futils Documentation

Release alpha

Felix Ganz

Aug 03, 2018

Contents

CHAPTER 1

Introduction

1.1 Abstract

This repository is divided into two parts. Firstly, a description of my utils, a collection of header-only c++ solutions to recurring problems. I am aware that alternatives exist such as boost, but as a dev in training, I often had to restrain myself from using already existing solutions. Secondly, you'll find documentation on Fender, the only existing project (for now!) that actually uses these utils.

1.2 Philosophy

I believe sharing is caring, and therefore everything is open-source, fully readable on Github. Consider this website as a Request For Comments, feel free to send feedback through github/issues. My goal is to keep everything I do neatly in one place, thus I can improve upon whenever I feel like, and hopefully it may help others.

1.3 Collabs

I want to thank everyone who's helped me, I don't intend to list them, they'll know. However, I want to thank the devs at ReadTheDocs.io for their work :)

Happy learning !

CHAPTER 2

Installing fender

Fender is a cross-platform (windows, linux, [WIP]osx) static library in c++.

2.1 Dependency

2.1.1 Linux

On Linux, you'll need to install the latest version of the SFML (if you intend to use the SFML).

2.1.2 Windows

You should not have anything special to do on Windows regarding the SFML.

2.2 Download

```
$ git clone git@github.com:/ganzf/futils.git
```

2.3 Creating a project

Now that you've cloned the repository, you can go to `futils/projects/fender`

```
$ cd futils/projects/fender
```

And copy both `createFenderProject.sh` and `fenderGettingStarted` directory.

Use the script next to the directory to create a new directory.

```
$ ./createFenderProject.sh MyProjectName
```

This will create a directory named MyProjectName and update the CMakeLists.txt to build an executable of the same name.

2.4 Build

```
$ cd MyProjectName
$ ./all.sh
$  - Downloading futils...
$ Cloning into '.dl_futils'...
$ remote: Counting objects: 536, done.
$ remote: Compressing objects: 100% (329/329), done.
$ remote: Total 536 (delta 147), reused 518 (delta 143), pack-reused 0
$ Receiving objects: 100% (536/536), 15.64 MiB | 496.00 KiB/s, done.
$ Resolving deltas: 100% (147/147), done.
$  - Done.
$ Fetching update...
$ From github.com:ganzf/futils
$  * branch            master     -> FETCH_HEAD
$ Already up to date.
$ Update complete.
$ -- The C compiler identification is GNU 7.2.0
$ -- The CXX compiler identification is GNU 7.2.0
$ -- Check for working C compiler: /usr/bin/cc
$ -- Check for working C compiler: /usr/bin/cc -- works
$ -- Detecting C compiler ABI info
$ -- Detecting C compiler ABI info - done
$ -- Detecting C compile features
$ -- Detecting C compile features - done
$ -- Check for working CXX compiler: /usr/bin/c++
$ -- Check for working CXX compiler: /usr/bin/c++ -- works
$ -- Detecting CXX compiler ABI info
$ -- Detecting CXX compiler ABI info - done
$ -- Detecting CXX compile features
$ -- Detecting CXX compile features - done
$ -- Configuring done
$ -- Generating done
$ -- Build files have been written to: /home/arroganz/cpp/MyProjectName/build
$ Scanning dependencies of target MyProjectName
$ [ 50%] Building CXX object CMakeFiles/MyProjectName.dir/src/main.cpp.o
$ [100%] Linking CXX executable ../MyProjectName
$ [100%] Built target MyProjectName
$ [100%] Built target MyProjectName
```

2.5 Update

Use the update.sh script to fetch new stable versions of the library.

CHAPTER 3

ECS Design Explained

Disclaimer : This is not actual working code, this is only intended to help you understand the underlying design of the engine.

The ECS is a design pattern used to favor **composition** over inheritance. You create entities, that are composed of components (most often Plain Old Data) and meaningful output is done through the systems in a regular (systemic) way.

For example, here is a Window :

```
class Window : public Entity
{
                Window(string name, string icon = "defaultIcon.ico") {
                      attach<WindowComponent>(name, icon);
                }
};
```

As you can see it is an entity (`public Entity`) composed of a WindowComponent component (`attach<WindowComponent>`) that holds the name of our Window, and a path to an icon.

Using the EntityManager (the glue that makes everything work in a magical way) we create a window:

```
entityManager->create<Window>("Demo");
```

And thus a Window is born with the name `Demo`, and the engine is notified.

> But wait... what happened ?

Well, Fender actually already loaded a system to handle `WindowComponents` named... `Window`.

When the EntityManager creates a `Window`, it knows its going to be composed of a `Window`. It will emit an event `ComponentAttached<Window>` and the `Window System` will catch that event and create an actual Window. (An SFML Window for example).

Now you should keep the return value of the `create<T>` function. Say we want to change the name of our window.

```
auto &win = entityManager->create<Window>("Demo")
auto &window = win.get<Window>();
window.name = "Awesome";
```

Since `window` 's name is now "Awesome", the `Window System` will now display "Awesome".

What if you want to add a background color to your Window ?

```
auto &bg = win.attach<Color>();
bg = Granite;
```

or

```
auto &bg = win.attach<Color>(Granite);
```

Now the `Window System` will know the window has a `Color Component` and will update the window with a Granite like background.

You want to remove the background ?

```
win.detach<Color>();
```

And we're back to a normal window with a black background color;

For now, there is no error checking.

> What errors ?

By error, I mean what if you attached a component to an entity but it made no sense ?

```
win.attach<onFire>();
```

Well, it will work. But it won't have any impact. If you want to know more about how components affect your program, you can read either the documentation or the source code of the systems. Beware what components you put together though, some combinations might not work as you'd expect, or work at all.

If you want to know what the actual fender-ecs classes look like, follow these links :

3.1 class futils::EntityManager

```
 class   EntityManager
{
public:
    EntityManager();

    template    <typename T, typename ...Args>
    T           &create(Args ...args);

    template    <typename System, typename ...Args>
    void        addSystem(Args ...args);

    void removeSystem(std::string const &systemName);

    template <typename T>
    std::vector<T *> get();

    bool        isFine();
```

(continues on next page)

(continued from previous page)

```
    int getNumberOfSystems() const;

    int run();
};
```

3.1.1 create<T>

This function allows you to create a futils::IEntity with variable argument list and returns a reference to the newly created entity.

```
auto &thing = entityManager->create<Thing>(12, "lol", nullptr);
```

3.1.2 addSystem<T>

This function allows you to create a futils::ISystem with variable argument list.

```
entityManager->addSystem<MySystem>(12, 13, 15);
```

3.1.3 get<T>

This function builds a vector of pointers to components of type T.

```
auto clickables = entityManager->get<Clickable>();
for (auto &clickable: clickables)
(...)
```

> You should always use `create` and `addSystem` to add entities or systems. Anything else results in **undefined behavior**

3.2 class futils::IEntity

```
class   IEntity
{
 public:
     IEntity();
     virtual ~IEntity() {}

     template    <typename Compo, typename ...Args>
     Compo       &attach(Args ...args);

     template <typename T>
     T &get() const

     template <typename Compo>
     bool detach()

     int         getId() const
 };
```

3.2.1 attach<T>

Builds and adds a component to the entity.

```
entity->attach<components::Color>(futils::Granite);
entity->attach<gameplay::components::Alive>("The Rock", 50);
```

3.2.2 get<T>

Get a reference to the component T held by the entity. **Throws std::runtime_error** if not found.

```
auto &color = entity->get<components::Color>();
color.color = futils::White;
```

3.2.3 detach<T>

Remove and destroy component of type T held by the entity. Returns **true** if detached, **false** otherwise.

3.3 class futils::IComponent

```
1  class   IComponent
2  {
3  public:
4      virtual ~IComponent() {}
5
6      IEntity &getEntity() const
7      {
8          return *__entity;
9      }
10 };
```

3.3.1 IEntity &getEntity()

This is the only function you should care about. It returns a reference to the pointer.

3.4 class futils::ISystem

```
class   ISystem
{
 protected:
     std::string name{"Undefined"};
     EntityManager *entityManager{nullptr};
     Mediator *events{nullptr};
     std::function<void(EntityManager *)> afterDeath{[](EntityManager *){}};

     template <typename T>
     void addReaction(std::function<void(IMediatorPacket &pkg)> fun);
 public:
```

(continues on next page)

(continued from previous page)

```
    virtual ~ISystem() {}
    virtual void run(float elapsed = 0) = 0;

    // You should not use these functions (they'll be friend of EntityManager soon)
    // {
    void provideManager(EntityManager &manager) { entityManager = &manager; }
    void provideMediator(Mediator &mediator) { events = &mediator; }
    std::string const &getName() const { return name; }
    std::function<void(EntityManager *)> getAfterDeath();
    // }
};
```

3.4.1 protected: void addReaction<T>

Protected addReaction binds the *fun* lambda to the event T. Be careful not to call this function in your system constructor, as `events` will be nullptr.

```
addReaction<SomeEvent>([this](futils::IMediatorPacket &pkg){
            auto &packet = futils::Mediator::rebuild<SomeEvent>(pkg);
            (...)
});
```

3.4.2 protected: name

Systems all have a name, and should be **unique**.

3.4.3 protected: entityManager

The entityManager pointer will be set by the entityManager itself when calling `entityManager->addSystem<T>()`

3.4.4 protected: events

The events pointer will be set by the entityManager itself when calling `entityManager->addSystem<T>()`

3.4.5 protected: afterDeath

This lambda allows you to specify a behavior after you've deleted your system.

> Do not capture **this** in your afterDeath function. It will be a dangling pointer.

3.4.6 virtual void run(float elapsed)

This pure method is the basic function of any system. You can implement whatever you like. The *elapsed* parameter describes in milliseconds the time elapsed from previous run. It highly depends on the number of systems in the engine and you **must** use it. Otherwise your system may run too fast or too slow. If you have question see FluidTimestep

CHAPTER 4

Getting Started with Fender

Since you cloned the repository, you can use the shell script `getStarted projectName`.

It will create the following architecture :

```
gettingStarted
|- all.sh
|- update.sh
|- build.sh
|- compile.sh
|- CMakeLists.txt
|- doc/
|- futils/
|- include/
|- lib/
|- src/
|- Readme.md
```

Use `update.sh` to fetch the latest Fender build and includes. Use `build.sh` on Unix to build the project Use `compile.sh` to create a binary executable file. Use `all.sh` for all of the above.

> The update.sh script will clone futils in a hidden folder.

4.1 Creating and adding your systems

In the newly created directory, you'll find basic main.cpp

```
# include "fender.hpp"

int main(int, char **av)
{
            fender::Fender engine(av[0]);

            if (engine.start() != 0)
```

(continues on next page)

(continued from previous page)

```
            return 1;
          engine.run();
          return 0;
}
```

This is a minimal working usage of the engine. Default systems will be loaded (by default it loads the SFMLSystems). If you want to add your own systems, here's where you should do it.

```
# include "fender.hpp"
# include "mySystem.hpp"

// mySystem.hpp
class mySystem : public futils::ISystem
{
          (...)
public:
          mySystem(bool someValue);
          void run(float) override;
};

int main(int, char **av)
{
          fender::Fender engine(av[0]);

          if (engine.start() != 0)
            return 1;
          engine.addSystem<mySystem>(true);
          engine.run();
          return 0;

}
```

> Do not add your systems before you call `engine.start()`

If everything goes well, here's what you could see

```
-->     LOG => [Window] loaded.
-->     LOG => [Input] loaded.
-->     LOG => [Camera] loaded.
-->     LOG => [Grid] loaded.
-->     LOG => [Children] loaded.
-->     LOG => [Border] loaded.
-->     LOG => [mySystem] loaded.
```

4.2 Shutting down

Fender will run as long as at least one system is up. So how do you exit your program cleanly ? send the `Shutdown` event declared in `events.hpp`.

> Therefore, all systems **must** require this event and remove themselves from the engine.

```
addReaction<fender::events::Shutdown>([this](futils::IMediatorPacket &){
          (...)
          this->entityManager->removeSystem(this->name);
```

(continues on next page)

(continued from previous page)

```
                (...)
});
```

> I didn't want to force all systems to react in a single way to Shutdown event. But don't forget to add your own reaction.

CHAPTER 5

Fender

Now that you know how to get a working environment and that you understand what an ECS is (if not, please go back a few steps!) lets see what Fender actually **IS**.

5.1 Fender - Introduction

5.1.1 Vocabulary

	Definition
Event	A type that you **send** and **require** through a mediator.
Trinity	Something that exists as an **entity**, a **component** and a **system**

5.1.2 Notations

Notation	Meaning
[name]	Indicates that name is a system
(name)	Indicates that name is a component
Name	Indicates that name is an Entity
{ Name }	Indicates that name is a trinity

5.1.3 Tutorial goal

In this tutorial you will learn how to use fender to create a simple snake game. **This is a work in progress** : you may not actually build an entire game with this tutorial, as it is written during development of the engine.

5.2 Fender

- Defined in `fender.hpp`

It all starts with `fender::Fender`.

Here's what it looks like :

```
class Fender
{
public:
     Fender(std::string const &av_0);

     int start();
     int run();

     template <typename System, typename ...Args>
     void addSystem(Args ...args);

                 template <typename System>
                 void addSystem();

     template <typename System>
     void removeSystem();
 };
```

5.2.1 Fender(string const &)

Fender requires `std::string const &` to `av[0]` in order to make all relative paths work properly.

5.2.2 int start()

Starting the engine is the second function call you should make. It will let the engine initialize itself. Right after starting the engine, you can add your own systems.

5.2.3 addSystem<T>(...)

Use this function to add your own systems after starting the engine.

5.2.4 removeSystem<T>

Use this function to remove a system (it can be one added by default, or your own).

Once you're set, simply run the engine.

5.2.5 int run()

This function returns the engine status. You should call this function only once when you're all set.

5.3 Fender events

One thing that's quite cool about fender is that it makes sharing information between different systems easy. You can **send** events to notify others and **require** events to bind an anoymous lambda function to any type.

This is done through the use of the `events` variable that any system will have. Note that you must create your system through a call to `addSystem<T>(...)` for the `events` variable to be set.

For now, only synchronous events are supported. Soon however, asynchronous tasks may be scheduled.

```
class Mediator
{
public:
     template <typename T>
     static inline const T &rebuild(IMediatorPacket &pkg);

     template <typename T>
     void send(T &&data = T());

     template <typename T>
     void send (T const &data);

     template <typename T>
     void require(void *callee, std::function<void(futils::IMediatorPacket &)>
↪onReceive);

     template <typename T>
     void forget(void *callee);

     void erase(void *callee);
 };
```

5.3.1 T &rebuild<T>(IMediatorPacket &)

This function will extract the data of type T hidden inside the IMediatorPacket. If there is a type mismatch (requested a type A in a packet that holds a type B) an **std::logic_error** will be thrown.

5.3.2 send<T const &>()

This function sends an event of type T constructed from an lvalue.

```
TextMessage message;

message.text = "Hello World !";
events->send<TextMessage>(message);
```

Here, message is an lvalue.

5.3.3 send<T &&>()

This function sends an event of type T constructed from an rvalue.

```
events->send<std::string>("Hello World!");
```

Here, the type is std::string and we're giving an rvalue to the send function.

5.3.4 require<T>()

This function binds a lambda function to a type in the event system.

```
events->require<std::string>(this, [this](futils::IMediatorPacket &pkg){
            auto &str = futils::Mediator::rebuild<std::string>(pkg);
            std::cout << "Received a string : " << str << std::endl;
});
```

This is **the only way** you should get events, rebuild their concrete type and use them.

5.3.5 forget<T>()

If you want to stop reacting to some events, or you want to change the binded function you need to call forget<T>.

```
events->forget<std::string>(this);
```

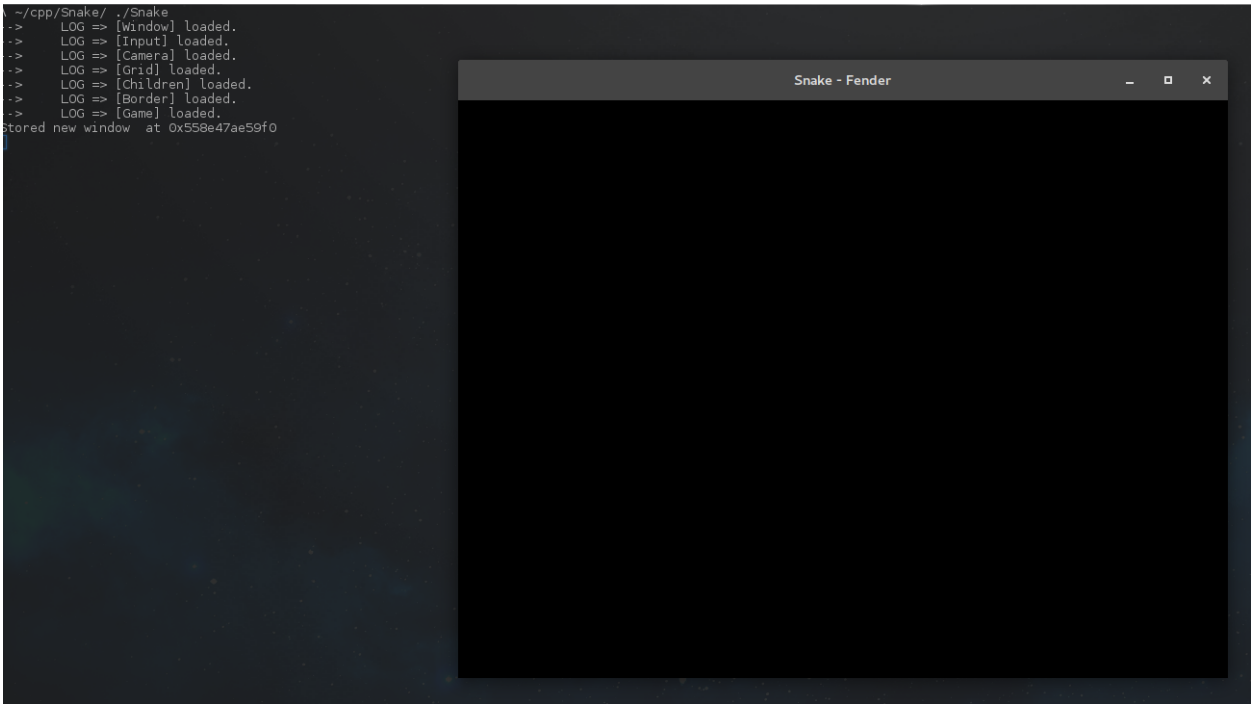
5.3.6 erase()

If you want to forget all events, simply call `erase(this)`.

5.4 Opening a Window

Now that you know the basics (**creating and adding a system**, **sending and receiving events**), let's see what you can do with the engine. Since this engine is targeted at developers, it requires you to create everything, starting with the window.

5.4.1 Expected result



5.4.2 Creating the game system

Remember, we want to create a simple snake game. We're gonna create a Game system.

```
# include "fender.hpp"
# include "events.hpp"
# include "Entities/Window.hpp"

class Game : public futils::ISystem
{
              fender::entities::Window *myWindow{nullptr};
              void init();
public:
              Game() = default;
              void run(float) final;
}

void Game::run(float)
{
  static int phase = 0;
  switch(phase) {
    case 0:
              phase = 1;
              return init();
    case 1:
              return ;
  }
}
```

(continues on next page)

(continued from previous page)

```
void Game::init()
{
  myWindow = &entityManager->create<fender::entities::Window>();
  auto &winComponent = myWindow->get<fender::components::Window>();
  if (!winComponent.isOpen) {
    winComponent.visible = true;
    winComponent.title = "Snake - Fender";
    winComponent.size.w = 800; // in px
    winComponent.size.h = 600; // in px
  }
  addReaction<fender::events::Shutdown>([this](futils::IMediatorPacket &)
  {
    this->entityManager->removeSystem(this->name);
  });
};
```

Let's take some time to understand this bit of code. First, you should know that if you compile and run, it will only open a window and display a black background. But it's a start !

```
fender::entities::Window *myWindow{nullptr};
```

I'm holding a **pointer** to entity because I need to **create** the entity with the **entityManager**.

```
void Game::run(float)
{
  static int phase = 0;
  switch(phase) {
    case 0:
              phase = 1;
              return init();
    case 1:
              return ;
  }
}
```

I'm using a static int to switch **states** : i'll initialize once and then forever just return. You are not forced to have an `init` function, but its often required (if only for event reactions).

5.4.3 Window init

```
void Game::init()
{
  myWindow = &entityManager->create<fender::entities::Window>();
```

Here, you can see that I create my window using `entityManager->create<fender::entitied::Window>()`. Note that i'll take the **address of the reference** because i'm storing a pointer. You **cannot** have a reference as class member, because that would require initializing it in the init-list, and you **cannot** do that because entityManager is **nullptr**.

```
void Game::init()
{
  myWindow = &entityManager->create<fender::entities::Window>();
  auto &winComponent = myWindow->get<fender::components::Window>();
```

Note that i'm taking a reference to the component of myWindow of type `<fender::components::Window>`.

> **Never take a copy of the component**. This is a **common** mistake. **Always** take a reference or pointer to it.

```
void Game::init()
{
  myWindow = &entityManager->create<fender::entities::Window>();
  auto &winComponent = myWindow.get<fender::components::Window>();
  if (!winComponent.isOpen) {
    winComponent.visible = true;
    winComponent.title = "Snake - Fender";
    winComponent.size.w = 800; // in px
    winComponent.size.h = 600; // in px
  }
```

If the window isn't open already, then i'll set **visible** to **true** (the system will know it must render this window) and set other self-explanatory variables.

5.5 Creating a world

Its nice to have a window... but right now there's just nothing to show. What you need to do is create a **World** entity.

5.5.1 What's a World ?

The World Entity specifies two important things

- How big is it ?
- How many pixels per square ?

For our Snake game, we'll make the world a 16x16 square world. And following tilemap conventions, a square will be 32x32 pixels.

Therefore our Game systems will now look like this :

```
namespace Snake
{
   class Game : public futils::ISystem
   {
     fender::entities::Window *myWindow{nullptr};
     fender::entities::World *myWorld{nullptr};

     void init();
   public:
     Game() {
       name = "Game";
     }

     virtual ~Game() {}
     virtual void run(float);
   };
}
```

And the Init function will now init `myWorld`

```
myWorld = &entityManager->create<fender::entities::World>();
auto &worldComponent = myWorld->get<fender::components::World>();
worldComponent.name = "SnakeWorld";
```

(continues on next page)

(continued from previous page)

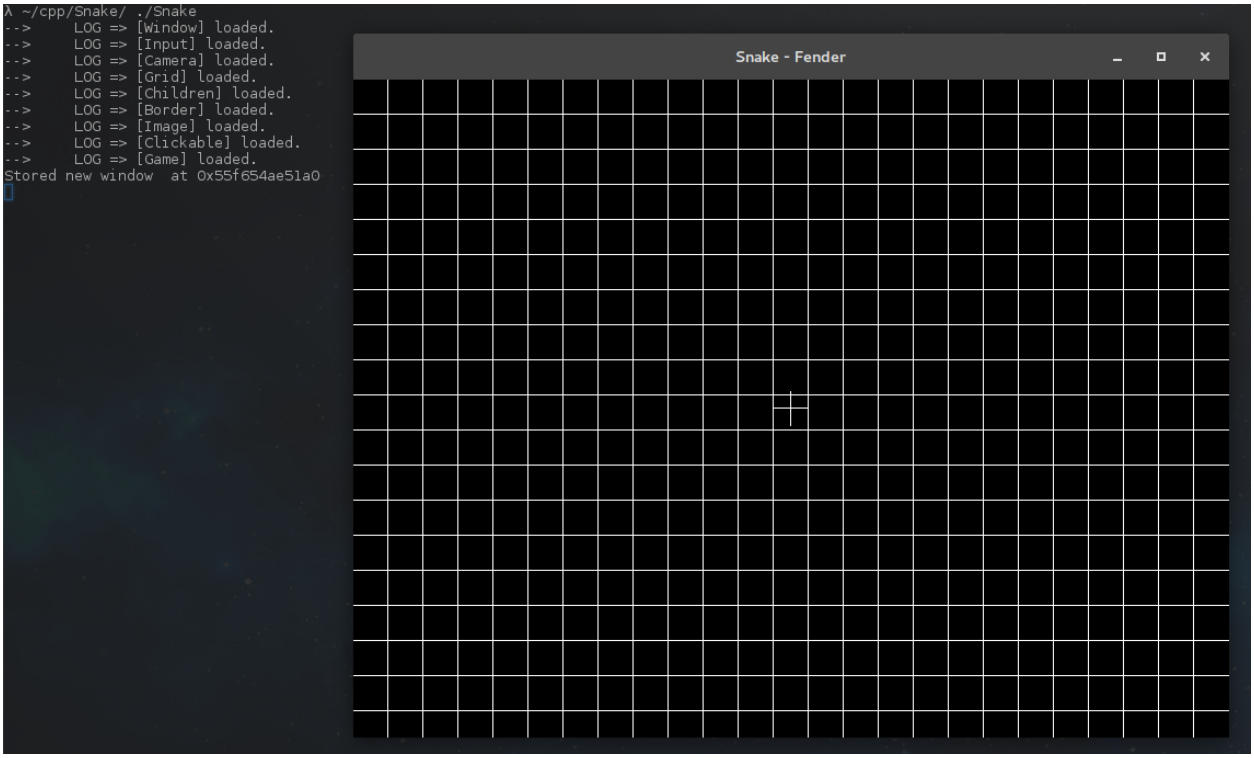
```
worldComponent.unit = 32;
worldComponent.size.w = 16;
worldComponent.size.h = 16;
```

As you can see, simply created the World entity, and initialized its World component.

5.6 Creating a Camera

We now have a Window and a World. In order to render anything to the screen, we'll now learn about the `Camera`.

5.6.1 Expected output



This step will allow you to see your world grid as well as a crosshair in the middle of the screen. This crosshair is equal to the Camera world position. The size of each square in the grid is equal to (World).unit pixels.

5.6.2 Updating Game

```
namespace Snake
{
  class Game : public futils::ISystem
  {
              fender::entities::Window *myWindow{nullptr};
              fender::entities::World *myWorld{nullptr};
              fender::entities::Camera *myCamera{nullptr};
```

(continues on next page)

(continued from previous page)

```
            void init();
  public:
            Game() {
              name = "Game";
            }
            virtual ~Game() {}
            virtual void run(float);
  };
}
```

[Game] now looks like this. As you can see we simply added a Camera entity.

5.6.3 Camera initialization

```
myCamera = &entityManager->create<fender::entities::Camera>();
auto &camComponent = myCamera->get<fender::components::Camera>();
camComponent.name = "SnakeCamera";
camComponent.window = myWindow;
camComponent.debugMode = true;
camComponent.activated = true;
camComponent.viewDistance = 10;
```

You should be familiar with this code by now.

If you do not **activate** your camera (`camComponent.activated = true`) it won't render anything. What you should understand is that the Camera is **the most important piece of the rendering process**.

Basically, the camera will follow these steps :

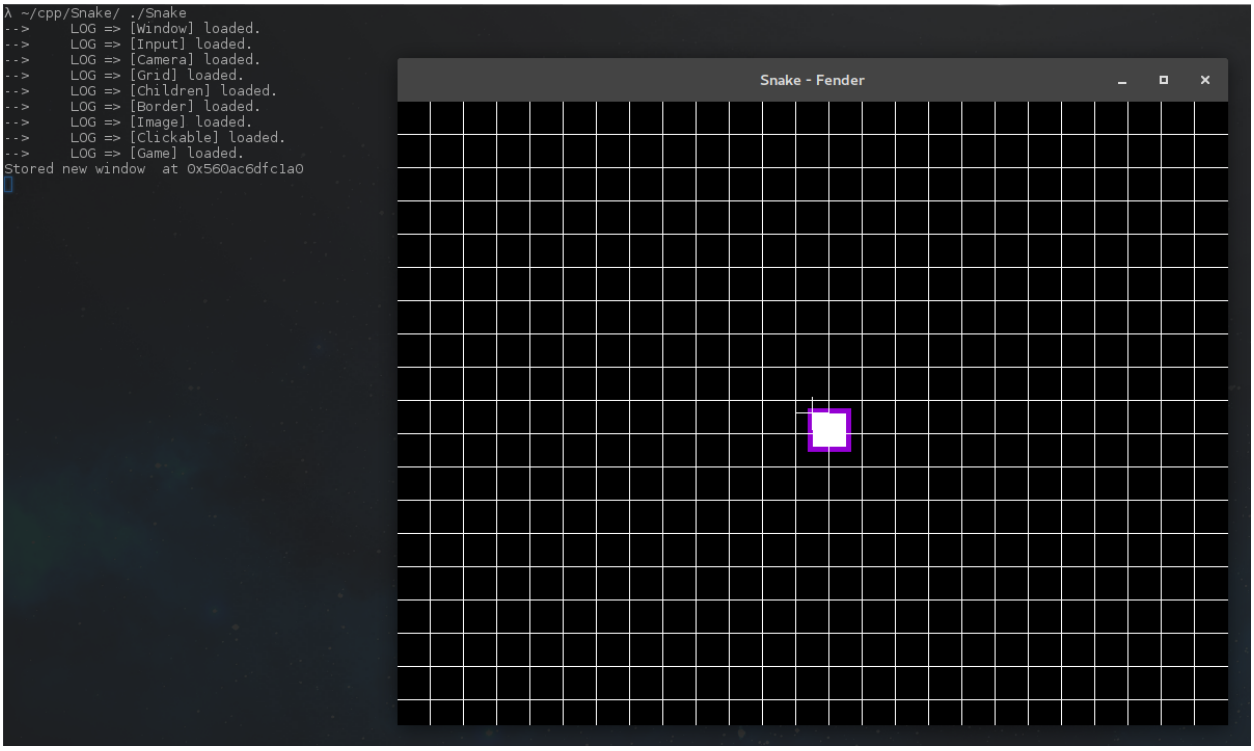
1. Sort all game objects by z-index
2. For each visible layer (between the camera position in z-axis and Camera.(Transform).position.z + Camera.(Camera).viewDistance) it will send an event `RenderLayer`
3. Rendering systems can `require<RenderLayer>` and access its objects to render them to the screen

Therefore, **you should never worry about rendering order, only about your objects world positions**.

5.7 Creating gameObjects

It's nice to have a Grid on the screen... but let's add gameObjects. Let's add a simple square for now.

5.7.1 Expected output



5.7.2 What is a GameObject ?

```
class GameObject : public futils::IEntity {
 public:
     GameObject() {
          attach<components::GameObject>();
          auto &transform = attach<components::Transform>();
          transform.position.z = 1;
          attach<components::AbsoluteTransform>();
          auto &border = attach<components::Border>();
          border.color = futils::Indianred;
          border.thickness = 2;
          border.visible = true;
     }
     ~GameObject() {
          detach<components::GameObject>();
          detach<components::Transform>();
          detach<components::AbsoluteTransform>();
     }
 };
```

A GameObject is an entity with a position and a border by default. It also has an AbsoluteTransform component, mostly used for rendering systems that the [Camera] will update.

> You should not modify (AbsoluteTransform) yourself.

5.7.3 Update [Game]

```
namespace Snake
{
  class Block : public fender::entities::GameObject
  {
  public:
     Block()
     {
          auto &transform = get<fender::components::Transform>();
          transform.size.w = 1;
          transform.size.h = 1;
          auto &border = get<fender::components::Border>();
          border.thickness = 5;
          border.visible = true;
          border.color = futils::Darkviolet;
     }
  };

  class Game : public futils::ISystem
  {
     fender::entities::Window *myWindow{nullptr};
     fender::entities::World *myWorld{nullptr};
     fender::entities::Camera *myCamera{nullptr};
     Block *myBlock{nullptr};

     void init();
  public:
     Game() {
          name = "Game";
     }
     virtual ~Game() {}
     virtual void run(float);
  };
}
```

And of course we'll simply create the entity in our init function :

```
myBlock = &entityManager->create<Block>();
```

That's it !. There's nothing more to do to output this gameObject. As you can see, creating it is enough, it will be displayed.

This is almost everything you need to know to render anything on the screen. Read the documentation of the entities, components and system to know what you can do. Congratulations, you now know the basics !

5.8 Handling inputs

Hum... You're right. It's all nice and well to render to the screen. But handling user input is just as important.

Check out the futils/inputKeys.hpp to know what events you can get.

5.8.1 The basics

This code right here shows how to react to basic user inputs.

```
addReaction<futils::Keys>([this](){
  auto &key = futils::Mediator::rebuild<futils::Keys>(pkg);
  if (key == futils::Keys::Escape)
    events->send<fender::events::Shutdown>();
});
```

It's that simple.

5.8.2 InputSequences (WIP)

You can also require events in a more advanced way.

```
input = &entityManager->create<fender::entities::Input>();
auto &component = input->get<fender::components::Input>();
component.name = "WindowTest";
component.activated = true;
futils::InputSequence escape;
futils::InputAction action(futils::Keys::Escape, futils::InputState::Down);
escape.actions.push_back(action);

futils::InputSequence generate;
futils::InputAction gen_action(futils::Keys::Space, futils::InputState::Down);
generate.actions.push_back(gen_action);

component.map[escape] = [this](){
            events->send<fender::events::Shutdown>();
};
component.map[generate] = [this](){
            createGo(*entityManager);
};
```

You can create Input (an Entity) and get its (Input) to set bind together InputSequence (a collection of InputAction) and a lambda. If you need more advanced input control, prefer this solution. You may want to swap between different configurations of inputs, so all you'd have to do is `send<InputSwitch>("WindowTest")`. # Not Implemented yet.

5.9 Network communications

CHAPTER 6

fender::Entities

Window	It opens a system window	Almost Done
World	It describes the world of your project	Done
Camera	It films your world	Done
Input	It describes how to react to user input	WIP
GameObject	Anything that can be rendered to the screen	Done
Image	A simple static image	Done
Text	A simple text	Done
Button	A simple clickable and hoverable button	X
ListView	A container that manages object placement of its contained elements	X
Music	Seriously?	X
Sprite	An animated image for advanced graphics	X
ProgressBar	A bar indicating progress	X
Stream	A box where text is automatically displayed in a readable way	X
InputField	An input containing user data	X
Slider	A point you can drag to set a value	X

CHAPTER 7

fender::Components

CHAPTER 8

fender::Systems

CHAPTER 9

Utils

fenfendeUtils is a collection of c++ header only solutions to recurring problems. They are all located in `futils` namespace.