

Dual State Framework

Design Documentation

Yu Chen - C00151352

2015/04/06 23:51:55

Contents

1	Introduction	1
2	Framwork Design	3
2.1	Architecture Design	3
2.2	Procedural Design	4
2.3	Interface Design	4
2.3.1	Namespace	4
2.3.2	dsf::Runnable	4
2.3.3	dsf::DualStateFramework	5
2.3.4	dsf::Synchronisable	5
2.3.5	dsf::TaskBox	5
2.3.6	dsf::SynchronizedObject	5
2.3.7	dsf::Task	6
2.3.8	dsf::Argument	6
2.3.9	dsf::TaskFunction	6
2.3.10	dsf::SynchronizedVar	6
3	Benchmark Program Design	7
3.1	GUI Framework	7
3.2	GUI Configuration	7
3.3	Outputs	8

Chapter 1

Introduction

This document gives an overall description of the project, and shows the design of the project that can be easier to figure out the change in design. The project is going to build C++ library, so a benchmark program is required.

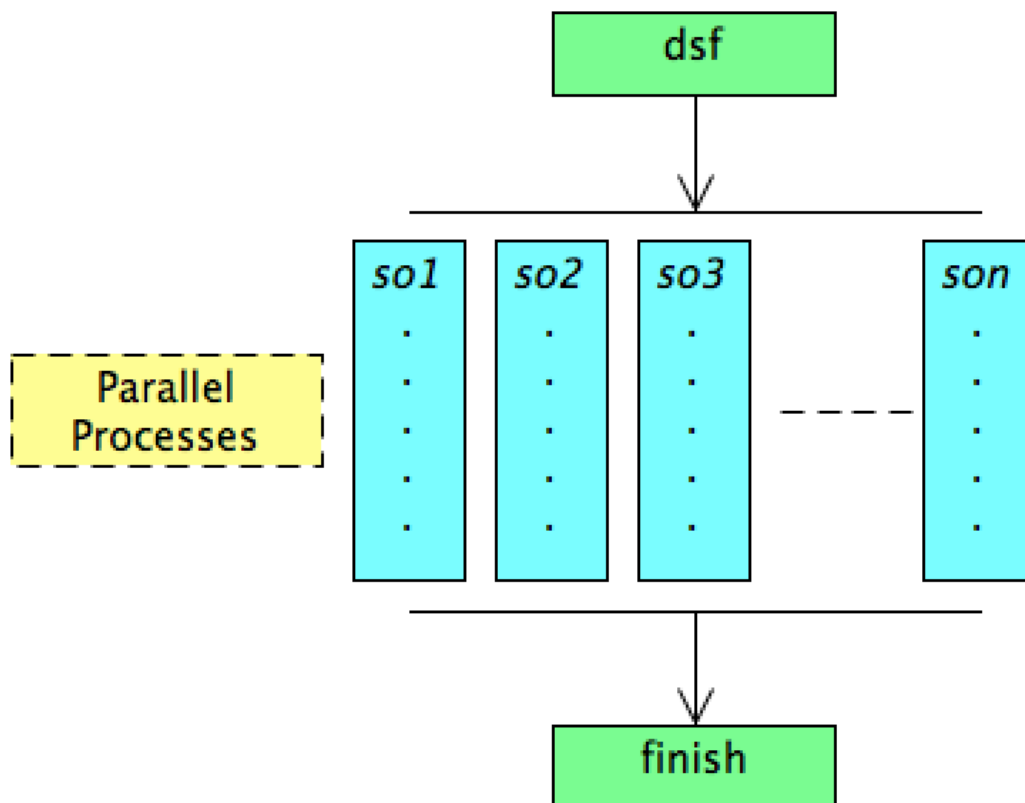
- Framework Design
- Benchmark Program Design

Chapter 2

Framwork Design

2.1 Architecture Design

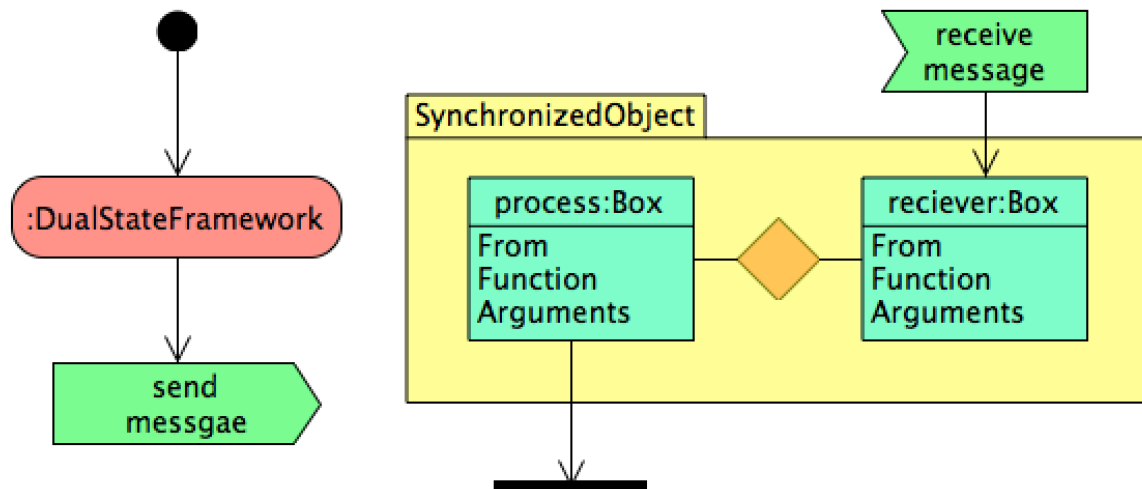
The goal of this project is to implement concurrency and parallel. In this framework, each Synchronized Object process simultaneously after Dual State Framework started. The process shows as follow:



- "dsf" refers to Dual State Framework
- "so" followed by a number refers to Synchronized Object

2.2 Procedural Design

This diagram shows activities and interactions in one success evaluation(one frame in game).



As it shows, the system create a `DualStateFramework` object. A `DualStateFramework` object can have many `SynchronizedObject` objects. Each `SynchronizedObject` object has two Boxes. One is for processing messages. The other one is for receiving messages. The sequence is like this:

1. The `DualStateFramework` object starts with sending a "start" message.
2. Messages from receiver are popped out and then be pushed into processor.
3. Processor unpacks messages and evaluates function with arguments in each message sequentially.

After step 3, it will go back to step 2. Step 2 and step 3 process alternatively and endlessly until the `DualStateFramework` object stopped.

2.3 Interface Design

2.3.1 Namespace

All classes in this framework is under the namespace "dsf", which is abbreviation of the project name "Dual State Framework".

2.3.2 dsf::Runnable

The interface provides a protected pure virtual function `run()`, which will be called when subclass object starts. For example:

```

class MyDSF : public dsf::DualStateFramework
{
protected:
    void run()
    {
        // do something here...
    }
}

void main()
{
    MyDSF dsf;
    dsf.start() // It will process MyDSF::run();
}
  
```


2.3.3 dsf::DualStateFramework

The starting pointer for the framework is the abstract class `dsf::DualStateFramework`. It provides essential functions for associating and managing its components (SynchronizedObject objects, function points, and etc.).

2.3.4 dsf::Synchronisable

Class `dsf::Synchronisable` is a template interface. It has a template member variable “next” and a pure virtual function “synchronise”. The variable is a copy of current class. The function synchronises the current state.

```
#include <dsf/Synchronisable.h>

class Vector3D
{
public:
    float x, y, z;
    Vector3D(float x, float y, float z) : x(x), y(y), z(z){}
};

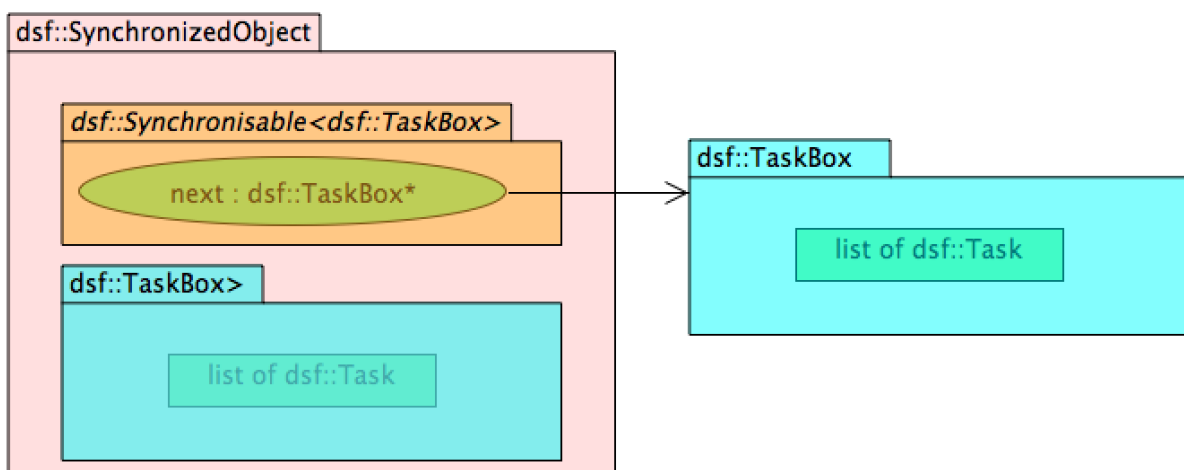
class SyncVector : public dsf::Synchronisable<Vector3D>, public Vector3D
{
public:
    SyncInt(float x, float y, float z) : Vector3D(x, y, z) {
        this->next = new Vector3D(x, y, z);
    }
    void synchronise() override {
        this->x = this->next->x;
        this->y = this->next->y;
        this->z = this->next->z;
    }
};
```

2.3.5 dsf::TaskBox

The `dsf::TaskBox` contains a list of `dsf::Task` objects. It provides essential methods to control the list such as “push”, “pop”, and “isEmpty”.

2.3.6 dsf::SynchronizedObject

The `dsf::SynchronizedObject` is a subclass of `dsf::TaskBox`. In this framework, you can regard it as thread. It provides methods for implementing parallelism such as “send”, “receive”, and etc. Also, it implements interface `dsf::Synchronisable<dsf::TaskBox>`. Therefore, it has two copies of `dsf::TaskBox`. One is for receiving messages, and the other one is for executing messages.



2.3.7 dsf::Task

This class have three members: from, function, and arguments, where "from" is a dsf::SynchronizedObject object who sent message to you.

2.3.8 dsf::Argument

A dsf::Argument object can refer to any type of object. The following code is allowed:

```
dsf::Argument i = int(10);
dsf::Argument str = std::string("Hello");
```

2.3.9 dsf::TaskFunction

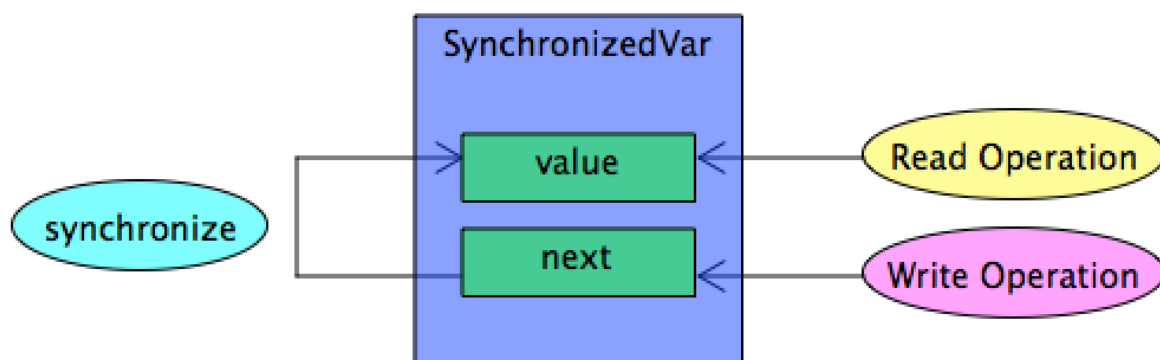
The message in a dsf::Task is an object of dsf::TaskFunction. An object of dsf::TaskFunction is a function pointer whose function takes two dsf::SynchronizedObject objects, a dsf::Argument object as arguments. The constructor of dsf::TaskFunction takes a function pointer or a lambda expression as arguments. This is an example that uses lambda expression.

```
dsf::TaskFunction* print = new dsf::TaskFunction([this](dsf::SynchronizedObject* to,
                                                       dsf::SynchronizedObject* from,
                                                       dsf::TaskArgument* args)
{
    auto syncObj = args->to<SyncObj*>();
    std::cout << syncObj->getValue();
    this->dsf->remove(to);
});
```

2.3.10 dsf::SynchronizedVar

The purpose of this class is to make thread-safe variables for dsf::SynchronizedObject objects. A dsf::SynchronizedVar object has two member variables - "value" and "next". The "value" is for read operation, and the "next" is for write operation. The function "synchronize" signs "next" to "value". see example below:

```
dsf::SynchronizedVar myInt;
myInt = int(8); // value == NULL, next == 8
myInt.synchronise(); // value == 8, next == 8
std::cout << myInt.to<int>() << std::endl; // output 8
myInt = int(9); // value == 8, next == 9
std::cout << myInt.to<int>() << std::endl; // output 8
myInt.synchronise(); // value == 9, next == 9
std::cout << myInt.to<int>() << std::endl; // output 9
```



Chapter 3

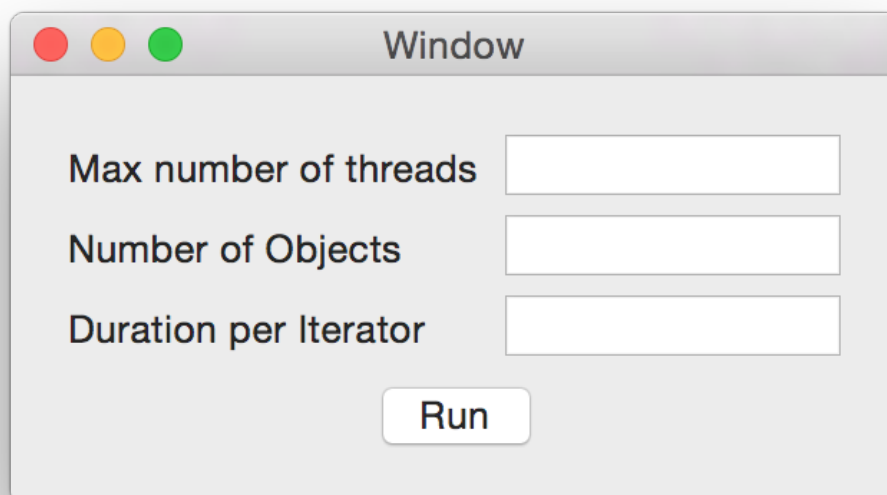
Benchmark Program Design

3.1 GUI Framework

To benchmark the framework, I will use SFML to create a graphics application. When running the application, each frame runs many objects. The maximum, minimum, and average FPS(Frame per Second) is what I need for the benchmark. The application allows using different number of threads to run it.

3.2 GUI Configuration

When you start the application, it will show you a configuration form like this:



The image shows a graphical user interface window titled "Window". Inside the window, there are three text labels on the left, each followed by a rectangular input field on the right. The labels are "Max number of threads", "Number of Objects", and "Duration per Iterator". Below these input fields, centered at the bottom of the window, is a button labeled "Run". The window has a standard macOS-style title bar with three colored buttons (red, yellow, green) on the left.

The application will run by using one thread to the max number of threads that user configured. The maximum, minimum, and average FPS will be recorded each iterator. Number of objects and duration for each iterator can be

configured by this form.

3.3 Outputs

The output is a bar graph which shows all information of the benchmark.

- x axis representing number of threads
- y axis representing frames per second(FPS)
- values with the maximum, the minimum, and the average.

