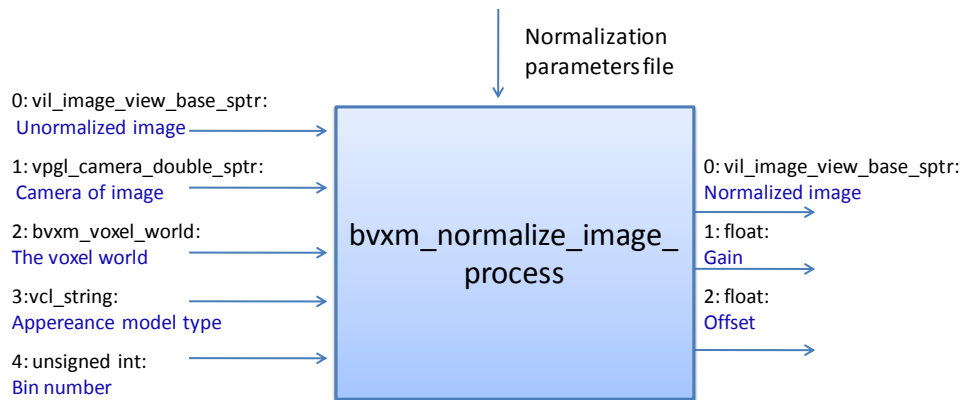
There are two types of algorithm to discretize the illumination sampling space, one of these options must be specified in the input field *Map Type*:

1. *10bins\_radial\_1d*: This algorithm discretizes, in 10 radial regions, the ellipse obtained by projecting the hemisphere containing all illuminations directions onto the xy-plane.
2. *equal\_area*: In this algorithm, the number of bins is determined by the user. It uses a hemisphere as the space where all possible illumination directions are distributed. Such space is discretized in equal area regions based on the number of longitudinal and latitudinal section given in the input



**Figure 24. Illumination Grid**

### Image normalization



**Figure 25: bvxm\_normalize\_image\_process**

There can be considerable variation in EO image appearance due to haze and other atmospheric effects. To reduce this source of variability, each image is adjusted in gain and offset to produce as nearly similar global appearance of the image as possible. The gain and offset pa-

rameters are found by maximizing the total probability of the image, with respect to the current appearance states of the voxels.

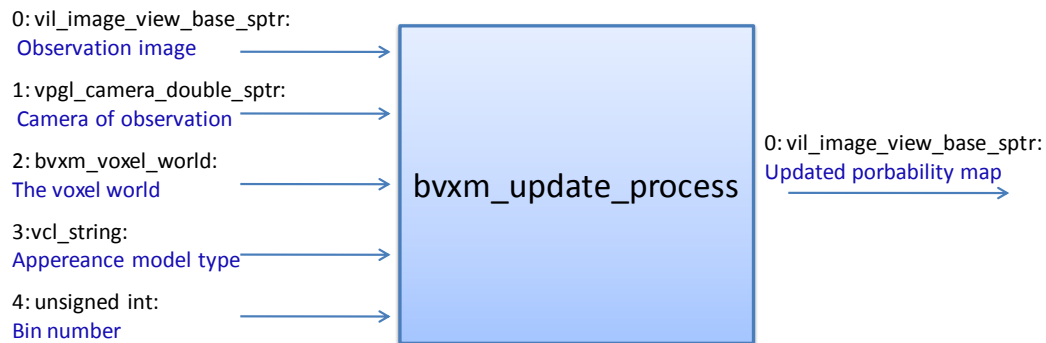
The **appearance model type** must be one of the following:

- `bvxm_apm_mog_grey`
- `bvxm_apm_mog_rgb`

The **bin\_index** has to be specified. If multiple illumination directions are not used, the bin number should be zero.

The output is a radiometrically normalized image.

#### Update the appearance distributions in each voxel



**Figure 26: bvxm\_update\_process**

This process is the key step in the learning process. It accepts a normalized and registered image and updates each voxel's Gaussian mixture and surface probability values. It is necessary to specify what type of appearance model should be used by the updater with one of the following strings:

- `bvxm_apm_mog_grey`
- `bvxm_apm_mog_rgb`

In addition the **bin index** of the model needs to be specified. If multiple illumination directions are not used, the bin number should be zero.

### Change detection

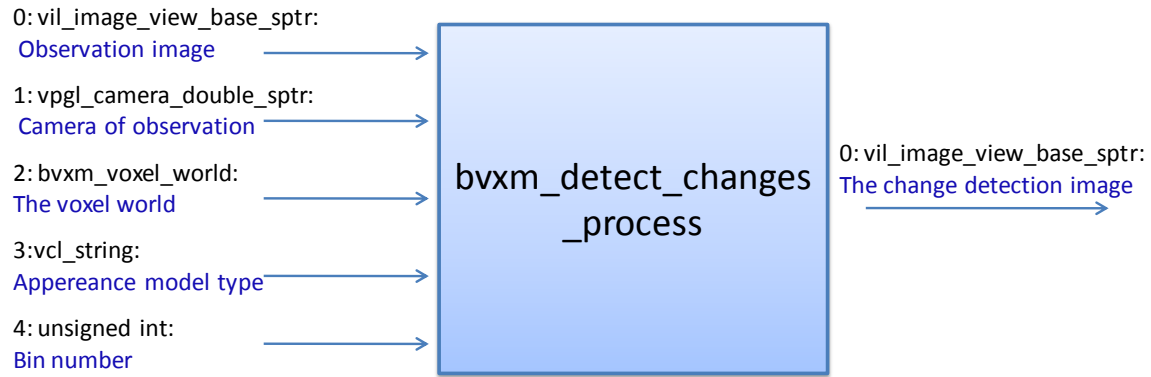


Figure 27: `bvbm_detect_changes_process`

After the voxel world has been updated with the training images, this module is used to detect changes. It takes an observation image, the voxel world and it outputs an image containing the probability of change

The **appearance model type** must be one of the following:

- `bvbm_apm_mog_grey`
- `bvbm_apm_mog_rgb`

The **bin\_index** has to be specified. If multiple illumination directions are not used, the bin number should be zero.

### Produce the change detection result

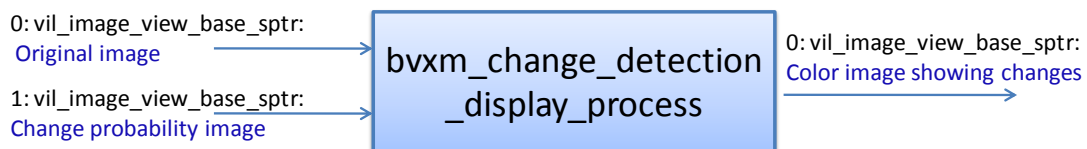
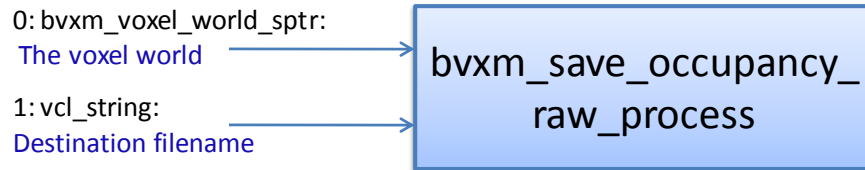


Figure 28: `bvbm_change_detection_display_process`

This module is the final process in the overall chain. It takes an image representing the probability of change at each pixel and transforms it into a red coded change image that is somewhat similar to two-color multi-view.

## Miscellaneous processes to visualize output

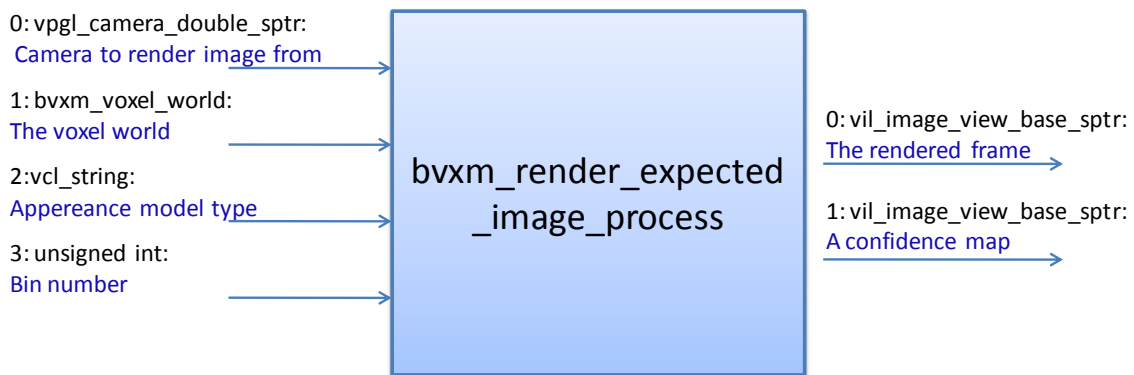
### Saving the occupancy grid



**Figure 29: bvxm\_save\_occupancy\_raw\_process**

This process saves the voxel world occupancy grid in a binary format that is readable by the Drishti volume rendering program (<http://anusf.anu.edu.au/Vizlab/drishti/>)

### Rendering the expected image

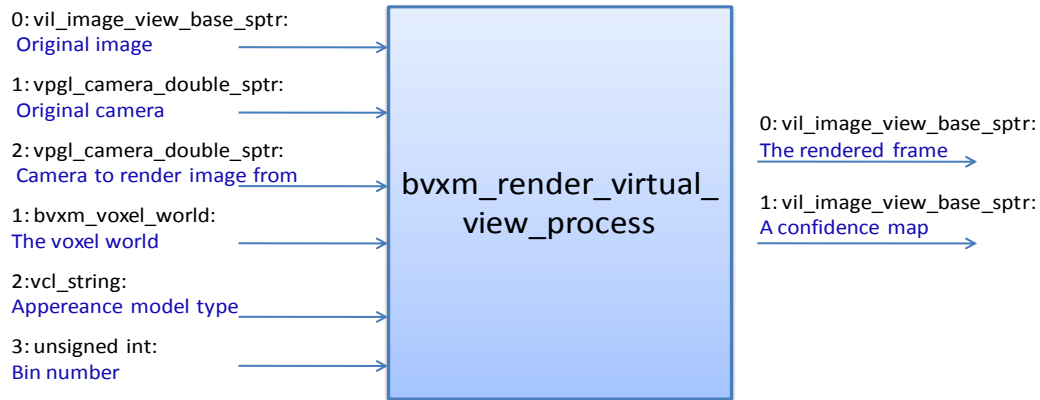


**Figure 30: bvxm\_render\_expected\_image\_process**

This process renders an expected image of the world from given viewing and illumination directions.



## Rendering a virtual view: 3-D Registration tool



**Figure 31: `bvxml_render_virtual_view_process`**

This process was created as a 3-D registration tool that allows rendering a video frame from a given viewpoint.

## IV. BVXM\_BATCH

This library provides a flexible interface for running *bvxml* processes in batch mode. *bvxml\_batch* uses Python as the scripting language. Therefore, a set of Python wrappers have been implemented to allow interpretive execution of the modules via calls to the *brpb\_batch\_process\_manager*.

There are two classes:

- *batch\_bvxml*: This is where all Python's external declarations reside.
- *reg\_bvxml*: All processes and data types need to be registered in order to be recognized by the *brpb\_batch\_process\_manager*. Registration takes care of creating instances of each process defined, as well as a database relation for different data type registered. Therefore if we run the simple python commands

```
import bvxml_batch
bvxml_batch.register_processes();
bvxml_batch.register_datatypes();
bvxml_batch.print_db();
```

We will observe the following empty relations in the database.

[illegible]**byxm\_batch methods:**

These functions are the interface between Python and c++. Once the user imports *bvxm\_batch* into Pyhton, all of these functions become accessible.

- *register\_processes()*: Create instances of each defined process
- *register\_datatypes()*: Insert tables in the database for each type
- *init\_process(string “ProcessName”)*: Creates a new process instance by the name specified in “ProcessName”
- *set\_input\_bool(unsigned i, bool input)*: Sets input i on current process to a bool value.
- *set\_input\_string(unsigned i, string input)*: Sets input i on current process to a string value.
- *set\_input\_int(unsigned i, int input)*: Sets input i on current process to an int value.
- *set\_input\_long(unsigned i, long input)*: Sets input i on current process to a long value.
- *set\_input\_float(unsigned i, float input)*: Sets input i on current process to a float value.
- *set\_input\_double(unsigned i, double input)*: Sets input i on current process to a double value.

- *process\_print\_default\_params(string "ProcessName", string "ParamName")*: Prints the default values of the process by name.
- *set\_params\_process(string "XMLpath")*: Sets the parameter values of the current process from the XML file
- *run\_process()*: Runs the current process
- *commit\_output(unsigned i)*: Puts output i in the database
- *set\_input\_from\_db(unsigned i, unsigned id)*: Sets input i of the current process to data base id value
- *print\_db()*: Prints the database
- *clear()*: Clears the database tables

## Running *bvxm* pipeline

VoxGIS software can be run from a Python executable. *bvxm\_batch* has been configured to be importable as a python module (.py file). Therefore, once inside python the first step consists of importing *bvxm\_batch* module with the command:

```
import bvxm_batch
```

After this command is executed, all *bvxm\_batch* methods are accessible

The following Python script accompanied by explanations serves as an outline of the entire *bvxm* pipeline.

1. Import *bvxm\_batch* module and register data types and processes needed for VoxGIS software

```
import bvxm_batch
bvxm_batch.register_processes();
bvxm_batch.register_datatypes();
bvxm_batch.print_db();
```

2. Create the voxel world: This is the first process in the pipeline. It creates a voxel world with the parameters given in *world\_model\_params.xml*. After the process is initialized by name and the parameter are set from file, the process is executed and the output (*bvxm\_voxel\_world*) is committed to the database under the id *voxel\_world\_id*. This id, is "kept around" inside the script to be passed as an input in future processes.

```
print("Creating Voxel World");
bvxm_batch.init_process("CreateVoxelWorldProcess");
```

```

bvxm_batch.set_params_process("./world_model_params.xml");
bvxm_batch.run_process();
voxel_world_id = bvxm_batch.commit_output(0);

```

3. Retrieve image filenames as well as cameras filenames

```

f=open('./full_hiafa_images.txt', 'r')
image_fnames=f.readlines();
f.close();

```

```

f=open('./full_hiafa_cam.txt', 'r')
cam_fnames=f.readlines();
f.close();

```

4. For all desired images, run the processes of VoxGIS software to either update the world or detect changes

```

i=0;
for fname in image_fnames[0:25]:

```

5. Get camera name

```

image_filename=fname[:-1];
cam_name=cam_fnames[i];
cam_name=cam_name[:-1];

```

6. Load rational camera using *vppl\_load\_rational\_camera\_process*. Alternatively, the user could load the rational camera directly from the NITF header using *vppl\_load\_rational\_camera\_nitf\_process* (see figure 19 for details on this process).

```

print ("Load Camera");
bvxm_batch.init_process("LoadRationalCameraProcess");
bvxm_batch.set_input_string(0,cam_name);
bvxm_batch.run_process();
cam_id = bvxm_batch.commit_output(0);

```

7. Initialize region of interest base on the parameters specified in *roi.params.xml* for the current image/camera pair and voxel world

```

print("Initializing ROI");

```

```

bvxm_batch.init_process("bvxmRoiInitProcess");
bvxm_batch.set_params_process("./roi_params.xml");
bvxm_batch.set_input_string(0,image_filename);
bvxm_batch.set_input_from_db(1,cam_id);
bvxm_batch.set_input_from_db(2,voxel_world_id);
bvxm_batch.run_process();
cam_id = bvxm_batch.commit_output(0);i
mage_id = bvxm_batch.commit_output(1);
curr_image_id=image_id;
curr_cam_id=cam_id;

```

#### 8. Save roi image for debugging processes

```

print("Saving Image");
bvxm_batch.init_process("SaveImageViewProcess");
bvxm_batch.set_input_from_db(0,image_id);
bvxm_batch.set_input_string(1,"./ini"+str(i)+".png");
bvxm_batch.run_process();

```

#### 9. Only after a certain number of training images (in this case two) camera correction is performed using edge detection.

```

if i>2:

```

#### 10. Generate the edge map of current image and save it to disk

```

print("Edge Detection");
bvxm_batch.init_process("bvxmGenerateEdgeMapProcess");
bvxm_batch.set_params_process("./edge_map_params.xml");
bvxm_batch.set_input_from_db(0,image_id);
vbm_batch.run_process();
edge_map_id = bvxm_batch.commit_output(0);

print("Saving Edgemap");
bvxm_batch.init_process("SaveImageViewProcess");
bvxm_batch.set_input_from_db(0,edge_map_id);
bvxm_batch.set_input_string(1,"./emap.png");
bvxm_batch.run_process();

```

#### 11. Correct rational camera offset

```

print("RPC Correction");
bvxm_batch.init_process("bvxmRpcRegistrationProcess");
bvxm_batch.set_params_process("./rpc_registration_parameters.xml");
bvxm_batch.set_input_from_db(0,voxel_world_id);
bvxm_batch.set_input_from_db(1,cam_id);
bvxm_batch.set_input_from_db(2,edge_map_id);
bvxm_batch.run_process();
corrected_cam_id = bvxm_batch.commit_output(0);

```

12. Obtain the clipped image containing projection of voxel world as well as the camera reflecting the offset changes.

```

print("Get Corrected ROI");
bvxm_batch.init_process("bvxmRoiInitProcess");
bvxm_batch.set_params_process("./corrected_roi_params.xml");
bvxm_batch.set_input_string(0,image_filename);
bvxm_batch.set_input_from_db(1,curr_cam_id);
bvxm_batch.set_input_from_db(2,voxel_world_id);
statuscode=bvxm_batch.run_process();
if statuscode:
curr_cam_id = bvxm_batch.commit_output(0);
orig_image_id = bvxm_batch.commit_output(1);
print("Saving Image");
bvxm_batch.init_process("SaveImageViewProcess");
bvxm_batch.set_input_from_db(0,orig_image_id);
bvxm_batch.set_input_string(1,"./ini"+str(i)+".png");
bvxm_batch.run_process();
curr_image_id=orig_image_id;

```

13. At this stage of the pipeline, cameras and images have been corrected. This step obtains the illumination bin number, based on the elevation and azimuth in the NITF header, and the number of regions given by the user.

```

map_type="eq_area";
print("Illumination Index");
bvxm_batch.init_process("bvxmIllumIndexProcess");
bvxm_batch.set_input_string(0,map_type);
bvxm_batch.set_input_string(1,image_filename);
bvxm_batch.set_input_unsigned(2,8);

```

```

bvxm_batch.set_input_unsigned(3,0);b
vxm_batch.run_process();
bin_id = bvxm_batch.commit_output(0);a
pp_type="apm_mog_grey";

```

#### 14. Normalize image intensity

```

print(" Normalizing Image ");
bvxm_batch.init_process("bvxmNormalizeImageProcess");
bvxm_batch.set_params_process("./normalize.xml");
bvxm_batch.set_input_from_db(0,curr_image_id);
bvxm_batch.set_input_from_db(1,curr_cam_id);
bvxm_batch.set_input_from_db(2,voxel_world_id);
bvxm_batch.set_input_string(3,app_type);
bvxm_batch.set_input_from_db(4,bin_id);
bvxm_batch.run_process();
normalized_img_id = bvxm_batch.commit_output(0);
float1_id = bvxm_batch.commit_output(1);
float2_id = bvxm_batch.commit_output(2);

```

15. Update the world with the current image. If instead of updating the voxel world, the user wishes to detect changes, the process *bvxm\_detect\_changes* should be called here.

```

print("Updating World");
bvxm_batch.init_process("bvxmUpdateProcess");
bvxm_batch.set_input_from_db(0,curr_image_id);
bvxm_batch.set_input_from_db(1,curr_cam_id);
bvxm_batch.set_input_from_db(2,voxel_world_id);
bvxm_batch.set_input_string(3,app_type);
bvxm_batch.set_input_from_db(4,bin_id);
bvxm_batch.run_process();
out_img_id = bvxm_batch.commit_output(0);
mask_img_id = bvxm_batch.commit_output(1);

```

#### 16. Display Changes

```

print("Display changes");
bvxm_batch.init_process("bvxmChangeDetectionDisplayProcess");
bvxm_batch.set_params_process("./change_display_params.xml");

```

```

bvxm_batch.set_input_from_db(0,orig_image_id);
bvxm_batch.set_input_from_db(1,out_img_id);
bvxm_batch.set_input_from_db(2,mask_img_id);
bvxm_batch.run_process();
change_img_id = bvxm_batch.commit_output(0);

print("Saving Image");
bvxm_batch.init_process("SaveImageViewProcess");
bvxm_batch.set_input_from_db(0,change_img_id);
bvxm_batch.set_input_string(1,"./change"+str(i)+".png");
bvxm_batch.run_process();

```

17. Save occupancy probability of the world. This step is optional.

```

print("Writing World");
bvxm_batch.init_process("bvxmSaveOccupancyRaw");
bvxm_batch.set_input_from_db(0,voxel_world_id);
bvxm_batch.set_input_string(1,"./world.raw");
bvxm_batch.run_process();

```

18. Run processes on next image

```

i = i + 1;
reply

```

## V. Additional Documentation

### Tests

Testing code has been written for all the libraries described above. Under each library there is a folder containing testing code for all the classes. This tests programs guarantee the proper functioning of classes and serve as a good example of how classes are used.

### VXL libraries

In addition to the libraries just discussed there are dozens more that need to be mastered in order to fully understand the code base. The following is a list of some of the key libraries. Many of these libraries are documented in the VXL “book” on the VXL website:













<http://vxl.sourceforge.net/>

- vcl – compatibility library; allows seamless use of C++ across a patchwork quilt of compilers and operating systems
- vbl – basics; ref counting and smart pointers



- vnl – numerics library; vector, matrix and numerical computations, optimization, linear algebra...
- vil – image library; read, write, process images, templated over pixel type
- vgl – geometry and projective transformations of geometry (vgl/algo)
- vppl – photogrammetry library; camera classes, rational polynomial model, back-projection.
- bsta – statistics; general distributions, Gaussian, Gaussian mixture, probability functors
- bsta/algo – algorithms using distributions
- bbgm – normalcy modeling using bsta. (This library is not directly involved in the delivery but is a 2-d version of the 3-d voxel world and has many similarities to the code being developed for the voxel model). Understanding how this library works using functor-style programming, will provide valuable insight into the code to be delivered.

The best way to learn is to download the latest (bleeding edge) version of VXL and build the libraries and test cases and then modify the test cases to get them to do something different to test your understanding. It is advised to check out the dashboard to be sure there isn't a lot of compile problems. Here is what the dashboard looks like right now (all green compilations which is good).

Site	Build Name	Update	Cfg	Build		Test		
				Error	Warn	NotRun	Fail	Pass
GE	Cygwin_gcc-3.4.4_-O0_static 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">18</a>	<a href="#">0</a>	<a href="#">2</a>	<a href="#">656</a>
GE	FreeBSD-6.2_gcc-3.4.6_-O2_shared 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">86</a>	<a href="#">0</a>	<a href="#">3</a>	<a href="#">655</a>
GE	FreeBSD-6.2_gcc-4.0.4_-O2_shared 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">140</a>	<a href="#">0</a>	<a href="#">3</a>	<a href="#">655</a>
GE	FreeBSD-6.2_gcc-4.2.3_-O2_shared 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">122</a>	<a href="#">0</a>	<a href="#">4</a>	<a href="#">654</a>
GE	FreeBSD-7.0_gcc-4.2.1_profile_static 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">19</a>	<a href="#">0</a>	<a href="#">2</a>	<a href="#">651</a>
cs.rpi.edu	FreeBSD-fresh-gcc-3.4.2 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">6</a>	<a href="#">649</a>
lems.brown.edu	Linux-2.6_gcc-4.1.3_-Wall 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">244</a>	<a href="#">0</a>	<a href="#">2</a>	<a href="#">655</a>
lems.brown.edu	Linux-2.6.18_gcc-4.1.2 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">3</a>	<a href="#">654</a>
imorphics.com	Linux-2.6.22_gcc-4.1.2_RelWithDebInfo 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">1</a>	<a href="#">450</a>
GE	Linux-2.6.9_icc-9.1-64bit_-O2_shared							
GE	MinGW-3.8_gcc-3.4.4_-O2_static 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">1315</a>	<a href="#">0</a>	<a href="#">3</a>	<a href="#">655</a>
GE	Win2k_bcc-5.5.1_Release 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">5</a>	<a href="#">246</a>
GE	Win2k_msvc-7.1_Debug 	<a href="#">10</a>	<a href="#">0</a>	<a href="#">0</a>	<a href="#">102</a>	<a href="#">0</a>	<a href="#">2</a>	<a href="#">656</a>
imorphics.com	WinXP_msvc-7.1_RelWithDebInfo							