

galileo Design Document

Solomon Boulos

Contents

1	Introduction	3
2	Overview	3
3	Code Organization	4
3.1	Core	4
3.1.1	API	4
3.1.2	Scene Class	4
3.1.3	Context Class	4
3.1.4	Plugin Interfaces	4
3.2	Math	5
3.3	Utilities	5
3.4	User Interfaces	5
3.5	Plugins	5
3.5.1	Plugin Types	5
3.5.2	Writing plugins	6
3.6	Parser	6
4	Development Guidelines	7
4.1	Style Guide	7
4.2	Misc	7

1 Introduction

galileo is a highly realistic, yet flexible, renderer being developed in an effort to provide an efficient, state of the art, Monte Carlo renderer for use in research and artistic endeavors. The two main goals of this document are to provide a brief introduction to *galileo*'s design and to present a set guidelines for continued *galileo* development.

2 Overview

galileo is written in C++ and is built around a plugin architecture. Most desired extensions to *galileo* should be able to be integrated into the system by simply writing a plugin which conforms to the provided class interfaces. The source code can be divide into several logical groups: core software, utility modules and classes, parser, user interfaces, math, and plugins. Each group is contained in its own sudirectory within the `galileo/src` directory with the exception of the plugins which are further seperated into directories based on the plugin type. These groups are discussed further in Section 3. One important part of the core software that is critical to the overall architecture is the *galileo* api. The api is the interface presented to the parser, the user interface, and any additional software that utilizes the *galileo* renderer. The basic flow of execution is shown in Figure 1.

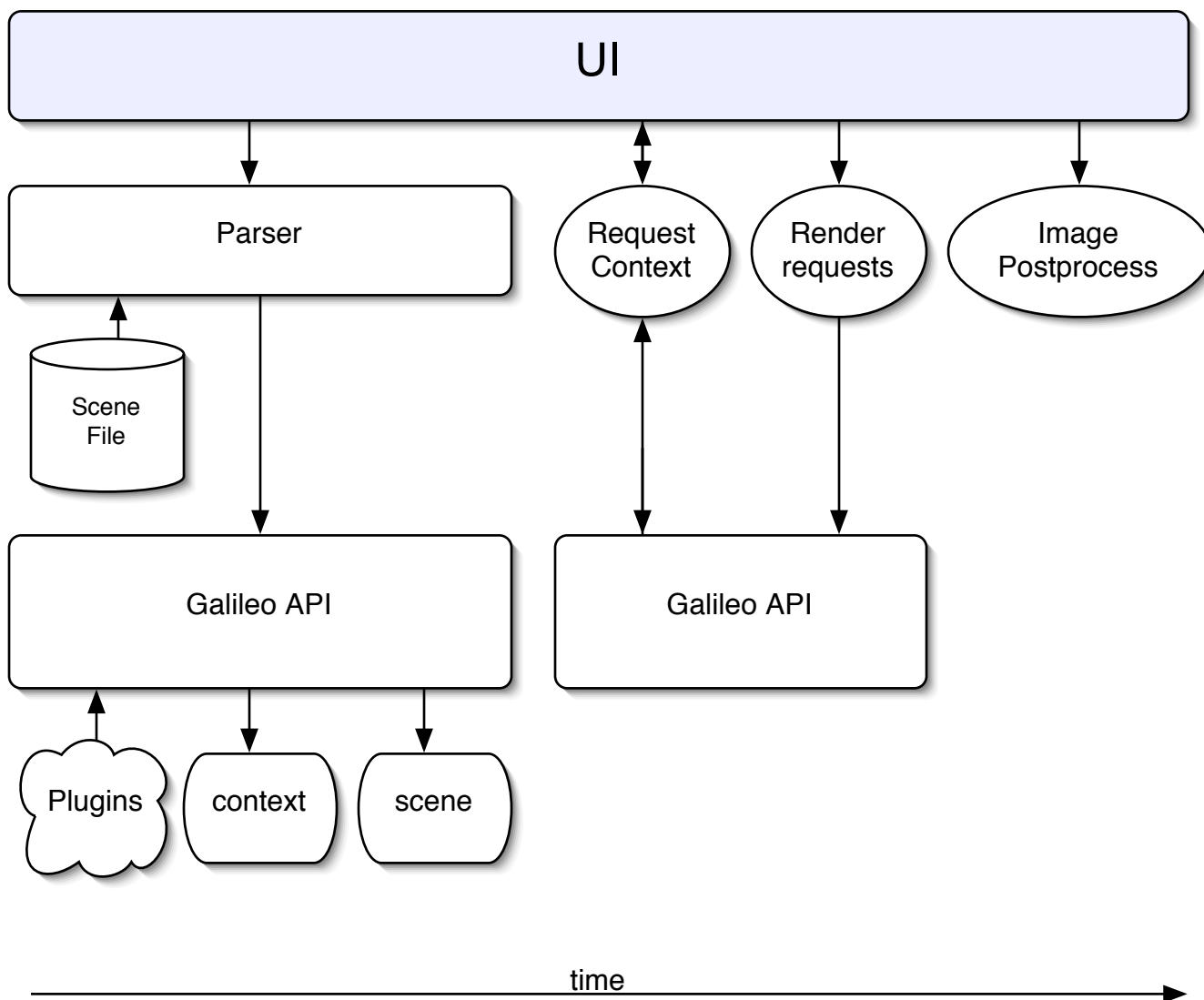


Figure 1: The basic flow of a *galileo* execution. First the **UI** invokes the parser.

The parser reads a scene description file from disk and calls the appropriate API functions to build up a scene description and rendering context. Control then returns to the **UI** which passes render requests to the API and receives back rendered image tiles. Finally, the **UI** performs any postprocessing.

3 Code Organization

Below is a list of the code divisions present in the galileo source tree.

3.1 Core

The `src/core` directory contains code that is integral to the rendering pipeline and all plugin interfaces. Hopefully this code will rarely need to be touched.

3.1.1 API

The *galileo* API, found in `api.h/cc` is similar to the OpenGL or Renderman interfaces. Most of the API functions modify the rendering context (class **Context**) or build up the current scene description (class **Scene**). The API also contains hooks for the parser to access named scene elements and hooks for the user interface (**UI**) to access the rendering context and make rendering requests. The *gr* file format has a one-to-one mapping to the API; each *gr* command corresponds to a single API function.

3.1.2 Scene Class

The **Scene** class stores all scene lights and geometry. It also contains a set of member functions for performing global lighting operations. A pointer to the global **Scene** object is stored in the **Context** class and, making these member functions available to all **SurfaceShaders** for use in shading operations.

3.1.3 Context Class

This class contains all options necessary for a rendering execution. These options can be separated into several categories:

- Rendering options - such as maximum ray depth and near and far clipping distances.
- Image options - these are placed in **Context** so they will be available to the **UI** for image creations, tiling, and postprocessing. Examples are image Gamma and pixel dimensions.
- Thread context - memory specific to the current thread, such as per-thread random number generators and space for storing pixel samples.
- Trace arguments - arguments that are passed down as a ray path is created during rendering. Examples are: current ray, the ray depth and seeds for MC sampling.

3.1.4 Plugin Interfaces

The (mostly) abstract interfaces for the plugin types such as **Sampler** and **SurfaceShader**. See the code docs for detailed explanation of all plugin interfaces.

3.2 Math

Inside of the `src/math` directory, there are classes such as **Vector3**, **MTRand**, and **Matrix**. Each of these classes is available to the entire system including plugins. Any new math classes being added to *galileo* that will most likely have wide applicability should be added here and referenced from within plugins and the rest of the system.

3.3 Utilities

The `src/utilities` directory contains useful utility classes and functions that are not specifically math related. Any code that could be shared among several plugin or core classes should be placed here. An illustrative case is the **Spectrum** class which is used in both the **SpectralLight** class as well as in **SpectralTexture**.

3.4 User Interfaces

User interfaces are found in the `src/ui` directory and adhere to the abstract interface given by the **UI** class. Currently, the only implementation is **BatchUI**, which is a simple command line interface. User interface classes are responsible for invoking the parser, scheduling render tasks, handling image post-processing, and outputting the resulting image (see Figure 1).

3.5 Plugins

One of the overriding design choices made when developing *galileo* was to implement the system as an easily extensible plugin architecture. As a result, most modifications, new algorithms, and extensions to *galileo* should be able to be implemented as non-disruptive, easily integrated C++ plugins. The classic example is the plugin shader design which is used in many modern renderers. In *galileo*, the **SurfaceShader** class hierarchy provides an interface that all shaders representing surface shading models must implement. In order to add a new **SurfaceShader**, a developer only has to implement a new class and place it in the `src/surfaceshaders/` directory.

3.5.1 Plugin Types

galileo goes further than most renderers renderers by designing most major class hierarchies as plugins. Below is a complete list of the plugin types currently available:

- **SurfaceShader** - Shaders that implement a surface lighting algorithm and material model. Most new shaders need only implement the material model portion due to an efficient default implementation lighting algorithm given in the parent class, **SurfaceShader**. For more details refer to the *galileo* source code documentation.
- **LightShader** - Shaders that represents surfaces' self emittance. Included are shaders that can be attached to **Surfaces** to create area lights, environment maps, and traditional singular lights such as directional and point lights.
- **Surface** - Geometric objects that can be intersected with a **Ray**. Both traditional rendering primitives such as sphere and triangle as well as ray intersection acceleration structures such as uniform grids and bounding volume hierarchies fit into this category. Traditional primitives are placed in the `src/surfaces` directory while accelerators are in `src/accels`.
- **ImageIO** - These objects enable image I/O for arbitrary image file formats. *galileo* assumes that a given file extension such as `.tif` is supported by at most one **ImageIO** plugin.

- **Filters** - Pixel sampling filters. Unlike some filter designs which weight samples according to their filter value, *galileo* filters warp the sample positions such that their density is proportional to the filter function. In this manner, we avoid a proliferation of samples with low weights which contribute little to the final image yet are still expensive to evaluate.
- **Samplers** - These samplers are used for creating 1D and 2D samples for Monte Carlo evaluation of integrals. The **Sampler** interface requires that a given sampler be able to create an entire set of samples at once. Functions also exist to support iterative sampling where samples are created on the fly. However, iterative sampling is not currently used anywhere in *galileo*.
- **Cameras** - Virtual camera models. Currently, only a very simple thin lens camera model is implemented. We are in need of a realistic camera class.
- **Textures** - A templated class for storing 2D (such as image maps) and 3D (such as Perlin noise) textures. Support for `float` textures and **rgb** textures is currently available in the API and parser. Additional texture template types can be added as necessary. Some of the uses of textures are: representing participating media densities, surface reflectance properties, and light emittance properties.

Cases might arise in which a new feature will not fit into one of the above plugin categories. If this circumstance occurs, a new plugin type may be added or the core library may need to be modified. Either of these options will require modification of the API, the renderer, and most likely other core classes.

3.5.2 Writing plugins

Many standard objects in a path tracer are plugins in *galileo*. For example, a Triangle is a Surface plugin. The plugin architecture for *galileo* is fairly straightforward. Each extensible core class has a directory for its plugins. The Triangle plugin is located in the `src/surfaces` directory and is implemented in `Triangle.cc`. The *galileo* Makefile is configured to automatically detect new plugins, which should be implemented in a single `.cc` file with the same filename as the intended class name. This is to aid the dynamic object loader in finding the appropriate object and class when parsing the scene description. Note that no modification to the Makefile is required to add a new plugin to *galileo*.

Each extensible core class also requires that a construction function be provided by a plugin. The Surface plugins must all implement a `createSurface` function that takes a `SurfaceShader`, a transformation `Matrix` and list of parameters from the parser. This function should be placed in the implementation file (`PluginName.cc`) and declared as follows:

```
extern "C" Surface* createSurface(SurfaceShader*, const Matrix&, Params&);
```

When the scene parser is asked to create a `PluginName Surface`, it will attempt to load the `PluginName` plugin from the `dsos` directory (where plugins are stored as `PluginName.so`) and call the `createSurface` function. This function is declared `extern C`, so that the C++ compiler does not mangle the name into something other than the expected `createSurface` function.

3.6 Parser

galileo contains a *gr* file format parser implemented with *flex* and *bison*. The *gr* format is a text based format in which each command corresponds to a single *galileo* API function call. The parser and API are designed in such a way that the addition of new plugins does not require modification to the parser. The flexible **Param** class enables the parser and API to pass arbitrary constructor arguments into new plugin classes. Compact file formats for representing meshes (such as the PLY format) can be supported via **Surface** plugins, such as the **PLYMesh** class.

4 Development Guidelines

OCD Keith's guidelines for keeping the source tree clean. (ignore if you dont ever want to check in code).

4.1 Style Guide

Here are a few fun guidelines for keeping a somewhat consistent style among source files in the repository:

- Use the globals (such as `GR_TWO_PI`) found in `utilities/GalileoGlobals.h` when possible. If you come across a useful constant not already present, feel free to add it.
- Use the functions (such as `GRLerp`) found in `utilities/GalileoFunctions.h` when possible. If you come across a useful function not already present, feel free to add it.
- Keep line widths to 80 columns wide or less. Some old editors assume this property, I like it for aesthetic reasons.
- Naming conventions: `ClassName`, `functionName`, `variable_name`, `CONSTANT_NAME`.
- Put a comment header in all header files with author name in case a user or developer has questions about a class implementation.
- Use an indent tab width of four (space filled).

4.2 Misc

Using the `benchmarks.txt` to ensure the code remains fast, etc, etc