Level Streaming Report

Matt Filer
17021871

University of the West of England

May 7, 2020

This report outlines the creation of a level streaming system within DirectX 11, and the background research undertaken to implement it.

1 Introduction

It can be argued that in the current gaming landscape graphical fidelity is more important than ever. A growing focus is placed on quality of assets, particularly in triple-A titles, which results in an exponential growth of memory budgets as assets increase in size. While varying methods exist for compressing file sizes such as the direct draw surface (DDS) textures supported by DirectX, and index lists for vertices in model files such as object files (OBJ), keeping runtime memory usage low remains a challenge.

Figure 1: *An example of a modern game reliant on streaming, The Division 2.*

Methods must be implemented to optimise the amount of data kept in memory (RAM) at any one time to avoid exceeding hardware limitations, particularly when targeting console platforms. A popular method utilised ever more frequently is known as streaming. In a streaming system, assets such as models and textures in a level are only loaded when required; for example, if in the player's view distance. Doing this allows for large (even infinite) levels, without keeping every object in the level in memory at runtime.

Systems such as this were unachievable in the past due to slower read speeds on older hardware, however with the increased usage of SSDs and faster hard disks, assets can now be quickly called from a filesystem and loaded into memory when required with minimal visual "pop-in". As such, in recent years a number of titles have adopted this model to allow for expansive environments without constant loading screens, such as God of War and Spiderman; with support for streaming even being natively implemented into popular public engines such as Unreal and Unity.

This report will detail research into the field, and an implementation of a level streaming system in DirectX11, with optimal runtime memory usage through asset pooling, and efficient read speeds through custom binary formats for assets.

2 Related Work

Researched heavily for this project were the streaming systems implemented into 2014's Alien: Isolation, developed by Creative Assembly. In this cross-platform system, runtime asset streaming was implemented for models and textures; with per-level sounds, shaders, and low quality mipmapped texture variants being persistently loaded in memory during gameplay. Streaming of scripts and full levels (eliminating loading screens) was experimented with, although these experiments ultimately failed to ship with the final project due to development time constraints.

Alien: Isolation's streaming system utilised a multi-part proprietary archive format which supplied asset information as well as the content of assets themselves; with information available in a .BIN file (acting as a

compilation of header data), and content itself available in a .PAK file (able to be located with offsets from the header data). Asset content within the .PAK files could be ordered by the team's in-house tools to optimise reading efficiency, based on the asset's location in world and frequency within levels, particularly useful as the game shipped across past-gen consoles with slow drive read speeds.

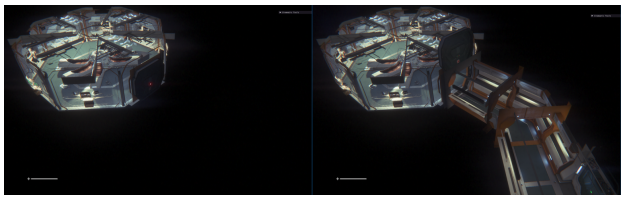

Figure 2: *A demonstration of an area loading in Alien: Isolation after a player opens a door (left unopened, right opened).*

Zones within a level determine which assets should be loaded at any one time. These zones are either activated by doors which trigger the load of the next zone and open when finished, or invisible linking entities which allow for neighbouring areas to be loaded based on distance; useful for unloading things around corners in corridors. Initially entities were determined to be within a zone based on their origin in the world, however this was changed during development to be something controlled by the artists; with each entity in the editor then being colour coded to match its corresponding zone.

For unloading content in zones a timer was used, ensuring that if the player had to quickly double back on themselves content wasn't unnecessarily unloaded and then reloaded, optimising calls to disk. This timer could be overridden by artists to make the system flexible for certain level designs. Lead engine programmer Michael Bailey noted that one section of the game was intentionally redesigned to suit the game's streaming system, being a large shopping mall: the decision was made to split this section in half to block visibility, reducing the amount of geometry in view at any one time, allowing the streaming's performance benefits to be more effective (Bailey, 2020).

Other similar systems looked at in the research for this project aside from Alien: Isolation's include Insomniac Games' 2019 title Spiderman, and Santa Monica Studios' 2018 title God of War. While Spiderman, like Alien: Isolation, failed to ship as a seamless experience, streaming was heavily utilised for moving around its open world. Based in Manhattan, the real-world grid layout of the city provided the developers a way to segment the world into uniform square zones in a grid formation. These zones were internally broken down into different subsets such as props, buildings, and markers; allowing multiple disciplines to work on an area of the map at any one time. A zone is called to load into detail at runtime based on the player's current position, loading their current zone along with all neighbouring zones in the grid. When the player is travelling at speed while web swinging, only the current zone and three ahead in the forward direction are loaded into detail, with motion blur being utilised to hide the load of additional tiles if the player changed direction (Ruskin, 2019).

This zone technique is similar to that used in God of War, where in which the world is one large map broken up into segments known as "wads" which are streamable zones internally subdivided into smaller chunks that can be locked when being edited to avoid merge conflicts. These wads are generated from a reference hierarchy of Maya files, making the system easier to understand for artists. Wads are connected by "refnodes" in editor, similar to the invisible linking entities in Alien: Isolation's system, rather than being on a neighbouring grid like Spiderman, with three wads always being loaded at any one time: one that the player is currently within, one that the player has come from, and one that the player is about to enter; this process was described as a "conveyor belt". Due to this linear loading system, God of War successfully managed to ship as a seamless experience (Hobson, 2019).

Figure 3: *A screenshot of God of War.*

3 Method

Initially a method was implemented similar to that of Alien: Isolation's original zoning system, with users being able to place boxed zones in a level, which automatically captured all models (entities) with an origin within that zone. This became hard to manage with larger levels however, as model origins were often away from the visible position of the model, placing objects in unexpected zones and leaving holes in levels when unloaded. For this reason, the system was changed to allow zones to be created and models (entities) to be created as children to them. While this system worked, it was difficult to find the right way to implement zone linking; and for that reason a grid based approach was taken, similar to that used in Spiderman.

This system divides the entities in a level into tiles in a grid, with the grid stretching just further than the

extents of each origin in the level. The dimensions of the grid can be customised by the user, as for example on big levels you would require a higher subdivision count than a smaller level. This speeds up the process of using the system enormously and although takes away some artist control, means that zones are automatically aware of their neighbours without links being set up, and always to a uniform size.

With this system in place, using knowledge of neighbouring zone tiles in the grid, level of detail functionality was implemented; with the tile that the player is currently within being loaded in full detail, and all tiles neighbouring the active tile being loaded in low detail; all other tiles are unloaded fully. This implementation is customisable, so for example, tiles neighbouring the active tile could be loaded in full detail, while all other tiles in the world within a given radius could be loaded in low detail. To generate the low levels of detail for the demo scenes in the project, Blender was utilised with its decimate functionality, to reduce the poly count of models. In a live project these lower poly assets would be created by artists.

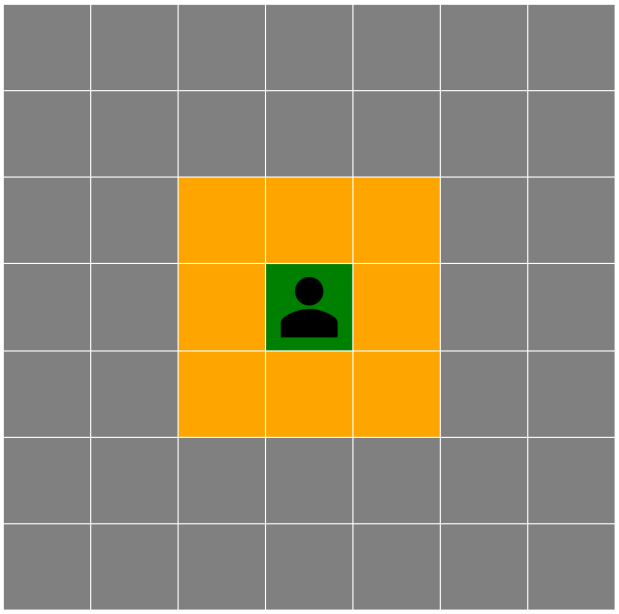

Figure 4: *Loaded tiles around the player. Green = high LOD, orange = low LOD, grey = unloaded.*

To stream content in, a two part archive format was created, similar in style to Alien: Isolation's, with header information contained in a smaller .BIN file, and content in a large .PAK file. This two part system allowed metadata for models and textures to be read in at the load of a level, stored per zone, and then easily called to load from the .PAK on the filesystem when content was required. This system worked efficiently at runtime, with the archive files being created using a toolkit developed for the project, which could also instance levels, able to be dynamically read in by the engine's scene manager to list available levels to load at runtime from a debug menu.

In order to reduce memory overheads, when models are loaded in from the filesystem they are added to a per-level pool. Each model instance within the level then references model data from this pool when it is time to render, rather than storing it within its own instance. This exponentially saves on memory when multiple objects in the level use the same model, as the buffer only exists once for all instances, rather than for every instance; for example: 10 models using a 50mb buffer would only use 50mb of memory with this system, but 500mb without.

To unload models from the per-level pool, useage counters are kept for each buffer. When an instance of a model uses the buffer it increments the count, and when and instance of a model is unloaded (within a zone that is no longer active) it decreases the count. If a buffer in the pool has a useage count of zero it is unloaded after the next zone load (or unload) is finished – this means that models can be shared across zones, but are swiftly unloaded if not required, minimising memory useage, and also saves reloading model buffers across zones.

Figure 5: *The system's editor in action.*

The loading of content within tiles in the environment was performed on a separate thread from the main render and update loop, allowing for no interruptions to gameplay or noticeable performance impact. To do this, content is pulled in and instantiated in memory on a "loading thread", and when ready to be rendered, is pushed to the main thread when the current render call has finished. This prevents null references to content which has been partially loaded.

4 Evaluation

The final system produced for this report is efficient, customisable, and robust. With that said, a potential improvement would be targeting memory overheads for the application, based on the amount of content within a level and the allowances of the hardware. This memory overhead could then be used to keep frequently occurring assets within the level loaded into memory ready to be called again, rather than pulling

them from disk often. This could potentially give a performance gain on a large test scene.

Changes were made to the system from its initial OBJ reading implementation to the BIN/PAK configuration mid-development which saw a performance boost. Not only did this optimise loading times due to the switch to a binary model format, but also tidied the project's assets folder – being just two files for all models, rather than a large directory of them as was previously the case.

Model pooling also saw a noticeable impact on memory usage. Differences before and after implementing the system can be seen in the figure 6, with a similar test level.

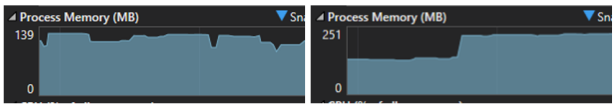

Figure 6: *Two similar small scenes loaded before (right) and after (left) model batching.*

5 Conclusion

The result of this project is an engine that can load large areas with very little overhead, customisable by artists or technical designers through a simple UI. The grid system is simple to use and can be modified through code to expand the number of currently loaded tiles to suit the requirements of the implementation.

If the project was to be expanded further, it would be useful to implement further artist controls for the grid system, allowing tweaks to certain extra parameters without modifying code: such as view radius, or level of detail cut-off. Similarly, it would also be good to include some debugging controls aside from the current grid visualisation in editor mode, to allow for developers to test overheads and performance issues with certain zones.

Bibliography

Bailey, Michael (2020). "Michael Bailey Interview (by Matt Filer)". In: URL: https://drive.google.com/file/d/1Z7Eh0bkBpBRWCsiluMsgGaA9D7oiOR5s/view?usp=sharing.

Hobson, Josh (2019). "The Indirect Lighting Pipeline of 'God of War'". GDC. URL: https://www.gdcvault.com/play/1026046/The-Indirect-Lighting-Pipeline-of.

Ruskin, Elan (2019). "'Marvel's Spider-Man': A Technical Postmortem". GDC. URL: https://www.gdcvault.com/play/1026496/-Marvel-s-Spider-Man.