# GraviT User Guide

Version 0.1

## Contents



## Overview

GraviT is a software library for the class of simulation problems where insight is derived from actors operating on scientific data, i.e., data that has physical coordinates. This data is often so large that it cannot reside in the memory of a single compute node. While GraviT is designed with many types of actors and use cases in mind, the canonical usage of GraviT is with the actors that are rays and data that are tessellated surfaces. In this case, GraviT produces ray-traced renderings.

GraviT's design focuses on three key elements: interface, scheduler, and engine. The interface element is how users interact with GraviT. The scheduler element focuses on how to bring together actors with the appropriate pieces of data to advance the calculation. The engine element performs the specified operations of the actor upon the data. This design is intentionally modular: developers can opt to extend GraviT with their own implementations of interface, scheduler, or engine, and to re-use the implementations from the other areas. In short, GraviT provides a fully working system, but also one that can be easily extended. Finally, GraviT is intended for very computationally heavy problems, so it aims to carry out calculations in the most efficient way possible while maintaining modularity and generality. This goal impacts the scheduler and engine elements in particular.

GraviT is divided into "core" infrastructure and domain-specific library. The "core" infrastructure, abbreviated GVT-Core, contains abstract types, as well as implementations that are common to domain-specific libraries, for example scheduling. The domain-specific libraries build on GVT-Core to create a functional system that is specialized to their area. The domain-specific libraries that have been discussed by the GraviT team are:

- GVT-Render, for ray-tracing geometric surfaces
- GVT-Volume, for volume rendering
- GVT-Advect, for particle advection
- GVT-Photo, for modeling photon interactions

## Engines

GraviT's engines are the modules that carry out the calculation to advance an actor using information from the appropriate data. The choice of the word "engine" conveys a connotation that this operation is computationally intensive, and this is purposeful. GraviT is aimed at problems where determining the behavior of an actor is time-intensive because (1) there are many actors, (2) the data is very large, or (3) both. Building high-quality engines takes significant development time. Fortunately, existing third-party products can fill this role in key instances. GraviT's development strategy is inclusive of these third-party products, and provides options for leveraging them. For these cases, the GraviT engine acts as an adaptor, converting requests from scheduler into instructions that the third-party product can accept. At this time, the third-party products most considered by the GraviT team are Intel's Embree/OSPRay and NVIDIA's Optix Prime/Optix. These libraries are being used for GVT-Render, but also have been discussed for GVT-Volume and GVT-Advect. Of course, a GraviT engine does not need to be an adaptor around a third-party product. Native GraviT engines are also supported, i.e., engines that take instructions from the schedules and carry them out directly. An example of a native engine is the ray tracer written specifically for GraviT, using data-parallel primitives in the EAVL framework.

## Interfaces

GraviT's interface is how external applications interact with GraviT. GraviT has a single interface, and all domain-specific applications use this interface. The interface is key-based, which means that each must specify its set of supported keys, and document that list for library users. (For example, GVT-Render will support the notion of a camera position, and its documentation must make clear that the key associated with this position is "Camera", as opposed to "CamPosition".) One motivation behind this design is the code for the interface can be written once, and re-used for each domain-specific application, modulo defining the sets of keys that are used.

Beyond the general interface, new interfaces to GraviT can added as wrappers around the main interface. These interfaces will define their own library functions, and implement the functions by translating the incoming information to the main GraviT interface. One reason to define a new, wrapper interface is to simplify the interface for users, i.e., to reduce barriers to community usage. Another reason is to fit a legacy interface, once again to reduce barriers to community usage. An example of this latter type exists in GVT-Render, which provides a GLuRay interface. The GLuRay interface meets the OpenGL 1.x standards, which is used by multiple visualization tools, and thus allows them to use GraviT with minimal overhead. The GLuRay wrapper interface then implements its functions by calling functions in the main GraviT interface.

GraviT's interface also sometimes needs to affect application behavior. This is done through callbacks. Applications can register callbacks to load or unload data with GraviT, and GraviT's schedule can issue these callbacks while it is executing.

## Data Model

GraviT's data model varies from domain-specific library to domain-specific library. The application callbacks to load data need to be aware of the data model of the domain-specific library to load it into GraviT's form. Our team has discussed issues such as "zero-copy" in situ, but this discussion has not yet led to a fixed model. As this issue is large in scope, our short term plan appears to be that we make a copy of the data in GraviT's format.

GraviT does not only accept data through load and unload callbacks. Applications can specify the data before execution starts. This is the case with GLuRay, where data is acquired incrementally through its interface, and then registered with the scheduler.

## Scheduler

The job of GraviT's scheduler is to get actor and data together so that an engine can carry out its operations. When GraviT runs in a distributed-memory parallel setting, getting the right actor and data together can have significant latency. There is a spectrum of algorithms that respond to this issue. On one extreme, actors stay on the same node throughout execution, and the needed data is imported to carry out execution. On the other extreme, data stays on the same node throughout execution, and actors are passed around between nodes.

Our goal with the GraviT schedulers is that we develop building blocks to facilitate rapid development of new schedulers. We endeavor to have schedulers that take approximately forty lines of code, and whose implementations highly resemble the pseudocode we would use to describe them in a publication.

GraviT does make assumptions about the data it operates on. Specifically, it assumes that data that is very large can be decomposed into domains, i.e., partitioning the set of all data so that each domain is spatially contiguous. Further, GraviT's schedulers assume that they can access information about: (1) the spatial extents of each domain and (2) the cost of acquiring each domain. These requirements are important for enabling schedulers to perform efficiently

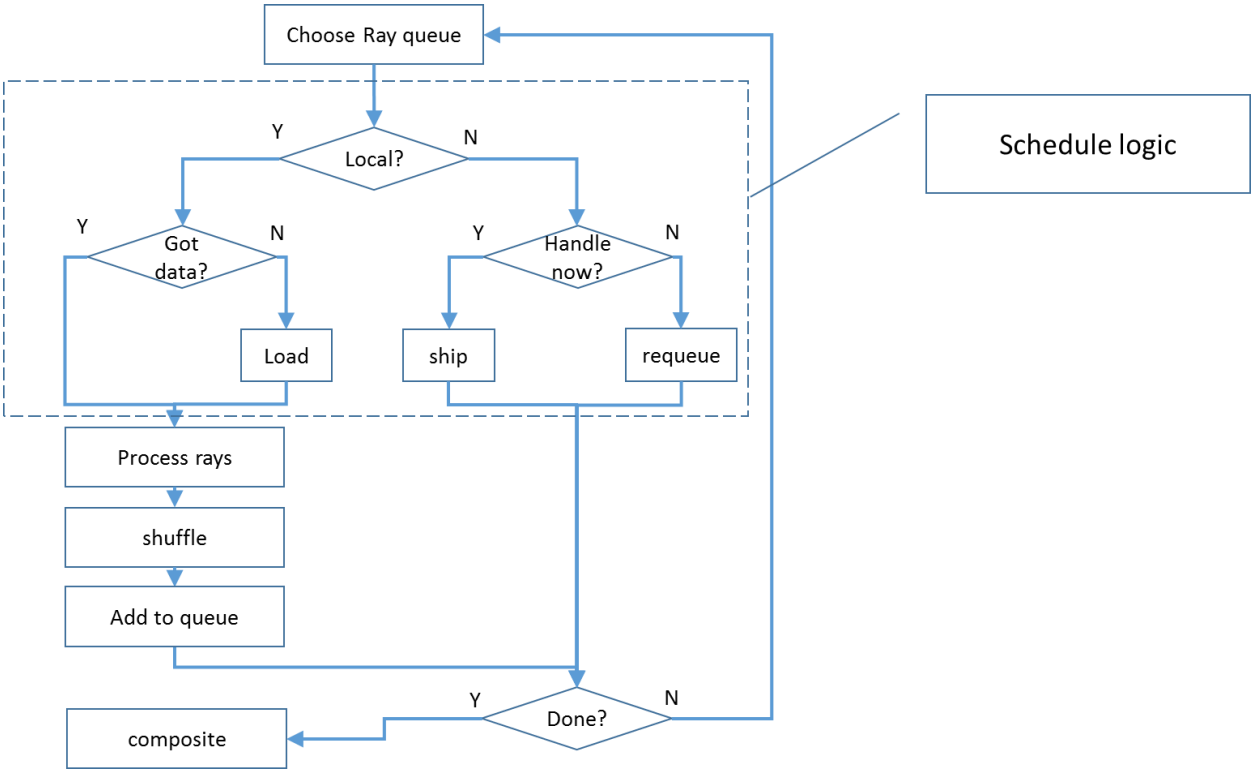


*Figure 1 Scheduler logic*

The diagram Figure 1 Scheduler logic describes what each rank does in the application. It represents what goes on after the rays have been arranged into a number of queues. The first operation is to choose a queue to operate on.

The determination is then made whether the rays in that queue are to be processed locally (on this rank). So that means, for instance, are those rays from my tile (image schedule) or are those rays going through any of the domains I have (domain schedule).

If it is determined that the rays are to be processed on this rank a determination is made as to whether all the necessary data is at hand. If it is then the rays are processed. If not then a loading of data must be done. This process can be done by requesting and waiting for data or by making a data request and jumping to another ray queue for processing. Hopefully data will arrive in a timely manner and rays can be processed.

If it is determined that this queue of rays must be handled on another rank they are dispatched. This either happens immediately by shipping the rays off to another rank (placing them on the outbound data queue) or by putting this collection of rays back on the queue and dealing with it later.

Some evaluation of whether the process is finished is done at this time and the final image composite is performed.

## Software Walkthrough

GraviT functions by managing data across MPI Ranks. The data consists of Ray based data, environment data such as lights and cameras, and (for lack of a better word) science data. Once the data are in place GraviT does its thing. This section of the documentation walks through the structure of a GraviT application. It begins with initialization of data and then describes how GraviT renders the data. Understanding this is important in order to effectively apply GraviT and very important if extending the software.

Since GraviT is a parallel application keep in mind that all of the things described below occur on all of the ranks of the MPI app. Also note that no attempt has been made in the application examples to do efficient IO. What data lives where depends on the type of scheduler. Some applications have been modified to only read a portion of the total data available and thereby pass by the problem of all the data being read on all the nodes.

### Applications

The general flow of an application is to initialize the GraviT state and then to fire off the tracer. At present the state is stored in what is called a "context" object which is a wrapper for a database tree structure. This structure provides scene like functionality with additional options related to the ray tracing task. The logic of an application goes something like this:

- Create a Context object to hold metadata.
- Read model data (geometry/volume)
- Create a GraviT data object to hold the geometry/volume and the material associated with it.
- Build up the Context
    - Load the mesh object into the context
        - Load the mesh itself ( well a pointer to the mesh object)
        - Load transformation data that relates the mesh position in object space to global space into the context
        - Load mesh bounding box information (global space) into context
    - Load lighting data into context
        - Light position, color, type
    - Load Camera data into context
        - Position
        - Focus
        - Upvector
        - Fov
        - rayMaxDepth
        - raySamples
    - Load film into context
        - Width, height
    - Load Scheduler information into context
        - Type (image, domain, other…)
        - Adapter ( choose rendering engine, Embree, Manta, Optix, OSPRay

- Create concrete rendering structure from items in the context.
    - Camera
    - Image (from film node)
- Create a Tracer object based on scheduler type.
- Call the () operator on the tracer object to cause tracing to begin.
- Write out or process the resulting image in some way
    - Call Write() on the image object.

That is pretty much how a GraviT application is structured at present. (July 15, 2016). As can be seen from the outline above the majority of work done in setting up an application is setting up the state (just like an opengl application). The framework is supposed to then do the manipulation or rays and other chores to produce an image. There are three framework objects that are manipulated by the application (other than the context itself). These objects are the camera, image, and tracer objects. The camera and image objects are used in the construction of the tracer object. It is in the constructor of the tracer and the overloaded () operator where the majority of the process logic takes place. This is where adapters are created, rays are sorted and sent etc.

## Tracer Framework

The tracer is where all the action is in this code. At present the tracer is a templated class based on the type of scheduler. The tracer constructer takes a std::vector<ray> and an image object as arguments and builds up the necessary data structures to support rendering. Much of the structure exists in the base class abstractTrace. This base class has the following instance variables:

- RayVector rays
- Image image
- Cntxt (ref to instance of context singleton)
- Std::vector<instance nodes>
- Std::map< int, Mesh>
- Std::vector< light>
- Acceration structure pointer.
- Std::map <int, RayVector>

These are not the actual declarations but they give you an idea of what variables are manipulated by the framework. Of particular interest are the rays object and the image object. The image is the final image buffer. The vector of rays grows and shrinks depending on the tracing methods and parameters. There is also a map of RayVectors. This is a queue of sets of rays. Different sets are destined for different “domains”. Domains are an abstract reference to instances of geometry or to discrete volumes.

### Tracer Initialization (calling the constructor does what?)

A tracer object is initialized via the constructor. Two objects are passed to the ctor. The first is a RayVector of camera rays and the second is an image object or frame buffer to contain the pixel

data. Both of these are copied to instance variables in the abstract base class. Other initialization done in the base class are:

- Initialize the int, mesh map with pointers to mesh instances. This is done by pulling the instance data from the context.
- Initialize light vector to contain pointers to appropriate light types for each light. This is done again by referring to the context objects. Light types are initialized as gvt::render::data::scene::PointLight, AmbientLight, or AreaLight.

Derived tracer initialization depends on the type of scheduler. A Domain scheduler based tracer will create a map of domain instances to mpi ranks.

### Tracer Execution.

Execution of the tracer performs the following tasks:

- Locally filter the rays such that rays that match the instance objects are kept and others are dropped.
- While there are lists of rays to process
    - For the instance corresponding to the longest ray list
        - Obtain an adapter for the instance (create one if it does not exist) based on adapter type. Cache the new adapter if it is being created.
        - Call adapter::trace(rayqueue, raysout, M, Minv, MinvN, lights). Raysout contains rays that need to be returned to the queues, shadow rays, missed rays, etc.
        - Shuffle returned rays into appropriate ray queues.
        - Send rays to appropriate rank.
- Gather frame buffer results.

The key thing to note here is that the rendering back end engine is initialized in two places. Some of it is done in the adapter constructor and some in the adapter trace call itself.