I-DO: A “Deformable Organisms” framework for ITK

Release 0.50

Chris McIntosh and Ghassan Hamarneh

July 23, 2006

Medical Image Analysis Lab
School of Computing Science, Simon Fraser University
Burnaby, BC, Canada
{cmcintos,hamarneh}@cs.sfu.ca

Abstract

Medical image analysis is an important problem relating to the study of various diseases. Since their introduction to MICCAI in 2001, ”deformable organisms” have emerged as a fruitful methodology with examples ranging from 2D corpus callosum segmentation to 3D vasculature and spinal cord segmentation. Essentially we previously have developed an artificial life framework that complements the geometrical and physical layers of classical deformable models (snakes and deformable meshes) with high-level behavioral and cognitive layers that facilitate anatomically-driven control mechanisms. This paper describes the integration of deformable organisms into the Insight Toolkit (ITK) www.itk.org. In our proposed implementation we attempt to bridge the ITK framework and coding style with deformable organism design methodologies. In the interest of open science, as the framework develops it will serve as a basis for the community to develop new deformable organisms as well as experiment with those recently published by our group. Further, as the design of the ITK Deformable Organisms (I-DO) is highly modular, researchers and developers can exchange components (spatial objects, dynamic simulation engines, image sensors, etc) allowing in the future for fast development of new custom deformable organisms for different clinical applications.

Contents

1 Introduction

In medical image analysis strategies based on deformable models, controlling the deformations of the models is a desirable goal to produce proper segmentations. Incorporating expert knowledge to automatically guide deformations cannot be easily and elegantly achieved using the classical deformable model low-level energy-based fitting mechanisms. Deformable Organisms (DOs), are a decision-making framework for medical image analysis that complements bottom-up, data-driven deformable models with top-down, knowledge-driven mode-fitting strategies in a layered fashion inspired by artificial life modeling concepts. Intuitive and controlled deformations are carried out through behaviors. Sensory input from image data and contextual knowledge about the analysis problem govern these different behaviors.

Since their introduction in 2001 [3], various DOs-based approaches for medical image analysis have been developed (Figure 1). In this original work, a variety of DOs where demonstrated with applications to locating the lateral ventricles, caudate nuclei, and putamina structures in transversal brain magnetic resonance image (MRI) slices, as well as DOs for the segmentation of vessels in 2D angiography. In [4], DOs were augmented to include physically-based and controlled deformations demonstrating an application to corpus callosum segmentation in mid-sagittal magnetic resonance images (MRI). Recently, DOs were extended to 3D and applied to vascular segmentation and analysis. The so called ‘vessel crawlers’ were equipped with sensors, decision modules, and deformation layers suited for vasculature [7]. An extension of that work introduces DOs for spinal cord segmentation and analysis and demonstrates the ability to efficiently replace modules of existing DOs to create new solutions. The ‘spinal crawlers’ no longer possessed a decision module to detect branching and their sensors were adapted to detect elliptical cross sections [6]. In each case DOs have demonstrated their key advantages over other leading techniques. Namely, their ability to produce increased accuracy, allow intuitive user-interaction to control or repair the segmentation where other methods would require being restarted with some type of parameter adjustment, facilitate greater analysis and labeling abilities than those methods producing binary outputs, the ready ability to incorporate image or shape-based prior-knowledge, and a modular framework allowing for incorporating new sensors (image filters), decision models, shape representations, and deformation mechanisms.

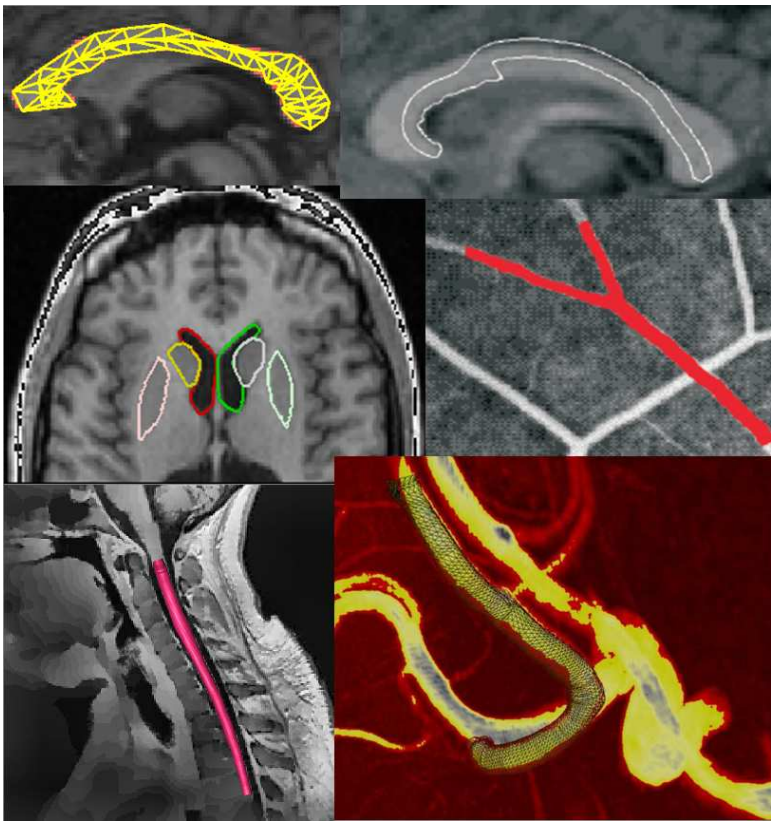

Figure 1: An assortment of deformable organisms showing(left to right, top to bottom): Physically-based corpus callosum, Geometrically-based corpus callosum, Putamina and ventricle organisms, 2D Angiography, 3D 'spinal crawler', and 3D 'vessel crawler'. Related images and videos can be found at http://mial.fas.sfu.ca/researchProject.php?s=157

Though a summary is provided here, a complete research-oriented look at DOs can be found in [5]. DOs are built following a multilevel AL modelling approach consisting of four primary layers: ***cognitive, behavioral, physical, and geometrical***. Specifically, the cognitive layer makes decisions based on the DOs current state, anatomical knowledge, and its surrounding environment (the image). Decisions could be made to sense information, to deform based on sensory data, to illicit help from the user, or to terminate the segmentation process. All of these actions are described under the behavioral layer of the organism, and they rely upon both the physical and geometrical layers for implementation. For example, in the context of our 'vessel crawlers' [7], the act of moving towards a sensed target location is described by the *'growing'* behavioral method. The cognitive center gathers sensory input using the *'sense-to-grow'* sensory module, decides the correct location via the *'where-to-grow'* decision module, elicits the act of *'growing'* , and then conforms to the vascular walls by *'fitting'*. In turn, each of these methods relies upon the physical and geometrical layers to carry out tasks, such as maintaining model stability. Consequently, we have a framework with many independent layers of abstraction, each built upon the implementation of independent modules and or processes.

We begin with a motivation of our ITK-Deformable Organisms (I-DO) framework in section 1.1, and a discussion of the general requirements of DOs that the framework is set out to meet in 1.2. Sections (2.1-2.7) provide an overview of how each layer is designed and implemented in the framework. We summarize in section 3. The appendices provide the most information on using the framework with a requirements listing

(section A), examples of layers and organisms (section B), a description of our visual interface (section C), a guide to building and running your first organism (section D), and information on extending organisms and the framework (section D).

1.1 ITK Deformable Organisms: Motivation and Introduction

Previously, the major drawback of DOs has been their restriction to a closed-source MATLAB framework. Though straightforward and intuitive in design they are not readily extendable by the general medical image analysis community in this form. ITK, however, enjoys a large user base and exemplifies the notion of an open-source, adoptable, and extendable framework. Furthermore, the incorporation of ITK grants DOs access to faster processing, multi-threading, additional image processing functions and libraries, and straightforward compatibility with the powerful visualization capabilities of the Visualization Toolkit (VTK) `www.vtk.org`.

1.2 DOs Requirements

DOs are constructed through the realization of many abstract and independent concepts/layers (cognitive, behavioral, physical, geometrical, sensors). As such, a DO framework must reflect this modular design by allowing users to replace one implementation (layer) for another. For example, new shape representations should be introducible without re-designing existing cognitive layers. To this end, the interface between layers must be consistent across implementations (plug and play), and clearly defined.

The framework must also be extendable, allowing it to grow and advance as the concept of DOs does. That is to say, it should support current research into new types of DOs designed for different applications, with increasingly advanced decision making and deformation abilities.

2 Implementation

This section provides details on the implementation of the I-DO framework. Each section (2.1-2.7) describes a DO layer in detail within the context of our I-DO framework. A high level overview of the DOs framework is shown in Figure 2.

2.1 Organism

The `organism` is the abstract base class (ABC) that acts as a container for most of the framework. Each organism posses its world, a control center, a physics layer, and a geometrical layer. It provides public interfaces through which users can add deformations and behaviors, as well as attach the cognitive, physical, and geometrical layers. Its important to understand that as an ABC, the organism class itself is not instantiated. It is designed as such so that no matter the derivation (type of organism), a DO application can simply call its associated public interface. Consequently, of most interest are the derived classes themselves.

The `itkOrganism` derived class can be instantiated and used as a fully functional organism, or can be used as a base class of another more specialized organism. It inherits from both the Organism ABC, and ITK's ImageToImageFilter class. Though many other classes could be used, the ImageToImageFilter class allows these particular DOs to be incorporated as autonomous tools in existing ITK filtering pipelines (taking as

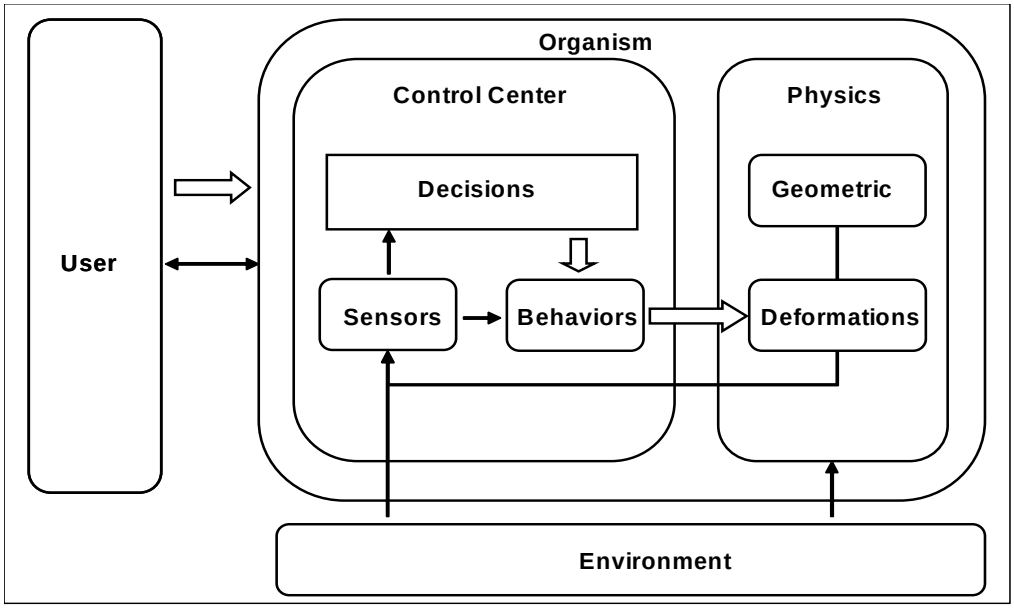

Figure 2: The basic outline of the deformable organism framework. Dark arrows represent directions of communication between objects, while hollow arrows represent one class running another's public `run` method, and encapsulation represents one class containing another. For example, the behavior class controls the deformations class through the physics class.

input an image and producing as output a segmented image). More details on this derived class are provided at http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_organism.html.

2.2 Control Center

The control center is designed to handle all "intelligent" aspects of the organism. It has associated behaviors and sensory modules, and provides the organism with its ability to make decisions (e.g. next behavior to run, image data to sense, etc.). It monitors the status of the behaviors, deformations, and sensors, then makes decisions based upon their states and outputs.

Consequently, this class exploits much of the complex versatility of the framework obtained through the use of ABCs, streams, and structures. Through a single list of sensors and behaviors, the cognitive center can perform a variety of actions on any defined geometrical or physical type regardless of the varying input requirements they may have. For example, the decision to "translate" will trigger a spatial translation behavior, which will in turn trigger the appropriate translate deformation as it pertains to the particular physical layer of the model. All without the cognitive layer having any regard for which derived physical layer and deformation class, or geometrical layer and shape representation is being called.

The control center accomplishes this by using a "run-by-name" design methodology, where once it decides upon (or is asked to run) a particular named behavior it will search its list of known behaviors for one with the matching name.

By calling a control center's `Update` method the organism will conceptually cause the control center to do its thinking. If no current behavior exists it will decide on one (via the derived classes provided `DecideNextBehavior` method). Otherwise, it will check the status of the behavior (via its `IsFinished` method), then clean up (`CleanUp` method) and decide on a new behavior if it has finished, or update it

(`Update` method) if it has not.

http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_control_center.html

2.3 Sensor

Organisms perceive their surroundings through sensory modules. They provide a means by which to gather statistics and characteristics of its own geometry and the world (image data) in which it resides. At any given time a decision function may possess many different sensory objects, each of which can report back different sensory information (e.g. gray level intensity, gradient magnitude and direction, texture features, etc). It is important to note that some sensors will be implementation dependent, while others will not. For example, it makes no sense to run a vasculature bifurcation sensory module on a corpus callosum organism because the latter is only 2D and has a completely different topology and appearance characteristics.

In order to run a sensor one must use its publicly defined `sensorIn` and `sensorOut` types to create the input arguments and receive the output. This allows maximum flexibility in the parameters a sensor can have, while still enabling any sensor to be ran abstractly. Through this flexibility users can setup and run complex pipelines of ITK filters within the sensors, while passing their variety of input requirements in via the sensorIn type.

http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_sensor.html

2.4 Behavior

Behaviors are basically actions, or sequences of actions. As such, each behavior has a name, a state, a pointer to the physical layer, and multiple sub-behaviors, and deformations. To ensure meaningful interaction with other organisms and users each behavior has a name. So for example, despite the action “running” being carried out differently by different animals each can always be told to run, or report that it is running. Upon being executed the behavior simply begins executing its main body. Again, the behavior class is simply an ABC. So let’s consider a few example derived classes to illustrate the subtleties of this class.

The first simple example behavior is ‘inflate for 30 cycles’. The act of the organism inflating itself is physics system dependant, so the behavior runs its associated inflate deformation by calling the `runDeformation` method of the physics object. The behavior then sets its status to incomplete. At the next run of its `decideNextBehavor` method the control center checks the status of the inflate behavior, and upon seeing `incomplete` runs the behavior’s `update` method. Now upon executing, the behavior checks to see if its ran for 30 cycles by examining the physics objects time counter, if so it sets its status to `complete`. Now suppose a more complex behavior inflates then moves forward. First it runs its inflate sub-behavior by checking its list of behaviors for one with a matching name, then checks its status. Upon confirming that its first sub-behavior is complete it moves forward, and sets its own status to `complete`.

It is also possible for the `decideNextBehavior` method to use a decision function to decide that a given behavior is finished executing, regardless of its current status. Of course, a behavior may also fail, resulting in some action by the control center.

Sub-behaviors are smaller behaviors performed as part of a larger action. This enables significant levels of abstraction, allowing users to issue single commands and carry out vast and complex sequences of actions, or small exact ones. For example, one could instruct the organism to simply inflate, or one could tell it to segment which includes inflation [2].

http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_behavior.html

2.5 Physics

The `Physics` layer is responsible for simulating the deformations and handling the organisms interaction with its environment through external forces. Each physics object possesses a list of executable deformations and a geometric object. The main public interface of interest is the simulate method, which actually causes forces to be calculated and exerted. Again, as the physics layer is merely an ABC, it is of much more interest to discuss this class through an example of one of its derived classes.

An example derived class is the `Phys_Euler` physics object. This implementation relies on the simulation of a spring-mass system to perform deformations. When the organism calls the simulation method, the `Phys_Euler` object runs its simulation cycle for a set number of times, and then increments the global timer. During the simulation cycle the physics layer has control of the CPU, and can not be interrupted. Consequently, the length of this cycle should be kept short in order to allow the organism to check behavior status states, run decision functions, etc. If the length of the cycle is longer than the time required to run a single behavior, then the organism will basically be idle for the remaining iterations. However, the running deformation also has a runtime set by its calling behavior. So the physics object can stop simulating after that runtime has expired.

http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_physics.html

2.6 Deformations

The `Deformation` classes manipulate the geometry of the organism. For example, in a physically-based spring-mass implementation deformations move nodes, actuate springs, apply forces, and basically deform the geometrical model. Much like behaviors, each deformation has an associated status and runtime, as well as run method for its public interface. However, in this case deformations do not posses many sub-deformations.

As an example let us consider the inflate deformation. Upon being executed by an associated behavior it begins applying forces normal to the model's surface, causing it to inflate. In the case of a spring-mass system these forces may be carried out by applying forces on individual nodes, or by increasing the rest-lengths of springs. The concept of reversing the inflation to a deflation once the organism has passed from dark to bright (for example when segmenting dark object on a white background) is delegated to the control center of the organism, and does not take place here. Instead only low-level tasks like actuating springs, moving nodes, etc are carried out. This enables the execution of both prior and learned deformations [8], where learned deformations are carried out by the associated learned behavior causing a sequence of spring actuations. However, if the underlying shape representation is level sets based the inflation takes the form of adding a constant to the embedding function in order to expand the zero-level set.

2.7 Geometric

The `Geometric` object houses the the actual topology of the organism. It handles adding and removing nodes, as well as reading and writing the meshes to file. Consider two different hypothetical derived classes: a VectorGeometry class and a TubularGeometry class. The VectorGeometry class would be implemented entirely with vector geometry, while the TubularGeometry class would also derived from an ITK spatialobjects class. Both classes would provide the same public interface in terms of getting nodes, setting nodes, writing to file, reading from file, etc. However, they each would allow the user to take advantage of their inherit properties. So the user can write a custom sensory class, that uses the additional functionality of the

TubularGeometry class without having to modify any internal code of the organism itself. In essence, the user can be dependent on the implementation when they want to be, and remain totaly independent in other situations by sticking to the `Geometric` base class interface.

3 Conclusions

We have developed a powerful new framework for medical image segmentation and analysis that offers both great flexibility and rigid design enforcement, thereby, ensuring maximum reusability, portability and sustainability. Our framework makes use of many powerful features in ITK including filters, meshes, file IO, smart pointers, and spatial objects. We have also created a robust spring-mass physically-based deformation layer, which can be seen as a contribution in itself.

Furthermore, the added ability to convert BYU surfaces or binary volumes into `itk::MeshSpatialObjects` and consequently, into deformable organisms should prove a useful tool allowing level-set refinement, or physics-based interaction with segmentation results of various existing projects. For example, both explicit physically based (spring mass) and implicit level set based classical deformable models are special cases of DOs and their implementation is a special case of the IDO framework. They now simply emerge in IDO by setting the proper geometrical and physical layers (spring mass vs level set) and having behavioral and cognitive layers that simply simulate the deformation dynamics without any top down control or scheduling.

4 Acknowledgements

We would like to thank Andy Rova for his development of the Phys_LevelSet class, Vincent Chu for his role as lead developer of the KWWidgets viewer application (section C), and Aaron Ward for his technical expertise and discussions on fundamental framework design choices.

A Requirements

Though the framework itself only requires ITK 2.4 or greater, building the provided viewer (section C), has additional requirements:

- VTK 5.0.0 `http://www.vtk.org`
- SOViewer (Feb 8, 2006) `http://www.vtk.org/Wiki/SOViewer`
- KWWidgets (Feb 8, 2006) `http://www.kwwidgets.org/Wiki/KWWidgets`

B Examples

B.1 Layer Examples

Various examples of the layers/modules explained in section 2 are available, with details provided in the frameworks online documentation.

- Geom_MeshSpatialObject<dType,nDims, MType, VType>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_geom___mesh_spatial_object.htm
- Phys_Euler<DataType,TGradientImage,nDims,MType,VType>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_phys___euler.html
- Phys_LevelSet<DataType,InputImageType,nDims,MType,VType>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_phys___level_set.html
- Beh_TranslateAll<Type,nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_beh___translate_all.html
- Beh_UniformScale<Type,nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_beh___uniform_scale.html
- Beh_SearchForObject<Type,TInputImage,nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_beh___search_for_object.html
- Def_TranslateAll<Type,nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_def___translate_all.html
- Def_UniformScale<Type,nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_def___uniform_scale.html
- Ctrl_ScheduleDriven<class Type, int nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_ctrl___schedule_driven.html
- Sense_Gradient<DataType,TInputImage, TGradientImage, nDims>
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_sense___gradient.html

B.2 Deformable Organism Examples

There are numerous example DOs included with the framework.

- **itkOrganism<ImageType, ImageType, GradientImageType, dType, nDims>** A derived organism based on a `itk::ImageToImageFilter` that contains no default layers.
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classitk_1_1_itk_organism.html
- **Org_LevelSetSchedule<ImageType, ImageType, GradientImageType, dType, nDims>** A geodesic active contours [1] based DO that uses a schedule driven cognitive layer.
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classitk_1_1_org___level_set_schedule.html
- **Org_EulerSchedule<ImageType, ImageType, GradientImageType, dType, nDims>** A 3D spring-mass [7] based DO that uses a schedule driven cognitive layer.
  http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classitk_1_1_org___euler_schedule.html

C The Visual Interface to I-DO

We have also developed a graphical user interface to the I-DO framework, that allows its users to visualize the geometry of created DOs as well as observe their deformations in real time. It gives the user the ability to load DOs as dll files, while allowing the developer to define customized interfaces via the `DefOrgAdapter` class. The GUI is based on, and therefore requires, KWWidegets, VTK, and SOViewer. Future versions will facilitate interaction with DOs through mouse click driven forces, and possibly other forms of input. Complete documentation of the viewer will be made available at a later date, but many details reside in its doxygen. A binary of the viewer is available for Windows at http://hdl.handle.net/1926/228/viewerApplication.zip.

http://www.sfu.ca/~cmcintos/IDO/doxygen/html/classmial_1_1_def_org_viewer_adapter_base.html

D Guide to users

This section provides information to those who wish to use, or contribute to the framework.

Hello I-DO

In this section we present a simple “Hello [I-DO] World” example that provides a step by step guide to how a new user can build and run a simple DO.

1. Download and compile ITK 2.4 or greater (see www.itk.org).
2. Download (http://hdl.handle.net/1926/228/IDO.zip) and configure the I-DO framework using CMake (www.cmake.org) and the CMakeLists.txt file found in the root-most directory. Make sure to leave ”Build Examples” set to ”ON”.
3. Compile the created project. This will build the I-DO library, and two executables.
4. Run YourBuildDirectory/examples/basic/defOrg_basic from command line, providing input and output image names, a schedule name, and a mesh name. (e.g. cube.mhd out.mhd eulerSchedule3d.txt cubeMesh3d.meta)
5. The DO will run, and output a final binary image using the file name provided.

Users can follow these procedures for any of the provided examples in the examples directory.

- Basic - The same example as shown in “Building A Deformable Organism”. A spring-mass DO using a schedule driven cognitive layer along with a few example behaviors and deformations (Figure 3 top).
- Advanced - A multi-organism application that uses two pre-made DOs in sequence. `Org_EulerSchedule` begins the segmentation process and initializes `Org_LevelSetSchedule` with its output, which then proceeds to refine the segmentation results before writing out to file (Figure 3 bottom).

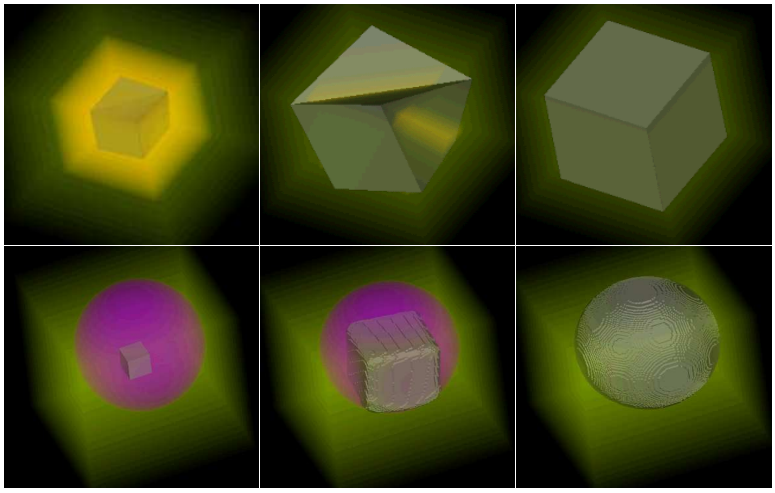

Figure 3: Two example DOs progressing from left to right. Top: The basic example initialized with a cube, performing a `Beh_UniformScale`, and coming to rest. Bottom: The advanced example initialized with a cube, smoothing under `Phys_LevelSet` after a `Beh_UniformScale` under `Phys_Euler`, and coming to rest via image forces. The complete videos are available at http://hdl.handle.net/1926/228/{basic,advanced}.wmv

Building A Deformable Organism

This example walks the reader through creating a DO by individually instantiating and attaching the layers. This is in contrast to using an already created DO, which can be instantiated, setup, and used just like any ITK filter.

The first step is to choose and instantiate a DO shell (one having no built in layers) using the standard ITK `itk::SmartPointer` approach. In this case the DO is an ITK `itk::ImageToImageFilter`, and must be provided with an input image via the `SetInput` method.

```
typedef itk::ItkOrganism <ImageType, ImageType, GradientImageType, float, 3> organismType;
organismType::Pointer testOrg = organismType::New();
std::cout << "Organism created..." << std::endl;
testOrg->SetInput(reader->GetOutput());
```

Now we can begin instantiating and attaching implementations of the layers/components the DO needs to function. For simplicity, all derived classes of a particular layer are prefixed with an abbreviation of that layer (Org for Organism, Ctrl for Control, Beh for Behavior, Sense for Sensor, Phys for Physics, Def for Deformation, and Geom for Geometrical). Next we will instantiate a sensor to calculate the gradient information used as an external force during the deformation simulations by the `Physics` layer.

```
typedef Sense_Gradient<float,ImageType,GradientImageType,3> gradientSensorType;
gradientSensorType::Pointer gradientSensor = gradientSensorType::New();
```

The sensor requires its publicly defined `sensorIn` as input. Here we create a pointer to the class, and set its values. This allows all sensors to be ran from a common `run` method, with their own customized input.

```
gradientSensorType::sensorIn::Pointer input = gradientSensorType::sensorIn::New();
input->sigma = 1.0;
reader->Update();
input->imageIn = reader->GetOutput();
```

The gradient sensor can then be ran. Note that at this time Sensors themselves do not fit into the ITK pipeline, and thus the reader's Update() method must be called prior to running the sensor.

```
gradientSensor->run(input);
```

Finally, its output can be obtained by constructing a sensorOut itk::SmartPointer and providing the appropriate downcast on the pointer returned by the getOutput method.

```
gradientSensorType::sensorOut::Pointer output = (gradientSensorType::sensorOut *) (gradientSensor->
```

Next create the Physics and Geometrical layers. Notice that the type of external force image is provided as an input type to the Physics layer.

```
typedef Phys_Euler<float,GradientImageType,3> PhysLayerType;
typedef Geom_MeshSpatialObject<float,3> GeometricType;

PhysLayerType::Pointer physLayer = PhysLayerType::New();
GeometricType::Pointer geomLayer = GeometricType::New();
```

Then set the Physics layer to use the external force image calculated by the gradient sensor and the newly constructed Geometrical layer, and setup the topology of the Geometric layer (in this case an ITK itk::MeshSpatialObject). Finally, attach both to the Organism.

```
physLayer->setExternalForces((void *) &(output->imageOut));
physLayer->setGeometry(geomLayer);
std::cout << "External forces set." << std::endl;

geomLayer->readTopologyFromFile(topologyInputFileName);
std::cout << "Topology read from '" << topologyInputFileName << "'..." << std::endl;

testOrg->setPhysicsLayer(physLayer);
testOrg->setGeometricLayer(geomLayer);
std::cout << "Physics and Geometric layers added..." << std::endl;
```

Create a Cogntive layer, set its appropriate options, and attach it to the DO. In this case it only requires a Schedule text file (e.g. eulerSchedule3D.txt).

```
Ctrl_ScheduleDriven<float, 3>::Pointer cgL = Ctrl_ScheduleDriven<float, 3>::New();
cgL->setSchedule(scheduleFileName);
testOrg->setCognitiveLayer(cgL);
```

Now begin creating and attaching simple behaviors, and deformations. Note in this case, the behaviors and deformations do not require any additional parameters or settings.

```
Beh_TranslateAll<float, 3>::Pointer beh1 = Beh_TranslateAll<float,3>::New();
Beh_UniformScale<float, 3>::Pointer beh2 = Beh_UniformScale<float,3>::New();
Def_Translation<float, 3>::Pointer def1 = Def_Translation<float,3>::New();
Def_UniformScale<float, 3>::Pointer def2 = Def_UniformScale<float,3>::New();
```

```
testOrg->addBehaviour(beh1);
testOrg->addBehaviour(beh2);
testOrg->addDeformation(def1);
testOrg->addDeformation(def2);
```

Attach a more advanced behavior and set its additional parameters. In this case it needs an image and a `Geometric` pointer for its internal `Sense_AvgIntensity` sensor.

```
Beh_SearchForObject<float,ImageType,3>::Pointer beh3 = Beh_SearchForObject<float,ImageType,3>::New(
beh3->image = reader->GetOutput();
beh3->geomLayer = geomLayer;
testOrg->addBehaviour(beh3);
```

The `Organism` is ready to run. Calling `Update()` on the writer will cause the DO to simulate for a set amount of DO time. Here we set the DO to `run` for 25 iterations with a single `Update()`.

```
testOrg->setRunTime(120);
writer->SetInput(testOrg->GetOutput());
try
{
    writer->Update();
}
catch(itk::ExceptionObject & err)
{
    std::cout << "ExceptionObject caught!" << std::endl;
    std::cout << err << std::endl;
    return -1;
}
```

Finally, in addition to the binary output available on the writer the DO's mesh can be written back to file.

```
testOrg->writeNodesToFile(nodeOutputFileName);
std::cout << "Nodes written to '" << nodeOutputFileName << "'." << std::endl;
```

Extending Existing DOs

Extending existing organisms is as easy as following the *Building A Deformable Organism* example and attaching additional layers.

Creating New DOs and Layers

Detailed information about creating new DOs and layers will be included in this document in a later revision. In the mean time, interested users are referred to the doxygen documentation which outlines how each pure virtual function of the ABCs should be defined in a derived class. We will also provide skeleton code generators, that will give those wishing to create new layers a "fill in the blanks" option.

http://hdl.handle.net/1926/228/doxygenManual.pdf
or
http://www.sfu.ca/~cmcintos/IDO/doxygen/html/index.html

References

[1] Vincent Caselles, Ron Kimmel, and Guillermo Sapiro. Geodesic active contours. In *ICCV*, pages 694–699, 1995. B.2

[2] Laurent D. Cohen. On active contour models and balloons. *CVGIP: Image Underst.*, 53(2):211–218, 1991. 2.4

[3] Ghassan Hamarneh, Tim McInerney, and Demetri Terzopoulos. Deformable organisms for automatic medical image analysis. In *MICCAI*, pages 66–76, 2001. 1

[4] Ghassan Hamarneh and Chris McIntosh. Physics-based deformable organisms for medical image analysis. *SPIE Medical Imaging*, 5747:326–335, 2005. 1

[5] G. Hamarnerh and C. McIntosh. *Parametric and Geometric Deformable Models: An application in Biomaterials and Medical Imagery*, chapter 12: Deformable Organisms for Medical Image Analysis. Springer Publishers, 1 edition, 2006. 1

[6] C. McIntosh and G. Hamarnerh. Spinal crawlers: Deformable organisms for spinal cord segmentation and analysis. *MICCAI*, 2006. 1

[7] C. McIntosh and G. Hamarnerh. Vessel crawlers: 3d physically-based deformable organisms for vasulature segmentation and analysis. *IEEE Conference on Computer Vision and Pattern Recognition*, 2006. 1, 1, B.2

[8] D. Terzopoulos, X. Tu, and R. Grzeszczuk. Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life*, 1(4):327–351, 1994. 2.6