MacPac

Game Programming

Assignment 3

Name: Stuart Bryson
Student #: 98082365

MacPac was written for assignment 3 of Game Programming at UTS. Building on assignment 2, it has the following new features. Lua-bridge for debugging and game scripting,

Features

Lua-bridge

Lua is a light weight scripting language that may companies are integrating into their development pipeline and release of games. It was designed as an embeddable, extensible, dynamic language that can easily expose data structures and objects from within an application. Lua enables a quick and easy way to interface with complex run time game logic. It is also highly useful for debugging. There are many ways to interface with Lua and another programming language such as C++.

As MacPac was written in Objective-C, it is very easy to interface with Lua. Objective-C is binds messages to implementations at runtime, not compile or link time. All messages sent between objects in Objective-C use the following function.

OBJC_EXPORT id objc_msgSend(id self, SEL op, …);

The Objective-C runtime also makes the method signatures available. With this understanding of the Objective-C runtime, we can write an interface such that all Objective-C runtime messages are made available to Lua. We can now simply push Objective-C objects onto the Lua stack and Lua can then manipulate these object at runtime. In fact, we can even create new or sub class existing Objective-C classes all from Lua. This enables fantastic game expandability in the future.

MacPac currently has limited application of this bridge, although no doubt will be extended in future. When loaded, the game makes available the Controller class and the Scene class to Lua. This way, absolutely every message of each of these classes is available to Lua. Some examples of how to see this in action follow.

MacPac also provides a utilities.lua script which gets sourced when the game launches. This utilities would be an ideal location to store commonly used functions or game logic that could in future be changed without requiring a recompilation.

Lastly, MacPac has a Lua console to provide a simple interface to the user to enter Lua commands. Example commands that may be used in MacPac follow.

Lua Console

The Lua-bridge simply maps the Objective-C messages into an acceptable Lua syntax. For example, the Objective-C message:

[scene setSize:5]

would translate into the following Lua code:

Scene:setSize(5)

Other Lua commands that can be tried include:

```
Scene:spawnEnemy()  -- remember there is only a maximum of 4 enemies
Scene:removeAllEnemies()
Scene:removeAllPellets()
Scene:endGame()

Controller:toggleWireframe()
Controller:toggleDemo()
```

Feel free to have a look through the Scene and Controller header file to try some others.

Active Enemies

MacPac employs various techniques to enable active enemies that pursue the player and kill them or the player in turn may pursue them. They bring the game a more active and intelligent feel. Following is a discussion of the features required to implement active enemies.

A Path Finding*

In order to provide active enemies that pursue the player, some sort of path finding algorithm needs to be employed. The A* algorithm has proven very popular as it is a fast and fairly accurate method to finding the best path from a to b. In fact, if we assume that our heuristic in the search is admissible, that is it never overestimates its cost to the goal, then we are guaranteed to find the shortest path in the best possible time.

MacPac employs the A* path finding algorithm and indeed uses an admissible heuristic. For this reason, calculating the best path for each enemy to the player each frame is more than feasible.

In order for A* to work, we need a set of interconnected nodes or points that can be used to determine path. As MacPac is essentially a grid, it is easy to determine the points and how they are connected. Currently, a maze is procedurally generated. During this

generation, we simply build up a list of points and a list of other points that a given point is connected to.

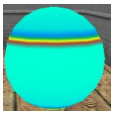
In order to test that the set of A* points have been generated correctly and also that our chosen path has been correctly generated, it is important to be able to visualise what our paths and their points look like. For this reason there is an option to visualise the generated interconnected path points. There is also a relatively minor adjustment to the code to visualise an A* path. A path simply needs to be set in SBGLView and the path will be drawn.

Finite State Machine

Another requirement for active enemies is to implement a basic finite state machine. This state machine will be responsible for controlling the states and behaviours of the enemies.

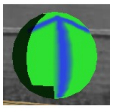
The states and behaviours of the enemies follow:

- Attacking. This is the enemies default state. When the enemies are in their attacking state, they employ A* path finding to pursue the player. Their speed is slightly faster than most other states and when they collide with the player they will kill the player.
- Retreating. This state occurs when the player eats a retreating power up. When in this state, the enemies turn sad and blue and they are edible. That is, when the player collides with them, they will eat the enemy, the enemy will then dissapear and respawn at a later time.



- Bouncing: This state occurs when the player eats a bouncing power up. When in this state, the enemies are still dangerous and if the player collides with one, the player will die. However, the enemies are not actively pursuing the player. Rather they enemies will bounce up and down on the spot giving the player enough time to duck under them.

- Elevated: This state occurs when the player eats an elevated power up. This state is similar to the bouncing state in that the enemies are still dangerous, however the enemies do not bounce, rather they are momentarily elevated so the player can easily move under them.



- Eating: This state occurs when the player eats an eating power up. When in this state, the enemies will eat any pellets they collide with. This enables the player to finish the game (or level) much faster. Unfortunately the player doesn't earn points for the pellets that the enemies collide with.

All states other than the attacking state last for a certain amount of time and then return to the attacking state. When returning to the attacking state, there is a nice blend between the current and attacking state. For example, if the enemy is in the elevated state in the air and they are then switched to the attacking state, they will start to descend from their elevated position and at the same time, begin to pursue the player along their calculated A* path.
All states are also only activated when there are spawned enemies in the scene.

Lights

MacPac implements basic per-vertex lighting. There is one directional light created when the game launches. The game uses the built-in lights in openGL so therefore supports up to 8 lights. There a light class for storing the light parameters, an rgb vector class for storing the colours and applying colour calculations and the ability to add multiple lights to the scene.

Not only are there basic static lights, but MacPac supports different lighting effects too. The effects can be seen when the player eats the various different power ups. Each power up has its own strobe of colour. When eaten, these colours will strobe back and forth indicating that there are some enemies in the scene that are in this state.

Not all the enemies necessarily will be in this state though. Some may have spawned after the power up was eaten.

Sounds

MacPac employs basic sounds to provide a more immersive experience for the user. It provides audible feedback for when different pellets, powerup or enemies are eaten, for when enemies spawn, and for when the player dies or the game finishes.

Game Play

Lastly, MacPac implements some basic game play features. These features include:

- A counter of remaining lives that the player has. The player begins with 3 lives at the start of the game and will be deducted a life when they collide with an attacking enemy. The number of remaining lives is visible on screen.
- A score. Each time a pellet is eaten, the player gains 5 points. Each time a retreating enemy is eaten, the player gains 20 points. The number of points gained is visible on screen.