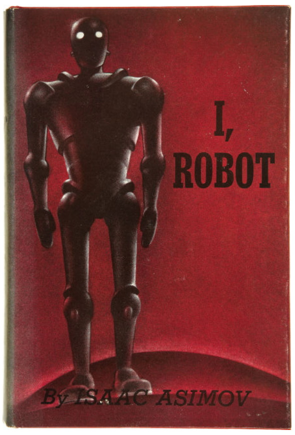
AI Tutorial 3:
AI in Games Development



Summary

This tutorial provides an overview of artificial intelligence in games development, highlighting some realities which are often overlooked in the rush towards human-like behaviour. We discuss AI at a higher level, before introducing some specific concepts which are relevant (both in good and bad senses) to game engineering. We finish with discussion of the ways game AI has influenced research AI, with the GOAP algorithm as a case study.

New Concepts

Decision Trees, Rubber Banding, Adaptive AI, Machine Learning, Goal-Oriented Action Planning

Introduction

This tutorial provides an overview of how artificial intelligence (AI) behaviours are achieved in game development. The first section is a discussion of why games use AI, in terms of what it is designed to achieve; this gives some context for the rest of the workshop. We then explore various approaches and tools which are typically used within games programming. Finally, there is some discussion of how to approach AI coding for games as an engineer.

The processes are discussed at a high level, and with some personal opinion. The workshop is intended as an introduction to the areas of AI most often used in video games which may lead you to find out more, and try some techniques of your own.

What is Artificial Intelligence *for* in Games?

Before embarking on our tour of artificial intelligence techniques, it is worth considering what it is that we want AI to contribute to a game.

Firstly, the AI in a game should contribute to the players sense of immersion in the games setting. In other words, AI behaviours should be believable within the context of the game this is often stated as making the AI realistic, but remember that many games bear little relationship to reality. Whether a character is an enemy, a friend, or is completely indifferent to the player, its behaviour should seem natural within the rules and conventions of the game world.

Secondly, of course, AI should provide the player with a challenge. Whether the AI is an opponent to vanquish, a friend to lead, or a settlement to rule, the AI code needs to provide a level of apparent intelligence which can be influenced by the players actions.

These two elements should be combined to provide a game AI that helps make the gamer feel good. At least one major publisher refers to this as making the player feel like a gaming God. What this means is that an AI opponent should be beatable by most players. From a technical point of view, it is very easy to develop code for an unbeatable AI opponent (e.g. a guy with a gun who spots the player immediately and shoots to kill, or an opponent race car that steers the course at maximum speed at all times). It takes much more time and skill to develop an AI which behaves believably, with elements of its behaviour that can be exploited by the player.

It is worth noting that AI code can be applied to a wide range of different agents in many game genres. This workshop largely uses characters or vehicles as examples, but AI routines can govern any aspect of a game that needs to make intelligent-seeming decisions. Examples include the behaviour of groups of agents (e.g. a swarm or herd, as discussed previously), squad-based tactics, entire communities, some kind of voiceover narration, a mechanic for showing the player where to go next, and so on.

Random versus Predictable

In order to provide a player experience which is both immersive and entertaining, its important to strike a balance between randomness and predictability in the AI behaviours.

In many games it is important for the player to be able to learn patterns of AI behaviour, and utilise a weakness in that pattern (for example observing the patrol route of a guard in a stealth game such as Deus Ex or Metal Gear Solid). However, if the behaviours are too repetitive, then the sense of immersion is lost, and the player feels like a weakness in the software is being exploited, rather than a chink in the armour of a believable character.

When planning an AI system for a game, it is vital to spend as much time as possible with the game designers, discussing what the system needs to provide from a players perspective. Balancing predictable behaviour with unexpected events is a crucial part of these discussions, and needs to be factored into the system design early on.

AI Processes

There are multiple avenues of development and research in artificial intelligence applicable to many fields of computer science and beyond. The application of AI to video games is influenced by one over-arching fact the game must run in real-time on a specific platform which must also provide the processing power and memory required by the graphics, physics, audio, networking, etc. Consequently the AI routines most applied to video games are the ones that can be ran with the least overhead.

Furthermore AAA titles will potentially be played by millions of players any algorithms used need to be reliable 100% of the time. If something works most of the time or 999 times out of 1000, then there are still potentially thousands of people who will see the edge-cases and have their game ruined.

The two most-used areas of AI are Finite State Machines and Path-finding, which have been covered in detail in previous tutorials. In this section, a range of AI techniques are discussed within the context of how they are commonly applied to video-games.

Decision Trees

The most common use of Artificial Intelligence in games is in making decisions, and presenting those decisions so that they appear to be an intelligent choice. If a decision can be broken down into a sequence of questions where the only valid answers are yes and no, then a Decision Tree can be used to model the process.

An implementation of a decision tree, with which you are likely to be acquainted, is an automated telephone enquiry line. You have almost definitely rang a business and been confronted with a set of options by a computerised voice; something along the lines of "press 1 for a current balance, press 2 for a change of address, press 3 for a long wait...". It is easy to imagine a branching structure of questions moving from the first one, each answer taking you to the next question. The questions are nodes in the tree.

Decision-making code can be constructed in the same way, with a series of binary tests leading to a selection of different outcomes. This is a pretty simplistic approach, and is limited in its AI-related applications, but for simple character decision-making it may suffice. A Finite State Machine is a much more powerful and flexible approach.

Embedded Information

We have already discussed the idea of embedded data specifically in the context of map navigation; here, we discuss the concept more broadly in the terms of other areas in which it is applied. It is a very common technique to embed additional information, which is of use to the AI agents, into the world data.

- Extra information can be added to world objects, which helps AI agents determine how they interact with them.
- Invisible objects can be included in the level, which trigger some response from a nearby AI agent.
- Entire extra layers of information can be stored to aid the AIs navigation of each level (such as 'cover', discussed in the path-finding tutorial).

A simple example of embedded data is a box covering a doorway in a section of a level. The box is not rendered, and there is no effect if the player moves into it; however, if an AI character moves inside it, then some change to the AIs behaviour is triggered. The collision detection routines are used to detect when an AI agent moves inside the box; there is no physical collision response, but instead a state change can be triggered in the AI (it may be that the AI is required to draw its weapon on entering the room). Such trigger boxes are likely to be displayed in the level-editing tool as an overlaid layer that can be toggled on and off. They are exported as part of the world data as objects with specific flags set, so that the game engine knows to treat them as trigger boxes, and not draw them.

Embedding the information in this manner greatly reduces the complexity of the AI code, and improves the flexibility of updating behaviours. The approach becomes particularly beneficial when there are a large variety of objects, or level elements, that the Ai will interact with in different ways. The logic for the interaction is associated with the types of object, giving a more modular approach than having to include all the different interactions in the AI agent code.

Interaction Scripts

Some games have many autonomous AI agents and world objects many God simulations, or realtime strategy games may have dozens of different types of entity, and many instances of each type. Different behaviours may well be required of each AI type when coming into contact with each of the other AI types or other entities. This could lead to each of the AI types having a very complex state machine governing a large number of fairly simple behaviours.

In order to simplify the state machine, the behaviours can be abstracted into a set of interaction scripts. Each combination of AI type with another type, or with a type of world object, has a corresponding script which is activated when the two types meet (for example, if a rabbit meets a carrot, the rabbit eats it; if a rabbit meets a fox, it runs away and the fox gives chase). The scripts are typically held in a series of interpreted data files. The state machine becomes considerably less complex, as all of the interaction behaviours are encapsulated by a following script node. The actual behaviours are contained within the specific scripts.

If this approach is taken, it requires some very careful design work up front, in collaboration with the game designers. The rules for the kinds of behaviour that can be instigated by a script need to be laid out, and a system built to provide that flexibility, with access to the relevant data of the AI agents. The behaviours which are actually coded are very generic (eg move towards point A, run away from AI Agent B), so the actual behaviours instigated by the scripts need to put together these building blocks to make a believable behaviour.

Spline Following

A common alternative to computing a path through an environment using an algorithm such as A*, is to construct splines for the AI agents to follow, and to store the spline information as part of the level data. This is especially common in track-based racing games.

A spline is a curve which smoothly connects a number of points in 3D space. The parameters used to generate the spline determine the smoothness of the resulting curve, and how close the line gets to the actual points used to generate it. It is easy to imagine a race course with a series of points around the track, which are connected by a spline. Such a spline then gives a path for an AI-controlled vehicle to follow. The spline is a continuous smooth line, so following it looks considerably better than simply going from one checkpoint to the next in a series of straight line sections.

Spline following is the basis of the AI for most track-based racing games. Typically a number of different splines are generated (ie with different parameters, or varying positions of the points), so that there are a number of subtly different paths to follow around the track. If a vehicle is trying to overtake, or has got into a tussle with the player, then shorter splines are calculated to achieve the short term objective, before returning to a main track spline. Short cuts and alternative routes are easily integrated by the addition of extra splines, or branches from the main splines.

Rubber-Banding

AI controlled vehicles in racing games cheat. The reason they cheat is to improve the players enjoyment of the game. This may seem counter-intuitive, but it makes some sense when you factor in the purpose of AI in games.

Consider a car racing game which needs to have an arcade feel to it, rather than a terribly serious simulation. Depending on the game design, a lot of the fun of playing the game will come from vying with the other racers if the player barely sees any of the other vehicles during the race it could be a rather dull experience. There is much more fun to be had from regularly trying to overtake, or trying to avoid being overtaken, and a last ditch blast at the finish line can make the player feel far more exhilarated than crossing the line with no competitors in sight.

Most racing games include some AI code for keeping one or more adversary more-or-less within range of the player. This is known as rubber-banding. It can be fairly obvious (e.g. Mario Kart), or it can be a lot more subtle (e.g. Burnout Paradise). The way in which rubber-banding is achieved varies from game to game, but the general aim is to monitor how far away the player is from AI contestants, and to speed up, or slow down the AI to try and bring them closer together.

This is pretty straightforward on a spline-based track game, as its just a case of following the pre-set route more or less quickly. The rubber-banding information can also be used to trigger specific events or behaviours for example a car ahead of the player may cause a pile-up that the player has to avoid, or it may increase the likelihood of a more powerful boost power-up appearing for the player to take advantage of.

Scripted Events

Many games will include in-game cut scenes (IGCS). These are sections of the game, usually used for story-telling, where the player has no input, and the game engine is used to move characters, vehicles etc., around according to a script. Developing the technology to provide these IGCSs is a significant amount of work, which involves tool-chain development as well as engine code.

If the technology has been developed for presenting an IGCS, it can also be adapted to provide scripted events while the player is still in control (such an approach is taken by Blizzard in recent Starcraft games, for example). Depending on the game design, taking this scripted approach may well be a better solution than trying to use AI systems.

An example may be a game such as Stuntman, where the player has to drive a stunt car safely through a vehicle pile-up while the crash is actually happening. Expecting an AI system to meticulously drive a number of vehicles into each other, so that there is always a path through for the player, often with only inches to spare, is a big ask, and is unlikely to be 100% reliable. A far better solution is to place the vehicles on predetermined paths, which are timed to coincide with the players progress. Repeatable set-pieces can be much more cinematic using this kind of system, if that is what the game design requires.

Adaptive AI

As AAA games become more and more expensive to develop, there is a push from publishers to make them more accessible to anyone. One of the ways of achieving this directly affects the AI systems. In order for as many customers as possible to feel that they have had their moneys worth from a game, the difficulty level can be dynamically adapted by the game itself.

There are two ramifications for AI systems here, and the complexity of the solution to both will vary depending on the intentions.

Firstly, the game needs to detect when a player is struggling. This could be something as simple as counting how many times a boss fight has been failed, or it could involve logging specific behaviours and comparing them to test data.

Secondly, the game needs to dynamically change the difficulty of the game. In the simple case of the repeatedly failed boss fight, this might just involve toning down the power of the boss attacks. More often than not, the difficulty changes will require the AI behaviours to be modified in some way. This may be as simple as decreasing their health, changing trigger values in the state machine, or even entirely replacing the state machine. If the game already features some pre-set difficulty levels, then the AI is likely to have been developed with various settings available.

Machine Learning, Genetic Algorithms, Neural Networks...

There is a great deal of artificial intelligence research and development in areas which involve the AI routines adapting their behaviour over time, learning as they go. This type of approach is outside the scope of this document, although it should be pointed out that applications in the game industry are rare (Alien: Isolation being one example).

If such an approach is undertaken, then it is unlikely to involve a real-time learning process while the game is being played by a customer (i.e., a player who has bought the game). In some cases one of these techniques may have been used to "train" an AI to perform particular tasks for example, steering a racing car around a track. The data generated from that learning process, is then encapsulated in the game data, and is used while the game is played, but is not evolved further.

The reason these learning techniques tend not to be used in commercial games comes from one of our starting points for this discussion: commercial games need to present a high quality experience for potentially millions of players. Artificial learning techniques, by design, make mistakes and learn from those mistakes a player experience which starts with AI characters blundering around like fools isnt really applicable to many game genres.

Additionally, there are a couple of disadvantages in implementing a training process for the AI during the game development, and then using the trained AI in the published game:

- The rest of the game often needs to be complete before the AI can be trained to play in it. Any changes to level structure, or to vehicle handling models, weapon parameters, character abilities will require retraining the AI. This is potentially an expensive and risky process, as most games are still undergoing tweaks right up to the submission deadline (often beyond with patches, downloadable content, etc.)
- The parameters arrived at by the training are often not human-readable (e.g., in the case of artificial neural networks). They cant be meaningfully tweaked at the last minute; the only way to change them is to run the learning algorithms again.
- The trained AI requires extensive QA time to see how it reacts to anything the player may do. Any failures or bugs here will result in a change to the system, and retraining the AI there are unlikely to be direct bug fixes for this type of issue.

While this all sounds very negative, it doesnt necessarily rule out the use of learning techniques in game AI development. However, if you are planning on going down this route, you need to be very certain that it is appropriate to the requirements of the game design, and that the implications (both technical and production-based) are understood.

Engineering an AI System

The key to providing a good system for providing the artificial intelligence for a game (or indeed any system) is to understand what the requirements of the system are. Once the general requirements are understood, a solution can be chosen and the system designed and built. Remember that the system may well involve a number of different AI solutions, contributing to various parts of the game.

Also bear in mind that the specifics of the game design will evolve during the development cycle of the game. This is a good thing! As a prototype gradually turns into a fully-rounded game, extra ideas come up and further insights into the gameplay arise, which will probably require some changes to the AI behaviours. A well designed AI system should be able to handle such changes with minimal fuss. For example, during the discussion of Finite State Machines, the concept of character types combining behaviours was introduced a well-engineered system will allow the addition of a new character type using existing behaviours in a new combination.

It is also important not to over-engineer a system. If the game requirements for interactive characters are largely about set-pieces, or face-to-face conversations, there may be no need to implement a full path-finding system. With this in mind, if a small part of the game design requires a massive investment of technology, talk to the designer and producer, explaining the cost.

Level of Detail

For complex games, especially open-world games, it is likely that you will need two or more levels of complexity for the AI behaviour of agents. Those closer to the camera, or more likely to interact with the player, require a higher complexity of behaviour, while those further away may be able to use much less complex behaviour. We have seen a similar thing to this on the graphics course, with level-of-detail polygonal models, and also with mip-mapping.

Consider the vehicles in an open-city game. It may be possible to see several vehicles, maybe even dozens of vehicles, on a street going into the distance. The vehicles which are closest to the viewpoint need to behave in a believable way, including some interaction with the player. Those further down the street dont need such complexity, and it may well be sufficient to keep them moving along their lanes, stopping for traffic lights, but not needing any complex decision-making or path-finding. In an extreme case, vehicles in the far distance may just be modelled as an animated texture of blobs moving along a distant road.

As with mip-mapping, there can be issues when changing the AI LOD, resulting in a popping of behaviour. This can be especially apparent if the AI is changing back and forth between LODs every frame or so. There are many methods to disguise this, with the most common solution being to take care in defining the triggers for changing LOD. You may well have played a game where you knew an enemy AI would never react to you beyond a certain distance in that case the AI LOD is purely based on the distance to the player. A simple improvement to this would be to wake up any enemy that the player fires at, or which has the player in its line of sight for any length of time.

Spread the Load

In most game situations, the AI code is not time critical, in the way that the physics and graphics updates are. It is unlikely that an AI behaviour needs to be updated every single frame of a game if the update takes several frames the player is unlikely to notice, as it is unlikely to affect the gameplay. This means that the processor load can be spread over a number of frames.

If there are many AI agents to update, a few of them can be updated each frame. If this approach is combined with the level of detail concept discussed above, then more frames of AI calculation can be devoted to those agents that are most apparent to the player, while the more distant agents are updated less often. Of course, this does not mean that the AI characters remain static during the frames when they receive no AI attention; it just means that they continue with the behaviour that has already been assigned to them, so they continue to update the animation frame, to move toward a goal, and so on.

Complex calculations, such as path-finding can also be distributed across a number of frames. In this case, if a new path is requested, it may not become available for a few frames of action, at which point the agent starts to use it. Again this should not be noticeable to the player, although in some cases it may be worth halting the character and instigating a ”thinking” or ”scanning” animation, which could actually make the change of behaviour seem more believable.

Empower the Designers

As a game developer, you need to accept that aspects of the game design will change as the game development cycle progresses. In fact, you should not only accept this, but embrace it! It is unlikely that any great game that you have played started with a design that did not evolve over the time of development.

A well-engineered system will include a toolset that the game designers can use to make changes, and to experiment with new ideas. A development tool could be as minimal as a spread-sheet which exports a data-file that the game reads on start-up, through to a fully editable application with a graphical interface, or even an in-game debug mode which provides real-time ability to change AI parameters.

There are a couple of reasons why empowering the designers in this way is a good thing. Firstly, it allows them to experiment and change elements of the game with a fast turnaround, so they can quickly play and test the result. Secondly it frees up your time as a programmer for bigger systems-based tasks, instead of spending days tweaking hard-coded numbers while a designer stands over your shoulder saying things like ”make it more wobbly”.

When developing a tool, always bear in mind the target user (i.e. who will actually be using it). Firstly, it is likely to be used by a handful of people in the same building as you it doesnt need to be an all-singing all-dancing application that the whole world has access to. Secondly, designers are not programmers, they probably arent very technically-minded at all so any options need to be couched in terms that they understand. A bad example of this was a tool which allowed the initial facing direction of a character to be set the options in a pull-down menu were "Zero", "Pi over 2", "Pi", and "3 Pi over 2" unsurprisingly the level designers just looked at this in confusion and left every character facing along the Z axis.

The World is Artificial

An enormous advantage that you have as an AI coder on a game project is that you already know everything about the world that the AI interacts with. Most AI developers in other areas are trying to interpret elements of the real world, but game worlds are completely artificial. Not only are they artificial, they are entirely created and controlled by the development team. Use this fact to your advantage.

There is no need for a piece of AI code to scan the environment in order to interpret it for its needs. Everything that the AI code could want to know about the environment is either already known or can be easily queried; I.e. the data can be accessed from somewhere in the game structure. Of course, you may want to have the AI character appear to scan the environment to increase immersion in the game, but that can be achieved by playing a suitable animation, with minimal load on the AI code itself.

Not only do you have access to all the data that goes into creating the world, you also know exactly what the player is doing, where the player is, and what the player has done to get to that point. Again, this knowledge should be used to your advantage as an AI coder. Obviously this data should not be used in a way that makes the player feel cheated; it should be used to provide a better experience for the player. For example, its far more fun if zombies are lurking around the corner that the player is about to peep around, than if they are on the other side of town so the player never sees them.

Goal-Oriented Action Planning

Goal-oriented action planning, or GOAP, is a means of employing graph search algorithms to generate strategies for game agents. It does this by abstracting the notion of a graph away from navigation, and into the arena of ordered actions. In this sense, it can be considered a combination of cost-based path planning and the dynamic construction of a finite state machine.

Let us consider our agent's state not as an integer but, instead, as a series of binary values (e.g., `[0|1|0|0|0]`). Let us consider the actions our entity can take as similar binaries, with values of either `x` or `y` (e.g., `[x|y|y|y|x]`). If our agent engages in a given action, elements mapped to an `x` are flipped, and elements mapped to a `y` are not.

If these actions can be mapped onto a graph - interconnected, with prerequisite criteria - then it stands to reason that a path between actions (nodes) exists which can lead our agent to a desired state (such as `[0|0|0|0|0]`). A graph search, guided by a priority queue or appropriate heuristic, can determine this 'plan of action' for our agent, leading to potentially sophisticated strategy using relatively inexpensive algorithms.

This technique is a variation on goal-oriented action planning, which was an A.I. approach developed for the game F.E.A.R.. Many publications have been written on the topic since, and GOAP is a good example of industrial developments feeding back into academic research. The strength of the approach is in its ability to generate flexible strategies at run-time. One downside is that the design of the graph of actions can be an arduous, design-heavy task for sophisticated games.

The search employed varies depending upon the nature of the scenario. If there is a readily mappable heuristic which can be applied to the graph of actions, then A* can be appropriate. If not, a priority-queue managed Dijkstra implementation might be more suitable.

Summary

Developing code in the video game industry is an engineering job. A good engineer uses a variety of tools to resolve a technical challenge. This workshop has described some of the tools available for developing an artificial intelligence system suitable for a video game. A well-engineered system will utilise a range of these approaches, along with those addressed in previous tutorials, as appropriate to the design of the game