.slide 1

Hello everyone, welcome to the last Game AI Design Crash Course, the last one we'll have this semester. I have another selection of – hopefully interesting – game AI problems prepared for you today. Before we dive in, let me remind you that the deadline for putting yourselves into teams for the group assignment is today, those who won't will be grouped together by me. And for next week, you're supposed to pick a game to work on. Remember that the point of the assignment is to design an AI yourselves, *not* find out how it was designed, so pick with that in mind.

Alright, with that out of the way, let's jump into today's selection, starting with-

.slide 2

Horizon Zero Dawn. This is another open-world post-apocalyptic RPG. In this one, you inhabit a world populated by machines whose designs are based on animals and you have to fight through them in order to achieve your goals, uncover who you are and how the world came to be the way it is. There are a bunch of different machines in this game, but we are specifically going to take a look at two of them. The Glinthawk and the Stormbird, because they are the only ones that can fly.

.slide 3

And flying, of course, entails navigating through 3D space. So, the question is, how to do this navigation in 3D space, how to make their movements seem natural while doing it and how to make it all efficient since this game runs in real time. So, let's break the problem down a little.

Questions:

How to represent 3D space?

How to do pathfinding within that representation?

How to smooth out the paths?

How to handle collisions?

Well, if there are no more ideas, let's take a look at what the designers did.

.slide 4

Doing full 3D navigation is theoretically possible, but can be quite expensive unless your space representation is very sparse. Instead, the designers decided to partition the 3D space into cubes and use height maps to answer queries about whether a creature can pass through some point in space. Since height maps only contain one height, that means no flying under arches, rock outcrops and the like, but that wasn't too much of an issue since that isn't a very common feature of the land in Horizon Zero Dawn.

These height maps are generated at runtime from the world geometry – I won't get into the details of how that's done – and that is done in a sphere around the flying agent.

.slide 5

The actual pathfinding is done using 2D A* - not 3D. So it's only done in a plane and if the algorithm finds that the creature can't get somewhere, it is allowed to fly up, but doing so has a higher cost – positions that are at the same height or lower use Euclidian XY distance as cost, while positions that are higher use Euclidian XYZ distance. Therefore flying up is penalized but flying down isn't.

Just using A* led to some performance issues though, since there usually aren't very many obstacles in the air and if you have vast open spaces then that means you're going to have a lot of equally good paths to choose from.

The author's solution for this was something they called-

.slide 6

Hierarchical path planning over mipmaps. You may know what mipmapping is from some graphic's-oriented subject, but if you don't, it means that you take a texture and create versions of it with different resolutions. And, when rendering an object, you pick an appropriate resolution to use based on how far away the object is.

What it means here is that the designers created multiple height maps with different resolutions. This obviously led to some memory overhead, around 33%, but that was acceptable.

With this, the pathfinding can be done hierarchically – first, a path is found at the lowest level of resolution and then progressively refined using higher resolution height maps. There is a fixed number of A* iterations allowed for each query, so once that runs out, the best path found thus far is returned.

Once the path is found, it is further smoothed out using raycasting. So, that means that you first shoot a ray from the start of the path to the end. If it doesn't hit anything then the creature can just fly in a straight line. If it does, then there is some obstacle, so you split the path in two and repeat the process for both segments. You do this until you have a sequence of line segments that don't intersect with anything.

.slide 7

Here you can see a part of the terrain.

.slide 8

And here you have the height maps at different levels of resolution.

.slide 9-12

.slide 13

And here you have an example of what the found paths look like at the different levels. The green path is the one found by A* at that level, the pink squares are the regions of space to which pathfinding is restricted at higher levels of resolution and the pink path is the final path after the raycasting step.

As you can see, the final path is still pretty jagged – it would be weird if the creature followed it precisely because it would very abruptly change its direction and speed.

.slide 14

So, in order for the movements to look natural, they are actually implemented using some simple physics. Each flying creature has a velocity, angular velocity, acceleration and angular acceleration. And just by manipulating the acceleration vector, you can make the creature follow the found path while also keeping its movements smooth and natural.

Now you might be thinking, 'how does any of this solve collision avoidance?'. And the answer is, it doesn't.

.slide 15

Because the player isn't likely to notice two flying creatures pass through one another in the air, especially in the heat of battle.

As for the landing, the air navigation system communicates with the ground-based system to first generate some candidate positions. The sources that I used didn't go into detail about how that's done, all I know is that the system tries to find a position that is a little bit above the average height of the surrounding area. But, in principle, I believe this can be anywhere.

When crashing, on the other hand, the candidate positions are restricted by the creature's current direction.

There's a nice little detail I want to mention here. The Stormbird has this attack where it dives straight at the player, which also uses the same mechanism for communication between air and ground navigation systems, except there is only one candidate position – the one where the player was standing a moment ago. But what's interesting is that the Stormbird actually tries to block out the sun or the moon while doing that. It happened by accident first when the designers were testing the game, but they decided to try and make the creature do it deliberately. So that's just a cool detail, I think.

Now, if you're familiar with this franchise, you know that Horizon: Zero Dawn has a sequel –

.slide 16

Horizon: Forbidden West. And I want to briefly mention this game as well, because its 3D navigation system is a bit different from the one I described just now. Namely-

.slide 17

Instead of using heightmaps for collision computations, it uses octrees. These are data structures where, in order to represent a space, you start with a single cube, you take a look at whether it contains any obstacles, subdivide it if it does and do this recursively up to some given level of detail. The advantage of this is the fact that it also allows for flying under things like bridges or overhangs.

The main reason for the change was probably the fact that you can also fly in this game, so that probably necessitates more detailed flying behaviour from other creatures as well. But, given that this seems to me a much more natural way of representing 3D space for the purposes of navigation, it makes me wonder why they didn't do this in the first instalment. I don't see it being significantly more memory-intensive, so my best guess is that running A* in this representation is still much more demanding than doing so in what is basically 2D and perhaps they managed to create some more optimized version of the algorithm here. But, as I said, that's just a guess, so don't quote me on that.

Alright, moving on, today's game number two is-

.slide 18

A MOBA game called Paragon.

.slide 19

The idea behind this game was to combine a third-person shooter with a MOBA. So, you have a standard MOBA map – three lanes, bases at the ends, towers along the way, minions coming in waves, some special areas for gathering experience, the whole schtick. But, instead of controlling your avatar from on high, like in classic MOBAs such as League of Legends and Defence of the Ancients, you control it like you would in a third-person shooter.

And, of course, each player gets to pick a hero with unique abilities and each hero is good for something.

Now, the problem here is a bit different than others we've encountered thus far and a bit harder to formulate. Most problems I've shown you up until now were quite specific and we tried to arrive at a pretty low-level solution. Here, we're interested in the big picture – how do you design an AI that can play a game like this, but in a team of bots and in a team combined with human players. So, let me try to specify this a bit more.

.slide 20

Imagine you're making a game like this and you've spent some time observing how players play other MOBA games. Based on that, you've identified certain objectives that they may try to pursue. These are things like 'attack an enemy tower' or 'protect your base' or 'get XP from an experience well'. These objectives are competing with one another and which objective an agent might pursue and when depends on the current context – for example, if you base is under attack, it is a good idea to head over there to protect it. But if an ally is already taking care of it, you may want to prioritise something else. So, how would you utilise this knowledge to create an AI capable of playing this game? Does anyone have any thoughts?

…

Well, if there are no more ideas, let's take a look at how the designers tackled this problem.

.slide 21

First of all, just to allow the AI to move around in a reasonable way, it's probably a good idea to create a graph where the nodes are all the important places on the map – bases, towers, experience wells, but also designer-defined places like good spots for ambushes – and the edges represent paths between them. The edges can be generated automatically and then possibly be tidied up a little by hand.

This whole graph can be defined statically for any map – although I suppose there's only one for this game.

.slide 22

With this, the pathfinding itself can be done hierarchically. So, you first find the path in the graph, then you take each edge and try to find a real-world path corresponding to it.

We can also update the graph's edges with "influence" values from the enemy. Influence here means the level of control that the enemy has over an area. For example, if all the enemy heroes are in one spot, the enemy will have a very high influence over that one area and that means that if you go there, you will most likely be killed.

Using this information, we can do some more goal-oriented pathfinding. For example, we can find a path that avoids the enemy. Or, conversely, we may want to seek out the enemy. Or we can want to stick to one lane, etc.

Now, the fact that we can do some reasonable pathfinding is nice, but it doesn't give us a way of determining what the bots should do. So, the key idea here is-

.slide 23

Adding objectives to the graph nodes. These are things like "attack here" or "defend this structure" and the like. They tell an agent where to go and what they're supposed to do there. And all the information about how many agents are needed to satisfy an objective and what abilities they should have is contained with the objective itself.

.slide 24

These objectives are generated at the start of a match, though many will be dormant – for example, it would make no sense to defend a structure that is at full heath and isn't being attacked.

They then get assigned priorities at runtime, which dictate in what order they should be assigned to heroes. These priorities are based on things like the type of the objective, the location of the associated graph node, the enemy influence in the area, etc.

.slide 25

The system in charge of assigning these objectives to agents is called the AI Commander. This happens in the following way. First, each objective pics a state – dormant or active. Then, priorities are computed for all the active objectives. After that, each objective goes through all the members of the team, filters them and assigns them scores that reflect how desirable the agent is from the objective's point of view – how well they can help satisfy it. Then comes the actual assignment. The objectives are picked in order of their priority and each can then pick the minimum number of agents it needs to be satisfied. Its priority is then decreased based on which agents were picked and the next objective is selected. This either repeats until each agent gets assigned an objective or until all the objectives have been assigned, I'm not sure.

Once an agent has been sent somewhere to do something, it can also query all the graph nodes it passes for objectives it can fulfil there and the objectives then internally decide whether they want to be assigned this way.

Now, you hopefully have an idea of how this works when only AIs are involved. But what about when the team partially consists of humans?

.slide 26

Well, the assumption here is that the AI Commander is equipped with reasonable objectives and can assign them reasonably. Therefore, the human players should go around fulfilling these objectives without being explicitly told to, just by knowing how to play the game. But there is also a textual interface through which the players can be notified that something's going on and prompted to react – for example "go defend a tower that is under attack" or something.

Here is a quote from the authors about the effect this system had on their game.

.slide 27

"As proven by our internal user experience tests, the introduction of a strategy layer to the Paragon bot AI greatly improved players' experiences. The game did not instantly become harder, because no behavioral changes have been made to the bots' individual behaviors, but users did notice the game being more interesting."

.slide 28

"Bots started showing signs of a deeper strategic understanding of the game: filling in for fallen comrades, switching lanes, attacking enemy harvesters, and even [ambushing], although the latter behavior was entirely emergent. Players will generate their own explanations for what is going on in the game as long as the AI is doing its job well enough!"

And that's it for Paragon. The reason I decided to include this game here is because I find this to be quite an interesting algorithm that is quite different from the ones you've encountered in the course thus far and might be useful in other games besides MOBAs as well.

Ok, moving on, the third game I have prepared for you today is-

.slide 29

Watch Dogs 2. Another open-world RPG adventure. In this one, you play as a hacker trying to take down a system for spying on citizens. An important part of this game is the immersion of the open world which is populated by procedurally generated NPCs, particularly because you get to hack into their devices and thus peek into their lives, so they have to come off as real and fleshed out.

.slide 30

So, the problem facing us is as follows: We have an open world filled with procedurally generated NPCs who are supposed to have interesting emergent interactions both by themselves and in response to what the player does. How to ensure that these interactions are spontaneous, varied and semi-realistic? To give you an idea of what we're going for, let's start with a simple scenario. Imagine there's a person playing guitar in the park. How might passers-by react? Now what if someone makes fun of that person?

Questions:

Explore the scenario further.

How to make sure these sorts of things arise naturally in the world?

If we start discussing hard-coded rules – what would be their inputs and outputs?

How would you organize the rules in such a way as to keep them ordered and allow the designers to adjust them?

How is a character's personality defined and how does it affect their reactions?

How is a character's mood defined and how does it affect their reactions?

If you have nothing more to add, let's take a look at the original solution. So, we want to create interesting, emergent behaviour. Let me start with a quote from the authors about what an emergent behaviour is.

.slide 31

It is a behaviour that "is not predicted, but is predictable" – a definition I find quite eloquent. This means that it can't just be randomness – one thing has to lead to another in a way that is understandable for the player. So, how did the authors achieve this?

.slide 32

There are two main components to their solution – attractors and a reaction system. The attractors provide stimuli that the NPCs then react to. These can be static – that is, placed in the level at specific spots by designers (e.g. someone playing a guitar) – or dynamic – these are scheduled at runtime through an event system (e.g. two NPCs greeting one another).

.slide 33

These stimuli have zones defined around them within which NPCs can react to them – so a similar idea to what we discussed with Hitman last time. They may choose not to do so, of course, but if they do, then their reaction becomes another stimulus for other NPCs to react to. In the example here, let's say the first NPC is playing the guitar in a park. That creates this reaction zone and the NPC that is inside it reacts by making fun of the player. This creates this larger reaction zone and some other NPCs may choose to react to this, let's say by voicing their disapproval or by pulling out their phone and recording what is going on, creating more reaction zones.

.slide 34

An NPC's choice of action depends primarily on their personality, which is defined by five traits – violent, pessimistic, optimistic, neutral and heroic. Each of these has a corresponding metre associated with it and the values of those metres have to add up to one hundred. So, this tells you something like 'in what percentage of cases is the character violent, pessimistic, etc.'.

Each trait is used to pick an action separately and a probability distribution is then constructed based on the values in the traits' metres, after which an action is picked at random from that distribution.

For some situations, it would be weird if the NPC just transitioned right back to what they were doing before once they finish reacting to some stimulus. For example, if they are laughing with a friend and you make them really angry, they shouldn't just go back to being upbeat and chatting like nothing happened.

.slide 35

For this reason, the NPCs also have a mood attribute, which is governed by the simple finite state machine that you can see here.

.slide 36

Ok, what we have now seems like a pretty good system, it just may require a lot of work defining all the possible reactions and especially defining the rules that govern when a given reaction should be triggered. By the way, the way that the designers came up with these was that they had sessions where they role-played different scenarios and how different characters might react to them and then translated those into the game.

Reportedly there are around 750 logic rules, 200 animation rules and 220 audio rules in the game, which all jointly determine which reaction a character should have and which animations and sounds should accompany it. So, how do you organise this in such a way that it isn't a mess for the programmers and, importantly, allows the designers to define new rules with ease? Well, the answer is-

.slide 37

An Excel spreadsheet. This definitely surprised me when I first heard of it, but, apparently, spreadsheets are a pretty useful thing when it comes to AAA game development – they allow designers to define various aspects of the game and can then be easily injected into the game. In fact, the pipeline for this game was designed in such a way that they could update the tables at runtime. Here is one of the authors showing a little of how it was used.

For your next assignment, you'll be programming Visual Basic macros for Excel computations.

Ok, the system we've described here works well for simulating characters that make an open world feel alive. Now, how to integrate them into level design?

.slide 38

.slide 39

The authors created three types of NPCs. First, you have your run-of-the-mill bystanders and passers-by. When the action starts these guys flee. Then, you have people who belong to a specific place, such as construction workers at a construction site. Their behaviour differs slightly in that they don't necessarily flee when trouble starts and they can also notice that you don't belong and alert guards to your presence. And then there are the combat-capable NPCs – their purpose is to shoot at you.

In the talk that this segment is mostly based on, the speaker said that when they created some working version of the system I just described, the level designers actually didn't want to use the NPCs in their missions because they couldn't be made to fit them, which led to the creation of this system, or specifically the NPC who belongs to a given place and behaves as such. I found this very strange – how could you design an elaborate system like this without having properly communicated how it would fit into the gameplay? I'll return to this thought a bit later on.

For now, I'd like to make a brief detour and take a look at this game's sequel:

.slide 40

Watch Dogs: Legion. This game is quite similar genre-wise and story-wise to its predecessor, but it features a very ambitious 'play as anyone' game mechanic.

.slide 41

In case it wasn't clear from the name, this game, like its predecessor, contains a large number of procedurally generated NPCs and you can actually recruit and play for almost any one of them. Let's take a look at how that's possible.

.slide 42

The key is that the NPCs are generated at different levels of resolution. The random people you pass in the street just have their appearance and a few bits of info generated, such as their name and occupation. Then, if you engage with them, the system generates their life in more detail. One aspect of this is generating further attributes of their personality, such as their hobbies. I'm not sure how many levels of resolution there are, but at the highest, a character's whole daily schedule is generated, which they actually follow.

An interesting aspect of this is that the system is designed in such a way as to take the most important current piece of information (this depends on how you came to interact with the character – if you meet them at a law firm, it's their job, if you meet them while they are out running, it's this hobby) and generates other pieces of info based on that.

.slide 43

What's more, the system tries to ensure that you run into NPCs you've interacted with already multiple times. So, a guy you got into a fight with could be a bartender at a hotel that you have to visit on your next mission.

And the system also generates whole recruitment missions for NPCs you want to recruit.

Now, I don't know about you, but I found this very impressive when hearing about it for the first time – it truly seems like a unique level of immersion in a game, right? Well-

.slide 44

.slide 45

What went wrong? There are a couple of possible reasons, but two things that I read when looking at some of the reviews and reactions were that A) the characters didn't feel all that interesting compared to the previous title where you played for an actual, specific, fully designed protagonist, and B) that it actually didn't matter that much who you recruited because the missions weren't that dependent on the character's various abilities. Instead, you could solve everything by hacking and driving – something that everyone you recruit is capable of, interestingly enough. So any random person you pick on the street can become a pro hacker.

Now, it isn't the case that there are absolutely no differences between the different NPCs, for example, you can recruit a police officer and gain unrestricted access to areas which you would otherwise have to break into. But it seems that these differences are more just little tidbits sprinkled into the game, instead of a core feature of the gameplay. Which, again, strikes me as rather bizarre – why would you design this elaborate system and not make sure that it is properly integrated into the gameplay? Hard to say.

Ok, that's enough about Watch Dogs, let's move on to the final game that I have prepared for you today, which is-

.slide 46

Battle of the Bulge.

This is a strategy mobile game based on a real historic event from the second World War, when the Germans ambushed the Allies in a last-ditch effort to break apart their armies and thereby isolate them and get enough of an upper hand to force the Allies to sign a peace treaty. There are two sides in the game – the Allies and the Axis – and you can play for either one. The rules are kind of complex, but we don't really need to get into that here. All that you need to know is:

.slide 47

This is a turned-based game with a large branching factor. Each player has some units at their disposal which they can move around. When opposing units occupy the same region, they fight. Now, the interesting thing here is this – when choosing your opponent, you can always choose from among three different generals. These generals are supposed to have distinct fighting styles. So, the question is, how do you achieve that?

First of all, let's consider the general AI architecture. I left out a number of details, but what you need to know is that this game is mainly about moving your units around the areas of the map, a part of which you can see here, having them fight – this part is probabilistic and depends on a variety of units' attributes – and your goal is either to annihilate the enemy or to reach some specific destination and take control of it. What sort of algorithm or architecture would you use for the AI in a game like this? Does anyone have any ideas?

Questions:

What is a play-style? How do different ones look in a game like this?

How to make sure the proposed algorithms create different play-styles?

Alright, looks like there are no more thoughts on the matter, let's take a look at how the authors solved the problem.

.slide 48

The authors' AI has three components, which are inspired by different parts of the human brain, hence their names. The first layer is the Reptile Brain.

.slide 49

This part applies various weighted utility functions to moves to determine how good they are. This will be important and we'll get back to it in a moment.

.slide 50

The Neocortex uses the move ordering provided by the Reptile Brain – as well as other optimizations – inside an implementation of Alpha-Beta Search. This is the part that actually picks moves.

.slide 51

And the third part, called the Limbic Brain, serves to learn from the player. It does this by storing what moves the player makes and when, though I'm not sure what sort of data structure is used for this – the authors just mention they use heat maps which somehow incorporate time, which I have trouble wrapping my head around, but it's not that important. What is important is that this part of the AI learns from its opponent and uses the knowledge it gains when modelling them during Alpha-Beta search.

So, that's the general overview. Now, I want to get back to the Reptile Brain, because that's wherein lies the solution to the problem of personalities that I posed at the start. And I would like to mention a quote from the authors here, which I find quite insightful.

.slide 52

"Many developers try to create the perfect player first, then down-tune to create personality. This prioritization discounts the great wealth of expression players have on each of their chosen battlefields. Within every game, there are very different ways to play."

So, how did the authors go about taking this into account?

.slide 51, 50

.slide 49

Well, as I mentioned, the Reptile Brain is equipped with various utility functions for moves – the authors don't go into detail about this, but I assume these can be things like 'this move brings us closer to a desirable position' or 'this move destroys some enemy units'. Generally simple stuff that can be directly constructed from the game rules. These utilities are computed and then combined using weights to give a final move ordering. And herein lies the magic –

.slide 50-52

.slide 53

By manipulating the weights, one can create different play styles. And this can be specifically done in such a way as to create some archetypical behaviour, such as 'cowardly', 'defensive' or 'rash'. Simple, but elegant, right?

And an interesting aspect of this whole design is that each of the three ‘brains’ can actually be built and tested independently and then put together, which, I suppose, must make it quite pleasant to work with.

The authors used this system in two more similar games besides Battle of the Bulge and I would like to close this part with one more quote from them.

.slide 54

“Each game grew in complexity, and the number of game states with it. While not explicitly a general game player, the triune brain system has the flexibility to deal with each new situation without major architecture reworking. By focusing on personality rather than winning, and data collection over code crunching, interesting AI opponents are created even in the face of complex game systems.”

.slide 55

Alright, that’s it for Battle of the Bulge and for today’s lab as a whole. Thank you for coming and see next week, when we’ll take a look at how to design AI for RTS games. Until then, take care.