Dual State Framework

CODE

Yu Chen - C00151352

2015/04/15 17:53:03

Contents

1 | Dual State Framework

1.1 CMakeLists

1.1.1 CMakeLists.txt

Path: *$PROJECT_DIR/CMakeLists.txt*

```
cmake_minimum_required(VERSION 2.8)

# project name
project(dsf)

# setup version numbers
set(VERSION_MAJOR 1)
set(VERSION_MINOR 0)
set(VERSION_PATCH 0)


# disable the rpath stuff
set(CMAKE_SKIP_BUILD_RPATH FALSE)


# detect the architecture (note: this test won't work for cross-compilation)

include(CheckTypeSize)
check_type_size(void* SIZEOF_VOID_PTR)
if("${SIZEOF_VOID_PTR}" STREQUAL "4")
    set(ARCH x86)
elseif("${SIZEOF_VOID_PTR}" STREQUAL "8")
    set(ARCH x64)
else()
    message(FATAL_ERROR "Unsupported architecture")
    return()
endif()

# configure links and headers

set(LIBS tbb yctools)

if (MSVC)
    # Windows VC
    # Activate C++ exception handling
    if (NOT CMAKE_CXX_FLAGS MATCHES "/EHsc")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /EHsc")
    endif ()

    # Set Warning level always to 4
    if (CMAKE_CXX_FLAGS MATCHES "/W[0-4]")
        string(REGEX REPLACE "/W[0-4]" "/W4" CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}")
    else ()
        set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /W4")
    endif ()

    set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-msvc/${ARCH}/libs)
elseif(APPLE)
    # Mac OS X Xcode
    set(CMAKE_MACOSX_RPATH 1)
    set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-osx)
    OPTION(OSX_FRAMEWORK "Build a Mac OS X Framework")
    ADD_DEFINITIONS(-std=c++11)
else()
    # Unix
    set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-unix/${ARCH})
    ADD_DEFINITIONS(-std=c++11)
endif()
```

```

# Source
set(INCROOT ${CMAKE_CURRENT_SOURCE_DIR}/dsf/include)
set(SRCROOT ${CMAKE_CURRENT_SOURCE_DIR}/dsf/src)
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/dsf)

# add the header path
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/extlibs/headers)
include_directories(${INCROOT})

# output
add_library (${PROJECT_NAME} SHARED ${DSF_INC} ${DSF_SRC})
set_target_properties(
    ${PROJECT_NAME} PROPERTIES
    ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
    LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
    RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
)

if(OSX_FRAMEWORK)
    set_target_properties(
        ${PROJECT_NAME} PROPERTIES
        FRAMEWORK TRUE
        FRAMEWORK_VERSION ${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}
        MACOSX_FRAMEWORK_IDENTIFIER ie.itcarlow.yuchen.dsf
        MACOSX_FRAMEWORK_SHORT_VERSION_STRING ${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}
        MACOSX_FRAMEWORK_BUNDLE_VERSION ${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}
        PUBLIC_HEADER "${DSF_INC}"
    )
endif()

foreach(LIB ${LIBS})
    find_library(LIB_${LIB} NAMES ${LIB} PATHS ${LIBS_DIR})
    set_source_files_properties(${LIB_${LIB}} PROPERTIES MACOSX_PACKAGE_LOCATION Frameworks)
    target_link_libraries(${PROJECT_NAME} ${LIB_${LIB}})
endforeach()
```

1.1.2 dsf/CMakeLists.txt

Path: *$PROJECT_DIR/dsf/CMakeLists.txt*

```
# source files
set(DSF_SRC
    ${SRCROOT}/dsf/DualStateFramework.cpp
    ${SRCROOT}/dsf/Lock.cpp
    ${SRCROOT}/dsf/SynchronizedObject.cpp
    ${SRCROOT}/dsf/SynchronizedVar.cpp
    ${SRCROOT}/dsf/Task.cpp
    ${SRCROOT}/dsf/TaskBox.cpp
    PARENT_SCOPE
)
source_group("" FILES ${DSF_SRC})

# headers

set(DSF_INC
    ${INCROOT}/dsf/Config.h
    ${INCROOT}/dsf/Declaration.h
    ${INCROOT}/dsf/DualStateFramework.h
    ${INCROOT}/dsf/Export.h
    ${INCROOT}/dsf/Lock.h
    ${INCROOT}/dsf/Runnable.h
    ${INCROOT}/dsf/SynchronizedObject.h
    ${INCROOT}/dsf/SynchronizedVar.h
    ${INCROOT}/dsf/Synchronisable.h
    ${INCROOT}/dsf/Task.h
    ${INCROOT}/dsf/TaskArgument.h
    ${INCROOT}/dsf/TaskBox.h
    ${INCROOT}/dsf/TaskFunction.h
    PARENT_SCOPE
)
source_group("" FILES ${DSF_INC})

```

1.2 Config

1.2.1 Config.h

Path: *$PROJECT_DIR/dsf/include/dsf/Config.h*

```
//
//  config.h
//  DualStateFramework
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef dsf_Config_h
#define dsf_Config_h

#if defined(_WIN32)

// Windows compilers need specific (and different) keywords for export and import
#define DSF_API_EXPORT __declspec(dllexport)
#define DSF_API_IMPORT __declspec(dllimport)

// For Visual C++ compilers, we also need to turn off this annoying C4251 warning
#ifdef _MSC_VER

#pragma warning(disable : 4251)

#endif

#else // Linux, FreeBSD, Mac OS X

#if __GNUC__ >= 4

// GCC 4 has special keywords for showing/hidding symbols,
// the same keyword is used for both importing and exporting
#define DSF_API_EXPORT __attribute__ ((__visibility__ ("default")))
#define DSF_API_IMPORT __attribute__ ((__visibility__ ("default")))

#else

// GCC < 4 has no mechanism to explicitely hide symbols, everything's exported
#define DSF_API_EXPORT
#define DSF_API_IMPORT

#endif

#endif


#endif
```

1.3 Declaration

1.3.1 Declaration.h

Path: *$PROJECT_DIR/dsf/include/dsf/Declaration.h*

```
//
//  Declaration.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_Declaration_h
#define dsf_Declaration_h

#include "Export.h"

namespace dsf {
    class DSF_API Task;
    class DSF_API TaskBox;
    class DSF_API DualStateFramework;
    class DSF_API SynchronizedObject;
}

#endif
```

1.4 DualStateFramework

1.4.1 DualStateFramework.h

Path: *$PROJECT_DIR/dsf/include/dsf/DualStateFramework.h*

```
//
//  DualStateFramework.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_DualStateFramework_h
#define dsf_DualStateFramework_h

#include <vector>

#include "TaskBox.h"
#include "Task.h"
#include "SynchronizedObject.h"
#include "Runnable.h"

namespace dsf {
    class DSF_API DualStateFramework : public Runnable
    {
    public:
        DualStateFramework();

        ~DualStateFramework();

        virtual void initialize() = 0;

        void start();

        void doOneFrame();



        void add(SynchronizedObject* syncObj);

        void remove(SynchronizedObject* syncObj);

        void send(SynchronizedObject* to,
                  SynchronizedObject* from,
                  TaskFunction* taskFunction,
                  TaskArgument* args);

        State getState() override;

        void setNumberOfThreads(int NumberOfThreads);

    private:
        int NumberOfThreads;
        std::vector<SynchronizedObject*>* syncObjs;
        State state;
    protected:
        virtual void refresh();
         virtual void run() override;
    };
}

#endif
```

1.4.2 DualStateFramework.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/DualStateFramework.cpp*

```
//
//  TaskManager.cpp
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#include <tbb/parallel_for_each.h>
#include <dsf/DualStateFramework.h>
```

```
#include <tbb/task_scheduler_init.h>
#include <algorithm>

namespace dsf
{
    DualStateFramework::DualStateFramework()
    {
        this->NumberOfThreads = tbb::task_scheduler_init::automatic;
        this->syncObjs = new std::vector<SynchronizedObject*>();
        this->state = DualStateFramework::State::STOPPED;
    }

    DualStateFramework::~DualStateFramework()
    {
        while (!this->syncObjs->empty()) {
            SynchronizedObject* syncObj = this->syncObjs->back();
            this->syncObjs->pop_back();
            delete syncObj;
        }
        delete this->syncObjs;
    }

    void DualStateFramework::start()
    {
        if(!this->syncObjs->empty())
        {
            this->doOneFrame();
            this->start();
        }
    }

    void DualStateFramework::doOneFrame()
    {
        this->refresh();

        if(!this->syncObjs->empty())
        {
            this->state = DualStateFramework::State::RUNNING;
            this->run();
            if(this->state == DualStateFramework::State::RUNNING)
                this->state = DualStateFramework::State::STOPPED;
        }
    }


    void DualStateFramework::add(dsf::SynchronizedObject *syncObj)
    {
        this->syncObjs->push_back(syncObj);
    }

    void DualStateFramework::remove(dsf::SynchronizedObject *syncObj)
    {
        syncObj->distroy();
    }

    void DualStateFramework::send(dsf::SynchronizedObject *to, dsf::SynchronizedObject *from, TaskFunction
   *taskFunction, TaskArgument *args)
    {
        from->send(to, taskFunction, args);
    }

    DualStateFramework::State DualStateFramework::getState()
    {
        return this->state;
    }

    void DualStateFramework::setNumberOfThreads(int NumberOfThreads)
    {
        if(NumberOfThreads == 0)
            this->NumberOfThreads = tbb::task_scheduler_init::automatic;
        else
            this->NumberOfThreads = NumberOfThreads;
    }

    // Private


    // Protected


    void DualStateFramework::refresh()
    {
        this->syncObjs->erase(std::remove_if(this->syncObjs->begin(),
                                             this->syncObjs->end(),
                                             [](SynchronizedObject* sb)
                                             {
                                                 if (sb->getState() == SynchronizedObject::State::DELETED)
    {
                                                    delete sb;
                                                    return true;
                                                }
                                                return false;
                                            }) ,
                              this->syncObjs->end());
    }

    void DualStateFramework::run()
    {
        tbb::task_scheduler_init init(this->NumberOfThreads);
        tbb::parallel_for_each(this->syncObjs->begin(),
                               this->syncObjs->end(),
                               [](SynchronizedObject* sb)
                               {
                                   if(sb->getState() == SynchronizedObject::State::STOPPED)
                                   {
                                       sb->start();
                                   }
                               });
    }
}
```

1.5 Export

1.5.1 Export.h

Path: *$PROJECT_DIR/dsf/include/dsf/Export.h*

```
//
//  Export.h
//  DualStateFramework
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef dsf_Export_h
#define dsf_Export_h

// Headers
#include "Config.h"


// Define portable import / export macros
#if defined(dsf_EXPORTS)

#define DSF_API DSF_API_EXPORT

#else

#define DSF_API DSF_API_IMPORT

#endif

#endif
```

1.6 Lock

1.6.1 Lock.h

Path: *$PROJECT_DIR/dsf/include/dsf/Lock.h*

```
//
//  Lockable.h
//  dsf
//
//  Created by Yu Chen on 12/10/14.
//
//

#ifndef dsf_Lock_h
#define dsf_Lock_h
```

```

#include <mutex>
#include "Export.h"

namespace dsf
{
    class DSF_API Lock
    {
    protected:
        std::mutex locker;
    public:
        void lock();
        void unlock();
    };
}

#endif
```

1.6.2 Lock.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/Lock.cpp*

```
//
//  Lock.cpp
//  dsf
//
//  Created by Yu Chen on 12/10/14.
//
//

#include <dsf/Lock.h>

namespace dsf
{
    void Lock::lock()
    {
        this->locker.lock();
    }

    void Lock::unlock()
    {
        this->locker.unlock();
    }
}
```

1.7 Runnable

1.7.1 Runnable.h

Path: *$PROJECT_DIR/dsf/include/dsf/Runnable.h*

```
//
//  Runnable.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_Runnable_h
#define dsf_Runnable_h

namespace dsf
{
    class Runnable
    {
    public:
        enum State
        {
            RUNNING, STOPPED, READY, DELETED
        };
        virtual State getState() = 0;
    protected:
        virtual void run() = 0;
    };
}

#endif
```

1.8 Synchronisable

1.8.1 Synchronisable.h

Path: *$PROJECT_DIR/dsf/include/dsf/Synchronisable.h*

```
//
//  Synchronisable.h
//  dsf
//
//  Created by Yu Chen on 3/24/15.
//
//

#ifndef dsf_Synchronisable_h
#define dsf_Synchronisable_h

namespace dsf {
    template<class T> class Synchronisable
    {
    protected:
        T* next;
    public:
        virtual ~Synchronisable() {
            delete this->next;
        }
        virtual void synchronise() = 0;
    };
}

#endif
```

1.9 SynchronizedObject

1.9.1 SynchronizedObject.h

Path: *$PROJECT_DIR/dsf/include/dsf/SynchronizedObject.h*

```
//
//  SynchronizedObject.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_SynchronizedObject_h
#define dsf_SynchronizedObject_h

#include <vector>
#include <string>

#include "Export.h"
#include "Declaration.h"
#include "Task.h"
#include "TaskBox.h"
#include "TaskFunction.h"
#include "TaskArgument.h"
#include "Synchronisable.h"
#include "Lock.h"
#include "Runnable.h"

namespace dsf
{
    class DSF_API SynchronizedObject : public Runnable, public TaskBox, public Synchronisable<TaskBox>, public Lock
    {
    public:
        SynchronizedObject();
        virtual ~SynchronizedObject();
        State getState() override;
        int receive();
    private:
        State state;

        void push(Task* task);
        void send(SynchronizedObject* to,
                  TaskFunction* taskFunction,
                  TaskArgument* args);
        void start();
        void stop();
        void distroy();
        friend class DualStateFramework;
    protected:
        virtual void run() override = 0;
        void synchronise() override;
    };
}

#endif
```

1.9.2 SynchronizedObject.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/SynchronizedObject.cpp*

```
//
//  SynchronizedObject.cpp
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#include <dsf/SynchronizedObject.h>

namespace dsf
{
    SynchronizedObject::SynchronizedObject()
        : TaskBox()
    {
        this->next = new TaskBox();
        this->state = State::STOPPED;
    }
    SynchronizedObject::~SynchronizedObject()
    {
    }

    SynchronizedObject::State SynchronizedObject::getState()
    {
        return this->state;
    }


    int SynchronizedObject::receive()
    {
        int count = 0;
        while (!this->next->isEmpty())
        {
            this->tasks->push_back(this->next->pop());
            count ++;
        }
        return count;
    }

    void SynchronizedObject::push(dsf::Task *task)
    {
        this->lock();
        this->next->push(task);
        this->unlock();
    }

    // Privates

    void SynchronizedObject::send(dsf::SynchronizedObject *to,
                                  TaskFunction *taskFunction,
                                  TaskArgument *args)
    {
        to->push(new Task(to, this, taskFunction, args));
    }


    void SynchronizedObject::start()
    {
        this->state = State::RUNNING;
        this->run();
        if(this->state == State::RUNNING)
            this->stop();
    }

    void SynchronizedObject::stop()
    {
        this->state = State::STOPPED;
    }

    void SynchronizedObject::distroy()
    {
        this->state = State::DELETED;
    }



    // Protected

    void SynchronizedObject::synchronise()
    {
        *((TaskBox*) this) = *this->next;
    }
}
```

1.10 SynchronizedVar

1.10.1 SynchronizedVar.h

Path: *$PROJECT_DIR/dsf/include/dsf/SynchronizedVar.h*

```
//
//  SynchronizedVar.h
//  dsf
//
//  Created by Yu Chen on 12/2/14.
//
//

#ifndef dsf_SynchronizedVar_h
#define dsf_SynchronizedVar_h

#include "Export.h"
#include <yctools/Any.h>
#include "Synchronisable.h"
#include "Lock.h"

namespace dsf
{
    class DSF_API SynchronizedVar : public Synchronisable<yc::Any>, public yc::Any, public Lock
    {
    public:
        template<typename T> SynchronizedVar(T && value);
        template<typename T> void operator=(T && value);
        void synchronise() override;
    };
}

namespace dsf
{

    template<typename T> SynchronizedVar::SynchronizedVar(T && value)
    : Any(value)
    {
        this->next = new Any(value);
    }

    template<typename T> void SynchronizedVar::operator=(T && value)
    {
        this->lock();
        this->next = value;
        this->unlock();
    }
}


#endif
```

1.10.2 SynchronizedVar.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/SynchronizedVar.cpp*

```cpp
//
//  SynchronizedVar.cpp
//  dsf
//
//  Created by Yu Chen on 12/2/14.
//
//

#include <dsf/SynchronizedVar.h>

namespace dsf
{
    void SynchronizedVar::synchronise()
    {
        *((yc::Any*)this) = *this->next;
    }
}

```

1.11 Task

1.11.1 Task.h

Path: *$PROJECT_DIR/dsf/include/dsf/Task.h*

```cpp
//
//  Parallel.h
//  DualStateFramework
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef dsf_Task_h
#define dsf_Task_h

#include <vector>
#include <memory>

#include "Export.h"
#include "Declaration.h"
#include "TaskBox.h"
#include "TaskFunction.h"
#include "TaskArgument.h"


namespace dsf
{
    class DSF_API Task
    {
    public:
        SynchronizedObject* to;
        SynchronizedObject* from;
        TaskFunction* taskFunction;
        TaskArgument* taskArgument;
        explicit Task(SynchronizedObject* to,
             SynchronizedObject* from,
             TaskFunction* taskFunction,
             TaskArgument* taskArgument);
        ~Task();
    };
}

#endif
```

1.11.2 Task.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/Task.cpp*

```cpp
//
//  Parallel.cpp
//  DualStateFramework
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

```

```
#include <dsf/Task.h>
namespace dsf
{
    Task::Task(SynchronizedObject* to,
               SynchronizedObject* from,
               TaskFunction* taskFunction,
               TaskArgument* taskArgument)
    {
        this->to = to;
        this->from = from;
        this->taskFunction = taskFunction;
        this->taskArgument = taskArgument;
    }

    Task::~Task()
    {
        delete this->taskArgument;
    }
}
```

1.12 TaskArgument

1.12.1 TaskArgument.h

Path: *$PROJECT_DIR/dsf/include/dsf/TaskArgument.h*

```
//
//  TaskArguments.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_TaskArgument_h
#define dsf_TaskArgument_h

#include <vector>

#include <yctools/Any.h>
#include <yctools/AnyException.h>

namespace dsf
{
    typedef yc::Any TaskArgument;
    typedef yc::Exception::AnyException TaskArgumentException;
}




#endif
```

1.13 TaskBox

1.13.1 TaskBox.h

Path: *$PROJECT_DIR/dsf/include/dsf/TaskBox.h*

```
//
//  Parallelable.h
//  DSF
//
//  Created by Yu Chen on 10/14/14.
//
//

#ifndef dsf_TaskBox_h
#define dsf_TaskBox_h

#include <vector>
#include <string>

#include "Export.h"
```

```
#include "Declaration.h"
#include "Task.h"
#include "Runnable.h"

namespace dsf
{
    class DSF_API TaskBox
    {
    public:
        TaskBox();
        virtual ~TaskBox();
        bool isEmpty();
        void push(Task* task);
        Task* pop();
        void process();
    protected:
        std::vector<Task*>* tasks;
    };
}
#endif
```

1.13.2 TaskBox.cpp

Path: *$PROJECT_DIR/dsf/src/dsf/TaskBox.cpp*

```
//
//  TaskBox.cpp
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#include <dsf/TaskBox.h>

namespace dsf
{
    TaskBox::TaskBox()
    {
        this->tasks = new std::vector<Task*>();
    }

    TaskBox::~TaskBox()
    {
        for (std::vector<Task*>::iterator i = this->tasks->begin(); i != this->tasks->end(); ++i)
        {
            delete *i;
        }
        delete this->tasks;
    }

    bool TaskBox::isEmpty()
    {
        return this->tasks->empty();
    }

    void TaskBox::push(dsf::Task *task)
    {
        this->tasks->push_back(task);
    }

    Task* TaskBox::pop()
    {
        Task* task = this->tasks->back();
        this->tasks->pop_back();
        return task;
    }

    void TaskBox::process()
    {
        while (!this->isEmpty())
        {
            Task* task = this->pop();
            (*task->taskFunction)(task->to, task->from, task->taskArgument);
            delete task;
        }
    }
}
```

1.14 TaskFunction

1.14.1 TaskFunction.h

Path: *$PROJECT_DIR/dsf/include/dsf/TaskFunction.h*

```
//
//  TaskFunction.h
//  dsf
//
//  Created by Yu Chen on 10/17/14.
//
//

#ifndef dsf_TaskFunction_h
#define dsf_TaskFunction_h

#include <functional>
#include <vector>

#include "TaskArgument.h"

namespace dsf
{
    typedef std::function<void (dsf::SynchronizedObject*, dsf::SynchronizedObject*, TaskArgument*)>
   TaskFunction;
    typedef void function;
}

#endif
```

2 | yctools

2.1 Any

2.1.1 Any.h

Path: *$PROJECT_DIR/dsf/include/yctools/Any.h*

```cpp
//
//  Any.h
//  YCTools
//
//  Created by Yu Chen on 11/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef YCTools_Any_h
#define YCTools_Any_h

#include <memory>
#include <typeindex>
#include <string>

#include "Export.h"
#include "AnyException.h"

namespace yc
{
    class YCTOOLS_API Any
    {
    public:
        Any(void);
        Any(Any& that);
        Any(Any && that);
        template<typename T> Any(T && value);

        bool isNull() const;
        template<class T> bool is() const;
        template<class T> typename std::decay<T>::type& to();

        template<class T> operator T() const;
        Any& operator=(const Any& any);

    private:
        class Base;
        template<typename T> class Derived;
        std::unique_ptr<Base> Clone() const;
        std::unique_ptr<Base> pointer;
        std::type_index typeIndex;
    };
}


//  Class yc::Any

namespace yc
{

    template<typename T> Any::Any(T && value) :
        pointer(new Derived <typename std::decay<T>::type> (std::forward<T>(value))),
        typeIndex(std::type_index(typeid(typename std::decay<T>::type)))
    {}

    template<class T> bool Any::is() const
    {
        return typeIndex == std::type_index(typeid(T));
    }

    template<class T> typename std::decay<T>::type& Any::to()
    {
        if (!this->is<T>())
        {
            throw Exception::AnyException(typeid(T).name(), typeIndex.name());
        }

        typedef typename std::decay<T>::type U;
        auto derived = static_cast<Derived<U>*> (pointer.get());
        return derived->value;
    }
    template<class T> Any::operator T() const
    {
        return this->to<typename std::decay<T>::type>();
    }

}

// Inner Classes

namespace yc {

    class YCTOOLS_API Any::Base
    {
        public:
        virtual ~Base();
        virtual std::unique_ptr<Base> Clone() const = 0;
    };
    template<typename T>
    class Any::Derived : public Any::Base
    {
    public:
        template<typename U>
        Derived(U && value) : value(std::forward<U>(value)) { }

        std::unique_ptr<Base> Clone() const
        {
             return std::unique_ptr<Base>(new Derived<T>(this->value));
         }

         T value;
     };
 }



 #endif
```

2.1.2 Any.cpp

Path: *$PROJECT_DIR/dsf/src/yctools/Any.cpp*

```
//
//  Any.cpp
//  YCTools
//
//  Created by Yu Chen on 11/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#include "../../include/yctools/Any.h"

namespace yc
{
    Any::Any() :
    typeIndex(std::type_index(typeid(void) ))
    {}

    Any::Any(Any& that) :
    pointer(that.Clone()),
    typeIndex(that.typeIndex)
    {}

    Any::Any(Any && that) :
    pointer(std::move(that.pointer)),
    typeIndex(that.typeIndex)
    {}

    bool Any::isNull() const
    {
        return !bool(pointer);
    }

    Any& Any::operator=(const Any& any)
    {
        if (this->pointer == any.pointer)
            return *this;

        this->pointer = any.Clone();
        this->typeIndex = any.typeIndex;
        return *this;
    }

    Any::Base::~Base() {}

    std::unique_ptr<Any::Base> Any::Clone() const
    {
        if (pointer != nullptr)
            return pointer->Clone();

        return nullptr;
    }
}
```

2.2 AnyException

2.2.1 AnyException.h

Path: *$PROJECT_DIR/dsf/include/yctools/AnyException.h*

```
//
//  AnyException.h
//  YCTools
//
//  Created by Yu Chen on 11/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef YCTools_AnyException_h
#define YCTools_AnyException_h

#include <exception>
#include <string>

#include "Export.h"
namespace yc
{
    namespace Exception
    {
        class YCTOOLS_API AnyException : public std::exception
        {
        public:
            AnyException(std::string from,
                         std::string to);
            virtual const char* what() const throw() override;
            std::string details() const throw();
        private:
            std::string from;
            std::string to;
        };
    }

}
#endif
```

2.2.2 AnyException.cpp

Path: *$PROJECT_DIR/dsf/src/yctools/AnyException.cpp*

```
//
//  AnyException.cpp
//  YCTools
//
//  Created by Yu Chen on 11/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#include "../../include/yctools/AnyException.h"
```

```

namespace yc
{
    namespace Exception
    {

        AnyException::AnyException(std::string from,
                                   std::string to) :
            from(from),
            to(to)
        {}

        const char* AnyException::what() const throw()
        {
            return "\"Any\" exception happened!";
        }
        std::string AnyException::details() const throw()
        {
            return "Can not cast " + from + " to " + to;
        }
    }
}
```

2.3 Config

2.3.1 Config.h

Path: *$PROJECT_DIR/dsf/include/yctools/Config.h*

```
//
//  config.h
//  YCTOOLS
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef yctools_Config_h
#define yctools_Config_h

#if defined(_WIN32)

// Windows compilers need specific (and different) keywords for export and import
#define YCTOOLS_API_EXPORT __declspec(dllexport)
#define YCTOOLS_API_IMPORT __declspec(dllimport)

// For Visual C++ compilers, we also need to turn off this annoying C4251 warning
#ifdef _MSC_VER

#pragma warning(disable : 4251)

#endif

#else // Linux, FreeBSD, Mac OS X

#if __GNUC__ >= 4

// GCC 4 has special keywords for showing/hidding symbols,
// the same keyword is used for both importing and exporting
#define YCTOOLS_API_EXPORT __attribute__ ((__visibility__ ("default")))
#define YCTOOLS_API_IMPORT __attribute__ ((__visibility__ ("default")))

#else

// GCC < 4 has no mechanism to explicitely hide symbols, everything's exported
#define YCTOOLS_API_EXPORT
#define YCTOOLS_API_IMPORT

#endif

#endif


#endif
```

2.4 Export

2.4.1 Export.h

Path: *$PROJECT_DIR/dsf/include/yctools/Export.h*

```
//
//  Export.h
//  YCTOOLS
//
//  Created by Yu Chen on 10/13/14.
//  Copyright (c) 2014 Yu Chen. All rights reserved.
//

#ifndef yctools_Export_h
#define yctools_Export_h

// Headers
#include "Config.h"


// Define portable import / export macros
#if defined(yctools_EXPORTS)

#define YCTOOLS_API YCTOOLS_API_EXPORT

#else

#define YCTOOLS_API YCTOOLS_API_IMPORT

#endif

#endif
```

2.5 Random

2.5.1 Random.h

Path: *$PROJECT_DIR/dsf/include/yctools/Random.h*

```
//
//  Random.h
//  yctools
//
//  Created by Yu Chen on 2/9/15.
//
//

#ifndef yctools_Random_h
#define yctools_Random_h

#include <stdlib.h>
#include <ctime>
#include <random>

#include "Export.h"
namespace yc
{
    class YCTOOLS_API Random
    {
    public:
        int randInt(int min, int max);
        float randFloat(float min, float max);
    };
}

#endif
```

2.5.2 Random.cpp

Path: *$PROJECT_DIR/dsf/src/yctools/Random.cpp*

```
//
//  Random.cpp
//  yctools
//
//  Created by Yu Chen on 2/9/15.
```

```cpp
//
//


#include <yctools/Random.h>

int yc::Random::randInt(int min, int max)
{
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_int_distribution<int> distribution(min, max);
    int dice_roll = distribution(generator);
    return dice_roll;
}

float yc::Random::randFloat(float min, float max)
{
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_real_distribution<float> distribution(min, max);
    float dice_roll = distribution(generator);
    return dice_roll;
}
```

3 | Benchmark Program

3.1 CMakeLists

3.1.1 CMakeLists.txt

Path: *$PROJECT_DIR/profiler/CMakeLists.txt*

```
cmake_minimum_required(VERSION 2.8)

# project name
project(profiler)

# setup version numbers
set(VERSION_MAJOR 1)
set(VERSION_MINOR 0)
set(VERSION_PATCH 0)


# disable the rpath stuff
set(CMAKE_SKIP_BUILD_RPATH FALSE)


# detect the architecture (note: this test won't work for cross-compilation)
include(CheckTypeSize)
check_type_size(void* SIZEOF_VOID_PTR)
if("${SIZEOF_VOID_PTR}" STREQUAL "4")
    set(ARCH x86)
elseif("${SIZEOF_VOID_PTR}" STREQUAL "8")
    set(ARCH x64)
else()
    message(FATAL_ERROR "Unsupported architecture")
    return()
endif()

# links
set(SFML_LIBS sfml-audio sfml-graphics sfml-network sfml-system sfml-window)
set(DSF_LIBS tbb yctools dsf)
set(LIBS ${SFML_LIBS} ${DSF_LIBS})

# headers
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/extlibs/headers)
set(INCROOT ${CMAKE_CURRENT_SOURCE_DIR}/include)
set(INC
    ${INCROOT}/BouncingCircleManager.h
    ${INCROOT}/MyDSF.h
    ${INCROOT}/SyncFlockingBoid.h
    ${INCROOT}/DSFSFML.h
    ${INCROOT}/RandomCircleManager.h
    ${INCROOT}/SyncVector3D.h
    ${INCROOT}/ResourcePath.hpp
    ${INCROOT}/Vector3D.h
    ${INCROOT}/FPS.h
    ${INCROOT}/SyncBouncingCircle.h
    ${INCROOT}/FlockingBoidManager.h
    ${INCROOT}/SyncCircle.h
)

# source
set(SRCROOT ${CMAKE_CURRENT_SOURCE_DIR}/src)
set(SRC
    ${SRCROOT}/BouncingCircleManager.cpp
    ${SRCROOT}/DSFSFML.cpp
    ${SRCROOT}/SyncBouncingCircle.cpp
    ${SRCROOT}/SyncCircle.cpp
    ${SRCROOT}/FPS.cpp
    ${SRCROOT}/SyncFlockingBoid.cpp
    ${SRCROOT}/FlockingBoidManager.cpp
    ${SRCROOT}/SyncVector3D.cpp
    ${SRCROOT}/MyDSF.cpp
    ${SRCROOT}/Vector3D.cpp
    ${SRCROOT}/RandomCircleManager.cpp
    ${SRCROOT}/main.cpp
)

# resources
set(RESRCROOT ${PROJECT_SOURCE_DIR}/resource)

# os configurations
if (MSVC)
    # Windows VC
    # Activate C++ exception handling
    if (NOT CMAKE_CXX_FLAGS MATCHES "/EHsc")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /EHsc")
    endif ()

    # Set Warning level always to 4
    if (CMAKE_CXX_FLAGS MATCHES "/W[0-4]")
        string(REGEX REPLACE "/W[0-4]" "/W4" CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}")
    else ()
        set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /W4")
    endif ()

    # Add Math definitions
    add_definitions(-D_USE_MATH_DEFINES)

    # Add Source
    set(SRC ${SRC} ${SRCROOT}/ResourcePath.cpp)
    add_executable(${PROJECT_NAME} ${INC} ${SRC})
    # Add Links
        set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-msvc/${ARCH}/libs)
        set(DEPENDENCIES_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-msvc/${ARCH}/dlls)
    set(DEPENDENCIES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>)
    set(RESOURCES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>/Resources)

elseif(APPLE)
    # Mac OS X Xcode
        set(CMAKE_MACOSX_RPATH 1)
    ADD_DEFINITIONS(-std=c++11)

    # Add Source
    set(SRC ${SRC} ${SRCROOT}/ResourcePath.mm)
    add_executable(${PROJECT_NAME} MACOSX_BUNDLE ${INC} ${SRC})
    # Add Links
    SET_TARGET_PROPERTIES(${PROJECT_NAME} PROPERTIES
               XCODE_ATTRIBUTE_LD_RUNPATH_SEARCH_PATHS @executable_path/../Frameworks/)
    set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-osx)
    set(DEPENDENCIES_DIR ${LIBS_DIR})
    set(LIBS ${LIBS} SFML)
    set(DEPENDENCIES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>/../Frameworks)
    set(RESOURCES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>/../Resources)
else()
    # Unix
    ADD_DEFINITIONS(-std=c++11)
    # Add Source
    set(SRC ${SRC} ${SRCROOT}/ResourcePath.cpp)
    add_executable(${PROJECT_NAME} ${INC} ${SRC})
    # Add Links
        set(LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extlibs/libs-unix/${ARCH})
        set(DEPENDENCIES_DIR ${LIBS_DIR})
    set(DEPENDENCIES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>)
    set(RESOURCES_TARGET_DIR $<TARGET_FILE_DIR:${PROJECT_NAME}>/Resources)

endif()


foreach(LIB ${LIBS})
    find_library(LIB_${LIB} NAMES ${LIB} PATHS ${LIBS_DIR})
    target_link_libraries(${PROJECT_NAME} ${LIB_${LIB}})
endforeach()

set_target_properties(
    ${PROJECT_NAME} PROPERTIES
    ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
    LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
    RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/build"
)

# copy dependencies and resources
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                    COMMAND ${CMAKE_COMMAND} -E copy_directory
                    ${DEPENDENCIES_DIR}
                    ${DEPENDENCIES_TARGET_DIR})
```

```
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E copy_directory
                   "${PROJECT_SOURCE_DIR}/resource"
                   ${RESOURCES_TARGET_DIR})
```

3.2 BouncingCircleManager

3.2.1 BouncingCircleManager.h

Path: *$PROJECT_DIR/profiler/include/BouncingCircleManager.h*

```
//
//  BouncingCircleManager.h
//  profiler
//
//  Created by Yu Chen on 2/22/15.
//
//

#ifndef profiler_BouncingCircleManager_h
#define profiler_BouncingCircleManager_h

#include "MyDSF.h"
#include "SyncBouncingCircle.h"
#include <yctools/Random.h>
#include <vector>

class BouncingCircleManager
{
public:
    BouncingCircleManager(MyDSF* dsf);
    ~BouncingCircleManager();
    MyDSF* dsf;
    dsf::TaskFunction* create;
    dsf::TaskFunction* update;
    dsf::TaskFunction* destroy;
    std::vector<SyncBouncingCircle*>* createRandomCircles(int number, int radius, int boundX, int boundY);
private:
    SyncBouncingCircle* createRandomCircle(int radius, int boundX, int boundY);
};


#endif
```

3.2.2 BouncingCircleManager.cpp

Path: *$PROJECT_DIR/profiler/src/BouncingCircleManager.cpp*

```
//
//  BouncingCircleManager.cpp
//  profiler
//
//  Created by Yu Chen on 2/22/15.
//
//

#include "../include/BouncingCircleManager.h"
#include <cmath>

BouncingCircleManager::BouncingCircleManager(MyDSF* dsf)
{
    this->dsf = dsf;
    this->create = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from,
    dsf::TaskArgument* args)
                                          {
                                              SyncBouncingCircle* syncObj;
                                              std::vector<SyncBouncingCircle*>* syncObjs;
                                              std::tie(syncObj, syncObjs) = args->to<
   std::tuple<SyncBouncingCircle*, std::vector<SyncBouncingCircle*>*>>();
                                              if(this->dsf->window->isOpen())
                                              {
                                                  this->dsf->send(to, from, this->update, new
   dsf::TaskArgument(std::make_tuple(syncObj, syncObjs)));
                                                  this->dsf->lock();
                                                  this->dsf->drawables->push_back(syncObj);
                                                  this->dsf->unlock();
                                              }
```

```
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, new dsf::TaskArgument(syncObj));
                                             }
                                         });
    this->update = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                         {
                                             SyncBouncingCircle* syncObj;
                                             std::vector<SyncBouncingCircle*>* syncObjs;
                                             std::tie(syncObj, syncObjs) = args->to<std::tuple<SyncBouncingCircle*, std::vector<SyncBouncingCircle*>*>>();
                                             if(this->dsf->window->isOpen())
                                             {
                                                 auto size = this->dsf->window->getSize();
                                                 for(auto & sb : *syncObjs)
                                                 {
                                                     if(sb != syncObj)
                                                         syncObj->collide(sb);
                                                 }
                                                 syncObj->move(size.x, size.y);
                                                 this->dsf->send(to, from, this->update, new dsf::TaskArgument(std::make_tuple(syncObj, syncObjs)));
                                             }
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, new dsf::TaskArgument(syncObj));
                                             }
                                         });
    this->destroy = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                          {
                                              auto syncObj = args->to<SyncBouncingCircle*>();
                                              auto drawable = (sf::Drawable*) syncObj;
                                              this->dsf->lock();
                                              this->dsf->drawables->erase(
                                                                          std::remove_if(
                                                                                         this->dsf->drawables->begin(),
                                                                                         this->dsf->drawables->end(),
                                                                                         [&](sf::Drawable* d)
                                                                                         {
                                                                                             return d == drawable;
                                                                                         }),
                                                                          this->dsf->drawables->end());
                                              this->dsf->remove(to);
                                              this->dsf->unlock();
                                          });
}
BouncingCircleManager::~BouncingCircleManager()
{
    delete this->create;
    delete this->update;
    delete this->destroy;
}

std::vector<SyncBouncingCircle*>* BouncingCircleManager::createRandomCircles(int number, int radius, int boundX, int boundY) {
    int numberOfCircles = number;
    auto circles = new std::vector<SyncBouncingCircle*>();
    SyncBouncingCircle* circle;
    while(circles->size() < numberOfCircles) {
        std::vector<SyncBouncingCircle*>::iterator itr = circles->begin();
        circle = createRandomCircle(radius, boundX, boundY);
        while (itr != circles->end()) {
            if (circle->hasCollision(*itr)) {
                delete circle;
                circle = createRandomCircle(radius, boundX, boundY);
                itr = circles->begin();
            } else {
                itr ++;
            }
        }
        circles->push_back(circle);
    }
    return circles;
}

SyncBouncingCircle*  BouncingCircleManager::createRandomCircle(int radius, int boundX, int boundY) {
    SyncBouncingCircle* circle = new SyncBouncingCircle();
    int direction = yc::Random().randInt(1, 360);
    int posx = yc::Random().randInt(0, boundX - radius * 2);
    int posy = yc::Random().randInt(0, boundY - radius * 2);
    float x = cosf(direction * M_PI / 180);
    float y = sinf(direction * M_PI / 180);
    circle->setRadius(radius);
    circle->setPosition(sf::Vector2<float>(posx, posy));
    circle->setVelocity(sf::Vector2<float>(x, y));
    return circle;
}
```

3.3 DSFSFML

3.3.1 DSFSFML.h

Path: *$PROJECT_DIR/profiler/include/DSFSFML.h*

```
//
//  DSFSFML.h
//  profiler
//
//  Created by Yu Chen on 2/16/15.
//
//

#ifndef profiler_DSFSFML_h
#define profiler_DSFSFML_h

#include <SFML/Graphics.hpp>
#include <vector>

namespace dsf
{
    namespace sfml
    {
        class RenderWindow
        {
        public:
            explicit RenderWindow();
            virtual ~RenderWindow();
            sf::RenderWindow* window;
            std::vector<sf::Drawable*>* drawables;
        protected:
            virtual void draw() = 0;
        };
    }
}
#endif
```

3.3.2 DSFSFML.cpp

Path: *$PROJECT_DIR/profiler/src/DSFSFML.cpp*

```
//
//  DSFSFML.cpp
//  profiler
//
//  Created by Yu Chen on 2/17/15.
//
//

#include "../include/DSFSFML.h"

namespace dsf
{
    namespace sfml
    {
        RenderWindow::RenderWindow()
        {
            this->window = new sf::RenderWindow();
            this->drawables = new std::vector<sf::Drawable*>();
        }

        RenderWindow::~RenderWindow()
        {
            delete this->window;
            delete this->drawables;
        }
    }
}

```

3.4 FlockingBoidManager

3.4.1 FlockingBoidManager.h

Path: *$PROJECT_DIR/profiler/include/FlockingBoidManager.h*

```
//
//  FlockingBoidManager.h
//  profiler
//
//  Created by Yu Chen on 3/12/15.
//
//

#ifndef profiler_FlockingBoidManager_h
#define profiler_FlockingBoidManager_h

#include "MyDSF.h"
#include "SyncFlockingBoid.h"
#include <yctools/Random.h>
#include <vector>

class FlockingBoidManager
{
public:
    FlockingBoidManager(MyDSF* dsf);
    ~FlockingBoidManager();
    MyDSF* dsf;
    dsf::TaskFunction* create;
    dsf::TaskFunction* update;
    dsf::TaskFunction* destroy;
};

#endif
```

3.4.2 FlockingBoidManager.cpp

Path: *$PROJECT_DIR/profiler/src/FlockingBoidManager.cpp*

```
//
//  FlockingBoidManager.cpp
//  profiler
//
//  Created by Yu Chen on 3/12/15.
//
//

#include "../include/FlockingBoidManager.h"


FlockingBoidManager::FlockingBoidManager(MyDSF* dsf)
{
    this->dsf = dsf;
    this->create = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from,
    dsf::TaskArgument* args)
                                        {
                                            SyncFlockingBoid* syncObj;
                                            std::vector<SyncFlockingBoid*>* syncObjs;
                                            std::tie(syncObj, syncObjs) = args->to<
    std::tuple<SyncFlockingBoid*, std::vector<SyncFlockingBoid*>*>>();
                                            if(this->dsf->window->isOpen())
                                            {
                                                this->dsf->send(to, from, this->update, new
    dsf::TaskArgument(std::make_tuple(syncObj, syncObjs)));
                                                this->dsf->lock();
                                                this->dsf->drawables->push_back(syncObj);
                                                this->dsf->unlock();
                                            }
                                            else
                                            {
                                                this->dsf->send(to, from, this->destroy, new
    dsf::TaskArgument(syncObj));
                                            }
                                        });
    this->update = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                        {
                                            SyncFlockingBoid* syncObj;
                                            std::vector<SyncFlockingBoid*>* syncObjs;
                                            std::tie(syncObj, syncObjs) = args->to<std::tuple<SyncFlockingBoid*, std::vector<SyncFlockingBoid*>*>>();
                                            if(this->dsf->window->isOpen())
                                            {
                                                syncObj->run(syncObjs, this->dsf->window);
                                                this->dsf->send(to, from, this->update, new dsf::TaskArgument(std::make_tuple(syncObj, syncObjs)));
                                            }
                                            else
                                            {
                                                this->dsf->send(to, from, this->destroy, new dsf::TaskArgument(syncObj));
                                            }
                                        });
    this->destroy = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                          {
                                              auto syncObj = args->to<SyncFlockingBoid*>();
                                              auto drawable = (sf::Drawable*) syncObj;
                                              this->dsf->lock();
                                              this->dsf->drawables->erase(
                                                                            std::remove_if(
                                                                                           this->dsf->drawables->begin(),
                                                                                           this->dsf->drawables->end(),
                                                                                           [&](sf::Drawable* d)
                                                                                           {
                                                                                               return d == drawable;
                                                                                           }),
                                                                            this->dsf->drawables->end());
                                              this->dsf->remove(to);
                                              this->dsf->unlock();
                                          });
}
FlockingBoidManager::~FlockingBoidManager()
{
    delete this->create;
    delete this->update;
    delete this->destroy;
}
```

3.5 FPS

3.5.1 FPS.h

Path: *$PROJECT_DIR/profiler/include/FPS.h*

```
//
//  FPS.h
//  profiler
//
//  Created by Yu Chen on 2/8/15.
//
//

#ifndef profiler_FPS_h
#define profiler_FPS_h

#include <SFML/Graphics.hpp>

class FPS
{
public:
    FPS(float refreshTime = 1.0, float startTime = 1.0);
    ~FPS();
    void refresh();
    void restart();
    float current;
    float average;
    float max;
    float min;
private:
    bool started;
    float refreshTime;
    float startTime;
    sf::Clock clock;
    sf::Clock clockFps;
    sf::Clock clockStart;
    float temp;
};

#endif
```

3.5.2 FPS.cpp

Path: *$PROJECT_DIR/profiler/src/FPS.cpp*

```
//
//  FPS.cpp
//  profiler
//
//  Created by Yu Chen on 2/8/15.
//
//

#include "../include/FPS.h"

FPS::FPS(float refreshTime, float startTime)
{
    this->refreshTime = refreshTime;
    this->startTime = startTime;
}

FPS::~FPS()
{

}

void FPS::refresh()
{
    if(started)
    {
        if(!this->temp)
            this->temp = 1.0f / this->clockFps.getElapsedTime().asSeconds();
        else
            this->temp = (this->temp + 1.0f / this->clockFps.getElapsedTime().asSeconds()) / 2.0f;
        this->clockFps.restart();
        if(this->clock.getElapsedTime().asSeconds() >= refreshTime)
        {
            this->current = this->temp;
            this->temp = 0;
            if(this->average)
                this->average = (this->average + this->current) / 2.0f;
            else
                this->average = this->current;
            if(!this->max || this->max < this->current)
                this->max = this->current;
            if(!this->min || this->min > this->current)
                this->min = this->current;

            this->clock.restart();
        }
    }
    else if(this->clockStart.getElapsedTime().asSeconds() >= startTime)
    {
        this->started = true;
        this->clock.restart();
        this->clockFps.restart();
        this->current = this->average = this->max = this->min = this->temp = 0;
    }
}

void FPS::restart()
{
    this->clockStart.restart();
    this->started = false;
    this->current = this->average = this->max = this->min = this->temp = 0;
}
```

3.6 MyDSF

3.6.1 MyDSF.h

Path: *$PROJECT_DIR/profiler/include/MyDSF.h*

```cpp
//
//  MyDSF.h
//  profiler
//
//  Created by Yu Chen on 2/8/15.
//
//

#ifndef profiler_MyDSF_h
#define profiler_MyDSF_h

#include <dsf/DualStateFramework.h>
#include <dsf/TaskFunction.h>
#include <dsf/Lock.h>
#include "DSFSFML.h"
#include "FPS.h"
#include "ResourcePath.hpp"

class MyDSF : public dsf::DualStateFramework, public dsf::sfml::RenderWindow, public dsf::Lock
{
private:
    class Sender;
    FPS* fps;
    std::vector<std::tuple<float,float,float>> fpsList;
    sf::Clock* clock;
    sf::Font font;
    unsigned int numberOfCores = 1;
    std::vector<std::tuple<float,float,float>> stretch(std::vector<std::tuple<float,float,float>> arr,
                                                        std::vector<std::tuple<float,float,float>> strelen,
                                                        int maxLen);
public:
    MyDSF();
    ~MyDSF();
    void initialize() override;
    Sender* sender;
    unsigned int duration = 10;
    unsigned int maxNumberOfCores = 8;
protected:
    void refresh() override;
    void run() override;
    void draw() override;
};

class MyDSF::Sender : public dsf::SynchronizedObject
{
public:
    Sender(dsf::DualStateFramework* dsf);
    ~Sender();
    dsf::DualStateFramework* dsf;
    dsf::TaskFunction* create;
    dsf::TaskFunction* update;
    dsf::TaskFunction* destroy;
protected:
    void run() override;
};


#endif
```

3.6.2 MyDSF.cpp

Path: *$PROJECT_DIR/profiler/src/MyDSF.cpp*

```cpp
//
//  MyDSF.cpp
//  profiler
//
//  Created by Yu Chen on 2/8/15.
//
//

#include "../include/MyDSF.h"
#include "../include/FPS.h"
#include <dsf/TaskArgument.h>
#include <SFML/Graphics.hpp>
#include <iostream>
#include <yctools/Random.h>
```

```

MyDSF::MyDSF()
: DualStateFramework()
{
    this->initialize();
}
MyDSF::~MyDSF()
{
    delete this->fps;
    delete this->clock;
}

void MyDSF::initialize()
{
    this->sender = new Sender(this);
    this->fps = new FPS();
    this->clock = new sf::Clock();
    this->add(this->sender);
    this->send(this->sender, this->sender, this->sender->create, new dsf::TaskArgument((
   dsf::sfml::RenderWindow*)this));
    this->window->create(sf::VideoMode(800, 600), "DSF Profiler");
    this->setNumberOfThreads(numberOfCores);
    this->font.loadFromFile(resourcePath() + "sansation.ttf");
}

void MyDSF::refresh()
{
    dsf::DualStateFramework::refresh();
}

void MyDSF::run()
{
    if(this->numberOfCores <= this->maxNumberOfCores
       && this->clock->getElapsedTime().asSeconds() >= this->duration)
    {
        this->fpsList.push_back(std::make_tuple(this->fps->average,
                                                this->fps->min,
                                                this->fps->max));
        this->clock->restart();
        this->fps->restart();
        this->numberOfCores ++;
        this->setNumberOfThreads(this->numberOfCores);
    }
    dsf::DualStateFramework::run();
    if (this->window->isOpen())
    {
        sf::Event event;
        while (this->window->pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
            {
                this->window->close();
            }
        }
        this->window->clear();
        this->draw();
        this->fps->refresh();
        if(this->numberOfCores <= this->maxNumberOfCores)
        {
            if(this->fps->current)
            {
                std::string msg = "Number of Core: " + std::to_string(this->numberOfCores) + "\n";
                msg += "FPS \n";
                msg += " Current: " + std::to_string(fps->current) + "\n";
                msg += " Average: " + std::to_string(fps->average) + "\n";
                msg += " Min: " + std::to_string(fps->min) + "\n";
                msg += " Max: " + std::to_string(fps->max);
                sf::Text text(msg, font);
                this->window->draw(text);
            }
        }
        else
        {
            const float width = 700;
            const float height = 500;
            const sf::Vector2<float> origin(50, 550);
            const float thickness = 3;
            const float barThickness = 10;
            sf::RectangleShape x(sf::Vector2<float>(width, thickness));
            sf::RectangleShape y(sf::Vector2<float>(thickness, height));
            sf::RectangleShape fill(sf::Vector2<float>(thickness, thickness));
            x.setPosition(origin);
            y.setPosition(origin);
            fill.setPosition(origin - sf::Vector2<float>(thickness, 0));
            y.rotate(180);
            this->window->draw(x);
             this->window->draw(y);
            this->window->draw(fill);
            auto bars = stretch(this->fpsList, this->fpsList, height);
            for(int i = 0; i < this->fpsList.size(); i ++)
            {
                unsigned int charSize = 12;
                float x = width / (this->fpsList.size() + 1) * (i + 1);
                float average;
                float min;
                float max;
                float averageBar;
                float minBar;
                float maxBar;
                std::tie(averageBar, minBar, maxBar) = bars[i];
                std::tie(average, min, max) = this->fpsList[i];
                sf::RectangleShape bar(sf::Vector2<float>(barThickness, maxBar - minBar));
                bar.setPosition(origin + sf::Vector2<float>(x, -minBar));
                bar.rotate(180);
                sf::Text textAverage(std::to_string(average), this->font);
                textAverage.setCharacterSize(charSize);
                textAverage.setPosition(origin + sf::Vector2<float>(x, -averageBar));
                sf::Text textMin(std::to_string(min), this->font);
                textMin.setCharacterSize(charSize);
                textMin.setPosition(origin + sf::Vector2<float>(x, -minBar));
                sf::Text textMax(std::to_string(max), this->font);
                textMax.setCharacterSize(charSize);
                textMax.setPosition(origin + sf::Vector2<float>(x, -maxBar));
                sf::Text textCore(std::to_string(i + 1), this->font);
                textCore.setCharacterSize(14);
                textCore.setPosition(origin + sf::Vector2<float>(x, 0));
                this->window->draw(bar);
                this->window->draw(textAverage);
                this->window->draw(textMin);
                this->window->draw(textMax);
                this->window->draw(textCore);
            }
        }
        this->window->display();
    }
}

void MyDSF::draw()
{
    std::for_each(this->drawables->begin(), this->drawables->end(), [this](sf::Drawable* drawable)
                  {
                      this->window->draw(*drawable);
                  });
}

MyDSF::Sender::Sender(dsf::DualStateFramework* dsf)  : SynchronizedObject()
{
    this->dsf = dsf;
    this->create = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                         {
                                             auto rw = args->to<dsf::sfml::RenderWindow*>();
                                             if(rw->window->isOpen())
                                             {
                                                 this->dsf->send(to, from, this->update, new dsf::TaskArgument(rw));
                                             }
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, nullptr);
                                             }
                                         });
    this->update = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                         {
                                             auto rw = args->to<dsf::sfml::RenderWindow*>();
                                             if(rw->window->isOpen())
                                             {
                                                 this->dsf->send(to, from, this->update, new dsf::TaskArgument(rw));
                                             }
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, nullptr);
                                             }
                                         });
    this->destroy = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                          {
                                              this->dsf->remove(to);
                                          });
}

MyDSF::Sender::~Sender()
```

```
{
    delete this->create;
    delete this->update;
    delete this->destroy;
}

void MyDSF::Sender::run()
{
    if(this->receive())
        this->process();
}

std::vector<std::tuple<float,float,float>> MyDSF::stretch(std::vector<std::tuple<float,float,float>> arr,
                                                          std::vector<std::tuple<float,float,float>> strelen,
                                                          int maxLen)
{
    bool canDouble = true;
    for(int i = 0; i < arr.size(); i ++)
    {
        float average;
        float min;
        float max;
        float averageOrigin;
        float minOrigin;
        float maxOrigin;
        std::tie(average, min, max) = arr[i];
        std::tie(averageOrigin, minOrigin, maxOrigin) = strelen[i];
        if(max + maxOrigin > maxLen)
        {
            canDouble = false;
            break;
        }
    }
    if(canDouble)
    {
        for(int i = 0; i < arr.size(); i ++)
        {
            float average;
            float min;
            float max;
            float averageOrigin;
            float minOrigin;
            float maxOrigin;
            std::tie(average, min, max) = arr[i];
            std::tie(averageOrigin, minOrigin, maxOrigin) = strelen[i];
            arr[i] = std::make_tuple(average + averageOrigin,
                                     min + minOrigin,
                                     max + maxOrigin);
        }
        return stretch(arr, strelen, maxLen);
    }
    return arr;
}
```

3.7 RandomCircleManager

3.7.1 RandomCircleManager.h

Path: *$PROJECT_DIR/profiler/include/RandomCircleManager.h*

```
//
//  RandomCircleManager.h
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#ifndef profiler_RandomCircleManager_h
#define profiler_RandomCircleManager_h

#include "MyDSF.h"
#include "SyncCircle.h"
#include <yctools/Random.h>

class RandomCircleManager
{
public:
    RandomCircleManager(MyDSF* dsf);
    ~RandomCircleManager();
    MyDSF* dsf;
    dsf::TaskFunction* create;
    dsf::TaskFunction* update;
    dsf::TaskFunction* destroy;
};

#endif
```

3.7.2 RandomCircleManager.cpp

Path: *$PROJECT_DIR/profiler/src/RandomCircleManager.cpp*

```
//
//  RandomCircleManager.cpp
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#include "../include/RandomCircleManager.h"

RandomCircleManager::RandomCircleManager(MyDSF* dsf)
{
    this->dsf = dsf;
    this->create = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                         {
                                             auto syncObj = args->to<SyncCircle*>();
                                             if(this->dsf->window->isOpen())
                                             {
                                                 this->dsf->send(to, from, this->update, new dsf::TaskArgument(syncObj));
                                                 this->dsf->lock();
                                                 this->dsf->drawables->push_back(syncObj);
                                                 this->dsf->unlock();
                                             }
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, new dsf::TaskArgument(syncObj));
                                             }
                                         });
    this->update = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                         {
                                             auto syncObj = args->to<SyncCircle*>();
                                             if(this->dsf->window->isOpen())
                                             {
                                                 auto size = this->dsf->window->getSize();
                                                 auto radius = syncObj->getRadius();
                                                 syncObj->setPosition(
                                                                      yc::Random().randInt(0, size.x - 2 * radius),
                                                                      yc::Random().randInt(0, size.y - 2 * radius)
                                                                      );
                                                 this->dsf->send(to, from, this->update, new dsf::TaskArgument(syncObj));
                                             }
                                             else
                                             {
                                                 this->dsf->send(to, from, this->destroy, new dsf::TaskArgument(syncObj));
                                             }
                                         });
    this->destroy = new dsf::TaskFunction([this](dsf::SynchronizedObject* to, dsf::SynchronizedObject* from, dsf::TaskArgument* args)
                                          {
                                              auto syncObj = args->to<SyncCircle*>();
                                              auto drawable = (sf::Drawable*) syncObj;
                                              this->dsf->lock();
                                              this->dsf->drawables->erase(
                                                                          std::remove_if(
                                                                                         this->dsf->drawables->begin(),
                                                                                         this->dsf->drawables->end(),
                                                                                         [&](sf::Drawable* d)
                                                                                         {
                                                                                             return d == drawable;
                                                                                         }),
                                                                        this->dsf->drawables->end());
                                                this->dsf->remove(to);
                                                this->dsf->unlock();
                                            });
}
RandomCircleManager::~RandomCircleManager()
{
    delete this->create;
    delete this->update;
    delete this->destroy;
}
```

3.8 ResourcePath

3.8.1 ResourcePath.hpp

Path: *$PROJECT_DIR/profiler/include/ResourcePath.hpp*

```
//
// SFML - Simple and Fast Multimedia Library
// Copyright (C) 2007-2013 Marco Antognini (antognini.marco@gmail.com),
//                         Laurent Gomila (laurent.gom@gmail.com),
//
// This software is provided 'as-is', without any express or implied warranty.
// In no event will the authors be held liable for any damages arising from the use of this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it freely,
// subject to the following restrictions:
//
// 1. The origin of this software must not be misrepresented;
//    you must not claim that you wrote the original software.
//    If you use this software in a product, an acknowledgment
//    in the product documentation would be appreciated but is not required.
//
// 2. Altered source versions must be plainly marked as such,
//    and must not be misrepresented as being the original software.
//
// 3. This notice may not be removed or altered from any source distribution.
//

#ifndef RESOURCE_PATH_HPP
#define RESOURCE_PATH_HPP

// Headers
#include <string>

std::string resourcePath(void);

#endif
```

3.8.2 ResourcePath.cpp

Path: *$PROJECT_DIR/profiler/src/ResourcePath.cpp*

```
#include "../include/ResourcePath.hpp"

std::string resourcePath(void)
{
    return "Resources/";
}
```

3.8.3 ResourcePath.mm

Path: *$PROJECT_DIR/profiler/src/ResourcePath.mm*

```
//
// SFML - Simple and Fast Multimedia Library
// Copyright (C) 2007-2013 Marco Antognini (antognini.marco@gmail.com),
//                         Laurent Gomila (laurent.gom@gmail.com),
//
// This software is provided 'as-is', without any express or implied warranty.
// In no event will the authors be held liable for any damages arising from the use of this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it freely,
// subject to the following restrictions:
//
// 1. The origin of this software must not be misrepresented;
//    you must not claim that you wrote the original software.
//    If you use this software in a product, an acknowledgment
//    in the product documentation would be appreciated but is not required.
//
// 2. Altered source versions must be plainly marked as such,
//    and must not be misrepresented as being the original software.
//
// 3. This notice may not be removed or altered from any source distribution.
//

// Headers
#include "../include/ResourcePath.hpp"
#import <Foundation/Foundation.h>

std::string resourcePath(void)
{
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

    std::string rpath;
    NSBundle* bundle = [NSBundle mainBundle];

    if (bundle == nil) {
#ifdef DEBUG
        NSLog(@"bundle is nil... thus no resources path can be found.");
#endif
    } else {
        NSString* path = [bundle resourcePath];
        rpath = [path UTF8String] + std::string("/");
    }

    [pool drain];

    return rpath;
}
```

3.9 SyncBouncingCircle

3.9.1 SyncBouncingCircle.h

Path: *$PROJECT_DIR/profiler/include/SyncBouncingCircle.h*

```
//
//  SyncBouncingCircle.h
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#ifndef profiler_SyncBouncingCircle_h
#define profiler_SyncBouncingCircle_h

#include <dsf/SynchronizedObject.h>
#include <SFML/Graphics.hpp>
#include <cmath>

class SyncBouncingCircle : public dsf::SynchronizedObject, public sf::CircleShape
{
public:
    SyncBouncingCircle();
    sf::Vector2<float> getVelocity();
    void setVelocity(const sf::Vector2<float>& velocity);
    float getMass();
    void move(int width, int height);
    void collide(SyncBouncingCircle* sbc);
    bool hasCollision(SyncBouncingCircle* sbc);
protected:
    void run() override;
private:
    sf::Vector2<float> velocity;
    float mass = 1;
};

#endif
```

3.9.2 SyncBouncingCircle.cpp

Path: *$PROJECT_DIR/profiler/src/SyncBouncingCircle.cpp*

```
//
//  SyncBouncingCircle.cpp
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#include "../include/SyncBouncingCircle.h"


SyncBouncingCircle::SyncBouncingCircle()  : SynchronizedObject(), CircleShape()
{
}

void SyncBouncingCircle::run()
{
    if(this->receive())
        this->process();
}

sf::Vector2<float> SyncBouncingCircle::getVelocity() {
    return this->velocity;
}
void SyncBouncingCircle::setVelocity(const sf::Vector2<float>& velocity) {
    this->velocity = velocity;
}
float SyncBouncingCircle::getMass() {
    return this->mass;
}
void SyncBouncingCircle::move(int width, int height) {
    sf::Vector2<float> nextPosition = this->getPosition() + this->velocity;
    this->setPosition(nextPosition);
    if (getPosition().x <= 0 || getPosition().x >= width - this->getRadius() * 2) {
        this->velocity = sf::Vector2<float>(-this->velocity.x, this->velocity.y);
    }
    if (getPosition().y <= 0 || getPosition().y >= height - this->getRadius() * 2) {
        this->velocity = sf::Vector2<float>(this->velocity.x, -this->velocity.y);
    }
}

void SyncBouncingCircle::collide(SyncBouncingCircle *sbc)
{
    if (this->hasCollision(sbc))
    {
        sf::Vector2<float> v1 = this->getVelocity();
        sf::Vector2<float> v2 = sbc->getVelocity();
        sf::Vector2<float> pos1 = this->getPosition();
        sf::Vector2<float> pos2 = sbc->getPosition();
        sf::Vector2<float> n = sf::Vector2<float>(pos2.x - pos1.x, pos2.y - pos1.y);
        sf::Vector2<float> un = n / sqrtf(n.x * n.x + n.y * n.y);
        sf::Vector2<float> ut = sf::Vector2<float>(-un.y, un.x);
        float v1n = un.x * v1.x + un.y * v1.y;
        float v1t = ut.x * v1.x + ut.y * v1.y;
        float v2n = un.x * v2.x + un.y * v2.y;
        float v2t = ut.x * v2.x + ut.y * v2.y;
        float m1 = this->getMass();
        float m2 = sbc->getMass();
        float v_1t = v1t;
        float v_2t = v2t;
        float v_1n = (v1n * (m1 - m2) + 2 * m2 * v2n) / (m1 + m2);
        float v_2n = (v2n * (m2 - m1) + 2 * m1 * v1n) / (m1 + m2);
        sf::Vector2<float> v__1n = v_1n * un;
        sf::Vector2<float> v__1t = v_1t * ut;
        sf::Vector2<float> v__2n = v_2n * un;
        sf::Vector2<float> v__2t = v_2t * ut;
        sf::Vector2<float> v_1 = v__1n + v__1t;
        sf::Vector2<float> v_2 = v__2n + v__2t;
        this->setVelocity(v_1);
        sbc->setVelocity(v_2);
    }
}

bool SyncBouncingCircle::hasCollision(SyncBouncingCircle *sbc)
{
    float distanceSqr = std::pow(this->getPosition().x - sbc->getPosition().x, 2) + std::pow(this->
   getPosition().y - sbc->getPosition().y, 2);
    return distanceSqr <= std::pow(this->getRadius() + sbc->getRadius(), 2);
}
```

3.10 SyncCircle

3.10.1 SyncCircle.h

Path: *$PROJECT_DIR/profiler/include/SyncCircle.h*

```
//
//  SyncCircle.h
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#ifndef profiler_SyncCircle_h
#define profiler_SyncCircle_h

#include <dsf/SynchronizedObject.h>
#include <SFML/Graphics.hpp>

class SyncCircle : public dsf::SynchronizedObject, public sf::CircleShape
{
public:
    SyncCircle();
protected:
    void run() override;
};

#endif
```

3.10.2 SyncCircle.cpp

Path: *$PROJECT_DIR/profiler/src/SyncCircle.cpp*

```
//
//  SyncCircle.cpp
//  profiler
//
//  Created by Yu Chen on 2/21/15.
//
//

#include "../include/SyncCircle.h"

SyncCircle::SyncCircle()  : SynchronizedObject(), CircleShape()
{
}

void SyncCircle::run()
{
    if(this->receive())
        this->process();
}
```

3.11 SyncFlockingBoid

3.11.1 SyncFlockingBoid.h

Path: *$PROJECT_DIR/profiler/include/SyncFlockingBoid.h*

```
//
//  SyncFlockingBoid.h
//  profiler
//
//  Created by Yu Chen on 3/12/15.
//
//

#ifndef profiler_SyncFlockingBoid_h
#define profiler_SyncFlockingBoid_h

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

```
#include <dsf/SynchronizedVar.h>
#include <SFML/Graphics.hpp>
#include <yctools/Random.h>
#include "SyncVector3D.h"

class SyncFlockingBoid : public dsf::SynchronizedObject, public sf::CircleShape
{
public:
    SyncVector3D* loc;
    SyncVector3D* vel;
    SyncVector3D* acc;
    float r;
    float maxforce;    // Maximum steering force
    float maxspeed;    // Maximum speed
    SyncFlockingBoid();
    ~SyncFlockingBoid();
    SyncFlockingBoid(Vector3D* loc, float ms, float mf);
    void run(std::vector<SyncFlockingBoid*>* boids, sf::RenderWindow* window);
    Vector3D steer(Vector3D* target, bool slowdown);
    Vector3D separate (std::vector<SyncFlockingBoid*>* boids);
    Vector3D align (std::vector<SyncFlockingBoid*>* boids);
    Vector3D cohesion (std::vector<SyncFlockingBoid*>* boids);
protected:
    void run() override;
};

#endif
```

3.11.2 SyncFlockingBoid.cpp

Path: *$PROJECT_DIR/profiler/src/SyncFlockingBoid.cpp*

```
//
//  SyncFlockingBoid.cpp
//  profiler
//
//  Created by Yu Chen on 3/12/15.
//
//

#include "../include/SyncFlockingBoid.h"

SyncFlockingBoid::SyncFlockingBoid() : SynchronizedObject(), CircleShape()
{
}
SyncFlockingBoid::~SyncFlockingBoid()
{
    delete this->acc;
    delete this->vel;
    delete this->loc;
}
SyncFlockingBoid::SyncFlockingBoid(Vector3D* loc, float ms, float mf) : SynchronizedObject(), CircleShape()
{
    this->acc = new SyncVector3D(0,0);
    this->vel = new SyncVector3D(yc::Random().randFloat(-1,1), yc::Random().randFloat(-1,1));
    this->loc = new SyncVector3D(loc->getX(), loc->getY(), loc->getZ());
    delete loc;
    r = 2.0f;
    maxspeed = ms;
    maxforce = mf;
}

void SyncFlockingBoid::run(std::vector<SyncFlockingBoid*>* boids, sf::RenderWindow* window) {
    //flock(boids);
    //update();
    //borders(window->getSize().x, window->getSize().y);
    //render(window);

    // We accumulate a new acceleration each time based on three rules
    Vector3D sep = separate(boids);   // Separation
    Vector3D ali = align(boids);      // Alignment
    Vector3D coh = cohesion(boids);   // Cohesion

    // Arbitrarily weight these forces
    sep *= 2.0f;
    ali *= 1.0f;
    coh *= 1.0f;

    auto acc = Vector3D(*this->acc);
    auto loc = Vector3D(this->loc->getX(), this->loc->getY());
    auto vel = Vector3D(*this->vel);

    // Add the force vectors to acceleration
    acc += sep;
    acc += ali;
    acc += coh;

    // Method to update location
    // Update velocity
    vel += acc;

    // Limit speed
    vel.limit(maxspeed);

    loc += vel;

    // Reset accelertion to 0 each cycle
    acc.setXYZ(0,0,0);

    // Wraparound
    auto width = window->getSize().x;
    auto height = window->getSize().y;
    if (loc.getX() < -r)
        loc.setX(width+r);
    if (loc.getY() < -r)
        loc.setY(height+r);
    if (loc.getX() > width+r)
        loc.setX(-r);
    if (loc.getY() > height+r)
        loc.setY(-r);


    this->acc->setXYZ(acc);
    this->loc->setXYZ(loc);
    this->vel->setXYZ(vel);
    this->setPosition(this->loc->getX(), this->loc->getY());
    this->setRadius(r);
}


// A method that calculates a steering vector towards a target
// Takes a second argument, if true, it slows down as it approaches the target

Vector3D SyncFlockingBoid::steer(Vector3D* target, bool slowdown) {
    Vector3D steer;  // The steering vector
    Vector3D desired = *target - *this->loc;  // A vector pointing from the location to the target
    float d = desired.magnitude(); // Distance from the target is the magnitude of the vector

    // If the distance is greater than 0, calc steering (otherwise return zero vector)
    if (d > 0) {
        // Normalize desired
        desired.normalize();

        // Two options for desired vector magnitude (1 -- based on distance, 2 -- maxspeed)
        if ((slowdown) && (d < 100.0f))
            desired *= maxspeed * (d / 100.0f); // This damping is somewhat arbitrary
        else
            desired *= maxspeed;

        // Steering = Desired minus Velocity
        steer = desired - *this->vel;
        steer.limit(maxforce);  // Limit to maximum steering force

    } else {
        steer = Vector3D(0,0);
    }
    return steer;
}

// Separation
// Method checks for nearby boids and steers away
Vector3D SyncFlockingBoid::separate (std::vector<SyncFlockingBoid*>* boids) {
    float desiredseparation = 25.0f;
    Vector3D sum = Vector3D(0,0,0);
    int count = 0;

    // For every boid in the system, check if it's too close
    for (auto other: *boids) {
        float d = this->loc->distance(*other->loc);

        // If the distance is greater than 0 and less than an arbitrary amount (0 when you are yourself)
        if ((d > 0) && (d < desiredseparation)) {
            // Calculate vector pointing away from neighbor
            Vector3D diff = *this->loc - *other->loc;
            diff.normalize();
            diff /= d;        // Weight by distance
            sum += diff;
            count++;            // Keep track of how many
        }
    }

    // Average -- divide by how many
    if (count > 0)
        sum /= count;
    return sum;
}



// Alignment
// For every nearby boid in the system, calculate the average velocity
Vector3D SyncFlockingBoid::align (std::vector<SyncFlockingBoid*>* boids) {
    float neighbordist = 50.0f;
    Vector3D sum = Vector3D(0,0,0);
    int count = 0;
    for (auto & other : *boids) {
        float d = this->loc->distance(*other->loc);
        if ((d > 0) && (d < neighbordist)) {
            sum += *other->vel;
            count++;
        }
    }
    if (count > 0) {
        sum /= count;
        sum.limit(maxforce);
    }
    return sum;
}

// Cohesion
// For the average location (i.e. center) of all nearby boids, calculate steering vector towards that
   location
Vector3D SyncFlockingBoid::cohesion (std::vector<SyncFlockingBoid*>* boids) {
    float neighbordist = 50.0f;
    Vector3D sum = Vector3D(0,0,0);   // Start with empty vector to accumulate all locations
    int count = 0;
    for (auto & other: *boids) {
        float d = this->loc->distance(*other->loc);
        if ((d > 0) && (d < neighbordist)) {
            sum += *other->loc; // Add location
            count++;
        }
    }

    if (count > 0) {
        sum /= count;
        return steer(&sum,false);  // Steer towards the location
    }
    return sum;
}

void SyncFlockingBoid::run()
{
    if(this->receive())
    {
        this->loc->synchronise();
        this->acc->synchronise();
        this->vel->synchronise();
        this->process();
    }
}
```

3.12 SyncVector3D

3.12.1 SyncVector3D.h

Path: *$PROJECT_DIR/profiler/include/SyncVector3D.h*

```
//
//  SyncVector3D.h
//  profiler
//
//  Created by Yu Chen on 3/22/15.
//
//

#ifndef profiler_SyncVector3D_h
#define profiler_SyncVector3D_h

#include "Vector3D.h"
#include <dsf/Synchronisable.h>
```

```

class SyncVector3D : public dsf::Synchronisable<Vector3D>, public Vector3D {
public:
    explicit SyncVector3D(float x=0, float y=0, float z=0);
    void setX(float x) override;
    void setY(float y) override;
    void setZ(float z) override;
    void add(const Vector3D& v) override;
    void sub(const Vector3D& v) override;
    void mul(float n) override;
    void div(float n) override;
    void synchronise() override;
};

#endif
```

3.12.2 SyncVector3D.cpp

Path: *$PROJECT_DIR/profiler/src/SyncVector3D.cpp*

```
//
//  Vector3D.cpp
//  profiler
//
//  Created by Yu Chen on 3/22/15.
//
//

#include "../include/SyncVector3D.h"


SyncVector3D::SyncVector3D(float x, float y, float z) : Vector3D(x, y, z) {
    this->next = new Vector3D(x, y, z);
}

void SyncVector3D::setX(float x) {
    this->next->setX(x);
}

void SyncVector3D::setY(float y) {
    this->next->setY(y);
}

void SyncVector3D::setZ(float z) {
    this->next->setZ(z);
}

void SyncVector3D::add(const Vector3D& v) {
    this->next->add(v);
}

void SyncVector3D::sub(const Vector3D& v) {
    this->next->sub(v);
}

void SyncVector3D::mul(float n) {
    this->next->mul(n);
}

void SyncVector3D::div(float n) {
    this->next->div(n);
}

void SyncVector3D::synchronise() {
    this->x = this->next->getX();
    this->y = this->next->getY();
    this->z = this->next->getZ();
}
```

3.13 Vector3D

3.13.1 Vector3D.h

Path: *$PROJECT_DIR/profiler/include/Vector3D.h*

```
//
//  Vector3D.h
//  profiler
//
//  Created by Yu Chen on 3/22/15.
//
//

#ifndef profiler_Vector3D_h
#define profiler_Vector3D_h

#include <cmath>

class Vector3D {
protected:
    float x;
    float y;
    float z;
public:
    explicit Vector3D(float x=0, float y=0, float z=0);
    virtual ~Vector3D();
    virtual void operator=(const Vector3D& obj);

    float getX();
    float getY();
    float getZ();

    virtual void setX(float x);
    virtual void setY(float y);
    virtual void setZ(float z);

    void setXY(float x, float y);
    void setXYZ(float x, float y, float z);
    void setXYZ(const Vector3D& v);

    float magnitude();
    void normalize();
    void limit(float max);
    virtual float heading2D();
    float distance (const Vector3D& v);

    virtual void add(const Vector3D& v);
    void operator+=(const Vector3D& v);
    virtual void sub(const Vector3D& v);
    void operator-=(const Vector3D& v);
    virtual void mul(float n);
    void operator*=(float n);
    virtual void div(float n);
    void operator/=(float n);


    Vector3D operator+(const Vector3D& v) const;
    Vector3D operator-(const Vector3D& v) const;
    Vector3D operator/(float n) const;
    Vector3D operator*(float n) const;

};

#endif
```

3.13.2 Vector3D.cpp

Path: *$PROJECT_DIR/profiler/src/Vector3D.cpp*

```
//
//  Vector3D.cpp
//  profiler
//
//  Created by Yu Chen on 3/22/15.
//
//

#include "../include/Vector3D.h"


Vector3D::Vector3D(float x, float y, float z) : x(x), y(y), z(z) {
}

Vector3D::~Vector3D(){
}

void Vector3D::operator=(const Vector3D& obj)
{
    this->setXYZ(obj.x, obj.y, obj.z);
}
```

```

float Vector3D::getX() {
    return this->x;
}
float Vector3D::getY() {
    return this->y;
}
float Vector3D::getZ() {
    return this->z;
}

void Vector3D::setX(float x) {
    this->x = x;
}

void Vector3D::setY(float y) {
    this->y = y;
}

void Vector3D::setZ(float z) {
    this->z = z;
}

void Vector3D::setXY(float x, float y) {
    this->setX(x);
    this->setY(y);
}

void Vector3D::setXYZ(float x, float y, float z) {
    this->setXY(x, y);
    this->setZ(z);
}

void Vector3D::setXYZ(const Vector3D& v) {
    this->setXYZ(v.x, v.y, v.z);
}

float Vector3D::magnitude() {
    return std::sqrt(std::pow(this->getX(), 2)
                      + std::pow(this->getY(), 2)
                      + std::pow(this->getZ(), 2));
}

void Vector3D::normalize() {
    float m = magnitude();
    if (m > 0)
        div(m);
}

void Vector3D::limit(float max) {
    if (magnitude() > max) {
        normalize();
        mul(max);
    }
}

float Vector3D::distance (const Vector3D& v) {
    float dx = x - v.x;
    float dy = y - v.y;
    float dz = z - v.z;
    return std::sqrt(dx*dx + dy*dy + dz*dz);
}

float Vector3D::heading2D() {
    return -std::atan2(-this->getY(), this->getX());
}

void Vector3D::add(const Vector3D& v) {
    x += v.x;
    y += v.y;
    z += v.z;
}

void Vector3D::operator+=(const Vector3D& v) {
    add(v);
}

void Vector3D::sub(const Vector3D& v) {
    x -= v.x;
    y -= v.y;
    z -= v.z;
}

void Vector3D::operator-=(const Vector3D& v) {
    sub(v);
}

```

```
void Vector3D::mul(float n) {
    x *= n;
    y *= n;
    z *= n;
}

void Vector3D::operator*=(float n) {
    mul(n);
}

void Vector3D::div(float n) {
    x /= n;
    y /= n;
    z /= n;
}

void Vector3D::operator/=(float n) {
    div(n);
}

Vector3D Vector3D::operator+(const Vector3D& v) const {
    return Vector3D(x + v.x, y + v.y, z + v.z);
}

Vector3D Vector3D::operator-(const Vector3D& v) const {
    return Vector3D(x - v.x, y - v.y, z - v.z);
}

Vector3D Vector3D::operator/(float n) const {
    return Vector3D(x/n, y/n, z/n);
}

Vector3D Vector3D::operator*(float n) const {
    return Vector3D(x*n, y*n, z*n);
}
```

3.14 main

3.14.1 main.cpp

Path: *$PROJECT_DIR/profiler/src/main.cpp*

```
#include <SFML/Graphics.hpp>
#include <dsf/DualStateFramework.h>
#include "../include/ResourcePath.hpp"
#include "../include/FPS.h"
#include "../include/MyDSF.h"
#include "../include/SyncCircle.h"
#include "../include/RandomCircleManager.h"
#include "../include/SyncBouncingCircle.h"
#include "../include/BouncingCircleManager.h"
#include "../include/SyncFlockingBoid.h"
#include "../include/FlockingBoidManager.h"

void profile(int maxNumberOfThreads, int numberOfObjects, int durationPerIterator, int method);
void configure();

// Constances
const sf::Vector2f WINDOW_SIZE(800, 600);
const int CHAR_SIZE = 20;
const sf::Vector2f LEFT_CORNER(100, 100);
const sf::Vector2f RIGHT_CORNER(700, 100);
const sf::Vector2f INTENT(500, 25);
const sf::Vector2f SHADOW_SIZE(WINDOW_SIZE.x - LEFT_CORNER.x * 2, INTENT.y);
const int MAX_NUMBER_OF_THREADS = 64;
const int MIN_NUMBER_OF_THREADS = 2;
const int MAX_NUMBER_OF_OBJECTS = 2000;
const int MIN_NUMBER_OF_OBJECTS = 100;
const int MAX_DURATION = 360;
const int MIN_DURATION = 5;
const int NUMBER_OF_PROFILING_METHODS = 3;
const std::string profilingMethods[NUMBER_OF_PROFILING_METHODS] = {
    "Random", "Collision", "Flocking"
};
const int NUMBER_OF_SELECTIONS = 4;
// Default Configurations
int currentSelection = 1;
int maxNumberOfThreads = 4;
int numberOfObjects = 1000;
int durationPerIterator = 60;
int profilingMethodIndex = 0;
```

```cpp

int main()
{
    configure();
    return 0;
}

void configure()
{
    bool ready = false;
    // Create the main window
    sf::RenderWindow window(sf::VideoMode(WINDOW_SIZE.x, WINDOW_SIZE.y), "DSF Profiler");
    sf::Font font;
    sf::Text maxNumberOfThreadsText;
    sf::Text numberOfObjectsText;
    sf::Text durationPerIteratorText;
    sf::Text profilingMethodText;
    sf::Text maxNumberOfThreadsValue;
    sf::Text numberOfObjectsValue;
    sf::Text durationPerIteratorValue;
    sf::Text profilingMethodValue;
    sf::Text help;
    font.loadFromFile(resourcePath() + "sansation.ttf");
    maxNumberOfThreadsText.setFont(font);
    numberOfObjectsText.setFont(font);
    durationPerIteratorText.setFont(font);
    profilingMethodText.setFont(font);
    maxNumberOfThreadsValue.setFont(font);
    numberOfObjectsValue.setFont(font);
    durationPerIteratorValue.setFont(font);
    profilingMethodValue.setFont(font);
    help.setFont(font);
    maxNumberOfThreadsText.setCharacterSize(CHAR_SIZE);
    numberOfObjectsText.setCharacterSize(CHAR_SIZE);
    durationPerIteratorText.setCharacterSize(CHAR_SIZE);
    profilingMethodText.setCharacterSize(CHAR_SIZE);
    maxNumberOfThreadsValue.setCharacterSize(CHAR_SIZE);
    numberOfObjectsValue.setCharacterSize(CHAR_SIZE);
    durationPerIteratorValue.setCharacterSize(CHAR_SIZE);
    profilingMethodValue.setCharacterSize(CHAR_SIZE);
    help.setCharacterSize(CHAR_SIZE);
    maxNumberOfThreadsText.setPosition(LEFT_CORNER);
    numberOfObjectsText.setPosition(LEFT_CORNER.x, LEFT_CORNER.y + INTENT.y);
    durationPerIteratorText.setPosition(LEFT_CORNER.x, LEFT_CORNER.y + INTENT.y * 2);
    profilingMethodText.setPosition(LEFT_CORNER.x, LEFT_CORNER.y + INTENT.y * 3);
    maxNumberOfThreadsValue.setPosition(LEFT_CORNER.x + INTENT.x, LEFT_CORNER.y);
    numberOfObjectsValue.setPosition(LEFT_CORNER.x + INTENT.x, LEFT_CORNER.y + INTENT.y);
    durationPerIteratorValue.setPosition(LEFT_CORNER.x + INTENT.x, LEFT_CORNER.y + INTENT.y * 2);
    profilingMethodValue.setPosition(LEFT_CORNER.x + INTENT.x, LEFT_CORNER.y + INTENT.y * 3);
    help.setPosition(LEFT_CORNER.x, LEFT_CORNER.y + INTENT.y * 5);
    maxNumberOfThreadsText.setString("Max Number of Threads");
    numberOfObjectsText.setString("Number of Objects");
    durationPerIteratorText.setString("Duration per Iterator");
    profilingMethodText.setString("Profiling Method");
    help.setString("Press KeyUp, KeyDown, KeyLeft, and KeyRight to edit settings. \nPress Enter to run the application.");

    sf::RectangleShape shadow;
    shadow.setFillColor(sf::Color::Blue);
    shadow.setSize(SHADOW_SIZE);
    shadow.setPosition(LEFT_CORNER.x, LEFT_CORNER.y + INTENT.y * (currentSelection - 1));
    while (window.isOpen())
    {
        // Process events
        sf::Event event;
        while (window.pollEvent(event))
        {
            // Close window: exit
            if (event.type == sf::Event::Closed)
                window.close();
            // Keyboard Events
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Return)) {
                ready = true;
                window.close();
            }
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
            {
                if(currentSelection > 1)
                {
                    currentSelection --;
                    auto spos = shadow.getPosition();
                    shadow.setPosition(spos.x, spos.y - INTENT.y);
                }
            }
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
            {
                if(currentSelection < NUMBER_OF_SELECTIONS)
                {
                    currentSelection ++;
                    auto spos = shadow.getPosition();
                    shadow.setPosition(spos.x, spos.y + INTENT.y);
                }
            }
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
            {
                switch (currentSelection) {
                    case 1:
                        if(maxNumberOfThreads == MIN_NUMBER_OF_THREADS)
                            maxNumberOfThreads = MAX_NUMBER_OF_THREADS;
                        else
                            maxNumberOfThreads --;
                        break;
                    case 2:
                        if(numberOfObjects == MIN_NUMBER_OF_OBJECTS)
                            numberOfObjects = MAX_NUMBER_OF_OBJECTS;
                        else
                            numberOfObjects -= MIN_NUMBER_OF_OBJECTS;
                        break;
                    case 3:
                        if(durationPerIterator == MIN_DURATION)
                            durationPerIterator = MAX_DURATION;
                        else
                            durationPerIterator -= MIN_DURATION;
                        break;
                    case 4:
                        if(profilingMethodIndex == 0)
                            profilingMethodIndex = NUMBER_OF_PROFILING_METHODS - 1;
                        else
                            profilingMethodIndex --;
                        break;
                }
            }
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
            {
                switch (currentSelection) {
                    case 1:
                        if(maxNumberOfThreads == MAX_NUMBER_OF_THREADS)
                            maxNumberOfThreads = MIN_NUMBER_OF_THREADS;
                        else
                            maxNumberOfThreads ++;
                        break;
                    case 2:
                        if(numberOfObjects == MAX_NUMBER_OF_OBJECTS)
                            numberOfObjects = MIN_NUMBER_OF_OBJECTS;
                        else
                            numberOfObjects += MIN_NUMBER_OF_OBJECTS;
                        break;
                    case 3:
                        if(durationPerIterator == MAX_DURATION)
                            durationPerIterator = MIN_DURATION;
                        else
                            durationPerIterator += MIN_DURATION;
                        break;
                    case 4:
                        if(profilingMethodIndex == NUMBER_OF_PROFILING_METHODS - 1)
                            profilingMethodIndex = 0;
                        else
                            profilingMethodIndex ++;
                        break;
                }
            }
        }

        maxNumberOfThreadsValue.setString(std::to_string(maxNumberOfThreads));
        numberOfObjectsValue.setString(std::to_string(numberOfObjects));
        durationPerIteratorValue.setString(std::to_string(durationPerIterator));
        profilingMethodValue.setString(profilingMethods[profilingMethodIndex]);
        // Clear screen
        window.clear();
        // Draw Items
        window.draw(shadow);
        window.draw(maxNumberOfThreadsText);
        window.draw(numberOfObjectsText);
        window.draw(durationPerIteratorText);
        window.draw(profilingMethodText);
        window.draw(maxNumberOfThreadsValue);
        window.draw(numberOfObjectsValue);
        window.draw(durationPerIteratorValue);
        window.draw(profilingMethodValue);
        window.draw(help);
        // Update the window
        window.display();
    }
    if (ready)
```

```cpp
        profile(maxNumberOfThreads, numberOfObjects, durationPerIterator, profilingMethodIndex + 1);
}

void profile(int maxNumberOfThreads, int numberOfObjects, int durationPerIterator, int method)
{
    if(method == 1)
    {
        auto dsf = new MyDSF();
        auto rcm = new RandomCircleManager(dsf);
        std::vector<SyncCircle*> circles(numberOfObjects);
        for(auto & circle : circles)
        {
            circle = new SyncCircle();
            circle->setRadius(2);
            circle->setFillColor(sf::Color::Cyan);
            dsf->add(circle);
            dsf->send(circle, dsf->sender, rcm->create, new dsf::TaskArgument(circle));
        }
        dsf->duration = durationPerIterator;
        dsf->maxNumberOfCores = maxNumberOfThreads;
        dsf->start();
        delete dsf;
        delete rcm;
    }
    else if(method == 2)
    {
        auto dsf = new MyDSF();
        auto bcm = new BouncingCircleManager(dsf);
        auto bouncingCircles = bcm->createRandomCircles(numberOfObjects, 2, 800, 600);
        for(auto & bouncingCircle : *bouncingCircles)
        {
            bouncingCircle->setFillColor(sf::Color::Cyan);
            dsf->add(bouncingCircle);
            dsf->send(bouncingCircle,
                      dsf->sender,
                      bcm->create,
                      new dsf::TaskArgument(std::make_tuple(bouncingCircle, bouncingCircles)));
        }
        dsf->duration = durationPerIterator;
        dsf->maxNumberOfCores = maxNumberOfThreads;
        dsf->start();
        delete dsf;
        delete bcm;
    }
    else if(method == 3)
    {
        auto dsf = new MyDSF();
        auto flockingBoids = new std::vector<SyncFlockingBoid*>();
        auto fbm = new FlockingBoidManager(dsf);
        for (int i = 0; i < numberOfObjects; i++)
            flockingBoids->push_back(new SyncFlockingBoid(new Vector3D(dsf->window->getSize().x/2,dsf->
  window->getSize().y/2),2.0f,0.05f));
        for(auto & flockingBoid : *flockingBoids)
        {
            flockingBoid->setFillColor(sf::Color::Cyan);
            dsf->add(flockingBoid);
            dsf->send(flockingBoid,
                      dsf->sender,
                      fbm->create,
                      new dsf::TaskArgument(std::make_tuple(flockingBoid, flockingBoids)));
        }
        dsf->duration = durationPerIterator;
        dsf->maxNumberOfCores = maxNumberOfThreads;
        dsf->start();
        delete dsf;
        delete fbm;
    }
    configure();
}
```