Session 5: Physical Environment... and Logging

Alex Kröller

Shawn Winter School 2008, 17.-21.11.08, Lübeck, Germany

November 19, 2008

1. Readings
2. Topology
3. Logging
4. Exercises

Measuring the Environment

Reading Fundamentals

- Optional application in `src/apps/reading`
- Must be explicitely enabled with `ccmake ../src`:
  `MODULE_APPS_READING ON`
- Has shrunken by $\approx 90\%$ last weekend, currently a lot of functionality is missing.

Reading Types

- A Reading is essentially a mapping from $\mathbb{R}^3$ into a value domain
- Access `SimpleReading<ValueType>::value( Vec )`,
  typical use in a Processor:
  `my_reading_->value( owner().real_position() );`
- Specializations exist for:
  - IntegerReading
  - DoubleReading
  - StringReading
  - BoolReading

Keeper Managed

- Readings are held in `ReadingKeeper`
- Accessed via their `name()`
- Add to Keeper

```
shawn::SimulationController& sc;
sc.reading_keeper_w().add( new MyReading(sc) );
```

- Get from Keeper

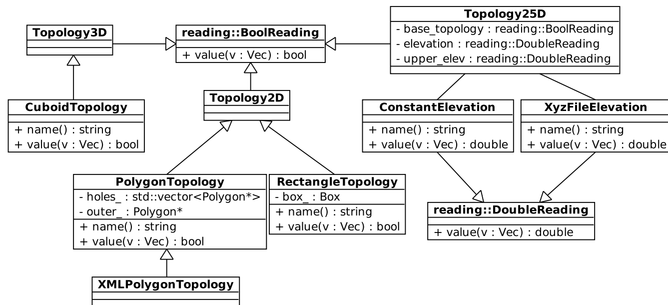
```
shawn::SimulationController& sc;
reading::ReadingHandle rh =
  sc.reading_keeper_w().find_w("my_reading");
reading::IntegerReading* ir =
  dynamic_cast<reading::IntegerReading*>( rh.get() );
if( ir != 0 )
{
    // do something
}
```

Topology Fundamentals

- Optional application in `src/apps/topology`
- Must be explicitely enabled with `ccmake ../src`:
  `MODULE_APPS_TOPOLOGY          ON`
- Advanced topology generation
- Flexible approach
  - Separated in different task areas
  - Each of them designed with Keeper Concept
- Topologies are `BoolReadings`, so they are held in the `ReadingKeeper`

Topologies (1 of 3)

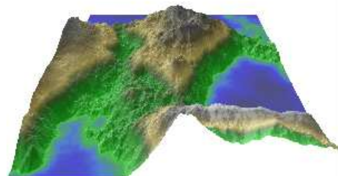
- Objective: Return for a given point if it is valid
- Each topology with an own task for generation



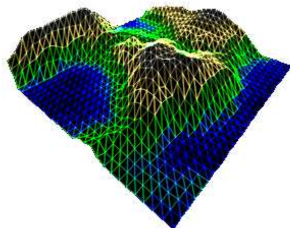
Topologies (2 of 3)

- **CuboidTopology**
  - Creates Cuboid with Task `cuboid_topology`
  - Parameters `x1`, `y1`, `z1`, `x2`, `y2`, `z2` required
- **RectangleTopology**
  - Creates Rectangle with Task `rectangle_topology`
  - Parameters `x1`, `y1`, `x2`, `y2` required
- **XMLTopology**
  - Creates topology from XML file with `xml_polygon_topology`
  - Parameter `file` required (example follows on later slide)

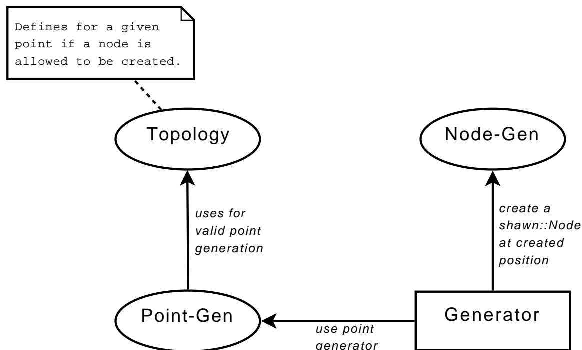
Topologies (3 of 3)



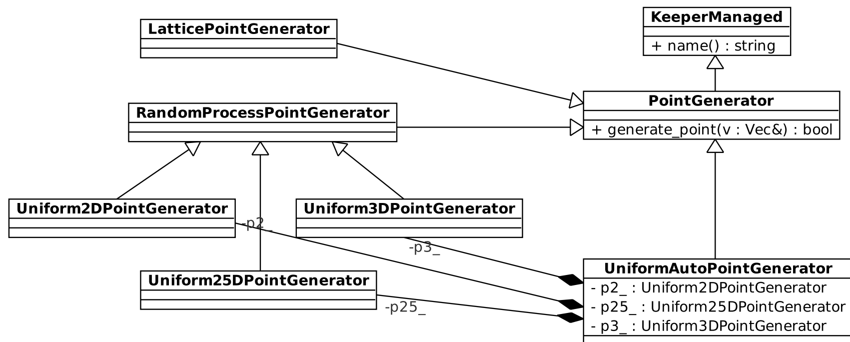
- **Topology25d**
  - Corresponding task is `topology_25d`
  - Composed of `base` Topo2D, and DoubleReading `elevation`
  - For 2D point from `base`, take z-axis from `elevation`
  - Give `upper_elevation` to elect positions between `elevation` and upper one



Interrelationship



Point Generation (1 of 2)



- `generate_point` may be an infinite process: It randomly generates points until the Topology agrees to the choice

Point Generation (2 of 2)

- **LatticePointGenerator**
  - Create points on lattice
  - Parameter `spacing` used for grid cell size
- **UniformXdPointGenerator**
  - Creates points for 2D, 25D, and 3D Topologies
  - Create points in Domain of corresponding Topology
- **UniformAutoPointGenerator**
  - Automatically uses the correct Xd-PointGenerator
  - Depending on associated Topology

Node Generation and Population

- **NodeGenerator** Create `Node`, add `Processor(s)`

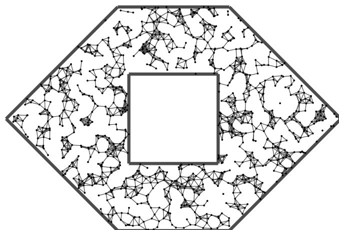
```
class NodeGenerator: public shawn::KeeperManaged {
  ...
  shawn::Node* generate_node( const shawn::Vec& );
```

- **Generator** put it all together
  - Task `populate` add `count` nodes
  - Uses `topology`, `point_gen`, and `node_gen`

Topology from XML

```
xml_polygon_topology \
  name=xml_top file=test.xml
populate topology=xml_top \
  point_gen=uniform_2d \
  count=1000 processors=

polygon_topology_postscript \
  file=test.ps topology=xml_top
```



```
<topology>
  <polygon blocking="0" type="outer">
    <vertex x="50" y="0"/>
    <vertex x="100" y="0"/>
    <vertex x="150" y="50"/>
    <vertex x="100" y="100"/>
    <vertex x="50" y="100"/>
    <vertex x="0" y="50"/>
  </polygon>
  <polygon blocking="1" type="hole">
    <vertex x="55" y="30"/>
    <vertex x="95" y="30"/>
    <vertex x="95" y="70"/>
    <vertex x="55" y="70"/>
  </polygon>
</topology>
```

Basics

- Shawn supports two alternatives for output messages:
  1. Built-in: Everything goes to `cout`, limited configurability
  2. Using log4cxx: Fine-grained control over logging streams
- To enable/configure:
  - In `ccmake ../src`, set
    `OPT_CONFIGURE_LOGGING` `ON`
  - Choose between alternatives
    `LOGGER_TYPE` `STDOUT`
    `LOG4CXX`
  - If you choose log4cxx, download and install log4cxx from `http://logging.apache.org`, configure
    `INCLUDE_PATH_LOG4CXX`
    `LIB_PATH_LOG4CXX`

Logging Usage

- Logging runs through `Logger` instances, these are
  - Any `KeeperManaged`, e.g., `SimulationTasks`, `Readings`, ...
  - `Processors`
  - `World`
  - `SimulationController`
- There are six logging functions:
  - `USER  (           message );` (for user-queried output)
  - `FATAL ( logger(), message );` (critical errors)
  - `ERROR ( logger(), message );` (non-critical errors)
  - `WARN  ( logger(), message );` (warnings)
  - `INFO  ( logger(), message );` (general bulk messages)
  - `DEBUG ( logger(), message );` (debugging information)
- Use stream operators for output, e.g.,

```
FATAL( logger(),
       "You must be " << local_drinking_age
       << " to drink here!" );
```

Simple logging configuration

- Using either stdout and log4cxx, disable `DEBUG` or `INFO` in `ccmake`:
  ```
  LOG_DEBUG_ON_RELEASE          OFF
  LOG_INFO_ON_RELEASE           ON
  ```
- Disables evaluation of all `DEBUG()` or `INFO()` calls in "release" versions, i.e., with `-DNDEBUG`.
- That's all you can do with stdout logging.

Configuring log4cxx

- Write a configuration file, load it using task
  `logging_load_cfg log_cfg_file=`*filename*
- Two separate concepts:
  - *Loggers* produce logging messages (Automagically done)
  - *Appenders* write them somewhere (Only defaults provided)

Appenders

- An appender receives logging message and writes them to the console, files, sockets, ...
- Simple appender configuration, defines appender "stdout"

```
log4j.appender.stdout=ConsoleAppender
log4j.appender.stdout.layout=SimpleLayout
```

Appenders: PatternLayout

- PatternLayout allows to configure the format:

```
log4j.appender.patlayout=ConsoleAppender
log4j.appender.patlayout.layout=PatternLayout
log4j.appender.patlayout.layout.ConversionPattern=
        %5p %c1 - %m%n
```

  With some pattern replacements:

  %p Level of the message
  %c*n* Source (Logger) of the message, truncating the name to *n* parts (e.g., "myprocessor" instead of "processors.myprocessor"
  %m The message itself
  %n A line break

- Example: `FATAL(logger(),"I'm outta here!");`
  Produces with "`%5p %c1 - %m%n`":

```
FATAL myprocessor - I'm outta here!
```

Appenders: RollingFileAppender & TTCC

- RollingFileAppender writes to files with maximum size and backup

```
log4j.appender.rfile=RollingFileAppender
log4j.appender.rfile.File=mylog.log
log4j.appender.rfile.MaxFileSize=10MB
log4j.appender.rfile.MaxBackupIndex=1
log4j.appender.rfile.layout=TTCCLayout
log4j.appender.rfile.layout.ContextPrinting=enabled
log4j.appender.rfile.layout.DateFormat=ISO8601
```

- Example: `FATAL(logger(),"I'm outta here!");`
  Produces in mylog.log:

```
2008-11-17 16:11:36,955 [0xa036ffa0] FATAL
processors.myprocessor - I'm outta here
```

Wiring logging output to appenders

- Loggers are hierarchical:
  `log4j.rootLogger` is the root,
  `log4j.logger.user` is the source for all `USER()` messages,
  `log4j.logger.processors.myproc` for Processor "myproc",
  `log4j.logger.tasks.mytask` for SimulationTask "mytask",
  `log4j.logger.simulation_controller` for the SimulationController,
  `log4j.logger.world` for the World, etc.
- For the general setting, assign
  - a logging level ("DEBUG", "INFO", ..., "FATAL", "OFF") and
  - an appender

  to the root logger:
  `log4j.rootLogger=INFO, patlayout`

Tweaking logging output

- Define an additional or specialized configuration for parts of the hierarchy:

```
# suppress all non-fatal output:
log4j.rootLogger=FATAL, appender1

# output from Processors goes to appender2 instead
log4j.logger.processors=INFO, appender2
log4j.additivity.processors=false

# All messages on all levels from Processor "myproc"
# go to an additional appender:
log4j.logger.processors.myproc=DEBUG, appender3
log4j.additivity.processors.myproc=true
```

Exercises

- Main exercise:
  1. Download `ws08-session5-template.zip`
  2. Copy `ws08_temperature_reading.*` to your ws08 app.
  3. Uncomment `sc.reading_keeper_w().add( new ws08::Ws08TemperatureReading` in `ws08_init.cpp`
  4. Change your protocol so that all nodes report the temperature of Reading "`ws08_temperature`" to the gateway.
- Extra assignments for the adventurous:
  - Write a sensor for your processor that reads the reading values, but with a probability of .1 returns meaningless values. Extend your protocol to detect outliers by comparing to the neighbor's readings. Stop sending outliers to the gateway.
  - Install and use log4cxx