Lone Sorcerer Game Framework
Getting Started

Cyrus Radfar – cyrus@cradfar.com
Helene Brashear – brashear@cc.gatech.edu

May 7th, 2006
# Table of Contents:

1 SUMMARY ........................................................................................................................................... 3

2 WHAT IS THE LONE SORCERER? ..................................................................................................... 3
   2.1 INTRODUCTION .......................................................................................................................... 3
   2.2 GOALS ........................................................................................................................................... 3
   2.3 SUMMARY OF SOLUTION ........................................................................................................... 3
   2.4 PLOT ............................................................................................................................................... 3
   2.5 GAME-PLAY .................................................................................................................................. 3
   2.6 WIZARD OF OZ INTERFACE ...................................................................................................... 4
   2.7 DISCUSSION OF GOALS ............................................................................................................. 4

3 DISCUSSION OF FRAMEWORK IMPLEMENTATION ........................................................................... 4
   3.1 IMPLEMENTATION OVERVIEW .................................................................................................. 4
   3.2 UNDERSTANDING THE PACKAGE STRUCTURE ......................................................................... 4
      3.2.1 bin/images Directory Structure .......................................................................................... 6
         3.2.1.1 animations folder ........................................................................................................... 6
         3.2.1.2 icons folder ................................................................................................................... 8
         3.2.1.3 level_* folders .............................................................................................................. 8
         3.2.1.4 rawlevels folder ........................................................................................................... 8
   3.3 UNDERSTANDING THE INHERITANCE STRUCTURE ................................................................... 8
      3.3.1 Hierarchy of ASLAttackAction ........................................................................................... 9
      3.3.2 Hierarchy of ASLAttackAction ........................................................................................... 9
      3.3.3 Hierarchy of Farmer ............................................................................................................. 9
      3.3.4 Hierarchy of DamageCausingInterface ................................................................................ 10
      3.3.5 Hierarchy of Character Behavior Interface ......................................................................... 11
      3.3.6 Character and Item Inheritance Hierarchy .......................................................................... 12
      3.3.7 Complete Networking Command Structure ....................................................................... 13

4 GETTING STARTED ............................................................................................................................. 14

5 TUTORIALS .......................................................................................................................................... 15
   5.1 ADDING A NEW ENEMY WATER MONSTER .............................................................................. 15
   5.2 HOW TO START THE GAME ...................................................................................................... 19
   5.3 HOW TO MAKE A NEW LEVEL ................................................................................................. 20
1 Summary
This document includes: an overview of the framework; its current capabilities; an explanation how the system was designed; and a few sample walkthroughs of making some changes or adding to the gaming framework.

2 What is the Lone Sorcerer?
The Lone Sorcerer will attempt to elucidate a better understanding of how to design software driven by gestures.

2.1 Introduction
American Sign Language (ASL) is the most common language in the North American Deaf community. We built a fantasy adventure game where the interaction is driven by ASL gestures, and feedback relies on neither sound nor English.

2.2 Goals
The goals for the system are three-fold:
To demonstrate that ASL can be utilized as a structured gesture command language to drive computer applications.
Learn how to design a usable gesture-driven interface
The Lone Sorcerer may demonstrate to both the gaming industry and gamers that gestures can be a fun and novel interaction method.

2.3 Summary of Solution
We have designed and implemented a fully ASL gesture-driven fantasy adventure game called the Lone Sorcerer.

2.4 Plot
The game is centered on the user’s character that attempts to take over the role of his wizardry mentor as the protector of his town. After initially learning how to spell-cast, the player must make use her/his spells and problem solving ability to destroy the Witch and her minions that torment the town.

2.5 Game-Play
The Lone Sorcerer has been designed for a full range of gaming and computing experience. The system includes predictive automatic hints and static help that respond on the users request. The user moves using a joystick and uses ASL gesture commands to command the character. All feedback to the user is either iconic or descriptive ASL videos.
2.6 Wizard of Oz Interface
To ensure that we separate the human factors issues from the ongoing research and development of the recognition system, a “Wizard of Oz” (WOz) interface has been implemented to command the game while the recognition system is trained and improved. The WOz interface has been built to allow the “Wizard” to quickly choose commands and send them to drive the game in lieu of the research recognition system.

2.7 Discussion of Goals
The data gathered by game will be used to train the backend ASL recognition system. Also, a usability evaluation will be conducted of both the user interface and user experience to elaborate on how to design a system that is ASL gesture driven. Finally, the research will demonstrate a new engaging interaction method for both gamers and the computing community at large.

3 Discussion of Framework Implementation

3.1 Implementation Overview
The system was built in Java language, Quicktime, and the SWING framework. The code-base is all built in Java. The Movies are all shown through Quicktime components, and the Game GUI is built as an extension of the SWING JFrame.

3.2 Understanding the Package Structure
![Package Structure Screenshot]

The package structure showing the networking.command files in Mac's Finder™.

There are three main folders: bin (compiled class files), doc (javadocs), src (java source files). The bin folder parallels the src folder in what it contains, other than the bin contains compiled (runnable) versions of the raw *.java files below the src folder. Therefore, we go into detail of the src folder and then only cite the differences in the bin folder.

The doc folder is where the Javadoc should be compiled. To view the Javadoc, open the file: doc/index.htm
Below is a description of the src (source) folder package structure. Note that the package names begin below the src folder, therefore all the folders listed are inside the src folder.

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gamestate</td>
<td>For all source files that control the monitoring of the game-state and context.</td>
</tr>
<tr>
<td>main</td>
<td>Holds all the source files that should be run to play the game. Specifically, ASLGame and WizardGUI. Also, this package contains the LevelBuilder, which should be used to process a level and collision map.</td>
</tr>
<tr>
<td>main.ui</td>
<td>These are the UI components used in the ASLGame GUI.</td>
</tr>
<tr>
<td>movies</td>
<td>This package contains all the source for the loading and playing of ASL videos in the game. The SignVideoPane is the extended SWING component where the video is shown to the user.</td>
</tr>
<tr>
<td>networking</td>
<td>The networking components of the GameServer and WizardClient.</td>
</tr>
<tr>
<td>networking.command</td>
<td>These are the commands that are instantiated server-side on the parsing of the command. The parsing occurs in the ASLCommand.</td>
</tr>
<tr>
<td>util</td>
<td>This package is for the placement of java classes that don’t necessarily fit elsewhere, but aid in the game. GC (stands for GameConstants) is where most of the non-movie constants are instantiated statically.</td>
</tr>
<tr>
<td>world</td>
<td>This package and it’s sub-packages includes the code control the level and the characters and objects that make it up.</td>
</tr>
<tr>
<td>world.characters</td>
<td>The characters including the user’s Player are all in this package.</td>
</tr>
<tr>
<td>world.characters.interfaces</td>
<td>These are the interfaces, which describe the functionality that the characters have.</td>
</tr>
<tr>
<td>world.characters.spell</td>
<td>This is the where the character’s spells are.</td>
</tr>
</tbody>
</table>

The main difference between the bin and src is the fact that all the images and animation frames are stored in or below the bin/images folder.
3.2.1  bin/images Directory Structure

This image depicts the structure of the bin/images/* subfolders. More specifically, it shows the path to one of the skeleton’s animation frames.

A thorough understanding of this folder structure is key to making any change to the system, which may include the modification or addition of animations, images, or icons. The remainder of this section will go into an in depth discussion of each type of folder.

3.2.1.1  animations folder

Below this folder is where all the characters and items are found. The structure is meant to allow for multiple animation types for each object. The ambiguity exists so the animation loader (world.Animation.java) can easily load the correct animations by just knowing the highest level folder name, e.g., farmer, fire, gate, skeleton, etc.

Currently, there is a lack of content and this flexibility is not being completely demonstrated. The best example is if you look at the skeleton’s animation in the screenshot above. You will notice that the folder structure, which is consistent for all the animations:

```
animation_name/row_x/dir_y/animation_name_frame.png.
```

The state_# refers to the state of the character in the animation. This allows for you to make multiple folders for different states. For example, a character may be in the spell-casting state, the idle-state, the walking-state, etc. Each one of these animations will be stored as state_x, state_y, and state_z. Because there are also multiple directions for each state, e.g. facing left, front, facing right, back to screen, etc., each state_# folder has it’s directional subfolders dir_#.

Now let’s clarify the reasoning for the seemingly ambiguous naming scheme. The animation loader attempts to load all the subfolders following the syntax stated above. Therefore, DO NOT name the folders any different than the convention of state_# and dir_. To follow the existing convention, make the idle-animation state_0, and the walking-animation state_1. Nonetheless, the naming convention becomes clearer if you look at the util.GC.java file, which contains several constants beginning with ‘ANIMATION_’. This section of util.GC.java has been copied below.
These constants can be used to remove the ambiguity from the code. The switching of animation states will be covered later.
3.2.1.2  icons folder

Icon’s images can be added simply by adding the image to the bin/images/icons/ folder. All "feedback" icons and the menu icons are stored in this folder. Below is a quick walkthrough of how to add a feedback icon:

1.) Make an icon and store it as a PNG in the bin/image/icons/ folder.
2.) Add the feedback icon to the util.GC.java file (the file was saved as my_new_feedback.png. Notice the file extension is not included.

```java
public static FeedbackIcon FEEDBACK_NEW = new FeedbackIcon("my_new_feedback");
```

3.) Then to get the image of the feedback icon from anywhere (it is statically referenced) simply call:

```java
GC.FEEDBACK_NEW.getIcon();
```

3.2.1.3  level_* folders

The level_* folders are created when you utilize the main.LevelBuilder.java class to create the levels tiles and collision maps from two initial large files located in the bin/images/rawlevels/ folder. Currently, there is only one level, called the “real_level” located in the bin/images/level_real/ folder.

The processing and building of a level is described in depth later.

3.2.1.4  rawlevels folder

This folder is where all the raw (full-size) levels are placed. Levels are initially large (4000px by 4000px) and are stored in this folder until the LevelBuilder breaks it up and stores it as separate tiles in the level_* folder. The naming of the files should be based on the level name first and then the part of the level (levelname_part).

The two parts of a level are the collision map and the image of the level. Using the current “real_level” as an example, its raw files are in the rawlevels folder. It follows the naming convention real_level.jpg and real_collisions.png. Notice that the collision map and level are stored as different types. The collision map MUST be stored as a PNG because the compression will interfere with its eventual collision map processing. Stick to the convention of levelname_level.jpg and levelname_collisions.png.

The processing and building of a level is described in depth later.

3.3 Understanding the Inheritance Structure

Inheritance is heavily utilized in this framework. So, begin by ensuring that you have a firm understanding of constructor chaining, polymorphism, and the overriding of methods.

Here are some hierarchies of the different classes.
3.3.1 Hierarchy of ASLAttackAction

The ASLTalkAction is used when the Player wants to TALK to a character. To TALK to a character the character must use a descriptor (color) that is painted behind the character they wish to talk to. The command is structured as TALK <descriptor>, for example, TALK RED. Because this command needs a descriptor it extends the abstract class ASLAbstractDescriptableAction.

3.3.2 Hierarchy of ASLAttackAction

The ASLAttackAction is used when the Player wants to ATTACK another character. To ATTACK a character the character must use both a descriptor and a spell type. The command is structured as ATTACK <SPELL NAME> <DESCRIPTOR>, for example, ATTACK FIREBALL RED. Hence, this class extends the ASLAbstractDescriptableSpellAction because the command includes a descriptor and a spell type.

3.3.3 Hierarchy of Farmer

The Farmer is a character that helps the Player throughout the game. To build a character, first decide whether the character is going to be friend or foe. The
difference is the fact that enemies may be able to be hurt and most likely will cause harm to the Player.

In this case, the Farmer can be TALKed to and EXAMINEd. Therefore, it or one of its superclasses must implement the TalkableInterface (which is a sub-interface of ExamineInterface). Learn more about the Character interfaces in §3.3.4 and §3.3.5 below.

Also, it is important to note that ALL characters and items have WorldObject as a superclass. All characters, which can be defined as objects which are animated, moveable, and can be interacted with begin with the WorldCharacter (a subclass of WorldObject). Farmer extends a character class of FriendlyCharacter, which again extends the WorldCharacter.

3.3.4 Hierarchy of DamageCausingInterface

Implementers of the DamageCausingInterface is for characters that may be able to be hurt and cause damage to the Player. The difference between this and the AttackableInterface is that implementers of the DamageCausingInterface are not drawn with descriptors.

Some example implementers BackgroundFire and InvisibleCharacter both show examples of “characters” that do not necessarily want to be drawn to show the color background and feedback icons. This interface allows for characters to look deceiving to the Player.
These interfaces are used to describe the behavior of the characters. If the character implements one of the behavior interfaces, including the overall parent `InteractiveInterface`, they include the behavior and the feedback icon when drawn to screen.

Notice that the hierarchy is a sort of force-function on what behaviors are connected. For example, anything that can be ATTACKED also can be EXAMINED. Similarly, if you can TALK to a character you can also EXAMINE it. More generally, any item which may have a descriptor or name should be EXAMINEable.
3.3.6 Character and Item Inheritance Hierarchy

The characters and objects all stem from the WorldObject. All items that can be lifted or added to the Player inventory are subclasses of the PickupItem.

All characters are subclasses of WorldCharacter. Note that it is important to note that characters are different from general objects because they may have animations, AI, can hold items, and have the concept of life/health.

Spells also extend WorldCharacter as their casting time is analogous to life, and they have animations and possibly a bit of AI. Spells are currently categorized as two types, ProjectileSpell and TileScopeSpell. Any spell that is focused and effects a single character (e.g., needs a descriptor) should extend ProjectileSpell. Any spell that effects the whole viewable tile should extend TileScopeSpell.
3.3.7 Complete Networking Command Structure

All actions begin from the WizardClient and are sent as a string to the GameServer. The GameServer acts as a receiving area and takes the command and instantiates an ASLCommand(String command, Boolean shouldParse). Here is a code snippet from the networking.GameServer.java, where the command is received:

```java
String str = br.readLine();
if (str == null) {
    return;
} else {
    if (str.equals("exit")) {
        pw.println("WAITING FOR CLIENT");
        waitForClient();
    } else {
        pw.println(str);
        new ASLCommand(str, true).execute(main);
    }
}
```

Line 11 is where all the action happens. Because all commands implement ASLAction, which compels all to implement: public void execute(ASLGame);

The actions all extend from three abstract classes:

<table>
<thead>
<tr>
<th>Abstract Class</th>
<th>Use description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASLAbstractAction</td>
<td>Subclass if no descriptor object is needed.</td>
</tr>
<tr>
<td>ASLAbstractDescriptableAction</td>
<td>Subclass if a descriptor object is needed.</td>
</tr>
<tr>
<td>ASLAbstractDescriptableSpellAction</td>
<td>Subclass if a descriptor object and a integer constant spell type is needed.</td>
</tr>
</tbody>
</table>
4 Getting Started

To understand this framework, it is important to understand how it all pieces together.

It all starts in the ASLGame. This class initially instantiates the core components. After initializing its parent JFrame first, the class continues to instantiate the SignVideoPane. Then utilizing its threaded implementation, while it plays the introduction video, it calls the loader function loadstuff().

Inside of loadstuff(), the class instantiates two of the core components the CharacterOrganizer, which stores and maintains ALL the characters and object and the World, which hold the TerrainMap that stores the TileLoader. Also, loadstuff() is where the ASLGame adds its KeyListener, which currently sends keypresses to the Player to be processed for movement (arrow keys) and pausing/playing (p – key).

It is important to understand each of these core components:

<table>
<thead>
<tr>
<th>Class</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>CharacterOrganizer</td>
<td>Stores all the characters, objects, and static Player. All the characters and objects other than the Player in a 2d array by the tile they are located on. Currently, there are 100 tiles 10 by 10, which are accessed [col][row]. Also, all the characters are instantiated and added to the game in CharacterOrganizer constructor. Note that this has a static</td>
</tr>
<tr>
<td>TerrainMap</td>
<td>This is stored within the World object, which is stored in ASLGame. The TerrainMap gives access to the TileLoader, which loads tiles and collision maps into the buffer from file.</td>
</tr>
<tr>
<td>TileLoader</td>
<td>The TileLoader loads tiles and collision maps. It also stores the BufferedImage for the current tile and the 2D collision map. The collision map is loaded from a text file and is stored in a 2D integer array. Each location in the array holds a number 0, 1, or 2. 0 means no collision, 1 means unbreakable collision, and 2 mean water collision. Currently, unbreakable and water collisions are treated equally, but there is this information if need be.</td>
</tr>
<tr>
<td>WorldObject</td>
<td>This is the superclass to all objects and characters. It holds the functionality to draw the image, store the position and much more.</td>
</tr>
<tr>
<td>WorldCharacter</td>
<td>Subclass of WorldObject, which adds functionality for movement, animations, life, hurting, healing, and item storage.</td>
</tr>
</tbody>
</table>

5 Tutorials

5.1 Adding a New Enemy Water Monster

Adding a new character should begin with deciding on how you want the character to behave. In this tutorial we will add a new Water Monster with the class name `WaterMonster.java`.

The `WaterMonster` should only be susceptible to attacks by fire (`util.GC.ATTACK_FIRE`). Also, the water monster will pace back and fourth and will have an animation for the front and back directions.

The process can be broken down into a few steps:

- Decide what to subclass/extend
- Decide on which Behavior Interfaces are necessary, and implement the necessary methods.
- Implement the `public Timer getTimer()` method and add the timer’s `ActionListener`. Make sure to call the animation’s `step()` method on each timer tick. Also, the AI should be added in here.
- Override the `boolean isEffectedBy(int)` method.
- Override the `int getFullHealth()` method.
- Make the animation frames and store them according to the animation package convention (see §3.2.1).

In this case, we will extend the `WorldCharacter` class and implement the `AttackableInterface`. By implementing the `AttackableInterface` we must implement all the methods in the `AttackableInterface`, `ExamineInterface`, and `InteractiveInterface`. These methods that you are forced to implement include

```java
public WaterMonster(String name, Point p) {super(name, p);} 
public Timer getTimer() {} 
public double getFullHealth() {} 
public boolean pause() {} 
public boolean play() {} 
public void getSurroundingGraphics(Graphics g) {} 
public int getAttackStrength() {} 
public Movie getExamination() {} 
public Color getDescriptorColor() {} 
public void setDescriptorColor(Color c) {} 
```

Then you will want to override the `public boolean isEffectedBy(int)` to ensure that it only returns true when the parameter is the `util.GC.ATTACK_FIRE`. 
Then you want to make the animation frames for both directions (front and back) views of the character walking. Then place them into the folders:

For the first frame in the front animation:
bin/images/animations/watermonster/state_1/dir_0/watermonster_1.png

For the first frame in the back animation:
bin/images/animations/watermonster/state_1/dir_3/watermonster_1.png

Here is the full code for the WaterMonster class:

```java
package world.characters;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import movies.Movie;
import movies.MovieConstants;
import util.GC;
import world.characters.interfaces.AttackableInterface;

/**
 * Simple enemy. Water Monster.
 * @author Cyrus B. Radfar
 */
public class WaterMonster extends WorldCharacter implements AttackableInterface {
    private Color myColor = null;
    /**
     * This is the number of steps the monster takes before turning
     */
    private final int PACEDISTANCE = 10;
    /**
     * Distance the monster has walked
     */
    private int displacement = 0;
    /**
     * The timer that controls the steps of the monster
     */
    private Timer myTicker = null;
    /**
     * The initial unit x direction
     */
    private double dirX = 0;
    /**
     * The initial unit y direction
     */
    private double dirY = -1;
```
/**
 * Constructor
 * @param n, name of animation
 * @param s, start position
 */
public WaterMonster(String n, Point s) {
    super(n, s);
    getAnimation().setState(GC.ANIMATION_STATE_WALK,
                            GC.ANIMATION_DIR_FRONT);
}

/**
 * Not used
 */
public void getSurroundingGraphics(Graphics g) { }

/**
 * Accessor to the timer
 */
public Timer getTimer() {
    if (myTicker==null) {
        myTicker = new Timer(150, new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                step(); //step forward in animation
                ai();  //move
            }
        });
    }
    return myTicker;
}

/**
 * Controls the movement and behavior of the character
 */
private void ai() {
    //if going down, keep going down until dist reached
    if (getAnimation().getDirection() == GC.ANIMATION_DIR_FRONT) {
        if (displacement < PACEDISTANCE) {
            dirY = -1;
        } else {
            //switch directions
            getAnimation().setState(GC.ANIMATION_STATE_WALK,
                                    GC.ANIMATION_DIR_BACK);
            dirY = 1;
        }
    } else if (getAnimation().getDirection() == GC.ANIMATION_DIR_BACK) {
        //if going up, keep going until 0
        if (displacement > 0) {
            dirY = 1;
        } else {
            //switch directions
            getAnimation().setState(GC.ANIMATION_STATE_WALK,
                                    GC.ANIMATION_DIR_FRONT);
        }
    }
}
displacement+=dirX;
move(dirX, dirY);

/**
 * Overriding the WorldCharacter method
 * @return true => attack by fire, false=>other attack type
 */
public boolean isEffectedBy(int attack) {
    if (attack == GC.ATTACK_FIRE) {
        return true;
    }
    return false;
}

/**
 * The step function moves animation
 */
public void step() {
    super.getAnimation().step();
}

/**
 * Accessor for the Maximum health
 */
public double getFullHealth() {
    return 200.0;
}

/**
 * @see world.characters.WorldCharacter#pause()
 */
public boolean pause() {
    myTicker.stop();
    return true;
}

/**
 * @see world.characters.WorldCharacter#play()
 */
public boolean play() {
    myTicker.start();
    return true;
}

/**
 * Plays the MovieConstants.EXAMINE_SKELETON movie
 * @see world.characters.interfaces.ExamineInterface#getExamination()
 */
public Movie getExamination() {

String desc = "This is a water monster, it's weakness is fire attacks.";
Movie m = new Movie(MovieConstants.EXAMINE_WATERMONSTER, desc, "Examine WaterMonster");
return m;

/**
 * Accessor for the amount of damage a skeleton causes
 * @return 5
 */
public int getAttackStrength() {
  return 5;
}

/**
 * Accessor to the descriptor color
 * @return Color of the descriptor
 */
public Color getDescriptorColor() {
  return myColor;
}

/**
 * Modifier to the descriptor color
 * @param c Color to set the descriptor
 */
public void setDescriptorColor(Color c) {
  myColor = c;
}

5.2 How to Start the Game

There are two files to run to start the game (both are run from the just inside the bin/ directory.

> java -Xmx512m main/ASLGame
> java main/WizardGUI

The first loads the game GUI and the second opens the Wizard control GUI. Note that -Xmx512m increases the java runtime heap size to 512 mb.
5.3 How to make a new Level

Making a new level has a few steps:

- Create the level and collision map in Adobe PhotoShop or any layer-based drawing program.
  - The size of the raw level is 4000 by 4000, but can be any size where both the width and height are divisible by 400.
  - You should name the level something, in this tutorial I just call it “yourname.”
  - The collision map currently consists of three colors. White RGB(255, 255, 255), Blue RGB(0, 0, 255), and Black RGB(0, 0, 0).
    - White signifies a non-collision, black is an unbreakable collision, and blue signifies water collision.
    - As a convention the upper left corner of the collision map should be black pixels (0-3, 0-3), and the next 3 by 3 square of pixels should be blue (4-7, 4-7). This is so the CollisionFileGenerator can map the colors to the RGB int values.

- Store the images as a yourname_level.jpg and the collision map as a yourname_collisions.png and store them in the bin/images/rawlevels/ folder.

- Move to the bin/ folder and run the LevelBuilder:
  - > java –Xmx512m main/LevelBuilder yourname
  - This should build a new folder below the bin/images/ directory called level_yourname.

- Goto the ASLGame() default constructor and pass in the name of your new level instead of “real_level.” OR from the command line when running the game use > java –Xmx512m main/ASLGame yourname_level