Session 1: Shawn Installation

(on Windows)

Claudia Becker

Institute of Telematics, University of Lübeck

November 17th, 2008

1. A few additional tools required...
   - Cygwin and Cmake
   - JRE
   - SVN
2. Shawn Setup
   - Download Shawn
   - Generate makefile and compile Shawn
   - Import Shawn into Eclipse
3. Running a first application
   - Hello World

Outline

Install Cmake with Cygwin (1/3)

1. Install cygwin from:
   `http://cygwin.com/setup.exe`
   or:
   `http://wisebed.eu/ws08`
2. Execute `setup.exe`
   and install cygwin to
   `C:\cygwin`
   (please avoid blanks!).

Install Cmake with Cygwin (2/3)

3 Select:

- boost, boost_devel
- cairo, pixman, libpng, freetype2, fontconfig, zlib
- cmake: *A cross platform build manager*
- gcc-g++: *C++ Compiler*
- gdb
- make
- unzip

Install Cmake with Cygwin (3/3)

4 If not already included, add
C:\cygwin\bin;
to the system Path environment variable.

Install a Java Runtime Environment

To use JShawn, you need an installed Java Runtime Environment on your system (minimum required is Java 1.6). You can download it at
`http://www.java.com/en/`
or
`http://wisebed.eu/ws08`.

Install Tortoise SVN

You can download TortoiseSVN here:
`http://tortoisesvn.net/downloads`
or here:
`http://wisebed.eu/ws08`

Outline

Download Shawn source code (1/2)

1. Open your Windows explorer and create a new folder named
   Shawn
   somewhere on your disk.
   Note that the path name must not contain any spaces.

Download Shawn source code (2/2)

2 If you have installed TortoiseSVN, right-click on your Shawn-folder, select `SVN Checkout`, and use `https://shawn.svn.sourceforge.net/svnroot/shawn` as the URL for *URL of repository*.

Generate makefile (1/5)

Generate a makefile with CMake. This makefile is then used to compile Shawn with the GCC.

1. Go to
   Shawn/buildfiles
   in cygwin shell (e.g.: cd /cygdrive/C/Programme/Shawn/buildfiles)
2. Call `ccmake ../src`

Generate makefile (2/5)

3 Press c to create the initial configuration.

Generate makefile (3/5)

4 Go down to line `CONFIGURE_APPS` and press `enter` to turn it to `ON`.

5 Again, press `c` to configure.

Generate makefile (4/5)

6 Turn all needed modules (starting with `MODULE_APPS_`) to `ON`.

- `EXAMPLES`
- `READING`
- `TOPOLOGY`
- `VIS`

If you do not develop against the iSense-API, **do not** select `*_ISENSE`, because it would result in a linker error.

7 Turn `OPT_ENABLE_CAIRO` to `ON` (used by `VIS`)

8 Set

- `INCLUDE_PATH_CAIRO` to *`/usr/include/cairo`*
- `LIB_PATH_CAIRO` to *`/usr/lib`*

Generate makefile (5/5)

9 Press c again afterwards to update the configuration.

10 Finally, press g to generate the makefile that is used for compiling Shawn.

Compile Shawn

Call

```
make
```

to start the compilation process.
When finished, there is the executable `shawn.exe` located in the current directory (*Shawn/buildfiles*).

Import Shawn into Eclipse (1/9)

1. Download Eclipse from
   `http://www.eclipse.org/downloads/`.
   Either you can choose Eclipse IDE for C/C++ Developers directly, or you should additionally download CDT.

Import Shawn into Eclipse (2/9)

2 Create a new C++-project in Eclipse.

Import Shawn into Eclipse (3/9)

3. Deselect "Use default location".
4. Choose the directory that contains Shawn, and enter a project name. In addition, select "Makefile project" and "Cygwin GCC" as toolchain.

Import Shawn into Eclipse (4/9)

Import Shawn into Eclipse (5/9)

5 Create a new make target for this project in Shawn/buildfiles. For this purpose open the `Make Targets`-View, go to Shawn -> buildfiles, right-click there and choose `Add Make Target`.

Import Shawn into Eclipse (6/9)

6 To finish click the `Create`-Button.
With this standard make target, a complex process is started. Dependencies are recalculated, and the CMake build system is checked for any changes. Especially the latter behavior leads to complete recompilation of the whole code, even when a comment has been added to any CMake configuration file.

Import Shawn into Eclipse (7/9)

7 It is also possible to use a *fast* target that only recompiles any changed source file, and rebuilds the binary.

Import Shawn into Eclipse (8/9)

8 Set in project properties -> C/C++ Build:

```
Build directory: ${workspace_loc:/Shawn/buildfiles}
```

Import Shawn into Eclipse (9/9)

9 Double click on one of your new created make targets to compile Shawn.

Outline

Hello World: boot

```
1  void
2  HelloworldProcessor::
3  boot( void )
4      throw()
5  {
6      last_time_of_receive_ = simulation_round();
7      send( new HelloworldMessage );
8  }
```

Hello World: process_message

```
1   bool
2   HelloworldProcessor::
3   process_message( const ConstMessageHandle& mh )
4       throw()
5   {
6       const HelloworldMessage* hmsg =
7        dynamic_cast<const HelloworldMessage*> ( mh.get() );
8
9       if( hmsg != NULL )
10      {
11            last_time_of_receive_=simulation_round();
12            neighbours_.insert( &hmsg->source() );
13            if( owner() != hmsg->source() )
14                 cout << "ID '" << owner().label() << "' GOT HELLO FROM '"
15                  << hmsg->source().label() << "'" << endl;
16            return true;
17      }
18
19      return Processor::process_message( mh );
20  }
```

Hello World: work

```
1  void
2   HelloworldProcessor::
3   work( void )
4      throw()
5   {
6      if( simulation_round()-last_time_of_receive_ > 4 )
7         {
8            cout << "ID " << id()
9                 << " DONE: "
10                << (unsigned int)neighbours_.size()
11                << " neighbours: ";
12
13           for( std::set<const Node*>::const_iterator
14                   it    = neighbours_.begin(),
15                   first = it,
16                   endit = neighbours_.end();
17                it != endit; ++it )
18              {
19                 if( it != first )
20                    cout << ", ";
21                 cout << "'"<<(**it).label()<<"'";
22              }
23           cout << endl;
24           set_state( Inactive );
25        }
26  }
```

Run Hello World

1. Open a cygwin terminal.
2. Go to */Shawn/buildfiles*
3. Type:

   ```
   ./shawn -f ../src/apps/examples/processor/
    helloworld.conf
   ```

   to run the Hello World example.
4. You may also add the Shawn/buildfiles path to your system Path environment variable. Then go to the directory which contains helloworld.conf and run the Hello World application by typing

   ```
   shawn -f helloworld.conf
   ```

Outline

Add own applications to cmake (1/3)

Setting the *LEGACYAPPS_PATH*:

1. Create a new directory called *legacyapps* in *Shawn/src* or somewhere else on your disk.
2. Copy *apps_init.h* and *apps_init.cpp* (from /Shawn/src/apps) into your legacyapps directory and rename these files to *legacyapps_init.**.
3. Replace every occurrence of *apps* with *legacyapps* and *APPS* with *LEGACYAPPS* in these files.
4. Call *ccmake ../src* in your terminal at *Shawn/buildfiles*.
5. Enter your *LEGACYAPPS_PATH*. E.g. to */Shawn/src/legacyapps*.
6. Press c to configure.
7. Press q to quit.

Add own applications to cmake (2/3)

Creating a new application folder:

1. Create a new directory with the name of your application in *legacyapps* (e.g. *ws08*).
2. Also create a file named *module.cmake* in your new application folder. You may also copy an existing module.cmake there.
3. Insert the following text in file *module.cmake*:

```
#=============
# Shawn module configuration for cmake build system
#=============
# Name of this module
set ( moduleName WS08 )
# Default status (ON/OFF)
set ( moduleStatus OFF )
# List of libraries needed by this module, seperated by white space
set ( moduleLibs  )
```

In the case you copied an existing module.cmake file just replace the name of the module. (moduleName is the same as directory name, but written in upper case instead of lower case.)

Add own applications to cmake (3/3)

Adding the new application to cmake:

1. Call *ccmake ../src* in your terminal at *Shawn/buildfiles*.
2. Turn *CONFIGURE_LEGACYAPPS* to *ON*.
3. Press c to make your own application appear.
4. Turn your new application module to *ON* (e.g. *MODULE_LEGACYAPPS_WS08*).
5. Press c to configure.
6. Press g to generate a Shawn makefile.
7. Type *make* to compile Shawn.

Example application: ws08 (1/2)

```
1  #include "_legacyapps_enable_cmake.h"
2  #ifdef ENABLE_WS08
3
4  #include "legacyapps/ws08/ws08_processor.h"
5  #include "legacyapps/ws08/ws08_message.h"
6  #include "sys/simulation/simulation_controller.h"
7  #include "sys/node.h"
8
9  namespace ws08
10 {
11     Ws08Processor::
12     Ws08Processor()
13     {}
14     //------------------------------------------------------------------------
15     Ws08Processor::
16     ~Ws08Processor()
17     {}
18     //------------------------------------------------------------------------
19     void
20     Ws08Processor::
21     boot( void )
22         throw()
23     {
24         send( new Ws08Message() );
25     }
```

Example application: ws08 (2/2)

```
1     bool
2     Ws08Processor::
3     process_message( const shawn::ConstMessageHandle& mh )
4        throw()
5     {
6        const Ws08Message* message = dynamic_cast<const Ws08Message*> ( mh.get() );
7        if ( message != NULL )
8        {
9           INFO( logger(), owner().id() << ": Received message from " << message->source().id()
10          return true;
11       }
12
13       return shawn::Processor::process_message( mh );
14    }
15    //-----------------------------------------------------------------------
16    void
17    Ws08Processor::
18    work( void )
19       throw()
20    {}
21
22 }
23 #endif
```

Outline

Exercises

1. Install Shawn.
2. Compile Shawn.
3. Create an own application entry in cmake.
4. Compile Shawn again.
5. Run your new application in Shawn.