Timo Luukkonen

Resource Management in Game Engine Development

Business Information Technology

School of Business

Spring 2015

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä(t): Luukkonen Timo

Työn nimi: Resurssienhallinta pelimoottorikehityksessä

Tutkintonimike: Tradenomi(tietojenkäsittely)

Asiasanat: peliohjelmointi, käsittely, hallinta, peliala

Tämän opinnäytetyön toimeksiantaja on Frozenbyte Oy, joka on pelejä kehittävä yritys, jonka kehittämä pelimoottori tukee monia laitealustoja. Tämä usean laitealustan tuki vaatii peliin sisältyvien resurssien muokkaamista eri muotoihin ja formaatteihin kullekin alustalle. Tämä puolestaan vaatii tehokkaan järjestelmän, joka kykenee hallitsemaan resurssien muokkausprosessia alkuperäisistä tiedostoista sopiviin muotoihin.

Tämä aihe valittiin opinnäytetyötä varten johtuen yrityksen tarpeesta jatkokehittää pelimoottorin resurssienprosessointijärjestelmää. Suunnitelmat jatkokehitykseen puolestaan ajoittuivat samoihin aikoihin kirjoittajan opinnäytetyön aiheen tarpeen kanssa.

Opinnäytetyö tutkii pelimoottorien resursseja ja niiden hallintaa pelimoottorin kehittäjien näkökulmasta, aloittaen määrittelemällä mitä nämä resurssit peleistä ulkoisena tietona ja pelimoottorit pelien kehitystä tukevina sovelluksina. Hyvää resurssienhallintaa edistävät asiat, kuten tehokas muistinkäyttö ja resurssien pakkaus arkistoihin kohdelevyille selitetään. Pelimoottoriin ulkoisesti luodut tiedostot saattavat tarvita muokkauksen eri muotoon, ja tämä mahdollisesti monimutkainen prosessi täytyy hallita hyvin.

Osa opinnäytetyöstä sisältää myös dokumentaatiota ja ajatuksia toimeksiantavan yrityksen resurssienkäsittelyjärjestelmään tehtyyn työhön liittyen. Tämä osuus toimii myös käytännön esimerkkinä muussa opinnäytetyössä esitettyihin asioihin. Kehitystarpeisiin pohjautuneet suunnitelmat ja työn tarkka luonne selvitetään. Osiossa myös ilmenee käsittelyprosessin hallinnan kehityksen suurehkot aikatarpeet ja testauksen tärkeys.

Työn käytännön osuuden aikana jatkokehitetty resurssien käsittelyjärjestelmä on käytössä toimeksiantavassa yrityksessä ja on siltä osin onnistunut. Kehitys tosin vaati alkuperäistä enemmän aikaa, mutta työn tulokset ovat suurimmilta osin onnistuneet.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author(s): Luukkonen Timo

Title of the Publication: Resource Management in Game Engine Development

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: resource, game engine, management asset, preprocessing

This is a thesis commissioned by Frozenbyte, Inc., a game development company with a game engine supporting various different target platforms produced by the company itself. The support for many platforms requires using different forms and variations of game assets and other resources which in turn necessitates an efficient system producing the different variations from the original files.

This topic was chosen for the thesis due to the company's game engine's resource processing system requiring maintenance and further development. The plans for improvement happened to overlap with the author's need for a topic for his thesis.

This thesis itself investigates resources and resource handling in game engines from the game engine developer's perspective, starting by defining resources as external data and game engines as frameworks for games. Aspects found in a good managing system such as resource archiving for file systems as well as effective memory management for runtime resource handling are explained and appraised. Data exported into the game engine may also require alteration, and the process of altering the data must be well managed.

One part of the thesis also contains documentation and thoughts on the work done on the commissioning company's resource processing system. This also serves as a practical example for the topics covered by the rest of the thesis. Planning based on set targets of improvement and the precise nature of the work is explained. In this part it is found that sizeable amount of time and testing needed to develop processing management systems.

The resource processing system that was further developed when writing this thesis is actively in use in the commissioning company, and thus is a success. The development required noticeably more time than initially estimated, but the results are more or less successful.

TABLE OF CONTENTS

LIST OF SYMBOLS

Asset	A part of a game that is created with an external program, and added to the game afterwards. For example, models, textures, sound effects and animations.
Atlas	A composite of various different textures in one image file.
Compression	Act of altering data in a way that the result contains less bits of data, and the operation is reversible either completely or partially.
Destination File	A preprocessed file that has been somehow processed from a source file.
Game Engine	A framework containing existing functionality upon which games are built upon using code and different resources. .
Resource	A part of a game that is not programmed into the game, but is a vital part of the game itself. Usually used to refer to non-asset data, such as scripts, shaders, level data.
Source File	A file that is being processed, and will result in one or many destination files.
Texture	An image consisting of pixels that is used to show imagery in games.

.

1 INTRODUCTION

Gameplay in a complete game is built using code provided by programmers, but most of what the player sees and experiences is not defined in code, but instead produced assets, such as images, 3D-models and sound. These are usually created in external applications, which may or may not be able to supply the data to the game engine in the form it is able to comprehend. And even if they are compatible, some ways of arranging the information of these assets are bound to be more optimal to others. Game Engines, with which games are built, should be able to aptly perform some of this work, and separate systems should be made to handle the rest.

The purpose of this thesis is to investigate the management of assets and resources that are created to be used in a game, and to try to discover different useful ways to approach this problem. The thesis approaches the problem with efficient workflow and maximal compatibility with different platforms and circumstances in mind whilst keeping in mind the performance requirements of modern video games. The goal of the practical portion of the thesis is to further develop and maintain a resource processing system used daily in real-life game development scenarios by the commissioner, Frozenbyte, Inc.

The topic of the thesis came from Frozenbyte, Inc. during the spring of 2015, the commissioner of the thesis and a long-lived game developer located in Helsinki, Finland. The topic was relevant for the company at the time as the company's processing system was deemed to require revising and maintenance.

The optimal end result of the thesis is for the author to become acquainted enough with the premise and objectives of asset management to work with systems built to perform such tasks. Some additional emphasis should be placed on resource processing management due to the practical work being performed for Frozenbyte, Inc. The commissioner's system which was planned to be improved should also be functional in its refactored state and in use internally by the company before the year 2016.

2 RESOURCES AND ASSETS IN GAME ENGINES

It is often that resources are more in relation to game engines than the game itself. This is at least true when discussing about the pipeline the external files go through to actually become tangible visuals or other objects in the game itself. So in order to effectively examine factual information about resources, the concept of what a game engine is and what it encompasses must be established.

2.1 Game Engines

A game engine is perhaps best described as a framework upon which games are built. It is a system which has some basic functionality games require already implemented. Usually this functionality is general in nature, so that most games can efficiently utilize the engine. There is no need for a game to use everything a specific engine can provide, however. (Bensmiley, 2012.)

2.1.1 Games vs Game Engines

Rather than being entirely separate entities, game engines are often somewhat entwined to the game or at least type of games they are used to make, though this may vary. Some game engines may, for example, contain certain functionality that may be directly part of a game. Others may provide tools, settings and other ways to build this functionality along with other variations of it. While having game features built into the engine may provide faster development time, they can also hinder making games that require different sets of features. Therefore the term ‘game engine’ is best used to describe systems which allow for the efficient production of various different types of games, rather than only having ready feature sets for a one type of game. (Gregory, 2015)

It is impossible for a game engine to cater to the development process of all games, though it may overall be more useful for a certain type of game. While a first-person shooter game may require a highly responsive camera control system, the same system will not be applicable for a real-time strategy game. Massively multiplayer online games require management for large amounts of servers, which other types of games do not. This is not to say that these features get in the way of developing different types of games, but if they are hardcoded in, it may be difficult to create something that the engine was not made for. (Gregory, 2015)

2.1.2 Common Functions of Game Engines

The most common pieces of functionality a certain game engine may provide are graphics, sound, physics, visual (particle) effects, user interface, and networking. These functions usually have their own systems in the engine that may or may not be somewhat separate from each other. (Bensmiley, 2012.)

Graphics in game engines can mean drawing potentially animated 2D textures on screen to form images that make up what the player sees in the final product. In 3D games, the textures are placed to envelop 3D meshes, which are forms which define a shape in a 3D environment. (Bensmiley, 2012.)

Sound effects and music are what the players hear in a game. A game engine may support these two in a bit separate manner, but will usually at least have a way to efficiently play sound effects with little latency. (Bensmiley, 2012.)

Physics usually means having support for rigid body physics, different joints between objects and collisions. These can be used to simulate real world physics as well as to trigger gameplay events when two objects collide or overlap, for example. (Bensmiley, 2012.)

Visual effect systems provided by a game engine are usually based around particles. These are small visual objects that are simulated by using variables like velocity, acceleration, size and color. (Bensmiley, 2012.)

User interface support supplies the developer with ways to easily compose menus for the player to navigate and implement text input and output. At the very least, game engines should have existing functionality for handling different fonts, so that the developer can implement the beforementioned systems. (Bensmiley, 2012.)

2.2 Resources and Assets

Games are composed of external data that makes up for most of what the players experience when they play the game. These external data files are called resources. The information these files contain may be related to various different subsystems and features of the game engine. Some common resource types include art and sound, scripts, shader files, materials, and level data. (Jason Gregory, 2015.)

An asset is a type of resource file that is in a format that can be used by the game engine, and will be presented to the player. This can be interpreted as nearly every part of a game aside for code scripts, internal data and documentation. Though the word asset can also be used to refer to a visually or aurally representable piece of data such as sound, art or video (DigitalTutors, 2013). (Ben Davis, 2009.)

2.3 Data Formats

A file format defines how information is expressed inside a file. There are different ways to sort data into a file for different types of data. Image files, for example, can be stored in JPEG (Joint Photographic Experts Group) or TIFF (Tagged Image File Format) formats, which differ greatly in both function and format. Documents, however, use different formats altogether, such as PDF (Portable Document Format). (Andersen, 2015.)

2.3.1 Binary vs. Text

A binary file is a file where the file consist of bits, each of which storing a value of one and zero. These form bytes, which consist of 8 bits. Each byte is a single unit in storing different types of data, abstracting the concept of ones and zeroes to a more useful data unit. It is up to the programs to interpret these values to human-readable data. (University of Maryland, 2003.)

A text file is a file where the data it contains is comprehended, viewed, and edited as a series of characters. The way data is stored on the memory, is no different from binary files, however. Text files are in fact just binary files that are regarded as containing text data, and thus the program may display it in user-friendly text format. (University of Maryland, 2003.)

Object serialization is an operation that usually has the option of storing the data in either text or binary format. One text format is the human-readable XML file format, which allows users to manually modify the file contents. It is also more compatible with other software as the information is stored in standard format. For binary file formats, manual modifications are very difficult, but instead it is simpler to get complete object copies of the serialized objects. This is because there is no need to convert the data that is stored in memory in binary format before saving the result. Binary files are also more compact. (CareerRide.com, 2015.)

2.3.2 Compression

Data compression is an operation where the required amount of bits to express a certain piece of information is reduced. This is performed in a way which enables the operation to either completely or partially undoable, so that the compressed data can be recovered. Compression can result in smaller memory footprint and need of network bandwidth and less consumed disk space depending of the specific scenario of usage (Rouse, 2015a.)

This is usually accomplished by specific algorithms which define how the data is altered to achieve compression. For example, a simple way of compressing text is specifying a single character to mean a set string, and replacing instances of said string with the single character. (Rouse, 2015a.)

As an example of this, consider the following text string:

aaaabaaaabaaaabaaaabaaaabaaaab

This string can be compressed by replacing the portions that are ‘aaaa’ with the letter c. This results in the following string:

cbcbcbcbcbcb

The result is significantly shorter. To reverse the operation, one has to replace the instances of letter c back to the string ‘aaaa’. Most compression methods are much more complex and more efficient, though.

Compression algorithms can be either lossy or lossless. Lossless compression means that the original data is fully and completely recoverable from the compressed version of data. Lossless compression is often the most viable option for sensitive types of data, such as executables and text. Slight variations in file contents could make the uncompressed data unusable and faulty. The GIF (Graphics Interchange File) is an example of a lossless image compression file format. (Rouse, 2015b.)

Lossy compression, on the other hand, permanently loses parts of the original data in the compression process. Not all of the data is recoverable from the compressed file. This is acceptable for video and audio data, for example, where small loss in image quality may not be noticeable or harmful. The JPEG format is an example of a lossy image compression format, where users can adjust how much image quality they will trade for smaller file size. (Rouse, 2015b.)

3 RESOURCE MANAGEMENT IN GAME ENGINES

As much of the user experience is defined by singular resources, managing them correctly and efficiently in a game session is vital. Nearly all games utilize at least one sort of resources, but usually do not need all resources to be ready for use at all times. This is why effective managing of resources is highly dependent of loading and unloading the right resources to and from memory at the correct time, and ensuring that they are in the correct format and specifications for use. (Thorn (2011.)

Asset pipeline is a term used to refer to the different steps taken by assets before they are in use on their intended application. So the pipeline is as the name implies, one puts assets constructed by the developers on one end, and game-ready versions of the assets come from the other end. The result that comes from the pipeline is the main objective of the process that forms the middle part of the pipe. The particular use for the ready assets may vary somewhat. Not all assets are even needed in the final product. The end result should be as complete as possible, and as easy for the target application to utilize as possible, so that the end-user does not have to wait for the resources to be further processed. This pipeline concept is extensible to resources as the two concepts are somewhat similar in nature, and both types of data require managing. (Carter, 2004.)

3.1 Resource Management

The need for proper resource management comes from the inherent nature of the devices the games developed using game engines are run on. It is often the case that not all resources are able to remain in the system’s memory, as it is limited in capacity. If the capacity that is available for the game is exceeded, the results are undefined and usually unwanted. To avoid this, the resources must be either able to all fit into the allocated memory, or be dynamically loaded and unloaded based on what is happening in the game. The latter option is more feasible on engines used on large-scale games, as the amount of resources in the game is very large, as not all resources are necessarily loaded at any given time, reducing the memory required. (Thorn (2011.)

There is also the problem of arranging files efficiently on the disk. Some engines allow and prefer to have the resources intuitively in a tree of directories. For the engine, the exact location of the resource in this tree does not necessarily matter, but for the people creating and managing these assets this is beneficial. (Gregory, 2015)

3.2 Performance

While dynamically loading and unloading resources to make sure the memory quota is not reached or exceeded, performance is not a given. It has to be kept in mind that while memory is fast enough to allocate and unallocated on the fly, the disk from which the content is loaded usually is not. There are some techniques that can be used to minimize the negative effect accessing files on disk can have on the game’s performance. (Lönn, 2015.)

3.2.1 Streaming From Disk

One way to limit the memory footprint of managed resources is to dynamically load them from disk during gameplay. Implementing this transparently so that the user does not notice any decreases in the game’s performance is not simple, though. The game must have the necessary resources already loaded when they are needed, so loading must be implemented so that it loads the resources before-hand. This can be achieved by triggering the resources’ loading process by some gameplay event that precedes another where the resource is needed. The other is to load and unload resources based on certain parameters, such as the player's location in the game world. (Lönn, 2015.)

Streaming assets to memory during gameplay needs to be designed with care. The average seek time of the source media, and the actual transfer rate, is subject to variance. This means that there are very little guarantees as to exactly when the resource loading process completes. It may potentially take hundreds of milliseconds, and thus it is a must to implement the loading operation asynchronously so that the game runs normally while the loading occurs. (Lönn, 2015.)

It is possible to load textures at detail level at a time during the game's execution. By doing this the game may display lower quality textures to the player, before all detail levels are loaded into memory. This is often preferable to having the game's performance degrade or having the texture be temporarily unavailable altogether. Figure 1 shows the difference between the lowest level and highest detail level textures that are both visible by the players during gameplay. On the lower left is what the player might see for an instant if he is suddenly placed in the environment as the game engine quickly loads the lowest-level textures. The scenery is quickly replaced with the high-detail version that can be viewed on the upper right. (Reto.Injection, 2013)

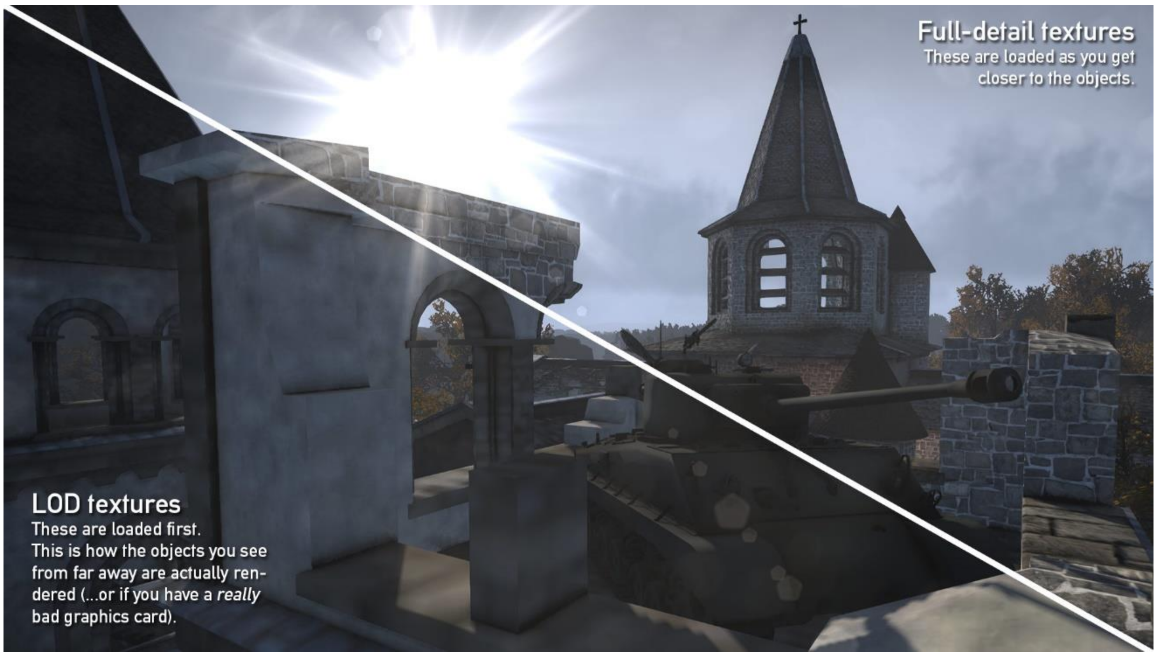
Full-detail textures
These are loaded as you get closer to the objects.
LOD textures
These are loaded first.
This is how the objects you see from far away are actually rendered (...or if you have a *really* bad graphics card).

Figure 1. Streaming Lower detail level textures first (Reto.Injection, 2013)

3.2.2 Archiving Resources

As most game engines attempt to support a variety of different gaming platforms, the resource loading from physical media must be designed to be functioning efficient in a variety of file systems. File path length limits and other variables may work differently on different systems, and yet organizing resources in a sensible way when developing the game is desired. If nothing is done, loading resources from different folders, for example, may result in additional overhead. Despite this, from the perspective of project resource management having resources in neatly separated folders is optimal. The solution to some of these problems is compressing the files to archives, files containing the data of multiple resource files. (Lönn, 2015.)

The major benefit to using archives instead of singular files is that the file itself may have very simple properties that support most if not all file systems. The potential complex properties such as a tree of directories may exist inside the archives, so the organized fashion the resources may have originally been in is also available when the game software is released. There are performance benefits to archiving as well. If the seek time of the physical media is high, archiving can significantly reduce its effect, as only few archive files need to be sought and opened. If archives are not utilized, this seek has to be done for each file resource accessed. Since there are only a few archive files on disk, the files can also be left open by the code to further optimize performance. (Lönn, 2015.)

3.3 Resource Processing

Preprocessing data before handing it to the next step in the pipeline is not uncommon outside games. For example, C++, the programming language, uses different directives the user can input into the source code to alter the content that is given to the compiler. This makes getting a multitude of different end results from the same piece of source code possible, as it is given different parameters. (CppReference.com, 2014)

For game engines, this preprocessing refers to modifying resources to be more applicable in different circumstances. For assets such as textures this usually means reducing file size at the expense of quality so that the data will better fit in memory and depending on the asset require less processing power from the hardware. Another objective that can be achieved is change in the file format to better accommodate the game engine's and target platform's requirements. Changing the file format can also have other benefits. As an example of this, compiling Lua scripts to bytecode via the Lua compiler improves the load time for the scripts, protects the normally text-based files from accidental user changes, and provides syntax checking before the scripts are executed in the game engine (Figueiredo & Ierusalimschy & Celes(2015). Preprocessing does not necessarily have to relate to externally produced files. It can also be performed on other data, such as level files. For example, Unreal Engine 4 bakes static lighting into lightmaps using the level that has been made using the engine's internal editor (Epic Games, 2015). (Carter, 2005)

Figure 2 portrays a possible resource pipeline from the original resource to the intermediate and final version for textures and 3D models. These assets are first processed into an intermediate form, which is used as a quick alternative to the final produce of the system. These are put through conversion software, which processes the data to a format optimal for each target platform, making them ready for use in the final product. (Noel, 2004.)

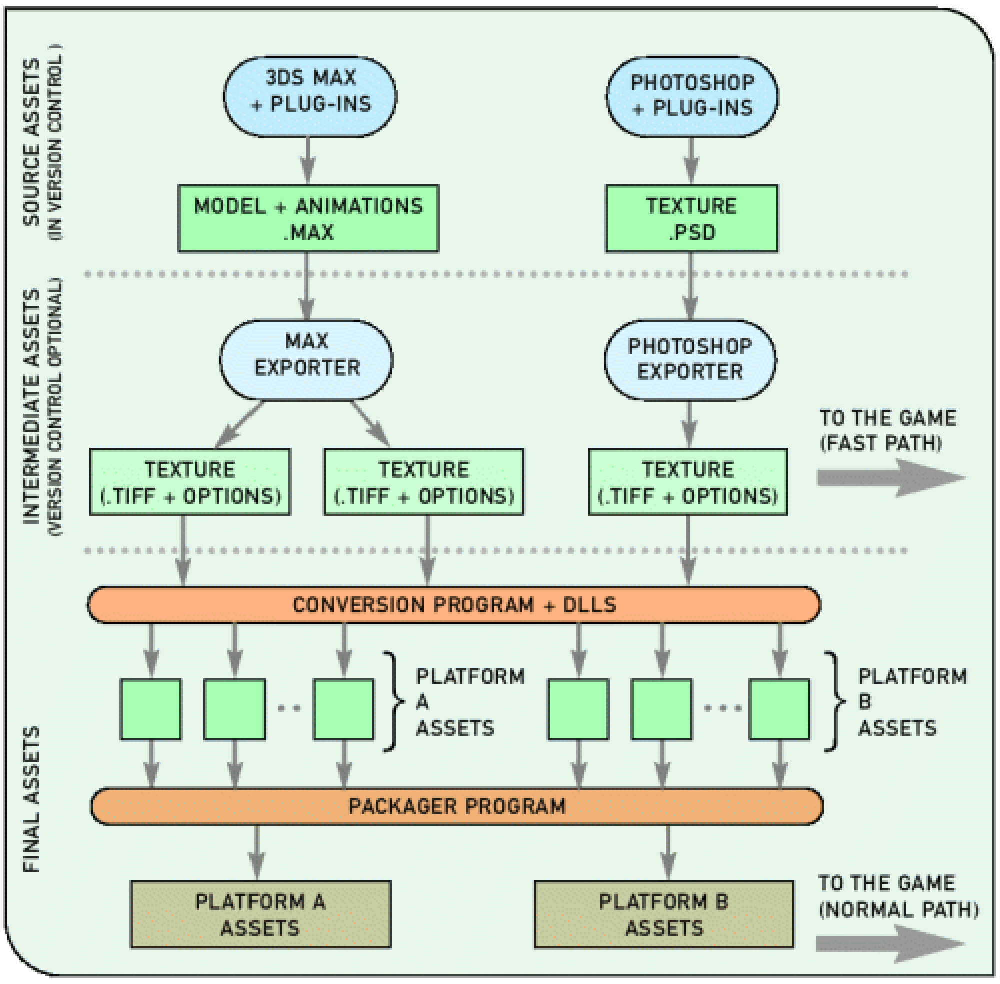

Figure 2. Potential asset pipeline with preprocessing for games (Noel, 2004.)

3.3.1 Management

Managing a preprocessing system that has to modify data reliably and with good performance is not simple. There are a lot of aspects that have to be implemented that are very separate from the actual processing logic that makes the alterations to the data. For instance, it is not reasonable to always reprocess all the resources when in reality only a single file has changed. It is often reasonable to implement a whole different system to manage the process, as the effectiveness of the management directly translates to performance. This is because the less false positives there are for processing targets, the less preprocessing needs to occur in the first place. (Carter, 2005.)

There are various ways to handle preprocessing, and one way is to set up dependencies and composite intermediate versions of the different resource types. For example, a complete 3D model may consist of the mesh and a texture, both of which must be successfully preprocessed before the mesh can be rendered in the game. After the successful processing of these two files, they may be again modified to fit each different parameter the system may enforce. (Carter, 2005.)

The process of running assets through an established processing pipeline is one that is easily automatable. Acquiring files as well as executing the different steps along the procedure can be accomplished by using scripts. To effectively run the system with little to no human interaction, it should be able to remain functional to an extent even when special circumstances such as cases of network outage occur. (Noel, 2004.)

It is important to note that even after the actual preprocessing logic is managed and the data is successfully run through the pipeline automatically at certain intervals or when required, it is beneficial to collect statistics and diagnostics of the process. It is simple to troubleshoot the system if one can receive errors and warnings regarding its process, and other statistics such as total data size is useful information to have. (Noel, 2004.)

3.3.2 Changes in Source Files

Before the system may start to process any resources, it should know that the source files in question have changed, that updating the preprocessed data is required. This can be accomplished in various different ways. If the resources are under version control, the simplest way would be to compare the revision numbers of the resources to the revision number of the resource that has been previously processed. (Carter, 2005.)

If this is not available a common solution is to have the system work by checking the ‘last modified’ dates for the source and destination files. If the source file is newer than the latest preprocessed file, then it can be deduced that the preprocessed destination file requires updating. This is not completely reliable, as having an incorrect time set in the preprocessing system’s computer may lead in inconsistencies and errors. This is fast to perform though, and should be a useful alternative if performance is especially critical. (Carter, 2005.)

Perhaps the most robust way is to check the file’s checksums using a strong enough of an algorithm such as MD5. This gives each unique resource file its own very near-unique descriptor of its contents. This checksum can be used to accurately determine whether a file has changed. This is a relatively slow operation, as the whole source file has to be read, and the checksum for the read data must be calculated. (Carter, 2005.)

3.3.3 Changes in Processing Code

While the resources themselves can change and thus require reprocessing, the processing code can also be modified when the game engine is being developed. This will result in relevant resources having to be reprocessed. This should be handled efficiently if changes in the processing logic are common. If the engine and the processing logic is not being further developed, the development team may opt to simply reprocessing all files if the code happens to change. If the logic changes often enough, the system should handle the changes more intelligently, or valuable time and processing power will be wasted. (Carter, 2005.)

There are options for managing these changes. One may apply a version number into the preprocessing code itself. This version would be updated along with the processing code. The system only has to check whether the code’s version number has changed, and reprocess all assets that depend on the code if it has. This way, there will be nearly no redundant reprocessing done. This also allows greater control for the engine developer as to when the reprocessing has to be performed. (Carter, 2005.)

It is also possible to have the version data inside the output files. This allows for per-resource handling of the versioning. It also allows finer control over the reprocessing. For example, when updating the processing code to version three, the developer may choose to reprocess a file which has been preprocessed with version one of the software, but not reprocess ones that have been processed by version two of the software. This is useful if there are no relevant changes for the asset between version two and three. (Carter, 2005.)

4 CASE – RESOURCE PROCESSING IN THE FROZENBYTE GAME ENGINE

This chapter depicts and documents the work and background knowledge of the resource processing system development and maintenance done for Frozenbyte, Inc. Work was done 40 hours per week during weekdays, and with this pace the specified changes were implemented and the system renewed in roughly four months. This includes research on the original system, planning, and project management, as well as time not spent on actively working on the system such as breaks and meetings..

4.1 The Frozenbyte Game Engine

Frozenbyte, Inc. has developed its own game engine to power games on multiple console and desktop platforms. It is based on a multitude of third party libraries. It utilizes DirectX, OpenGL and SDL, as well as Wwise for audio and Nvidia PhysX for physics simulation. The engine itself is written in C++, and features Lua for scripting, and C# for Editor UI.

4.1.1 Main Features of the Frozenbyte Game Engine

The latest iteration of the game engine, which was made for Trine 2 in 2011, has been used to publish games for seven platforms: Nintendo Wii U, Sony Playstation 3 and Sony Playstation 4, Microsoft Xbox 360, Android, Windows, Linux and OSX.

The Frozenbyte engine supports 3D game development, and has libraries and functions that enable effective game development. It features a complete editor which is extensively used within the compan*y*, and which is also publicly available as a level editor for Trine 2, Trine Enchanted Edition and Trine 3(O’Connor, KaiFB, 2014).

4.1.2 Resources in the Frozenbyte Game Engine

Frozenbyte’s Game Engine supports various different asset types which empower the company’s employees to create content for their games. The engine supports only a single or few data formats per asset type, as there is no need to implement alternatives due to the company having set asset pipelines. This is in contrast to commonly available game engines, as they usually have to support several different developers having different asset pipelines altogether. In this chapter the most numerous of resources in the Frozenbyte’s development pipeline are examined.

4.1.2.1 FBX

FBX data format can store animations, 3D models and rigging information in a format which was originally developed by Kaydara, a company Autodesk Inc. bought in 2006 along with the rights for the format. The FBX format and the software required for its effective use are free, though the source code and license are regulated by Autodesk. This means that there is no real documentation on the format that would allow more customized solutions for the developers using FBX-files. Even so, the format is widely utilized by 3D modelers. The FBX file format supports saving data in both binary and text format. (Blender Foundation, 2013).

4.1.2.2 TrueVision Targa

The TrueVision Targa image format is a bitmap format which supports colours up to 32-bits per pixels, both compressed and uncompressed binary bitmap data. It is useful as it is widely supported, has a relatively simple format to manipulate manually, and is a lossless format. (O’Reilly)

4.1.2.3 Lua

Lua is a high-performance scripting language used in the Frozenbyte's in-house engine that is one of the most widely utilized scripting language in games. Lua is also easily usable on various platforms and programming languages, making the language very versatile. It is also free for most purposes. (PUC-Rio, 2015)

Using scripts in a program enables changing the program's functionality without compiling any code, which can be strenuous and lengthy when fine-tuning or debugging programs especially in programs with large codebases. By using Lua-scripts, functionality can be swapped on the fly by reloading specific files. (PUC-Rio, 2015)

4.1.2.4 Shaders

The Frozenbyte engine supports High Level Shading Language (HLSL) shaders for most platforms and PlayStation® Shader Language (PSSL) shaders for Playstation 4 as input. These shader formats are responsible for defining the exact way models and meshes are represented in-game, and also power many visual effects. The files normally stored in text format can be compiled to binary or byte arrays to make them more compact, harder to accidentally modify, and faster to load. (Microsoft, 2015, Stenson & Ho, 2013).

4.1.3 The Preprocessing System

The most important part of the engine as far the refactoring is concerned, was of course the existing preprocessing system. This would remain relatively unchanged on the higher levels, though it would see some major changes in how functionality is delegated between different objects to make the renewed system more flexible to develop further.

The system is built around modules, which were added to a manager object. The modules are responsible for the actual processing steps, and include the functionality to transform a specific type of source resource file to the preprocessed destination file format. The manager keeps record of the modules, and initiates individual sessions of preprocessing as well as having support for preprocessing tasks to be executed on multiple threads in parallel for added performance.

The system is very straightforward in how it manages the preprocessing and the files. Individual resources are handled per-file, and rather than dealing with intermediate composite objects, they are mostly processed one source file at a time. In case of multiple different source files being dependent of each other, the processing code handles this by setting up dependencies between processing steps. Changes in resource files are detected by inspecting source and destination file timestamps, and changes in processing code are handled by versioning each version of the code.

4.2 Targets for Improvement on the Original System

There were a series of planned improvements with the system that rendered it suboptimal and brought upon the need for its refactoring. These were the biggest reason a rework of the system was requested by Frozenbyte, Inc. A secondary objective was to make the system more intuitive and easier to manage, debug, and maintain with slight alterations to the code structure.

Perhaps the biggest singular problem was that the system was unreliable in cases where some resources were already preprocessed. This would sometimes result on some of the new, unprocessed resources not being properly recognized to be in need of processing. This meant that the preprocessed files for some resources ended up being either missing or outdated, which usually resulted in confusion and usually deletion of all the preprocessed files, and a complete preprocessing of all source files.

In addition to being unreliable, the system was in a way, unpredictable. It was impossible to predict the type, name, and destination path of any processed files before the actual processing was complete. This added to the systems unreliableness, as it was highly likely that should the amount or type of the processed files change, some remains of the old processed files would remain, and potentially cause problems. There was a need for a system that would know the destination files that a source file would generate. By knowing this, the system could erase all previous files before it even began processing the new ones, making sure that there were not any redundant preprocessed files remaining for the source file.

Another noticeable issue was that checking for needed processing required a formidable amount of time (1-2 hours on average) on a high-performing computer, and the actual processing did not begin before the checking was complete. The processing itself was a process roughly similar in duration. It was decided both the processing and the checking for processing tasks should be run in parallel, as this could effectively halve required time for the processing of tasks.

An aspect that was missing from the resource processing system was a way to handle dependencies between different source files. There was no proper way to have the code reprocess texture atlases, for example, should only a single texture in the atlas be modified. Cases like this were implemented in a way that would have been hard to maintain and continue to work upon in the future.

There was also need for an improved resource caching system which would work in the company’s internal network. A system which would make sure that most employees should very rarely have to process resources locally on their computer. Instead, the system would make sure it had all the preprocessed versions of the source files, and the employee’s software would acquire them from the system. There already was a caching system like this in place, though it had its own problems and limitations. The system worked by preprocessing all the resources by itself, and compressing all preprocessed files to considerably large archives, each up to several hundreds of megabytes large. The client’s software down-loaded the archives and extracted them, thus acquiring the files. This could generate a lot of unnecessary network traffic and require time that could be used for other purposes when the client only required a few files in the archive, which was more often than not the case. A smarter hash-based system was envisioned to replace the previous one.

4.3 Work on the Existing System

The actual work done on the resource processing management system was centered on improving the system and adding missing features as per the specifications. The process will be detailed in roughly chronological order.

4.3.1 Planning & Beginning

While it was well known what in the existing processing system required improvement, there were not any concrete plans on how the work was organized and performed. It was clear that the work was to be done by a single employee to avoid conflicting modifications and for the whole code base to be fully understood by a single person in order to allow faster and more accurate alterations to it. Support from other employees would be provided where needed and available.

The targets of improvement were identified from a comment an employee had made, which talked about the need of certain betterments and features. From this, the renewal project was split to tasks, each one corresponding to an aspect in the system to be improved. These tasks were used to record time working on the system, to provide a centralized places to store information regarding the tasks, and track the work’s progress. For the entirety of the tasks, a time estimate of 240 work hours was set.

The work would happen separately from the main code branch in the version control, so that the changes could be safely stored and reverted. It was also to prevent the author’s modifications from interfering with most of the other programming work happening in the company. The changes would be introduced as a singular update for others to use as they became stable and beneficial, except for the resource cache functionality which would be done separately after the other changes else were deployed.

4.3.2 Working With a Large System

It quickly became clear that the undertaking was larger than expected. The main functions were simple to understand and to work with, but the different ways they could be utilized was less so. For instance, processing had to be run to at application startup, and when certain files change, and when the user initiates the process with either of the two ways available to him. These were complimented by different parameters given to the system when the processing occurs, which could alter the way the operation is done. One could have the system only process a certain resource, resources that were managed by a specific module, or resources that were manually collected and designated for processing. Adding the custom logic that was needed for some types of resources, different paths of execution for the preprocessing were numerous.

The work was also a constant process of testing. It was important that when the system would be deployed, it would work reliably. Yet the testing was slow, as it usually meant reprocessing every resource available. This was somewhat countered by having a testing session running at all times, but it was still not realistic to test most possible circumstances the processing would run in when it was deployed.

4.3.3 Deployment of the Processing Management System

After rigorous testing the main system was deployed and in use in the company. This resulted in the need for notable amounts of small even though the work was tested as the ways the deployed software was used were more numerous than one could test. Also, when the system was first set in use, some aspects of the company’s employees’ workflow employees was still slightly unclear for the author which also caused some additional confusion. Overall the deployment was a rather painless procedure.

After deploying and beginning work on the second part of the refactoring project, the resource cache system, the work was occasionally interrupted by bug reports of varying urgency. These were produced by the employees and other systems which were using the previously deployed system, and was to be expected. The changes were large enough to leave some bugs remaining in them and the additional error reporting uncovered less critical existing ones. These could be fixed relatively well alongside developing the cache server, but still slowed progress down.

4.3.4 Resource Cache

Specifications for the cache server were thorough and most of the details were clear before starting work on the system, which was in contrast to the processing system where the task was more iterative than clearly set. It was to be tied to the editor found in the company’s game engine so that it would not have to process any resources locally if they were found from the cache.

The system would map processed destination files for every unique combination of source file and its processing parameters, where processed files could be retrieved. The parameters included information like processing code version, target platform. This effectively made sure that every unique destination file corresponded to a single source file. The preprocessed data could thus be retrieved with information of the source file and the parameters.

The server itself was a simple separate application programmed using some built-in functionality from the company's engine. It would listen for incoming connections on the TCP protocol, and open additional asynchronously handled TCP connections for each connected client. The clients would query the server for files corresponding to certain identifiers. It was also possible to check if files for identifiers exists on the server.

Deploying the resource cache for use in the company's editor software resulted in noticeably less errors and problems than the resource processing system. This was due to many different factors, the most notable one being the more extensive testing performed on the system whilst dealing with problems and errors that appeared in the processing system during the time. Also the system and its role in the company's environment had become more clear when working with the cache. The full-time work on the changes ended a week after deployment. After this, the work will be mostly related to maintaining the system, and providing support in its use.

5 SUMMARY

The result of the study on resource management systems performed is not of the nature to give direct answers, but rather directions and facts to keep in mind when working with different kinds of external data. It is also clear that the practical application of this information in the systems of Frozenbyte, Inc., while being valuable experience, is only a single take on the possible implementation methods while numerous reasonably valid alternatives exist..

5.1 Effective Resource Management

It is clear that managing resources has to conform to the game engine or system's specifications and nuances. Hence there is no one system design that suits all, but there are some tasks that most systems need to be able to perform, and techniques that benefits. Of course, there are also other aspects to consider, such as the amount of available work hours. Not all game engines require or are able to have a complex systems just for managing resources, but can and should opt for more simple contraptions.

Most game engines attempting to support console platforms should archive their resources when the game is shipped for sale, for instance. This is clearly beneficial to the game's load times, and if the games made with the engine contain notable amounts of resources, is practically a must for the engine to be ideal for use. Streaming content, while a worthy addition to a resource management system, is not necessary by itself. This is the case for asynchronous loading from disk as well. While it would be beneficial to have all of these features, it is simply not useful to implement them with no reason, as all game engines do not benefit from these features. Thus the need for the features should be evaluated by examining the typical genre, scale, and platforms of the games made with the game engine.

5.2 Resource Processing System’s Development

The system that was worked on is now working more reliable and efficient to use, even though there may still be some minor problems with it. Most of these will be fixed as they come. The resource cache system is functioning as desired and is being actively used at the time of writing. Considering these facts it can be said that the result of the development is as planned and expected.

While the result is a success, there are a number of aspects that could have been done more efficiently. For one, creating the Linux build of the server software for the resource cache consumed two weeks almost by itself. This was mostly due to the author’s relative inexperience with makefiles and Linux in general. Also, some of the more serious bugs would also have been easily avoidable with some failsafe logic behind them. These can easily be interpreted as learning experience, though.

5.3 Evaluation & Improvement

The knowledge that has been assembled in this thesis is proven and sound, and is useful as a source of information for developers with little to no experience with resource management. It is good to remember though, that while the information presented has a valid source, the ones who have provided the source have their own perspective on the matter which usually correlates with the writer’s experience on working in specific environments. This means that there usually is a certain viewpoint or a composite of multiple viewpoints behind much of the information that is very likely to be similar to the system or systems the person behind the knowledge has worked on.

To gather more objective information, one would have to have solely worked on resource management for an unusually sizeable amount of time which is not usual. The alternative would be actually use and analyze a large amount of these systems, which is not simple as most game engines and especially their source code are proprietary, or at least not readily available for study.

The work done on the resource processing system can be thought of as a success. The specified features were implemented and the flaws worked on successfully, and no new major flaws have not been found by neither the users nor the systems. The work itself lasted roughly twice the amount of time originally expected, though. This is mostly because of the nature of the system. There are many different instructions that can be given to it, and many resource types it handles. This made testing the system quite lengthy, and even when tested the system had bugs in its programming that did not appear until specific conditions were met, which required allocating time to fix the problems.

REFERENCES

Anderson(22.09.2015) File Format Overview http://www.dpbestflow.org/file-format/file-format-overview (Read 2015)

Bensmiley(05.04.2012) What is a game engine? http://www.deluge.co/?q=what-is-a-game-engine (Read 2015)

Ben Carter(21.02.2005). Book Excerpt: The Game Asset Pipeline: Managing Asset Processing() http://www.gamasutra.com/view/feature/2203/book_excerpt_the_game_asset_.php(Read 2015)

Ben Davis(08.02.2009) What are game assets? http://conceptdevelopmentbendavis.blogspot.be/2009/02/what-are-game-assets.html (Read 2015)

Blender Foundation(10.08.2013) FBX binary file format specification https://code.blender.org/2013/08/fbx-binary-file-format-specification/(Read 2015)

CareerRide.com(2015) XML Serialization http://careerride.com/XML-Serialization.aspx (Read 2015)

Carter(2004) The Game Asset Pipeline. Hingham: Charles River Media Inc.

CppReference.com(30.12.2014). Preprocessor http://en.cppreference.com/w/cpp/preprocessor(Read 2015)

Digital Tutors(2013) Demystifying Game Development Terms - Your Guide to Understanding Industry Terms http://blog.digitaltutors.com/demystifying-game-development-terms-your-guide-to-understanding-industry-terms-and-working-like-a-pro/ (Read 2015)

Reto.Injection(14.10.2013) Developer’s Corner: Asset Streaming http://www.heroesandgenerals.com/community/15598/developers-corner-asset-streaming (Read 2015)

Epic Games, Inc(2015). Unreal Engine 4 Documentation. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StaticLights/index.html (Read 2015)

Gregory (2015) Game Engine Architecture, Second Edition. Boca Raton: CRC Press Taylor & Francis Group

KaiFB(21.04.2015). Trine 3: The Artifacts of Power Out Now on Steam Early Access! http://www.frozenbyte.com/2015/04/trine-3-the-artifacts-of-power-out-now-on-steam-early-access/ (Read 2015)

L. H. de Figueiredo & R. Ierusalimschy & W. Celes(2015). LUAC – Lua compiler http://www.lua.org/manual/5.1/luac.html (Read 2015)

Lönn(2015) Streaming for Next-Generation Games http://www.gamasutra.com/view/feature/1769/streaming_for_next_generation_games.php

Microsoft(2015) Compiling Shaders https://msdn.microsoft.com/en-us/library/windows/desktop/bb509633(v=vs.85).aspx (Read 2015)

Noel(2004) Optimizing the Content Pipeline http://gamesfromwithin.com/optimizing-the-content-pipeline (Read 2015)

O'Connor(25.09.2014). Conjure Up New Levels With Trine & Trine 2's Editor

http://www.rockpapershotgun.com/2014/09/25/conjure-up-new-levels-with-trine-trine-2s-editor/ (Read 2015)

O'Reilly(Unknown). TGA File Format Summary http://www.fileformat.info/format/tga/egff.htm (Read 2015)

PUC-Rio (17.07.2015) Lua: about http://www.lua.org/about.html(Read 2015)

Rouse (06.2015a) What is lossless and lossy compression http://whatis.techtarget.com/definition/lossless-and-lossy-compression (Read 2015)

Rouse (03.2015b) What is Compression? http://searchstorage.tech-target.com/definition/compression (Read 2015)

Stenson & Ho (2013) PlayStation® Shader Language for PlayStation®4 http://twvideo01.ubm-us.net/o1/vault/gdceurope2013/Presenta-tions/825424RichardStenson.pdf (Read 2015)

Thorn (2011) Game engine Design and Implementation. Sudbury: Jones &Bartlett Learning

University of Maryland(2003) Ascii vs. Binary Files http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/asciiBin.html