

The Scientist's Expert Assistant (SEA) Simulation Facility

Release 1

July, 2000



National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland

NGST Scientist's Expert Assistant (SEA) Simulation Facility

Release 1

June 2000

Prepared by:

Advanced Architectures and Automation Branch

Date

Approved by:

Advanced Architectures and Automation Branch

Date

Goddard Space Flight Center
Greenbelt, Maryland

Table of Contents

Simulation in the SEA	5
Simulation in the SEA	5
What is the Approach?.....	5
Architecture	7
<i>Photon Pipeline</i>	7
Pipeline Splitting.....	9
Aggregation	9
Visualizations.....	10
Imaging.....	10
Spectroscopy.....	10
Other Visualizations.....	11
Pipeline Processing	11
Simulation Data	14
Timing.....	15
How do you use the Simulation Facility in the SEA?.....	16
Okay, so these are the things we want to do, so how does the user do them?.....	17
Implementation: A Difficult Balancing Act.....	20
Computation Load	21
Memory Requirements.....	21
Binning - A Balanced Solution	21
Evaluation	22
Final Comments.....	23
In the Future.....	23
The Current Design.....	24
Simulation Class Design	24
The Classes	24
High Level Pipeline Processing	30
User Interaction.....	31

Table of Figures

Figure 1: Example Photon Pipeline	7
Figure 2: Pipeline Splitting	9
Figure 3: Multi-Object Spectroscopy.....	11
Figure 4: Example Pipeline Processing	13
Figure 5: Data Cube	14
Figure 6: Simulation Module	17
Figure 7: Simulation Facility General Layout	18
Figure 8: Binning Example	22
Figure 9: Simulation Class Design.....	24
Figure 10: Simulation High Level Pipeline Processing	30

Simulation in the SEA

Below is an attempt to articulate my understanding of what “Simulation” means to the SEA. The paper is intended to stimulate discussion and to act as a basis for a design of the facility.

Primary Goal: To create an integrated Imaging and Spectroscopic Simulation facility within the SEA. The simulation must be scientifically accurate (to a reasonable epsilon) and provide a meaningful and representative simulation (prediction) of what to expect from the real world instruments given the same inputs and constraints. The simulations must be useful to the GO.

That’s a tall order...

There have been a number of simulation efforts around the astronomical community. Currently, we are researching these to see what we can learn/obtain from them. Ideally, we would like to create a framework where we can easily integrate existing algorithms into the SEA.

What is the Approach?

We want to simulate, as closely as possible, the expected results for an astronomical observation. We want to create simulated images that take into account the real world environmental aspects and the behavior of specific detectors.

We believe one way to do this is by using layers of models. During our brainstorm session back in March we came up with the idea of Model layers for simulation. We identified four layers:

1. Target Parameters
2. Intergalactic or Interstellar Parameters
3. Observatory Parameters
4. Instrument Parameters

The idea was the user could modify one set of parameters while leaving the other three models fixed. The user would then request a simulation run. Presumably the user would see the results of his modifications in a simulation image at the end.

While we were thinking about this model idea we conceived a broader approach. We’re trying to model what we know about the real universe and how photons are affected while traveling to our detectors. We feel the new approach discussed below fits nicely with the new SEA internal classes Sandy has been working on. We see the following model classes. Note: the listed attributes are expected to be a subset of the final attributes for each model. We will add attributes as they become necessary.

Emitters – photon sources (This is what the aperture sees of the Target)

- Attributes
 - Emit photons at one or more ranges of wavelengths. May be a function.
 - Magnitude
 - Size
 - Shape
 - Location? (so can properly map over target, or at least for alignment purposes)
- Examples:
 - Point source
 - Extended sources
 - Galaxy
 - Sky background (homogenous extended source covering entire aperture)
 - Diffuse source

- A photon emission plane. Any portion of the background not emitting is considered black or transparent.
 - Can have multiple (stacked) emitters, some of which may overlap. Overlap may be partial.

Attenuators - reduce photon count

- Attributes
 - Size
 - Shape
 - Function of wavelengths (possibly two functions, one for absorption the other scattering?)
 - Can be aggregated

Two subtypes:

1. **Absorbers** –reduce photon count through absorption.
 - Absorb a specific and/or ranges of wavelengths. % absorption as a function of wavelength and shape.
 - Examples:
 - Filters
 - Dust Clouds
 - Earth's Atmosphere
2. **Scatterers (Distorters)** –photons get “redirected”
 - Bend the incoming light in some specified way.
 - Likely these will be used in conjunction with at least one Absorber since no lens or mirror is perfect.
 - Diffuser?
 - Examples:
 - Lenses
 - Mirrors
 - Dust Clouds

Slit, Prism, Grism - whatever Spectroscopy needs

Observatory - Every aberration or environmental attribute contributing to a change in photons

- Attributes
 - Location (as a function of time)
 - All of our current Observatory class attributes
 - Primary Mirror
 - Secondary Mirror (may be N number)
 - PSF (does this vary across the detector?)
 - Random cosmic rays
 - Magnetic Field Effects
 - Atmospheric distortions
 - Dust on the mirror?
 - Internal reflections
 - Internal absorption
 - Thermal Effects
 - Sky Background (isn't this really an Emitter?)

Instrument – structure describing inlet for photons and the path to the detector.

- Attributes
 - All of our current Instrument class attributes

- Focus capabilities.
- Thermal effects?
- Filter (a wavelength specific Attenuator)

Detector - End of the line for those photons. Converts photon counts to pixel values.

- Attributes
 - Arrangement of pixels
 - Type of pixels
 - Size of pixels
 - Photon detection characteristics
 - Temperature
 - Read noise
 - Dark current noise
 - Depth of well
 - Dead (cold) pixels
 - Hot pixels
 - Diffraction spikes
 - Saturation

Architecture

Photon Pipeline

We are envisioning a “photon pipeline” starting at one end with one or more Emitters supplying photons. The photons travel a path through Models of Absorbers, Distorters, Aberrations, Environments, down to one Observatory then one Instrument and finally to a Detector. All those Models affect the probability that a given photon will strike a given pixel (or set of pixels?) on a Detector. Maybe we should think of a stream of photons and each Model affects the dispersion of the stream. See Figure 1

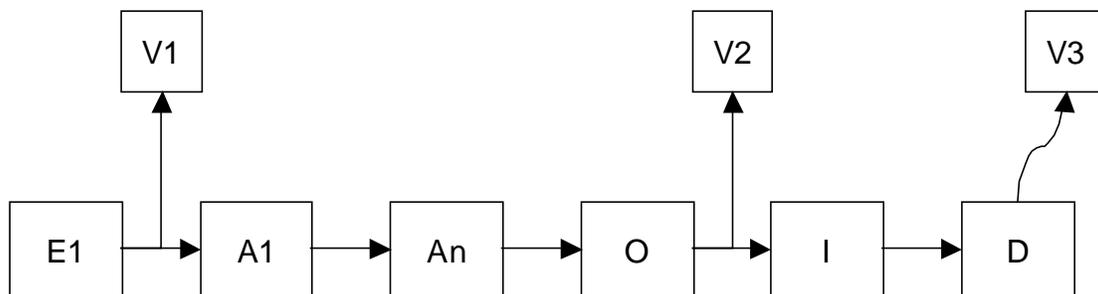


Figure 1: Example Photon Pipeline

E1 - Emitter 1
 A1 - Attenuator 1
 An - Attenuator N
 O - Observatory
 I - Instrument
 D - Detector

V1, V2, V3 - Visualizers. V1 attached to the output of E1. V2 at the output of O. V3 is the final simulation visualizer (attached to the output of the Detector).

There can be any number of Emitters, Attenuators and Visualizers. There can only be one Observatory, Instrument and Detector, except when branching. More on this later.

The above figure is an example of a pipeline. The pipeline will be “pluggable”, meaning: internally the pipeline will have a plug-in architecture. The user will be able to add and remove models as appropriate to their simulation. For example: The user could add another Emitter to the beginning of the pipeline. One Emitter could model the sky background and a second Emitter could model one or more stars of interest. Another Emitter could model an intervening galaxy. Likewise any number of Attenuators or Visualizers could be inserted in the pipeline.

Models can be ordered and applied only within the constraints of the real observatory (most observatories can't Model one instrument on top of another without breaking something) so we will restrict Models as appropriate. Note: we should also support an experimentation mode where there are no constraints.

We need both ends of the photon pipeline and at least one of the modifying Models, probably one or more Attenuators. A pipeline must have at least one Emitter on the left or front (if one is not defined then the Simulation Manager will complain when told to run). The far right must end with a Detector, usually followed by a Visualizer. That Detector will define the pixel size and pixel arrangement for the pipeline. More on this later.

Note: A point source on an Emitter Model is not necessarily lined up with a pixel in the Detector. The source may straddle more than one pixel. Anuradha gave an example of a 2x2 matrix of pixels where the photon stream (a circle) fell centered somewhat southeast of the center of the matrix. Therefore each pixel saw a differing percentage of the photon stream. The ETC must be able to calculate photon counts on a specific pixel basis, taking this percentage into account.

It is expected that in our early implementation stages, the models will be fairly simplistic. Later on, as we develop the Simulation tool we can refine the models as necessary. The implementation of these models may be through mathematical formulae. Could we define Models as energy transfer distributions over the surface of the Model? I'm envisioning where a model defines how likely a photon will make it through a portion of the Model and the likely hood or some function defining the refraction. We suspect as we look at the other simulations already out there, we will be able to trim this down and refine our modeling. How well does this concept map to the available simulation algorithms?

Should we be concerned with Absorbers, Distorters, Environment and (others?) varying over time? For simplicity, we probably should ignore this...although we might want to plan for this. The time spans we are looking at are Exposure times. Do things vary enough during that time that we need to be concerned?

Anuradha stressed the ability to have simple (e.g. Integer multiplier) formulas as well as complex (e.g. function of wavelength) formulas. These need to be defined, refined, and swapped in at runtime. Ideally the user should be able to define the formulas. Whether this can be done at runtime, implying some form of on the fly formula interpretation, or only at startup after some external compilation step, remains to be seen.

Pipeline Splitting

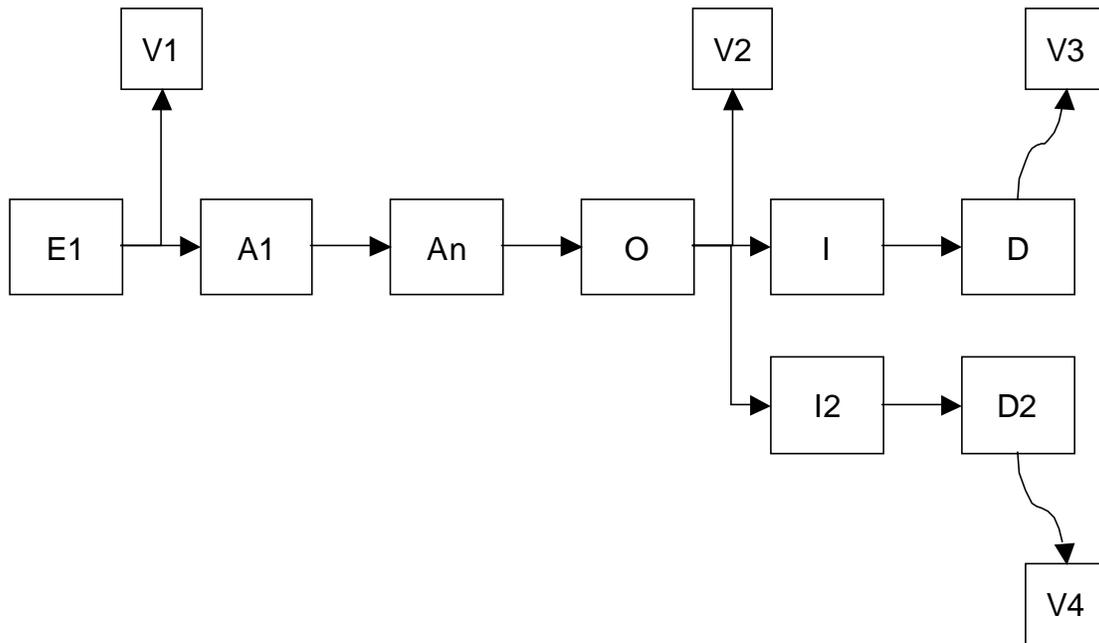


Figure 2: Pipeline Splitting

A pipeline may be split into a parallel path at any point along the main pipeline. The purpose is to provide a mechanism for the user to compare two or more simulation configurations with minimal duplication of processing. In the above example, the pipeline has been split into two parallel pipelines after the Observatory model. All simulation processing up to the Observatory (left to right) proceeds along the main pipeline. After the Observatory processing is complete the Pipeline Manager passes the results to Visualization (V2), and the two instruments I and I2. From there, further processing is divided along two separate sub-pipelines. The sub-pipelines will process in two independent Threads. The advantage here over two entirely separate pipelines is the processing from E1 to O only needs to be done once, possible a huge savings in processing requirements.

One point to make clear: once a pipeline splits, the divergent sub-pipelines never merge or come together at a later point.

As of this writing, we have chosen to defer pipeline splitting due to implementation resource constraints and the significant technical challenges to implement the splits.

Aggregation

The Simulation Pipeline will support the aggregation/deaggregation of models. The purpose of aggregation is to allow the user to group models together and replace the grouping with a single icon, thereby reducing the visual clutter of the pipeline display. In addition, the user can manipulate the aggregation as a unit. Moving it or reordering it within the pipeline as desired.

There are a few questions we need to address about aggregation:

- Can we aggregate non-contiguous models?
- Do models have to be the same type? Can you aggregate Visualizations?
- How does the user adjust the individual model parameters? How about a dialog appears containing tabbed panes? Each pane corresponds to one of the models in the aggregation.

- Can we have temporary grouping (using rubberbanding or other suitable multiple selection mechanism)?

Visualizations

There are two main types visualizations we will be simulating, imaging and spectroscopy. There are lots of other possible visualizations and we will add them as we have time, but for now we will concentrate on these two.

All Visualizations are static. There will be no attempt to “animate” data as it passes through the pipeline. Actually, the pipeline should be thought of as a sequencing mechanism, rather the being envisioned as flowing water through a pipe.

Some unanswered questions:

- Where will visualizations be displayed?
- When will visualizations be displayed?
- Does a visualization process even though it is currently not being displayed?
- If the answer to the above question is NO, what happens when the user requests an undisplayed visualization to display after the pipeline has processed passed that visualization? The context data will now be too new for that visualization.
- Can visualizations be resized? Probably on a case-by-case basis.
- Hardcopy?

Imaging

The obvious visualization is a viewable image like we see in the VTT right now. The image should present our best simulation of what the GO should expect. We should allow the image to be displayed in the VTT as if it was an image downloaded from the DSS.

The images will be photon counts, at various wavelengths, at each Pixel will get translated into display pixels through a normalization mechanism.

We probably should generate a gray scale image, like the ones we see from NED and DSS. The VTT or whatever display tool we use can convert that to false color images. Trying to generate true color images is probably way beyond the scope of what we are trying to do.

Additionally, we might want to:

- Create a Fits image.
- Create a jpeg, gif, or bmp
- Be able to overlay the simulated image over a DSS image (or any other image). This has been a suggested VTT enhancement. This could be an invaluable testing tool.
- For testing purposes, we might want to subtract one image from another (assuming they are properly aligned.) This is very useful for determining differences between two images.

Spectroscopy

The user selects a point (pixel?) and a 2D graph appears displaying a standard rainbow spectroscopic graph or something like it. A plot of photon counts vs. wavelength is a standard Spectroscopy chart visualization. Sandy probably has a better idea of what other graphics are desirable here.

Can a multi-object spectroscopy (a grid of fiber optic cables) be thought of as analogous to a grid of pixels on a detector? The difference being the pixels care about the photon energies (wavelengths) and therefore have depth. Each fiber optic cable is linked to a Detector row. The photons from the cable are prised over

the associated Detector row. There is an additional type of noise introduced with these bundles of fiber-optic cables.

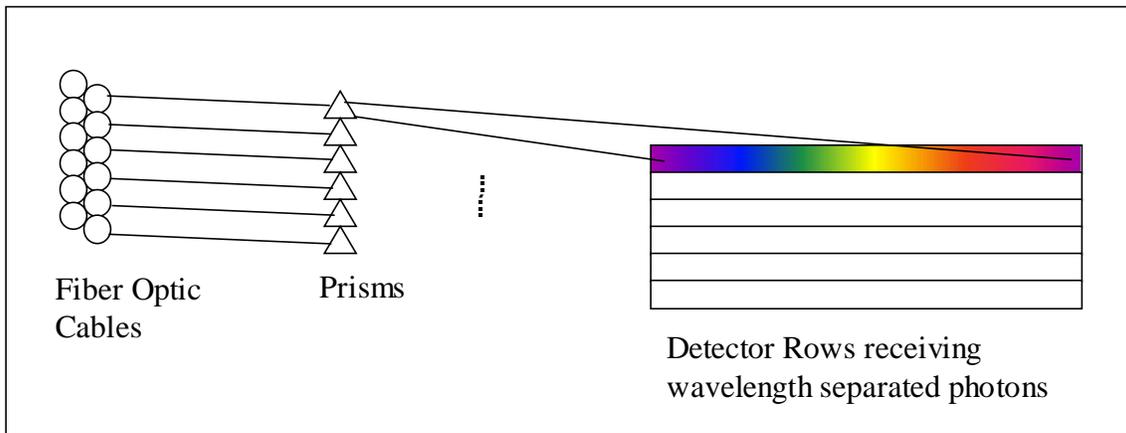


Figure 3: Multi-Object Spectroscopy

Other Visualizations

We are not attempting to define all of the Visualizations we might want to do. We just want to list some ideas for Visualizations we could try. The Visage code might help with the creation of these Visualizations.

Besides Image Simulation there may be other useful visualizations such as:

- 2D and 3D Graphs (for example: photon wave length vs. absorption factor)
- Histograms
- Scatter Plots

These Visualizations might be used to present some useful information about intermediate results during pipeline processing. The Visualizations act as READ ONLY models in the pipeline and can be attached before or after any other model.

Pipeline Processing

How does the data flow through the pipeline? We have had two ideas here.

1) Recursive pixel level processing

- A detector is asked to render itself.
- For each pixel, the detector asks the previous model for a photon count/wavelength array for that pixel. (Spatial coordinates and pixel size are known)
- The previous model asks its previous model for the same thing.
- And so on until an emitter is reached.
- Emitter returns the starting photon counts/wavelength array to the caller.
- The calling model either adds/multiplies or subtracts from the array and returns this to its caller.
- And so on until we return to the detector.
- The detector uses some algorithm to translate photon count into a pixel value.
- And so on for the next pixel.

Advantages

- Very light on the amount of data maintained at one time.

- Small amount of data passed around.
- Does not require a Pipeline manager class

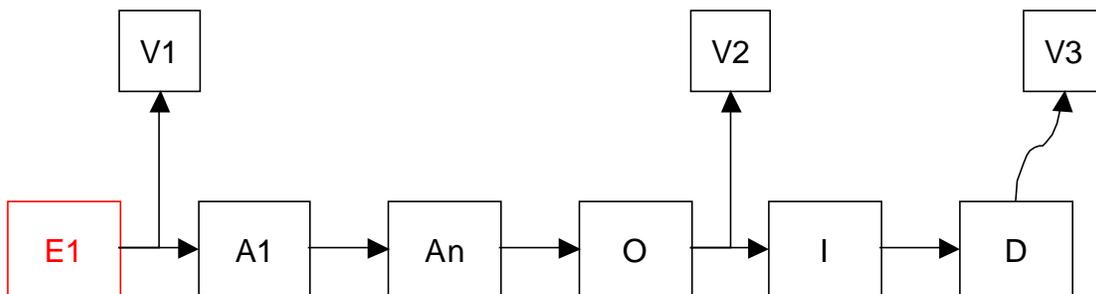
Disadvantages

- Very CPU intensive. Lots of recursive calls.
- Intermediate Visualizations (virtual detectors) will also be told to render. The processing for those visualizations will duplicate the processing done for the detector at the end of the pipeline.
- Models need to know their pipeline Parent and have to support changes in the pipeline ordering.

2) Left-to-Right full results processing

- A Simulation Pipeline Manager controls all sequencing of pipeline processing.
- Processing starts with an all black context data set. The data set represents the entire current context of the pipeline processing so far.
- The Simulation Pipeline Manager object passes the context to the left most model.
- The model applies its attributes to the Context, typically changing it by adding/subtracting/redistributing photons over the wavelength array for the entire “scene” as necessary. (Note: it is possible that we just produce a series of algorithms that can be used to compute the desired result)
- If a model is a visualizer, then the visualizer displays the data as appropriate and does not change the Context.
- Once the model is complete, the Context will have been updated and the Simulation Pipeline Manager calls the next model to the right.
- Processing stops at the end of the pipeline (often an Imaging Visualizer just after the Detector object).

An example of the processing flow is diagrammed. The red lines and boxes show the sequence of processing for the first three models. The context is passed from one box to the next, usually by reference for low overhead. However, a given “box” may pass the context information to somewhere external if necessary. That may result in copying the context, a potentially expensive thing to do. There may be ways to minimize the costs.



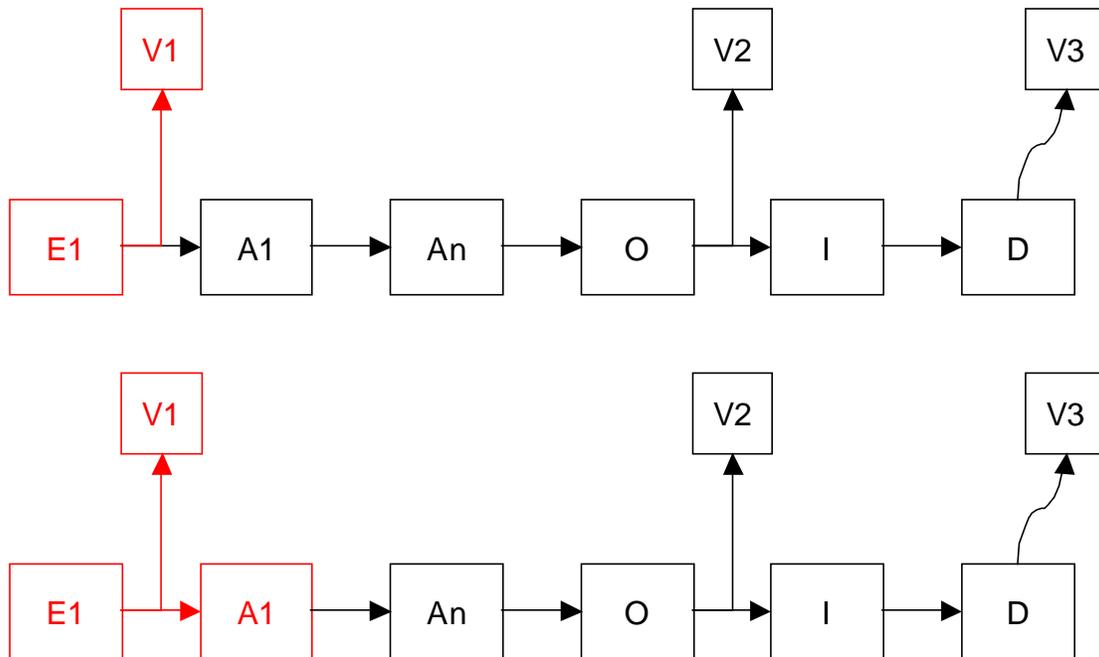


Figure 4: Example Pipeline Processing

Advantages

- No duplicate Processing for additional visualizers.
- More effectively simulates the real world.
- Essentially a Batch process.
- Models do not need to know anything about the pipeline ordering.

Disadvantages

- Context data may be very large. A 4K x 4K matrix of photon counts is at least 16MB. The depth will likely be defined as an array of counts for a range of wavelengths, effectively multiplying the byte count by 4 for each wavelength entry. There may be ways to reduce the Context data. Note: the data will usually be passed by reference eliminate any large scale copying. Some models may internally copy the data (say for use in an RMI call).

Currently we think number two (2) is the best choice. A potential mechanism to make choice two a little less memory intensive might be to logically chop the matrix into smaller chunks, say with a maximum of 1Kx1K size. In the case of a 4K x 4K simulation, the Simulation Pipeline would chop that into four pieces then send each piece through the pipeline for processing. The problem is, the “chunks” are not necessarily separate. For example, the processing may need to know about bright objects in the chunk next to this chunk so that saturation and bleeding can be properly simulated.

It seems like no matter which approach we choose, at some point along the pipeline, probably at the final Detector, we will need to reconstruct the full matrix. This implies we are going to require the memory anyhow, so why not just allocate it in the beginning and pass it from model to model? It should make the model processing much easier.

Simulation Data

The Simulation Data is the context data passed from one model to the next. It is this data that travels through the pipeline and gets modified as it passes through the models. Most models change the data in some way while other models only observe the data (visualizations).

The data can be represented as a Data Cube, see Figure 5 below. Let us consider any simulation (imaging or spectroscopy). The simulation's data is a 3 dimensional object with two (X and Y) spatial dimensions (RA and Dec) and one energy (Z) dimension. In Imaging the Z direction is projected onto the XY plane, because the filter is not differentiating between the various energies of the photons while in spectroscopy the spatial plane is projected onto a line or point and then the problem is worked out in the two dimensions of the spatial line/point and energy. The issue arises when we start thinking of the new detectors where we have multi-object spectroscopy. In this case although each object is as discussed earlier we have hundreds of objects and computationally it becomes a problem because each object can have its own set of target models.

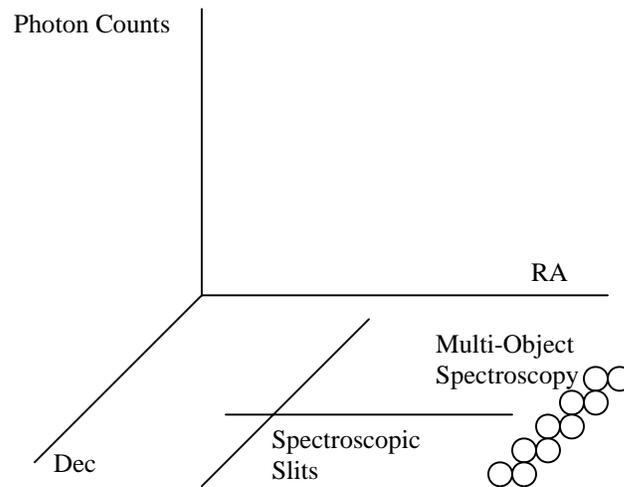


Figure 5: Data Cube

The photon counts are more than just a single value at each point, instead to support spectroscopy, there is an array of counts where each value is associated with a wavelength range. In addition, each count in the array is more than just a single value. We need to keep track of the number of “Good” photons and how many photons are due to noise. We also need to keep track of the type or cause of the noise. So there may be a number of noise values maintained for each point Noise can be a variety of noises such as:

1. Detector Noise = Read Noise (can also have thermal noise ...Is that dark current?)
2. Sky Noise
3. Photon Noise (\sqrt{N} is assumed to be noise) (do we need to store this or just calculate when needed?) Does this general term imply the average total for all noise sources?

As the Simulation Data passes through a model (other than the read-only Visualizations) they will add and/or subtract or redistribute Good and Bad photons as appropriate.

The Data cube will contain the bulk of the data. However, we need to have provisions so that models can append additional information to the simulation data. For example in the interest of performance, it may be helpful for a model to append some intermediate results to the simulation data. The downstream models

might be able to make use of the data without having to recompute them. We will design the data structures to be very flexible for this purpose.

Timing

We just wanted to point out that the simulation processes from the beginning to the end as fast as possible. We are not attempting to do any simulation of real-time processing. A 400 second exposure will take however long it takes to render the final results. We will not show intermediate results at, say, 100, 200 or 300 seconds.

How do you use the Simulation Facility in the SEA?

Within the SEA, you access the Simulation facility from the proposal tree (or whatever we replace it with). There will be a “Simulate” choice under each Exposure. Selecting the “Simulate” choice brings up the SimulationModule in the module area of the SEA display (optionally, it can be brought up in a separate window).

The SimulationModule will have a single PipelineGUI object with a single PipelineManager object whose task is to manage the pipeline simulation, of course. If this is the first time the user has displayed the Simulation facility for this Exposure, the PipelineManager will initialize the simulation’s models to correspond to the current information in the Exposure. If the simulation has already been initialized for this Exposure then what? We think we should provide a dialog box asking the user if he wants the Simulation updated with the current information.

The PipelineGUI will present a graphical representation of the currently defined Simulation Pipeline. It should look similar to the drawings above. (Maybe rather than boxes, some cool more representative icons would be better).

There are probably a large variety of manipulations we might provide to the user. Some are:

- Manipulations for assembling the Pipeline
 - Create a new Simulation Pipeline. It has been decided that providing default models is a bad idea. We want the user to explicitly identify all the pieces of the Simulation Pipeline so there is no chance of misunderstanding.
 - Add new Model
 - Delete Model (can not delete first Emitter)
 - Select one or more Models
 - Cut/Copy/Paste
 - Duplicate a pipeline (becomes a new, now unrelated pipeline)
 - Attach Visualizer
 - Remove Visualizer
 - Delete the Pipeline

Predefined Models

We don’t want the user to have to define every desired model from scratch. We will provide a variety of pre-built models and visualizations. Maybe we could offer starting simulations. Some examples are:

- Point Source - you specify magnitude and distance
 - Spiral Galaxy - you specify angle and distance
 - Name of a cataloged object (fits image) - will probably require X-Extractor support to convert a fits image into some useful form of morphology.
 - Models for HST and its instruments and detectors.
- Manipulations for tweaking the Pipeline
 - Open Model parameters (popup windows). Each model will have an associated parameter window available.
 - Rearrange Models
 - Manipulations for running the pipeline
 - Single Step through each Model one at a time.
 - Run through entire pipeline. (Each box should light up when being processed)
 - Stop
 - Start from the beginning

All manipulations may have certain constraints defined for them. Some will be our imposed constraints, some will be Observatory/Instrument/Detector imposed.

Constraints:

- At least one Emitter must be first in the pipeline. Other Emitters can be placed elsewhere but there must be one at the beginning.
- Emitters can be stacked on top of each other. This happens in the real universe (point source stars on top of a galaxy on top of the sky background).
- A Detector Model will always be arranged at the end of the pipeline, followed by one or more visualizations.

Okay, so these are the things we want to do, so how does the user do them?

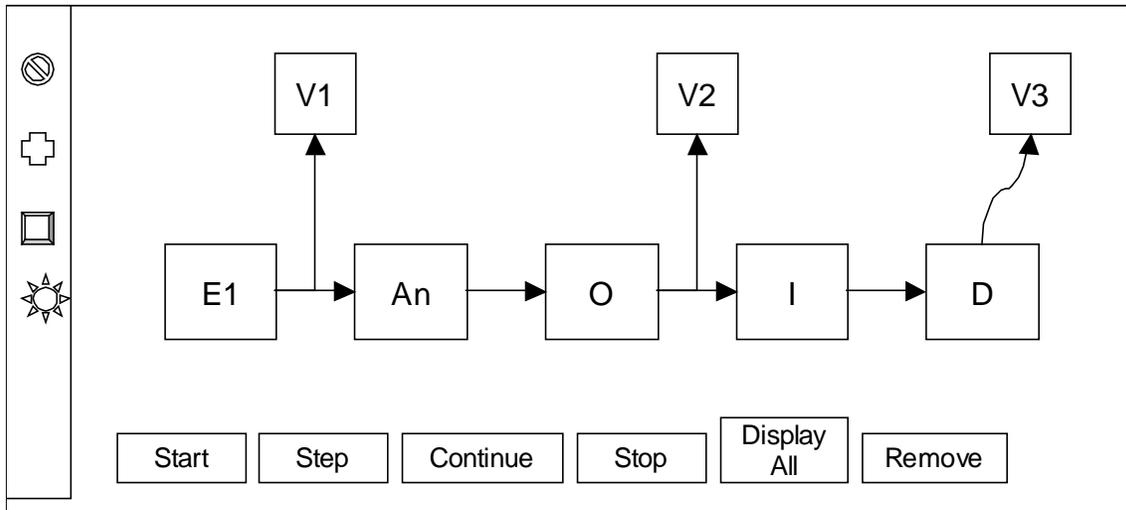


Figure 6: Simulation Module

The Simulation Module will look similar to Figure 6 above. Hopefully it will have expertly drawn icons (as opposed to the meaningless icons I've shown) to represent useful functions. Icons for the commands at the bottom could also be useful.

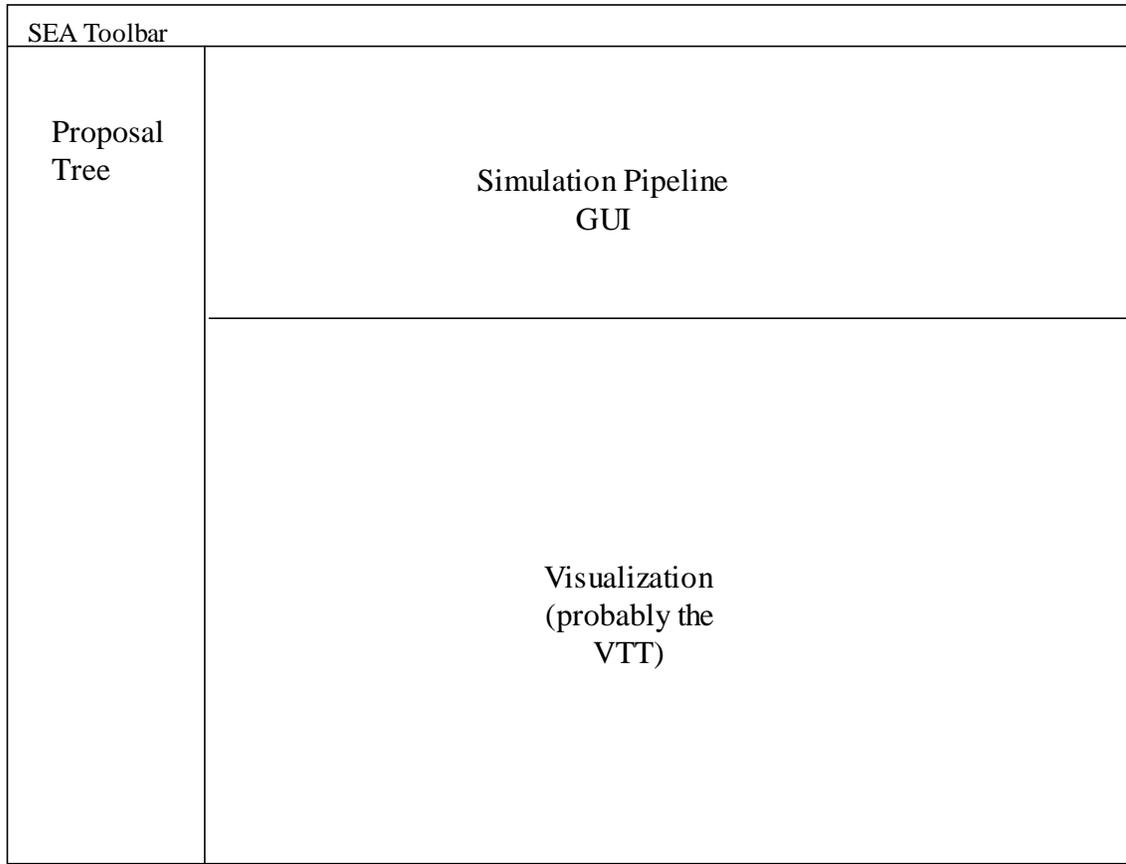


Figure 7: SimulationModule General Layout

The central area and focus of the SimulationModule is representation of the pipeline itself. The modules will be indicated by boxes or probably better, with well-labeled icons. Directional arrows indicating the directional flow of context data along the pipeline will connect the models.

Most models will arrange themselves horizontally along the pipeline. Visualizations are an exception. They will be arranged perpendicular to the pipeline, above and sometimes below the horizontal model line. This helps to emphasize the Read Only nature of Visualizers. They do not modify the context data in any way. The visualizers will go above the pipeline first. If a second visualizer is added to the same location, it will be placed below the pipeline. Note: someone may want to have multiple visualizers displaying different information about the same data.

If more visualizers are added to the same location how will they be positioned? Possibly next to the lower visualizer. This needs thought.

No changes to the pipeline can occur while the pipeline is processing a simulation. Either the user must stop the processing (Stop button) or we could automatically stop it (with an appropriate dialog) when the user requests an action affecting the pipeline.

Adding Models

The icons in the palette along the left side represent the various kinds of simulation models that can be inserted into the pipeline. For Example, to insert an Emitter, select the Star/Sun icon and drag it to the desired location on the pipeline. As the user drags, the corresponding pipeline section (the right facing

arrow) will highlight. If the user releases the button, the Emitter will be inserted at that location. Note: changing the pipeline is not allowed if it is currently processing (running) a simulation.

The models just added to the pipeline will be an undefined model of the appropriate type. A window will appear to allow the user to specify the specific model parameters. (See Specifying and Changing Models below.)

Removing Models

One or more models can be selected at once using either the common rubber band technique or by holding down the shift key and single clicking on each model's box. Once selected, the models can be removed by selecting the Remove button at the bottom of the Simulation Module display.

We MUST provide an Undo for this.

Specifying and Changing Models

Each model will likely have a variety of parameter settings. The user will access the settings by either double-clicking the model's box or by single selecting the box then selecting "Open" (not sure the best placement of this). On a PC, a Right-Click would have the same meaning. An editing module specific to that type of model will be displayed in a separate window. In many cases, these module editors are the same editors used in other places in SEA. The ETC has a number of possible editors for us to start with.

We suggest OK, Apply and Cancel buttons on these editor module windows.

A key parameter is the specific subclass of model. For example, there will likely be a variety of Emitters. Some may be predefined and some may be definable by the user. The user will need some way to specify which subclass to use in the simulation. Predefined types could be listed in a variety of ways. One choice in the list could be "Custom". When Custom is chosen, the user could be presented with another window for defining a new subclass of the appropriate type. For example, a Custom Emitter might be defined by presenting the user with a 2D matrix, representing the Detector mapped onto the Emitter. The user could select "cells" and define, emissive photon counts for those cells. Maybe some graphical approach rather than setting counts might be better. In any case, once defined, the new subclass (with a name specified) will be added to the list for future use. Note: the description of this new instance will need to be stored in the proposal along with the rest of the simulation information. Optionally, we should provide a facility to save these instance descriptions externally so they can be imported into other SEA proposals. XML may be an excellent technology to support this.

Another of the key parameters to control is the specification of transformation function(s) for the Attenuator models. It is unclear how best to do this. We will have to solve this during design and implementation. It might be useful if the user could select from a list of functions for the one closest to matching some criteria or from a selections of like model functions such as various "Dust Laws".

Rearranging Models

The user can select a model and drag it past/before another model to change the order. This is similar to the Orbit planner. If user drags to a module rather than one of the arrow lines, a before indicator and an after indicator will temporarily display so the user can drag to it. This gesture will tell the system whether to insert the new model before or after the receiving model. The pipeline will automatically reconfigure to accommodate the repositioning.

Other functions on Models

We would like to support the following functions:

- Cut/Copy/Paste models
- Duplicate models - replicates the selected models and inserts them in the same arrangement to the right of the right most selected object.
- Duplicate a pipeline - creates a new copy of the pipeline, as it is currently defined in a separate module window. There should be a corresponding entry added to the tree under Exposure to represent the new pipeline. We should probably prompt the user whether to do this duplication. If we have memory issues with one simulation, more than one is going to give us heartburn.
- SPLIT a pipeline - user selects one of the horizontal arrows then selects SPLIT. The pipeline to the right of the arrow will be replicated and placed below their counterparts on the current pipeline. The arrow line will now be drawn to show the new pipeline has a common input with the original pipeline section.
If the pipeline to the right of the selected arrow is already split, what do we do? We could replicate everything to the right, or just replicate the original pipeline? What else?

Simulation Controls

At the bottom of the Simulation Module, see Figure 6 above, are a set of controls. The controls have the following meanings:

- **Start** - Begins the simulation process from the first model on the left in the pipeline. As processing continues, the boxes will indicate where we are in the simulation process. I'm expecting some of these simulation transformations to take a long time. Processing continues from model to model until the very end. As each Visualizer is given the current context set, the Visualizer's display will appear in a separate window. If this is a repeated run, the Visualizer continues to use its current window.
- **Step** - Each time this control is selected, the simulation passes the context to the next model then pauses when that model has finished. If the simulation is currently running (via a Start or Continue) selecting this control will cause the Simulation to Pause after the current model has finished.
- **Continue** - Behaves like the Start control except the processing continues after the last model processed through the Step control.
- **Stop** - Immediately halts all processing. You can only start from the beginning, using the Start control, from this point.
- **Display All** - My thought was to request display of all Visualizers. It is unclear how useful this is.
- **Remove** - This control is used to remove the currently selected models from the pipeline. There must be a better place to put this. Maybe a better idea is to use a popup-menu when the user, say right-clicks on a model's icon. The menu could have Open, Cut, Copy, Remove (delete) for choices. The Open could request display of the model's parameter window. Sounds good.

During processing and especially when all processing has completed, there will be a display for current summary about the processing. This could be presented either within the Simulation Module or in a separate window. The contents of the Summary are TBD.

Implementation: A Difficult Balancing Act

Computation Load

In short we expect long computations for many of the models. Some simulation models, depending on fidelity, could be very complex and highly computation intensive. We need to provide mechanisms to support a “reasonable” amount of fidelity while not overly taxing the processor.

One idea to lower the computation load is to reduce the simulation resolution. The Simulation Manager can provide a parameter for the user to control the overall simulation resolution. The Simulation Manager could request each model to compute the approximate time they each may require to process at the specified resolution. The user can determine can then control the balance of time with detail.

Another approach might be to calculate only those values that are changed as a user explores the parameter space. We are not sure we can lump them upfront, because depending on what parameters are changed one may move from one regime of the formula to another. But conceptually we think it is good to save the values from each step of the calculation so that the system does not have to calculate them all the time.

Memory Requirements

For high resolution, we expect Huge Memory requirements. As described under Simulation Data above, there is potential for a large amount of data maintained for each pixel. In addition to Binning, described below, there may be memory compression techniques such as run-length encoding. Of course you pay a price for compression. The cost is processor time to encode and decode the data. We already may have costly processor time requirements for the model simulation processing. Compression may prove too much of an extra burden.

Compression might be most useful when a model needs to transfer the simulation data to some other location for offline processing. This might be required when passing the data through RMI or CORBA to another processor “out-there”.

Binning - A Balanced Solution

It is clear we need to minimize the data storage and processing requirements for simulation runs. One technique, borrowed from Detector technology, is called Binning. The user will have the capability to specify a reduced resolution or fidelity for the simulation. For example, say the final Detector defines a 4K by 4K arrangement of pixels. This means there are 16M pixels on that Detector. At full resolution, the simulation pipeline would need to compute and store multidimensional data for all 16M points. It is expected that the processing and memory requirements for full fidelity simulation may greatly tax the user’s system. A solution is to let the user specify a lower resolution to start with, say 256x256. In this case, each lower resolution bin would represent a 16x16 grouping of pixels. Now the simulation is only maintaining 65K data points or about 2/5 of 1% of the original required calculations. A potentially huge savings. It is expected that only a portion of the overall aperture is of real interest. The user would then select the appropriate subset of the bins to process at a higher resolution. The Simulation Pipeline would reprocess only those bins at the higher resolution. Subsequently, the user could specify another subset at another resolution and reprocess. The main point is, only areas of interest will be processed at a high resolution, potentially reducing processing and memory requirements while giving the user a useful simulation.

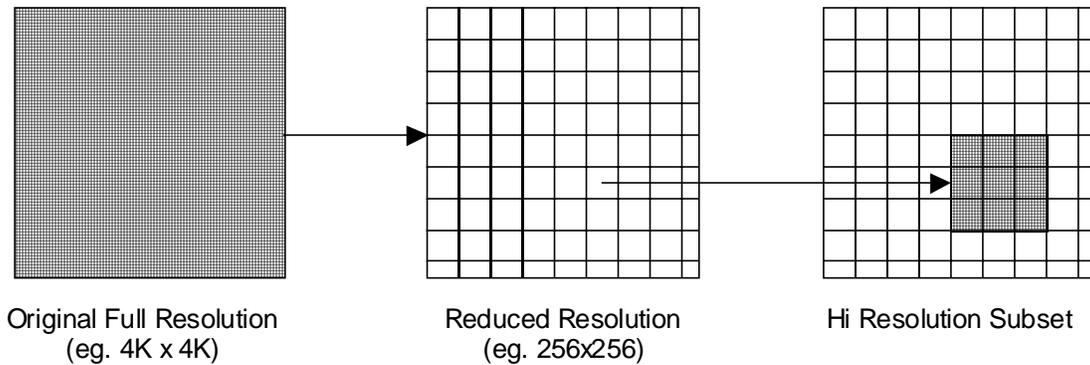


Figure 8: Binning Example

Again, the user would run the simulation at this lower resolution, then looking at one or more Visualization results, the user could decide which portions of the overall data set are the most interesting. The user would indicate these to the simulation and specify another resolution (usually higher) to use just for those areas. The simulation would run again just for those areas, at the revised resolution. Therefore the user would get the required fidelity in his area of interest, without unduly overtaxing the system running the simulation pipeline.

A final note on Binning. Bins are quadrilateral; they may not necessarily be rectangular. It all depends on how the aperture maps to the sky. The detector may not be perfectly parallel to the sky background. This results in a skewed shape of the bins. For our first cut, we may assume rectangular bins.

Evaluation

As the models are developed we will need to verify their scientific accuracy. Each model should be evaluated separately then in combination. Ideally we would like to have one or more real world examples we can compare to. The simulations need to be thoroughly validated. Where deviations occur, we need to understand why they occur and what are the limitations of the simulations.

It is expected that our first attempts will be very crude approximations of the desired results. The timeframe between now and the end of September 2000 is probably far too short to expect high precision. However, with proper early evaluation, feedback and refinement we may be able to have a reasonably faithful simulation.

One of the Simulation Data requirements is to maintain data on both “good” and “noise” photons. One technique for reducing memory requirements might be to disable all target emitters then run a simulation. This should produce results representing the noise in the system. The noise results could be saved somewhere (disk perhaps?) Then a second pass could be run with the emitters re-enabled. The results from that run could be compared to the noise run to determine the “good” data. We would make two passes through the simulation. One pass producing noise and one pass producing the final results. The noise would then be subtracted if necessary to determine just the “good” pixels.

Actually a better way would be for one run to be a normal run and the other run to turn off all “noise” producers resulting in a noiseless set of data. The difference would be the noise.

Final Comments

What is key here is to lay the proper groundwork for the pipeline. It is very modular concept and we can implement the simple models first. They can become increasingly sophisticated as we progress.

We expect a large portion of the code to be in common the between imaging and spectroscopy simulations. We believe the only real differences will be the Visualizations at the end

We want to be able to define these various Model types easily. We want to make it easy to add new Observatories along with the many instrument/filters/detectors they contain. My initial feeling is we should use a XML based approach for defining these models and their characteristics. Much like IRC has done with Command-Control and Astronomical Instruments.

This document strayed into design concepts and issues. That's probably okay. None of the design ideas is locked in concrete, except possibly the Pipeline concept. The whole "Pipeline" view and concept have a strong software engineer feel to them. Will the GOs feel comfortable with the concept?

There are probably lots of other issues we haven't discussed here, such as inserting a Detector in to the pipeline that gobbles the photons, then placing another Detector somewhere downstream of it. The second Detector will see nothing. Should we prevent the user from doing this?

We have not addressed the science issues of implementing the simulations. Those will need to be raised and answered as we continue refining the design.

In the Future

In future releases we may explore the following:

- Add in more cosmology to our models. Improve the fidelity of the models.
- Support Multi-Object Spectroscopy.
- Add more canned Observatories, Instruments, and Detectors.
- Provide mechanisms for the user to specify transformation formulas and to have more control over the models.
- Provide a way for the user to define experimental observatories/instruments/filters/detectors? They could be useful for exploration and experimentation but not useful for a proposal. (How can you propose for something that doesn't exist) Possibly use XML to define these models. Note: this is more than just defining the general characteristics of the models. It must include something to define the behavior of the model with respect to the photons.
- Improved Pipeline branching. Possibly multiple pipelines with multiple branching.
- More processing and internal storage optimizations.
- Incorporation of other sites simulations. In particular we need investigate the HST Simulation software, along with IRAF and JOIN.
- Tighter integration of the Simulation Facility with the rest of the SEA.

The Current Design

Simulation Class Design

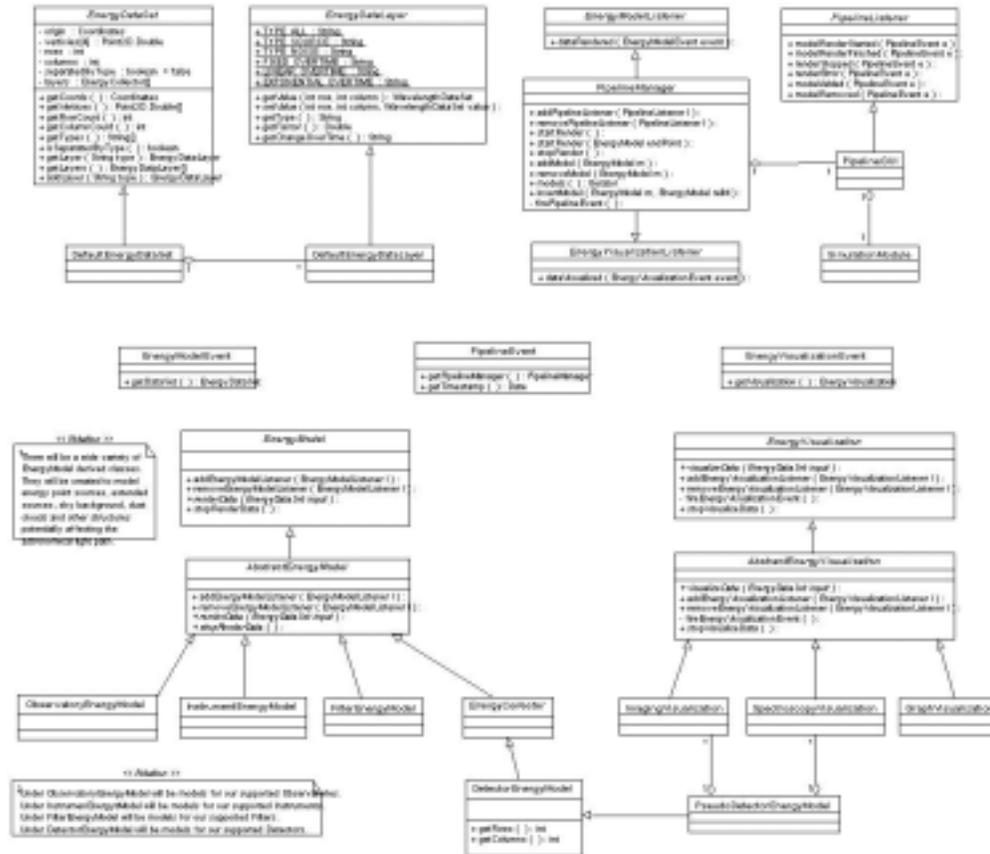


Figure 9: Simulation Class Design

Please refer to Figure 9 above for the following discussion. The classes indicated above form the primary classes for the SEA Simulation Facility. Many of them will be sub-classed for specific implementations. The classes toward the right side of the figure represent the Simulation Pipeline, its GUI and Visualizations.

The classes on the upper-left of the design represent the simulation data. The classes on the upper-right represent the Simulation Pipeline control and user interface. The classes on the lower-left represent the simulation models that actually modify the data to reflect the simulation. The classes on the lower right represent various types of visualizations of the simulation data. Finally, the classes in the middle section represent the various Event classes used to signal and transport data among the other simulation classes.

The Classes

Simulation Data

EnergyDataSet

The EnergyDataSet encapsulates the Simulation Pipeline context data passed from EnergyModel to EnergyModel along the Simulation Pipeline. As the pipeline processes through the EnergyModels, the EnergyDataLayers in the EnergyDataSet are modified by each EnergyModel to reflect its effect on the current state of the pipeline data.

There will be a single EnergyDataSet instance for a Simulation Pipeline context. The class maintains the overall context specific information including the spatial coordinates (if any) of the data set, the resolution of the data (rows, columns) and the arrangement of pixels or bins. The EnergyDataSet maintains a list of the EnergyDataLayer objects representing the actual computed context data for the simulation.

EnergyDataLayer

The EnergyDataLayer contains the bulk of the simulation data. The data within a layer is arranged in a matrix of WavelengthDataSet objects. The number of rows and columns in the matrix are determined by the parent EnergyDataSet object which in turn is determined by the final DetectorEnergyModel in the Simulation Pipeline. The rows and columns of pixels of the DetectorEnergyModel define the maximum row and column values in the EnergyDataSet. Binning is implemented by specifying values less than these maximums.

Each WavelengthDataSet contains one or more values corresponding to the rate of photons/sec for each of one or more Wavelengths. The values are rates over time rather than specific photon counts so that the Exposure time does not have to be taken into account until the very end of the pipeline at the DetectorEnergyModel. The getChangeOverTime method can be used to indicate how to apply the values in the layer over the Exposure time. The default is LINEAR_OVERTIME, where the rates are assumed to be rates per second. Therefore, a value over the entire Exposure time can be calculated by multiplying the rate per second times the Exposure time in seconds. The getFactor method defines the slope of the line. The default is a factor of 1.

Other getChangeOverTime values are FIXED_OVERTIME and EXPONENTIAL_OVERTIME. The first indicates the rates are fixed and are not cumulative over the Exposure time duration, for example Read Noise is fixed regardless of Exposure time. The EXPONENTIAL_OVERTIME indicates the rate is a starting value and the factor (obtained through the getFactor method) represent an exponential change over time (y^x), where x is the exponent.

An EnergyDataLayer may represent either Source photon rates (TYPE_SOURCE) or Noise photon rates (TYPE_NOISE) or the combination of both (TYPE_ALL). When type=TYPE_ALL, then there will only be one EnergyDataLayer in the EnergyDataSet. Otherwise there can be one EnergyDataLayer for Source photon rates and one for Noise photon rates.

There is a big challenge here to handle a potential huge data set. Real world astronomical detectors are up to 4K x 4K pixels (could be larger in the future?) or 16 mega-pixels. An EnergyDataLayer representing the full resolution will have 16 mega-WavelengthDataSets. Memory requirements become very large. Additionally, if we have separate Source and Noise EnergyDataLayers we require at least twice the storage. The binning concept will help to reduce storage requirements at the expense of some simulation fidelity.

DefaultEnergyDataSet

A DefaultEnergyDataSet is a default EnergyDataSet created when a new pipeline is told to render. The PipelineManager instantiates the DefaultEnergyDataSet and passes it to the first EnergyModel in the Simulation Pipeline.

DefaultEnergyDataLayer

The DefaultEnergyDataSet creates a DefaultEnergyDataLayer at the start of a new pipeline. The DefaultEnergyDataLayer will have all of its values set to 0.

EnergyModel

The EnergyModel defines the interface for all model objects affecting the data in the pipeline EnergyDataSet. The classes implementing this interface are responsible for providing the main simulation logic for the SEA Simulation Facility. Each EnergyModel implementing class represents some photon altering aspect of the photon pipeline. It is within these classes that the primary astronomical science modeling will be implemented. As such, these classes will be the most technically challenging to create and will require the bulk of our implementation schedule.

The EnergyModel interface defines support for EnergyModelListener objects to be notified when the EnergyModel has completed its rendering. During Simulation Pipeline processing, the PipelineManager tells the next EnergyModel to renderData, passing in the current EnergyDataSet. When the EnergyModel is finished with the EnergyDataSet it call the fireEnergyModelEvent method to notify its listeners (the PipelineManager here) and passes back the EnergyDataSet the EnergyModelEvent. **NOTE: the EnergyModel must never modify the EnergyDataSet or any of its EnergyDataLayers in any way after it has notified any of its EnergyModelListeners.**

EnergyModel objects may also have a mechanism to allow the user to specify certain parameters to the modeling process. The specific parameters are necessarily model dependent and the manner in which they are specified by the user are also left up to the implementation.

EnergyModelListener

A class uses the EnergyModelListener interface when it wants to be notified when significant events occur in an EnergyModel. The PipelineManager class is notified through this interface to determine when a specific EnergyModel has completed its EnergyDataSet rendering.

EnergyModelEvent

The EnergyModelEvent is the event object passed to a class implementing the EnergyModelListener interface whenever a notable event occurs in an EnergyModel that needs to be indicated to outside classes.

AbstractEnergyModel

The AbstractEnergyModel class provides a base class implementation of the EnergyModel interface. This class includes common implementations of EnergyModel methods. In particular, the AbstractEnergyModel contains support for EnergyModelListener objects to be notified when the EnergyModel has completed its rendering processing.

ObservatoryEnergyModel

The ObservatoryEnergyModel class extends the EnergyModel class. This class acts as a common class for models of Astronomical Observatories. Derived from this class will be other observatory specific classes such as HSTObservatoryEnergyModel.

InstrumentEnergyModel

The InstrumentEnergyModel class extends the EnergyModel class. This class acts as a common class for models of Astronomical Instruments. Other instrument specific classes such as HSTACSInstrumentEnergyModel will be derived from this class.

FilterEnergyModel

The FilterEnergyModel class extends the EnergyModel class. This class acts as a common class for models of filters on Astronomical Instruments.

EnergyCollector

The EnergyCollector serves as a base class for EnergyModels, which result in the total absorption of the pipeline energy data.

DetectorEnergyModel

The DetectorEnergyModel is the base class for Detector objects in the Simulation Pipeline. Models for real (or experimental) detectors will be derived from this class. The DetectorEnergyModel class will model the real world behavior of the detectors including such effects as saturation and bleeding and the contribution of read noise.

The Detector pixel arrangement defines the maximum resolution of the Simulation Pipeline. For example, if the pixels on the real detector were arranged in a 1K x 1K grid then the maximum resolution of a simulation EnergyDataSet for this detector would be limited to 1K x 1K bins or cells.

The typical primary role of the `DetectorEnergyModel` is to model the photon collecting behavior of the detector and the conversion of the photons into pixel values. The pixel values will usually be displayed by a following `ImagingVisualization`. The conversion process is also the first location where the Exposure time will be used in the Simulation Pipeline. The `EnergyDataSet` `EnergyDataLayers` contain values representing rates of photons/second. The Exposure Time is used to calculate the expected pixel value from the energy rates. See the discussion above for the `EnergyDataLayer` for a more complete discussion of these values.

Related to the `DetectorEnergyModel`, is the `PseudoDetectorModel` class, which acts as a temporary detector for use by certain `EnergyVisualizations` not located at the end of the Simulation pipeline. These `EnergyVisualizations` need many of the functions of the `DetectorEnergyModel` such as the conversion to pixel values, but we do not want to modify `EnergyDataSet` as a Detector would. See the discussion about the `PseudoDetectorEnergyModel` below.

PseudoDetectorEnergyModel

The `PseudoDetectorEnergyModel` is used in conjunction with an `ImagingVisualization` and `SpectroscopyVisualization` classes to help the Visualization render pixel data. The `PseudoDetectorEnergyModel` implements a subset of the modeling functionality that the corresponding `DetectorEnergyModel` implements. The purpose of this class is to provide a quick, but reasonably faithful model of what the Detector would see if it was located at this location in the Simulation Pipeline. The pixel results are passed on to the `ImagingVisualization` and not passed on further. The `EnergyDataSet` is not modified in any way.

SimulationModule

The `SimulationModule` is the SEA Module for simulation. Figure 7 on page 18, shows a stylized view of what the module will look like. As an SEA Module, the `SimulationModule` can be displayed wherever an SEA Module can be displayed.

The `SimulationModule` will contain one `PipelineGUI` object to control and to present a representation of the Simulation Pipeline to the user. The lower part of the module will be dedicated to displaying an `EnergyVisualization` or in some case, module parameter displays.

PipelineGUI

The `PipelineGUI` is the graphical user interface for controlling and displaying the SEA Simulation Pipeline.

The `PipelineGUI` displays the Simulation Pipeline in a manner similar to what is shown in Figure 6 on page 17. The `PipelineGUI` is responsible for providing the mechanism to:

- Assemble a pipeline
- Control pipeline processing including starting and stopping the render processing.
- Monitors the pipeline processing through the `PipelineListener` interface.
- Save the pipeline structure for later reuse during a future session.

The `PipelineGUI` contains one `PipelineManager` class and implements the `PipelineListener` interface to listen for pipeline processing changes.

PipelineManager

The `PipelineManager` class represents the Simulation Pipeline and controls the rendering processing. It is the `PipelineManager`'s responsibility to control the sequencing of the rendering and visualization. The `PipelineManager` maintains the ordered list of `EnergyModels` in the Simulation Pipeline. It also maintains a list of the various `EnergyVisualizations` and their relationship to the `EnergyModels` in the Pipeline.

The `PipelineManager` implements both the `EnergyModelListener` and `EnergyVisualizationListener` interfaces to determine when the various models and visualizations are done with the `EnergyDataSet`.

The PipelineGUI communicates with its PipelineManager instance to build, run and control the Simulation Pipeline. The PipelineGUI adds, inserts and removes EnergyModels by commanding the PipelineManager to change its EnergyModel list. The PipelineGUI also attaches EnergyVisualizations to the PipelineManager in much the same way.

PipelineListener objects register with the PipelineManager to listen for significant events during pipeline processing. It is up to the PipelineManager to notify the PipelineListener implementers when appropriate. The primary PipelineListener is the PipelineGUI. The PipelineManager calls the firePipelineEvent method to send a PipelineEvent object to all the PipelineListeners.

Once the Simulation Pipeline has been created, the PipelineGUI can command the PipelineManager to render the simulation. The sequence of steps to process render the simulation can be found in Table 1 below.

PipelineListener

The PipelineListener interface provides a mechanism to receive notifications of significant changes to the Simulation Pipeline. When a change occurs, a PipelineEvent is created and passed to all registered PipelineListener objects. The PipelineEvent details the context information about the particular pipeline event.

PipelineEvent

A PipelineEvent object is the context object passed to a PipelineListener when a significant Simulation Pipeline event occurs.

EnergyVisualization

The EnergyVisualization defines the interface for all Visualizations in the Simulation. EnergyVisualizations are used to display the current results of the Simulation Pipeline rendering processing. They can be thought of as a Read-Only view of the EnergyDataSet at the point when the EnergyVisualization has been inserted into the pipeline. Typically, an EnergyVisualization will display a window on the computer display, which will provide a (hopefully) meaningful representation of the data. Often this may be in the form of a 2D or 3D image. EnergyVisualizations can also be graphs or charts or even some hard copy output.

Each EnergyVisualization implementer maintains a list of registered EnergyVisualizationListener objects. When significant events occur, the EnergyVisualization calls the fireEnergyVisualizationEvent method to notify each of the EnergyVisualizationListeners with an EnergyVisualizationEvent.

EnergyVisualizations can only observe the current EnergyDataSet. They are forbidden from changing it.

EnergyVisualizationListener

A class wanting to be notified whenever significant events occur in an EnergyVisualization uses the EnergyVisualization interface. The PipelineManager class is notified through this interface to determine when a specific EnergyVisualization has completed its EnergyDataSet visualization.

EnergyVisualizationEvent

The EnergyVisualizationEvent is the event object passed to a class implementing the EnergyVisualizationListener whenever a notable event occurs in an EnergyVisualization.

AbstractEnergyVisualization

The EnergyVisualization class is a base class implementation of the EnergyVisualization interface. It provides common methods for use by all EnergyVisualization classes.

The AbstractEnergyVisualization class provides a common implementation of the mechanism to support EnergyVisualizationListener objects. It maintains a list of registered EnergyVisualizationListener objects. Whenever significant events occur in the EnergyVisualization, the fireEnergyVisualizationEvent method is called to notify each of the EnergyVisualizationListeners with an EnergyVisualizationEvent.

ImagingVisualization

The ImagingVisualization class is a common base class for Imaging Visualizations. ImagingVisualizations are usually pixel based and as such, need to be related to a DetectorEnergyModel for the proper pixel conversion of the EnergyDataSet. ImagingVisualizations that follow the final DetectorEnergyModel in the Simulation Pipeline already have their pixels available in the EnergyDataSet from the DetectorEnergyModel.

ImagingVisualizations located in the pipeline before the DetectorEnergyModel do not have pixel data available to them yet. To handle that problem, the PipelineManager automatically inserts a PseudoDetectorEnergyModel object between the Simulation Pipeline and the ImagingVisualization. The PseudoDetectorEnergyModel will render pixel data in much the same way the real DetectorEnergyModel would but the results only become a temporary part of the EnergyDataSet. Once the ImagingVisualization has completed its rendering, the pixel data is removed from the EnergyDataSet. It is up to the ImagingVisualization to store a copy of the pixel data if longer-term storage is required.

Spectroscopy Visualization

The SpectroscopyVisualization class serves as a base class for Spectroscopic Visualizations. The SpectroscopyVisualization may use a PseudoDetectorEnergyModel in a manner similar to the ImagingVisualization above.

GraphVisualization

The GraphVisualization class serves as a base class for Visualizations containing Graph displays such as 2D or 3D renderings of the EnergyDataSet. It is expected that graphs will just use the rate data as it is in the EnergyDataSet.

High Level Pipeline Processing

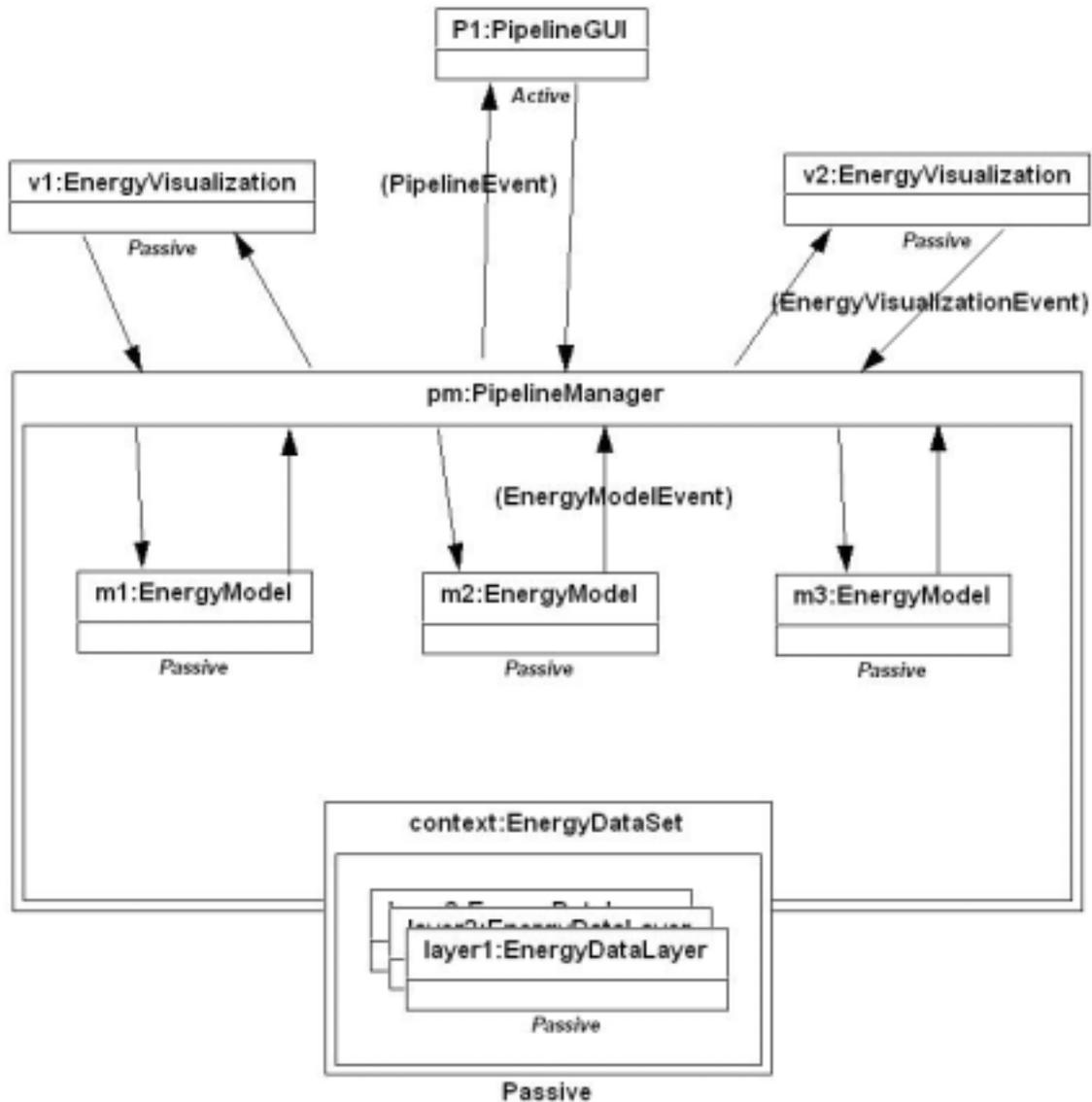


Figure 10: Simulation High Level Pipeline Processing

The following discussion refers to Figure 10 above. As you can see, the PipelineManager is the primary coordinator of the Simulation Pipeline. All classes are passive in respect to the PipelineManager class, except for the PipelineGUI class. The PipelineGUI class can command the PipelineManager to perform tasks on its behalf. The PipelineManager will also report changed back to the PipelineGUI through the PipelineListener facility.

Before a simulation run begins, the pieces of the simulation must be defined. The user interacts with the PipelineGUI to define what models to create, their parameters, and how the models are arranged. As EnergyModels and EnergyVisualizations are added to the Simulation Pipeline, the PipelineManager registers for EnergyModelEvents and EnergyVisualizationEvents. The Section below on User Interaction discusses more on this pipeline assembling interaction.

The following table lists the sequence of steps that occur when the user tells the PipelineGUI to run the simulation.

1. The user tells the PipelineManager, through the PipelineGUI, to render the Simulation.
2. If no EnergyDataSet exists, the PipelineManager builds an empty EnergyDataSet by creating and instance of DefaultEnergyDataSet. The DefaultEnergyDataSet constructor creates an instance of DefaultEnergyDataLayer.
3. The PipelineManager tells the first/next EnergyModel to render and passes in the EnergyDataSet.
4. The EnergyModel calls fireEnergyModelEvent to notify its EnergyModelListeners that rendering has completed.
5. The PipelineManager, through the EnergyModelListener interface, receives the completion notification.
6. The PipelineManager then notifies all PipelineListener objects registered for that EnergyModel that it has completed its rendering.
7. The PipelineManager tells each EnergyVisualization positioned after the current EnergyModel and the next EnergyModel, to visualize the data and passes in the current EnergyDataSet.
8. The EnergyVisualizations process the EnergyDataSet into a form they can graphically present to the user. The Visualizations can appear in separate windows or in the VTT area below the pipeline in the SEA Module window.
9. Each EnergyVisualization notifies the PipelineManager that it is finished with the EnergyDataSet by calling fireEnergyVisualizationEvent.
10. The PipelineManager ensures that all pending EnergyVisualization objects have completed their visualizations.
11. The PipelineManager sets up to process the next EnergyModel in the ordered list.
12. Loop back to step 3 and continue processing.
13. Once all EnergyModels and EnergyVisualizations have been processed the pipeline is complete.

Table 1: Pipeline Processing Steps

The processing that takes place within the EnergyModels and EnergyVisualizations are not discussed here. Each one will likely be a paper on its own due to the potential complexity when creating faithful simulation models and displays.

User Interaction

At this point we've left a lot of the GUI out of the discussion. The previous sections have described much of the internal structure of the SEA Simulation Facility. But what can the user see and do? The functionality required for the SimulationModule is discussed in the section titled "How do you use the Simulation Facility in the SEA?" on page 16.

The EnergyVisualizations may use the VTT area under the PipelineGUI for a display area. Additionally, they may appear in popup windows just like other Modules in the SEA. The actual look and feel of the EnergyVisualizations are TBD at the moment. We obviously want an ImagingVisualization to render the simulated pixels. We also want Spectroscopic displays. However, the nature of these displays will be decided later as we learn more about the simulation implementation.

Note: the EnergyVisualizations are not necessarily static. They may have controls and other parameters to change their look and feel. NASA ViSAGE graphs and displays may be a good starting point for some of the more graph oriented EnergyVisualizations. The SEA Visual Target Tuner (VTT) will likely be the basis for the ImagingVisualization because it has much of the imaging code already working. Likewise, the SEA Exposure Time Calculator (ETC) will likely form the basis for the SpectroscopyVisualizations.

Like the EnergyVisualizations, the EnergyModels will may also have a user interface to enable the user to specify processing parameters prior to a simulation run. The PipelineGUI will provide a mechanism for the user to request an EnergyModel to display its processing parameter dialog. The current thinking is the user

selects an EnergyModel icon on the PipelineGUI display then either through double-clicking or through a popup-menu, indicates that the EnergyModel should display its parameter dialog. This will likely appear as a separate window apart from the main SEA window with the PipelineGUI.

As long as the simulation is not “running”, the user may make changes to one or more EnergyModel windows. Once the Simulation is running, the dialogs may be viewed but changes will not be allowed. A possible modification of this may be that we could allow changes to EnergyModels “downstream” from the current processing position within the Simulation run. Probably a better technique would be a mechanism to first Pause the pipeline, make the downstream changes, and then proceed.

As you can see, much of the SimulationModule GUI is currently TBD. We will firm up the design here as we proceed with the Simulation Facility implementation.