JStella Learning Environment

Tutorial:
Adding A New Game

Preface:

The JStella Learning Environment (JSLE) allows users to write agents for any game that is in the system. However, not all games could be added to the system to start with. This tutorial will go through the process step by step to add a new game. To help in this walkthrough, DonkeyKong will be used as an example game throughout.

For more information about the JSLE consult the other game source code files and the Javadoc.

To begin, a **ROM** file is needed for the game that is being added to the JSLE. This file should have a '.bin' extension and should be added to the ROM folder in the project directory (JStellaLearningEnvironment/src/ROM). This file will be referenced by the agent to load the correct game and when resetting it.

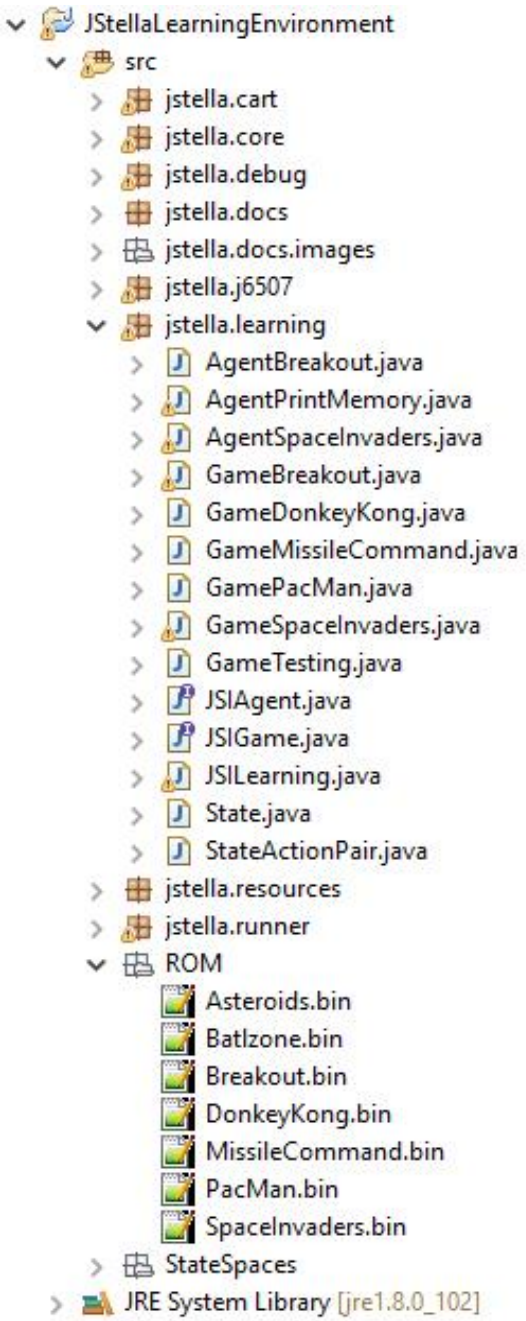

figure 1. File Structure

Next, a class will need to be created for the specific game in the **jstella.learning** package.

The class should follow the naming convention of "Game<Name of Game>.java" as can be seen in *figure 1*.

In our case: **GameDonkeyKong.java**

The class will need to **implement** the **JSIGame** interface to have the all the correct methods needed for base game functionality. This class will then be where game any game specific customization will go.

This customization will come from the learning the memory layout for this specific game. The process of determining this information is covered in detail later in this document.

The constructor for the new game class you created will always follow the same format as below:

```
package jstella.learning;

public class GameTutorial implements JSIGame {

	JSILearning JSI;

	public GameTutorial(JSILearning J){
		JSI = J;
	}

}
```

figure 2. Game Class Constructor

The set of method overrides in *figure 3* all need additional work and understanding of the games base mechanics.

The JStella emulator, like the original Atari, has 128 bytes of RAM that are used when playing a game. All the information needed by these methods exists there, however, it needs to be manually located.

Methods such as:

- getScore()
- getLives()
- getLevel()
- getPlayableStatus()

Are nearly identical across all games with the only variance being the memory locations of the values.

Methods such as:

- getState()
- computeState()

Vary vastly between games and different learning algorithms and as such have no one correct way to write them.

```
@Override
public int getScore() {
	// TODO Auto-generated method stub
	return 0;
}

@Override
public int getLives() {
	// TODO Auto-generated method stub
	return 0;
}

@Override
public int getLevel() {
	// TODO Auto-generated method stub
	return 0;
}

@Override
public int[] getState() {
	// TODO Auto-generated method stub
	return null;
}

@Override
public void computeState() {
	// TODO Auto-generated method stub

}

@Override
public boolean getPlayableStatus() {
	// TODO Auto-generated method stub
	return false;
}
```

figure 3. Game Class Methods

Before writing these methods, the game and agent classes need to be set in the **JSILearning class**. In the constructor of this class, find the following lines of code (*figure 4*) and change the game and agent names to those you will be using. This pre-built 'agent' does not do any learning, but will allow us to easily look at the games memory so the other methods can be written.

```
//Add Agent and Game------------------------------
//SELECT YOUR GAME BELOW
game = new GameDonkeyKong(this);
//SELECT YOUR AGENT BELOW
agent = new AgentPrintMemory(this);
//Add Agent and Game------------------------------
```

figure 4. JSILearning Game and Agent Selection

To find key values in the games memory such as the score or lives, the game will need to be played a few times and the output recorded. It is important to try as many things as possible when playing the game to do the following:

- Gain/Loss of score
- Gain/Loss of lives
- Lose the game (Usually the same as having no lives left)
- Win a level (If levels exist in the game)
- Move any controllable objects in obvious patterns (all the way to left, wait, all the way to the right, wait, etc..)

When these things happen, they will then need to be traced down in the games memory that is printing out. For some games this can be very easy and in others it is much more difficult. There is no best way to find things and it will require some trial and error. It is recommended to put the data into an excel document with **conditional formatting color scales** to easily see variables change over time. If you are still struggling to find the variables you are looking for you can always try printing specific memory locations while the game is running to watch them change.

Code to print memory exists in the **AgentPrintMemory** class in the following forms.
Section:

1. Will passively collect the unique values at each memory location and eventually create a text file (every x seconds) that has a sorted row for each location. Can be very useful to see the range of a specific memory value to better understand its purpose.
2. Prints out the full 128 bytes of memory each frame of the game. This is best used to track memory location over each step of the game and to find items that change over time, such as lives.
3. A section of code that allows users to directly change the games memory values. This is best used to test variables for a specific thing, such as increasing the score or lives to make sure of their position in memory.
4. A section of code that allows users to print out a single value in memory at a time. This is best used to keep track of specific portion of memory to track one or two variables.

Analyzing the games RAM

Refer to '**DonkeyKongMemoryExample.excel**'

Steps in playing the game

- Waited a few frames to start playing
- Moved around and lost a life on purpose
- Respawned and then Lost a second life on purpose
- Respawned and then completed the first level
- Started level 2 and lost my third and final life

If you pay close attention to the memory values, you can see these things happen in order. At memory position 35 you can see the players lives slowly counting down. Additionally, at position 78 you can tell when I beat the level 0 and progressed onto level 1.

The score is a little more difficult to find than some of the other fields as it will almost certainly take up multiple memory locations. In Donkey Kong, the score starts at 5000 and counts down by 100 every few seconds. This makes a clear pattern to look for in the scores value. However, since one byte can't hold the value 5000, the score will usually be divided up into 2 locations. One location for the upper bits '50' and the other for the lower bits '00'. Also, in many cases the value of the score is in hexadecimal when displayed on screen but in decimal in the memory. Donkey Kong is unique in that it only uses one memory location to store the number of 'hundreds' you have as score.

While there is no guaranteed strategy to find any of these memory values, here are some tips that work best.

1. **Have a goal**: Try and look for a specific value in a single execution of the game. For instance, keep track of the score achieved and try and make the score act in a pattern that can be seen. In this way, when analyzing the memory, the pattern can be tracked down and the value will be easier to find.
2. **Like values are close by**: For most games if an object like a ball has an X and Y position, they will be close together in memory. Use this to your advantage to find one of many values you are looking for and then search the memory locations nearby.
3. **Cross reference values**: Use all the tools you can to check the value of a memory location suspect to hold important information. The value can be printed, changed mid run, and observed during multiple runs of the game to confirm what it holds.

getScore()

The getScore() method of every game will look very similar. The difference will be how many bytes are used to make up the total score displayed on screen. For Donkey Kong we have a single byte location so the process is simplified. Examples of 2 or 3 byte scores exist in the other game classes provided.

Steps:

- Find the scores memory location(s)
- Read the location(s) as hex values using the 'getMemoryAsHex()' method
- Convert the values read into a single integer using 'Integer.parseInt()'
- Return the integer

```
public int getScore() {
	try{
		String Bits =  JSI.getMemoryAsHex()[36];
		String value = Bits + "";
		return Integer.parseInt(value);
	}
	catch(NumberFormatException e){
		return 0;
	}
}
```

figure 5. getScore Method

getLives()

The getLives() method of every game will look very similar. For any game that has lives they should be contained in a single memory location and will simply need to be returned. If they game does not have lives, simply return 0.

Steps:

- Find the lives memory location
- Return the value at this location

```
public int getLives() {
	return JSI.getMemory()[35];
}
```

Figure 6. getLives Method

getLevel()

The getLevel() method of every game will look very similar. For any game that has levels they should be contained in a single memory location and will simply need to be returned. If they game does not have levels, simply return 0.

Steps:

- Find the level memory location
- Return the value at this location

```
public int getLevel() {
	return JSI.getMemory()[78];
}
```

figure 7. getLevel Method

getPlayableStatus()

The getPlayableStatus() method of every game will look very similar. It may be more difficult to find in some game, or the lives value may be used again in this method. Regardless of the value used the general set up should be the same in every game.

Steps:

- Find the memory location that will be used to derive this status
- Return the value at this location in a way that has only two possibilities

```
public boolean getPlayableStatus() {
	return JSI.getMemory()[12] != 0;
}
```

figure 8. getPlayableStatus Method

computeState()

The computeState() method of every game will vary dramatically. This method is called before the state is sent to the agent and any action is returned. It is meant to be used to calculate any variables that cannot be directly derived from memory or take multiple frames of the game to get.

Example: You want to know the direction a ball is moving in a game, but do not want to use or do not have the ability to use a value from memory. By using the computeState() method to keep track of an X1,X2,Y1,Y2 the value can be computed passively and the retrieved when an agent gets the games state.

```
public void computeState() {

}
```

figure 9. computeState Method

getState()

This method can be customized to use as little or as much of the memory space or whatever you wish for the state. You may also wish to change it completely and have an entirely different state representation not related to the memory at all. There is no correct implementation of this method and it will change depending on the game and learning method.

For the sake of this example the games memory (RAM) is used as the state.

```
public int[] getState() {
	return JSI.getMemory();
}
```

figure 10. getState Method

getValidActions()

This method will allow an agent to get the possible actions for a specific game. The {0} action represents a player not doing anything or keeping their hands off the controls. On any game, there are at most 18 possible control inputs based on the joystick up, down, left, right, a single button, and the combinations of these actions. *(*See the JSIGame class for examples of all 18 controls)*

For Donkey Kong, it is possible to move up and down ladders, left and right on the walkways and to jump. While this means many control inputs are possible, this limited set is all that is needed as you can't jump on ladders.

```
public int[][] getValidActions() {
	return new int[][] {
		{0},
		{KeyEvent.VK_UP},
		{KeyEvent.VK_DOWN},
		{KeyEvent.VK_LEFT},
		{KeyEvent.VK_RIGHT},
		{KeyEvent.VK_SPACE},
		{KeyEvent.VK_LEFT, KeyEvent.VK_SPACE},
		{KeyEvent.VK_RIGHT, KeyEvent.VK_SPACE}
	};
}
```

figure 11. getValidActions Method