Brown University

VoxGIS Software Documentation

J.L Mundy, Isabel Restrepo, Gamze Tunali

March 2008

TABLE OF CONTENTS

I.   Introduction

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

II.   Auxiliary Libraries

i.   <u>Brown Relational Database Library: brdb</u> *(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***.**
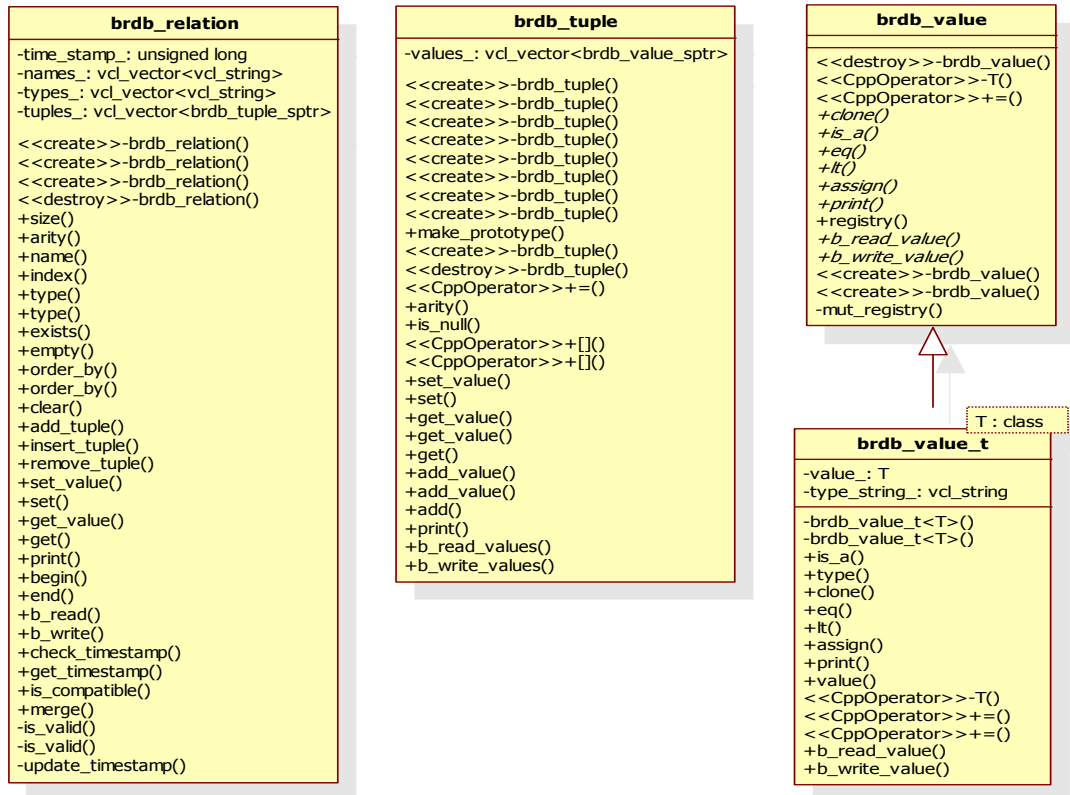
Class Diagram:
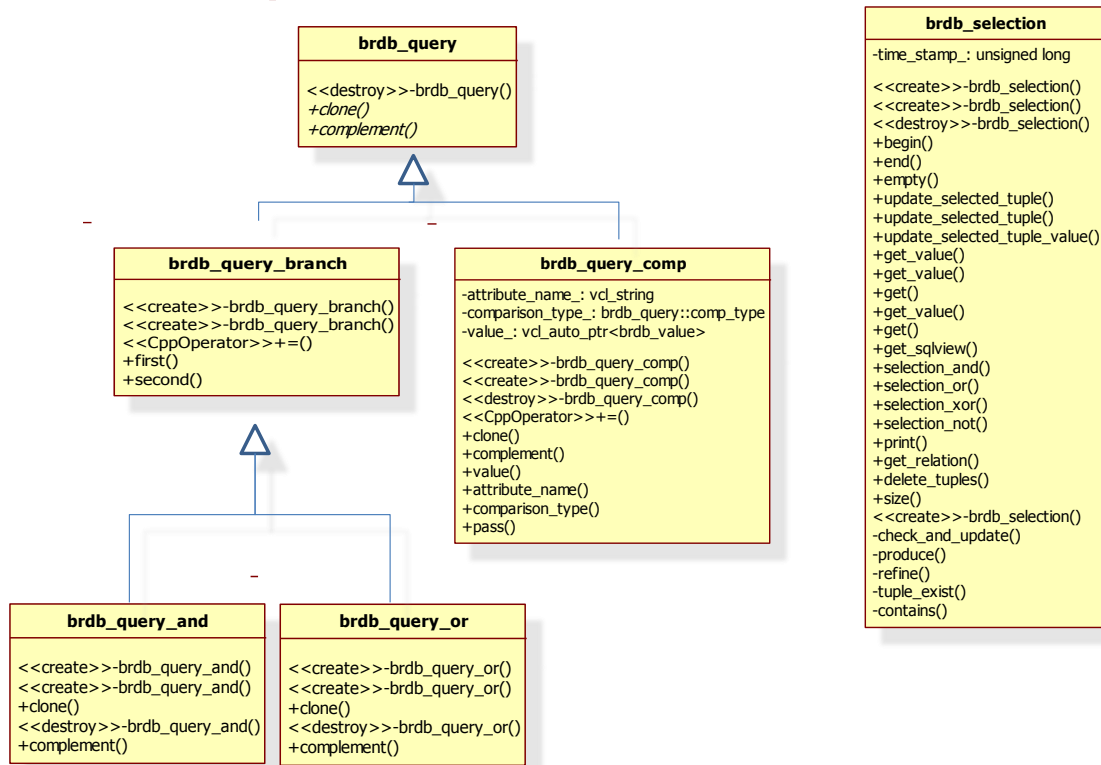

Figure 1. Database Objects :Relations, tuples and valued

brdb_query

<<destroy>>-brdb_query()
+clone()
+complement()

brdb_query_branch

<<create>>-brdb_query_branch()
<<create>>-brdb_query_branch()
<<CppOperator>>+=()
+first()
+second()

brdb_query_comp

-attribute_name_: vcl_string
-comparison_type_: brdb_query::comp_type
-value_: vcl_auto_ptr<brdb_value>

<<create>>-brdb_query_comp()
<<create>>-brdb_query_comp()
<<destroy>>-brdb_query_comp()
<<CppOperator>>+=()
+clone()
+complement()
+value()
+attribute_name()
+comparison_type()
+pass()

brdb_query_and

<<create>>-brdb_query_and()
<<create>>-brdb_query_and()
+clone()
<<destroy>>-brdb_query_and()
+complement()

brdb_query_or

<<create>>-brdb_query_or()
<<create>>-brdb_query_or()
+clone()
<<destroy>>-brdb_query_or()
+complement()

brdb_selection

-time_stamp_: unsigned long

<<create>>-brdb_selection()
<<create>>-brdb_selection()
<<destroy>>-brdb_selection()
+begin()
+end()
+empty()
+update_selected_tuple()
+update_selected_tuple()
+update_selected_tuple_value()
+get_value()
+get_value()
+get()
+get_value()
+get()
+get_sqlview()
+selection_and()
+selection_or()
+selection_xor()
+selection_not()
+print()
+get_relation()
+delete_tuples()
+size()
<<create>>-brdb_selection()
-check_and_update()
-produce()
-refine()
-tuple_exist()
-contains()

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

brdb_database

-relations_: vcl_map<vcl_string, brdb_relation_sptr>

<<create>>-brdb_database()
<<create>>-brdb_database()
+size()
+clear()
+exists()
+exists()
+empty()
+add_tuple()
+remove_relation()
+clear_relation()
+add_relation()
+get_relation()
+print()
+get_all_relation_names()
+join()
+join()
+select()
+merge()
+b_write()
+b_read()

brdb_database_manager

-id_: unsigned

+instance()
+id()
+clear_all_except()
+remove_all_except()
+save_database()
+load_database()
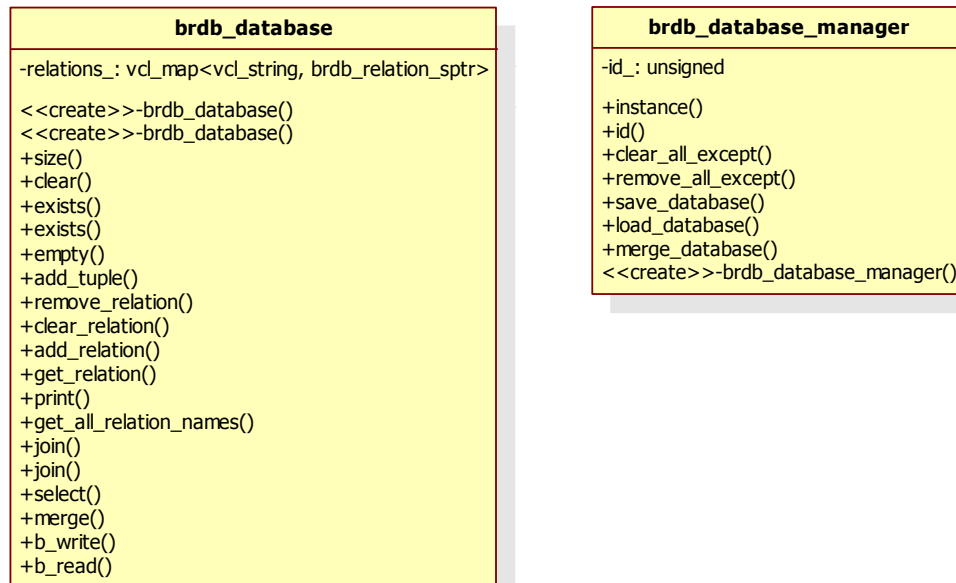+merge_database()
<<create>>-brdb_database_manager()

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

ii.   Brown Process Basics Library: brpb *(vxl/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*
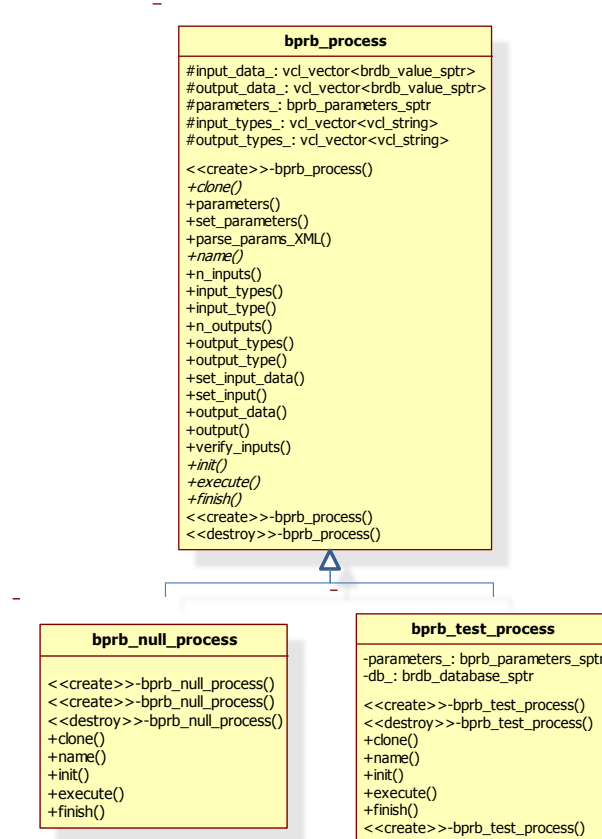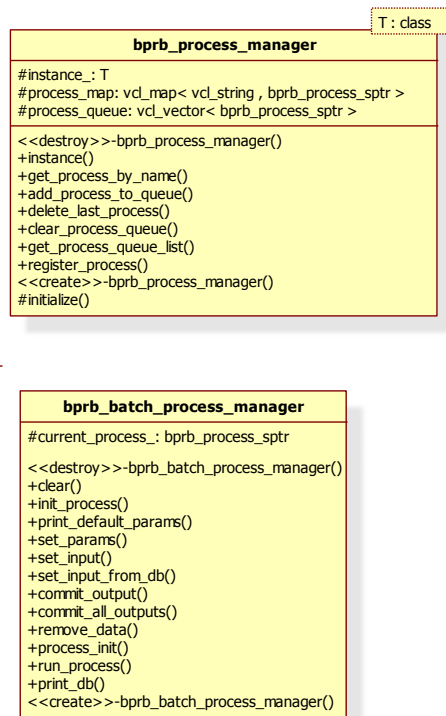 - *bprb_param_type*

Class Diagram:

T : class

bprb_process_manager

#instance_: T
#process_map: vcl_map< vcl_string , bprb_process_sptr >
#process_queue: vcl_vector< bprb_process_sptr >

<<destroy>>-bprb_process_manager()
+instance()
+get_process_by_name()
+add_process_to_queue()
+delete_last_process()
+clear_process_queue()
+get_process_queue_list()
+register_process()
<<create>>-bprb_process_manager()
#initialize()

bprb_batch_process_manager

#current_process_: bprb_process_sptr

<<destroy>>-bprb_batch_process_manager()
+clear()
+init_process()
+print_default_params()
+set_params()
+set_input()
+set_input_from_db()
+commit_output()
+commit_all_outputs()
+remove_data()
+process_init()
+run_process()
+print_db()
<<create>>-bprb_batch_process_manager()

bprb_process

#input_data_: vcl_vector<brdb_value_sptr>
#output_data_: vcl_vector<brdb_value_sptr>
#parameters_: bprb_parameters_sptr
#input_types_: vcl_vector<vcl_string>
#output_types_: vcl_vector<vcl_string>

<<create>>-bprb_process()
+clone()
+parameters()
+set_parameters()
+parse_params_XML()
+name()
+n_inputs()
+input_types()
+input_type()
+n_outputs()
+output_types()
+output_type()
+set_input_data()
+set_input()
+output_data()
+output()
+verify_inputs()
+init()
+execute()
+finish()
<<create>>-bprb_process()
<<destroy>>-bprb_process()

bprb_null_process

<<create>>-bprb_null_process()
<<create>>-bprb_null_process()
<<destroy>>-bprb_null_process()
+clone()
+name()
+init()
+execute()
+finish()

bprb_test_process

-parameters_: bprb_parameters_sptr
-db_: brdb_database_sptr

<<create>>-bprb_test_process()
<<destroy>>-bprb_test_process()
+clone()
+name()
+init()
+execute()
+finish()
<<create>>-bprb_test_process()

Figure 4: *bprb* **Process Managers**   **Figure 5:** *bprb* **Processes**

bprb_param

#has_bounds_: bool
#name_: vcl_string
#description_: vcl_string

<<destroy>>-bprb_param()
+clone()
+name()
+description()
+has_bounds()
+reset()
+set_from_temp()
+value_str()
+default_str()
+type_str()
+min_str()
+max_str()
+parse_value_str()
<<create>>-bprb_param()

bprb_choice_param_type

-choices_: vcl_vector<vcl_string>

<<create>>-bprb_choice_param_type()
+clone()
+choices()

T : class

bprb_param_type

-value_: T
-default_: T
-temp_value_: T
-min_value_: T
-max_value_: T

-bprb_param_type<T>()
-bprb_param_type<T>()
+default_value()
+min_value()
+max_value()
+value()
+temp_ref()
+set_from_temp()
+set_value()
+reset()
+clone()
+value_str()
+default_str()
+type_str()
+min_str()
+max_str()
+parse_value_str()
-create_string()
-parse_string()

bprb_parameters

-name_param_map_: vcl_map< vcl_string , bprb_param* >
-param_list_: vcl_vector< bprb_param* >
-xml_doc_: bxml_document

<<create>>-bprb_parameters()
<<destroy>>-bprb_parameters()
<<create>>-bprb_parameters()
+valid_parameter()
+valid_parameter_type()
+add()
+add()
+add()
+set_value()
+get_value()
+value()
+get_default()
+get_bounds()
+parse_XML()
+print_def_XML()
+print_current_XML()
+reset_all()
+reset()
+get_param_list()
+get_desc()
+print_all()
-add()
-get_param()

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 para-
meters (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 statis-
tical 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 that what needs to be un-
derstood 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

- bsta_gaussian_sphere: A (hyper-)spherical Gaussian distribution (i.e. single variance)
- bsta_mixture: A mixture of distributions
- bsta_mixture_fixed: A mixture of distributions of fixed size
- bsta_basic_functors:
  - bsta_prob_density_functor: A functor to return the probability density at a sample
  - bsta_probability_functor: A functor to return the probability integrated over a box
  - bsta_mean_functor: A functor to return the mean of the Gaussian
  - bsta_var_functor: A functor to return the variance of the Gaussian
  - bsta_diag_covar_functor: A functor to return the variance of the Gaussian
  - bsta_det_covar_functor: A functor to return the determinant of the covariance of the Gaussian
  - bsta_weight_functor: A functor to return the weight of the component with given index
  - bsta_mixture_functor: A functor to apply another functor to one distribution in the mixture
  - bsta_mixture_data_functor: A functor to apply another functor with data to one distribution in the mixture
  - bsta_weighted_sum_functor: A functor to apply another functor to each distribution and produce a weighted sum
  - bsta_weighted_sum_data_functor: A functor to apply another functor with data to each distribution and produce a weighted sum
  - bsta_mixture_size_functor: A functor to count the number of components in the mixture
- bsta_detector_gaussian: Designed to include detectors applying to Gaussians. Currently it has one detector: bsta_g_mdist_detector, a simple Mahalanobis distance detector for a Gaussian
- bsta_detector_mixture: Designed to include detectors applying to mixtures. Currently it has two detectors:
  - bsta_top_weight_detector: applies a detector to each component in order while the total weight is below a threshold
  - bsta_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:

- **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:
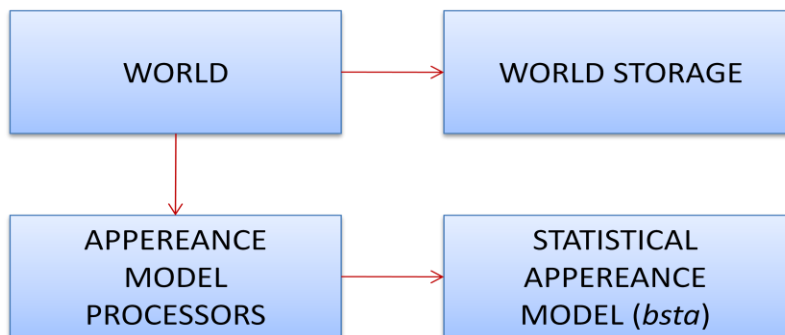
Figure 8: *BVXM Basics architecture diagram*

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_chunck*

Class Diagram

Figure 9: High-level *bvxm* storage

bvxm_memory_chunk

-data_: void
-size_: unsigned long

<<create>>-bvxm_memory_chunk()
<<create>>-bvxm_memory_chunk()
<<create>>-bvxm_memory_chunk()
<<CppOperator>>+=()
<<destroy>>-bvxm_memory_chunk()
+data()
+const_data()
+size()
+set_size()

T : class

bvxm_voxel_storage

#grid_size_: vgl_vector_3d<unsigned int>

<<create>>-bvxm_voxel_storage()
<<destroy>>-bvxm_voxel_storage()
+initialize_data()
+get_slab()
+put_slab()
+num_observations()
+increment_observations()
+nx()
+ny()
+nz()

T : class

bvxm_voxel_storage_header

+nx_: unsigned
+ny_: unsigned
+nz_: unsigned
+nobservations_: unsigned

<<create>>-bvxm_voxel_storage_header()
<<create>>-bvxm_voxel_storage_header()
<<create>>-bvxm_voxel_storage_header()
<<destroy>>-bvxm_voxel_storage_header()

<<CppTypedef>>
bvxm_memory_chunk_sptr

T : class

bvxm_voxel_slab

T : class

bvxm_voxel_storage_disk

#storage_fname_: vcl_string
#fio_: vcl_fstream
#slab_buffer_: bvxm_memory_chunk_sptr

<<create>>-bvxm_voxel_storage_disk()
<<destroy>>-bvxm_voxel_storage_disk()
+initialize_data()
+get_slab()
+put_slab()
+num_observations()
+increment_observations()

T : class

bvxm_voxel_storage_mem

-mem_: bvxm_memory_chunk_sptr
-nobservations_: unsigned

<<create>>-bvxm_voxel_storage_mem()
+initialize_data()
+get_slab()
+put_slab()
+num_observations()
+increment_observations()

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

---

bvxm_world_params

#model_dir_: vcl_string
#corner_: vgl_point_3d<float>
#num_voxels_: vgl_vector_3d<unsigned int>
#voxel_length_: float
#lvcs_: bgeo_lvcs_sptr
#min_occupancy_prob_: float
#max_occupancy_prob_: float

<<create>>-bvxm_world_params()
<<destroy>>-bvxm_world_params()
+set_params()
+model_dir()
+corner()
+num_voxels()
+voxel_length()
+min_occupancy_prob()
+max_occupancy_prob()
+lvcs()
+b_write()
+b_read()
<<CppFriend>>-<<()
<<CppFriend>>->>()
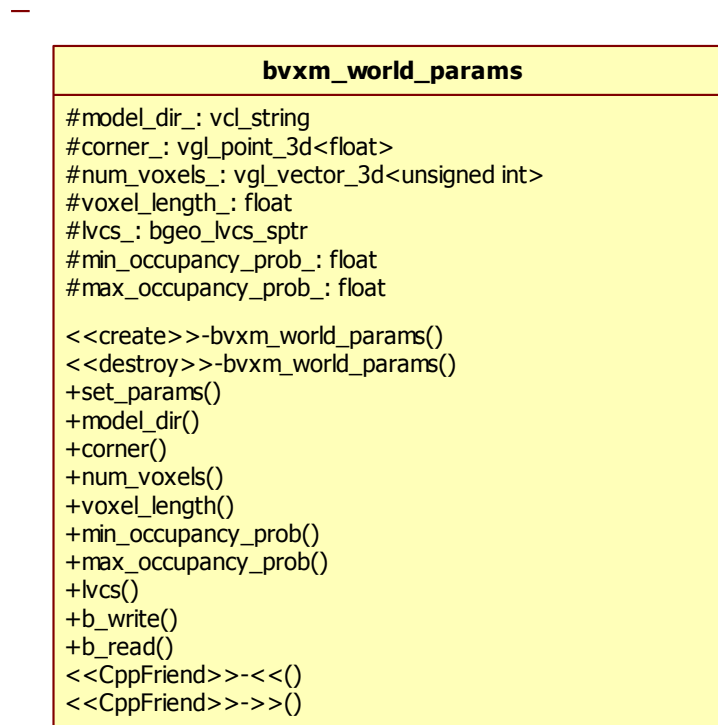
Figure 11: bvxm_world_params class

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

12

- *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:
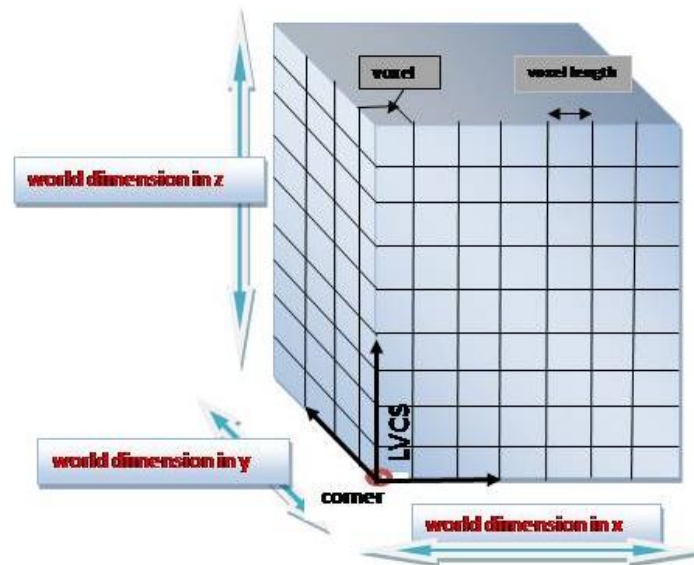
Figure 12: Voxel World

- **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>>
bvxm_voxel_type

+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.

```cpp
//: 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

```
┌─────────────────────────────┐
│          bvxm_util           │
├─────────────────────────────┤
│ +compute_plane_image_H()     │
│ +read_cameras()              │
│ +write_cameras()             │
│ +warp_slab_bilinear()        │
│ +img_to_slab()               │
│ +img_to_slab()               │
│ +slab_to_img()               │
│ +slab_to_img()               │
│ +multiply_slabs()            │
│ +add_slabs()                 │
│ +threshold_slab_above()      │
│ +smooth_gaussian()           │
│ +write_slab_as_image()       │
│ +write_slab_as_image()       │
│ +sum_slab()                  │
│ #bilinear_weights()          │
└─────────────────────────────┘
```
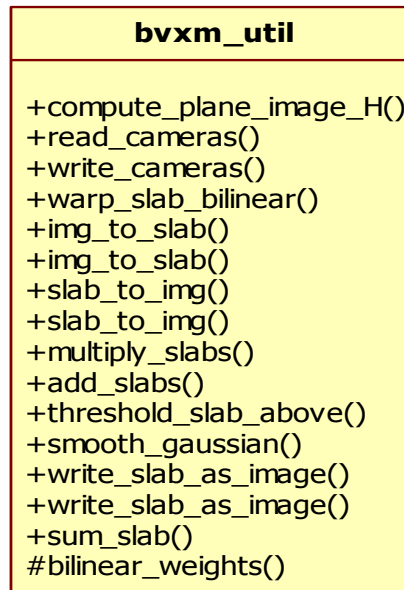
Figure 13: bvxm_util class

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.

—

bvxm_voxel_world
#grid_map_: vcl_map<bvxm_voxel_type, vcl_map<unsigned int, bvxm_voxel_grid_base_sptr> > #params_: bvxm_world_params_sptr
<<create>>-bvxm_voxel_world() <<create>>-bvxm_voxel_world() <<destroy>>-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() <<CppOperator>>+==() <<CppOperator>>+<() +get_grid() +save_occupancy_raw() +clean_grids() +num_observations() +increment_observations() -update_impl()

Figure 14: bvxm_voxel_world class

16

Detailed Description:

Constructor & Destructor Documentation

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

Member Variables Documentation:

- o **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 containng 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 appropiate 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)**

Observation:
bvxm_image_metadata

Storage for output
pixel probabilities
vil_image_view

Bin index: unsigned int

template <bvxm_voel_type APM_T>
pixel_probability_density

Success indicator: bool
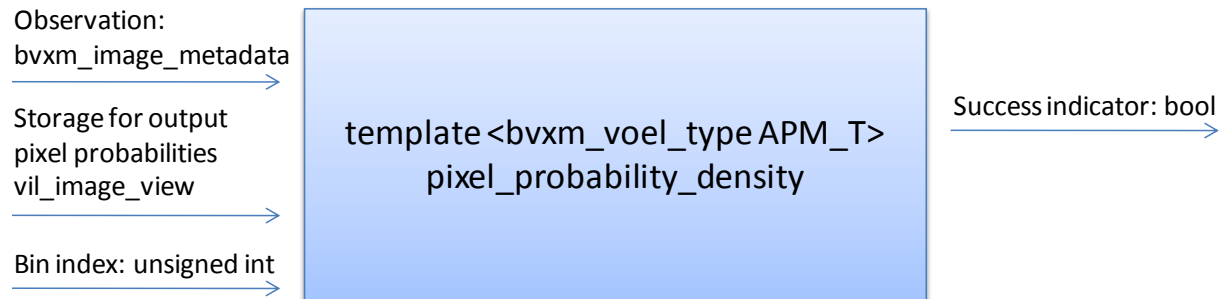
Figure 15: An example of member function of the *bvxm_voxel_world*

```
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;
```

```cpp
    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 appereance 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 al 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_f1 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
```

bvxm_mog_grey_processor
#n_gaussian_modes_ : unsigned
<<create>>-bvxm_mog_grey_processor() +prob_density() +prob_range() +update() +expected_color()

bvxm_mog_rgb_processor
<<create>>-bvxm_mog_rgb_processor() +prob_density() +prob_range() +update() +expected_color()
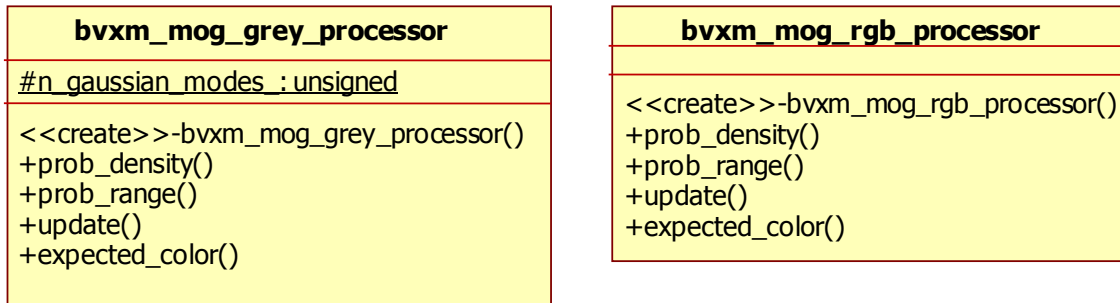
Figure 16: Appereance model procesors

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.
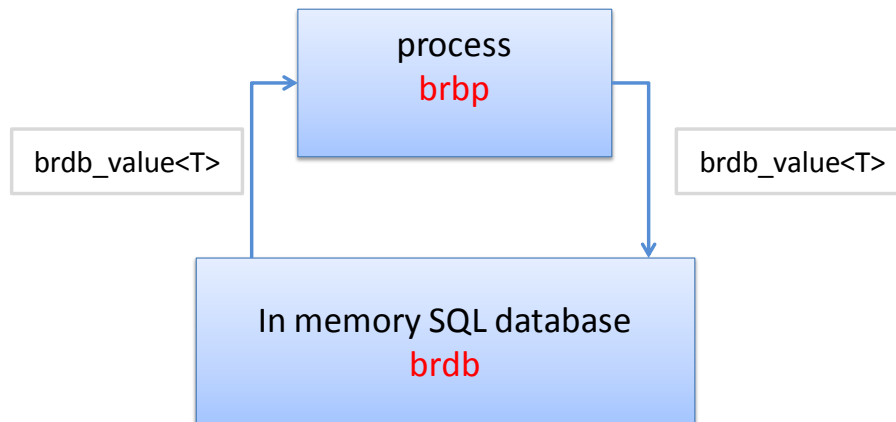
Figure 17: Process arquitecture

Inputs and parameters:

Every process in *bvxm_pro* inherits from *bprb_process* (see brpb descrption 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
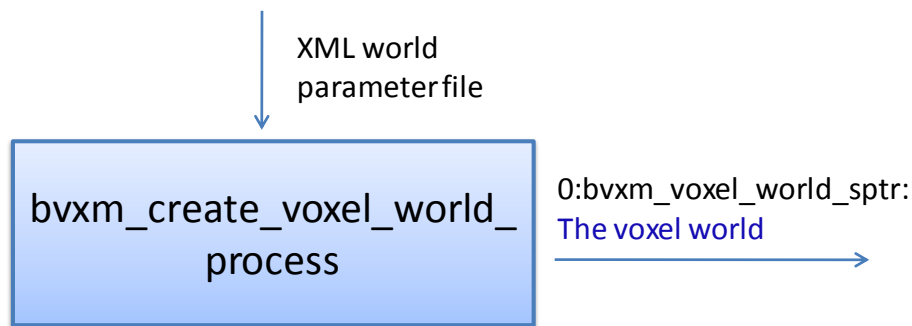
Figure 18: bvxm_create_voxel_world_process

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
<?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:

world_model_params_lvcs.txt
wgs84
meters
degrees
32.7242 -117.156 39.0
0 0
0 0 0

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):

parameter	Default value
input_directory	**"./"**
corner_x	**0.0**
corner_y	**0.0**
corner_z	**0.0**
voxel_dim_x	**10**
voxel_dim_y	**10**
voxel_dim_z	**10**
voxel_length	**1**
lvcs	**"./"**
min_ocp_prob	**1e-5f**
max_ocp_prob	**1-1e-5f**

Initialize the incoming image region of interest
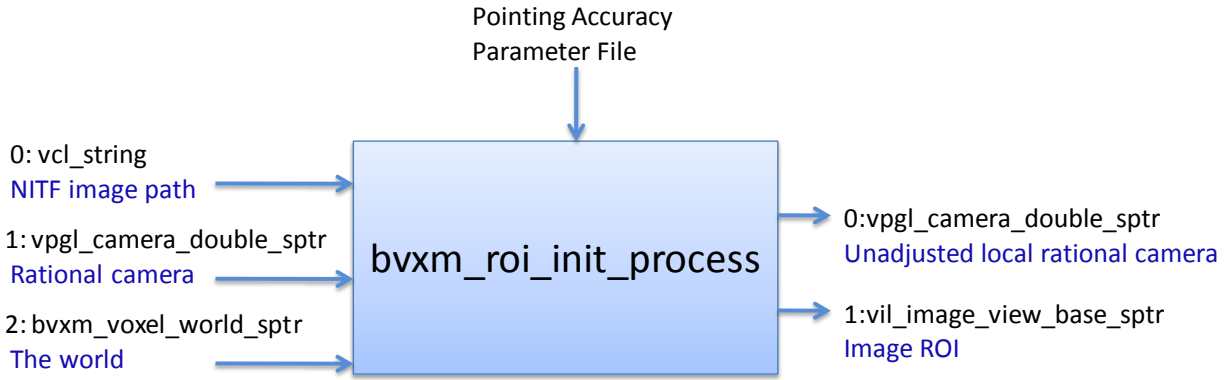
Figure 19: bvxm_roi_init_process

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

Parameter File:
•Offset search size
•Gaussian Sigma for edge distance map

0: bvxm_voxel_world_sptr
The voxel world

1: vpgl_camera_double_sptr
The current camera

2: vil_image_view_base_sptr
The current edgmap image

bvxm_rpc_registration_process

0: vpgl_camera_double_sptr
The optimized camera
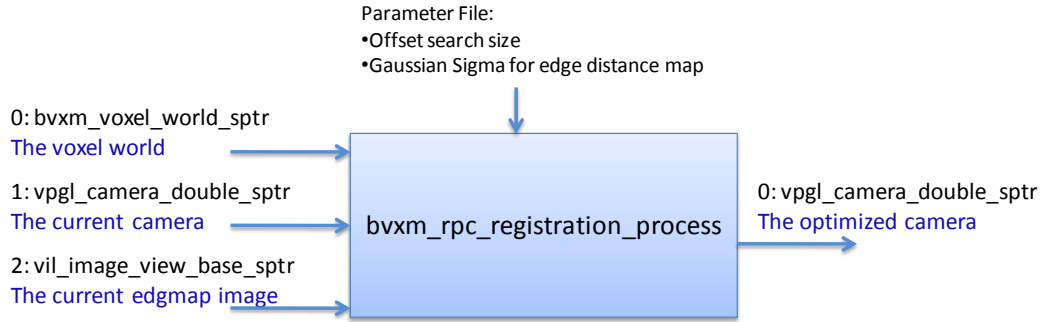
Figure 20: bvxm_rpc_registration_process

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 of 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

line_fitting_min_length

edge_detection_gaussian_sigma
use_lines

line_fitting_error_tolerance

0: vil_image_view_base_sptr
Input grey scale image

bvxm_generate_edge_map_process

0: vil_image_view_base_sptr
Edge image
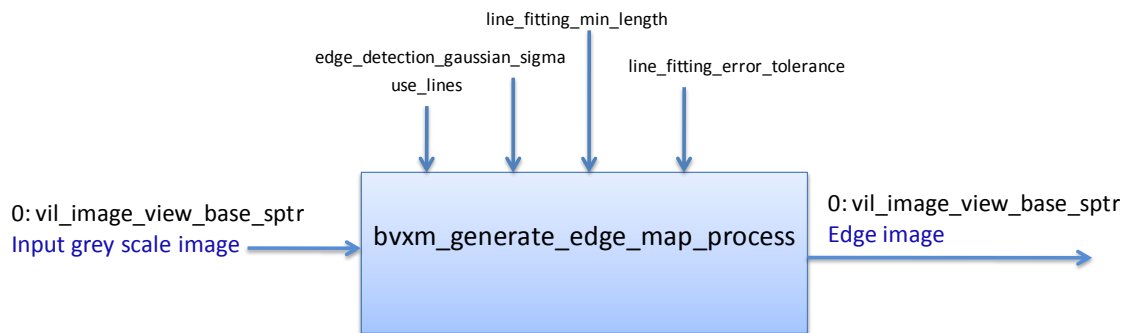
Figure 21: bvxm_generate_edge_map_process

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
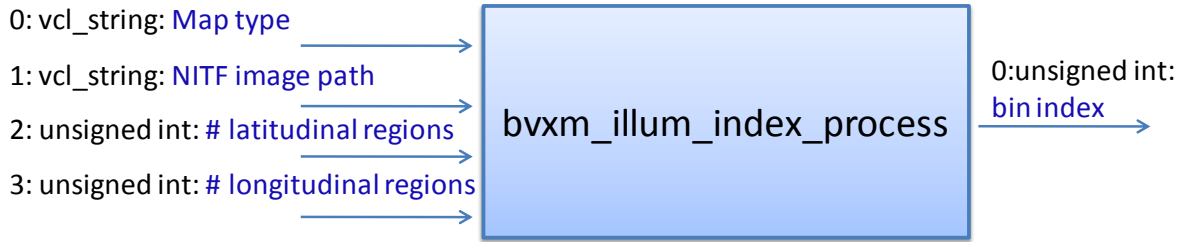
Figure 22: bvxm_illum_index_process

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 discritizes, 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
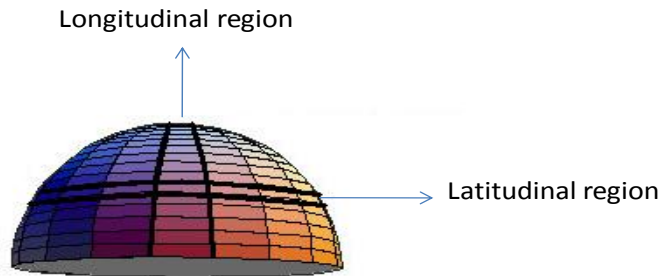
Figure 23. Illumination Grid

Image normalization
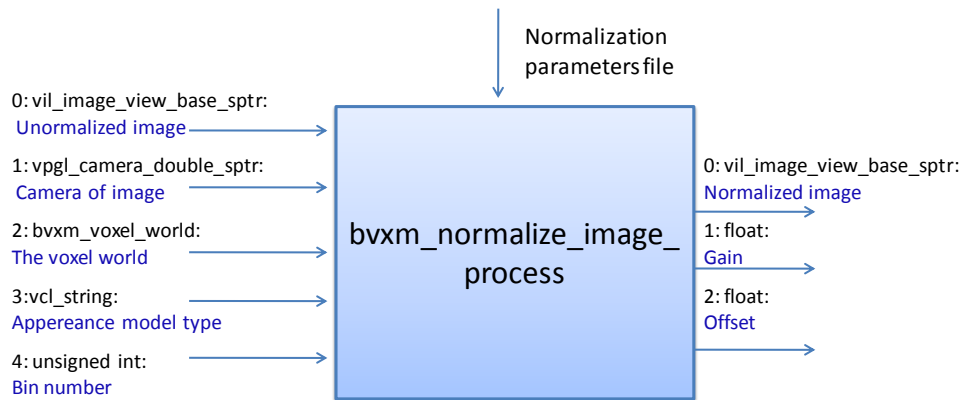
Figure 24: bvxm_normalize_image_process

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 parameters 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
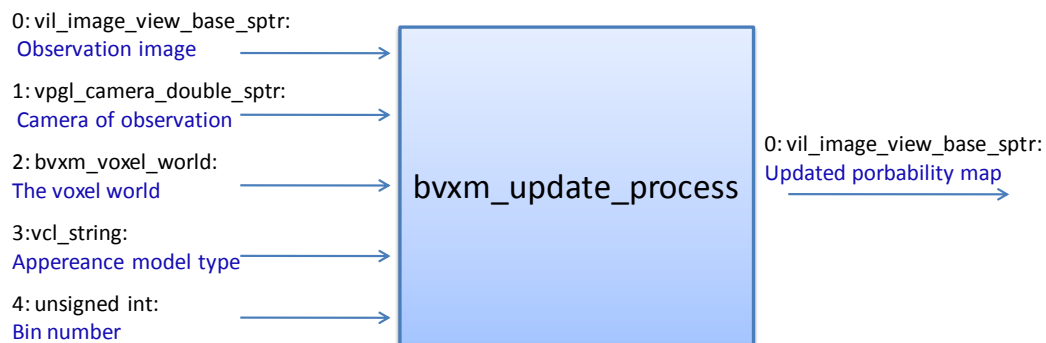
Figure 25: bvxm_update_process

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
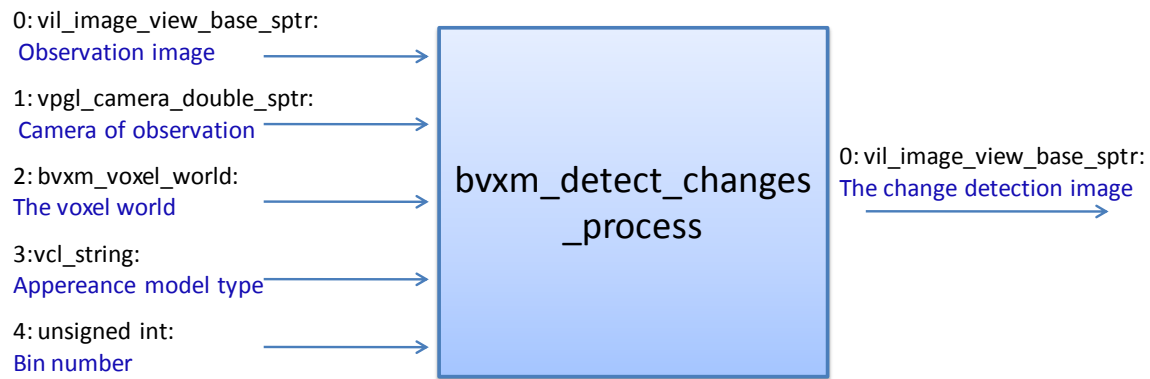
Figure 26: bvxm_detect_changes_process

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:

- 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.

Produce the change detection result

Figure 27: bvxm_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
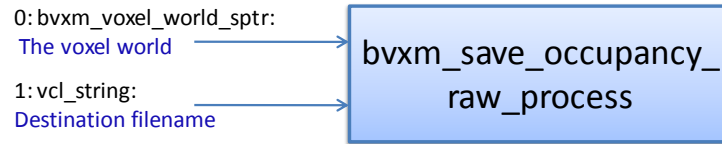
Figure 28: bvxm_save_occupancy_raw_process

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
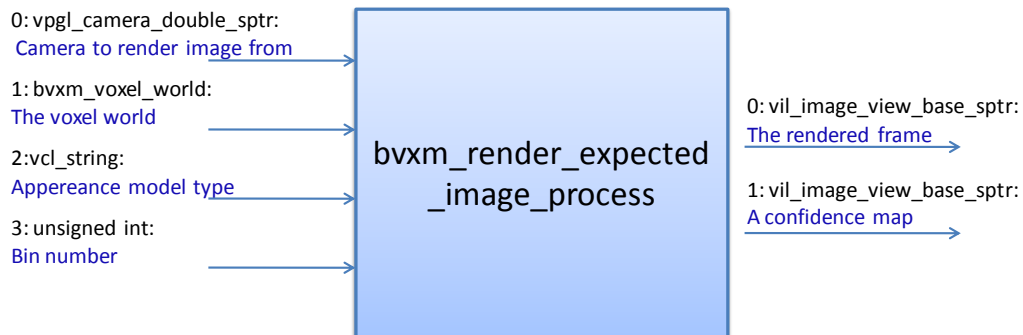
Figure 29: bvxm_render_expected_image_process

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

Rendering a virtual view: 3-D Registration tool
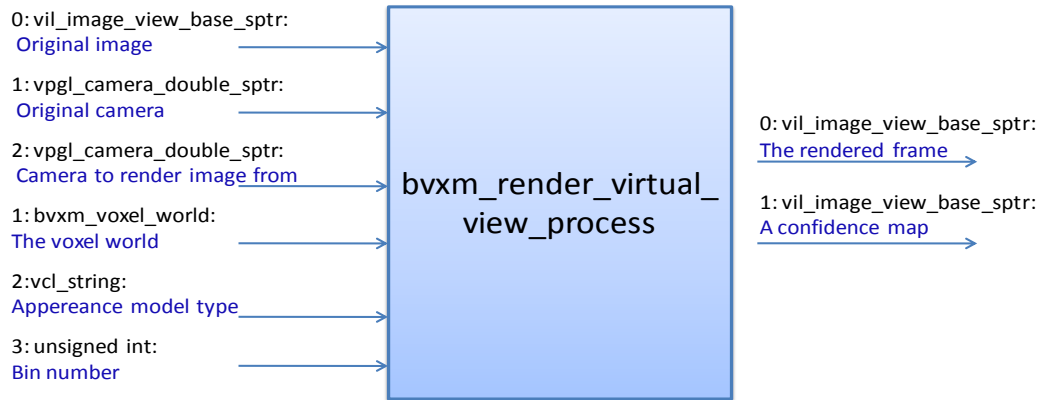
Figure 30: bvxm_render_virtual_view_process

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 *bvxm* processes in batch mode. *bvxm_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_bvxm:* This is where all Python's external declarations reside.
- *reg_bvxm:* All processes and data types need to be registered in order to be recognized by the *bprb_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 bvxm_batch
bvxm_batch.register_processes();
bvxm_batch.register_datatypes();
bvxm_batch.print_db();
```

We will observe the following empty relations in the database.

```
C: Cygwin                                                    ◁▷ ⊡  _ ⊡  ✕

<<<<<<<<<<<<<<<<<<---- Printing database ---->>>>>>>>>>>>>>>>>>>>>>>
───────────────────────────────────────────────────────────────────
Relation name: bool_data
id(unsigned)    value(bool)
───────────────────────────────────────────────────────────────────
Relation name: bvxm_voxel_world_sptr_data
id(unsigned)    value(bvxm_voxel_world_sptr)
───────────────────────────────────────────────────────────────────
Relation name: double_data
id(unsigned)    value(double)
───────────────────────────────────────────────────────────────────
Relation name: float_data
id(unsigned)    value(float)
───────────────────────────────────────────────────────────────────
Relation name: int_data
id(unsigned)    value(int)
───────────────────────────────────────────────────────────────────
Relation name: long_data
id(unsigned)    value(long)
───────────────────────────────────────────────────────────────────
Relation name: vcl_string_data
id(unsigned)    value(vcl_string)
───────────────────────────────────────────────────────────────────
Relation name: vil_image_view_base_sptr_data
id(unsigned)    value(vil_image_view_base_sptr)
───────────────────────────────────────────────────────────────────
Relation name: vpgl_camera_double_sptr_data
id(unsigned)    value(vpgl_camera_double_sptr)
───────────────────────────────────────────────────────────────────
<<<<<<<<<<<<<<<<<<<<<<<<---- end ---->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

bvxm_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()*: Createa 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(bool ProcessInput):* Prints the default values of the process by name
- *set input (i, v):* set input i on current process to a string 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 a string value
- *set_input_long(unsigned i, long input):* Sets input i on current process to a string value
- *set_input_float(unsigned i, float input):* Sets input i on current process to a string value

- *set_input_double(unsigned i, double input):* Sets input i on current process to a string value
- *process_print_default_params(string "ProcessName", string "ParamName"):* Print sthe 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 form 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 *vpgl_load_rational_camera_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);
vxm_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_paramete
rs.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 re-
    flecting 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 whishes 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)
- vpgl – 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	10	0	0	18	0	2	656
GE	FreeBSD-6.2_gcc-3.4.6_-O2_shared	10	0	0	86	0	3	655
GE	FreeBSD-6.2_gcc-4.0.4_-O2_shared	10	0	0	140	0	3	655
GE	FreeBSD-6.2_gcc-4.2.3_-O2_shared	10	0	0	122	0	4	654
GE	FreeBSD-7.0_gcc-4.2.1_profile_static	10	0	0	19	0	2	651
cs.rpi.edu	FreeBSD-fresh-gcc-3.4.2	10	0	0	0	0	6	649
lems.brown.edu	Linux-2.6_gcc-4.1.3_-Wall	10	0	0	244	0	2	655
lems.brown.edu	Linux-2.6.18_gcc-4.1.2	10	0	0	0	0	3	654
imorphics.com	Linux-2.6.22_gcc-4.1.2_RelWithDebInfo	10	0	0	0	0	1	450
GE	Linux-2.6.9_icc-9.1-64bit_-O2_shared							
GE	MinGW-3.8_gcc-3.4.4_-O2_static	10	0	0	1315	0	3	655
GE	Win2k_bcc-5.5.1_Release	10	0	0	0	0	5	246
GE	Win2k_msvc-7.1_Debug	10	0	0	102	0	2	656
imorphics.com	WinXP_msvc-7.1_RelWithDebInfo							