Performant, persistent procedurally generated open worlds

Light Up Research

Contents

The intended outcome

There are two game styles that have become particularly prevalent within the last decade: open world, sandbox style games, and procedurally generated (often within the subset of roguelike / roguelite) games. Open world games typically have a non- or semi- linear structure of game with a limited concept of levels; instead the player is free to explore areas as they wish. The world is often populated with side missions / quests that are optional though provide the player with some benefit, with a single larger main quest that is typically more story-driven.

Procedurally generated games have core game mechanics that remain stable with a huge number of permutations of game levels / game world, procedurally generated at the start of a particular play session. Procedural generation is also used in a semi-automated way to augment the authoring of content within more linear game worlds, often used in the creation of larger open worlds when it would be impractical to manually detail every feature.

There are few examples of procedurally generated open worlds that combine both mechanics in a single game. There are a number of reasons for this:
- Open worlds often have a large amount of hand authored content that, while optional, is intended for the player to see and consumes a large amount of development effort
- There are a number of engineering problems that hinder the performance of larger worlds that are generated on runtime
- Retaining persistence and some level of simulation in an open world is difficult, with limited proven patterns and architectures in this space

There is no well known game example that provides performant, procedurally generated open worlds; while there are examples of the individual components of this framework, it was unclear whether an ‘open world’ by modern game standards could be performantly run after being procedurally generated.

This research paper tackles the second of these challenges: **performance**.

Existing system examples

As mentioned there are examples of the individual components of the problem statement at hand, the most well known of which are described below.

Note that these descriptions are based on gameplay observations rather than source code reviews; all of the games described below are closed source.

Game	Performant	Persistent	Procedural	Open	Notes

		simulation	generation	world	
Minecraft	Yes	Partial	Yes	Yes	Partial persistence - ‘blocks’ placed by the player remain, but no simulation occurs beyond a radius surrounding the player
Skyrim	Yes	No	No	Yes	
MMOs	Yes	Partial	Usually no	Partial	Partial persistence - MMOs usually run on ‘zones’ that separate players or reset on a regular basis. Performance - given MMOs can use a server-client framework with an arbitrary server size, performance options are much greater Partial open world - MMOs usually have predetermined quests / activities that are semi-linear based on player experience
Breath of the Wild	Yes	No	No	Yes	
Dwarf Fortress	Partial	Partial	Yes	Yes	Partial performance - low res graphics, known performance issues when running the game for a longer period of time Partial simulation - extremely (possibly highest) simulation fidelity though some elements of simulation radius / bubble
Starbound	Yes	No	Yes	Yes	Player actions on ‘planets’ are persistent, simulation does not take place when player is not present

Maintaining performant gameplay

Player devices are ever more powerful which enables a greater level of compute power available to game developers. However there are limitations, particularly for intense simulation mechanics in games. There are many options available for dealing with rendering limitations (LOD systems, distance fog, occlusion with carefully placed terrain) and while these may detract from the overall experience of the player, they typically do not affect the actual gameplay. This is unlike actor and world simulation, which must occur for the gameplay to actually take place, and usually occurs on the CPU.

The main challenges in procedurally generated open worlds (some shared by other game types but exacerbated in these cases) are:
- AI agent simulation and decisioning
- Physics simulations
- Navigation
- Runtime world generation
- Limited static occlusion

Solutions

As part of the research we undertook we explored four areas with potential for maintaining performance:

- Tiles & world streaming
- Seed based generation
- Navigation options
- AI LOD

Tiles & world streaming

Despite the power of modern computers, few devices have the computational power to simply render and simulate a world of any significant size that could be considered open world. Some form of tiling / chunking is typically used, where tiles surrounding the player are loaded, and tiles further away from the player are unloaded. This provides a simulation ‘bubble’ surrounding the player where action and gameplay takes place, and avoids using computational power for areas where nothing is occurring. Similarly world streaming provides a way to ‘stream’ the world into being as the player travels in a certain direction.

This poses a problem for open world simulation focused games, where agents and entities cannot simply be unloaded if they happen to be on a tile far away from the player. This can cause a phenomenon where time seems to pass only where the player is present, which is in a way true as this is only where simulation occurs. Alternatively it can cause randomly generated entities to simply disappear out of existence as their state is unloaded from memory.

We propose an additional ‘persistence layer’ that maintains agent and entity state irrespective of whether they have been loaded in a tile or not. The data stored about entities in the persistence layer is of a higher level than that of the entity itself, but retains enough state to ensure that actions the player has taken relating to entities persist beyond the time they take place.

When a tile is loaded (or generated as in the case of seed based generation detailed below), the TileManager makes a request to the PersistenceManager to check if there are any persistent entities in that tile. If there are, the TileManager will match the persistent entity to the entity it was going to generate and update the state accordingly. Similarly, when a tile is due to be unloaded, the persistent state of all persistent entities is stored in the PersistenceManager.

Persistence in procedurally generated open worlds is a detailed topic of its own accord, but the persistence layer described here enables a good level of performance while retaining an appropriate level of persistence in a tile / chunk based or world streaming setup.

Seed based generation

Most procedural generation relies on some form of randomness or noise. It is relatively common practise to use an initial piece of data or ‘seed’ that informs subsequent stages of random input, but allows a deterministic output i.e. a specific seed will also result in the same output.

Combining seed based generation with a tile based system described in the previous section provides a mechanism to generate sections of the world in runtime. When further combined with asynchronous generation, a performant means of loading near endless tiles becomes available, assuming tile generation can start and finish before the player needs to be able to access any content on a particular tile.

This removes the need to load relatively large game assets into memory, specifically terrain meshes and textures, though adds cost to the cpu in the generation of tiles. We have found this an acceptable trade-off with deliberation and tuning of the following factors:
- Tile size
- Player speed
- Player visibility
- Spawn time generation complexity for individual entities

Navigation

Navigation and pathfinding is a common requirement in games yet remains difficult and non-trivial, in part because of performance trade-offs. We explored three approaches for solving navigation:
- Navmesh baking

- Moving navmeshes
- Per location Navmesh

Navmesh baking

Many games use a pre-runtime form of this approach: baking a navigation mesh (navmesh) based on a terrain / level plus objects that need to be taken into account. A navmesh may contain areas that are simply passable or impassable, or may also contain 'costs' for area types eg a road is preferable to a swamp, so the former will have a lower cost and the latter a higher cost when agents are choosing a path.

However, non-procedural games have the benefit of being able to take any amount of time to bake the navmesh, a process which can take tens of minutes for particularly large or complex worlds, or when calculated to a low level of granularity. This is not feasible for procedurally generated worlds where the computational cost is already high given the need to compute the world itself.

In order to make this approach work, we recommend the following considerations:
- Some form of navmesh slicing, potentially on a tile by tile basis if using a tile based system as mentioned above
- Asynchronous navmesh generation, where navmeshes can be generated without impacting frame rate
- Navmesh LOD - generating a very basic navmesh first followed by subsequent more detailed iterations (potentially just one additional iteration) with the final level of granularity desired
- Navmesh caching - retaining generated navmeshes in memory if there is an impact to tile generation

With these considerations we have found tile-based navmesh baking to be a reasonable approach.

Moving navmeshes

Rather than bake a navmesh for all areas a player might explore, one alternative is to constantly update a single navmesh that surrounds the player to a predetermined distance, following the players movements to ensure there is always a navigation mesh in the players surrounding area.

Typically this area is smaller than the currently loaded world (and may even be smaller than a single tile in a tile-based system), which means although the overall computation over time for calculating runtime navmeshes may be higher, the computation at any given time is typically lower. Additionally if the player doesn't move for a length of time or moves a limited distance, there is no additional computational cost for navmesh generation.

This approach can work if the surrounding area is sufficiently large to encompass entities the player will encounter as soon as they would otherwise be encountered, and when combined with separate simulation logic for non navmesh navigation for offscreen / distant agents can work in a larger world simulation as well. However, we have found that the knock on impact of needing to tune entity

behaviour to account for a moving navmesh outweighs the benefits, as agent navigation abilities are decoupled from agent distance simulation, whereas these would typically be combined. In other words, agents far away or on an offloaded tile usually don't need a navmesh - those that are close by or on a loaded tile usually do.

Per location navmesh

To reduce the total navmesh calculation cost an alternative approach is to only bake navmeshes at key locations that entities will be encountered at. This could include settlements, enemy bases etc. This ensures that where there is otherwise empty space in the world that no entity would be using for navigation, there is no cost expended generating a navmesh for those spaces.

When combined with a player-centric moving navmesh there is some conceptual merit to this approach - navmesh generation only occurs where agents reside, and any player encounters are covered by the player navmesh. However we have found that the effectiveness of this approach depends on the wider game mechanics and world style.

Where a world has:
- A greater distance between agents that need to navigate
- terrain intended to be navigated by the player alone (potentially with a party of NPCs)
- Entities that do not travel beyond their designated 'homes'

then this approach can work. However, when a world has:
- A smaller distance between agents that need to navigate (thus reducing the optimisation and potentially creating multiple overlapping navmeshes)
- Terrain intended to be navigated by all a variety of agents, not just the player
- Entities encountered outside of designated homes

then this approach has limited performance benefit, and can in some cases limit the game's mechanics.

Overall our recommendation is that a runtime navmesh baking approach is taken for the best balance of performance and functionality, though where this calculation can occur before runtime, it should be. Where performance is critical and free roaming agents are not required, a moving navmesh approach is recommended.

AI LOD

The concept of a Level Of Detail (LOD) system in games is well known, predominantly for rendering. The application of LOD to AI is also not new, though is rarer in practice and has various versions depending on the intended outcome (simulation on / off, LOD Broker among others).

The majority of commercially implemented AI LOD systems aim to increase immersion through agents as background material - populated cities and villages with crowds. Generally the AI LOD systems function similarly in practise to the ‘simulation bubble’ idea, though with additional layers to the bubble. There are few if any AI LOD systems that provide a world simulation with activities that take place with or without the player's input.

This is for good reason: it is extremely difficult to control the player experience when agents are free to do as they wish, let alone develop and test in this environment. Previous high profile attempts at this have resulted in a cut down, lighter version of the simulation (e.g. Shenmue, Skyrim’s Radiant AI) as often critical events could take place before the player reaches or even becomes aware of a location.

Assuming a non-linear, fully open world game framework, this can be considered a positive assuming it is bounded. For instance, in an open world game with a main ‘questline’, the player can take as much time as they choose in engaging with sidequests, activities, exploration etc, and the main questline will remain static and unchanging until the player progresses it. This avoids key characters in the main questline changing state so much that the questline narrative becomes contradictory or unable to be fulfilled.

In an open world game without a main questline, this can increase immersion in a number of ways - imagining a typical hero vs villain quest, over the course of the player’s activities the previously known villain could become stronger, or defeated by a different entity that takes the original villain’s place, or the villain could defeat a known ally.

Typically simulating every action of every entity in a game world is beyond the compute power available to even modern player devices; this is where an AI LOD system becomes useful. It is of no value to a player for finely grained AI simulation to take place if the player cannot experience the simulation itself, but it is of value to a player to experience the outcome of the simulation. With an AI LOD system, an approximate simulation outcome can be reached without the need for the same level of compute power for more distant entities.

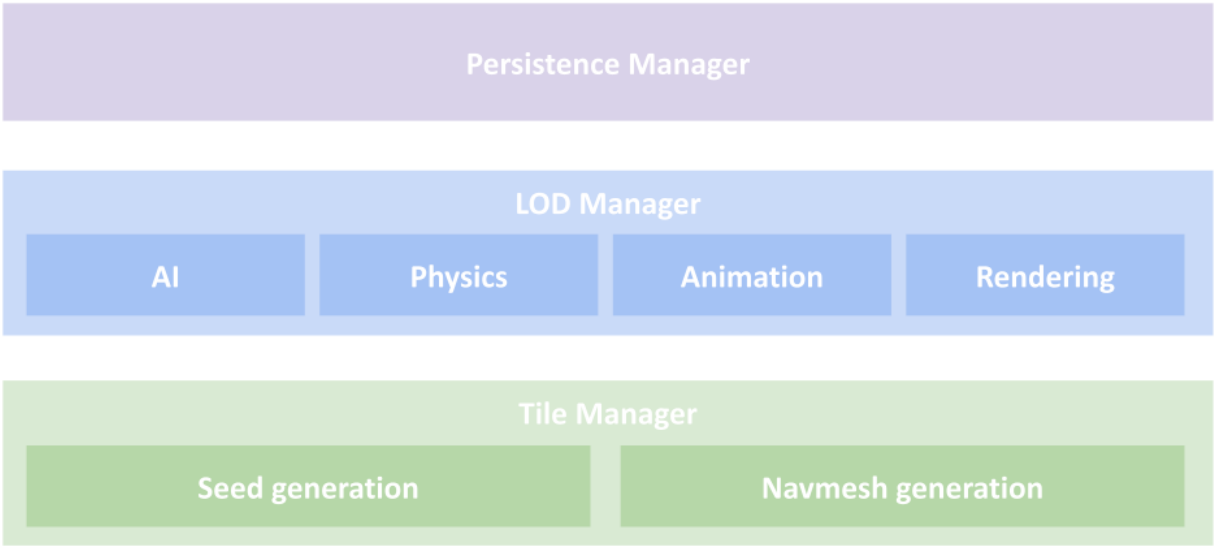
For example, rather than simulating a minor NPC going about their day, pathfinding from place to place, undertaking the activities that they would normally undertake minute after minute, the approximate results of those activities can be calculated at an hourly, daily or even weekly level. Suppose an NPC farmer is capable of producing ten units of crops per game day in ‘real time’ simulation, the same outcome can be achieved without computing the simulation that creates ten units of crops.

The key to an AI LOD system is to ensure that all of the ‘minor’ effects of a finely grained simulation are also covered (suppose in the NPC farmer example they also build skills in farming, build relationships with other NPCs, and have a random chance of events happening) - otherwise odd scenarios can occur when a player leaves a location, returns some time later, and finds specific elements of the location have advanced but others have remained stagnant. It is also important to ensure the ‘pace’ of advancement is

the same - otherwise players will find locations they spend more time in further advanced than others, which can result in a loss of credibility in immersion.

Our recommendations

To cater for situations where persistent agent simulation is desired irrespective of distance to the player, we propose a system that draws on several of the concepts mentioned previously. We call this system the “Persistent Predictive World”.



Our recommended architecture has several layers:

- Persistence Manager
- LOD Manager
- Tile Manager

The **Persistence Manager** retains the state of all entities that require simulation to some degree. This is true irrespective of their visibility, distance to the player, or any other state. The Persistence Manager should hold a minimal data structure relating to each entity, as the list / array / other container it holds will be iterated through on a highly regular basis. A side benefit of the Persistence Manager is that it can also be used as a core component in a save / load mechanism.

The **LOD Manager** covers LOD for four sub aspects - AI, Physics, Animation and Rendering - with similar but not necessarily the same LOD distances.

For AI LOD, AI should be designed such that actions can be summed up / calculated with assumptions - for example the NPC farmer described above who can produce a consistent amount of units of crops - combine with all of the minor effects that will also take place as well as random events to reflect the chaotic nature of reality. High level checks can occur for interactions with other agents - e.g. were two agents in the same location at the same hour / day / week, rather than simulating the specific location and pathfinding to calculate whether an interaction took place. Coarsely grained simulation will never exactly model complex simulation but can be tuned to advance the same way at the same rate - this requires trial and error and is specific to the exact nature of the simulation. An AI LOD system could also be linked to an optional story / drama manager with no significant performance considerations.

Physics and Animation systems should function the same way as a traditional Rendering LOD system - the further from the player, the lower the level of detail. Animations are typically more difficult to alter levels of details without manually creating additional animations, so are usually more binary (i.e. play animations or don't play animations) - some quick win Animation LOD benefits can be achieved with fewer update cycles to run the animation, or the creation of cross-entity standard animations for common tasks. Similarly Physics LOD systems should gain most benefit from either simulating or not simulating physics depending on distance - again quick wins can be achieved with fewer update cycles or more basic physics checks (e.g. AABB collisions detection rather than raycasts).

The **Tile Manager** will be a necessity in almost any open world case where some element of world streaming is needed, whether known as tiles or 'chunks', a tile will function as a discrete group of data to be loaded at a particular point in time. While the tile manager in our recommended architecture functions similarly, there should be a strong interface between the Persistence Manager and the Tile Manager that enables entities that are on a tile due to be 'unloaded' retain some element of their state in the Persistence Manager, and entities that already exist in the Persistence Manager and are queued on a tile due to be 'reloaded' are connected to the records in the Persistence Manager rather than duplicated.

This additionally can enable relationships between entities that exist across tiles, without needing to be encountered by the player and generated first. Conceptually this can be thought of as an additional conceptual layer above the tile data that is somewhat like a cache or a queue, where an entity / location / object could be procedurally generated at runtime and some data stored in the Persistence Layer, but not actually fully generated up until the point the tile it exists in is generated for the first time.

The exact nature of the tile manager will depend primarily on the compute / memory / disk budget available; tiles can be generated at the point they are encountered for the first time, then stored to disk for reloading at a later time, or regenerated from seeds every time. The Tile Manager can cheaply 'guess' which tiles to preload based on a player's direction, movement, active quest, and other factors. A more advanced 'guess' could be made based on the player's previous activities and frequently visited locations, though this will be computationally more expensive. Generally asynchronous loading of tiles is recommended as this avoids loading screens / loading bars, though a last resort loading bar is preferable to a player encountering a partially loaded tile.

Navmeshes are a significant consideration as if these are generated at runtime the computational cost is high; even if tile terrains, trees and other relatively static objects are regenerated, our recommendation is that navmeshes are cached, stored and reloaded as needed.

Summary

This research project was intended to identify whether persistent procedurally generated open worlds in games could be managed performantly, and explore and advance techniques used to achieve this goal. Our conclusion is that through the refined and advanced use of some traditional techniques (seed based generation, tiles, LOD) and new techniques (persistence management, runtime navigation etc.) then this is an achievable, though difficult, task.