Destroyer2D Documentation

2015011308 唐适之

Contents

1 Introduction

Destroyer2D is a game. The players' aim is to build a ship or other types of structures, and move to the destination under bad sea conditions and enemies' attack.

Destroyer2D makes use of *LiquidFun* to provide 2D physics simulation, and *GLFW* to load *OpenGL* on different platforms (*OpenGL* provides GUI rendering). All others including gaming workflow control, human interface, game items design, etc, are done by Destroyer2D.

2 Building

This section will describe how to build *Destroyer2D*. Please notice that, Destroyer2D won't be built on a **virtual machine**, unless there is an OpenGL with high enough version. Most virtual machines provide poor OpenGL support.

2.1 Dependencies

Both of *LiquidFun* and *GLFW* are included in the *third_party* directory, so you don't have to install them by yourself. But you have to install **their** dependencies as listed below:

For Linux

- cmake (>= 2.8.12.1)
- OpenGL: libglapi-mesa (>= 8.0.4)
- Xlib: libx11-dev
- libXi: libxi-dev
- xorg-dev
- libgl1-mesa-dev

If you use *apt-get*, just type

```
sudo apt-get install cmake libglapi-mesa libx11-dev libxi-dev xorg-dev libgl1-mesa-dev
```

For Windows

- cmake (>= 2.8.12.1)
- Windows (>= 7)
- Visual Studio (2010 or 2012)

They can be downloaded from their official sites.

2.2 Building

For Linux

Execute

```
cmake .
make
```

You will get executable *bin/Destroyer2D*.

If you want to build in Debug mode, execute

```
cmake -DCMAKE_BUILD_TYPE=Debug
make
```

If you want to switch back to Release mode, execute

```
cmake -DCMAKE_BUILD_TYPE=Release
make
```

For Windows

You can use CLI cmake command as below:

For Visual Studio 2012, execute

```
cmake -G "Visual Studio 11"
```

For Visual Studio 2010, execute

```
cmake -G "Visual Studio 10"
```

If you run cmake under Cygwin, you need to unset temp variables as

```
( unset {temp,tmp,TEMP,TMP} ; cmake -G "Visual Studio 11" )
```

or

```
( unset {temp,tmp,TEMP,TMP} ; cmake -G "Visual Studio 10" )
```

You can also use cmake GUI interface, but remember to choose Release mode, or it will be very slow.

Then, double-click on *Destroyer2D.sln* to open the solution. Select *Build -> Build Solution* from the menu.

You will get executable *bin/Release/Destroyer2D*.

You can perform a Debug build as described in Linux section, or use GUI to select Debug, but you should fully clean the binaries built previously.

For OS X

Unfortunately, there is a known issue about rendering textures with OpenGL on OS X, so building on OS X is not supported.

3 Usage

This section will describe how to play this game.

First, a player will enter the building scene. He or she can pick materials from the left, which are:

- Small Wooden Block
- Large Wooden Block – These blocks can be used as scaffolding.
- Small Engine
- Large Engine – These engines are used to provide force.
- Steel Ball
- Bomb – A bomb can be ignited manually or in a crash. It can destroy items.

- Steel Stick
- Wooden Stick – Sticks can be used to connect things together.

Click on the material to get one of them, and click again on the space to put it down. Click on the right button to cancel this putting process. For sticks, the player can draw lines with the cursor to connect things together. If the player place an engine or a bomb, he or she can choose LETTER key to control it. For example, choose key 'A' to start an engine or ignite a bomb. Moreover, the button with a red cross on the lower-left can be used to delete objects. The player can also drag things with the cursor.

Note that different materials have different properties. Apparently steel is much heavier than wood, and is harder. When crash happens, things may be damaged. However, it's not apparently that a single flying steel stick has more probability to be damaged than a wooden block, because it have more momentum. Players should also pay attention to the shape of what he or she builds, because the shape of an objects in water is more important in 2D world than 3D. If finding unable to build a complex structure, using wooden block as scaffolding is suggested.

Click LAUNCH button to battle. The structure built will be saved to file, so it can be improved again if failing the battle. In battle, use the key the player set before to control the movement. Move to the right most site across the red line to win the game. The distance that have to move through is approximately five times the width of the building area. If feeling despaired to win, for example in the situation of looseing your engine, click the red cross on the upper-left to go back to building scene.

Currently there are 3 level: 0, 1, 2. When the player win in one level, he or she will enter the next. He or she can improve on the structure and battle again. However, if closing the game, the player will go back to level 0.

4 System Architecture

4.1 Main Workflow

Destroyer2D runs step by step, i.e. there is a main loop that continuously repeats. In each step, data are updated by gaming control and physical simulation and finally passed to the next step.

There are 3 classes that play as the skeleton of the main loop as three levels: *Window*, *World*, and *MainWorld*. *Window::run()*, *World::step()* and *MainWorld::step()* are directly components of the main loop. Class *Window* handles I/O with *GLFW*, i.e. sends the rendered images and retrieve user inputs. It is the most basic level. *World* maintains a generic playing scene, which can be extended to more than one type of game. It can also be easily extended to a test class. It handles the inputs, renders the items and call *LiquidFun* to do the simulation. And it is the intermediate level. *MainWorld* inherits from *World* and extends it to be a specific

game, i.e. players build structure and battle. It handles the more specific gaming control and is the top level.

The whole picture of the main loop is depicted on the diagram below.

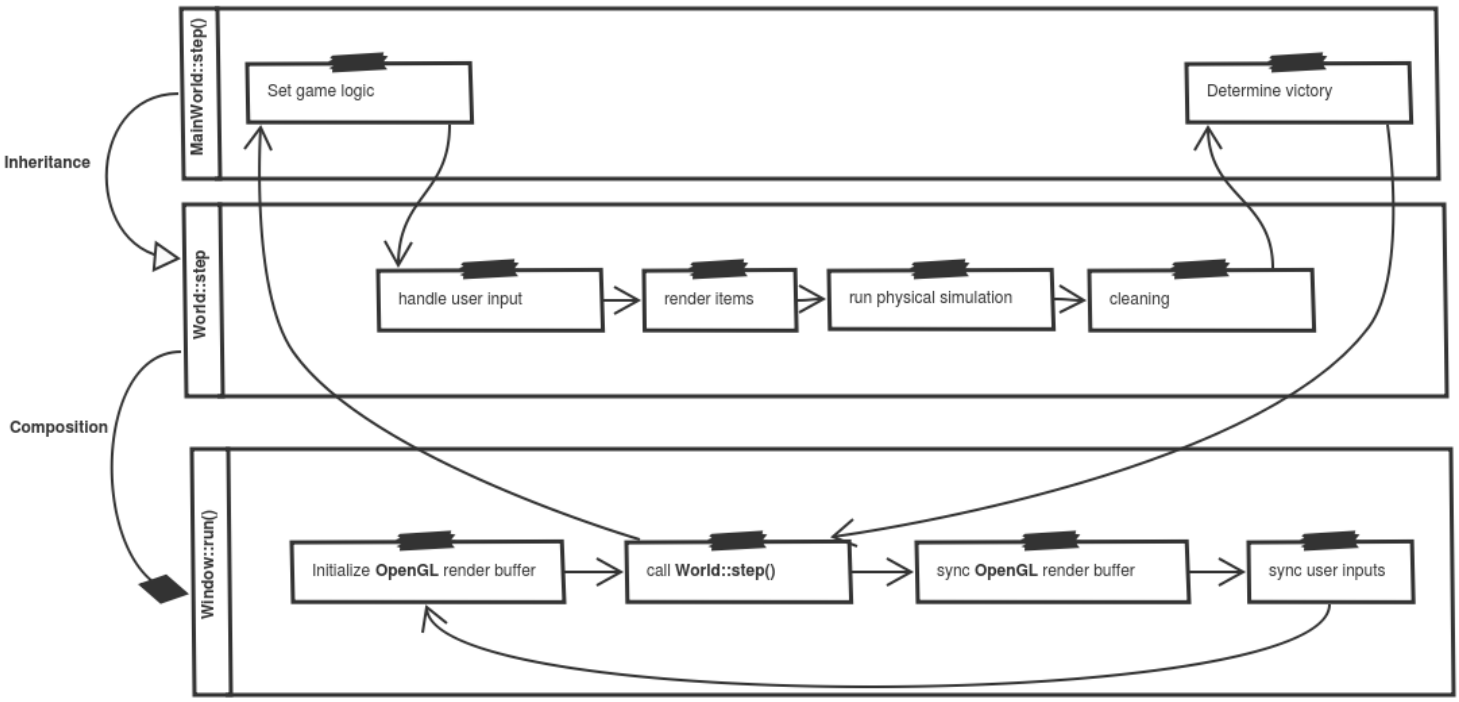

Figure 1: Main Loop

4.2 Subroutines

There are several subroutines in the main loop, some of which are relatively complex and are described below.

User inputs include keyboard input and mouse input. Raw data can be retrieve from *GLFW*. Keyboard input is relatively simple, so it can be handled directly in class *World*. Mouse input includes clicking buttons, clicking to put items, dragging items, etc. Class *MouseCallback* and its derived class are a polymorphism used to set specify changes accroding to the input. These classes are registered to class *MouseHandler* as observers, and *MouseHandler* preprocess the mouse input and call those observers. It is depicted as the diagram *MouseHandler* below.

Every body or particle system object in *LiquidFun* are connected with one *Matter* object in *Destroyer2D*. Data and strategies used in physical simulations, rendering, and gamming controls are all defined in these class. Several items are similar, so they are grouped together. Therefore, *Matter*'s polymorphism hierarchy is well organized. It is shown in the diagram *Matter* below.

Class *Render* takes charge of rendering items into images using *OpenGL* API. It can be

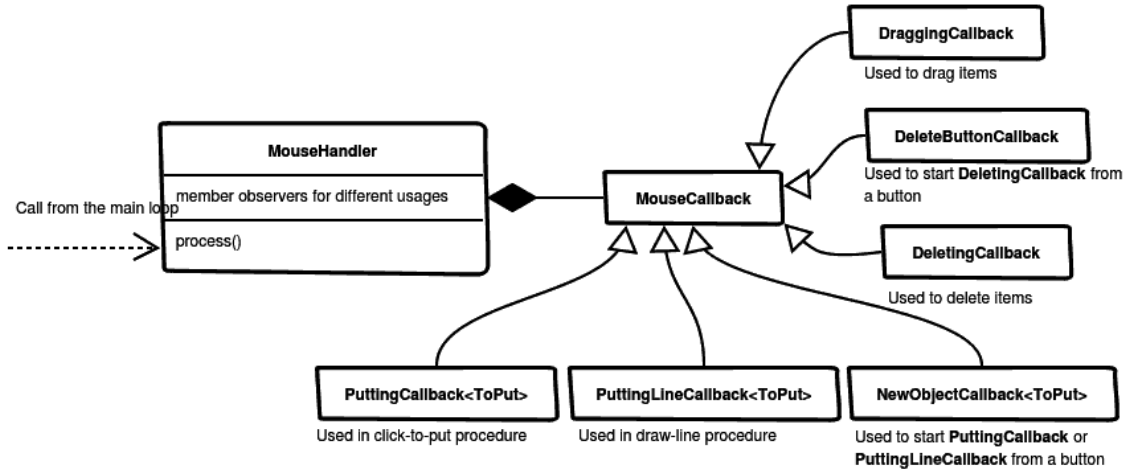

Figure 2: MouseHandler

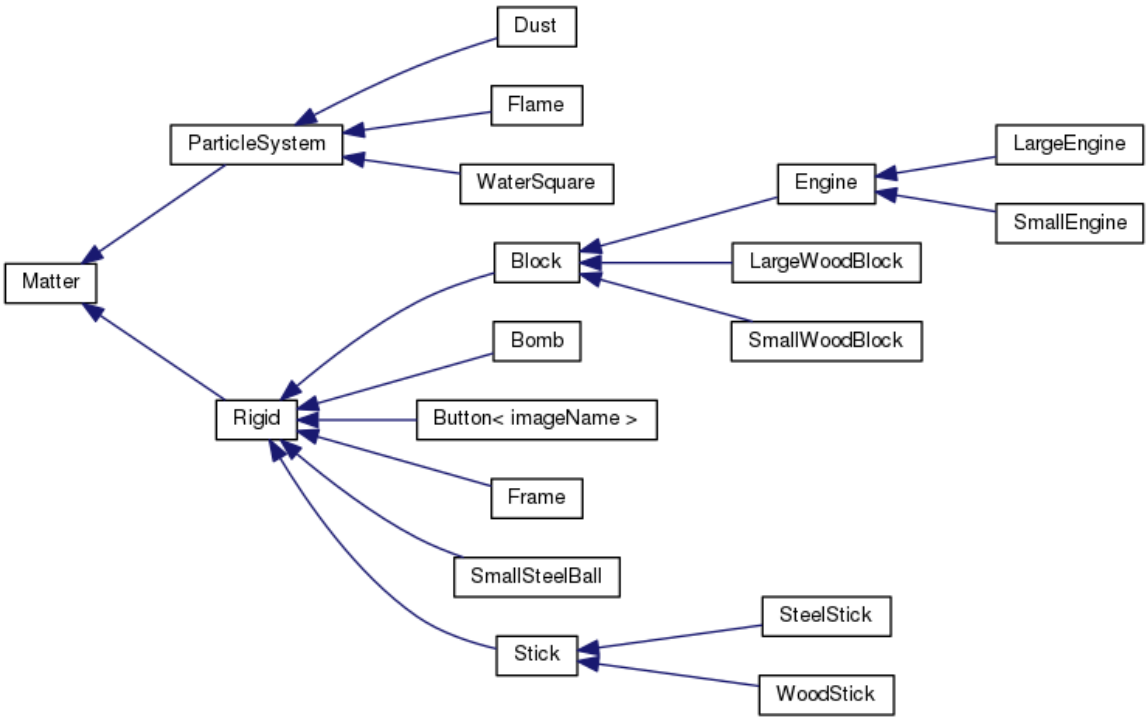

Figure 3: Matter

divided into three major parts.

1. Rendering particle systems. Particle systems especially water needs special rendering technique to display a smooth appearance. *Destroyer2D* first renders water particles into circles with gradient transparency into a buffer, then renders it again with a transparency threshold from the buffer onto the screen.

2. Rendering items. There are 4 shapes can be used in *LiquidFun*: polygon, circle, edge and chain. Currently only polygon and circle are used, and items in circle shapes are rendered as 36-gons. But for further extendence, a base class stub *Render::FixtureRenderer* is preserved and it has only one subclass *Render::PolygonRenderer*.

3. Rendering other UI components. Other UI components including pictures and texts are rendered in class *Render* for *OpenGL* doesn't provide open-use functions to render them.

5 Screenshots

Figure 4: Welcome Page

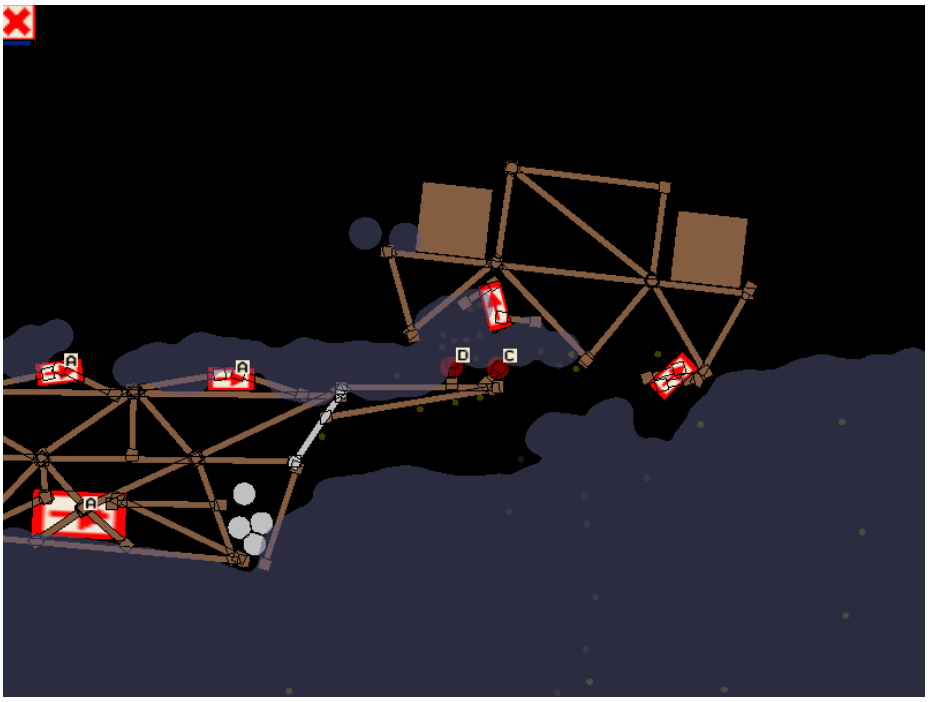

Figure 5: Battle Scene

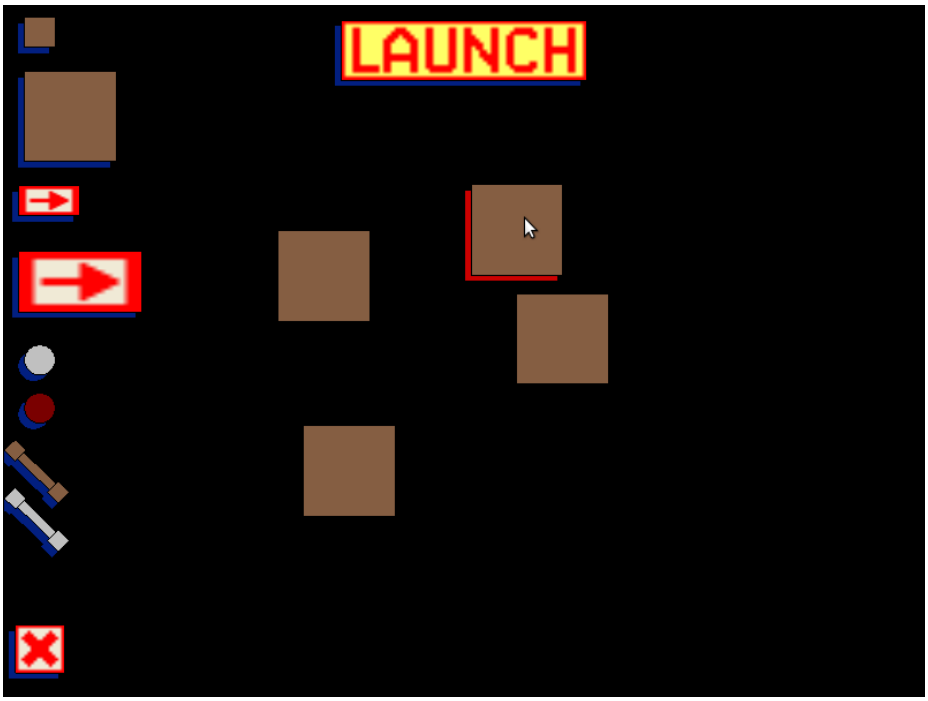

Figure 6: Putting Blocks in Building Scene

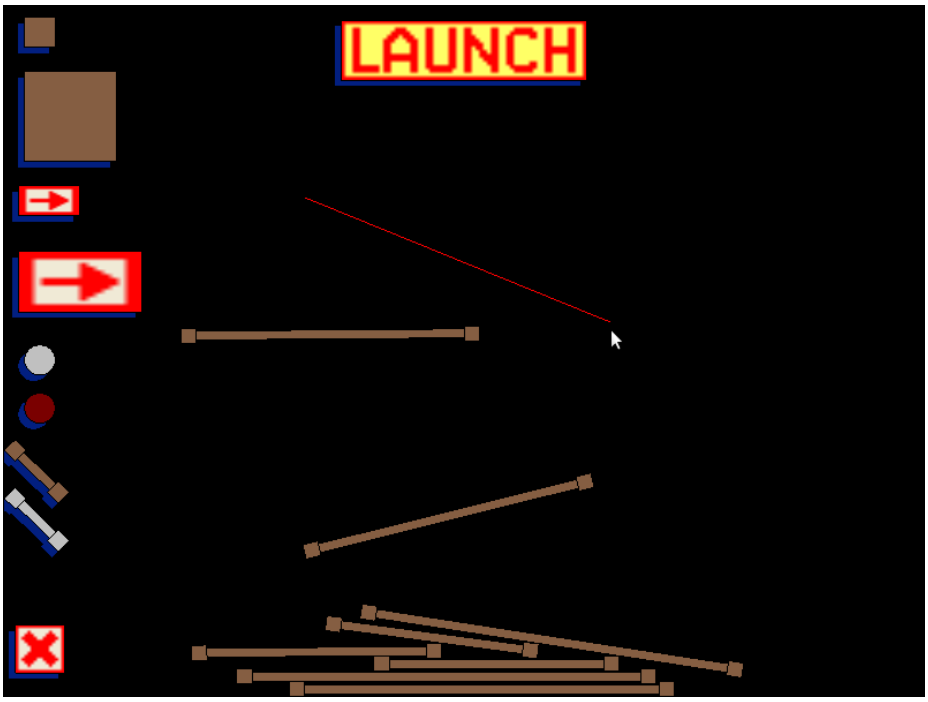

Figure 7: Drawing Sticks in Building Scene

6 Detailed Reference

Further reference to each class and function in the project can be found in *doc/reference/html/index.html*, which is generated by *Doxygen*.