kriti Documentation

Release 0.1

ethereal

February 18, 2016

Contents

CHAPTER 1

About

1.1 What is kriti?

Kriti is a 3D game engine for games written in C++. It aims to provide a medium-weight approach so that while it imposes some structure on the game you're trying to write, it also stays out of your way when you want to do things yourself.

Kriti is cross-platform, using OpenGL as a rendering method and has been written with platform compatibility in mind. Right now, it lacks support for mobile platforms, though support for such is planned.

1.2 Features of kriti

Current features:

- Full multi-stage rendering pipeline with RTT and blending
- GUI library
- Open Asset Import Library integration to load almost all 3D models
- Integration with bullet physics engine, either with render geometry or with simplified physics model
- XML-based configuration available for many engine objects, including rendering pipeline configuration

Planned features:

- Anti-aliasing (MSAA)
- Conditional rendering (à la level-of-detail rendering)
- Sound via OpenAL

1.3 Dependencies of kriti

Kriti needs:

- A C++11-capable compiler (gcc and clang tested so far)
- Boost (at least version 1.55)
- CMake (version 2.6 or later)
- SDL 2.0

- SDL_image 2.0
- OpenGL 3.1 (or later) development libraries
- GLEW
- bullet physics engine
- FreeType

1.4 Licensing

The main kriti codebase is released under a 3-clause BSD license. That essentially means you can do whatever you want with it, provided you don't claim it as your own. Kriti also contains parts of several other open-source libraries, relased under various licenses; see the COPYING file for more details.

1.5 Authors

- Kent "ethereal" Williams-King

CHAPTER 2

Kriti architecture

2.1 Submodules

Kriti can considered to be built out of several different submodules:

- The state engine; this is responsible for managing the state of different parts of a game and the engine.
- The math library; this provides general 3D mathematics data-types, functions, as well as some 2D computational geometry functionality.
- The rendering engine; this is where the 3D magic happens and pretty pictures are generated.
- The scene library; this is where models are created and animated, cameras set up, and lights tweaked. Basically, the scene graph.
- The interface library; this provides code to glue together the interface the user sees with the rest of the codebase.
- The GUI library; this provides for all your interactive needs.
- The physics integration; this provides a wrapper for Bullet, allowing for a unified interface within kriti for rendering and physics.

2.2 Message and logging system

Kriti has a very complete logging system, intended to be used mostly for debugging.

2.3 Resource system

Kriti also has an external resource management system, intended to be agnostic to the method used to actually store information. While at the moment this is a simple wrapper around the host filesystem, the intention is that it will also allow for loading from ZIP archives as well.

2.4 Buildsystem

Unlike some other game libraries, Kriti does not require much out of the buildsystem. While only CMake integration is currently tested, it should be straightforward to integrate with anything make-alike.

To add Kriti into a project using CMake, simply add the following line somewhere in your buildscripts:

```
include(path-to-kriti/cmake/kriti-env.cmake)
```

Once that has been added, simply change your `target_link_libraries` line to include `${kritiLibraries}`.

Kriti will take over the `main()` function, leaving you to implement a `gameEntryPoint()` function somewhere in your source tree. This will be explained further in the Basics section later.

TODO: finish explanation of project architecture.

CHAPTER 3

Kriti basics

Let's look at a simple example. As was discussed in the Buildsystem section earlier, we only need to consider one source file that exports the `gameEntryPoint` function. So, for us, a skeleton example might be something like:

```
void gameEntryPoint() {

}
```

Obviously, this isn't very interesting. So, let's start looking at some of the basic parts of the Kriti architecture.

3.1 Message and logging system

Let's start by taking a look at how the logging system works. Debugging information is very important, and while much of the time a `std::cout` does the trick, sometimes we want a more high-powered solution. Kriti's `MessageSystem` is intended to address that need. In particular, it:

- Allows multiple levels of verbosity on a per-module basis,
- Prepends timestamp to message,
- Automatically logs written content to a file.

Using it is very simple:

```
#include "kriti/MessageSystem.h"

void gameEntryPoint() {
    Message("This is a generic log message.");
    Message2(Debug, "This is a generic debugging message.");
    Message2(Error, "This is a generic error message.");
    Message3(Game, Debug, "This is a game debugging message.");
}
```

These will be formatted as:

```
[17:28:56 General      Log    ]         This is a generic log message.
[17:28:56 General      Debug  ]         This is a generic debugging message.
[17:28:56 General      Error  ]         This is a generic error message.
[17:28:56 Game         Debug  ]         This is a game debugging message.
```

TODO: elaborate how to create custom categories.

3.2 Context and state system

One of the most critical parts of kriti to manage is the state and context system. The idea is sort of to refactor the main event loop of a game into smaller, more easily-managed pieces.

There are three important classes that are involved here: `State::Context`, `State::Context::Event`, and `State::Context::Listener`. Each `State::Context` acts as a single event loop, which can optionally proxy events to another context if it proves to be useful. We'll start explaining how this all works with a simple code example:

```
#include "kriti/state/Context.h"
#include "kriti/MessageSystem.h"

using namespace Kriti;

void listener1() {
    Message("listener1()");
}

void listener2() {
    Message("listener2()");
}

void gameEntryPoint() {
    auto context = boost::make_shared<State::Context>();

    context->addListener("event1", listener1);
    context->addListener("event1", listener2);
    context->addListener("event2", listener2);

    context->fire("event1");
    context->fire("event2");

    context->processQueued();
}
```

This was intended to be fairly self-explanatory for people who have seen event systems before, but let's walk through it quickly:

- Contexts can have named events (such as `event1`) and have C-style functions registered as callbacks for when they are fired,
- Events can be fired by name,
- Nothing happens until the queue is processed.

Of course, this is somewhat boring. What if we want to pass around events by something other than an ugly string? We can do that; that's what `getEvent` does — it returns a `State::Context::Event` instance, which can then be fired off by itself if you so choose:

```
#include "kriti/state/Context.h"
#include "kriti/MessageSystem.h"

using namespace Kriti;

void listener1() {
    Message("listener1()");
}
```

```
void gameEntryPoint() {
    auto context = boost::make_shared<State::Context>();

    context->addListener("event1", listener1);

    context->getEvent("event1")->fire(boost::tuple<>());

    context->processQueued();
}
```

This got a little different in the middle there — in particular, it introduces the concept that events can have parameters, which are passed around as `boost::tuple` instances. However, in this case, our listener function doesn't take any parameters, so we just instantiate it to be an empty tuple.

This lets us fire events in arbitrary contexts, sure. But what if we want something other than a C-style global or static function to be the callback? Well, we can do this, because `addListener` actually takes a `boost::function`. So anything that has an `operator()` can be used, including lambdas or member functions. A simple example, this time including parameters:

```
#include "kriti/state/Context.h"
#include "kriti/MessageSystem.h"

using namespace Kriti;

class PairPrinter {
private:
    int m_private;
public:
    PairPrinter(int priv) : m_private(priv) {}

    void print(int n) {
        Message("Pair: (" << m_private << "," << n << ")");
    }
};

void gameEntryPoint() {
    auto context = boost::make_shared<State::Context>();
    auto event = context->addEvent<int>();

    context->addListener(event, boost::function<void (int)>(
        boost::bind(&PairPrinter::print, new PairPrinter(3), _1)));
    context->addListener(event, boost::function<void (int)>(
        boost::bind(&PairPrinter::print, new PairPrinter(4), _1)));
    context->addListener(event, boost::function<void (int)>(
        [](int v){ Message("lambda value: " << v); }));

    event->fire(boost::make_tuple(2));

    context->processQueued();
}
```

Finally, the `addListener` function returns an instance of `State::Context::Listener`, which has a `disconnect()` function to allow you to stop listening for events.

3.3 Resource system

Kriti has a resource system that is designed to hide the actual source of the data being loaded. The intention is to allow for files to be loaded from the standard filesystem, from a tarball/zip file, from the network, etc. This generality does make it slightly more complicated to use than a simple `fstream` or `FILE *`, but it also allows for resources to automatically be shared where appropriate and have parsers etc. applied to file contents.

There are two main classes to consider in the resource system. First is the base class of all loadable resources, `Resource`. This is essentially an empty base class, with its only function `loadFrom(std::string id)`, where `id` is a string to denote what to load from. This function is only relevant if you are implementing your own type of resource.

The second class is the `ResourceRegistry` class. This is a singleton class that stores references to loaded `Resource` instances. It provides a simple template accessor method that makes this straightforwards to use.

Let's show how this all works by using a simple example resource type, `FileResource`. This simply loads a file's content into memory and then allows it to be accessed through standard C++ data types:

```
#include "kriti/FileResource.h"
#include "kriti/ResourceRegistry.h"

void gameEntryPoint() {
    auto file = Kriti::ResourceRegistry::get<Kriti::FileResource>("file.txt");

    if(!file) {
        Message3(Game, Fatal, "Couldn't open required file!");
    }

    Message3(Game, Debug, "File content: " << file->fileContent());
}
```

The class `Kriti::FileResource` will load files from the data path specified in the XML configuration file, which we'll get to in a bit. If not specified, this is by default the path `data/`.

Once loaded once by the ResourceRegistry, the resource will be kept in memory until explicitly removed with the `ResourceRegistry::clear` function. Note that descriptors are kept in separate namespaces per resource type, so `ResourceRegistry::get<FileResource>("name")` will return a different result than `ResourceRegistry::get<Render::Texture>("name")`. To avoid confusion with this, you should avoid having resource types that are castable to each other.

Let's say, for the moment, that we want to add a new resource type, `NPCResource`. This needs some XML configuration information, a texture, and maybe some lines of dialogue. We can do this by creating a new subclass of `Resource` and specifying a `loadFrom` function, like so:

```
#include "kriti/FileResource.h"
#include "kriti/XMLResource.h"
#include "kriti/render/Texture.h"
#include "kriti/ResourceRegistry.h"

class NPCResource : public Kriti::Resource {
private:
    boost::shared_ptr<Kriti::XMLResource> m_config;
    boost::shared_ptr<Kriti::Render::Texture> m_appearance;
    boost::shared_ptr<Kriti::FileResource> m_dialogueFile;
    std::map<std::string, int> m_dialogueMap;
public:
    // should return true if the resource was loaded successfully, false otherwise
    virtual bool loadFrom(std::string identifier) {
        m_config = Kriti::ResourceRegistry::get<Kriti::XMLResource>(
```

```
            "npcs/" + identifier);
        if(!m_config) return false;

        m_dialogueFile = Kriti::ResourceRegistry::get<Kriti::FileResource>(
            "npcs/" + identifier);
        if(!m_dialogueFile) return false;

        m_appearance = Kriti::ResourceRegistry::get<Kriti::Render::Texture>(
            "npc_" + identifier);
        if(!m_appearance) return false;

        // construct mapping of dialogue names to lines in the dialogue file from config in XML file

        return true;
    }

    std::string getDialogueLine(std::string which) {
        return m_dialogueFile.fileLines()[m_dialogueMap[which]];
    }
};
```

As a rule of thumb, if you have any functions in a resource that involve expensive computation, you should probably cache the result. The intended design pattern is that `Resource` instances may be accessed from various locations in the codebase, including hot-paths.

3.4 Configuration

As was mentioned earlier in the resource system section, there is a 'configuration' resource. This is an XML file, usually called `kriti.xml`, placed in the same directory as the executable. This is accessed by the special `XMLResource` name "config". This stores information such as the directory to put log files in, where to load further data files, etc. A configuration file with all the default values present would be as follows:

```
<kriti>
    <general>
        <data-path>data/</data-path>
        <logfile>logs/kriti-%d.log</logfile>
        <profile>false</profile>
    </general>
    <video>
        <resolution width="800" height="600" bpp="0" fullscreen="false" />
        <msaa enabled="false" samples="4" />
    </video>
</kriti>
```

CHAPTER 4

Math library

The math library falls into two categories: vector math and computational geometry. All of the following functionality is present inside the `Kriti::Math` namespace, and as such this will be dropped from the descriptions for brevity.

4.1 Vector math

TODO: fill out

There are several useful classes in the vector math library. The first, and most-oft-used, is `Vector`. This represents a three-dimensional vector, and has overloaded operators for most common operations. Other member functions that may be of interest are `length()`, `length2()`, `cross()`, `dot()`, `projectOnto()`, and the `toString()` function. Individual components are accessible by `vector.x()` etc. as well as `vector[1]`.

There is also a `Point` class, which inherits from `Vector`. This is intended for use in places where you distinctly have a location in 3D space as opposed to a difference between locations. Due to the inheritance, a `Point` can be used anywhere a `Vector` can, but not vice-versa.

The `Matrix` class represents a standard 4x4 matrix, stored in column-major ordering internally. Components are accessible by `matrix(0, 3)`, which accesses the last element in the first column. Note that the `Matrix` class has overloaded operators, and in particular the `Matrix * Point` operator behaves differently than the `Matrix * Vector` operator.

Finally, the `Quaternion` class is present, providing a full library for the use of unit quaternions for representing rotations. Useful functionality is present in the overloaded operators, the `toMatrix()` function, and the `slerp()` function. The constructor takes an axis vector and an angle to use to represent the initial rotation.

TODO: add view calculation etc.

4.2 Computational geometry

The computational geometry library is currently not yet complete and includes basic 2D geometry functionality, such as:

- `Geometry::closestPoint`: finds the closest point on a line to another point `p`.
- `Geometry::closestSegmentPoint`: finds the closest point on a line segment to another point `p`.
- `Geometry::intersectAARect`: calculates the intersection of two axis-aligned rectangles and returns another axis-aligned rectangle.

TODO: finish filling out

CHAPTER 5

Rendering

5.1 Rendering model

The rendering model is based around three core objects: `Render::Pipeline`, `Render::Stage`, and `Render::Renderable`. The idea is that a `Render::Pipeline` describes an entire rendering process, including all the multi-stage passes, all the RTT, all the conditional rendering, everything. This is done by breaking up the rendering process into individual `Stage` components that can be evaluated individually, with optional inter-`Stage` dependencies.

Each `Stage` renders zero or more `Renderable` instances, which can be thought of as any sort of self-contained object that can be rendered to an image.

TODO: finish

5.2 Renderable objects

CHAPTER 6

GUI library

6.1 Event types

- **Generic mouse events:**
  - `mouseMovedTo(Math::Vector)`
  - `mouseEntered()`
  - `mouseLeft()`
  - `mouseButtonDown(int)`
  - `mouseButtonUp(int)`
  - `mouseClicked(int)`
- **`LineEdit`**
  - `gainedFocus()`
  - `lostFocus()`
  - `textChanged()`

CHAPTER 7

Indices and tables

- genindex
- search