BURLAP BAG
An Adjustable Game Engine for the BURLAP framework

Lee Painton
Email: lee_painton@gmail.com
As supervised by Michael Littman
Submitted to the Brown University CS department
as fullfillment of Masters Program requirements

***Abstract*—The BURLAP Adjustable Game engine, or BAG, is a custom game engine written in Java and integrated with the Brown-UMBC Reinforcement Learning and Planning Framework. The purpose of BAG is to provide a framework for the creation of games integrated with BURLAP domains and thus simplify the use of BURLAP driven agents in scenarios where a game is to be used for research purposes.**

I. Introduction

In AI research it can be useful to model the interaction between agent and environment or between multiple agents in the context of a game. For example, if an AI agent is tasked with navigating a maze or perhaps a human and AI agent must cooperate on solving a puzzle. In such cases it can be simplest to model the environment of the task in the form of a game, thus giving the investigator complete control over the parameters of the environment as well as the permitted actions of the agents.

The design of BAG attempts to anticipate the various needs of researchers and strike a balance between providing templates for the quick production of simple games and allowing enough flexibility to be highly customizable. As a Java based library BAG is portable to any system which can also run BURLAP.

II. Overview

The BAG library consists of the core BAG module as well as several example modules. The core module can be broken down into multiple sub-components. They are itemized in the following sections by functionality:

A. Core Game Engine

Instantiated as the BAGame class, the core engine which directs the processing of game logic and the order in which various components are updated and rendered, as well as providing services between components. Each BAGame object serves as a discrete instance of a game and all components in it are independent from components in other instances. Additionally, core components managed and linked to by the game engine can easily be replaced by custom components inheriting from those provided in the framework. All game specific objects handled by the engine are derived from the BAGObject class.

B. Graphics Engine

The Painter class provides a two-dimensional graphics engine which offers rendering services for bitmaps and text. Objects requiring rendering implement the IPaintable interface and register with the core engine, at which point during the rendering phase the objects are called on in no particular order and can electively submit rendering jobs to the Painter. The Painter stores rendering jobs in a priority queue based on a layer attribute and draws textures in a back-to-front method based on descending layer priority, thus textures with higher layer numbers can be hidden by overlapping textures submitted in lower layers. The Painter aggregates the provided rasters to produce a final rasterized array of pixel information which is passed directly to the graphics context. Objects can also request typesetting jobs from the Painter which will draw strings to the screen in a default font.

Also critical to the functions of the rendering component is the Texture class. This class acts as a data structure for storing bitmap information as well as a content pipeline for loading external bitmap files stored in the image directory. The Texture class also provides methods for creating primitive bitmap textures and deriving new textures from transformations of existing ones.

C. Collision Detection

BAG handles all collisions using the CollisionMap class. BAG uses a combination of object coordinates and Texture metrics to define collision bodies. BAGObjects which are flagged as collidable are automatically subscribed to the CollisionMap, which maps each object to a rectangular collision body. Actual collision checking is reactive and the responsibility for initiating a collision check and resolution of the result comes each object seeking collision information. Additionally an object can check for intersections which are not counted as collisions but used the same logic for detection. The CollisionMap maintains a list of detected collisions which is refreshed after each update cycle.

D. Input Detection

The InputMap class provides close to real time information on subscribed keyboard and mouse driven events. Objects needing information on keyboard or mouse inputs request monitoring of either specific keyboard or general mouse events. They can then poll the InputMap for information on the status of those events, the resolution of which is dependent on the update logic of the particular BAGObject.

E. BURLAP Domain Integration

Acting as bridge between BURLAP and BAG logic, the BAGDomain class provides wrapper methods for interfacing with BURLAP domains as well as acting as a BURLAP domain generator. The BAG framework integrates with domains in two key ways: one by primarily storing agent and location attributes in the domain and two by relying on BURLAP Actions to effect change in the game world. The BAGAgent and BAGLocation classes by default map themselves directly into the active BAGDomain. BAGAgent also provides pre-generated implementations of common movement actions which are executed via the domain. BAGDomain automatically updates a list of instantiated PropositionalFunctions after each State change which it stores as GroundedProp objects for evaluation.

F. Trigger System

Objects of the BAGTrigger class are intended to act as a form of scripting system for BAG. Each trigger consists of an isSatisfied flag, an evalution method and an resolution method. The core engine evaluates all registered triggers at the end of each update cycle. It then builds a list of any triggers that have ben satisfied and resolves those in a separate loop.

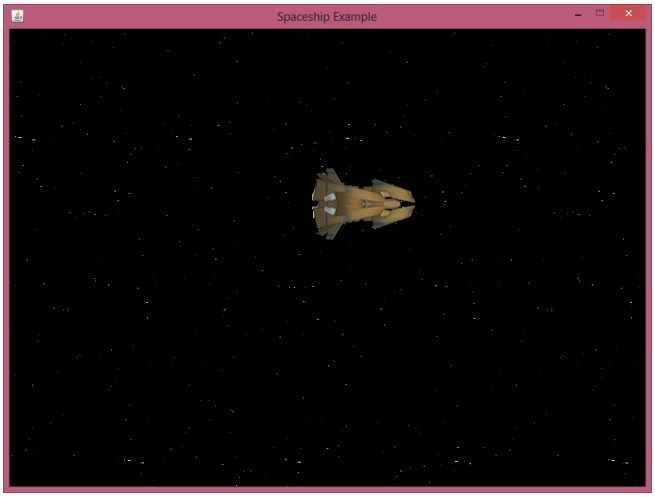
G. Map System

The BAGMap class is intended for the management of objects which, while grounded in the game world, have no direct correlation to BURLAP objects. A BAGMap has two major methods, the primary of which is generateMap. The game engine calls this when the map is first loaded and it is responsible for creating all necessary BAGObjects to represent the world map. The second method is a stub called renderMap which is only used for games which have their render mode set to map based rendering. This allows the map to bypass the normal painter and take over texture rendering for the game world. An example of this is seen in MazeGame where the map handles raycast rendering for the game world.

III. Examples

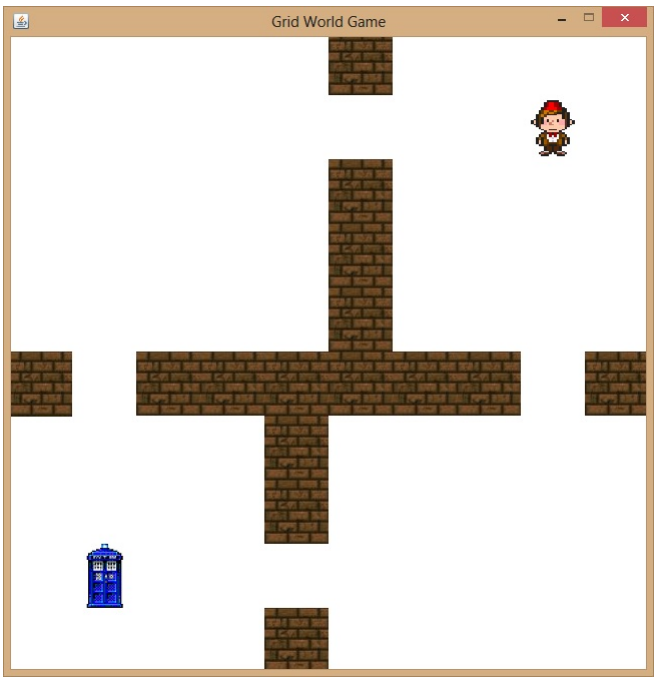
Three example games are provided with the framework to illustrate the simplicity and flexibility of composing games in BAG. Two of them serve as templates for games while the third is based off a BURLAP domain.

A. ShipGame



The ShipGame package is an example of a mapless game and serves as a template a free-roaming game with no map constraints. It uses a vehicle style of movement where the ship can move forwards and backwards or turn left and right. Thus all orientations are relative to the vehicle and not based on map coordinates. It can be found in the examples.shipgame package.

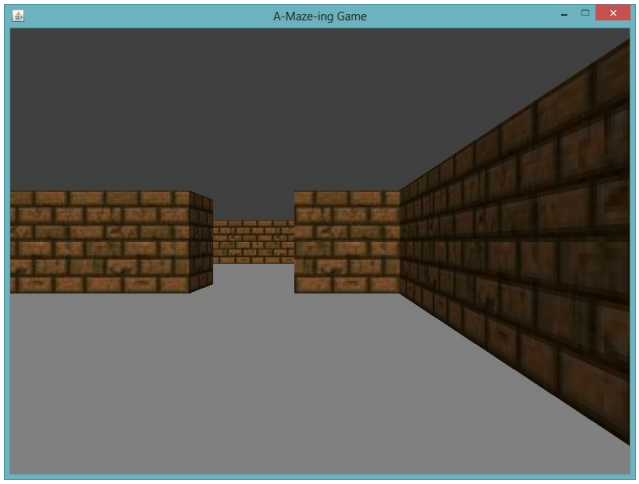
B. GridGame



This game is based on the GridWorld example presented in the BURLAP tutorial webpage. While BAG calculates movement based on pixels, many of its pre-generated methods and objects are designed to support grid based games. In this case the movement is North, South, East, West and the game calls appropriate BURLAP actions to achieve state transitions. The game also has a terminal condition set by means of a BAGTrigger.

The victory screen demonstrates the use of a BAGUIElement which generates a blue box texture and typesetting for itself. GridGame can be found in the examples.gridgame package.

C. MazeGame



MazeGame primarily serves to demonstrate the use of a custom map render to provide alternative graphics functionality, in this case raycasting for a 3D-like effect. The world displayed is generated from a 2 by 2 array using the RaycastMap class included with the framework. MazeGame can be found in the examples.mazegame package.

IV. Future Work

BAG serves its purpose well for the quick generation of games with simple parameters. That said, there's work to be done in adding breadth to its functionality. In designing BAG it was assumed that games would mostly be limited to one game map per domain, however domains which span multiple maps are certainly possible and better support for transitions between maps should be added to the core engine.

The collision system is simplistic and defers all questions of game physics to the objects requesting collision information. Additionally, collision checking has an asymptotic runtime of $O(n^2)$, though this will generally amount to $O(k * n)$ where k is the number of objects requesting collision data each cycle. As k grows this will eventually bog down, and thus the use of a quadtree for organizing collision bodies would be advisable.

Adding better support for continuous domains would be helpful for games which don't fit the grid format. MazeGame, for example, uses continuous movement but is not well supported by the framework's built in domain objects. Actions would need to be added supporting movement along a vector. Another part of this would involve expanding support for more Texture manipulations. Rotating textures by radians and scaling would all be useful applications for continuous domains and are not currently possible in BAG.

Finally, no networking functionality is included with BAG and this would be useful, along with client and server implementations, to support deployments for research requiring large numbers of subjects.

V. Summary

BURLAP BAG is a lightweight game engine designed for the rapid development and deployment of games integrated with the BURLAP framework. It is coded in Java to maximize compatability with BURLAP and includes many built-in objects and functions to make creating BURLAP enabled games quick and easy.

Acknowledgments

I would like to acknowldge my advisor, Michael Littman, without whom this project would not have happened. Additional thanks go to James MacGlashan for his fine work on BURLAP.

References

[1] MacGlashan, J. (BURLAP) Brown-Umbc Reinforcement Learning and Planning. http://burlap.cs.brown.edu/.