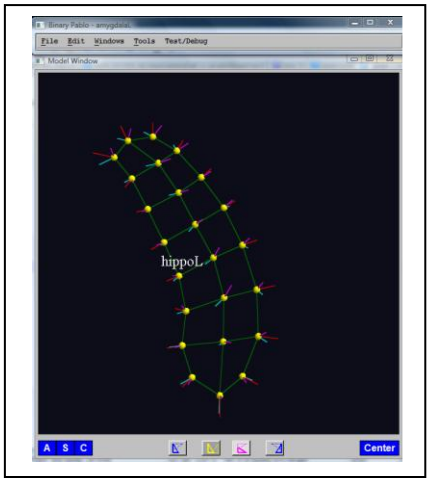
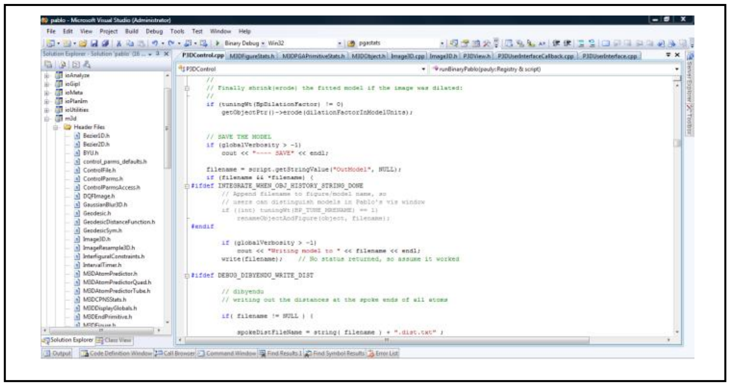
BINARY PABLO(v2)

DEVELOPERS MANUAL 2.0

Dibyendusekhar Goswami {dgoswami@cs.unc.edu}

Department of Computer Science, UNC Chapel Hill




pablo - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Tools Test Window Help
Binary Debug Win32
pgastats
P3DControl.cpp M3DFigureStats.h M3DPGAPrimitiveStats.h M3DObject.h Image3D.cpp Image3D.h P3DView.h P3DUserInterfaceCallback.cpp P3DUserInterface.cpp
P3DControl
runBinaryPablo(pauly::Registry & script)
Solution Explorer
Class View
Output
Code Definition Window
Call Browser
Command Window
Find Results 1
Find Symbol Results
Error List

```
    //
    // Finally shrink(erode) the fitted model if the image was dilated:
    //
    if (tuningWt(BpDilationFactor) != 0)
        getObjectPtr()->erode(dilationFactorInModelUnits);


    // SAVE THE MODEL
    if (globalVerbosity > -1)
        cout << "---- SAVE" << endl;

    filename = script.getStringValue("OutModel", NULL);
    if (filename && *filename) {
#ifdef INTEGRATE_WHEN_OBJ_HISTORY_STRING_DONE
        // Append filename to figure/model name, so
        // users can distinguish models in Pablo's vis window
        if ((int) tuningWt(BP_TUNE_PRENAME) == 1)
            renameObjectAndFigure(object, filename);
#endif

        if (globalVerbosity > -1)
            cout << "Writing model to " << filename << endl;
        write(filename);    // No status returned, so assume it worked

#ifdef DEBUG_DIBYENDU_WRITE_DIST

        // dibyendu
        // writing out the distances at the spoke ends of all atoms

        if( filename != NULL ) {

            spokeDistFileName = string( filename ) + ".dist.txt" ;
```

CONTENTS

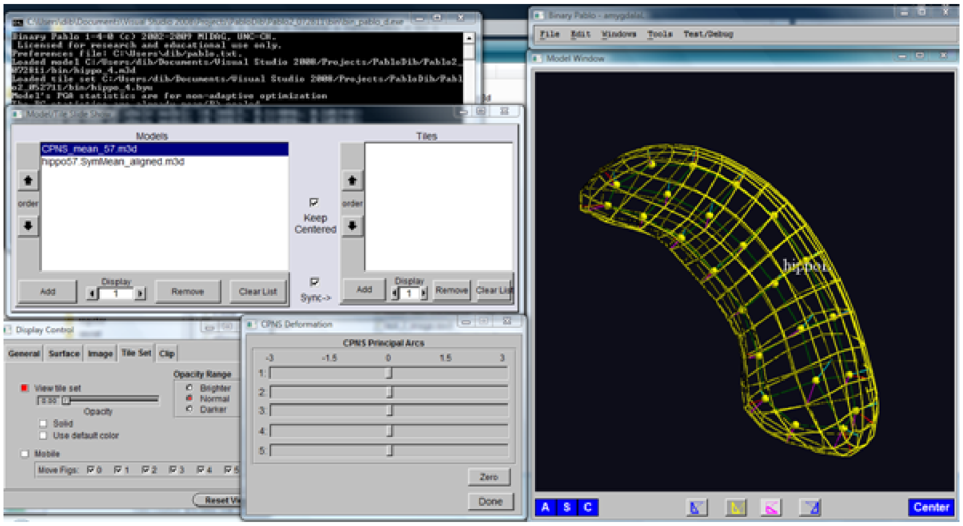
REVISIONS

Change by (name, email)	Date	Description	Version
Dibyendu (dibyendusekharg@gmail.com)	Aug 02, 2011	Created the document	1.0
Dibyendu (dibyendusekharg@gmail.com)	Aug 04, 2011	Modifications after presentation to the Group	1.1
Dibyendu (dibyendusekharg@gmail.com)	Sept 22, 2011	Additions suggested by Gregg Tracton (History Bits and ProAdvice)	2.0

I – INTRODUCTION AND BACKGROUND

INTRODUCTION TO PABLO

Pablo (v2) is a software package developed at UNC [Figure 1], used for a variety of medical imaging applications, primarily, model fitting (to medical images) and statistics on (fitted) models. A version of this program, known as *Grayscale Pablo* (or *Pablo* in general), is also used for segmentation in grayscale images via posterior optimization. This document is focused on *Binary Pablo*, the version of this software that operates on *Binary images* and *Distance maps*. *M-reps* and *S-reps* [Section 1.1] are the underlying structure for representation of models in this software. This document is organized in the following manner:



```
Binary Pablo 1-4-0 (c) 2002-2009 MIDAG, UNC-CH.
 Licensed for research and educational use only.
Preferences file: C:\Users\dib\pablo.txt.
Loaded model C:/Users/dib/Documents/Visual Studio 2008/Projects/PabloDib/Pablo2_
072811/bin/hippo_4.m3d
Loaded tile set C:/Users/dib/Documents/Visual Studio 2008/Projects/PabloDib/Pabl
o2_052711/bin/hippo_4.byu
Model's PGA statistics are for non-adaptive optimization
```

Figure 1: Snapshot of Binary Pablo. It is a versatile tool with multiple functionalities

Section – I deals with some of the background and theoretical knowledge required to use or modify Pablo.

Section – II is a brief overview of the various projects (libraries) in Pablo.

Section – III goes into detailed descriptions of relevant projects and also has examples, diagrams and sample code to present a better understanding of Pablo code.

Section –IV has some examples of code flow for certain tasks in Pablo. Some of these examples would be helpful for developers to modify or add code in Pablo.

Appendix – I is the old user manual that was written for Pablo in 2001. Many of these features are still relevant, and it is still a useful resource.

Format of the manual

Names of Classes are written in **bold**. Names of Projects are written in UPPERCASE. Names of Functions and Regular C++ Code is written in `code (courier new) font`. Filenames are written in ***italicized bold***. Terms with specific meanings are written in *italics*.

History Bits and *ProAdvice* are sections that have been added from suggestions by Gregg Tracton.

Pablo Licenses

Although runtime licenses are supported, they are disabled for UNC researchers. Pablo source should never be distributed without Steve Pizer's permission. Distributing executables requires us to create a runtime license file, and the global key is in the source code so we need to keep that code secret.

Please update this manual !

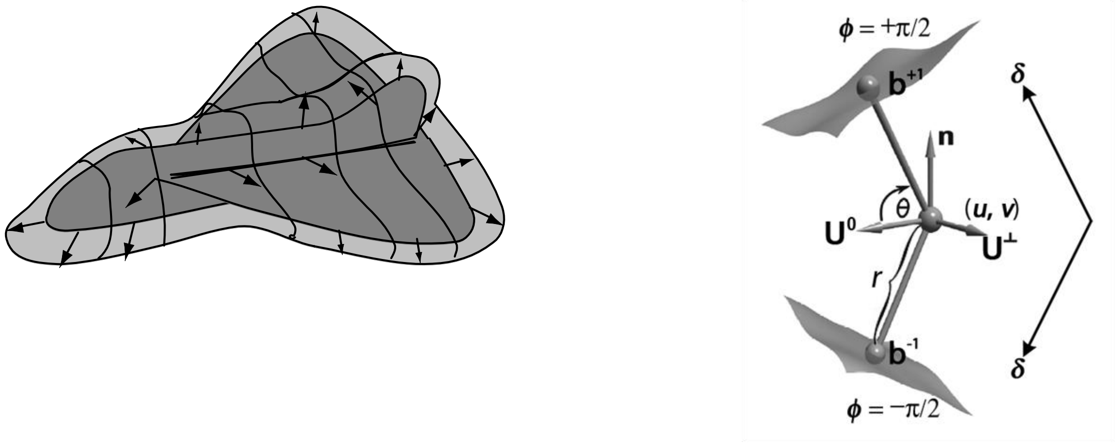
This document has been written with the hope that it would be consistently updated by Pablo programmers after addition or modification of features in Pablo. [If you are not able to follow the format in which the manual was written, please go ahead with the update; do document the fact that the format has not been followed.] Please do also document bugs and any miscellaneous information about Pablo. A Wiki has been setup setup for Pablo Developers:
https://sites.google.com/site/shapeunc/home

This site contains miscellaneous information about Pablo and also editable versions of this manual (and also editable versions of some figures in this document).

1.1 BACKGROUND: M-REPS AND S-REPS

Medial Representations (*m-reps*) are a powerful way of representing the skeleton as well as interior of an object. As shown in Figure 2(a), the medial representation of an object consists of the following elements.

- 1-D or 2-D medial axis (or medial sheet), which represents the skeleton of the object.
- Medial atoms that lie on the medial axis; they contain spokes from the sheet to the boundary. These spokes represent the interior of the object



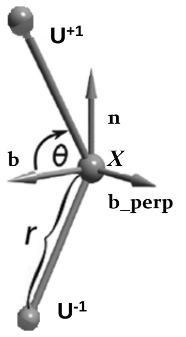
(a) Medial Representation of an airplane (b) An atom / hub in a medial sheet

Figure 2: Elements of Medial Representations

As shown in Figure 2(b), a medial atom has a pair of spokes that start at the medial sheet and touch the boundary of the object. These spokes are of equal length because they lie on a bi-tangent sphere to the object boundary whose center is at the position of the medial atom. There is also a fitted frame associated with every atom; this frame consists of the three orthogonal vectors b, n, b_perp, where b is the bisector of the two spokes and n is the normal to the medial sheet at that position.

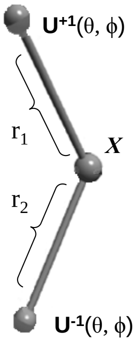
While *m-reps* have been successfully used for model fitting and statistics on objects [2], they suffer from some drawbacks. One of the prime drawbacks is that a small bump or dent in the object causes a huge change in the medial skeleton of the object, or else the spokes do not terminate very near to the object boundary. This causes inconvenience while doing statistics on a population of objects, or while determining the correspondence between two objects of the same class. One of the solutions proposed to solve this problem is to allow medial models higher flexibility, in that we allow medial atoms to have spokes with slight differences in length. This representation is referred to as the skeletal representation (*s-reps*). *S-reps* still have an underlying medial structure but are more flexible and can be forced to have different properties depending upon the requirements.

The primary difference between an *m-rep* atom and an *s-rep* atom is shown in Figure 3(a) and Figure 3(b). While in case of an *m-rep* atom, the spokes are tied to each other via the fitted frame, in the *s-rep* atom, each spoke has its own orientation and radius values.



- Position: X (also called *P* or *pos*)
- Fitted frame: b, n, b_perp
- Radius: r
- Angle: θ

Figure 3(a) Representation of an *m-rep* atom

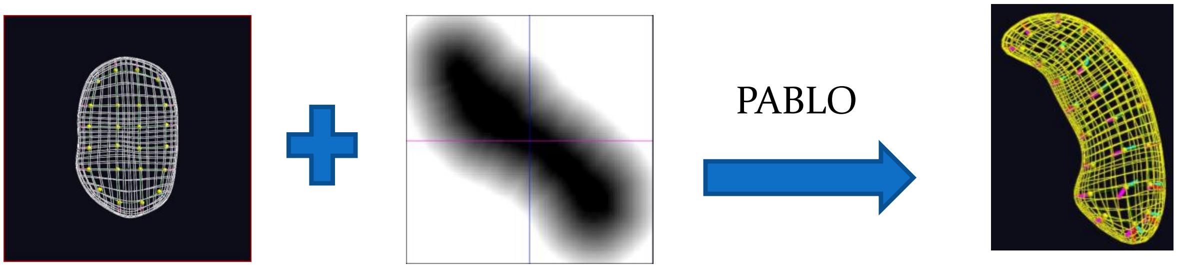


- Position: X (also called *P* or *pos*)
- For every spoke i
  - Radius: r_i
  - Orientation: U^i

Figure 3(b) Representation of an *s-rep* atom

1.2 BACKGROUND: DEFORMABLE MODEL FITTING

The model fitting process in Pablo [Figure 4] starts with an initial model and a distance map (which is computed from the binary image) of the *target object*. At every iteration, the initial model is deformed in a way to get it closer to match the *target object*. The process involves minimization of two energies at every step, the *geometry mismatch* and the *data mismatch*. *Geometry mismatch* refers to how different this object is from a typical instance of such an object, like the mean of a population, or a previous instance of the object. *Data mismatch* refers to how closely this object matches the data of this particular instance, that is, the distance map of the *target object*.



Initial Model

Distance Map

Fitted model

Figure 4: General Overview of the Pablo model fitting process

The deformation of the model is optimized by the *conjugate gradients* method. This deformation process takes place in several stages:

1. GLOBAL TRANSFORMATION

   This stage is also referred to as the *Method of Moments* (MOM) stage, *Registration* stage*, Setup* stage *or Ensemble* stage. This step is required to align the *initial model* with the *target image* in terms of translation, rotation and size.

   - Translation: Align COG of *initial model* with the COG of *target image*.
   - Scaling: Compute scale factor from 1st moments of *initial model* and *target image.*
   - Rotation: Compute rotation from 2nd moments of *initial model* and *target image.*

   If Landmarks are available in the initial model and the target image, then global transformation is also carried out using those landmarks instead of using MOM method.

2. FIGURE STAGE / MAHALANOBIS STAGE

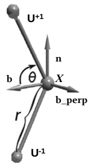
   This stage involves optimization over the space of coefficients of the principal eigenmodes of the initial model. The initial model is preferably a statistical mean of a training population. The current code supports two types of statistics on s-rep models, *PGA (Principal Geodesic Analysis)* and *CPNS* (*Composite Principal Nested Spheres*). The Mahalanobis stage can also take place at an atom level if Atom PGA Statistics are available.

3. ATOM STAGE / DEFORMATION STAGE

   This stage involves examining one atom at a time and optimizing over the tuple of parameters that represent the location of the atom, its orientation, and the orientation of its spokes. Figure 5(a) shows the parameters that are optimized via conjugate gradients in the Atom Stage. The Atom Stage is also referred to as the Deformation Stage at some places in the code, esp. when defining *optimizers* in the register project; *M3DDeformationOptimizer* refers to the Atom Stage optimizer.

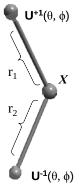
4. S-REP SPOKE STAGE (or SLAB SPOKE STAGE)

   This stage of optimization is presently only available for quasi-slabular s-reps. At the *s-rep* stage, for every atom we carry out separate conjugate gradient method optimizations for every spoke of that atom. The vector that is passed to the conjugate gradients method (the method itself undergoes several iterations) consists of three parameters per spoke, one parameter Δr for the change in the length of that spoke, and two parameters ($\Delta\theta$, $\Delta\phi$) for the rotation of that spoke (spherical co-ordinates). Figure 5(a) and 5(b) show the differences in the optimization vectors in the m-rep atom stage and the s-rep stage of the optimization process. The s-rep stage adds more freedom to the model, in that it converts the model from an *m-rep* to an *s-rep*.



- Position: ΔX [3]
- Rotation of Fitted frame: e [3]
- Radius: Δr
- Angle: $\Delta\theta$

Figure 5(a) Conjugate gradient vector for *m-rep* atom stage optimization



- For every spoke i
  - Radius: Δr_i
  - Orientation: $\Delta\theta^i$, $\Delta\phi^i$

Figure 5(b) Conjugate gradient vector for *s-rep* stage optimization

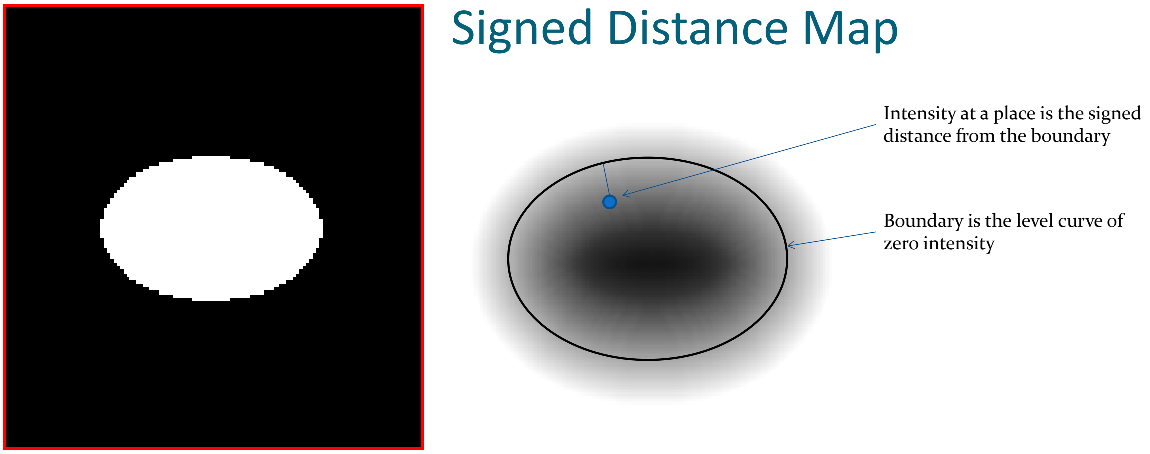
5. SPOKE STAGE (or TUBE SPOKE STAGE)

This optimization stage is only available for quasi-tubular m-reps, which are not discussed in this document.

HISTORY BITS

A deprecated "boundary displacement" stage used to adjust the length of each spoke, stored as delta of r per spoke, to match the spoke tip against the gray/binary intensity pattern expected at the boundary. Mexican hat filters and other non-trained matches were used and that code probably still exists. This was further branched by Joshua Stough to include shifting the spoke tips along the boundary to seek out intensities, using variables such as mask (stored in a file) & dMask.

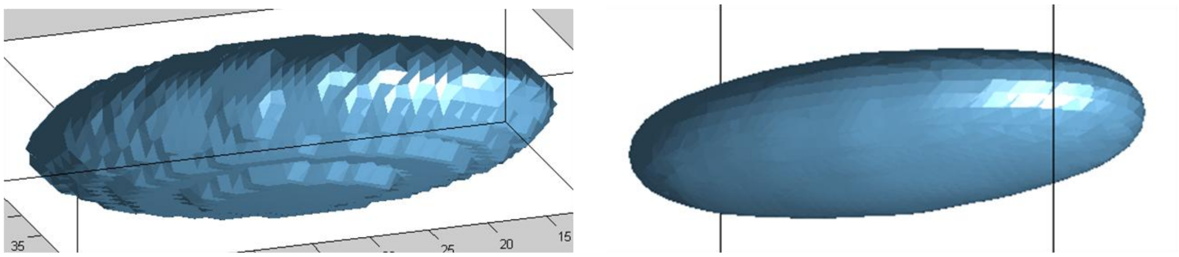
1.3 BACKGROUND: SIGNED DISTANCE MAPS


Signed Distance Map
Intensity at a place is the signed distance from the boundary
Boundary is the level curve of zero intensity

(a) A binary image (3D) (b) Signed distance map of the same object

Figure 6: What is a signed distance map?

A signed distance map [Figure 6] represents the object implicitly as the zero level set of this map. Though any convention is correct, in our case the distances are positive on the outside of the object and negative in the object interior. Every point in the distance map has a magnitude that is monotonic with respect to the Euclidean distance from that point to the object boundary.



(a) Before anti-aliasing (b) After anti-aliasing

Figure 7(a). Distance maps are anti-aliased by the Laplacian Flow of Curvature Method (Rohit Saboo)

Anti-aliasing is necessary to correctly determine higher order properties on the boundary like gradient, normal, curvature, ridges etc.

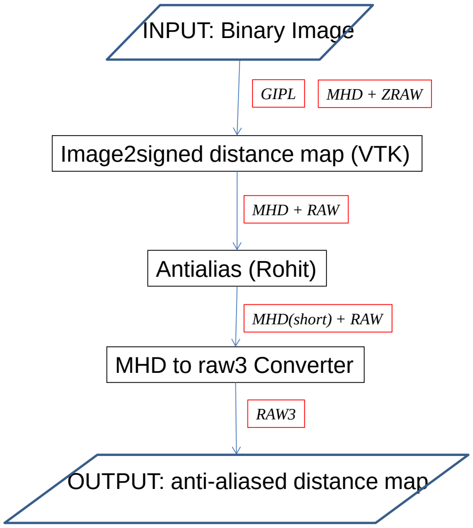


Figure 7(b): Pipeline for getting Anti-aliased image from Binary image

1. Run VTK Image2SignedDistanceMap <name>.<type> <name>
2. Run in matlab: antiAliasWrapper('<name>-ddm.mhd', '<name>', [0.5 0.5]);
3. Run a Perl script: mhd2raw3.pl <name>-antiAliased.mhd <name>-antiAliased.raw3

Table 0: Series of steps to calculate an anti-aliased distance map from a binary image.
(Requires access to Radonc AFS login)

II - OVERVIEW OF PABLO PROJECTS / LIBRARIES

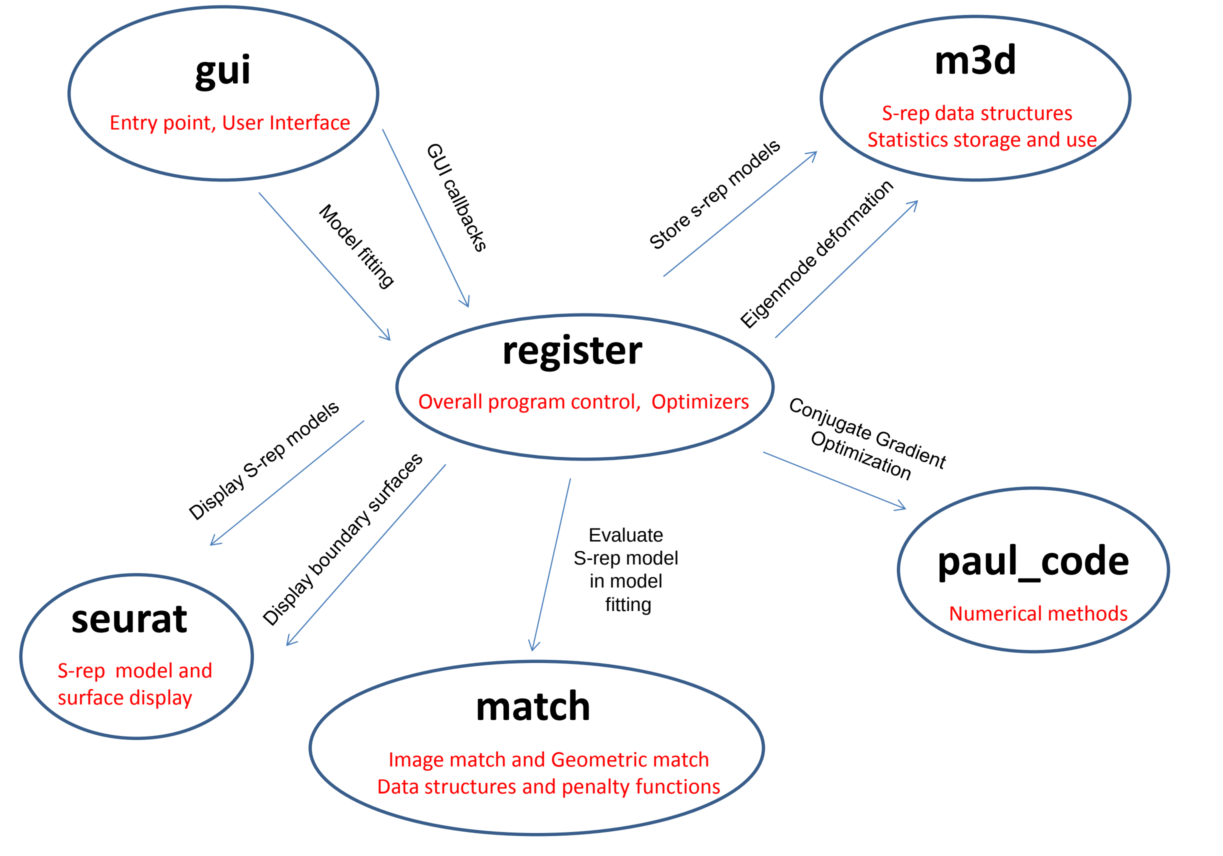


Figure 8: Typical Interactions between major Pablo projects / libraries

The figure above shows some typical interactions between the major projects (libraries) in Pablo. Pablo was designed using the *Model-View-Controller* software design pattern, in which the REGISTER project (**P3DControl** class) is the *controller*, **M3DObject** is the *model*, and the *views* are **P3DView** and **P3DCutPlaneView**. This section contains an overview of each project in the form of a short description of the project and relevant classes in it.

HISTORY BITS

A lot of projects in Pablo start with "3D" because there was a "2D" version before Pablo (atoms were limited to sequential planes).

GUI

This is the entry point for Pablo code execution. It contains the files ***main.cpp*** and ***Pablo.cpp***, which parse input arguments and then pass the control to P3DControl.cpp in the REGISTER project.

This project also contains code for design and execution of Pablo GUI. Pablo uses the FLTK library for GUI management. **P3DUserInterface** class contains code for creating windows, sliders, buttons, drop-down menus etc.; **P3DUserInterfaceCallBack** class contains the callback functions for all such GUI objects. **P3DUserInterface** class contains a **P3DControl** object (REGISTER project) and most of these callback functions eventually invoke relevant member functions of the **P3DControl** class.

ProAdvice

Pablo has multiple main functions. Instead of starting a new application, some students merely tacked additional functionality and set a tuning parameter to call their own code when they needed it. Some instances of this: the -pc switch just reads the image and writes the compressed version (which can also be used to convert image formats), the thin plate spline code, the distance map generator, reading a binary image to write tiles via simple marching cubes, etc.

A nifty project that was intended to stem that tide of adding to `main()` is the *Levy/Tracton PipeStage framework* for writing utilities, which tied a command-line parser to a set of templated readers and writers so that higher-level code was fairly simple to write - most simple app's could be completed in 30 minutes by starting from a similar app. Check SVN first, but then consult:
/afs/radonc/home/tracton/pipeline/pablo_util

Scripting means running a canned non-interactive Pablo from the command line. There are other meanings, such as running pablo from a bash/perl script. There have been many attempts at combining scripts into a system that understands data processing requirements such as the order that files must be processed: make, Rohit's unnamed perl modules, *Tracton/Levy perl pipeScripts* (can use any command-line app, like *PipeStage utilites*, or Pablo directly), Rohit's *vistrails* and many others.

M3D *(model 3D)*

This project contains data structures for storage and manipulation of s-rep models at different levels of the hierarchy (object, figure, atom, spoke etc.). Classes **M3DObject**, **M3DFigure**, **M3DPrimitive** and **M3DSpoke** are the basic classes for representing these objects (most of them are abstract classes). There are Renderer classes, like **M3DObjectRenderer, M3DFigureRenderer and M3DPrimitiveRenderer** for rendering these objects in the Pablo GUI display window. Also contains classes for matrices and vectors, like, **Matrix4D, Vector3D** etc.

This project also contains classes **M3DPGAStats** and **M3DCPNSStats**, which play a vital role in determining the use of PGA and CPNS statistics in Pablo. Pablo GUI contains interfaces to observe *eigenmode* deformations for both types of statistics.

This also contains the **Image3D** class which is used in Pablo to store a 3D image (binary image, grayscale image, distance maps etc.). An Image3D class contains a pointer to an *unsigned short array*, which is the image data.

MATCH

This project contains data structures to compute and store the geometry as well as image match between a model and an image. The **Match** class contains penalty functions for both types of match. Also contains class **ImageDistanceMap** for storing and manipulating a distance map. Also contains the infrastructure for storing and manipulating tuning parameters for the optimization process in **bpTuning** (*binary Pablo tuning*), **gpTuning** (*grayscale Pablo tuning*) and **Tuning** (an *abstract class*) classes.

PAUL_CODE *(coded by Paul Yushkevich)*

This project contains the infrastructure for performing optimization in Pablo. Contains classes for optimization problems (like **Problem, NumericalProblem** etc.), optimization functions (like **NumericalFunction**, **DifferentiableFunction** etc.), optimization solutions (like **Solution, NumericalSolution** etc.) and methods (like **ConjugateGradientmethod, BrentLinearMethod, SimplexMethod, EvolutionaryStrategy** etc.).

Also contains the **Registry** class, a powerful data structure used throughout the Pablo program. A **Registry** is a hash table which stores (key, value) pairs; keys are strings and values can be *integers*, *doubles*, *arrays* or *folders*. Registries can also be read from and written into ASCII files which can be hand-manipulated. Examples of Registry entries include tuning parameters for optimization and various control flags for Pablo operation.

REGISTER

This project contains the **P3DControl** class (*Pablo 3D Control*) which controls the entire functioning of Pablo. This class has functions for reading models, images, distance maps, etc. (by calling member functions of the respective objects). **P3DControl** class also contains member functions for executing the various stages in the Pablo model fitting process, like `binaryPabloFigureStage()`, `binaryPabloAtomStage()`, etc. This class thus interacts with all other projects and classes.

This project also contains an *Optimizer* for each stage of the optimization process (like **M3DFigureOptimizer**, **M3DSRepOptimizer**, etc.); these *optimizers* are an interface between the **P3DControl** class and the numerical methods in the PAUL_CODE project. An *Optimizer* defines a *Problem* (like **M3DMainFigureProblem**, **M3DSRepProblem** etc.) which is a class that calls the optimization functions in PAUL_CODE project. These *Problems* also call *evaluate()* functions which are members of the **Match** class and help to evaluate how well the current s-rep model fits the target data (binary image or distance map).

SEURAT

This project, initially coded by Andrew Thall, contains functions for rendering and display of models, and their implied surfaces and boundaries (the *Wired Mesh*, *Solid* etc. options in the Pablo model display option). The classes used for storage and display of s-rep models in this project are different from those used at other places in the optimization code (except for the code for rendering of *quasi-tubular objects*, written by Rohit Saboo, which uses the same classes). Instead of the **M3DPrimitive** class, the program for displaying and modifying s-reps within the Pablo display window uses the **XferAtom** class to represent an s-rep atom/hub. The program for calculation and display of surfaces and wire-meshes uses the **Diatom** to represent the s-rep atom/hub.

The **Mesh** class is used to store and display wired meshes. The **CCSubdivSurf** represents implied surfaces by the s-rep model, and the **Pointlist_Server2** generated a point cloud on the surface boundary of the model. These classes are used for generation and display of object boundaries (as well as for interpolation)

PLANES

Code for display and manipulation of the *cut-planes* viewing method for 3D images in Pablo.

INCLUDE

Code for saving and loading user preferences

Input Output classes for Images

ioAnalyze / ioGipl / ioMeta / ioPlanim

These projects contain IO functions for the following types of images (with their respective extensions)

Image Format	File extension
Analyze	.hdr
Gipl	.gipl
Metaheader	.mhd, .mha
PlanIm	.pim
diCom	.dcm
RAW3 (UNC image format)	.raw3

Note:

- Metaheader images contain a header which contains a pointer to the image data, which is typically in a .raw or .zraw format (compressed)
- Code to read RAW3 files is in the **m3d** project because of historical reasons (it was the first image format that Pablo could operate on)

mixedIO

This is an interface that calls the right type of IO functions depending on the type of the image

ioUtilities

Various utility functions for image IO.

VERSION

This class Keeps track of the Pablo version.

ProAdvice

version is a separate program that is run to figure out what version date to embed in a compilation of pablo. it scans the dates of checked-out source code and deduces the latest date found, then overwrites the revision string in ***pablo_version.h*** file for use in compiling the Pablo tree.

ZLIB

This is an external library that is used for image compression and decompression (reading and writing *.zraw images).

> Caution: The zlib library often has problems while running on different Windows platforms. Usually Pablo performs perfectly well unless the input images are compressed files.

zlib is copied straight from the open source gzip utility, without permission. It is used to compress voxel intensity in an image slice. Compression never spans across slices so that we can read a single slice from the middle of a large 3D image quickly.

ProAdvice

SConscript files are part of the *SCons build* system, which is an alternative to 'make'. This was used by Rohit to generate a *SWIG Pablo binding* that could be used from Python (and other interactive languages, potentially). *SWIG* converts internal data stored in classes to Python objects and allows one to call class methods directly from Python, which is both very cool and requires no compiling to whip up a quick optimization.

AE2: The *Anatomical Structure Editor 2* configuration of Pablo can be ignored. [It builds *ConStruct*, an interactive segmentation and registration app. If you compile the AE2 configuration by mistake, it is designed to fail to run as a reminder not to use it.]

Cluster: Pablo2 lacks support to run on cluster of computers.   There was an experimental version to run each atom via MPI, and synchronize them after each iteration, but that did not give us enough of a speedup to warrant the complexity. Instead, lots of people have leveraged coarse-grain multiprogramming techniques to run several pablo instances at once by scripting it in perl or bash or even make (using it's -j switch to run multiple jobs at once). Investigate the Emerald cluster on UNC's campus (open to all North Carolina researchers), and the BASS cluster (open to all UNC bio-medical researchers).

III - DETAILS OF PABLO PROJECTS / LIBRARIES

1. GUI project

The GUI project is the entry point for Pablo code execution. The two major functions of this library are as follows.

- Parsing of input parameters
- Pablo GUI management

1.1 Parsing of Input Parameters

Function pablo(…)

The `pablo()` function in Pablo.cpp does the work of parsing the input parameters to the Pablo program. Command-line tuning parameters override script values, if they are placed after the -cs option on the command. If placed before -cs, then a script value overrides the command-line value.

All input parameters (command-line as well as config. Script) are read into a **Registry** object, *scriptParms* defined in `pablo()`. Once the input parameters have been parsed, there is an input validation check. Some of the validation checks are as follows.

1. If optimize (-po) option is specified, then there must be a scriptFile name (-cs) specified with it.
2. If the interactive mode (-I) is selected, then optimize mode (-po) must also be selected.
3. Interactive mode (-I) and align mode (-pa) cannot be used at the same time.
4. Optimize mode (-po) and align mode (-pa) cannot be used at the same time.

`pablo()` uses the `getarg(…)` function in *pablo.cpp* for parsing input parameters.

control is a pointer to a **P3DControl** object defined inside the pablo() function. Once input parameters have been parsed and stored in *scriptParms*, the program then passes execution to *P3DControl.cpp* via the *control* object. *scriptParms* is passsed as an argument to the function `P3DControl::runBinaryPablo(…)`.

1.2 Pablo GUI management

Pablo uses FLTK library for GUI management. The major classes for Pablo GUI management are **P3DUserInterface** class and **P3DUserInterfaceCallback** class.

The **P3DUserInterface** class contains pointers to GUI objects like buttons (**FL_Button**), sliders (**FL_Slider**), menus (**Fl_Menu_Bar**), windows (**Movable_FL_Window**) etc. It also contains functions for callback for these GUI objects. These functions eventually call respective functions within the **P3DUserInterfaceCallback** class. **P3DUserInterface** also contains code for creating new windows and their associated child objects (buttons etc.).

For example, the following is a declaration of an **FL_Button** within P3DUserInterface and its callback functions.

```
Fl_Button * rotateRight90 ;

void cb_rotateRight90_i(…) {

        // Transfer to P3DUserInterfaceCallback
        P3DUserInterfaceCallback:: rotateRight90();
}

static void cb_rotateRight90(…) {

        P3DUserInterface::cb_rotateRight90_i(…) ;

}
```

There is a static function `cb_<GUI object name>(…)` in **P3DUserInterface** which calls another function `cb_<GUI object name>_i(…)` in **P3DUserInterface** which calls the `<GUI object name>(…)` function in the **P3DUserInterfaceCallback** class.

The **P3DUserInterfaceCallback** class contains callback functions for various GUI objects. Most of these functions eventually transfer control to the **P3DView** class (within the GUI project) or the **P3DControl** class (REGISTER project).

The **P3DView** class contains functions to control the view of the model or objects in the Pablo view window. Buttons like ***A, S , C*** for setting the object to *Axial, Sagital* and *Coronal* views are examples of GUI objects whose callbacks are finally handled by the **P3DView** class. Also all transformations of the object, like rotation, translation, scaling etc. are handled by **P3DView** class functions.

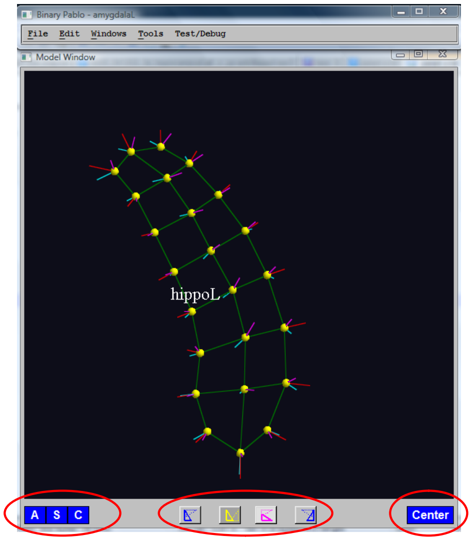


Figure 9: Pablo GUI. Encircled buttons have callbacks in the **P3DView** class

1.3 List of Command-line Input Options in Binary Pablo

In the command line, the input options are specified in the following format:

```
-bin_pablo_d.exe  - <option name 1> <option value 1>  - <option name 2> <option value 2> …
```

The option name can be a long name or a short name. Following is a list of Binary Pablo Command-line input options.

Short name	Long name	Value	Description
h	help		Displays list of command-line parameters
ii	image	Filename	Load image
id	distanceMap	Filename	Load distance map (recently activated by DG)
il	landmarkModel	Filename	Load Landmark model
im	model	Filename	Load m3d model (also with PGA/CPNS stats in it)
ix	transform	Filename	Load similarity transform
ip	pga	Filename	Use the PGA data from the specified model file discarding any PGA data loaded by -im or -model
ipr	PGAModel		Load a residue PGA model (figure or atom stats)
ic	contour	Filename	Load contour file
cd		Directory	Specify location of pablo.txt control file
caf	altFormat	Filename	Convert image file type based on extension and halt
cafc	altFormatc	Filename	Same as above but output compressed
cf	format	N (int)	Convert image file version to N and halt
cs		Filename	Load parameters from config script file
ci	ignore		Ignore any unknown parameters found in a script.
om	outModel	Filename	Save fitted model
omi	outModelImage	Filename	Save fitted model’s binary image
ot	outTile	Filename	Save BYU tiles of fitted model
oit	outImageTile	Filename	Save BYU tiles of binary surfaces within image
ostm	outSimTransModel	Filename	Save similarity transform after init
ox	outTransform	Filename	Save any transformation produced
vf	videoFolder	Directory	Save models in intermediate optimization phases to produce video (by DG)
pc	compress	Filename	Compress the image file, then quit
ph		Filename	Print header of image file; load image and continue
pa	align	Filename	Align using points from the file, then quit;
po	optimize		Optimize, then quit; open no windows
ps			Run landmark based diffeomorphism helper, then quit
pr			Print control script defaults, then quit
pd			Print control script descriptions, then quit
pq	quit		Verify arg syntax, then quit
pi	resample	Filename	Isotropically resample the image file, then quit
pm	extent	Filenames	Print max extent of all image files, then quit
pu	uncompress	Filename	Uncompress the image file, then quit
hc			Print change history, then quit
lV	version		Print the version number
lv	verbose		Verbose: print more than the usual info
lq	quiet		Quiet: print less than the usual info, mostly only errors
l	interactive		Show the scripted optimization running in the GUI. This may only be used with -cs and –po

Note: Inputs can specified at the command prompt or inside a *script* (an ASCII file referred to as the *config. script*; *-cs* option in the table above). The *config. file* also contains values of different tuning parameters used in Pablo.

2. M3D project

This project contains data structures for storage and manipulation of s-rep models at different levels of the hierarchy (object, figure, atom, spoke etc.). The prefix M3D stands for Model 3D and the prefix P3D stands for Pablo 3D. Classes **M3DObject**, **M3DFigure**, **M3DPrimitive** and **M3DSpoke** are the basic classes for representing these objects.

2.1 Pablo Model Hierarchy

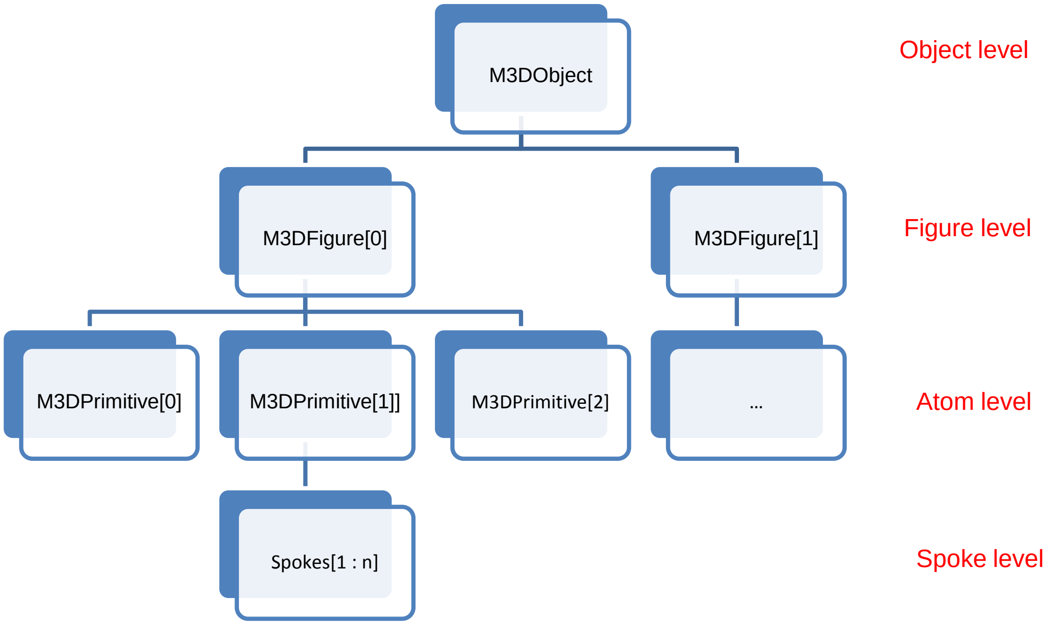
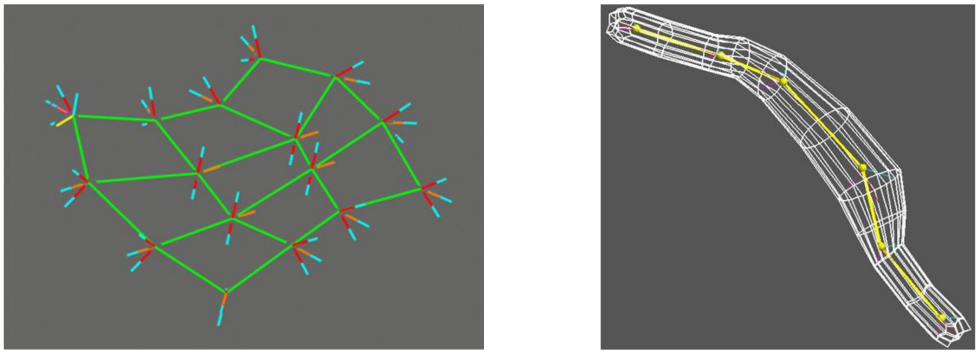


Figure 10: S-Rep object hierarchy in Pablo

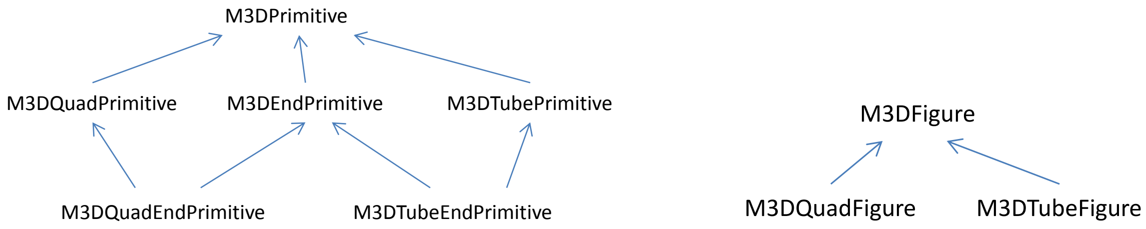
Figure 10 shows s-rep object hierarchies in Pablo. An s-rep (**M3DObject**) is represented by one or more figures (**M3DFigure**), which consist of atoms (**M3DPrimitives**). As seen in Figure 11, when the object is represented by a quasi-slabular s-rep, the figure (**M3DQuadFigure**) consists of a rectangular grid of atoms; when represented by a quasi-tubular s-rep, the figure (**M3DTubeFigure**) consists of a vector of atoms.



(a) Quasi-slabular s-rep (**M3DQuadFigure**) (b) Quasi-tubular s-rep (**M3DTubeFigure**)

Figure 11: Types of figures in Pablo

Also, we can observe in Figure 11 that there are differences in the atoms in **M3DQuadFigure** and **M3DTubeFigure** classes. Atoms in an **M3DQuadFigure** belong to the **M3DQuadPrimitive** class, whereas those in an **M3DTubeFigure** belong to the **M3DTubePrimitive** class. An **M3DQuadPrimitive** consists of two or three spokes, whereas an **M3DTubePrimitive** consists of a wheel of spokes. The *end primitives* are special types of atoms in both types of figures. An *end primitive* in an **M3DQuadFigure** belongs to the **M3DQuadEndPrimitive** class whereas one in an **M3DTubeFigure** belongs to the **M3DTubeEndPrimitive** class.



(a) Hierarchy for atom classes (b) Hierarchy for figure classes

Figure 12: Class hierarchies in Pablo

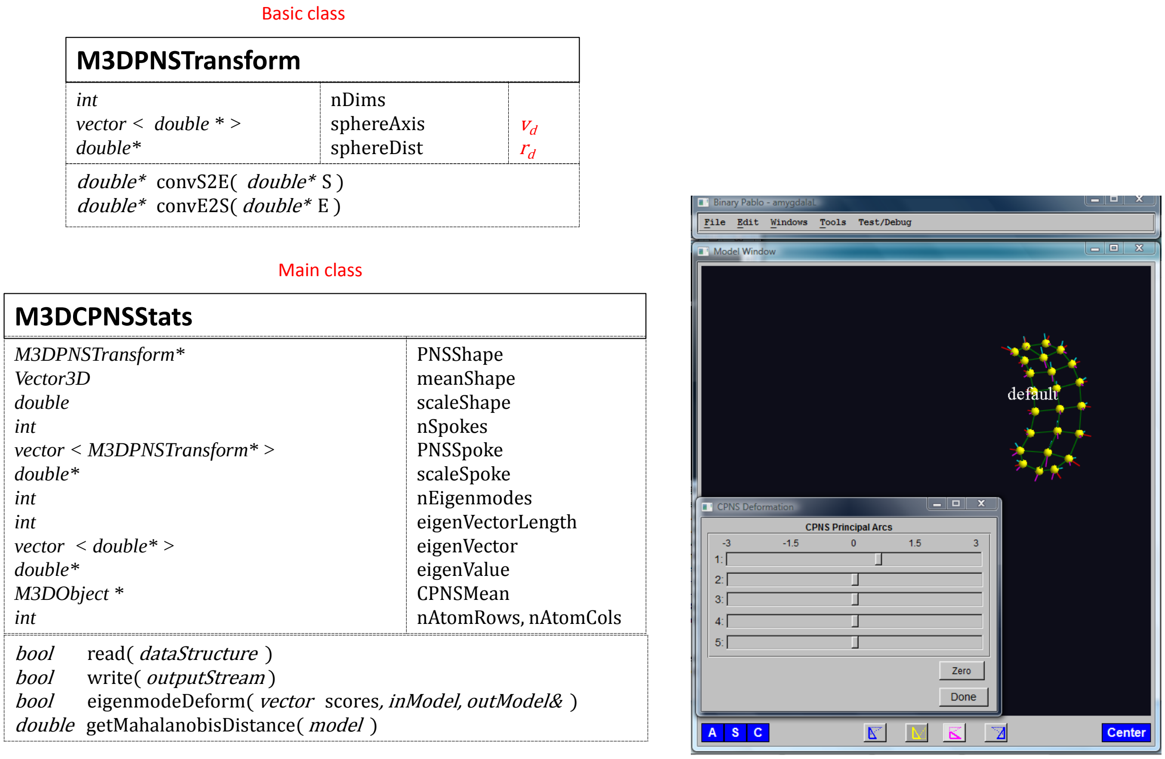
Figure 12 shows the hierarchy of classes in Pablo. As shown in Figure 12(a), **M3DQuadPrimitive**, **M3DEndPrimitive** and **M3DTubePrimitive** inherit from the **M3DPrimitive** class (abstract); **M3DQuadEndPrimitive** inherits from **M3DQuadPrimitive** and **M3DEndPrimitive** classes. Figure 12(b) shows that both types of figures inherit from the same **M3DFigure** class (abstract).

2.2 PGA and CPNS Statistics in Pablo

An **M3DObject** that represents a statistical mean also contains pointers to statistics (eigenmodes) that generated the mean. The objects that represent such statistics are **M3DPGAStats** and **M3DCPNSStats**. While the former stores statistics generated by *PGA (Principal Geodesic Analysis)* on an m-rep population, the latter stores statistics generated by *CPNS (Composite Principal Nested Spheres)* on an s-rep population. The Pablo GUI has ways to observe *eigenmode deformations* for both type of statistics under Tools →PGA Deformation and Tools → CPNS Deformation.

PGA Statistics can be computed for both figures as well as individual atoms. While **M3DPGAStats** represents the PGA statistics for a figure, **M3DPrimitivePGAStats** represents that for an atom. PGA data is used in deformable model fitting through a *Mahalanobis stage* and calculation of *Mahalanobis distance penalty* at every optimization stage.

Figure 13(a) shows the Software Design of CPNS statistics in Pablo. The basic class for CPNS statistics is **M3DPNSTransform**, a transformation (called *PNS Transformation*) that maps data from a spherical space to Euclidean space via *PNS (Principal Nested Spheres)*. **M3DCPNSStats** represents the CPNS statistics for an entire figure (**M3DQuadFigure** only). It contains a pointer to a *PNS transformation* for the shape PDMs and a vector of pointers to *PNS Transformations* for each of its spokes. Additionally it contains the eigenmodes for shape variation within the training population.



Basic class

M3DPNSTransform		
int	nDims	
*vector < double * >*	sphereAxis	v_d
*double**	sphereDist	r_d
*double** convS2E(*double** S)		
*double** convE2S(*double** E)		

Main class

M3DCPNSStats	
*M3DPNSTransform**	PNSShape
Vector3D	meanShape
double	scaleShape
int	nSpokes
vector < M3DPNSTransform >*	PNSSpoke
*double**	scaleSpoke
int	nEigenmodes
int	eigenVectorLength
vector < double >*	eigenVector
*double**	eigenValue
*M3DObject **	CPNSMean
int	nAtomRows, nAtomCols
bool read(*dataStructure*)	
bool write(*outputStream*)	
bool eigenmodeDeform(*vector* scores, *inModel, outModel&*)	
double getMahalanobisDistance(*model*)	

(a) CPNS Software Design in Pablo

(b) Snapshot of CPNS Deformation in Pablo

Figure 13: CPNS in Pablo

Figure 13(b) shows a snapshot of Pablo while observing CPNS eigenmode deformations. By varying the sliders in the *CPNS Deformation* window, a CPNS mean model can be deformed along its principal modes. The deformed model is displayed in the Pablo display window.

Matlab Code for computing PGA and CPNS statistics

The MIDAG group at UNC has a software package in Matlab known as *pablo_matlab*. The *shapeStat* library in this package has code to compute the mean and eigenmodes for CPNS as well as PGA. The Matlab programs *calculatePGA.m* and *calculateCPNS.m* calculate PGA and CPNS statistics for a population of fitted *s-reps* (or *m-reps*) placed in a directory. The instructions to run these programs can found in the text files README_PGA.txt and README_CPNS.txt placed in the 'pablo_matlab/shapeStats' directory.

Note:

- The *calculatePGA.m* program presently only works for *m-reps* and not *s-reps*. The inputs to the program, the .m3d files can be in the new model format [Figure 3(b)] or the old model format [Figure 3(a)] but as an intermediate step they are converted to the old format, and so the result is an *m-rep* and not an *s-rep*.

- The *calculateCPNS.m* program presently works for both m-reps as well as s-reps, provided the input models, the .m3d files are written in the new format [Figure 3(b)]. To convert a .m3d file written in the old format to the new format, open that file in Pablo (File→Load Model(s)) and save the model by selecting File→Save Model As. The model would be saved in the new format.

2.3 Miscellaneous classes in M3D project

The M3D project contains numerous utility classes; **M3DObjectFile** helps in reading and writing s-rep models; **M3DObjectRenderer, M3DFigureRenderer** and **M3DPrimitiveRenderer** render these objects in the Pablo GUI display window. There are also classes for matrices and vectors, like, **Matrix4D, Vector3D** etc.

The class **Image3D** is used in Pablo to store a 3D image (binary image, grayscale image, distance maps etc.). The data in the image is stored by an array of *unsigned short*'s. **RAWImageFile** is the class that handles IO as well as other operations for the RAW3 image format internally used at UNC. **BYU** class supports IO for *BYU Tiles* in Pablo.

HISTORY BITS

M3DObjectFile is a factory that creates instances of **M3DObject**. Other classes ending in "File" were intended to be factory classes but refactoring did not take place (students graduated).

Image3D contains unsigned intensities in RAM. To simplify the design, it was supposed to be able to read/write signed values but did not have enough runtime info to decide if which case to apply. It contains shift and scale parameters to allow a programmer to override the default behavior, but most programmers just set shift=0 and scale=1.0. An interesting example is *distance maps*, which contain signed intensities that are 8*pixelDistance. The values are interpreted by the distance code as signed even though the IO and image display code both use the intensities as unsigned, and the scale cannot be used because we're scaling the right direction (we want to represent values less than 1).

3. MATCH project

This project contains data structures to compute and store the geometry as well as image match between a model and an image.

3.1 Match class

The **Match** class is one of the most vital classes in Pablo code. It uses data structures like **matchResult** to store match results for various stages in Pablo optimization; these results are arrays with the names *figureResults, atomResults, sRepResults* etc. It contains a pointer to the **M3DObject** that we are trying to evaluate (known as the *referenceObject*), a pointer to the target image (named *targetImage*) as well as a pointer to the **ImageDistanceMap** of the target image (known as *binaryDistanceMap*).

Table 1(a): Geometry and Image mismatch penalties in model fitting

GEOMETRIC MISMATCH

1. Mahalanobis distance from the statistical mean
2. Deviation from the model at the beginning of the optimization stage
3. Spoke crossing penalty (S_{rad} penalty): spokes should not cross; avoid illegal shapes
4. High curviness penalty: avoid illegal shapes
5. Grid irregularity match (Atom stage and S-rep spoke stage only): penalize irregularity in atom grid

DATA MISMATCH

1. Image boundary mismatch (binary image match): spoke ends should be at the boundary of target object
2. Orthogonality mismatch (binary image normal match): spoke direction should be boundary normal
3. Crest plane mismatch (Image Plane Orient Match At End): Crest plane orientation mismatch
4. Crest plane vertex mismatch (Binary Image Vertex Match): End spoke vertex should be at the crest

Table 1(a) states the Geometry and Image mismatch terms used in deformable model fitting in Pablo. Currently Pablo can only interpolate the boundary points in an *s-rep*, but not the spokes. Hence the Data Mismatch terms 2, 3 and 4 [Table 1(a)] are done only on the level-0 interpolated *s-rep*, but with the integration of Jared Vicory's spoke interpolation code (formulated originally by Qiong Han), Pablo would have the capacity to compute these penalties at levels higher than 0.

Table 1(b) has descriptions of some member functions in the **Match** class. These member functions are available for calculation of mismatch between the *reference object* and the *target image* at every stage of the optimization process.

Table 1(b): Examples of Member functions in **Match** class

Functions to compute total penalty of mismatch (geometry + data) at a particular stage

`computeFigureMatchAndPenalties()`: compute mismatch for entire figure

`computeAtomMatchAndPenalties()`: compute mismatch for particular atom

`computeSRepMatchAndPenalties()`: compute mismatch for particular spoke in s-rep stage

`computeSpokeMatchAndPenalties()`: compute mismatch in spoke stage (tube m-reps)

Functions to compute particular penalties

`computeBinaryImageMatch()`: compute order-0 mismatch between *reference object* boundary and *target image* boundary for entire figure

`computeAtomMahamPenalties()`: compute Mahalanobis distance between the *reference object* and the *sample mean* for a particular atom

`computeSpokeBinaryImageNormalMatch()`: compute order-1 mismatch between *reference object* spoke direction and *target image* boundary normal for a particular spoke in s-rep stage

`computeAtomBinaryImageVertexMatch()`: compute order-2 mismatch between *reference object* end spoke crest and the *target image* crest for a particular atom in atom stage

<u>Note:</u>

- There is a slight confusion in the way penalty functions are named for the *s-rep spoke stage* of Pablo optimization. At the *s-rep stage*, the attempt is to calculate penalties for each spoke; so the penalty functions are named `computeSpokeBinaryImageMatch()`, `computeSpokeBinaryImageNormalMatch()`, `computeSpokeBinaryImagePlaneOrienMatchAtEnd()` and `computeSpokeBinaryImageVertexMatch()`. These functions are not associated with the *spoke stage*. The only functions associated with the *spoke stage* are: `computeSpokeMahamPenalties()`, `computeSpokeMatchAndPenaltiesAsComponents()`, and `computeSpokeMatchAndPenalties()`.

The **MatchUtility** class also holds results of the match process.

3.2 ImageDistanceMap class

The Match project also contains class **ImageDistanceMap** for storing and manipulating a signed distance map.

Facts related to distance maps:

- The name of the distance map can be passed by the option –*distanceMap* in Pablo command line.

- The distances in the distance maps used by Pablo are in voxel units. It is useful to know that distances for some entities in Pablo are in voxel units, whereas others are in normalized units [0, 1] (for example, the radius of a spoke etc.).

- **ImageDistanceMap** provides access methods to get the distance at a particular place in the image via the `getDistance()` function and to get the gradient of distance at a particular place via the `getGradDistance()` function.

- The gradient at a point in the distance map represents the direction of the normal at that point for that particular level set of the map. There is code in Pablo to calculate the curvature of a level set of the distance map at a point using second derivatives of distances. The detection of ridges and crests involve the gradient of curvature; that is carried out in code using third derivatives of distances. Thus penalty functions of order greater than one [Data Mismatch penalty 2, 3 and 4 in Table 1(a)] can be calculated using distance maps.

- The **ImageDistanceMap** class stores the distance map as an **Image3D** object. **Image3D** images store information in an *unsigned short* format (short integer), whereas distance map entries are *signed double* values. The way a *signed double* d is converted to an *unsigned short* ui (16 bits) is by
    - multiplying *d* by 100
    - storing the two's complement of negative numbers

  Earlier this used to be a multiplication by 8 but now it is by 100. These manipulations are done while writing and reading distance maps; the developer does not have to deal with them if he just wants to use the distance maps.

  Note: As we use the first, second as well as third derivatives of distances in the distance map, there is a high precision requirement for these distance values. Even after the multiplication by 100, we fail to provide with sufficient precision for correct calculation of third derivatives. Hence, eventually these distance maps have to be modified to store *doubles* or *floats* instead of *shorts*.

3.3 BpTuning class

The tuning parameters in Pablo are stored and globally available through a **BpTuning** class object named *tuneVals*. The **BpTuning** class stores parameters in a data structure named **BpTune_t**, which is an enumerated list of possible parameters. The values of these parameters are set in the `bpTuning::initialize()` function in *BpTuning.cpp*. The order in which the parameters are enumerated in **BpTune_t** and the `bpTuning::initialize()` must be the same. Table 2 below describes the process of adding a tuning parameter to Binary Pablo.

Parameters can be passed through the Pablo command line or they can be specified in the Pablo config script file. Command-line tuning parameters override script values, if they are placed after the -cs option on the command. If placed before -cs, then a script value overrides the command-line value.

The global function `tuningWt(<name of parameter>)` gives the value of any parameter at any place in the program. Table 3 in the next page describes the process of adding a new penalty function in Pablo. Table 1 shows some of the typical penalty functions that are used in Pablo.

ControlParms is an alternative to tuning parameters that is specific to the GUI, eg, (x,y,width,height) to place GUI windows.

3.3.1 HOW TO ADD A NEW TUNING PARAMETER IN BINARY PABLO

- Add the parameter with the list of parameters in **BpTune_t** list
- Add the initialization of the parameter with its default value in `bpTuning::initialize()` function

Note: The order of the new parameter must be the same in both the steps above.

- Add the parameter with its value in the cofig script file in Pablo (ASCII file).

Table 2: Procedure for adding a new tuning parameter in Pablo.

HISTORY BITS

In Pablo v1, *tuning parameters* -- double-precision floats that act as runtime switches or weights -- were passed from the GUI thru the controller to the model/view, and were touched by up to 8 routines, most of which only passed the values to other routines. This was replaced by a singleton of the **Tuning** class (**BpTuning**) where all values are kept and accessed globally. This vastly simplifies how much coding is required to add a parameter to only three places so a new value can be added in five minutes: the tuning cpp and h files that define the value, and the code that needs to use the value. The values are indexed by a compile-time key and each value is a double which can be interpreted as bool's or int's if needed. Unfortunately, strings are not supported -- see command-line and the config file for that. The most important part of adding a tuning parameter is to give it a good description so that others can figure out what it means, eg, avoid "josh's 4th tuning parameter"

3.3.2 HOW TO ADD A NEW PENALTY FUNCTION IN BINARY PABLO

- Add the new penalty function as a member function of the **Match** class.

- Add new parameter(s) which would be the tuning weight of this particular penalty, following the instructions in Table 2.

- In *Match.h*, depending upon which fitting stage the new function penalty was added to, add +1 to one of the following variables:

    - `MAX_NUM_FIG_MATCH_RESULTS`
    - `MAX_NUM_ATOM_MATCH_RESULTS`
    - `MAX_NUM_SREP_MATCH_RESULTS`
    - `MAX_NUM_SPOKE_MATCH_RESULTS`

- In *Match.cpp*, depending upon which fitting stage the new penalty function was added to, change the inputs to the appropriate `assert()` function (which verifies the maximum number of results in a particular stage).

- In *Match.cpp*, depending upon which fitting stage the new penalty function was added to, initialize the new penalty result value in one of the following data structures:

  `figureResults, atomResults, sRepResults, spokeResults`

  This initialization takes place right after the `assert()` expression in *Match.cpp*

- Depending upon which fitting stage the new penalty function was added, add the new penalty function within one of the following member functions of the **Match** class:

    - `computeFigureMatchAndPenaltiesAsComponents()`
    - `computeAtomMatchAndPenaltiesAsComponents()`
    - `computeSRepMatchAndPenaltiesAsComponents()`
    - `computeSpokeMatchAndPenaltiesAsComponents()`

  The new penalty can be added by following the same format in which all other penalties have been added in the respective function.

- Depending upon which fitting stage the new penalty function was added, add the printing of the new penalty value within one of the following member functions of the **P3DControl** class:

    - `binaryPabloFigureStage()`
    - `binaryPabloAtomStage()`
    - `binaryPabloSRepStage()`
    - `binaryPabloSpokeStage()`

  The new penalty can be printed to the screen by adding it to the *switch* statement (in the same way as the other penalty values are printed) in the respective function.

Table 3: Procedure for adding a new penalty function in Pablo.

4. REGISTER project

The major functionality of this project can be divided into the following categories.

4.1 P3DControl class

This project contains the **P3DControl** class (*Pablo 3D Control*) which controls the entire functioning of Pablo. This class has functions for reading models, images, distance maps etc. (by calling member functions of the respective objects). **P3DControl** class also contains member functions for executing the various stages in the Pablo model fitting process, like `binaryPabloFigureStage()`, `binaryPabloAtomStage()` etc. Most of these functions in turn call functions of the respective classes to perform the actual task; thus **P3DControl** acts as a controller class. Table 4 below gives descriptions for some of the member functions in the **P3DControl** class. The functions shown in Table 4 are only a handful of functions; **P3DControl** contains more than a hundred member functions as every operation in Pablo gets done through this controller class.

Table 4: Examples of Member functions in **P3DControl** class

Major functions to run entire Pablo operation

- `runBinaryPablo()`: complete the operation of Binary Pablo optimization
- `runThinPlateSpline()`: complete the operation of thin plate splines

Functions to load (read) and save (write) data (models, images etc.) during optimization

- `read()`: read a model file (*.m3d) and also load statistics (PGA or CPNS)
- `loadImage()`: load an image
- `loadDistMap()`: load a distance map (anti-aliased as well)
- `binaryPabloLoadLandmarkModel()`: load a landmark model file (*.lm)
- `write()`: write a model file (*.m3d) and associated statistics (PGA or CPNS)

Functions that are called from Pablo GUI (control passed from **P3DUserInterfaceCallback**)

- `loadTileSet()`: load a tile set (*.byu). Called from File →Load Tile Set
- `addQuadFigure()`: add a new quad figure. Called from Edit →Add Quad Figure

Functions to carry out optimization in various stages

- `binaryPabloFigureStage()`, `binaryPabloAtomStage()`, `binaryPabloSRepStage()`, `binaryPabloSpokeStage()`

Functions for statistics

- `cpnsDeform()`: to carry out CPNS eigenMode deformation
- `pgaDeform()`: to carry out PGA eigenMode deformation

4.2 Optimizers and Optimizer Problems

The REGISTER project also contains an *Optimizer* for each stage of the optimization process (like **M3DFigureOptimizer**, **M3DSRepOptimizer** etc.); these *optimizers* are an interface between the **P3DControl** class and the numerical methods in the PAUL_CODE project. An *Optimizer* defines a *Problem* (like **M3DMainFigureProblem**, **M3DSRepProblem** etc.) which is a class that calls the optimization functions in PAUL_CODE project. These *Problems* also call *evaluate()* functions which are members of the **Match** class and help to evaluate how well the current s-rep model fits the target data (binary image or distance map).

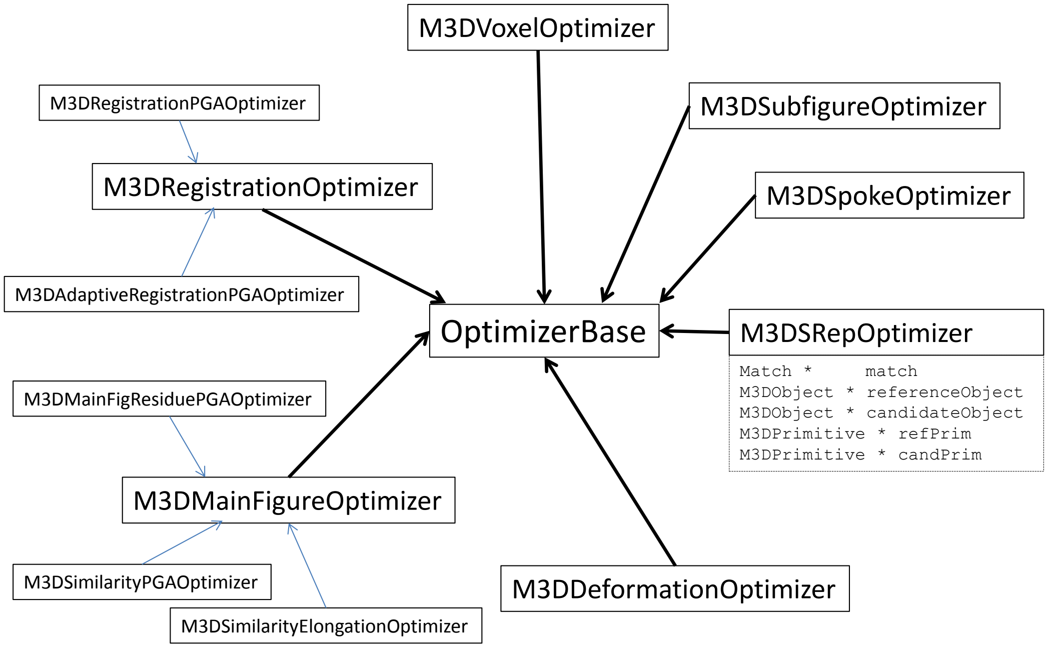


Figure 14(a). Class hierarchy for Optimizers in Pablo REGISTER project

Figure 14(a) shows the class hierarchy for *Optimizers*. All optimizers inherit from the **OptimizerBase** class. They contain pointers to a **Match** class object which helps to evaluate the *current model* (**M3DObject** * *candidateObject*) after every step of optimization.

Figure 14(b) shows the class hierarchy for *optimizer problem* classes. Each Optimizer has an associated Problem class which is an optimization problem defined in Euclidean space. These problem classes inherit from the **Function** class, which is an optimization problem defined in Euclidean space; **Function** class is defined in *Optima.h* and *Optima.cpp* in the PAUL_Y project and is part of the optimization infrastructure.

Note:

The optimization structure in Pablo is very similar to the one described in the book “Numerical Recipes”.

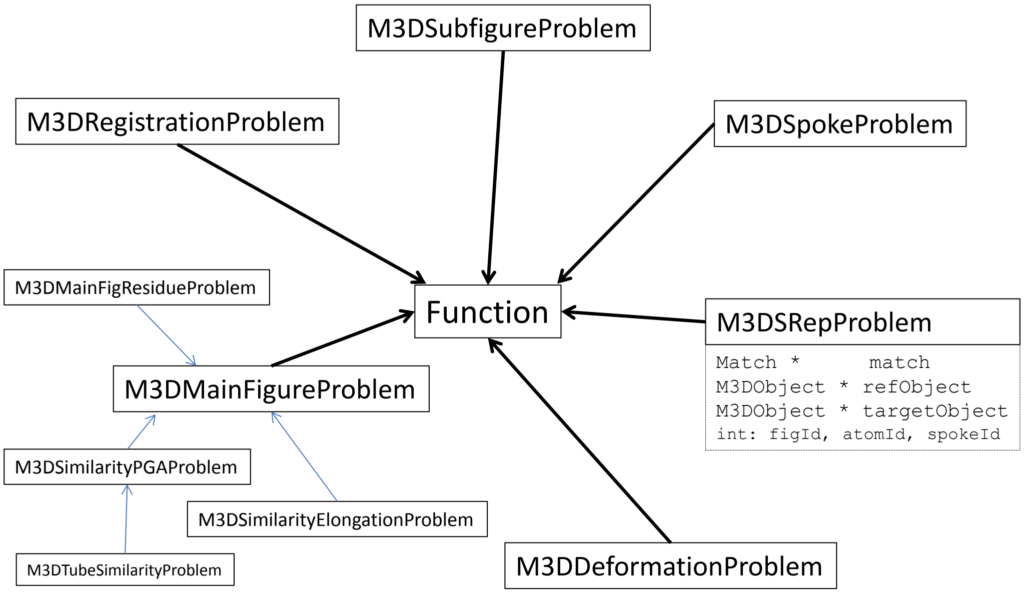
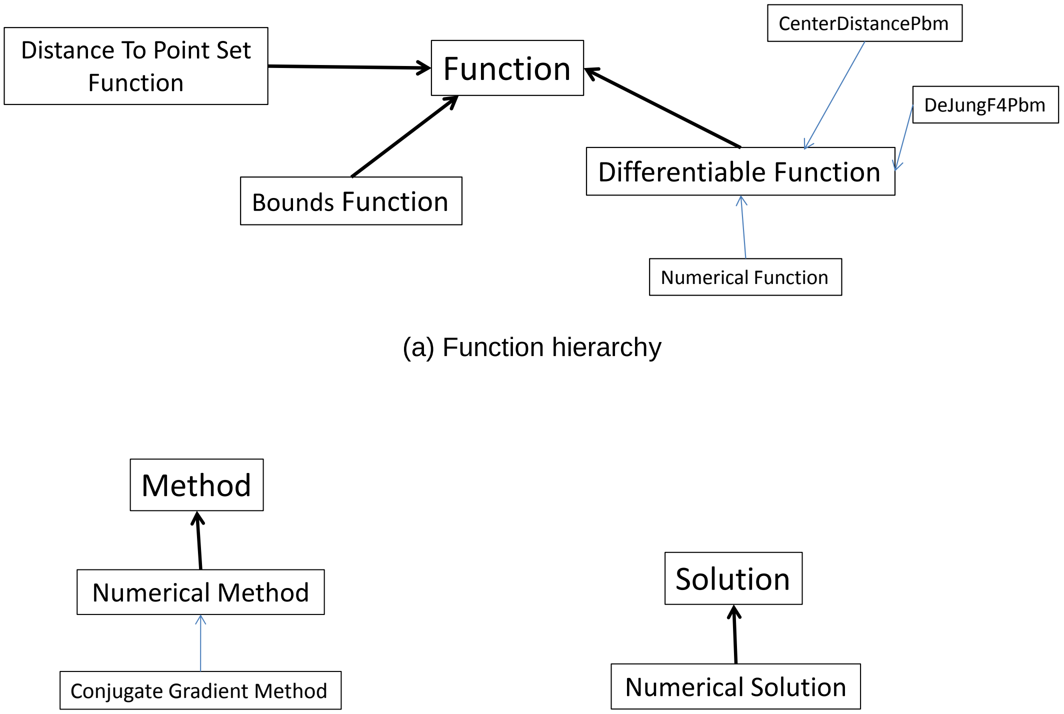


Figure 14(b): Class hierarchy for Optimizer Problems in Pablo REGISTER project

ProAdvice

Optimizer Passes: The *optimizers* are written with much support for debugging. This means that several passes thru the code are taken when a single pass with an internal loop may have sufficed. This allows additional debuggery to be added at the top levels, such as redrawing the model after each atom movement, without mucking with the low-level code. One can also print out values from the upper loops, save intermediate models or other param's such as for plotting convergence (or non-convergence!), or stop the optimizer prematurely based on time spent or patience endured.

Atom stage optimization is multi-pass for an additional reason: the order in which atoms are optimized can change. Some designs required sequential atom order, some were random, and some assigned a probability that an atom would be processed in any particular pass based on how much it changed the last time. Finding a scheme that works for a particular fitting project is mostly magic.



(a) Function hierarchy

(b) Method hierarchy

(c) Solution hierarchy

Figure 15: Class hierarchies in PAUL_CODE project

5. PAUL_CODE project

5.1 Numerical Methods

This project (written initially by Paul Yushkevich) contains the infrastructure for performing optimization in Pablo. Contains classes for optimization problems (like **Problem, NumericalProblem** etc.), optimization functions (like **NumericalFunction**, **DifferentiableFunction** etc.), optimization solutions (like **Solution, NumericalSolution** etc.) and methods (like **ConjugateGradientmethod, BrentLinearMethod, SimplexMethod, EvolutionaryStrategy** etc.).

Figure 15 above shows some class hierarchy diagrams for different types of objects in the PAUL_CODE project. Facts about these classes:

- **Function** is an optimization problem defined on Euclidean N-space

- A **Differentiable Function** (inherits **Function**) is a gradient problem, a optimization problem defined on Euclidean N-space, for which there must be a way to compute the gradient.

- **Numerical Solution** is a vector, a solution to a numerical problem

5.2 The Registry class

The PAUL_CODE project contains the **Registry** class, a powerful data structure used throughout the Pablo program. A **Registry** is a hash table which stores (key, value) pairs; keys are strings and values can be *integers*, *doubles*, *arrays* or *folders*. Registries can also be read from and written into ASCII files which can be hand-manipulated. Examples of Registry entries include tuning parameters for optimization and various control flags for Pablo operation.

PABLO REGISTRY DATA FORMATS

Regular value key:	keyName = value ;
Array key:	keyName = { ArrayLength ArrayType ArrayValues } ;
Folder key:	keyName { ... }

Array Types = { IntArray (0), FloatArray (1), DoubleArray (2) }

PABLO REGISTRY ACCESS FUNCTIONS

Registry::getIntValue(keyStr, defaultVal)
Registry::getDoubleValue(keyStr, defaultVal)
Registry::getDoubleArray(keyStr, & length)

Table 5: Details of the **Registry** class

The **Registry** class allows key value pairs to be stored in an ASCII file, called a *registry file*, which may be later read by this class. The class instance may then be queried to obtain the values for specific keys. The keys should be printable ASCII strings. The first letter of key names must be upper case, and if a lower case example is encountered in a registry file, it will be converted to upper case on input before other processing. Registry files are intended to be hand-editable.

Table 5 shows the three kinds of data formats that can be read into and written out of a *registry file*. Those are values (int/double), arrays, or folders. Usually the closing brace of folder keys is on a later line. The ellipsis consists of numerous key-value pairs, including other folder keys.Registry files may also contain single-line comments indicated by the first non-whitespace character being the pound sign, '#'. They may also contain statements for including other files, of the form :#include file-path.

Functions for storing and retrieving keys from an instance of this class are provided. Table 5 also shows some typical retrieval functions that are available in the **Registry** class. Deletion of regular value keys is supported. Functions to read from or write to a registry file, as well as for performing a number of utility operations are also provided.

A value of regular key may be boolean, integer, float, double or a string. Almost any legal strings may be passed as values to the functions of this class. In particular, strings should not contain numeric escape sequences (for example \012 or \0A to represent \n). Any non-printable ASCII characters, such as NEWLINE, will be encoded as 2-digit escape sequences in output registry files. On input, the original ASCII string will then be recovered. Thus 3-digit escape sequences will cause errors in the parsing of registry files and 2-digit escape sequences will not be treated as one might expect.

6. SEURAT project

This project, initially coded by Andrew Thall, contains functions for rendering and display of models, and their implied surfaces and boundaries (the *Wired Mesh*, *Solid* etc. options in the Pablo model display option). The classes used for storage and display of s-rep models in this project are different from those used at other places in the optimization code (except for the code for rendering of *quasi-tubular objects*, written by Rohit Saboo, which uses the same classes). Instead of the **M3DPrimitive** class, the program for displaying and modifying s-reps within the Pablo display window uses the **XferAtom** class to represent an s-rep atom/hub. The program for calculation and display of surfaces and wire-meshes uses the **Diatom** to represent the s-rep atom/hub.

The **Mesh** class is used to store and display wired meshes. The **CCSubdivSurf** represents implied surfaces by the s-rep model, and the **Pointlist_Server2** generates a point cloud on the surface boundary of the model. These classes are used for generation and display of object boundaries (as well as for interpolation)

6.1 Mesh class

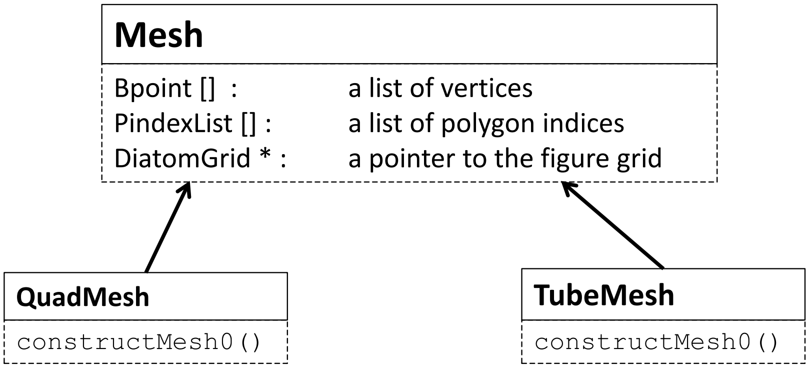

Mesh
Bpoint [] : a list of vertices
PindexList [] : a list of polygon indices
DiatomGrid * : a pointer to the figure grid
QuadMesh
constructMesh0()
TubeMesh
constructMesh0()

Figure 16: Class hierarchy for the **Mesh** class

Figure 16 shows the class hierarchies for the **Mesh** class. Depending upon the type of figure, **QuadMesh** or **TubeMesh** is used, both of which inherit from **Mesh**. The **Mesh** class contains a list of **BPoint** objects. **BPoint** is a class representing a point on the boundary of the s-rep, and contains the following members:

- position[3]
- normal[3]
- radius
- (u,v,t)

6.2 Pointlist_Server2 class

This is a class definition for a PointCloud pointlist server

This class takes an **Xferlist** or **CCSubdivsurf**, copies the **Diatomgrid**, and returns points, normals, and radii of the surface near a **Diatom** given an interpolation of the medial mesh to generate a dense pointset, but not generating a smooth subdivision surface.

Contains following relevant members:

- **Bpoint** *subdivbpoints
- **Bpoint** *subdivtileset
- **CCSubdivsurf*** thisSurf

Interpolation functions which are members of **Pointlist_Server2** class:

- `Pointlist_server2::InterpolateQuad()`
- `Pointlist_server2::InterpolateQuadControlMesh()`
- `Pointlist_server2::InterpolateVertex()`
- `Pointlist_server2::InterpolateVertexControlMesh()`

6.3 CCSubdivsurf class

CCSubdivsurf is a class declaration for a standalone class, creating and OpenGL rendering of subdivision surfaces as wireframes given an initializing Mfig model. (single figure).

- It uses the classes **CCVertex, CCPolygon, CCMesh**
- It contains a list of **CCMesh**'es

Note:

> Spoke interpolation, being just produced by Jared Vicory (formulated originally by Qiong Han) will replace subdivision surfaces for producing the boundary grid points used in the boundary display of models in Pablo.

IV – CODE FLOW EXAMPLES

1. Code flow for Pablo overall optimization process

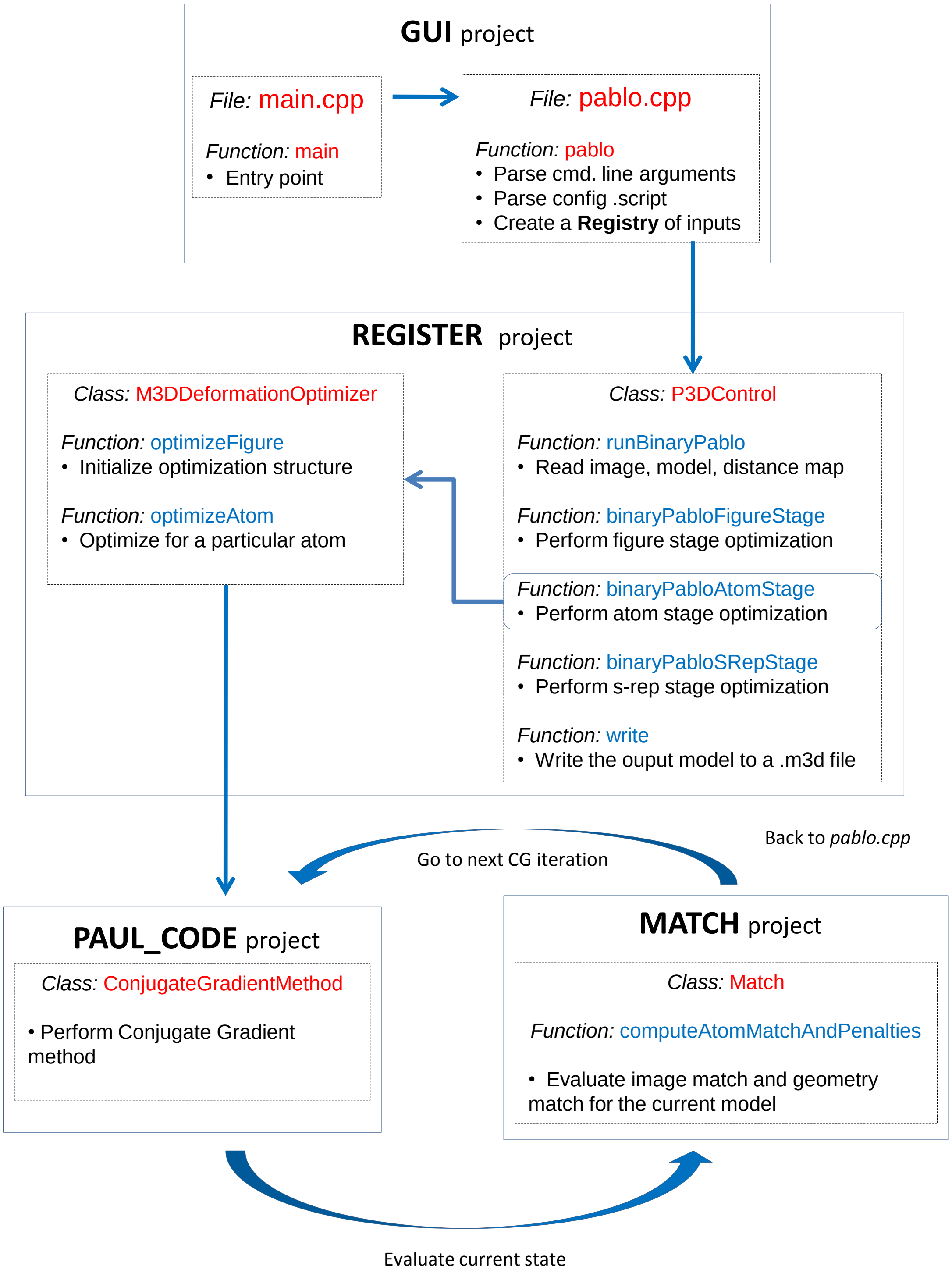

GUI project
File: main.cpp
Function: main
• Entry point
File: pablo.cpp
Function: pablo
• Parse cmd. line arguments
• Parse config .script
• Create a Registry of inputs
REGISTER project
Class: M3DDeformationOptimizer
Function: optimizeFigure
• Initialize optimization structure
Function: optimizeAtom
• Optimize for a particular atom
Class: P3DControl
Function: runBinaryPablo
• Read image, model, distance map
Function: binaryPabloFigureStage
• Perform figure stage optimization
Function: binaryPabloAtomStage
• Perform atom stage optimization
Function: binaryPabloSRepStage
• Perform s-rep stage optimization
Function: write
• Write the ouput model to a .m3d file
Back to pablo.cpp
Go to next CG iteration
PAUL_CODE project
Class: ConjugateGradientMethod
• Perform Conjugate Gradient method
MATCH project
Class: Match
Function: computeAtomMatchAndPenalties
• Evaluate image match and geometry match for the current model
Evaluate current state

Figure 17: Code flow in a typical optimization example in Pablo

2. Code flow for evaluation of the surface points

[Match::computeBinaryImageMatch()]

```
Match::computeSRepMatchAndPenaltiesAsComponents()

   // cleaning up list of surface bpoints
   if (targetSurfacePoints.points) {
      delete [] targetSurfacePoints.points;
      targetSurfacePoints.points = 0;
      targetSurfacePoints.numPoints = 0;
   }

   penalty = computeBinaryImageMatch()

      Xferlist * xferList = convertM3DtoXfer()

      // pList (Poinlist_server2 **): a member of Match (initialized
      somewhere else)

      pList[figureId]->UpdateSubdivPointCloud(level_number, XferList)

         Pointlist_server2::UpdateSubdivSurf()

            // thisMesh (Mesh *): a member of Pointlist_server2

            thisMesh->CopyXferList(thisList)

               // make changes to this function and Diatom class to enable s-
               reps
               DiatomGrid::copyXferList()

            thisMesh->UpdateVertexList()

               // loads level = 0 mesh from the Diatomgrid and constructs
               vertex list upon them.
               // this is a big function. Make changes to this to enable s-
               reps
               thisMesh->constructMesh0() [ QuadMesh::constructMesh0() ]

            // thisSurf (CCSubdivsurf *): a member of Pointlist_server2

            // This function transfers the vertices from the mesh to the
            surface
            thisSurf->InitValue(thisMesh->numverts(), thisMesh->vertlist())

            thisSurf->LoadLevelZeroValue()

               computeIISmesh() -- Iteratively compute a submeshes(0) that
               interpolates the fathermesh in the limit.  numiter gives the
               number of iterations.
               thisSurf->computeIISmesh() // lots of code – hard to
               understand

               computelimitmesh() -- Compute the limit mesh for the given
               submesh level
               thisSurf-> computelimitmesh()// lots of code – hard to
               understand
         end of thisSurf->LoadLevelZeroValue()

         // split the mesh to get a finer mesh
         if(subLevel-1>highestInitializedSubLevel) {
               for(slevel=highestInitializedSubLevel+1; slevel<subLevel;
                   slevel++)
                     thisSurf->splitandaverage(slevel);
               highestInitializedSubLevel=subLevel-1;
         }

         // interpolate to get a finer surface
         for(slevel = 0; slevel < subLevel; slevel++)
                           thisSurf->Interpolate(slevel);


      // end of Pointlist_server2::UpdateSubdivSurf()

      thisSurf->SubdivBoundaryInfo(subLevel, &numsubdivpoints,
      &subdivbpoints);
      thisSurf->GetVertexNeighbors(subLevel, &numvertneighbors,
      &vneighbors);
      thisSurf->GetNeighboringVertices(subLevel, &neighboringVerts);

   // end of pList[figureId]->UpdateSubdivPointCloud(…)

   copySubdivBoundaryInfo(): Returns pointer to internal list of the
   Bpoint-structs giving vertices of subdivsurf derived boundary
   pList[figureId]->copySubdivBoundaryInfo(num_points, surface points)

   // LOTS OF OTHER CODE

// End of computeBinaryImageMatch(…)
```

3. Code flow for initialization of the level-0 boundary points in Match

```
match->definePLists(M3DObject * object, int subdivLevel)

   // allocate memory for pList (Pointlist_server2)
   xferList = convertM3DtoXfer(figure);

   // initialize the pList (Pointlist_server2) from the list of XferAtoms
   pList[i]->init(xferList)

      // newGrid (Diatom *): member of Pointlist_server2
      newGrid->readXferlist( XferList )

         // base function for copying an XferAtom to a Diatom
         // make changes to this function to enable s-rep Diatoms
         Diatom::CopyXferlist( XferList )

   // End of pList[i]->init()

   // ComputeSubdivPointCloud() -- initialize subdivbpoints[] with
   subdivision surface boundary vertices generated from initial diatomgrid
   interpolated to mesh {row,col} spacing and then sent to Subdivsurf for
   subdivision 0 <= subdivlevel < NUMSUBLEVELS.(#defined in Subdivsurf.h)

   pList[i]->ComputeSubdivPointCloud(subdivLevel)

      Pointlist_server2::InitializeSubdivSurf(subLevel)

         thisSurf->init()

         // loadlevelzero() -- load values of fathermesh into submeshes[0]
         thisSurf->loadlevelzero()

            // first values are copied from father mesh
            CCSubdivsurf::LoadLevelZeroValue()

               CCSubdivsurf::computeIISmesh()
               CCSubdivsurf::computelimitmesh()

         // end of CCSubdivsurf::loadLevelZero()

         // to calculate the TOPOLOGY once and for all
         for(slevel = 0; slevel < <reqd. value>; slevel++)
            thisSurf->splitandaverage(slevel);

      // end of Pointlist_server2::InitializeSubdivSurf(subLevel)

      // SOME MORE CODE

   // end of computeSubdivPointCloud()

   // The above process is done for pList[figureCount + i] as well
   // Two copies of pList is saved per figure for some reason
```

4. Code flow for atom stage optimization

HOW PABLO OPTIMIZATION WORKS – AN EXAMPLE

Flow of control in code for the atom stage iterations

`P3DControl::binaryPabloAtomStage()`

Initialize the Deformation optimizer (M3DDeformationOptimizer class)

Optimizers are an interface between the P3DControl (REGISTER project) and the PAUL_CODE projects. An Optimizer defines a **Problem,** which is a class that uses the optimization functions in PAUL_CODE project. Eg. M3DFigureProblem, M3DAtomProblem, M3DDeformationProblem etc.

Every **Problem** class has an evaluate() or evaluateTerms() function.
These functions create a target **M3DObject** (m-rep/s-rep object) based on the current configuration computed by the optimization function and then call the computeAtomMatchAndPenalties() function (MATCH project) to evaluate that object

Problem (REGISTER) classes inherit from **Function** class (PAUL_CODE)

Function is an optimization problem defined on Euclidean N-space

For every iteration of the atom stage: do the following

Perform one iteration of this optimization. The return value of this function decides when the optimizer should stop when executing a particular iteration. At every iteration, only one atom is changed.

`P3DControl::doDeformationIterations()`

Call the Deformation optimizer to perform this iteration: return value decides when this iteration should terminate.

`M3DDeformationOptimizer::performIterations()`

`M3DDeformationOptimizer::optimizeFigure()`

Initialize the internal optimization structure *lastBestX*, a variable in the optimization framework that holds the last best configuration for every primitive in the figure.
Every configuration is a vector of parameters that is passed to the optimization function. We increase the length of this vector to take different radii for the same atom (implement s-reps from m-reps)

Select one atom (simple-order/random) and optimize over this atom

`M3DDeformationOptimizer::optimizeAtom()`

This function returns the cumulative change, *cum_change* in the total penalty value of an atom after the conjugate gradient method

Create a *start* vector, the initial state for optimization (this is the *lastBestX[this atom]*)

Create an *eps* vector, containing the step-size for each parameter in the optimization

Define **M3DDeformationProblem** (this atom, this figure, …)

Define **Numerical Function**: (this M3DDeformationProblem**,** eps vector)

Define **Conjugate Gradient Method**: (this Numerical Function, *start* state)

> **Conjugate Gradient Method** inherits **Numerical Method** inherits **Method**
>
> Call the initialize function
> `ConjugateGradientMethod::initialize()`
>
> **ConjugateGradientMethod** contains **Numerical Solutions** called *bestEver* and *current* and a **Differentiable Function** called *p*
>
> > **NumericalSolution** is a vector, a solution to a numerical problem.
> > **NumericalSolution** inherits **Solution**
> >
> > A **Differentiable Function** is a gradient problem, an optimization problem defined on Euclidean N-space, for which there must be a way to compute the gradient.
> > **Differentiable Function** inherits **Function**
>
> *current* is set by calling `DifferentiableFunction::computeOneJet`(*start*)
> *bestEver* is initalized to *current*
>
> End of define ConjugateGradientMethod

Get the initial value*initVal* by `ConjugateGradientMethod::getBestEverValue()`

> Returns the value of *bestEver*, the **Numerical Function**
>
> `NumericalSolution::getValue()`
>
> **NumericalSolution** inherits **Solution**
>
> > `Solution::getValue()` returns *value*

For the max number of conjugate gradient iterations, do:

`ConjugateGradientMethod::performIteration()`

> ---
> Several options available at this step that lead to the following steps (mentioned after the algo)
> They return a vector of values that represent the current state of the optimization
> ---
> Now evaluate the current state of optimization.
> These *evaluate*()* functions return double(s) that are a some of penalties calculated on the current object

```
                        M3DDeformationProblem::evaluate() OR
                        M3DDeformationProblem::evaluateTerms() OR
                        M3DDeformationProblem::evaluateImageMatch()

                            create a target primitive from the current state of the problem
                            M3DDeformationProblem::createTargetPrimitive()

                                This is where the vector that represents the current optimization state is converted into an actual M-rep object. This is where the code needs to be changed to incorporate s-reps
                                M3DDeformationProblem::applyVector()

                            call the penalty functions in Match to evaluate the current state

                            Match::computeAtomMatchAndPenalties()
                            OR,
                            Match::computeAtomMatchAndPenaltiesAsComponents()
                            OR,
                            Match::computeAtom…Match()

                    End of ConjugateGradientMethod::performIteration()

                    If current bestEver value, val is better (lower) than the initial bestEver value, initVal:
                        result = ConjugateGradientMethod::bestEverX
                        cum_change = fabs(initVal - val)
                    Else
                        cum_change = 0

                    Update the candidateObject (an atom because we are optimizing for one atom at this stage) with the lastBestX[ this atom] by creating a primitive
                    M3DDeformationProblem::createTargetPrimitive()
                        M3DDeformationProblem::applyVector(this atom, this vector)

                End of M3DDeformationOptimizer::optimizeAtom()

                Update the lastResult for this primitive with the cumulative change from optimizeAtom()
                lastResult[primId] = cum_change

            End of M3DDeformationOptimizer::optimizeFigure()

        End of M3DDeformationOptimizer::performIterations()

    End of M3DDeformationOptimizer::doDeformationIterations()

    Get the match results of this Deformation iteration and print them out
    Match::getAtomStageResults()

    Calculate average distance to the boundary etc
    MatchUtility::calculateAverageDistance

 Report results again after the last stage and calculate average distances etc
End of P3DControl::binaryPabloAtomStage()
```

Appendix – I: OLD PABLO USER MANUAL (2001)

Pablo User Guide

By Gregg Tracton
Pablo Version: 2001-08-01
Last Update to document: 8/3/01 1:32 PM

To view this document requires a HTML Browser capable of	HTML level 2, simple tables, color, images, no frames
To Print document, use	Adobe Acrobat version 4 or above

Pablo changes everyday. This document is a snapshot of the current version.

Pablo Programmers:

Tom Fletcher	fletcher@cs.unc.edu
Yonatan "Yoni" Fridman	fridman@cs.unc.edu
Graham Gash	gash@cs.unc.edu
Sarang Joshi	joshi@radonc.unc.edu
Martin Styner	styner@cs.unc.edu
Andrew Thall	thall@cs.unc.edu
Gregg Tracton	tracton@radonc.unc.edu
Paul Yushkevitz	pauly@cs.unc.edu
unnamed others...	gurus@anonymous.net ☺

1 Table of Contents

You can click on a section to speed you to that section.

2 Scope

2.1 Audience: Shape Researchers:

- do not need to know how to program,
- must know how to edit structured text files,
- should know 3D graphics and geometry terms (such as camera, view, surface, cut plane, isotropic, coordinate systems), and
- should be familiar with medical anatomy.

2.2 Related Documents:

- "Deformable M-Reps for 3D Medical Image Segmentation", Pizer et al, submitted to MedIA August 2000. (on defmreps web page)
- "***Pablo*** Tutorial on Crafting Shape Models", Gregg Tracton.
- "***Pablo*** Installation Guide", Tom Fletcher.
- **defmreps** web page contains links to all things m-reps: papers, segmentations results movies, Radiotherapy uses, statistical shape characterization for Neuroscience, etc. See http://www.cs.unc.edu/Research/Image/MIDAG/defmreps

3 Purpose: What is Pablo, Who Uses It

Pablo is research software used to position and shape m-rep models, possibly in the context of a 3D 12-bit grayscale image and/or 3D triangular tiles to approximate the segmentation being modeled. It is also a good 3D viewer of these structures; displays include a 3D perspective window, multiple 2D cut plane windows (anchored to the image's plane or anchored to one or more atoms), and a 2D ribbon view through multiple atoms.

At any one time, pablo can know about:

- a single model of multiple figures which represents one or more objects such as anatomy, surgical clips, tumors, etc
- a single target image and a single training image, both 3D
- a single tile set.

It can perform operations on arbitrary sets of atoms. It can interpolate between the atoms in a figure to compute both smooth medial sheets (the center of the object used by model builders) and the middle of the implied boundary (the surface of the object, estimated at some scale). It can fit a model to the underlying image in many ways: manual (under user control), whole-model (using a similarity transform over all figures), per object (geometric constraints), per figure (similarity transform), atom deformation (per atom), [LATER: and boundary displacement (per boundary position).] It can write an image representing the modeled object, for instance, with 1 pixel values inside the object and 0 pixels values elsewhere, [TECH: or intensities near the boundary masks which indicate values from the training image].

We anticipate two distinct classes of users: **model builders** and **model fitters**. Fitters adjust pre-built models to a particular image. Our plan is that they should only need to know about figures and surfaces. Builders need to know what fitters know, plus have a working knowledge of the medial sheet of all the models in a anatomical site (in the current user interface), how figures fit together to form objects, and the intimate details of atoms.

Pablo runs on Microsoft Windows, expects a 3-button mouse and uses advanced openGL 1.1 features of 3D graphics boards. A 2-button mouse can emulate a 3-button mouse using the Shift key to indicate the missing middle mouse button, useful for laptops having only 2-button capability. [TECH: A source-code compatible Solaris/SGI version is in the works and we always write our programs with the expectation of multiple OS environments.]

4 Nomenclature: Fonts, Colors, Styles in this Guide

Styles, fonts and colors may have specific meanings.

Underline	An underlined word indicates emphasis.
Bold	A word in the bold text attribute delimits: • words you'll see, literally, in the user interface or document • words that introduce a topic
italic	this text attribute defines proper words with meanings specific to this design. Most browsers draw emphasized text in an italic font.
Colors	See context. Usually indicates a color in the user interface or document. Blue text means the section is out of date.
`primitive[0][0]`	this font indicates something you type.
[TBD]	means "To Be Defined"
[LATER: text**]**	describes features that are planned
[TECH: text**]**	technical text intended for programmers, advanced users or model builders.
[WINDOWS: text**]**	describes features that are implemented for Microsoft Windows platforms only.
[0..2]	the closed range of numbers from 0 through 2, including both 0 and 2. This range can be either real numbers or integers, distinguishable from context.
20000614	versions are defined by dates in YYYYMMDD format. This sample means June 14, 2000, the first time m-reps worked on a real clinical image.
File -> Load Model...	means left-click on the **File** menu to expose the sub-menu, then select the **Load Model...** menu option. May also refer to a notebook tabbed window and it's tabs. **...** means that a dialog box will pop-up to collect more details before performing the action.
Cntl+z	is an "accelerator" - a keyboard equivalent for an action. For this example, hold down the **control** key and press the **z** key, then release both.

5 Concepts: Terms You Need to Know

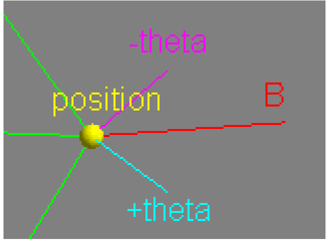
A user needs to understand these concepts for effective use.

5.1 M-rep model

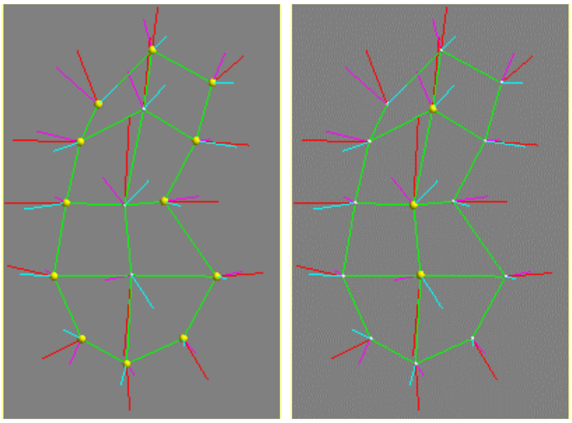
A model, as manipulated by this program, is a set of ***objects*** containing ***figures***, some per-figure attributes (such as polarity, display color or name) and relationship information between objects and/or figures that determine their properties during optimization. A figure represents a significant portion of an object, such as a finger of a hand or, to illustrate negative figures, a hole in a bone. A model is contained entirely in a single text file called a ***m3d*** file.

5.2 Atom, Primitive

An atom (previously called a “primitive”) determines the shape of two related patches of the object’s surface and the contained 3-D space in-between. Some atoms’ patches wrap around the edge of the object and so the surface portions are joined to form a single sheet. Each patch is usually small and is measured at a specific scale. [TECH: an atom has a **position** somewhere near the object’s medial axis (a curvy plane in the object’s center), an expandable **B vector** (Bisector) indicating the main direction of the narrowing of the figure, two Y vectors at **+theta** (called Y0) and **-theta** (Y1) degrees from the B vector that imply the object’s boundary, an N vector (not shown) perpendicular to the B vector and normal to the medial plane, a B-Perp vector (not shown) orthogonal to the B and N vectors, a width indicating the distance at which an implied boundary is most probable:



A figure's contains a NxM mesh of atoms, where N = 2^n+1 and M = 2^m+1 is suggested for ease of subdivision. Typical values are 3, 5, 9 (7 is not suggested). In this 3x5 mesh



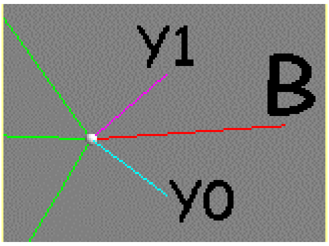
End Atoms in yellow

Standard Atoms in yellow

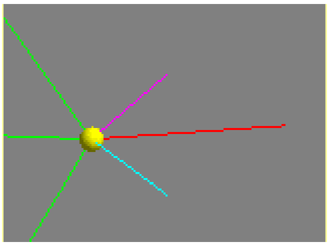
we see that each ***end*** atom has 3 neighboring atoms, indicated by the **green lines**, and that each ***standard*** type has 4 neighbors.]

5.3 Selecting & Marking Atoms

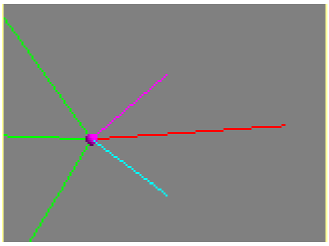
Atoms change color or size when **selected** or **marked** for manipulation. An unselected atom's center is a small white ball. The ball is connected to neighboring mesh atoms by green lines (color is user-chosen). When selected, large balls are drawn. When marked, magenta balls are drawn. One (only one) atom may be **marked** at at time.



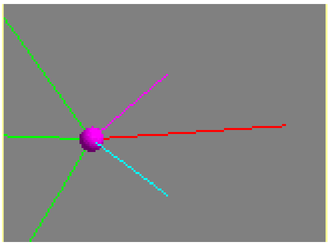
Normal



Selected



Marked



Marked & Selected

You can’t select/mark atoms that you can’t see (behind opaque tiles). See Mouse Buttons.

5.4 Image

A 3D grid of voxels each containing a 16-bit signed scalar and occupying a parallelpiped ("3D rectilinear field") of space. [TECH: Currently, the grid must be regularly spaced in each of it's X, Y and Z dimensions, but these spacings need not be equal (X is commonly the same as Y, however). The Z dimension is across the natural "slice" orientation of the image acquisition, but it is not defined as to how this correlates with true space in the real world (for instance, it could be rotated 90 degrees). The file formats are ***raw3*** and ***UNC metaheader***, but irregular Z spacings may be required at some point and so these will be expanded.]

5.4.1 Unit Grayscale/Color Space

Grayscale images may have up to 65,000 distinct grayscale pixel values. [TECH: For grayscale images, these are mapped to the float range [0..1] representing the values seen in that image. The image contains a header telling the possible range.]
Color images are used for display only and each pixel contains three real numbers [0..1] representing Red, Green and Blue color components which are additively mixed.

5.5 Tiles

An arbitrary set of 3D triangles representing a 3D surface. [TECH: The normals all face toward the outside of the "object" being defined. Note that holes and multiple surfaces are allowed and the direction the normal faces is defined in any way the tile creator desires in these cases. We have not defined what "outside" means in a negative figure yet.]

5.6 Unit Coordinate System

Models, images and tiles are all scaled to the same coordinate system which is defined as a isotropic unit cube in 3D with each dimension's domain [0..1]. The cube does not have to be completely filled. [TECH: if the image's space is not cubic then 1 or 2 of the dimensions will occupy [0..N] where N is somewhere in (0..1] and is the ratio of the smaller dimension to the larger dimension.]

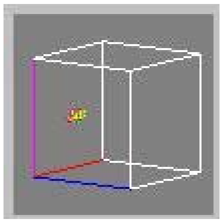
5.7 Object Hierarchy

This gives you a clue of what sort of information should be in each file and the attributes the user interface may be able to edit and/or display. An example of how to read this is "a model contains a set of figures and a set of figureTrees; each figure contains a figure ID, a color, ...":

- Model
    - Figures (set)
        - Figure ID: [0..N]
        - Figure's Name (user assigned)
        - display color: Red, Green, Blue
        - positivePolarity: 1 for positive, 0 for negative
        - positiveSpace: 1 for protrusion figure, 0 for indentation figure
        - type: QuadFigure
        - numRows
        - numColumns
        - atoms (Mesh)
            - r: length of Y vectors
            - elongation: [1..infinity]
            - orientation: qx, qy, qz, qx
            - selection flag
            - mark flag
            - theta angle: degrees
            - type: EndPrimitive/StandardPrimitive
            - position: x, y, z
    - FigureTrees (set)
        - parent figure ID
        - childLinks (set)
            - blendAmount: [0..1]
            - blendExtent: [0..1]
            - child figure ID
            - childLinks (set)
                - child atom ID
                - (u,v,t) coordinate of parent
- Image
    - textual description
    - voxel dimensions, eg, 256x256x62
    - voxel size, eg, .1x.1x.4 cm
    - voxel orientation: can invert direction of axes
    - voxel values
- TileSet
    - Tiles (set): x1,y1,z1, x2,y2,z2, x3,y3,z3

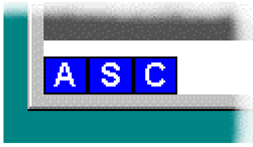
6 Conventions: What Pablo Assumes

- Reads these file formats:
  - images: ***raw3*** or ***mha*** (Aylward Metaheader) images, which the user generates,
  - tiles: ***UNC tile*** format, which the user generates,
  - models: ***m3d*** m-rep format, or unsupported ***mod*** format
  - Tiles and models are encoded in ***PaulY*** format (see Editing Model Files), which is easily read and changed by users with any text editor program.
- Writes these file formats:
  - ***raw3*** images
  - ***m3d*** m-rep format.
- To make tiles, use one of your own tools or a combination of the MASK contouring tool or any other contouring tool and the CTI contour-to-tile tool. (Both from http://www.radonc.unc.edu/tools). We find it more convenient to create tiles for each figure and then simply concatenate them together to form a single file containing tiles for the whole object (or any subset of figures), when needed, but smooth corners might not be represented accurately that way.
- Load the image before the tiles or model, because the image contains values from which is derived a scale which defines the size of a pixel (in centimeters). This allows one to use a model across multiple images with different pixel sizes and was chosen instead of cm as the coordinate system because a model ***has no actual physical size***, that is, coordinates are flexible enough to be interpreted as cm, inches, leagues, or even light years. The tiles are not in the correct position until a corresponding image has been loaded.
- Axes are color coded. X,Y,Z are **Red**, **Magenta**, **Blue**. The X & Y axes are considered "in plane" w.r.t. the image; the Z axis is "cross plane," by convention. Orientations are actually arbitrary in practice.



- **trackball** - the views are manipulated in 3D using a "virtual trackball." To rotate objects or cameras, imagine that on the model window is centered a 3D sphere that always touches the 4 corners of the window. The mouse click is projected up onto the sphere's surface and selects a position on the ball. Mouse movements are also projected up and the ball is rotated in 3D around it's center to maintain the selected model position under the mouse. The view and/or atoms follows the sphere. For example, to tumble the scene towards the camera, press the left mouse button at the top center of the window and drag the mouse to the bottom center. When the scene is oriented properly, release the mouse button.

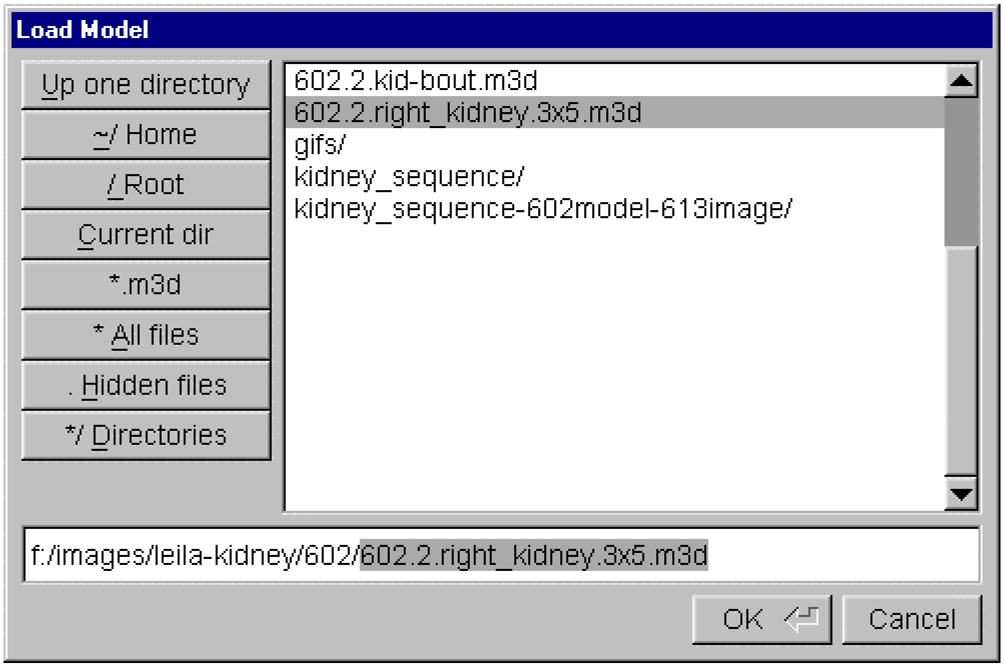
  Lost? Press **Display Control -> General -> Reset View** to set the camera to it's initial pose, position and scale. Or press a button (see picture at right) to reset the camera to Anterior (view from +Z), Superior (from +Y) or Coronal (from +X). **Undo** does not affect the camera.



- **Alpha** sliders control opacity of tiles, [LATER: images and model surfaces]. Move the slider to **1.0** for opaque, **.5** for transparent, **0.0** for invisible.
- Only the model view window may be resized. Cut Plane view windows are statically calculated and so are not resizable.
- Checkboxes are enabled when they are red, disabled when gray. Click the box to enable or disable

7 File Browser (FLTK Style)

When Pablo needs you to specify the names and paths (folders) that files should be read from and written to, a file browser will appear.



602.2.kid-bout.m3d
602.2.right_kidney.3x5.m3d
gifs/
kidney_sequence/
kidney_sequence-602model-613image/

f:/images/leila-kidney/602/602.2.right_kidney.3x5.m3d

It may be different from file browsers you have seen before:

- Read the prompt in the title. Sometimes a "save model" file browser will appear when you are expecting a "load model" browser, because Pablo can only hold a single model at a time and wants to give you a chance to save changes to the last model before replacing it with the new.
- [Windows] To look at the files/folders on a different disk, replace all the text on the bottom text entry with the disk letter, eg, "f:/" to look at the F disk.
- Clearing out the text will show your current working directory.
- Files have types (eg, raw3 for images, m3d for models) as indicated by last few letters of the file's name after the last dot. Only the files of the type that the application is expecting will be shown in the browser’s list initially, and only one type is specified by the program. You can force it to show all the files (including the hidden ones!) using the buttons on the left side of the browser window.
- The **Tab** key expands a partial selection. For example, if the name of a directory is "leila-kidney" you may type just `le` and then **Tab** and the rest of the directory's name will appear. It must be uniquely specified, that is, if there's another directory called "leonard-zeppelin" then "le" is not not enough letters for the browser to tell which you mean. Type more letters, then another **Tab**.
- Use single clicks to select directories. If you should use a double-click then you will select both the directory and some file in that dir, which can crash the program if the errant file is of the wrong format.
- Resize this window if the filenames are too long to be displayed fully.
- **Up** & **Down** arrow keys select the previous and next file/directory.

8 Mouse Buttons

This describes mouse actions in the 3D model view window.

- A 3-button mouse allows use of the Control and Alt "modifier" (shift) keys: hold down the modifier key and click the mouse button(s), then release the modifier key. Build into Pablo is to use a 2-button mouse by substituting Shift-left for the middle mouse button; as indicated at the bottom of the table below
- Scale-width (Alt&Shift-right) is only available by Shift'ing. We ran out of buttons…
- The *view* actions affect the camera only. The atoms are not moved relative to the axes, images, tiles or each other.
- *Select* and *mark* actions toggle atom states: if an atom had been selected before the action, the action will deselect the atom. When using the marquee, each atom is toggled independently of the others
- The marquee acts on multiple atoms at once. Drag out a rectangular region. You'll see a yellow line indicating the region. All the atoms within the rectangle will be affected, up to the limits of the graphics hardware (usually 255 atoms). If you start the drag on a atom then you'll just affect that one atom, which is probably not your intention.
- COG = Center of Gravity of all selected atoms.

shift]			
	figure, or select multiple atoms with marquee		
	rotate selected atoms with trackball	translate selected atoms in view plane (orthogonal to line of sight)	scale selected atoms around COG: drag mouse down screen to enlarge.
Shift	translate view	-	-
Shift & Alt	translate selected atoms (see Alt-middle)	-	Scale selected atoms ***and*** change their width

9 Views

9.1 Model View Window

This view shows a 3D perspective visualization of the tiles, image slices/axes and models. The user can interactively manipulate the pose/position/zoom of the camera and atoms directly in this window by translating, rotating, scaling, selecting and marking. See Mouse Buttons.

Tiles are drawn as two-sided: the inside face of each tile is a different color than the outside face for those situations in which the tiled surface folds back on itself.

If you resize the window to a non-square the scene will always remain square causing unused space at the side or bottom edge of the window.

9.2 Slice View Windows

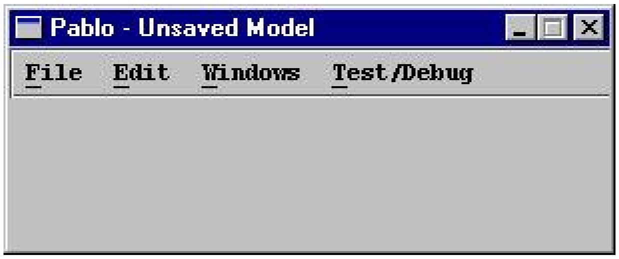
The **Main -> Windows -> Atom Cut Planes** views help to visualize the image boundaries close to a particular atom. It houses checkboxes that expose any of 6 windows, each with a 2D plane through or near the marked atom that can display an arbitrary image slice aligned with the marked atom, lines representing the atoms (with axes) in the marked atom's slice, a curve indicating the implied boundary, and the intersection of a slab aligned with the atom's geometry with the tiles. There are various orientations of this view with respect to the marked atom:

1. **Crest**: b/n plane
2. **Atom**: b/b-Perp plane
3. **BPerp-N**: n/b-Perp plane
4. **Starboard sail**: +theta/n
5. **Port sail**: -theta/n
6. **Involutes**: chord/n

[LATER: needs picture]

The **chord** is the line segment between the tips of the +theta and -theta vectors, and observant readers will note that it does not intersect the n vector in the last view. Instead, the n vector is translated so that it intersects the chord and that plane is used. The views are hard-coded.

10 Main Window

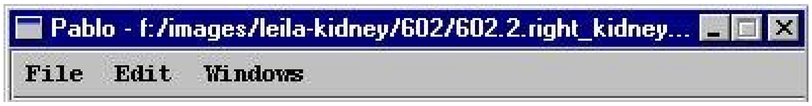


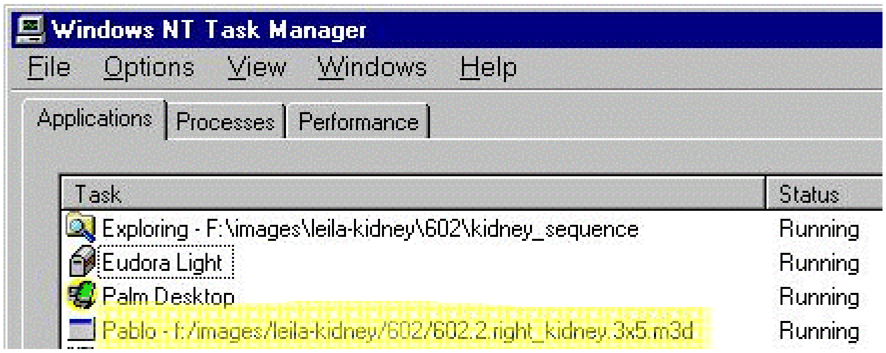
Contains top-level menus for:

- reading/writing files
- undo/redo atom modifications
- atom selection
- figure copy-and-paste
- test/debug controls
- mirror atoms around an X plane
- figure creation/destruction
- showing/hiding control/view windows: preferences, display, optimization, atom cut plane, visibility, object-object constraint

The name of the current model's file (see picture below), or **Unsaved Model** (see picture above), is displayed in the title bar of the window. If you are concerned about patient confidentiality (not disclosing your patient's name to outside parties) then you should be careful not to name the model or any folder in the path to the model file with the patient's name, as this may show up in window dumps useful for publishing results.

[WINDOWS NT: If you need to read the filename and it is clipped to the size of the bar, a trick is to start the Windows task manager and expand the width of the **Task** column until the filename can be read at the end of the task:

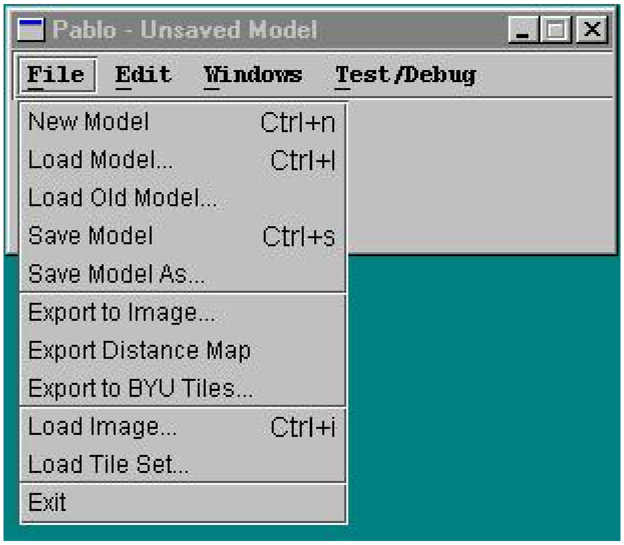




Task	Status
Exploring - F:\images\leila-kidney\602\kidney_sequence	Running
Eudora Light	Running
Palm Desktop	Running
Pablo - f:/images/leila-kidney/602/602.2.right_kidney.3x5.m3d	Running

. You can also read this info in **Windows -> Display Controls -> General**.]

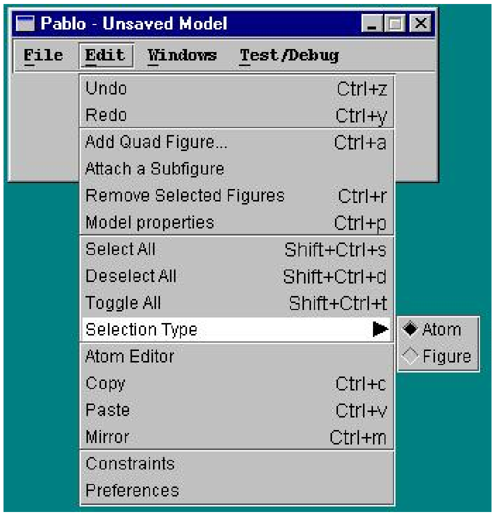
10.1 Main Window -> File



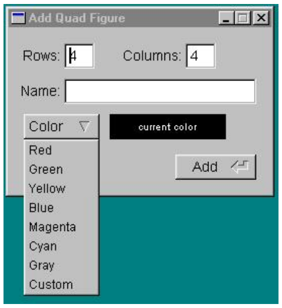
- **File -> New Model...** deletes the current model.
- **File -> Load Model...** reads a m-rep model (.m3d) from disk, replacing the current model. *Pay careful attention in case it a*sks if you want to save changes to the current model first
- **File -> Load Old Model...** is for backwards compatibility and probably will not work anymore. It's like **Load Model** except that it reads the old format (.mod). The atom types (EndPrimitive or StandardPrimitive) are calculated and the elongations are all set to 1.0.
- **File -> Save Model...** creates or overwrites (does <u>not</u> ask first) a file with the current model, which includes all of the figures currently defined except those in the cut buffer. Note that a model file records which atoms are selected and which atom is marked.
- **File -> Save Model As...** is like **Save Model** except that it displays a file browser to save to a different file than the file from which the model was read.
- **File -> Export to Image...** writes a binary "scan-converted" image of the current model to an image file (.raw3) containing a pixel value of 1 in voxels on or internal to the model's implied boundary and 0 elsewhere. This is used to gather statistics of the fit of the model to the grayscale image. The image has the same dimensions and pixel size as the loaded image, or some default if no image is loaded.
- **File -> Export Distance Map** is for testing purposes. Ignore it.
- **File -> Export to BYU Tiles...** writes a tileset file of the implied boundary for transferring the boundary to the boundary displacement program.
- **File -> Load Image...** replaces the currently loaded image with one from disk. Images can have the filename extensions **.raw3** or **.mha**/**.hma** (for UNC MetaHeader format). Press the file browser's **All Files** button to see the non-raw3 filenames. Since images are read-only (the program cannot change them), the only info that can be lost by replacing the image is the global scale from the unit cube to cm. The display is reset as well: each of the model view's 3 displayed slice locations are reset to display the center slice in each dimension. [This should not happen, I think -gst.]
- **File -> Load Tile Set...** replaces the currently loaded tile set with one from disk. The tiles are scaled to the same unit box as the image (if any) or to the unit bounding box of the tiles.
- **File -> Exit** - terminates the program. It will ask if you want to save changes to the current model first, if changes were made.

10.2 Main Window -> Edit

- **Edit -> Undo** reverses the last edit made to atoms, as if the last edit never occurred. Almost all edits can be undone, even **Add Quad Figure...**, **Remove Selected Figures...**, **Atom Editor** and **Load Model**. Changing the camera view is not considered an edit and so cannot be reversed - this would have been useful for moving the camera through a scene reproducibly. Multiple edits, up to 1000, can be undone in succession [TECH: and are stored in what is called the ***undo buffer***].
- **Edit -> Redo** reverses the effect of an **undo.** [TECH: the Undo buffer is unchanged by Redo, that is, a Redo does not count as a change to the atoms.]

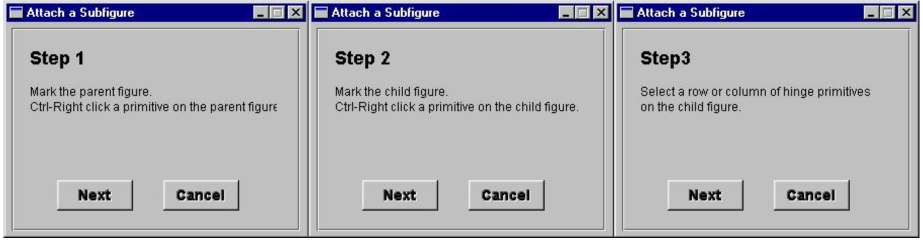


- **Edit -> Add Quad Figure...** adds figures to the model and assigns colors and arbitrary text names to figures. **Rows** and **Columns** define the **N** and **M** mesh dimensions seen in **Concepts**, page 5. Click **Add** to create a figure – the dialog box stays open so that multiple figures can be added efficiently. The last figure created is left selected so create a figure, move it a little, create the next figure, repeat.


Rows: 4 Columns: 4

Name:

- **Edit -> Attach a Subfigure** displays a sequence of windows to merge 2 independent figures into a parent and child relationship:



Step 1

Mark the parent figure.
Ctrl-Right click a primitive on the parent figure

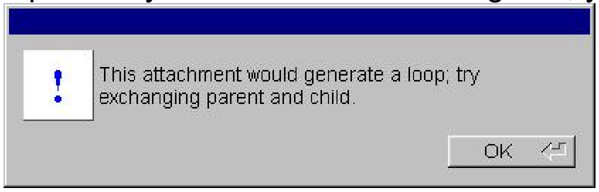
Step 2

Mark the child figure.
Ctrl-Right click a primitive on the child figure.

Step3

Select a row or column of hinge primitives on the child figure.

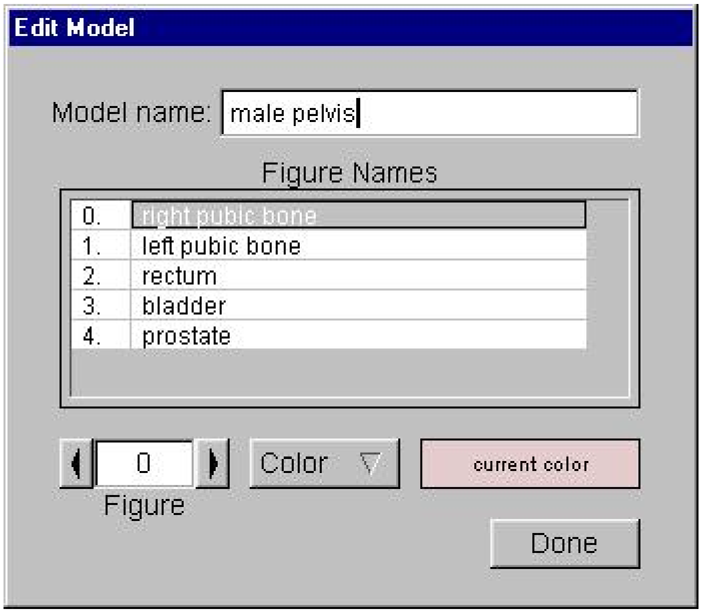
If a child-parent relationship already exists between the 2 figures, you may get this:


This attachment would generate a loop; try exchanging parent and child.

[**Important bug**: the child's hinge atoms may attach to either the parent's EndPrimitive atoms or StandardPrimitive atoms, but not both. In other words, if you think of the parent as a slab, the child may attach to one and ***only*** one of: the slab top, the slab bottom or the slab side. This is a design flaw that we plan to fix. Pablo may crash if you disregard this rule, so save your model just before you attempt to attach each subfigure.]

- **Edit -> Remove Selected Figures...** deletes figures. Select all the atoms in a figure, then click this and poof! the figures are gone. See **Edit -> Undo**.

- **Edit -> Model Properties** displays a dialog box to edit the names of the model and figures and choose a color for each figure. Figures may also be named and colored as they are created with **Add Quad Figure**. This dialog box blocks the interface (no other windows will respond) until **Done** is pressed.



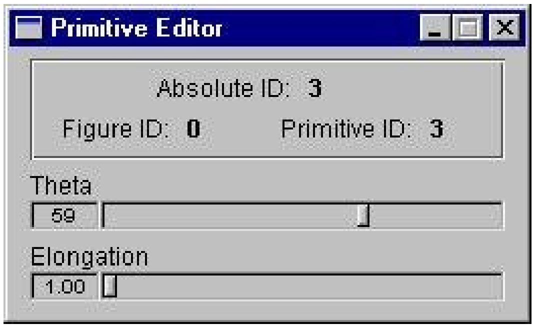
Model name: male pelvis

0.	right pubic bone
1.	left pubic bone
2.	rectum
3.	bladder
4.	prostate

Figure: 0

- **Edit -> Select All** selects all atoms in all figures regardless of their current selection state.
- **Edit -> Deselect All** like Select, but deselects instead.
- **Edit -> Toggle All** selects all unselected atoms and deselects all selected atoms. This is useful to perform some operation on a set of atoms and then another operation on the remaining atoms.
- **Edit -> Selection Type** controls what can be selected by the control-left mouse button in the model view window. To select an entire figure with each click, set the selection type to **Figure** and select one or more atoms with the mouse and you will see that the entire figure is selected (or deselected). You can also select/deselect multiple figures easily and quickly using the mouse's marquee feature to affect all figures containing atoms within a 2-D rectangle from the current camera angle. To select/deselect by atom, set the selection type to **Atom**.

- **Edit -> Atom Editor** displays the atom editor:



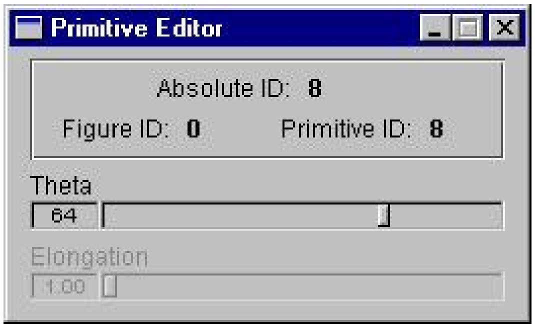
Primitive Editor

Absolute ID: 3

Figure ID: 0 Primitive ID: 3

Theta 59

Elongation 1.00



Primitive Editor

Absolute ID: 8

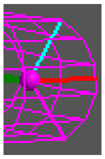
Figure ID: 0 Primitive ID: 8

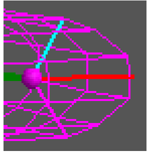
Theta 64

Elongation 1.00

The Atom Editor is the only place to manipulate **Theta** and **Elongation**, as there is no mouse-oriented manipulator in the model view window. You use these sliders by marking an atom and watching the 3D window, for feedback, as you move the sliders. This window updates as you change which atom is marked. Only EndPrimitive atoms have an elongation (left image, above), so StandardPrimitive atoms have a dimmed elongation slider (right image, above).
The Atom Editor also reports the **Figure ID** and **Atom ID** of the marked atom and it's figure, which are useful when editing model files by hand, see **Editing Model Files**, page 26. **Absolute ID** is a single number that identifies an atom across all figures, eg, if figure 0 contains atoms 0-8, figure 1 would start it's absolute atom IDs from 9. Users may need to edit model files by hand because not all of the functionality required to assign attributes (eg, polarity) are implemented in the user interface.

A theta of 90 degrees spreads the involute arms as far as they will go. 0 means the arms overlap. Elongation is 1 (left image, below) for a perfectly round EndPrimitive atom and increases as the edge of the figure gets more pointed (right image):

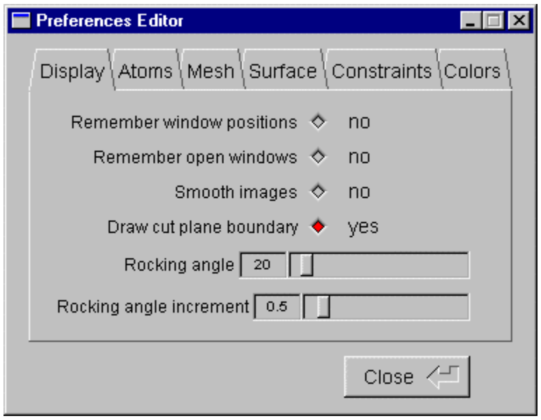




- **Edit -> Copy** places a exact duplicate of the selected figures into the cut buffer (a store of hidden figures). See **Paste**, below.
- **Edit -> Paste** duplicates the figures in the cut buffer as selected figures. As opposed to most other programs, Pablo Paste does ***not*** replace currently selected figures but instead de-selects them and selects the newly pasted figure(s). Copy and Paste can be used to accumulate figures across m-rep models (in place of editing the files containing those models) or to split a model's figures into separate model files. [Possible bug: I have not tested if subfigure attachments and object constraints are correctly deleted when removing a figure containing these. -gst]
- **Edit -> Mirror** reflects the selected atoms around the center X coordinate (X = .5), but does not affect the Y & Z coordinates. In other words, the atoms are moved to a new position based on each one's current X position only. This is useful because the bilateral symmetry of vertebrates can be used to model a structure on one side of the body and mirror it as a starting place to model the partner structure on the other side of the body. To reflect around a different axis, we suggest: rotate the figures so that the plane of reflection is at X = .5, mirror the atoms, then rotate back to the original orientation.

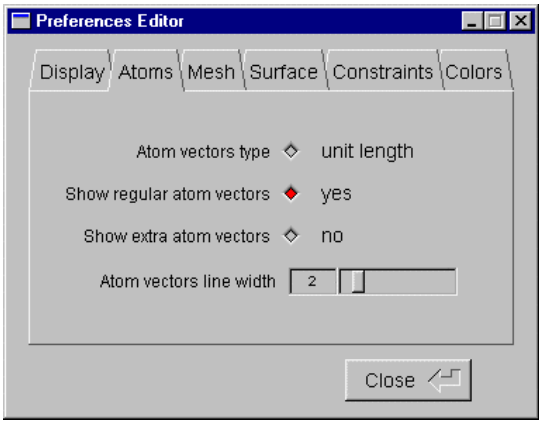
- **Edit -> Constraints** is to set or inspect object to object geometric constraints. [LATER: expand this when new interface is ready!!!]

○ **Edit -> Preferences** edit window positions, display attributes, etc. This information is stored in a file in your home directory. The filename is printed when Pablo starts up.



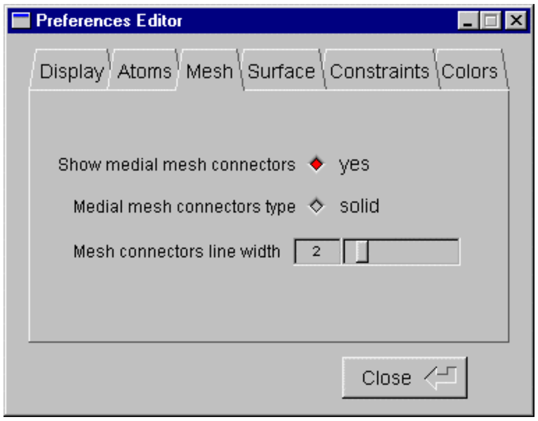
Remember window positions ◇ no
Remember open windows ◇ no
Smooth images ◇ no
Draw cut plane boundary ◆ yes
Rocking angle 20
Rocking angle increment 0.5

Remember window positions records the current size and position of each Pablo window for the next time you start Pablo.
Remember open windows notes which windows are displayed and which are hidden for the next time you start Pablo.
Smooth images changes the model view window's grayscale image display from nearest neighbor interpolation to tri-linear interpolation, making the images look smoother.
Draw cut plane boundary enables that display of a white line at the edges of the X, Y, and Z grayscale cut planes in the model view window
Rocking Angle is the number of degrees that the camera will rock (rotate from side to side) whenever the mouse stops moving.
Rocking angle increment is the rotational degrees to move the camera for each frame of the rocking sequence. Larger numbers appear to rotate the camera faster for the same number of frames.



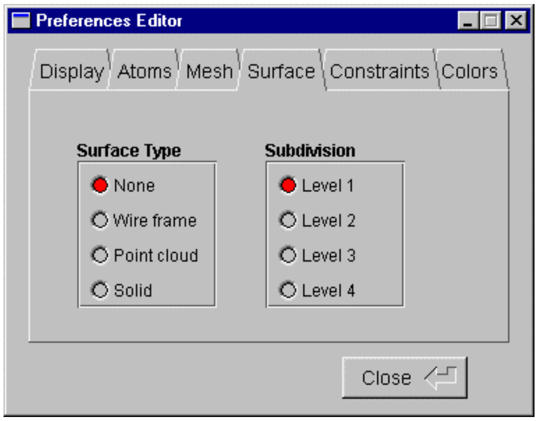
Atom vectors type ◇ unit length
Show regular atom vectors ◆ yes
Show extra atom vectors ◇ no
Atom vectors line width 2

Change the model view window display:
Atom vectors type selects the display of the red **B** vectors from involute length to (elongation * involute length).
Show regular atom vectors draws the B and involute vectors
Show extra atom vectors draws the atom's frame (X, Y, Z vectors)
Atom vectors line width adjusts the atom's vectors thickness. It is useful for illustrations and slide shows.



Show medial mesh connectors ◆ yes
Medial mesh connectors type ◇ solid
Mesh connectors line width 2

Show medial mesh connectors draws the green lines between atoms
Medial mesh connectors type selects solid, dashed or dotted connectors. It is useful for grayscale illustrations in which color cannot be used to distinguish lines.
Mesh connectors line width adjusts the thickness of connectors.

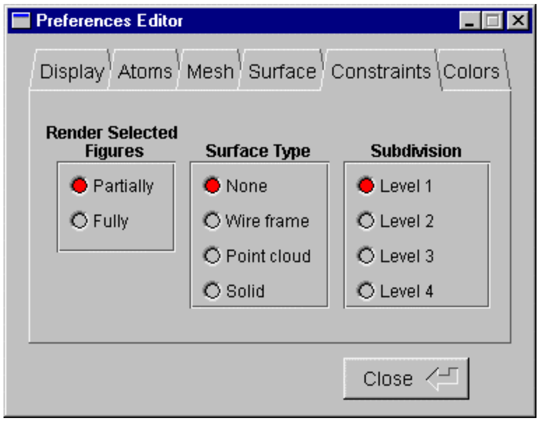


Surface Type
● None
○ Wire frame
○ Point cloud
○ Solid

Subdivision
● Level 1
○ Level 2
○ Level 3
○ Level 4

Surface Type selects the visualization type of the implied boundary

Subdivision implies the number of times the space between neighboring atoms will be divided to form either the meshed or tiled surface. Higher levels mean finer surfaces that also take more time to compute.



Render Selected Figures
● Partially
○ Fully

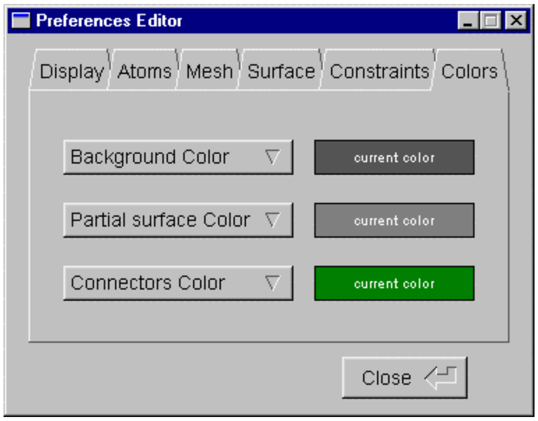
Surface Type
● None
○ Wire frame
○ Point cloud
○ Solid

Subdivision
● Level 1
○ Level 2
○ Level 3
○ Level 4

Render Selected Figures affects the constrained figure's surface visualization during constraint editing (see **Edit -> Constraints**). **Partial** draws the patches of the governed surface which are closer to the governor than the distance slider and does not draw more distant patches. **Fully** uses color changes: the closer portion is drawn in the **Partial Surface Color** (see **Edit->Preferences**) and the farther portion is drawn in the figure's assigned color.

Surface Type is the visualization used for the closer portion.

Subdivision is used for the closer portion.

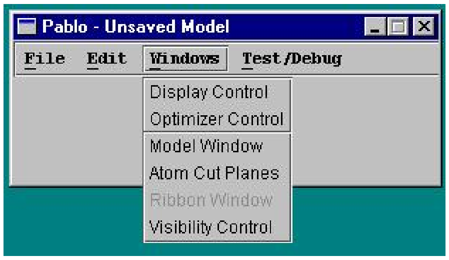


Background Color is the background of the model view window. For illustrations, some model colors work better with light background colors, other with dark background colors.

Partial surface Color See the **Constraints** tab, just above.

Connectors Colors is the color is the lines drawn between atoms in the model view window.

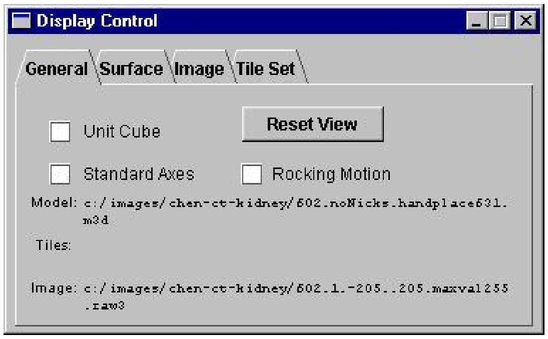
10.3Main Window -> Windows



The **Windows** menu shows or hides other windows in the application, which are described in this section.

10.3.1 Windows -> Display Control

A tabbed notebook window for controlling the appearance of objects in the 3D model window:



☐ Unit Cube

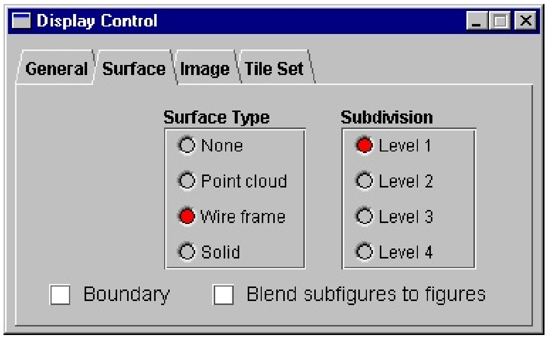
☐ Standard Axes ☐ Rocking Motion

Model: c:/images/chen-ct-kidney/602.noNicks.handplace631.m3d

Tiles:

Image: c:/images/chen-ct-kidney/602.1.-205..205.maxval255.raw3

Unit Cube shows white edges of the space.
Standard Axes shows the primary X, Y, Z colored axes.
Rocking Motion continually rotates the camera. Good for capturing smooth model animations.
Model, **Tiles** and **Image** show the most recently loaded files. **Undo** does not affect these.
Reset View sets the pose/position/scale of the camera to their initial values.

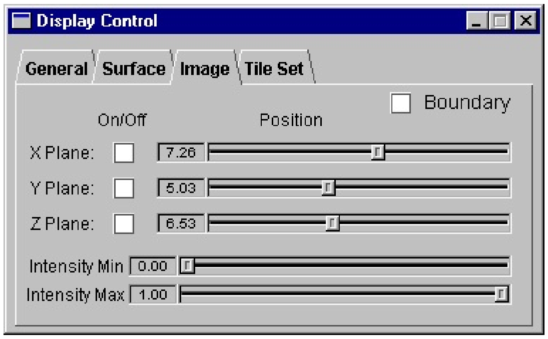


Surface Type: ○ None ○ Point cloud ● Wire frame ○ Solid

Subdivision: ● Level 1 ○ Level 2 ○ Level 3 ○ Level 4

☐ Boundary ☐ Blend subfigures to figures

Surface Type displays a surface interpolated from the atoms as dots, wire frame, or a tiled opaque surface.
None hides the surface.
Subdivision is the density of the dots or tile vertices. Higher levels yield a smoother surface.
Boundary shows the 2D intersection of each displayed image X-Y-Z plane with the model's surface.
Blend subfigures to figures shows the real surfaces at hinges where they pull away from either figure.



☐ Boundary

X Plane: ☐ 7.26

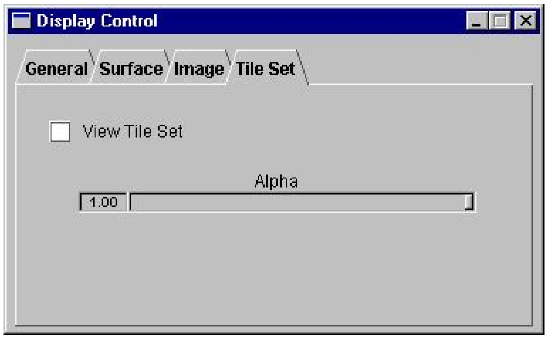
Y Plane: ☐ 5.03

Z Plane: ☐ 6.53

Intensity Min 0.00

Intensity Max 1.00

The **Image** tab controls image display and, surprisingly, the image match term computed in optimization.
Boundary (same button as in **Surface** tab)
On/Off shows any of the 3 image planes.
Position of each plane, in patient coordinate space, selects which 2D slice through the 3D image to draw.
Intensity Min/Max performs clamped intensity windowing: pixel values below **Min** are drawn black, values above **Max** are drawn white, and values between the two are linearly mapped to a grayscale ramp. Min larger than Max is undefined. These are in unit color space and so depend on the pixel values actually present in the image and so might not be portable across images. The image match term is computed from the post intensity windowed images, not from the original image intensities, to allow manipulation of the target image on the fly.



☐ View Tile Set

Alpha 1.00

View Tile Set shows the tileset surface
Alpha is the tileset's transparency. Tiles do not interact well with image planes in that the tiles may disappear if an image plane is in front of it.

10.3.2 Windows -> Optimizer Control

Fits a model (with an optional training image) to a target image. The user-guided sequence of optimization "stages" (methods) is presented to you in decreasing scale until the atom deformation stage. Boundary displacement is separate and not discussed here. The sequence varies based on what's in the current model, described in detail below:

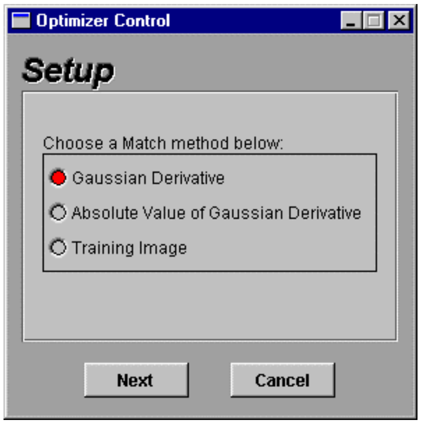
1. Setup - all models (required)
2. Model Stage - all models (optional)
3. If model has multiple figures:
   a. Main Figure Stage (optional)
   b. Atom Stage - for main figure only (optional)
4. Repeat, once per sub-figure:
   a. SubFigure Stage (optional)
   b. Atom Stage - for sub-figure only (optional)

Each figure gets evaluated down to the figure’s atom stage before starting on the next figure, that is, the second figure gets it’s subfigure and atom stages before the third figure gets it’s subfigure and atom stages.

Atom selection is managed by Pablo to indicate to the user which figure is being optimized. ***Only the selected atoms may move, so select them all before starting optimization.*** Since atom selection will be managed by the program, the user does not have to know the order of objects, but the user may control which atoms are to be optimized in some stages by selecting them before the optimization setup. The user also has control over which stage to perform next - that is, to repeat the previous stage, cancel out of all stages and go back to the beginning, or proceed to the next stage in the order listed above - so as to guide the program when it does not find the absolute maximum of it's objective function.

All but the atom stage iterate over changes in a similarity transform until a local maximum is found, seeking the best position, pose, isotropic scale and elongation (along the largest atom dimension) for the model, figures or atoms that matches the image but is penalized for moving too much from the original position/pose/scale both in terms of the image space and of object-object space. Press **Stop** if the model goes awry and Pablo will pause after the current iteration, which could be seconds to minutes.

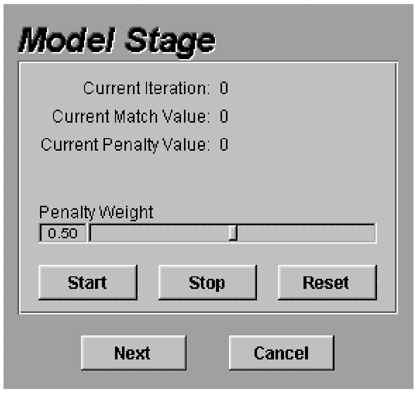
In all **Optimization Control** dialog boxes, **Next** displays the next pane, chosen in the sequence as explained above, and **Cancel** stops optimization and hides the **optimization control** window. The optimization stages are, using stage numberings from above:



Choose a Match method below:
● Gaussian Derivative
○ Absolute Value of Gaussian Derivative
○ Training Image

1 - Setup specifies the image match to be used for all the optimizations that follow, until **Cancel** is pressed. Intensity windowing settings are used (see **Display Control -> Image** tab), but this seems to introduce noise into our objective functions and so it is recommended that a wide-open window be used (0 .. 1).
Gaussian Derivative is for lighter objects on darker backgrounds, or vice-versa if ***positivePolarity*** is 1.
Absolute Value of Gaussian Derivative is the same as above

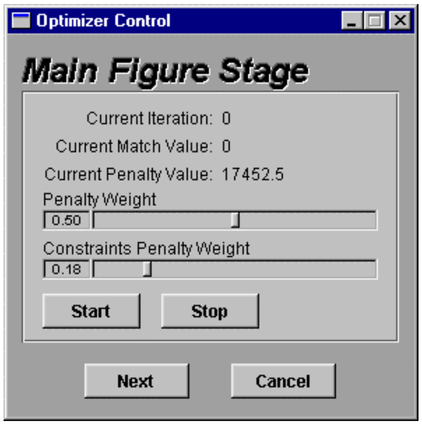
but does not care which side is darker as long as there is a difference in intensity.
Training Image correlates a second image with the currently loaded image in a collar around the implied boundary. Click **Next** and Pablo will prompt for the training image's filename. This filter responds well under almost any condition but is slower than the other methods.


Model Stage

Current Iteration: 0
Current Match Value: 0
Current Penalty Value: 0

Penalty Weight 0.50

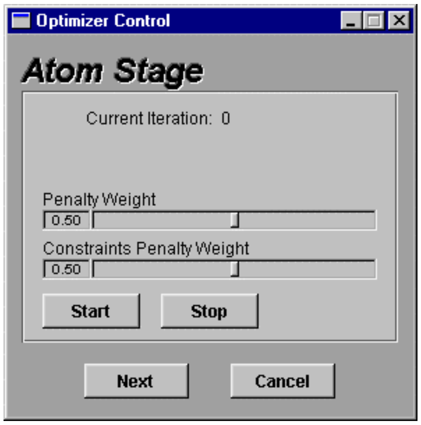
2 - Model Stage fits the model to the image using a similarity transform (pose, position, isotropic scale) plus elongation (along the longest figure axis as measured by the highest atom count).
Penalty Weight is the exponential strength of the geometric penalty that pegs the model close to it's starting position/pose/scale. Decrease the weight to allow the model to wander more.
Start begins iterating the fitting, using the current position/pose/scale as an anchor from which the model may not wander too much. Iterations will continue to accumulate until no more improvements are seen.
Stop temporarily ceases iterating the fitting at the next possible time. Resume from where it stopped by pressing **Start** once again.
Reset uses the current position/pose/scale as the starting position/pose/scale and may take up to 10 seconds to complete. Uses: you move the model, eg, because it wandered too much so you pressed **Stop**; you want to perform a second registration using a prior registration as a starting position/pose/scale, eg, you want to change the Penalty Weight halfway through the registration.



Main Figure Stage

Current Iteration: 0
Current Match Value: 0
Current Penalty Value: 17452.5
Penalty Weight 0.50
Constraints Penalty Weight 0.18

3.a - Main Figure Stage is the same as **Model Stage** except that just figure 0 may move and object-object constraints are added. The **Constraints Penalty Weight** .slider controls an exponential weighting factor applied. Higher numbers give more influence to the object-object constraints.



Atom Stage

Current Iteration: 0

Penalty Weight 0.50
Constraints Penalty Weight 0.50

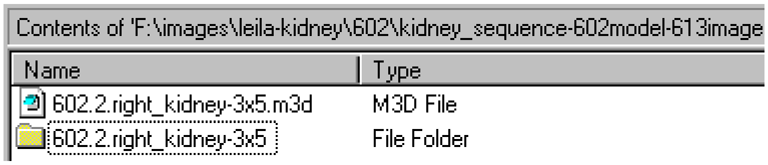
3.b – Atom Stage allows each atom in figure 0 to vary in position, pose, r (length of B vector) and theta (angle between B and Y vectors). Atom elongation is not currently optimized but is used to position of the implied boundary surface.

Stages for sub-figures are the same as for the main figure but add a slider for constraints penalty weight that balances it against the similarity transforms penalty weight.

Auto-save - to review the model as it changes during fitting.

The features in the blue section below must be enabled by a programmer

The model is written after each iteration. Enable before pressing **Start** by making a (folder) in the model's folder that's named the same as the model except for the ".m3d" extension:



Contents of 'F:\images\leila-kidney\602\kidney_sequence-602model-613image

Name	Type
602.2.right_kidney-3x5.m3d	M3D File
602.2.right_kidney-3x5	File Folder

In this lower folder will be written a series of model files, one after each iteration. Files are named **registration_0001.m3d**, etc. The filename's number is the iteration. *Pablo* will not write models if the directory does not exist and it will overwrite old model files.

10.3.3 Windows -> Model Window

Displays the 3D model window. See Mouse Buttons for more interactions.

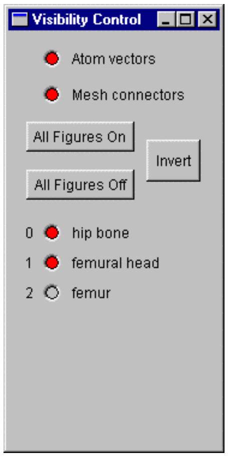
10.3.4 Windows -> Atom Cut Planes

Displays checkboxes for the 6 atom Cut Plane windows. See Slice View Windows. These windows may be drawn slowly and they do not respond to the mouse.

10.3.5 Windows -> Ribbon Window

[*DIMMED*: displays the Ribbon window. It is not clear if this window is useful yet.]

10.3.6 Windows -> Visibility Control



Sets visibility of all atoms' vectors and mesh connectors, and individual figures. Use to simplify the model view: to temporarily remove obscuring figures, to limit atom selection mouse actions to specific figures, for illustrative purposes, to teach model building. Figures exist even when hidden. [BUG: the marked atom is visible when in hidden figures.]

Atom vectors draw the B and involute vectors on each atom.
Mesh connectors draw the lines between atoms
All Figures On/Off turn all of the figures attributes visible or invisible.
Invert: visible figures are hidden and invisible figures are drawn.

The bottom section indicates each figure's visibility. Each line has the figure's ID, visibility, and each figure's name, if it exists. Names are assigned by **Edit -> Add Quad Figure...**, section 10.2.

11 Console window

Text messages are printed here mostly as diagnostics. Users can view the filenames loaded and saved and the file Pablo is seeking for your **preferences**.

12 Editing Model Files

Model files need to be edited by hand with a text editor (vi, emacs, Notepad, Word, etc) because some features are not yet supported within the user interface. Model files use a nested folder format, similiar in concept to files on disks, called the ***PaulY nested folder*** format. which is explained best with a sample in hand:

```
model {
   figureCount = 5;
   figureTrees {
      count = 2;
      tree[0] {
         blendAmount = 0;
         blendExtent = 0;
         childCount = 1;
         figureId = 0;
         linkCount = 0;
         child[0] {
            blendAmount = 0.5;
            blendExtent = 0.5;
            childCount = 0;
            figureId = 1;
            linkCount = 2;
            link[0] {
               primitiveId = 4;
               t = -1;
      }
   }
          …continued in next column…
```

```
   Continued from last column…

   figure[0] {
      name = hip bone;
      numColumns = 5;
      numRows = 3;
      positivePolarity = 1;
      positiveSpace = 1;
      type = QuadFigure;
      color {
         blue = 0.1;
         green = 0.2;
         red = 0.8;
      }
      primitive[0][0] {
         elongation = 1.2;
         qw = 0.889735;
         qx = -0.228895;
         qy = 0.388902;
         qz = 0.0687963;
         r = 0.0276761;
         selected = 1;
         theta = 72.4705;
         type =EndPrimitive;
         x = 0.413206;
         y = 0.339683;
         z = 0.123524;
      }
      primitive[0][1] {
         elongation = 1.2;
         qw = 0.989139;
         ...snipped...
      }
   }
}
```

A **folder**, in curly brackets associates some values together and keeps other apart:

```
name {
  contents...
}
```

In the sample above, `model` is the name of a folder whose contents are delimited by the first { and last } (color-coded in red above). Note that you must write the { on the same line as the folder's name, and that the } must be on a line by itself.

Folders may be nested within other folders. The model folder contains a blue folder for `figureTrees` and a orange folder for each. Each figure folder in turn holds folders for the figure's color and for each of the figure's atoms.

Values - a value is assigned to a name by an assignment line of the form:

name = *value*;

The names are reserved keywords - only a programmer may create new names. The values can be integers or real numbers or even short words, but all assignment lines must end with a semicolor (`;`). That assignment is specific to the folder in which one finds the assignment line, for instance, the `elongation` you assign to `primitive[0][0]` will not be confused with the `elongation` for `primitive[0][1]` because of the context, even though both lines look identical:

elongation = 1.2;

For multiple instances of a name or folder to exist within the same folder, they must be distinguished by one or more trailing numbers, each in square brackets, eg, `figure[0]` or `primitive[0][1].` In Pablo, this number is the figure ID when used in the first instance, and the index to the atom is the figure-specific atom id when the atoms are sorted in alphabetical order. An example of the latter for a 3-row, 5-column model:

primitive[0][0] = ID 0	primitive[1][0] = ID 5	primitive[2][0] = ID 10
primitive[0][1] = ID 1	primitive[1][1] = ID 6	primitive[2][1] = ID 11
primitive[0][2] = ID 2	primitive[1][2] = ID 7	primitive[2][2] = ID 12
primitive[0][3] = ID 3	primitive[1][3] = ID 8	primitive[2][3] = ID 13
primitive[0][4] = ID 4	primitive[1][4] = ID 9	primitive[2][4] = ID 14

Robust & flexible - values and folders may be missing and may be assigned default - but not necessarily useful - values by the program so that it will not crash. This is useful so that cleverly written programs can add keywords to the format and still allow old model files to be read with reasonable results, although this is not always possible. Properly formatted extra values, unrecognized by the program, will be silently ignored, again not crashing the program. This is useful if multiple programs will be updating the model file with their own values, perhaps that only the authoring program would care to understand.

The lines that you may need to edit by hand are listed in the sample model file above. Details:

- color - each figure may have a color assigned to it. Colors are represented as combinations of various amounts of Red, Green and Blue in the unit color space. Some examples:

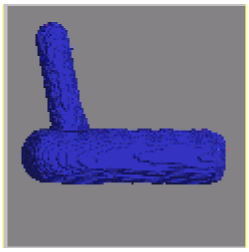
`red=1;`
`green=0;`
`blue=0;`
	The color red

`red=0;`
`green=1;`
`blue=1;`
	Cyan is equal parts green and blue

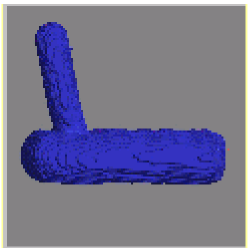
`red=1;`
`green=.5;`
`blue=0;`
	Orange has twice as much red than green

Unspecified color components default to those of cyan. Assign this inside each figure's folder.

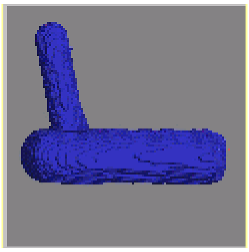
- `figureTrees` - are required to perform any type of fitting (optimization) or to display blending surfaces. See **Attach a Subfigure**, section 10.2**,** for the new user interface to these features. Each tree represents a hinge: a set of connections between pairs of a ***parent*** coordinate (u,v,t) and a ***child*** atom. It is required that the child figures' atoms at the ***hinge*** be positioned somewhere on the parent's surface. The parent and child figure IDs are the ***N***'s from the "figure[***N***]" folder names. The figureTrees folder is in the model folder.
  In the figureTrees folder are tree[***N***] folders, where you'll specify the `blendAmount` and `blendExtent` used for generating a surface both for display and for generating the geometric match (currently we calculate distance from the blended surface as one of the inputs) when blended figures are fitted. blendExtent indicates how far into the figures should be blended. For example, the web between human fingers is a low amount and the web between a duck's fingers is a high amount. blendAmount indicates the amount of bulge in the blend. Real numbers [0..1]. Visually:



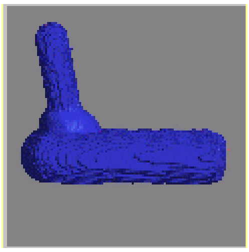
blendAmount=0
blendExtent=0



blendAmount=0
blendExtent=1



blendAmount=1
blendExtent=0



blendAmount=1
blendExtent=1

- `polarity` - both Gaussian image matches assume that the image intensities follow a pattern of bright object on a dark background (**`positivePolarity = 1`**;) or dark object on a bright background (**`positivePolarity = 0;`**). The background includes all the objects in the scene (even air) that are not the object. Although this is simplistic, it works in many cases. Assign this inside each figure's folder.
- `positiveSpace` **-** [LATER: 1 for a protrusion figure (figure is a solid part of the object) and 0 for indentations (figure is a hole in the object).]

13 Items to document

- Section 9.2: **Slice View windows** needs a picture
- Section : **Edit -> Constraints** should be

14 Index

Bold entries indicate buttons or window titles. Other entries are concepts or actions.