Abstract — Planning as an artificial intelligence technique in video games is becoming more and more popular. Many modern video games are now using various kinds of planning for their non-playable characters to drive interesting and engaging behaviours. This paper looks at the implementation of Sam Town, a small town simulation game that uses STRIPS planning to determine how agents in the town should act. The implementation of Sam Town was successful, but is not perfect, and this paper will look at these issues and how further work for a small town environment could be planned over.

Keywords—planning; STRIPS; games; artificial intelligence; town simulation

I. INTRODUCTION

Modern video games are using the artificial intelligence technique of planning to feature more intelligent and engaging non-playable characters that either work with or against the player. Many first person shooters now use planning of some description to create effective multiplayer bots. However, video games that use planning are not limited to first person shooters, with games like SimCity using it to solve problems – SimCity is the main inspiration for this work.

This paper will first look at what planning actually is, what the different types of planning are and how video games currently use it. It will then look at the implementation of a small town simulator, Sam Town, which uses planning to drive the behaviour and actions of people who exist in town. Finally, the paper will analyse how successful the implementation was, and where future work could take it to improve it.

II. RELATED LITERATURE AND WORK

A. Planning and Video Games

Planning can be described as “devising a plan of action achieve one’s goals” [1], so planning can be used to solve problems given an appropriate representation of world and the problem. Traditionally, planning was (and still is) being used to solve real world problems, such as with programming robots.

Planning can help video games provide interesting and engaging experiences by helping drive non-playable characters decisions and actions. Video games that use planning to drive their AI can be split into roughly two categories [2]: games that use STRIPS-style planning and games that use HTN-style planning.

STRIPS was the first major planning system implemented in 1970 to solve problems in robotics [7]. A STRIPS-style problem has a series of goals and actions: goals describe parts of the world state; and actions are defined in terms of preconditions that must be satisfied in order for the action to execute, and effects to the world state that happen if the action is indeed executed. The first game to use STRIPS-style planning successfully was F.E.A.R [3], which uses a STRIPS-style planner to drive all non-playable characters using the same system. This allows for emergent and interesting behaviours to appear from the characters as they determine what to do by themselves, rather than designers having to script this behaviour.

STRIPS-style problems can be expressed using a language known as PDDL, or the problem domain definition language, which was derived from the original STRIPS planning language [7]. PDDL domains express the predicates and actions available to problems, and PDDL problems express the initial world state and the eventual goal state using the predicates expressed in the domain file.

Different STRIPS planner implementations tackle the solving of problems in different ways, using A* with heuristics to either search backward from the goal state to the initial state, or to search from the initial state to the goal, where the former is faster but the latter is more flexible.

Various STRIPS planners exist, including FF [4] (Fast-Forward), and Java FF [5] (a Java implementation of the FF planner).

Video games that use STRIPS-style planning are becoming less frequent in comparison to other kinds of planning, but games that do use are praised by players and reviewers, often in open world games and games that promote emergent behaviours in comparison to more scripted action sequences [2].

Where STRIPS-style planning looks at very specific tasks, HTN-style planning, where HTN stands for hierarchical task networks, takes a higher and more abstract view of a problem [1]. This style of planning has actions remain abstract until they are executed, in which they are decomposed into more specific actions. These actions may be able to decompose further into even more specific actions, or they may be a primitive action that can simply be executed.
One of the first games that used HTN planning for their games non-playable character controllers was Killzone 2 [2], specifically, the multiplayer bots [6]. The more high level actions decided what the AI should do, and the more primitive actions looked at how the task it selected should be performed.

Regardless of the implementation or type, using planning allows for more flexible non-playable characters in video games that exhibit interest and emergent behaviours.

B. Planning and City Simulation Games

SimCity [7] is a city simulation video game that uses planning to a great extent, released recently in March 2013. Willmott's talk at GDC 2012 covered the game engine that powers SimCity, GlassBox, in great detail [8]. GlassBox is a new engine built for future Maxis games. It solves problems with its own custom planner and problem language (as opposed to using PDDL) to make the simulation come alive. Willmott defines several components that make up the SimCity world simulation. Some key ones include:

- Resources, which are defined as basic currencies in the world, such as an amount of coal, oil or water.
- Units which represent things in the world, such as factories, houses or people. Units have a number of resources associated with them and an internal state.
- Rules, which define and solve the problems of the SimCity simulation. They operate on resources and can perform different tasks, such as moving resources around the world or converting resources from one type to another. Rules can also give feedback to the user through audio or visual effects.
- Agents are created by rules, and are given destinations to head towards. SimCity uses a D* Lite based algorithm to plot paths.

These key components and others combine to create the simulation behind GlassBox, which was released to critical acclaim before online issues and bugs lowered the reputation somewhat [9]. Willmott [8] concludes with the phrase "What You See Is What You Sim", describing how SimCity doesn't visualise statistics but displays the actual state of the world.

C. Summary

The use of planning as an artificial intelligence technique allows real world problems to be solved. Specifically looking at the domain of video games, planning allows for them to provide rich and interactive experiences, with non-playable characters appearing to be more believable by providing emergent behaviour and realistic decision making.

As a genre, simulation games need to use planning of some description to provide an engaging experience for the player.

With this in mind, this work intends to create a simple simulation environment of a town, using planning to solve the problem of what agents want to do.

III. METHODOLOGY AND IMPLEMENTATION

From the initial idea of a small town being set, different planning approaches were looked to identify how suitable they were for implementation within the small time period available for the work.

Early in the development process, the simpler STRIPS-style problems were decided as being the most effective for the implementation and most suitable for the length of time available, and thus work was initiated into creating a problem domain for a town. Haslum’s PDDL overview [10] was used to identify the general structure of PDDL problem domain files. To facilitate the writing of the problem domain, the PDDL Studio [11] development environment was utilised, using features such as syntax highlighting, code completion and semantic checking to expedite the process.

The final PDDL domain created had a number of features:

- As a domain type, it has the ":strips" requirement for basic STRIPS functionality, and the ":typing" requirement for strongly typed objects.
- With the "typing" requirement, a series of objects were set. Base objects were person and location. To support the problems, a series of typed locations were added, including a generic function, path and house for basic navigation. Twelve locations based on common buildings from both classic towns and villages to modern ones were also added, such as a butcher, a greengrocers, and a gym.
- A number of predicates were added. The first predicate, "at" simply describes the current location of a person. The second predicate "connected" describes how two locations are connected via a path. Beyond these two predicates, are twelve others, one for each strongly typed location. These predicates signify that a person has been active at that location. For example, after a person has been to a butcher, the "hasMeat" predicate will be set.
- To fulfil and manipulate the predicates, various actions were implemented. The first action "MOVE" simply moves a person from one location to another provided they are connected by a path. Similar to the predicates and locations there were also twelve actions added, where each action fulfilled a specific location's predicate. For example, the action "ACTION-GETMEAT" sets the "hasMeat" predicate provided that the given person is at a butcher.

This problem domain would allow for a simple and abstract model of a town to be represented by PDDL problems and solved using a STRIPS planner.

To represent the world, the Cocos2d-x [12] game engine was utilised, which is an open source and cross platform 2D game engine written in C++. A World class was implemented, which describes a simple 2D grid structure for entities. The World maintains a list of different location and path structs: these structs were simple and contained some basic variables to describe the location or path in question.
• A location has an integer x and y co-ordinate pair that represents the position in the grid that it occupies, and a type that represents the domain location type.

• A path has two integer x and y co-ordinate pairs that describe the start and end positions for the path in question. For every co-ordinate pair added to the World, two paths were created to facilitate movement in both directions. While the PDDL domain and World could theoretically support one-way movement on a path, support was not implemented to display this to the user.

With the World successfully managing different entities, support for agents and PDDL problem creation functionality was added. As the World knows what kind of buildings exist within itself, agents can query the world to identify what goals they can aim for in the form of the domain predicates for any given location. Any time an agent queries the World for a list of goals, they are given a number of goals based on the number of unique types of buildings available, and won’t be given any goals they have recently completed to make behaviour more varied.

As problems were being created, a planner was needed to be able to solve them. Many planners were considered, but the Java-FF [3] planner was chosen for how it was easy it was to get working; other planners like FF [4] were problematic and simply would not work. Unfortunately, as Java-FF is implemented in Java, it could not easily be accessed from the C++ where a much more varied system was expected, support was not implemented to display this to the user.

As problems were being created, a planner was needed to be able to solve them. Many planners were considered, but the Java-FF [3] planner was chosen for how it was easy it was to get working; other planners like FF [4] were problematic and simply would not work. Unfortunately, as Java-FF is implemented in Java, it could not easily be accessed from the C++ where a much more varied system was expected, support was not implemented to display this to the user.

The World parses the solution, looking at each line of it to identify if a given line is a "MOVE" action or an "ACTION" action, and populates an Action struct with the data contained on that line. The list of these structs forms a queue for the agent to follow.

The world was also created in such a way that the user can modify the environment, by adding more locations and paths, to allow the user to be able to have more control of the simulation.

IV. ANALYSIS

Sam Town was a success in regards to what the work set out to create, which was a small environment in the form of a town in which agents wander about and fulfill objectives. However, as an implementation, there are some problematic areas.

The planner integration leaves a lot to be desired. A planner written in C++, or an environment written in Java would have allowed for a tighter and more efficient implementation. Fairly large bottlenecks are introduced due to the overhead of the file input and output operations, in addition to the memory overhead associated with the creation of the additional threads and processes to create problems and determine solutions. As the
amount of buildings and connections between them in the town gets larger and larger, the plan time also increases.

Despite this, the solution still scales fairly well. Fig. 2 shows a much larger town of 30 agents and a similar number of locations, in comparison to Fig 1, which has only 1 agent and 4 locations. For this town, there is negligible lag or performance drops caused by the planning and solution reading process, even on the ~5 year old computer used for development.

Some issues also exist with the planner. At times, it can be seen that agents do not take the most optimal path, instead it often opts to take longer routes that still satisfy the solution. While this behaviour may be interesting and wanted, such as the villagers opting to take a scenic route as opposed to a direct one, the root cause for this has not been determined.

Goal selection is another area that could use a more thorough approach. Though agents do select goals that they have not completed recently, their movements are still random and unpredictable.

V. CONCLUSIONS

This paper looked at how planning is used currently video games, which appears to be mainly driving multiplayer bots in first person shooters, but also is being used in the latest SimCity game to drive the simulation.

A small and abstract town simulation was created using a planner to determine what agents in the world should do, backed by a STRIPS-style PDDL domain and problems.

This implementation was successful in creating the abstract town envisaged, but had technical limitations that should be considered if the solution were to be applied again or in other similar areas.

Java-FF was used as a planner which solved problems quickly and adequately, but the implementation of Sam Town was written in C++, so the two could not be connected easily. This caused bottlenecks and slower performance than if the two were more compatible (either a Java implementation or a C++ planner), so this different language solution is highly not recommended. However, on the development machine a number of agents could still operate without any issues.

While the goal was to create an abstract view of a town, the end result of Sam Town may be too abstract. Goal selection at present is essentially completely random, but humans do not really function this way. For this reason, a recommended approach would be to not use a goals system, but to use a needs and wants system instead. For example, rather than a goal be to buy meat from a butcher, it could be to “eat”, which may require a visit to a butcher to satisfy. A reworked STRIPS domain could work adequately for this, but this approach would be more suited to HTN-style planning, which is where future work would lie.

All in all, Sam Town implemented STRIPS planning for a town simulation. The end result was satisfactory, with the potential for a number of agents to successfully wander around a user generated environment, proving STRIPS as an adequate planning solution for a small and abstract town simulation.

REFERENCES