The Video Game Asset Pipeline
A Pattern Approach to Visualization

James Lear
Student ID: 91002002
james.lear@uwe.ac.uk

The University of the West of England
Faculty of Environment and Technology

This thesis is submitted for the degree of
Doctor of Philosophy
March 2021

Director of Studies
Professor Richard McClatchey
richard.mcclatchey@uwe.ac.uk

Supervisor
Dr Simon Scarle
simon.scarle@uwe.ac.uk

Abstract

Video games consist of virtual worlds modelled as an approximation of either a real or imaginary environment. The amount of content required to populate the environments for Triple-A (AAA) video games doubles every few years to satisfy the expectations of the end-users. For this reason, the art and design discipline now constitute the majority of those employed in a video game studio. The artists use Digital Content Creation (DCC) tools to design and create their content; tools not originally designed for video game asset creation. Ultimately the artists require to preview their content in the form of source assets in the runtime environment, the game engine, to ensure they provide an accurate rendering of their original vision. However, there exists a barrier to achieve this workflow; the original source assets are persisted in a proprietary format, information rich to handle future edits, and the final runtime environment requires the assets to be lightweight ready for fast and efficient loading into the game engine.

The video game industry has solved this problem by introducing a fast and efficient workflow known as the asset pipeline. The asset pipeline is recognized within video games technology as a general reusable solution to the common problem of converting source assets into their final runtime form as expected by the runtime game engine. Although the asset pipeline defines a series of stages that all content must follow from inception to their final realization a single solution does not exist to satisfy all projects.

Likewise, within the discourse of patterns, a pattern is defined as a general reusable solution to a problem operating under a certain context. Originating in the field of architecture (Alexander, 1979) patterns have now been discovered and mined in numerous domains including software engineering (Gamma *et al.*, 1995). Within the field of software engineering patterns exists at several levels of abstraction including architectural, design-level and low-level idioms. The world of video games technology and patterns intersects in the form of one set of patterns identified by Nystrom (2014), although these are very much low-level idioms and certainly do not encompass the challenge of the asset pipeline as found in video game production.

This research addresses this shortfall and formalizes the asset pipeline into a catalogue of patterns for use both within the video game industry and to satisfy the wider audience of interactive real-time visualization in general. Interactive real-time visualizations consist of both the navigation and viewing application, executing in a runtime environment, and the digital content providing the data source for the visualization. Their workflow production draws parallels with that of the video game industry. The designers of such visualizations use the iterative process of create, review and modify. Creation of the source asset within the DCC tool, preview within the visualization runtime and subsequent modification in the DCC tool.

However, the video game industry is tempered by a number of problems hindering proliferation of the asset pipeline as a general reusable solution. The video game industry is shrouded in secrecy preventing the natural dissemination of information. Software developers operating within the industry are subject to Non-Disclosure Agreements (NDAs) protecting intellectual property invested in software tools such as the asset pipeline. The video game industry is relatively young, being fifty years old, and as such a set of agreed-upon terms and their definition has been slow to develop. This is compounded by the asset pipeline technology operating at the fault line of two disciplines: the engineers developing the runtime and the artists creating digital content. In such an interdisciplinary field language barriers exist. The characteristics and properties of patterns address these problems. They identify, name and provide a common vocabulary for specific problem-solution abstractions.

They capture expertise and make knowledge accessible to non-experts. The communication of which enables domain independent solutions.

This novel research formalizes the asset pipeline into a catalogue of patterns consisting of an architectural pattern named the ASSET PIPELINE and the component patterns of DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS. Under the iterative spiral model methodology aligned to the framework of the Pattern Languages of Programs (PLoP) workflow. This involves the unique approach of shepherding, pattern mining, a writers' workshop and pattern writing. All of which culminated in the publication of the pattern catalogue in the Association for Computing Machinery (ACM) for wider consumption and use in further domains of visualization.

The asset pipeline catalogue was instantiated and applied in two domains: architectural visualization (ArchViz) and graph visualization (GraphViz) under the process of sequential application of the pattern components. This resulted in real-time exploratory visualizations that not only validate the pattern application but serve for wider research opportunities in the future. Such opportunities include expansion of the pattern language, refinement of the instantiated visualizations, development of a software framework and further pattern mining in other avenues of games technology.

Acknowledgements

I would like to express my sincere gratitude to my supervisory team consisting of Professor Richard McClatchey and Dr Simon Scarle. Professor Richard McClatchey for his continual support and assistance navigating every stage of the doctoral journey; an extremely difficult and winding path to navigate. Nevertheless, a fascinating and interesting learning experience. Dr Simon Scarle for his extensive technical guidance related to the field of video games technology. Chris Simons provided valuable review feedback during progress milestone stages of the PhD journey.

I would like to thank the University of the West of England for funding both the Graduate Tutor Scheme and the European Conference on Pattern Languages of Programs 2019. The conference, its extensive year long process including shepherding and writers' workshop, has undoubtedly afforded me a deep understanding of the discourse of patterns.

I would like to attribute my love for patterns and everything pattern related to my time working as a Senior Software Engineer at Oracle Corporation UK Ltd in their Business Intelligence group. Oracle Corporation Ltd introduced me to the "Gang of Four" and I have been a pattern advocate ever since. Their professional grounding in true software engineering will undoubtedly last me a lifetime.

The greatest of thanks go to my parents whose continual and undying support has enabled me to get through this journey. Without my father I would not have the tenacity to persevere and keep going. Without my mother I would not see the world through the wonderful lens of creativity. Both are essential on my journey through life.

Ultimately, I would like to dedicate this piece of work to my daughters, Willow and Saffron. I wrote every word for them so that they can understand tenacity overcomes everything, no matter of the struggles they have been through in life, and that in the end perseverance and faith prevails.

Table of Contents

1 Introduction 1

1.1 Background and Context 1

1.2 Research Motivation 7

1.3 Research Problem Area 11

1.4 Research Aims and Objectives 13

1.5 Research Hypothesis and Associated Questions 14

1.6 Contribution to Knowledge 15

1.7 Methodology 16

1.8 Thesis Structure 18

1.9 Publications 19

1.10 Summary 19

2 Literature Review 21

2.1 Asset Pipeline 21
2.1.1 Source Assets 26
2.1.2 Extracting Source Assets 27
2.1.3 Pipeline Asset Format 29
2.1.4 Asset Build Process 30
2.1.5 Asset Dependencies 33
2.1.6 Scenario: Game Engine Asset Pipeline 34
2.1.7 Scenario: XNA Asset Pipeline 37

2.2 Idioms, Patterns and Pattern Languages 40
2.2.1 Pattern Definition 40
2.2.2 Pattern Properties 42
2.2.3 Pattern Form 46
2.2.4 Pattern Classification 51
2.2.5 Pattern Writing 52
2.2.6 Games Technology Patterns 54

2.3 Summary and Evaluation 58

3 Asset Pipeline Pattern Language 61

3.1 Introduction 61

3.2 Methodology 61

3.3 Outcomes 62
3.3.1 Pattern Format 62
3.3.2 Pattern Mining – Cycle 1 65
3.3.3 Pattern Writing – Cycle 1 66
3.3.4 Writers' Workshop 68
3.3.5 Pattern Mining – Cycle 2 69
3.3.6 Pattern Writing – Cycle 2 70

3.4 Summary 88

4 Use Case – Architectural Visualization 89

4.1 Existing Architectural Visualization 89

4.2 Architectural Visualization Workflow 90

4.3 Architectural Visualization Workflow Problems 91

4.4 Integration of Asset Pipeline and Architectural Visualization Workflow 92

4.5 Architectural Models and Building Information Modeling (BIM) 92

4.6 Use Case Prototype 93

4.7 ASSET PIPELINE 94

4.8 DCC EXPORT 96
4.9 INTERMEDIATE FILE FORMAT 97
4.10 ASSET BUILD PROCESS 98
4.11 Summary and Evaluation 103
5 Use Case – Graph Visualization 105
5.1 Data Acquisition 106
5.2 Graph Layout 107
5.3 Use Case Prototype 109
5.4 ASSET PIPELINE 109
5.5 DCC EXPORT 111
5.6 INTERMEDIATE FILE FORMAT 113
5.7 ASSET BUILD PROCESS 115
5.8 Summary and Evaluation 116
6 Conclusions and Future Work 118
6.1 Research Questions Revisited 118
6.2 Contributions to Knowledge 125
6.3 Research Limitations 126
6.4 Future Directions 128
6.5 Conclusion 131
7 Bibliography 134
Appendix 1 – Video Game Development 140
Appendix 2 – Games Technology Surveys 143
Appendix 3 – Cycle 1 Shepherding Process 145
Iteration 1 – Feedback and Resolution 145
Iteration 2 – Feedback and Resolution 147
Iteration 3 – Feedback and Resolution 149
Appendix 4 – Cycle 1 Asset Pipeline Patterns 150
ASSET PIPELINE 150
INTERMEDIATE ASSET FORMAT 162

List of Figures

Figure 1.1: Basic design cycle methodology (Koomen, 1991) 2
Figure 1.2: The asset pipeline as seen in video game production 3
Figure 1.3: Venn diagram highlighting the research study problem areas.......... 11
Figure 1.4: Spiral model methodology (Boehm, 1988) 16
Figure 1.5: Pattern language formulation workflow 17
Figure 1.6: Thesis structure in relation to research questions 18
Figure 2.1: Game engine architecture and runtime components.......... 23
Figure 2.2: Single stage asset pipeline 25
Figure 2.3: Multi-stage asset pipeline 25
Figure 2.4: Autodesk 3ds Max proprietary format.......... 26
Figure 2.5: Custom export plug-in extracting source assets from the DCC tool 27
Figure 2.6: The asset build process 31
Figure 2.7: Unity asset pipeline – exporting a 3D model from a DCC tool into a game engine for real-time visualization.......... 34
Figure 2.8: Unity asset import settings categorized into scene, meshes and geometry 35
Figure 2.9: XNA asset pipeline stages 37
Figure 2.10: Pattern writing process according to Wellhausen and Fießer (2011) 53
Figure 2.11: Asset pipeline workflow solution schema 59
Figure 3.1: Pattern language formulation workflow 62
Figure 3.2: Asset Pipeline Pattern Schema 64
Figure 3.3: Asset pipeline workflow 65
Figure 3.4: Writers' workshop discussing the proposed asset pipeline patterns.......... 68
Figure 3.5: ASSET PIPELINE workflow pattern integrating the three component patterns 69
Figure 3.6: ASSET PIPELINE architectural pattern diagram.......... 75
Figure 3.7: Design-review process.......... 76
Figure 3.8: Block diagram representing the stages of the asset pipeline 76
Figure 3.9: DCC EXPORT component pattern diagram.......... 78
Figure 3.10: Creation of a DCC export plugin.......... 79
Figure 3.11: ASSET BUILD PROCESS component pattern diagram.......... 81
Figure 3.12: Asset build process – Processing of high density meshes into lower density meshes 81
Figure 3.13: Asset build process 82
Figure 3.14: Dependencies of an asset build node.......... 83
Figure 3.15: INTERMEDIATE FILE FORMAT component pattern diagram 85
Figure 3.16: Asset build nodes performing algorithmically identical processing on different asset formats...... 86
Figure 3.17: Use of intermediate file format within the asset build process.......... 86
Figure 3.18: DCC Export provides conversion.......... 87
Figure 3.19: Conversion within the first stage of asset build process.......... 87
Figure 4.1: An example of a still render – the Faculty of Business and Law Building (UWE) 90
Figure 4.2: The traditional workflow for the production of a still render.......... 90
Figure 4.3: Architecture and the Built Environment Building Information Model 93

Figure 4.6: DCC EXPORT component pattern diagram.. 96
Figure 4.7: INTERMEDIATE FILE FORMAT component pattern diagram 97
Figure 4.8: ASSET BUILD PROCESS component pattern diagram.. 98
Figure 4.9: Single stage asset pipeline 99
Figure 4.10: Intermediate processing of materials.. 100
Figure 4.11: Configuration of object names.. 100
Figure 4.12: BIM semantic properties.. 101
Figure 4.13: BIM class hierarchy.. 101
Figure 4.14: BIM Visualization ASSET PIPELINE.. 103
Figure 4.15: BIM Visualization generated from ASSET PIPELINE instantiation.. 104
Figure 5.1: Netlytic (2020) data acquisition using the hashtag #COVID search query.. 106
Figure 5.2: Netlytic (2020) table of tweets containing #COVID displayed in relational format.. 106
Figure 5.3: Conceptual model of Twitter semantics 107
Figure 5.4: Netlytic (2020) network analysis.. 107
Figure 5.5: Gephi - Initial graph visualization using Force Atlas 3D. 108
Figure 5.7: Graph Visualization (GraphViz) Asset Pipeline.. 110
Figure 5.8: DCC EXPORT component pattern diagram.. 111
Figure 5.9: INTERMEDIATE FILE FORMAT component pattern diagram 113
Figure 5.10: Exported Gephi graph format GEXF.. 114
Figure 5.11: ASSET BUILD PROCESS component pattern diagram.. 115
Figure 5.12: Graph Visualization (GraphViz) ASSET PIPELINE.. 116
Figure 5.13: Graph Visualization (GraphViz) implemented in Unity. 117
Figure 6.1: ASSET PIPELINE architectural and component patterns.. 123
Figure 6.4: Alternative fast-path 128
Figure 8.1: The video game industry value network.. 140
Figure 8.2: The roles of the video game developer.. 141
Figure 11.1: Digital Content Creation and Interactive Real-Time Visualization 150
Figure 11.2: Design Feedback Loop 151
Figure 11.3: Custom export plug-in transforming internal source representation to a final asset.. 152
Figure 11.4: Simplified scene graph and dependency hierarchy of an architectural visualization.. 153
Figure 11.5: Block diagram representing the three stages of the Asset Pipeline.. 154
Figure 11.6: Asset Build Process 155
Figure 11.7: Dependencies of an asset build node.. 155
Figure 11.8: Asset Build Process – Processing of high density meshes into lower density meshes.. 156
Figure 11.9: Unity Asset Pipeline – exporting a 3D model from a DCC tool into a game engine for real-time visualization.. 158
Figure 11.10: Unity asset import settings categorized into scene, meshes and geometry 158
Figure 11.11: An example of a still render – the Faculty of Business and Law Building (UWE) 160
Figure 11.12: ArchViz Asset Pipeline – exporting a BIM model for real-time visualization in a game engine 161
Figure 11.13: Asset Build Nodes performing algorithmically identical processing on different asset formats 162
Figure 11.14: Use of Intermediate File Format within the Asset Build Process.. 163
Figure 11.15: Conversion of Source Asset format to Intermediate Asset Format within an Exporter Plug-in . 164

Figure 11.16: Conversion performed within the build asset node 164
Figure 11.17: Video game asset pipeline illustrating use of Intermediate Asset Format.................................. 166
Figure 11.18: Integration of the Fast Path in the Asset Pipeline.. 168

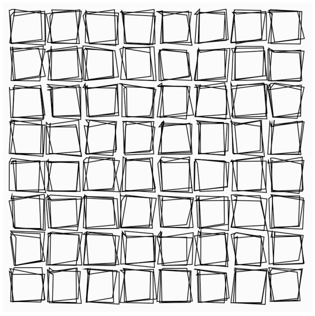
List of Tables

Table 2.1: Common technologies exhibited in a game engine 24
Table 2.2: Unity asset import settings and functionality 36
Table 2.3: XNA built-in content importers 38
Table 2.4: XNA built-in content processors 39
Table 2.5: Alexandrian pattern form 47
Table 2.6: Pattern template format as identified by Gamma *et al.* (1995, pp.16-18) 48
Table 2.7: Coplien pattern form 49
Table 2.8: Pattern Languages of Program Design pattern form 50
Table 2.10: The Game Programming Patterns as identified by Nystrom (2014) 56
Table 2.11: Pattern format as identified by Nystrom (2014) 57
Table 3.1: Pattern form comparison 63
Table 3.2: Overview of the pattern format 64
Table 3.2: Summary of the Asset Pipeline pattern 67
Table 3.3: Summary of the Intermediate Asset Format pattern 67
Table 3.4: Asset Pipeline pattern language summarizing category and intent 70
Table 3.5: ASSET PIPELINE pattern overview 71
Table 3.6: DCC EXPORT pattern overview 72
Table 3.7: ASSET BUILD PROCESS pattern overview 73
Table 3.8: INTERMEDIATE FILE FORMAT pattern overview 74
Table 4.1: Autodesk Revit export file formats. 97
Table 5.1: Graph visualization tool survey 108
Table 5.2: Gephi export file format 112
Table 5.3: Netlytic tweet format 114
Table 9.1: Game engine survey updated and extending survey of Andrade (2015) 143
Table 9.2: Survey of Digital Content Creation (DCC) tools 144
Table 11.1: Example DCC tools used in the production of an architectural visualization 151
Table 11.2: Unity asset import settings and functionality 159

Glossary

Term	*Definition*
AEC	Architecture, Engineering and Construction (AEC) is the collective term for the three associated industries.
ArchViz	Architectural Visualization (ArchViz) refers to the creation of three-dimensional computer models of architectural structure and their visualization in either static or dynamic form.
Asset Build Node	A single processing step within the asset build process performing a transformation of one or more input assets into one or more output assets. Typically, the asset build nodes can be categorized into three distinct flavours: convertors, calculators and packers.
Asset Build Process	The asset build process is the component of the asset pipeline tasked with transforming the source assets into their final realized form as expected by the game engine runtime.
Asset Pipeline	The asset pipeline is the path that all source assets follow, from their initial conception until they are loaded into the games engine (Llopis, 2004).
BIM	Building Information Modeling (BIM) is used within the Architecture, Engineering and Construction (AEC) industry to incorporate both the three-dimensional representation of building components and the data that describes how they behave.
CAD	Computer Aided Design (CAD) is the use of computers and associated software in the creation, modification and review of creative design.
CGI	Computer Generated Imagery (CGI) is the application of computer graphics in the construction of images in the domains of art, print media, video game industry, television and film industry. Typically, the term CGI refers to the creation of three-dimensional digital content.
COLLADA	The COLLADA (COLLABAborative Design Activity) file format was developed as a joint collaboration between Sony Computer Entertainment and the Khronos Group (Lengyel, 2010). The aim being to provide a standard intermediate representation for use by Digital Content Creation (DCC) tool vendors.
DCC Tool	A Digital Content Creation (DCC) tool is a category of application that is used to design and construct creative content. For example, Autodesk 3ds Max (2016a) and Autodesk Maya (2016b) are used to perform three-dimensional mesh modelling.
Game Engine	A game engine consists of "software that is extensible and can be used as the foundation for many different games without major modification." (Gregory 2014, p.ii). The game engine contains the following abstracted functional services: graphics renderer, collision and physics, audio, online multiplayer networking, artificial intelligence (AI) and front end (Bogdanowicz *et al.*, 2010).
Intermediate File Format	A common file format used by all stages in the asset build process. The intermediate file format decouples the Digital Content Creation (DCC) proprietary file format from the final runtime game engine.
SDK	A Software Development Kit (SDK) is a collection of tools, services, components and code fragments created for a specific purpose. For example, the Software Development Kit (SDK) of a Digital Content Creation tool such as Autodesk Maya (2016b) provides a set of interfaces and services to query the internal representations of the scene data.
SNA	Social Network Analysis (SNA) is a tool for modelling, visualizing and exploring the interactions between individuals within a group or organization.
Source Assets	Source assets include everything that is not the game algorithm or code: three-dimensional models, textures, materials, audio files, animations, cinematics and scripts to name a few (Llopis, 2004).

1 Introduction


Generative Pattern 1 – J Lear 2020

In this chapter, initially the background information related to the research problem area of the asset pipeline as found in the video game industry and the philosophy of design patterns is presented and located within existing work. The motivation of the research in the field of video game production is stated, primarily secrecy within the video game industry and a lack of associated terminology, technical resources and knowledge exchange.

The research focus leads into the aims and objectives of the research. That is to formulate the asset pipeline into a pattern language and to evaluate whether such an approach can assist both artists and designers with the creation of interactive visualizations. The research hypothesis with associated research questions and the methodology that will be used to evaluate the hypothesis in the thesis is also presented. The contribution to knowledge in the interdisciplinary research areas of video game production and the ethos of patterns is explicitly stated. Finally, at the end of this chapter the structure of the thesis is outlined.

1.1 Background and Context

During the coronavirus (COVID-19) pandemic eleven million people embraced and played the video game "Animal Crossing" whilst in isolation. During the two weeks from its initial release on the 20th March 2020 until the end of that month it sold 11 million units (Thier, 2020). The video game offered a welcome distraction from a world in isolation; providing a simulation of a virtual world operating in real-time facilitating communication to friends and family no matter of their distance apart. The presence of video games in media consumption was pervasive during the coronavirus pandemic. Although seclusion undoubtedly led millions of people to turn to video games for entertainment and safe social interaction the trend was not one-off by any means.

Computer games began in 1961 with "Spacewar" developed by MIT student Steve Russell. It played a diagnostic function and was used to demonstrate the ability and accessibility of the DEC PDP-1 mainframe computer (Burnham, 2003; Kent, 2001; Wise, 2000). The barrier of entry for consumers then was the prohibitive cost of computers (O'Donnell, 2014), at the time Russell said "we thought about trying to make money from it for two or three days but concluded that there wasn't a way that it could be done" (Kent, 2001). This seems extraordinary now that the games industry is an established part of the global entertainment economy (Marchand and Hennig-Thurau, 2013). Today more Americans play video games than go to the movies (Graham, 2009). In 2018, the US video game market revenue was $24.4 billion, more than double that of its film industry (Ballhaus, 2018). With an annual inflation rate near 15% over a 25-year period, the growth velocity is indisputably high.

To gain competitive advantage in this arena video game production companies invest heavily in the underlying technology, both hardware and software, used in the production of video games (Perron and Wolf, 2009). A proportion of this investment has led to the development of software with a requirement to optimize and increase the efficiency of the creative workflow involved in the video game production process. Indeed, the primary focus of this research is to identify, formalize and evaluate one such workflow used to assist the video game artists and designers during the pre-production and production phases of video game development.

Video game development has seen a dramatic shift in recent years. The amount of digital storage space available for developers to utilize has risen and subsequently the expectations of both the end user and publisher in terms of more immersive and detailed environments has risen as well. The original video game named "Doom" released in 1993 occupied a physical storage file size of merely 24MB in comparison to the reboot of the franchise released in 2016 with a storage requirement of 77GB. Since "the content is king" (Gregory, 2014) in a data-driven game engine architecture the art discipline increasingly composes the majority of those working in games studios. In video game production game content originates from game assets, and as Llopis (2004, p.36) defines, "game assets include everything that is not code: models, textures, materials, sounds, animations, cinematics, scripts, etc." The originating source assets should contain all the information necessary to build the game content. It should be possible to delete all the assets except for the source assets and still be able to rebuild the game content (Lengyel, 2010).

Artists and designers are not simply working with the suite of development tools that provide direct entry point into the body of the game engine; they are also working with a suite of external Digital Content Creation (DCC) tools. Two-dimensional image manipulation software such as Adobe Photoshop is used by artists to create textures and materials. Three-dimensional modelling software such as Autodesk 3ds Max (2016a), Autodesk Maya (2016b) and Blender (2020) is used to create three-dimensional environment models to be placed in the runtime setting. Three-dimensional animation software such as Autodesk 3ds Max (2016a) and Autodesk Maya (2016b) involves the application of keyframing or motion capture data to the environment models. Sculpting tools such as Pixologic ZBrush (2020) enable a character artist to organically sculpt very high levels of detail into the three-dimensional models (Charrieras and Ivanova, 2016). The digital content creation packages were not designed for the task of creating game assets (Blow, 2004) and offer very little in the way of interoperability support (Perron & Wolf, 2009); the source assets are heterogenous in format and diverse in character.

The Focus: The Asset Pipeline Workflow

The creative process of designing a video game asset is iterative in nature and follows the basic design cycle methodology; that of analysis, synthesis, simulation and evaluation (Koomen, 1991). Although the workflow illustrated in figure 1.1 can be achieved largely within a DCC tool, ultimately the asset must be integrated within the game for visual inspection and review under runtime conditions. For an artist this workflow validates the three-dimensional model renders as intended in the game engine scene. O'Donnell (2014, p.75) emphasizes the "real-time previews of game's content inside of a game's rendering engine is frequently cited as essential by artists."

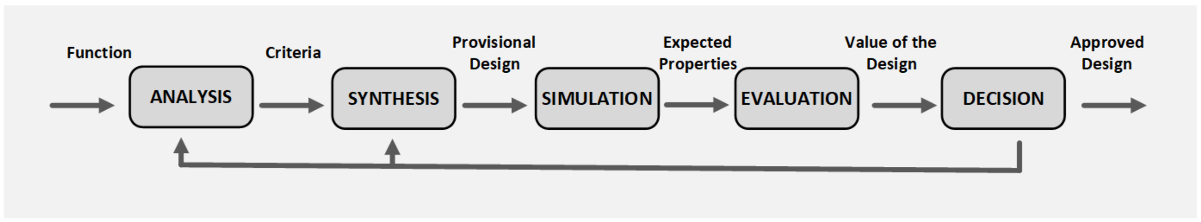


Figure 1.1: Basic design cycle methodology (Koomen, 1991)

However, a barrier exists preventing the artist from achieving this goal in an efficient manner. The requirements of the game asset designed within the DCC tool differ from those expected in the runtime game engine. The source asset is often held in a proprietary binary format of the DCC tool and is information rich allowing for future edits and modification by the artist (Lengyel, 2010). Whereas, the runtime final game asset is lightweight; optimized for real-time manipulation within the game engine runtime as specified by the software engineer.

To overcome this challenge the computer games industry has produced a workflow to streamline asset production known as the asset pipeline, figure 1.2. Llopis provides the following definition (2004, p.36), "[t]he asset pipeline is the path that all the game assets follow, from conception until they can be loaded in the game." Llopis (2004, p.36) continues to assert, "[the] goal of a well-designed asset pipeline is to act as its name suggests: you put the source data into one end of the pipe, and it comes out of the other end in a form the game can use, hopefully as fast and with as little human intervention as possible."

Source assets must be compiled and optimized for each target runtime. Texture format must be stored in a size and format for immediate hardware access by the target Graphics Processing Unit (GPU). Specifically, geometry data must be compiled for each target in terms of triangles, number of triangles per batch and the sorting order of the vertices. Final assets must be optimized for game loading speed (Lengyel, 2010). This represents a data driven approach - a significant shift with how developers approach video game development. The asset pipeline has become extremely important in this transformation since it is the path through which all source asset data must flow to become a component of the video game. The asset pipeline is an intricate mechanism that interfaces the artists to the game engine (Lengyel, 2010).

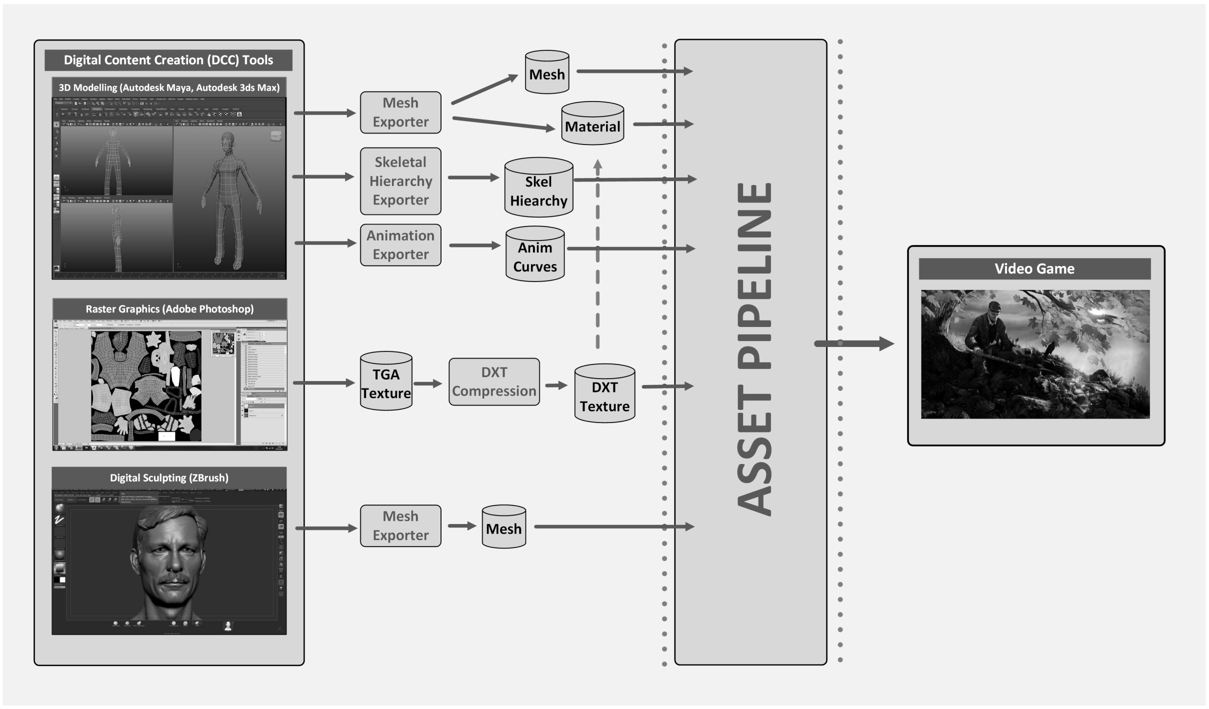

Digital Content Creation (DCC) Tools
3D Modelling (Autodesk Maya, Autodesk 3ds Max)
Raster Graphics (Adobe Photoshop)
Digital Sculpting (ZBrush)
Mesh Exporter
Skeletal Hierarchy Exporter
Animation Exporter
TGA Texture
DXT Compression
DXT Texture
Mesh Exporter
Mesh
Material
Skel Hiearchy
Anim Curves
Mesh
ASSET PIPELINE
Video Game

Figure 1.2: The asset pipeline as seen in video game production

The infrastructure specialist of the asset pipeline is known as the technical artist or the tools engineer. The technical artist forms a mediator between software engineer and artist. Since the software engineer and artist focus on both different languages and different aspects of the development process they evolve from different epistemic communities (Knorr-Cetina, 1999). The software engineers interact and attempt to optimize the algorithms of the game engine within the confines of the target platform. Subsequently, the artists adjust and optimize their three-dimensional models in ways to interoperate within the rigid confines defined by the software engineers. The technical artists emerge at the interface between the discipline of the artist and software engineer; they emerge at the disciplinary fault line (Gould, 2003). Anthropologists of science and technology have demonstrated that it is frequently at the fault lines of disciplines where most of the interesting and critically important outcomes occur (Gould, 2003). Operating within the confines of the fault line the technical artist often finds themselves at points of extreme friction; the resolution of which form the technical requirements for the asset pipeline. Technical artists develop DCC tool exporters, optimizers, compilers, user interface tools and other elements (Ansari, 2011) that facilitate a smooth transition from the DCC tool of the artist into the game engine to satisfy the design evaluation cycle (Koomen, 1991). The function of which replaces the communication and understanding between artist and software engineer with the technological system of the asset pipeline. Integrating technological systems at fault lines makes them manageable in a different way. "Good, usable systems disappear almost by definition" when they facilitate communication gaps or minimize them (Bowker and Star, 1999). For the artist and designer, the asset pipeline disappears into the background. The result of which has allowed the artist to concentrate on their creative process rather than the technicalities of converting and optimizing their design for ultimately visualization.

It is during the early phases of preproduction where the software engineer, artist and designer explore the fine interface between work and play. The technical artist resolves the discursive tensions that emerge between the non-visual engineering aesthetic of the game engine representing work and the decidedly visual properties of the game assets representing play. The two worlds contradict: first the orientation towards efficiency and performance that defines how software engineers and game engines develop, and second the creative and expressive openness of the collaborative design process, subjectivity and play. The tools, software systems and workflows that form the asset pipeline enable the kind of experimental systems critical for video game development, especially in the early stages of production. As the development process moves from the pre-production to the production phases the asset pipeline rapidly recedes into the background and the underlying technology quickly becomes an integral part of working life.

The asset pipeline is one aspect of video game development least studied, researched and communicated by those outside of the games industry and is an area only the experienced technical artist and tools engineer within the industry has demanded their focus (O'Donnell, 2014). The reasons for which are explored in the motivation for the research detailed in section 1.2.

During the course of video game production, the asset pipeline designed and implemented by the technical artist forms an essential and efficient workflow. It is a workflow utilized by the artist and designer to evaluate their realized design in a runtime game environment with the minimum of effort. Although the pipeline is a general reusable solution to the commonly occurring problem of converting a large number of source assets into their final runtime assets there will be elements of the pipeline that are used for one project, but not another. Ultimately the specifics of the asset pipeline are adapted to the given requirements of the game content. The pipeline is simultaneously an entity sui generis and motley (O'Donnell, 2014).

Similarly, within the context of architecture the architect and author Christopher Alexander established a pattern language to describe the abstract problems and solutions relating to the design and construction of towns and buildings. In his seminal architectural design book, "A Pattern Language – Towns, Buildings, Construction", Alexander (1977) defines the individual entities of the language as patterns. Alexander states, "[e]ach pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, Ishikawa and Silverstein, 1977).

Twenty years after Alexander first published his ideas for an architectural pattern language Gamma *et al.* (1995) extended the pattern philosophy into the domain of software engineering, specifically object-oriented design. The book "Design Patterns: Elements of Reusable Object-Oriented Software" was published and specified a catalogue of design patterns that "…describes simple and elegant solutions to specific problems in object-oriented design" (Gamma *et al.*, 1995).

This study claims the asset pipeline is a template solution that provides application within the domain of video game production; that can be extended into further domains with comparable workflows to assist practitioners in their creation.

Although the asset pipeline defines a series of stages that all content must follow from inception to final realization "… there isn't a single solution that can satisfy all projects." (Gregory, 2014). Llopis asserts (2004, p.44), "[g]ames are very different from each other and teams are organized differently so content pipelines will vary significantly from project to project."

The asset pipeline is currently restricted for use within the video game industry and has not been researched and recognized as a pattern language that can be communicated and applied for use by practitioners both within video game production and outside of this context. Such pipelines are proprietary in nature and not formalized for general purpose use.

Comparably other domains follow similar experimental workflows in their production; transformations of large amounts of source data assets into resulting exploratory three-dimensional visualizations. The three-dimensional visualizations apply "the use of computer-supported, interactive, visual representations of data to amplify cognition," where cognition is the acquisition and use of knowledge (Blythe, 2000). In recognition the human visual system (the eye and visual cortex of the brain) constitutes an effective parallel processor that supports the maximum communication channel into the human cognitive centre (Ware, 2012) scientists now say they cannot do scientific research without such visualizations (McGrath and Brown, 2005).

Related Research

The video game industry has a comparatively short history compared to other media industries, such as the film industry, which has a history that spans more than one hundred years. Aarseth (2001) editor of the Game Studies journal noted that "2001 can be seen as the Year One of Computer Game Studies as an emerging, viable, international, academic field." The primary Simulation and Gaming (2020) journal was published for the first time in March 1970 and was joined at the start of the millennium (February 2020) by Game Studies (2020) the international journal of computer game research. Subsequently several other peer-reviewed game studies journals have joined the assembly such as Games and Culture (2020), International Journal of Gaming and Computer Mediated Simulations (2020), The Computer Games Journal (2020) and International Journal of Game-Based Learning (2020). The Digital Games Research Association (2019) founded in 2003 focuses on game studies and associated activities and has grown over the years with multiple regional chapters.

Various fields contribute to the study of video games and many are interdisciplinary; the study of video games can be approached from a wide range of academic perspectives. Numerous studies have investigated the theoretical framework of video games with a focus on rules (the design of the game), play (the human experience of playing the game) or culture (the larger contexts engaged with and inhabited by the game) (Salen and Zimmerman, 2004). The theoretical foundation of which has been extended and applied to other domains under the nomenclature of both gamification (Deterding *et al.*, 2011) and the application of serious games (Boyle *et al.*, 2016; Ritterfeld, Cody and Vorderer, 2009).

From a purely algorithmic perspective the functionality of games engines has been leveraged and repurposed for use within the realm of computer simulation owing to their rendering capability, physics simulation framework and navigation capabilities (Paravizo and Braatz, 2019). However, studies of the asset pipeline workflow and its application are distinctly lacking beyond information shared in videogame production post-mortems communicated to industry insiders at the Game Developers Conference (2020) or within online trade publications such as "Gamasutra the art & business of making games" (2020).

The philosophy of design patterns, idioms and pattern languages has been extensively researched since their inauguration by Alexander in 1975 (Alexander, Ishikawa and Silverstein, 1977). Jim Coplien (1992) catalogued computer language-specific C++ solutions to problems named idioms in 1992. During this period Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, also known as the "Gang of Four", authored the object-oriented design patterns book (Gamma *et al.*, 1995). In 1994 The Hillside Group organized and hosted the first Pattern Languages of Programs (2020) conference (PLoP), the premier event for pattern authors and pattern enthusiasts to gather, discuss and learn more about patterns and software development. Subsequently patterns have been designed, implemented and communicated in many diverse areas such as computer science, pedagogy, business (Fellmann *et al.*, 2019), communication (Schmidt *et al.*, 2000) and even a meta pattern language for pattern writing (Object, Meszaros and Doble, 1996).

Taking inspiration from the patterns of Gamma *et al.* (1995) a single piece of literature devoted to the pattern-oriented approach to video game development was produced in 2014. The book titled "Games Programming Patterns" (Nystrom, 2014) identifies patterns to solve the design challenges commonly encountered in computer game development. Those functional challenges as identified by Nystrom (2014) are: sequencing and time management, behaviour interaction and performance. Although the application of the patterns devised by Nystrom (2014) are useful for the programmatic design and implementation of a video game application, they do not extend to the that of the overall production workflow; specifically, the asset pipeline.

Video games studies has moved into a second phase in which having set its foundations as an academic field of study it must now attempt to articulate its exact nature and scope, codify its tools and terminology, and organize its findings into a coherent discipline (Perron and Wolf, 2009). This novel study formalizes the asset pipeline into a pattern language, a multi-disciplinary approach, to allow for effective communication of best practise to those practitioners both within the industry and outside. That is the application of patterns to expand communication in an industry shrouded in secrecy – a secrecy motivated for the reasons elaborated in the following section.

1.2 Research Motivation

The following details the motivation for the research covering: the lack of game development language; the use of non-disclosure agreements (NDAs) to protect intellectual property such as proprietary asset pipelines; the interdisciplinary language barrier between creatives and software engineers and the lack of technical resources devoted to a data-oriented design. These problems have directly resulted in a lack of dissemination of technical resources to wider audiences that could benefit such a workflow. To the author's knowledge, no study has specifically looked at the asset pipeline; and certainly, no study has formalized the asset pipeline into a pattern language for wider consumption.

Terminology: The lack of game development language

One of the challenges for video game theory is the industry is fifty years old, relatively young, and as such a set of agreed-upon terms and their definition has been slow to develop. Even the name of the subject itself has multiple nomenclatures; "video game", "videogame", "computer game" and "digital game" (Perron and Wolf, 2009). This inconsistency of standardization of terms can even be seen with recent industry innovations; consider the rapid proliferation of mass-market consumer software that takes inspiration from computer games usually summarized as "gamification". Deterding *et al.* (2011) endeavoured to formalize a definition for the genre and identified the following disparate terms in use: "productivity games", "surveillance entertainment", "funware", "playful design", "behavioural games", "game layer" and "applied gaming". However, for the focus of this research, the asset pipeline, a common term has not been identified for communication both within the video game industry and for those domains that utilize a similar workflow. There is a great deal of disparity in the use of lexicon with the following terms generally applied: "toolsets" (Anderson *et al.*, 2008), "asset build pipeline" (Ansari, 2011), "game asset pipeline" (Lengyel, 2010) and "asset compiler" (Ansari, 2011). O'Donnell (2014) maintains the video game industry "has struggled with an adequate vocabulary or set of practices by which to systematize its internal logics and methods."

Likewise, an attempt was made within the world of journalism to assist reporters who write about video games inviting consistent style, vocabulary and accuracy regarding names and terms. The authors Orland, Steinberg and Thomas (2007) collated "The Videogame Style Guide and Reference Manual". They assert "a consistent style helps engender trust from readers, and, on a larger scale, lend legitimacy to our industry". With the focus of journalism on clear, concise communication the authors have used the following guidelines in making their decisions: ease of comprehension for a general audience, common usage and accuracy, convenience with respect to write use and remembrance and official styling as preferred by video game developers and publishers. Although valuable the terminology set out by the authors occasionally contradicts their guidelines. For example, their use of the single word "videogame" rather than "video game" runs counter to one of their research criteria "Common Usage and Accuracy". That is a query with the search engine Google found 701 million hits for "video game" but only 160 million for "videogame" (query issued May 2020).

Video game production relies on both a set of complex technologies and workflows the definition and boundaries for which are often not easily understood. Consider the game engine, a technology consisting of a number of abstract sub-components. There is a distinct disagreement about what exactly a game engine is, with sometimes fundamental differences between definitions. Simpson (2002) reports on the confusion between a game engine and the actual game, as well as the erroneous description of a game engine being the component of the game for displaying graphics. For the case of the asset pipeline, a workflow not studied academically, it can be difficult to visualize where the pipeline workflow sequence starts and exactly where the pipeline ends. Does the pipeline encompass all the workflow performed by an artist within the DCC tool such as the creation of assets? And does it end at the point of entry into the game engine application?

There is no common terminology and definition for the asset pipeline and as such it cannot be easily communicated within the industry itself and certainly cannot easily be communicated and applied to further domains that could benefit such a workflow. The reason for this is twofold: the video game industry is clouded in secrecy with limited concrete problem-solutions and the production workflows are interdisciplinary leading to a mismatch in both understanding and standardization of terms. Both of these characteristics are explored below.

Non-Disclosure Agreements (NDA): Secrecy within the industry

Secrecy pervades the video game industry and prevents the natural dissemination of information. Video game development companies attempt to control and structure communication outside of their companies with the inclusion of non-disclosure agreements (NDAs). A non-disclosure agreement is a legal contract intended to protect confidential or sensitive intellectual property (IP). This includes all tangible aspects of the game experience including the narration, artistic assets, audio and all production code including tools such as the asset pipeline (Chandler, 2019). Legally valid types of intellectual property (IP) include copyrights, trademarks, trade secrets and patents. Examples of non-disclosure agreements (NDAs) are business proposals, financial data, new intellectual property and trade secrets. Under a non-disclosure agreement, the signer promises the recipient that they will not disclose certain information to certain parties under circumstances described in the contract. The value of the non-disclosure agreement (NDA) is meant to be that: the signer is legally bound by their promise not to disclose the information to third parties and if the promise is broken the recipient is court summoned to stop them, and may sue them for damages or other legal remedies (Purewal, 2010). Publishers control the information flows to ensure marketing departments manage the public relations of the soon to be released intellectual property. The hardware manufacturers ensure their underlying technological infrastructure is not leaked to other manufacturers or computer hackers hoping to circumvent the copy protection on their devices to extend the functionality beyond that of their intended design. This level of self-censorship operates at all levels in the value chain.

A primary example is demonstrated by the high-profile case concerning Facebook and Zenimax Media. Facebook was sued for $2 billion by Zenimax Media in a claim that a former employee, turned Oculus chief technology officer (CTO), took trade secrets integral to the Oculus Rift Virtual Reality headset with them (Polygon, 2020). This resulted in a settlement of 500 million dollars four years after the initial litigation began. Thus, according to O'Donnell (2014, p.206), "[d]evelopers acquire a built-in paranoia about what can and cannot be discussed, resulting in a kind of constant self-policing that resides at the core of our consent to hegemony." Information, workflows and state of the art is not disseminated both within and outside of the industry.

One of the fundamental dimensions of secrecy within video game production is the importance of speaking the language of the industry. Language is a precursor to many of the other barriers of entry into the video game industry (O'Donnell, 2014). One example of game developer lexicon is the pervasive video game industry reference to "Super Mario Bros." (SMB) released for the Nintendo Entertainment System (NES) in October of 1985. The acronym SMB has now taken cultural temporal reference to categorize a video game embodying a certain genre. SMB is now a staple of the video game end-user vocabulary in a similar manner to the Gregorian calendar AD or BC. Since there is no discipline of game design or game development the games themselves have become a lingua franca.

Interdisciplinary: Language barrier

Video game production requires significant interdisciplinarity and technical skills variety. The software engineers privilege the engineering endeavour, designers privilege the design elements, artists privilege the aesthetic endeavour; and the other disciplines such as audio, script and production claim their keys to video game development. It is the coming together of all these elements into a system that results in video games being functional and interesting. However, interdisciplinarity collaboration is tempered by the ability to communicate and work across disciplinary divides (O'Donnell, 2014). Smith (2007, p.36) recognizes "[i]n a collaborative environment where each person brings ideas for improvement and innovation, getting the right people together is the key to creating quality." It is at the fault line where "creole languages" emerge (Galison, 1999). The process of communication is assisted by the creation of a new category of specialization – the technical artist. The technical artist speaks the language of the software engineer and artist. This specialization is difficult in young or small developer studios just commencing their journey into game development since the importance of these new categories is rooted in experience.

Technical resources: Data Oriented Design

Compounding this technical language barrier is the specialization of resources dedicated to imparting software engineers with the tools of the trade observed in both literature and online form. The literature is heavy engineering focused and lacks the information of how art, design and software engineering collaborate manifesting in the asset pipeline workflow (Baek and Yoo, 2015). Tutorials introducing the concept of rendering three-dimensional graphical models using the Graphics Processing Unit (GPU) of the target platform confuse the functional boundaries of the artist and the software engineer. Documented programming code demonstrations detail the data describing the three-dimensional geometric artist models hard coded in situ within the rendering engine (Shreiner, Kessenich and Sellers, 2016). The demonstrations would be better served reading the model polygon data from a file that can be further modified to produce other outputs rather than a basic rendering of the hard-coded object to the screen space. The data driven approach of the game engine dictates the vertices of a game asset are not hard coded in the actual video game application; they are externally defined in a digital content creation (DCC) tool and integrated through the conduit workflow of the asset pipeline. Guiding candidate video game developers towards conceptualizing the practice of video game development collaboratively would better serve the industry on best practices. However, unfortunately data oriented design is a departure from traditional code first thinking. The asset pipeline workflow is neglected in this discourse.

Quality of Life (QOL) Issues

The video game production process emphasizes extensive content over innovative gameplay mechanics leading to quality of life issues for those working in the industry (O'Donnell, 2014). Initially the quality of life issue was publicly revealed in an open letter on 10th November 2004 posted to a blog titled "EA: The Human Story" and signed "EA Spouse" (Dyer-Witheford and De Peuter, 2006). EA Spouse details how initial enthusiasm for a job with a company listed as one of the "100 Best Companies to Work For" according to Fortune had evaporated, as seven-day, 85-hour work weeks, uncompensated either by overtime pay or time off became routine. An International Game Developers Association (IGDA) survey (Weststar, Kwan and Kumar, 2019) reports that over half of video game developers (54 per cent) indicated they worked 40 to 44 hours per week during a regular schedule and 20 per cent worked 35 to 39 hours per week. Crunch time is the industry term that indicates an apparently unusual period of crisis in the production schedule. 41 per cent of participants reported their job involves crunch time and during crunch most employees reported working 50 and 59 hours (38 per cent) or between 60 and 69 hours per week (19 per cent). A sizeable minority (13 per cent) reported working more than 70 hours per week during development crunch.

Relationship with the ethos of patterns

The asset pipeline is a general-purpose workflow used during the production process of video game creation. However, the asset-pipeline remains a workflow not effectively communicated within the video game industry and not divulged to practitioners in further domains that could benefit from the application of such a workflow.

The reason being outlined above, namely the lack of formalization of video game language, the inherent secrecy maintained within the video game industry (NDAs); the interdisciplinary nature of the development process and use of creole languages; and the lack of concrete solutions oriented towards data-driven design leveraged by the asset pipeline. Since their introduction by Alexander (1977) and mainstream adoption by software designers (Buschmann *et al.*, 1996; Gamma *et al.*, 1995) patterns have been applied in a variety of other domains to solve these issues. Patterns provide a shared vocabulary, understanding and set of solutions to commonly recurring design problems.

Most recent literature on software development includes a definition of design patterns and the following is the definition Buschmann *et al.* (1996) supplies in a "A System of Patterns (POSA1)", the first volume of the Pattern-Oriented Software Architecture Series:

> "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."

Coplien and Schmidt (1995) provide a more succinct definition of a "pattern describes a solution to a problem in context." Design patterns capture the static and dynamic structure of solutions that occur repeatedly when producing applications in a particular context. Design patterns provide several positive benefits and properties that can assist with the mitigation of the problems identified above.

Patterns deliver a common vocabulary and shared understanding for design concepts (Rising, 1998). A shared vernacular eases the reuse of architectural knowledge and artifacts. Every pattern has a name; naming a pattern immediately increases the design vocabulary of a practitioner. The pattern name is a handle which can be used to describe a design problem, its solutions and consequences in a word or two. The video game industry has struggled with the formalization of common vocabulary and those languages that exist at the boundary of the asset pipeline, where engineer and artist converse.

Patterns document existing best practices built on tried and tested design experience (Rising, L., 1998). They "distill and provide a means to reuse the design knowledge gained by experienced practitioners," so that developers familiar with an adequate set of patterns "can apply them immediately to design problems without having to rediscover them" (Gamma *et al.*, 1995). Technical processes, understanding and experience are not commonly shared in the video game industry owing to the protection of intellectual property (IP) under the law of non-disclosure agreements (NDAs). Explicit highlighting of key design strategies and tactics through patterns can help novice developers and new participants ascend the learning curve on a project (Buschmann *et al.* 1996). Furthermore, patterns ease communication of design issues across development teams. Patterns give expert knowledge to non-experts; designers with less experience will have access to expert solutions that are proven to succeed. This can assist both new software designers and also experienced software designers working in new domains.

Patterns provide a common format for documenting software architectures (Rising, L., 1998). Even though there is no accepted definition of what constitutes a pattern there is a general agreement of the core format consisting of a name, context, problem and solution. The context is the situation in which the pattern applies, the problem refers to the goal you are trying to achieve in this context and the solution is the general design that anyone can apply which resolves the set of forces or constraints. With their explicit core pattern format patterns can be used to track the reasons why a certain design choice was selected, and others rejected.

Patterns capture experience in a form that is independent of specific project details and constraints, implementation paradigms and programming languages (Rising, L., 1998). Patterns represent abstractions of empirical experience and everyday knowledge. By abstracting from aspects that can distract from understanding the potential traps and pitfalls in a domain, patterns support developers in the selection of suitable software architectures, designs and coding principles to address new design challenges in new projects, without wasting time and effort implementing solutions that are known to be inefficient, error-prone, unmaintainable or simply broken.

The video game industry has only recently embraced the pattern movement with a set of low-level video game implementation idioms (Nystrom, 2014), although these have received popular uptake in an academic setting, the pattern paradigm has not extended to that of the asset pipeline. This novel research addresses this shortfall.

1.3 Research Problem Area

The research study will focus on the three broad domains of the asset pipeline, patterns and software engineering as shown in figure 1.3.

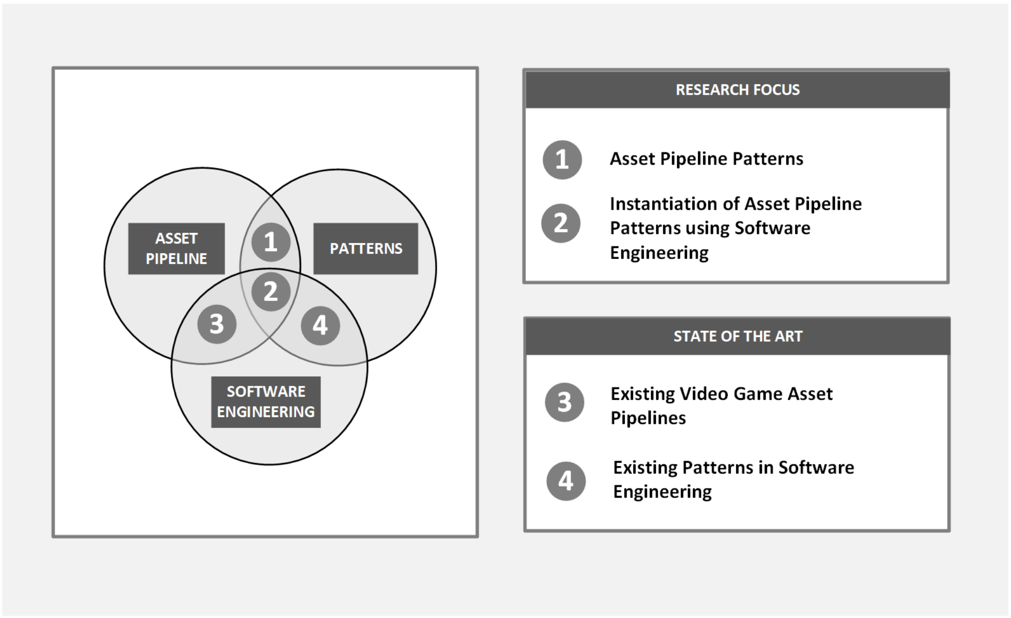

ASSET PIPELINE
PATTERNS
SOFTWARE ENGINEERING
1
2
3
4
RESEARCH FOCUS
1 Asset Pipeline Patterns
2 Instantiation of Asset Pipeline Patterns using Software Engineering
STATE OF THE ART
3 Existing Video Game Asset Pipelines
4 Existing Patterns in Software Engineering

Figure 1.3: Venn diagram highlighting the research study problem areas

The video game industry is inextricably tied to the creation of source code and the emergence of the software industry. Software industry luminaries such as Steve Jobs and Steve Wozniak made their early marks working for the video game developer Atari, Inc in 1974 (Zackariasson and Wilson, 2012). From those early days of an individual designing and working on all aspects of game development the production process has grown in parallel with the increase in computing power to incorporate artists, designers and software engineers. Where once video games were engineering bound or limited by the computational power of the systems on which they executed many game systems are now content bound or limited by the amount of material a studio can store on the target platform. Artists now incorporate the majority of a development team (Zackariasson and Wilson, 2012). This interdisciplinary fusion of talent has led to the creation of the asset pipeline; the workflow operating at the boundary of artist and software engineering. The first dimension of this research is the asset pipeline – an artifact born from an interdisciplinary field – better served viewed from the lens of the "world of creativity" than it does imagined from software (Zackariasson and Wilson, 2012).

The second dimension is the pattern, a general, reusable solution to a commonly occurring problem within a given context (Gamma *et al.*, 1995). The video game industry and the world of patterns has intersected in the form of a set of game programming patterns (Nystrom, 2014). The collection of patterns as identified by Nystrom (2014) are a set of low-level idioms that "make code cleaner, easier to understand, and faster." From their perspective of assisting video game application development at the source code level they are a far departure from the world of video game production and the asset pipeline. This research investigates the intersection of both the asset pipeline and a language of patterns (label 1, figure 1.3).

The third dimension is traditional software engineering. Patterns are a useful mechanism to communicate well proven design ideas. However, it is the application of patterns to a problem in context that ultimately leads to a solution. In the realms of the video game production workflow the technical artist is responsible for the actual realization of the software artifact known as the asset pipeline. Within the realm of this research the instantiation of the asset pipeline pattern language will be used to evaluate the effectiveness of the pattern language. Traditional software engineering development techniques will be used for this purpose (label 2, figure 1.3).

1.4 Research Aims and Objectives

Research Aims

The asset pipeline workflow has not been widely communicated by developers in the video game industry and has certainly not been disseminated to domains outside of the industry for wider application within the interactive real-time visualization field. The reasons being outlined in the research motivation section 1.2, namely the lack of formalization of video game language; the inherent secrecy maintained within the video game industry; the interdisciplinary nature of the development process and use of creole languages; and the lack of concrete solutions oriented towards data-driven design leveraged by the asset pipeline.

Given the above context the aim of this thesis is to formulate the asset pipeline into a pattern language and evaluate whether the approach can assist both artists and designers with the creation of such interactive real-time visualizations.

In order to address this aim and to both reveal new knowledge and illuminate the role of patterns in the development of interactive visualizations in real-time, this research will:

i) Determine how the asset pipeline assists in the design and production of visualizations as viewed within the video game industry. From this knowledge the functional components of the asset pipeline will be derived thus forming the seeds required to mine a pattern language.

ii) Determine the characteristics of a pattern language that make it a suitable approach to assist in the design and production of an asset pipeline workflow.

iii) The design and formalization of an asset pipeline pattern language for use both within the video game industry and for use in the wider context of interactive real-time visualizations.

iv) Determine whether the application of an asset pipeline pattern language can assist in the design and implementation of interactive visualizations.

Research Objectives

The research aims will be achieved by fulfilling the research objectives defined below:

i) Conduct desk-research of existing asset pipeline workflows as seen both in the video game industry and technologies such as the game engine and the XNA build framework. Create a common set of abstract requirements that all asset pipeline workflows represent.

ii) Conduct a literature review for pattern architectures, formats and methodologies for mining patterns; the state of the art for identifying and discovering patterns. Determine the characteristics of a pattern language that assist in the communication and design of development paradigms.

iii) Derive a pattern language using the spiral methodology aligned to the Pattern Languages of Program (PLoP) process. Submit patterns for peer review at the European Conference on Pattern Languages of Programs (EuroPLoP) 2019 conference and designation in the database for wider communication.

iv) Instantiate the asset pipeline pattern language resulting in the production of two interactive visualizations: A Building Information Management (BIM) architectural visualization (ArchViz) and an interactive real-time graph visualization (GraphViz).

1.5 Research Hypothesis and Associated Questions

This section provides the hypothesis that will drive the research study. Its related research questions are also listed and discussed.

> *"Application of a pattern language approach to the Asset Pipeline can assist both artists and designers in the communication, design and production of video game applications and interactive real-time visualizations."*

In order to evaluate the hypothesis, the following research questions will be answered.

RQ1. *How does an asset pipeline approach assist in the design and production of visualizations within the video game industry?*

The asset pipeline integrates an efficient workflow enabling designers to transform their source assets into the final runtime form (Llopis, 2004). Owing to the properties outlined in the research motivation (section 1.2) the asset pipeline has not been critically researched. Formalization of the asset pipeline into a pattern language requires an understanding of the set of problems solved by application of an asset pipeline, the context under which the problem exists and the applied solution. Research in the form of a comprehensive literature review is required to gain an understanding of these characteristics. These form the set of mined requirements and workflow abstractions for such a pipeline. It is only from this foundation that a set of patterns forming the asset pipeline pattern language may be generated.

RQ2. *What properties of a pattern language make it a suitable approach for the communication of nomenclature, good design and development practices within the context of an asset pipeline workflow?*

A pattern describes a solution to a problem within a context (Coplien and Schmidt, 1995). The general characteristics that all patterns exhibit will be elicited from a literature review of pattern definitions, properties and forms. The research motivation (section 1.2) highlights a number of problems that exist within the video game industry which inhibit the proliferation of the asset pipeline as a general-purpose workflow. From the correlation of this information a determination of whether a pattern approach is beneficial for the communication of nomenclature, good design and development practices of the asset pipeline.

RQ3. *To what extent can an asset pipeline be formalized into a pattern language?*

Application of the spiral model methodology will drive an iterative approach to the generation of an asset pipeline pattern language. The framework of the Pattern Languages of Programs (PLoP) conference will be followed. Initially the asset pipeline requirements identified in RQ1 will be used to mine for candidate patterns. The identified properties and forms of a pattern resulting from answering RQ2 input into the pattern writing iterative phases under supervision of a pattern shepherd. A writers' workshop is used to validate the worth and integrity of the generated patterns. The result of which is refinement and publication of the pattern language in the Association for Computing Machinery (ACM) proceedings for wider communication in related fields.

RQ4. Can an asset pipeline pattern approach be applied to assist in the design and production of a visualization?

The asset pipeline pattern language will be instantiated in two further domains of visualization. The first a real-time architectural visualization (ArchViz) and the second a real-time graph visualization (GraphViz). Both of which exhibit identical characteristics to those found within video game production. That is the conversion of source assets into a format appropriate for a runtime game engine. The former visualization utilizes Autodesk Revit as the originating Digital Content Creation (DCC) tool and the latter visualization utilizes the open-source graph visualization platform, Gephi (2020).

1.6 Contribution to Knowledge

The following are the key contributions of this research study.

- ***Asset pipeline workflow***

The asset pipeline workflow as seen in video game production lacks critical investigation. The workflow is core to the production of video game applications; however, the asset pipeline is both not commonly communicated within the industry and outside in further domains that could benefit from such application.

This is owing to the reasons outlined in the research motivation, section 1.2. Specifically, a lack of game development language; an inherent secrecy maintained in the industry with the use of non-disclosure agreements (NDAs) to protect intellectual property; a language barrier existing between creatives and software engineers and a distinct lack of technical resources devoted to the data-oriented design of the asset pipeline.

- ***Patterns in the domain of games technology***

Interdisciplinary research incorporating both games technology and the ethos of patterns has not been undertaken beyond the work accomplished by Nystrom (2014). The patterns of Nystrom are functional patterns at the low-level of idioms; useful optimizations at the algorithmic level. They do not leverage production lifecycle development such as the asset pipeline. This different angle adds value to a field little researched; the field of games technology patterns.

- ***Pattern Language in asset pipeline workflow***

A pattern language for the asset pipeline workflow, as seen in video game production, has been presented in chapter 3. This has been realized after combining the derived mined requirements with a pattern writing approach and following the rigorous iterative workflow of the Pattern Languages of Program (PLoP) process. The pattern language is published in the 24th European Conference on Pattern Languages of Programs Proceedings ACM (Lear, Scarle and McClatchey, 2019).

The production of the asset pipeline pattern language contributes to the following inherent properties of pattern generation identified in section 2.3.3. The properties and resulting contributions are:

i) Provide a common vocabulary for describing the asset pipeline workflow, including a set of nomenclature for the components of the pipeline. As a basis for application of the workflow into further domains a common nomenclature and vocabulary assists such communication amongst practitioners.

ii) The asset pipeline pattern language addresses and provides a solution to the recurring design problem as found in the context of video game production. The pattern language documents well proven solutions as identified in the literature review detailing the video game asset pipeline (sections 2.2.1 to 2.2.5), the scenario of the game engine asset pipeline (section 2.2.6) and the XNA build asset pipeline (section 2.2.7).

iii) A standardized pattern format for specifying the context, problem and solution of the asset pipeline components allowing for comparison and identification of the problem-solution dichotomy operating within a certain context. A critical evaluation of which is provided in section 2.3.4.

iv) Enables domain independent solutions to the problem of designers requiring a workflow to assist them in viewing their digital content in an interactive real-time visualization (sections 3.6.1 to 3.6.4).

- ***Instantiation of the asset pipeline pattern language in the production of interactive real-time visualizations***

The asset pipeline pattern language has been instantiated and applied in the fields of real-time interactive architectural visualization (section 4) and graph visualization (section 5). These exemplars provide a recognition of the application and use of games technology into further domains and provide a foundation for further research and development.

1.7 Methodology

In order to verify the hypothesis and understand the specifics of the problem initially a literature review will be performed in two domains. The first literature review domain, the asset pipeline workflow, will present the state of the art of the asset pipeline as seen in the video game industry. Two further scenarios will be researched: the asset pipeline component of a game engine and the asset pipeline contained within the integrated build framework of XNA. Analysis of the asset pipeline workflow literature will assist in answering Research Question 1. The findings will form the functional requirements and abstractions required for the derivation of a pattern language.

The second literature review domain, a discourse of patterns, will elaborate the definition of a pattern, the properties of a pattern that manifest in the solution to a problem within a certain context and the forms a pattern can take. These aspects together with the identified research motivations identified in section 2.1 provide an answer to Research Question 2 and form the foundation for the subsequent pattern language writing process.

Based on the above analysis application of the spiral model methodology will be used to drive an iterative approach for the derivation of a pattern language to answer Research Question 3, as illustrated in figure 1.4.

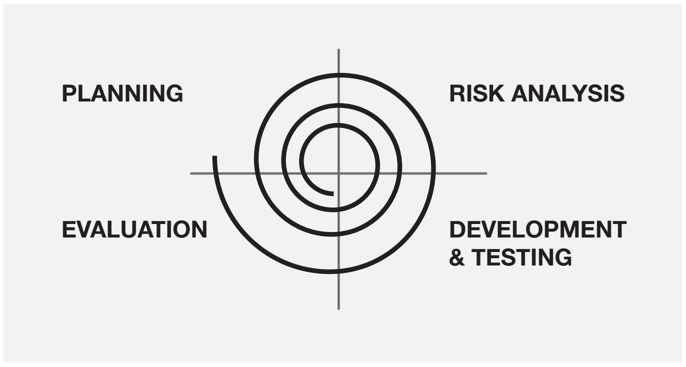


Figure 1.4: Spiral model methodology (Boehm, 1988)

The spiral model guides a designer to adopt elements of one or more process models such as incremental, waterfall, or evolutionary. It has two main distinguishing features. One is a cyclic approach for incrementally growing the degree of definition of a system and implementation while decreasing the degree of risk (Boehm, 1988). The second is a set of milestones for ensuring stakeholder commitment to feasible and mutually satisfactory solutions (Boehm, 1988). The anchor point milestones drive the spiral to progress towards completion.

Likewise, an iterative approach to writing patterns is recommended. John Vlissides (1998) in his work "Pattern Hatching, Design Patterns Applied" describes "iterating tirelessly" as one of the seven habits of effective pattern writers. The spiral model completely supports the ethos of pattern writing.

The spiral model is aligned with the framework of the Pattern Languages of Programs (PLoP) workflow encompassing two cycles as illustrated in figure 1.5. Cycle one incorporates the derivation of the pattern format; pattern mining for discovery of the individual patterns; pattern writing incorporating iterative feedback from a shepherd and finally completion of the cycle involving the first anchor point milestone of acceptance to the writers' workshop. Cycle two incorporates the feedback and findings from the writers' workshop and subsequent extensive refinement of the pattern language prior to the second anchor point milestone of acceptance for publication.

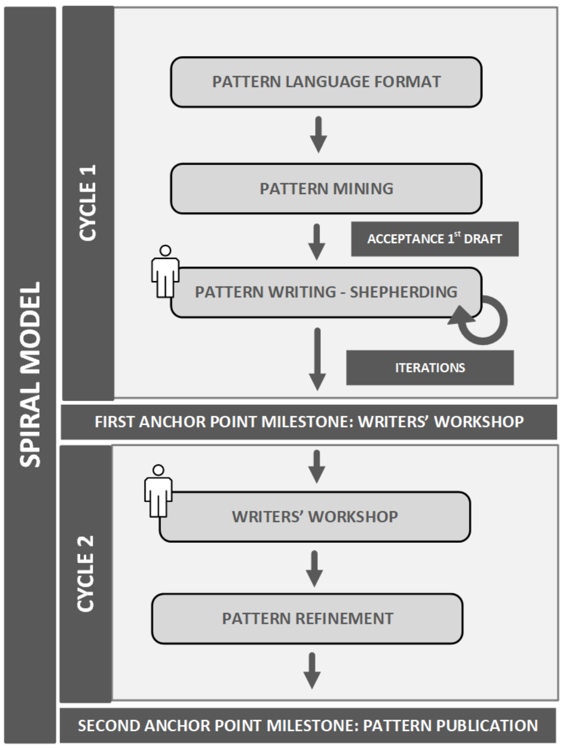


Figure 1.5: Pattern language formulation workflow

Finally, the asset pipeline pattern language will be instantiated in the production of two visualizations. The two real-time visualizations are within the domains of architectural visualization (ArchViz) and a graph visualization (GraphViz). The success of which answers Research Question 4.

1.8 Thesis Structure

The thesis is structured as indicated in figure 1.6 and elaborated below.

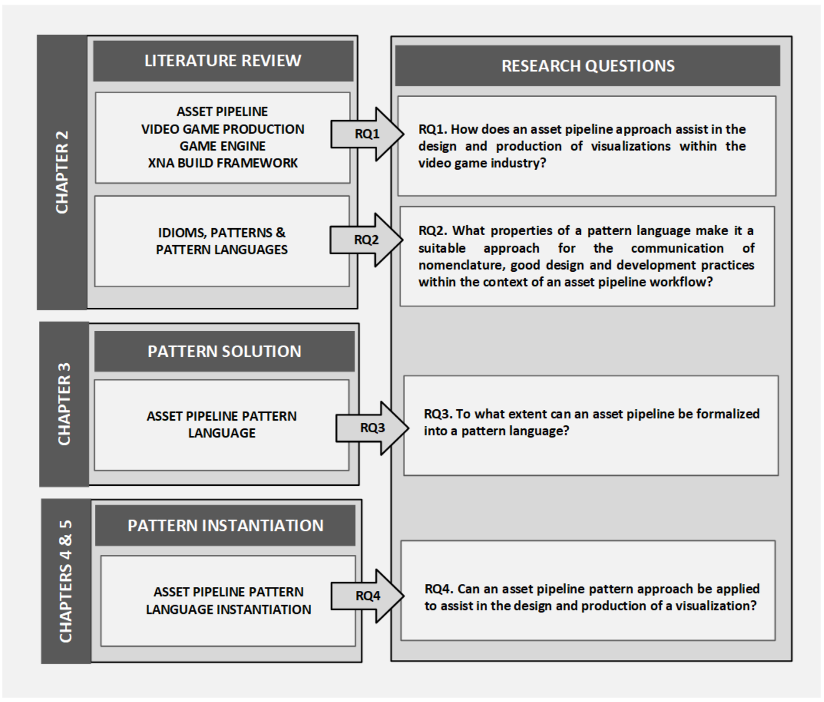

CHAPTER 2
LITERATURE REVIEW
ASSET PIPELINE
VIDEO GAME PRODUCTION
GAME ENGINE
XNA BUILD FRAMEWORK
IDIOMS, PATTERNS & PATTERN LANGUAGES
CHAPTER 3
PATTERN SOLUTION
ASSET PIPELINE PATTERN LANGUAGE
CHAPTERS 4 & 5
PATTERN INSTANTIATION
ASSET PIPELINE PATTERN LANGUAGE INSTANTIATION
RQ1
RQ2
RQ3
RQ4
RESEARCH QUESTIONS
RQ1. How does an asset pipeline approach assist in the design and production of visualizations within the video game industry?
RQ2. What properties of a pattern language make it a suitable approach for the communication of nomenclature, good design and development practices within the context of an asset pipeline workflow?
RQ3. To what extent can an asset pipeline be formalized into a pattern language?
RQ4. Can an asset pipeline pattern approach be applied to assist in the design and production of a visualization?

Figure 1.6: Thesis structure in relation to research questions

Chapter 2 – Literature Review

Chapter 2 provides a literature review in the two domains identified in figure 1.6; the asset pipeline and patterns. The chapter opens by providing the academic reader with an introductory contextualization regarding the functionaries in the video game industry and the video game development process. Essential information to situate the asset pipeline workflow.

Section 1 of the chapter discusses the asset pipeline as proposed in literature, particularly focusing on the pipeline utilized in the production of video games. It also presents the state of the art in the game engine asset pipeline and the XNA build framework. From this foundation a summary of the set of abstract requirements that all asset pipelines adhere to is stated. This section answers Research Question 1.

Section 2 of the chapter discusses the previous and related work in the area of idioms, patterns and pattern languages focusing on the pattern definition, its properties and its many forms. From this perspective an informed decision can be made of the constituent characteristics and benefits for the application of a pattern language to the asset pipeline. The related work in terms of game technology patterns is explored. This section answers Research Question 2.

Chapter 3 – Asset Pipeline Pattern Language

Chapter 3 presents the generated asset pipeline pattern language. The spiral model and framework of the Pattern Languages of Program (PLoP) conference is followed. The spiral model consists of two cycles. Cycle one consists of the functional steps of pattern language format, pattern mining and the iterative development of the patterns using a unique shepherding process. Cycle two expands the asset pipeline pattern development incorporating a writers' workshop. The results of which refine the asset pipeline pattern language to fulfil the milestone of acceptance and publication in the Association for Computing Machinery (ACM) proceedings for wider consumption. This chapter answers Research Question 3.

Chapters 4 and 5 – Asset Pipeline Pattern Language Instantiation

Chapters 4 and 5 evaluate the asset pipeline pattern language proposed approach. The pattern language is instantiated in two concrete solutions: a real-time architectural visualization (ArchViz) and a real-time graph network visualization. This section answers Research Question 4.

Chapter 5 – Conclusions and Future Work

Finally, the conclusions for the thesis are presented in chapter 5. The research questions are revisited in relation to the research work undertaken. The major contributions to knowledge are stated and the future directions are presented.

1.9 Publications

Following are the publications achieved as a result of the research study.

James Lear, Simon Scarle and Richard McClatchey. 2019. Asset Pipeline Patterns: Patterns in Interactive Real-Time Visualization Workflow. In 24th European Conference on Pattern Languages of Programs Proceedings (EuroPLoP'19). July 3-7, 2019, Irsee, Germany. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3361149.3361155

1.10 Summary

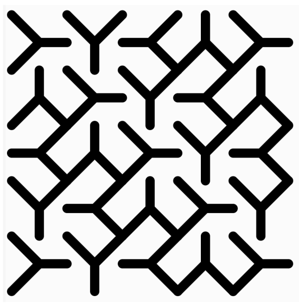
A video game consists of a virtual world modelled as an approximation of a real or imaginary environment. The expectation of the end-user and video game publisher has risen dramatically resulting in the amount of content required for triple-A video games to double every few years. The art discipline utilizes Digital Content Creation (DCC) tools to generate digital content in terms of source assets. The source assets are held in binary proprietary format and are information rich whereas the final runtime expects lightweight assets optimized for fast loading and execution. Artists use the workflow of the asset pipeline to create and review their designs in the runtime environment. The asset pipeline is the path that all game assets follow from their conception until they are loaded into the runtime (Llopis, 2014). The asset pipeline is one aspect of video game development least studied, researched and communicated outside of the video game industry. This is primarily owing to the inherent secrecy within the video game industry, the lack of game development terminology and the lack of data-oriented design examples.

The asset pipeline is a general-purpose workflow leveraged during the production process of video game creation. Likewise, within the domain of patterns a pattern language defines a general-purpose solution to a problem operating within a certain context. This study claims the asset pipeline is a template solution instantiated within the domain of video game production that can be extended into further domains with comparable workflows to assist practitioners in their creation. The hypothesis driving the research study is: application of a pattern language approach to the asset pipeline can assist both artists and designers in the communication, design and production of video game applications and interactive visualizations.

To verify the hypothesis and understand the specifics of the problem a literature review will be performed in the two domains of the asset pipeline workflow and the discourse of patterns. A spiral model methodology will be used to drive an iterative approach for pattern mining and the asset pipeline pattern language writing. The spiral model is aligned to the framework of the Pattern Languages of Programs (PLoP) workflow incorporating shepherding and a writers' workshop. Finally, the asset pipeline pattern language will be instantiated to evaluate its effectiveness in the production of two real-time visualizations: an architectural (ArchViz) and a graph visualization.

The following chapter provides a literature review in the two domains of the asset pipeline and patterns. The state of the art of the asset pipeline as utilized in the production of video games is presented, along with the asset pipelines of the game engine and XNA build framework. From this foundation a summary of the set of requirements that all asset pipelines exhibit is stated. The previous and related work in the area of idioms, patterns and pattern language is discussed, focusing on the pattern definition and the characteristics of a pattern that make it suitable for application within the domain of the asset pipeline.

2 Literature Review


Generative Pattern 2 – J Lear 2020

The literature review provides an assessment of the literature in the two domains of the asset pipeline and patterns. Appendix 1 provides the academic reader with the background information necessary to situate the workflow of the asset pipeline. That is the video game value chain, a definition of a video game and the roles of the development team.

Section 1 discusses the functionality of the asset pipeline as viewed within video game production. Two further scenarios are presented; the asset pipeline integrated within a game engine and integrated within a build framework. This section answers Research Question 1.

Section 2 discusses the previous and related work in the area of idioms, patterns and pattern languages. The focus being the pattern definition, the properties of pattern definition and its application, and the forms a pattern may take. The related work in the intersection of games technology and patterns, although minimal, is explored. This section answers Research Question 2.

2.1 Asset Pipeline

The very first visualization applications rendering three-dimensional graphics in real-time required expensive dedicated hardware and were mainly utilized in the domain of training simulations. The physical separation between the content and the runtime renderer did not exist in early applications such as the Lunar Module (LM) Mission Simulator (Rao, 1992). The Lunar Module Mission Simulator installed at the Kennedy Space Center between 1968 and 1972 was used by every Apollo astronaut to train for landing on the Moon prior to their mission. The content was embedded in code, or more specifically, a Fortran subroutine was coded to render a specific part of the content. Eventually the embedded content was abstracted to persist the data into general purpose arrays. The code evolved to become increasingly generic to facilitate the rendering of all types of data; and ultimately reuse across real-time simulations.

Similarly, during the beginning of the 1980s within the domain of the entertainment industry each new video game was developed from inauguration as single entity with no relation and reuse of technology from previous applications (Gregory, 2014). This was true for the home computer revolution and introduction of hardware platforms such as Commodore, Sinclair Research and Atari. Each video game was developed from the base layer upwards to make optimal and efficient use of the limited hardware resources of the time (Andrade, 2015). In parallel the rapid development of arcade hardware also meant that code optimized for one generation of hardware could not be used for subsequent generations. The underlying software technology stack was not utilized between generations, and so this foundation would remain through much of the 1980s.

Before the 1993 release of the upcoming game DOOM, "id Software" published a press release "heralding another technical revolution in PC programming, …, promising to push back the boundaries of what was thought possible on a 386sx or better computer" (Id Software, 1993). Within the press release John Carmack, the Technical Director of "id Software", introduced the term "DOOM engine". The DOOM engine not only signalled several technical advancements of the time such as texture mapping, but more importantly promised what Carmack described as "An Open Game". Owing to the popularity of WOLFENSTEIN 3D, the previous video game success of id Software, the general public responded with a "deluge of home-brewed utilities; map editors, sound editors, trainers, etc. All without any help on file formats or game layout from Id Software." (Lowood, 2014). In recognition of this Carmack conceived and developed a new way of organizing the components of video games by separating the core functionality of the game engine from the creative assets that filled the play space and content of a specific game title. Accordingly, Gregory (2014, p. ii) acknowledges:

> "Doom was architected with a reasonably well-defined separation between its core software components (such as three-dimensional graphics rendering system, the collision detection system, or the audio system) and the art assets, game worlds, and rules of play that comprised the player's gaming experience."

Kushner (2003, pp.71-72) who wrote a history of id Software during this period declares the separation of engine and assets as identified by Carmack resulted in a "radical idea not only for games, but really any type of media … It was an ideological gesture that empowered players and, in turn, loosened the grip of game makers." The advantages of the separation became evident as video game developers started licensing games and leveraging them into new products by introducing fresh content in the form of environmental art, world layouts, characters and game rules with only minimal changes to the "engine" software (Sanglard, 2018). This evolution marked the birth of the modification (mod) community; whereby individuals, groups and independent studios created new video games based on the game engines and public toolkits provided by the original developers.

During the latter period of the 1990s video games such as Quake III and Unreal were designed with the intent of reuse and modification of their underlying technology in the form of the game engine. The game engines were made highly customizable via the use of common abstract modules delivering specific technological functionality and scripting languages to assist in video game production.

In contemporary video game development, a developer studio has the choice of either licensing a game engine thereby reusing significant proportions of key software components or they can develop their own proprietary engine to meet the needs of their application requirements. From this foundation the contemporary game engine emerged based on the original concepts of reusability, modularity and extensibility (Charrieras and Ivanova, 2016). According to Lewis and Jacobson (2002) a game engine is a "collection of modules of simulation code that do not directly specify the game's behaviour (game logic) or game's environment (level data)." The game engine consists of a tool suite and a runtime component-based architecture (Gregory, 2014).

Garlan and Shaw (1993) proposes that "one characterization of progress in programming languages and tools has been the regular increase in abstraction level – or the conceptual size for software designers building blocks." A similar increase in abstraction level appeared in the architecture of video game development during this period. Typically, the layers of the runtime game engine architecture contain the following abstracted functional services: graphics renderer, collision and physics, audio, online multiplayer networking, artificial intelligence (AI), skeletal animation and front end (Bogdanowicz *et al.*, 2010; Gregory, 2014). Figure 2.1 highlights the major runtime components that compose a typical three-dimensional game engine.

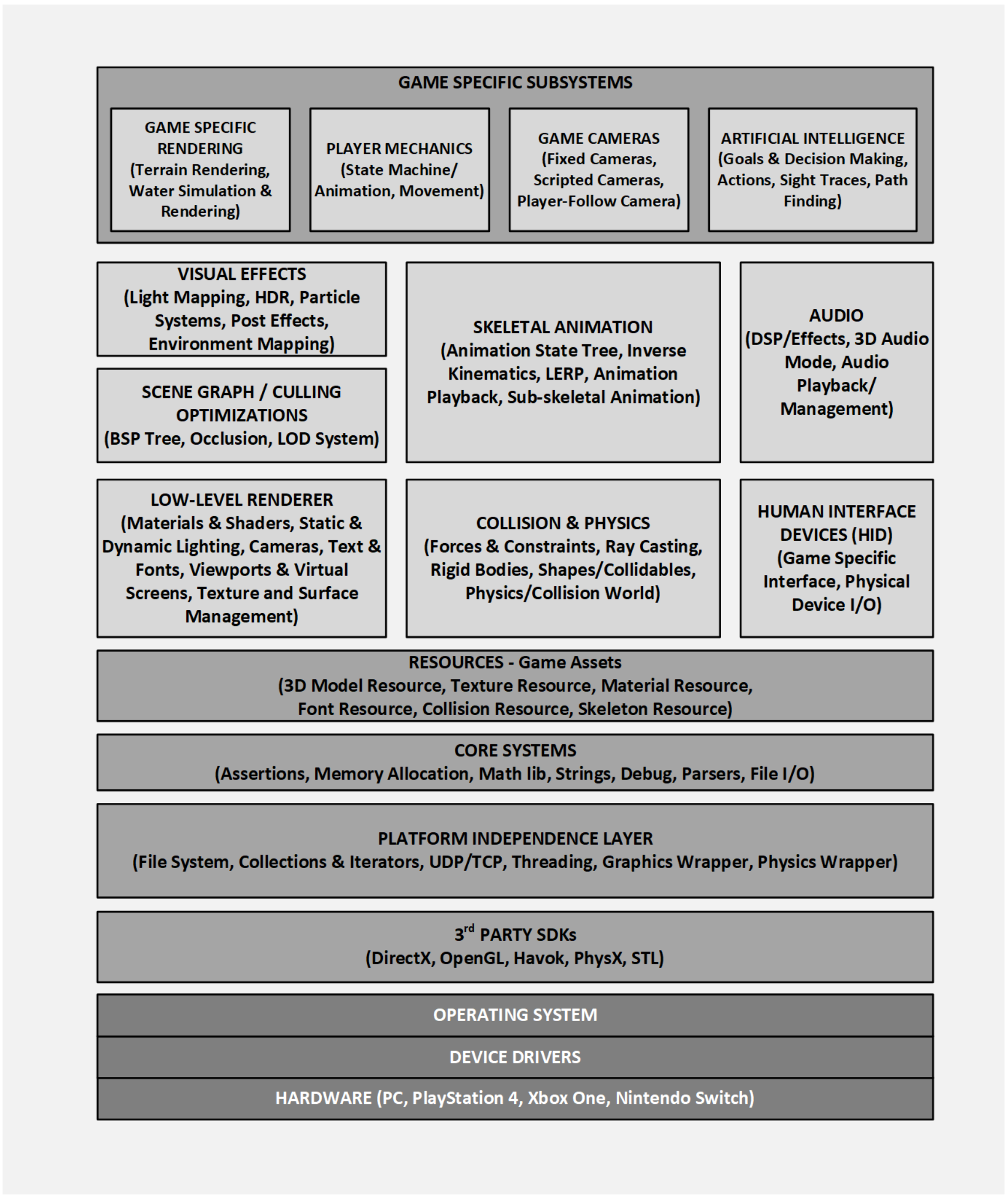

GAME SPECIFIC SUBSYSTEMS
GAME SPECIFIC RENDERING (Terrain Rendering, Water Simulation & Rendering)
PLAYER MECHANICS (State Machine/ Animation, Movement)
GAME CAMERAS (Fixed Cameras, Scripted Cameras, Player-Follow Camera)
ARTIFICIAL INTELLIGENCE (Goals & Decision Making, Actions, Sight Traces, Path Finding)
VISUAL EFFECTS (Light Mapping, HDR, Particle Systems, Post Effects, Environment Mapping)
SCENE GRAPH / CULLING OPTIMIZATIONS (BSP Tree, Occlusion, LOD System)
SKELETAL ANIMATION (Animation State Tree, Inverse Kinematics, LERP, Animation Playback, Sub-skeletal Animation)
AUDIO (DSP/Effects, 3D Audio Mode, Audio Playback/ Management)
LOW-LEVEL RENDERER (Materials & Shaders, Static & Dynamic Lighting, Cameras, Text & Fonts, Viewports & Virtual Screens, Texture and Surface Management)
COLLISION & PHYSICS (Forces & Constraints, Ray Casting, Rigid Bodies, Shapes/Collidables, Physics/Collision World)
HUMAN INTERFACE DEVICES (HID) (Game Specific Interface, Physical Device I/O)
RESOURCES - Game Assets (3D Model Resource, Texture Resource, Material Resource, Font Resource, Collision Resource, Skeleton Resource)
CORE SYSTEMS (Assertions, Memory Allocation, Math lib, Strings, Debug, Parsers, File I/O)
PLATFORM INDEPENDENCE LAYER (File System, Collections & Iterators, UDP/TCP, Threading, Graphics Wrapper, Physics Wrapper)
3rd PARTY SDKs (DirectX, OpenGL, Havok, PhysX, STL)
OPERATING SYSTEM
DEVICE DRIVERS
HARDWARE (PC, PlayStation 4, Xbox One, Nintendo Switch)

Figure 2.1: Game engine architecture and runtime components

The common functionality of the game engine is exposed to the higher-level video game application through the invocation of a Software Development Kit (SDK). The Software Development Kit consist of sets of development tools, libraries and applications that allow software developers to develop video game applications faster, at a lower cost and targeted to multiple hardware platforms. Gregory (2014, p.11) defines a "data-driven architecture is what differentiates a game engine from a piece of software that is a game but not an engine", highlighting the separation of the video game content with that of the algorithmic game engine runtime. As of 2020 game engines are licensed under three types of business models: open source, freeware and commercial engines. The choice of commercial game engines is large. The table 9.1 provided in Appendix 2 summarizes and updates a survey undertook by Andrade (2015) comparing the most popular game engines as utilized by both the hobbyist and the industry-level developer.

Although the line between the video game and the engine is often indistinct, Gregory (2014, p.ii) defines a "data-driven architecture is what differentiates a game engine from a piece of software that is a game but not an engine." The term "game engine" should be reserved for "software that is extensible and can be used as the foundation for many different games without major modification." (Gregory 2014, p.ii).

Given the primary goal of a game engine is to abstract common technological functionality for repurpose in further applications in a similar domain, the functionalities summarized in table 2.1 are typically exhibited.

Functionality	Description
Rendering Engine	Most two-dimensional and three-dimensional rendering engines utilize a hardware Application Programming Interface (API) such as OpenGL, DirectX and Vulkan. The rendering engine collects and submits geometric primitives such as meshes, point lists and particles to the hardware Graphic Processing Unit (GPU).
Scene Graph	A high-level component to limit the number of primitives submitted for rendering based on scene graph visibility determination.
Physics Engine	Consists of collision detection and rigid body dynamics supplied by third-party middleware libraries such as Havok, PhysX and Open Dynamics Engine (ODE).
Human Interface Devices (HID)	Processes input from the end-user from various Human Interface Devices (HIDs). Abstracts the low-level details of the HID on a particular hardware platform from the high-level video game controls.
Animation	A number of commercial animation packages exist such as Havok Animation and Edge to facilitate rigid body hierarchy animation and skeletal animation.
Artificial Intelligence (AI)	Either bespoke handled or middleware such as Kynapse is utilized for path finding, static and dynamic object avoidance and AI animation.
Online Multiplayer/Networking	Abstracts the low-level network layer as provided by the target platform to provide networked multiplayer modes whereby each platform is networked together to host one of the end-users.

Table 2.1: Common technologies exhibited in a game engine

After decades of being completely technology driven the motivation behind video games is now the video game content itself. The major technological leaps in terms of the advancement in Central Processing Unit (CPU) and the Graphic Processing Unit (GPU) do not make enough of a difference to distinguish video games within the marketplace (Llopis, 2004). A successful contemporary video game requires high-quality content. This evolution will persist as game content is continuing to get larger and more complex as the amount required for Triple-A (AAA) video games doubles every few years (Llopis, 2004). However, despite this, the production of video game content is a creative, time consuming process and does not get crafted any faster.

The critical path for the artists, designers and programmers working on the production of a video game is the game asset pipeline. Llopis (2004) defines the game asset pipeline as:

"The content pipeline is the path that all the game assets follow, from conception until they can be loaded in the game"

The game asset pipeline is the workflow or process needed to extract the asset from the editor in which it is being created or modified into the video game runtime environment so that it can be tested and examined (Carter, 2004). O'Donnell (2014, p.75) emphasizes the "real-time previews of game's content inside of a game's rendering engine is frequently cites as essential by artists." Since the vast majority of the asset creation process is iterative in nature and follows the basic design cycle methodology (Koomen, 1991) the bounding factor in video game production has been the time occupied in the asset feedback loop. With the majority of the video game production team consisting of artists and designers a slight inefficiency in the asset pipeline, such as taking one full minute from the time a change is made to the time it can be seen in the video game can easily cost a company thousands of wasted man hours during the course of production (Llopis, 2004).

Carter (2004, p.6) qualifies this with:

"The goal of a well-designed asset pipeline is to act as its name suggests: you put the source data in one end of the pipe, and it comes out of the other end in a form the game can use, hopefully as fast and with as little human intervention as possible."

The workflow and steps involved in the asset pipeline vary depending upon the scope and requirements of the development. For small productions the asset pipeline may be minimal and informal with the assets exported from their design tool and loaded directly into the runtime of the game engine, figure 2.2.

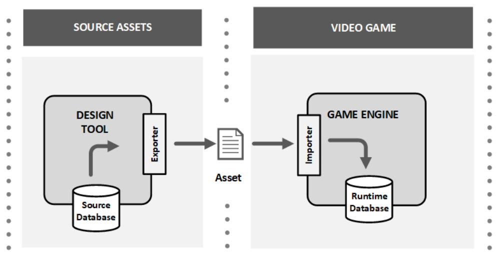


Figure 2.2: Single stage asset pipeline

Whereas, for larger productions and team sizes this approach cannot scale and an intermediate stage is introduced to provide extra processing. This additional stage is referred to as the asset build process and is illustrated in figure 2.3.

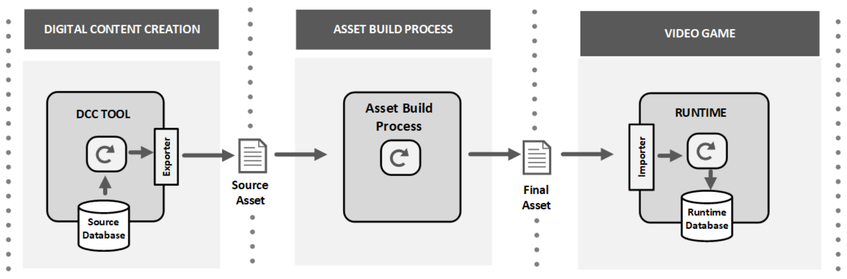


Figure 2.3: Multi-stage asset pipeline

The following sections elaborate on the stages of the asset pipeline commencing with the source assets; the first stage of the asset pipeline.

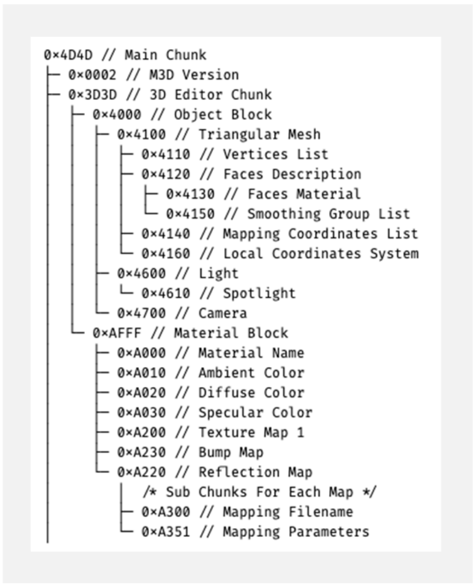
2.1.1 Source Assets

A video game is a type of multimedia application and as such the input data comes in a wide variety of forms; from three-dimensional mesh data, texture bitmaps, animation data and audio files (Gregory, 2014). The source assets "…include everything that is not code: models, textures, materials, sounds, animations, cinematics, scripts and so forth" (Llopis, 2004). All the source data must be created, modified and finalized by the artists and designers within the developer studio.

The application the designer uses is known as the Digital Content Creation (DCC) tool. Predominantly the DCC tool used by the designer falls into one of two categories. They are either one of the relatively small number of commercial applications or they are a bespoke tool used to create specific product content. A specialized DCC tool usually creates one particular type of asset data (Gregory, 2014), a tool for three-dimensional modelling, one for texture creation, one for audio and so forth (Llopis, 2004). Although some tools integrate functionality and can produce multiple data types. For example, Autodesk Maya and 3ds Max, both extensively used within video game production for the creation of three-dimensional meshes and also animation data.

There are situations whereby some types of data specific to the video game runtime cannot be created using an off the shelf DCC tool. For example, most game engines such as Unity and Unreal Engine provide a custom editor for designing environmental scenes within the game world. A survey of the typical DCC tools and their domains of use is summarized in table 9.2 provided in Appendix 2.

As identified in table 2.3 the source asset is often stored in a tool-specific binary format within the DCC tool persistent store known as the source database. Taking for example the three-dimensional DCC modelling tool, Autodesk 3ds Max 2021. The binary file format known as 3ds is held in a hierarchical structure of chunks similar to an XML DOM tree. Figure 2.4 shows an extract from a file representing the hierarchical structure and dependencies.



```
0×4D4D // Main Chunk
├─ 0×0002 // M3D Version
├─ 0×3D3D // 3D Editor Chunk
│  ├─ 0×4000 // Object Block
│  │  ├─ 0×4100 // Triangular Mesh
│  │  │  ├─ 0×4110 // Vertices List
│  │  │  ├─ 0×4120 // Faces Description
│  │  │  │  ├─ 0×4130 // Faces Material
│  │  │  │  └─ 0×4150 // Smoothing Group List
│  │  │  ├─ 0×4140 // Mapping Coordinates List
│  │  │  └─ 0×4160 // Local Coordinates System
│  │  ├─ 0×4600 // Light
│  │  │  └─ 0×4610 // Spotlight
│  │  └─ 0×4700 // Camera
│  └─ 0×AFFF // Material Block
│     ├─ 0×A000 // Material Name
│     ├─ 0×A010 // Ambient Color
│     ├─ 0×A020 // Diffuse Color
│     ├─ 0×A030 // Specular Color
│     ├─ 0×A200 // Texture Map 1
│     ├─ 0×A230 // Bump Map
│     └─ 0×A220 // Reflection Map
│        │  /* Sub Chunks For Each Map */
│        ├─ 0×A300 // Mapping Filename
│        └─ 0×A351 // Mapping Parameters
│
```

Figure 2.4: Autodesk 3ds Max proprietary format

Owing to their proprietary format the data formats used by DCC tools are rarely suitable for direct use within a video game runtime for two reasons. Firstly, the algorithms and in memory model of the DCC tool data is usually more complex than the expectations of the runtime game engine. For example, the three-dimensional modelling workflow within the DCC tool Autodesk Maya uses high level mathematical concepts for surface descriptions such as Non-Uniform Rational Basis Splines (NURBS) and splines. The history of all edits that have been stored on the file are also persisted to assist the designer with modifying their models.

Internally, Autodesk Maya persists a Directed Acyclic Graph (DAG) of scene nodes with a complex hierarchy of interconnections within the source database. This scene represents the position, orientation and scale of every object or entity within the scene as a relative hierarchy of three-dimensional transformations decomposed into mathematical translation, rotation and scale components. DCC tools and advanced rendering techniques such as shader languages and ray tracing require more advanced concepts than the game engine runtime database can handle. The game engine requires a tiny fraction of this information in order to render the model in real-time. Secondly the proprietary file format of the DCC tool is too slow to read and parse at runtime.

2.1.2 Extracting Source Assets

The source assets must be extracted from the DCC tool in a form digestible for the game engine runtime. The DCC tools provide a host of standard and semi-standard export formats to allow interoperability between each other. However, often none of the formats are perfectly aligned for the task of video game development; the impedance mismatch provides a challenge for video game development (Gregory, 2014). The internal format of the DCC tool is proprietary in nature and so it is not possible to explicitly design and implement an algorithm to import the necessary assets directly into the game engine runtime; even when the specification of the format is available.

To overcome this challenge an exporter must be written to extract the data from the source database of the DCC tool into a format sustainable for further stages of the asset pipeline, figure 2.5. This introduces flexibility in selecting the subset of information to export and the choice of format.

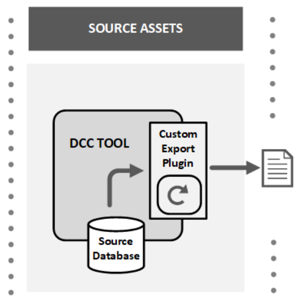


Figure 2.5: Custom export plug-in extracting source assets from the DCC tool

The exporter utilizes the built-in algorithms and functions of the DCC tool to convert the internal representation of the source asset into a more suitable usable format. To assist developers in extracting data the vendors of the DCC tools typically offer Software Development Kits (SDKs) that provide an Application Program Interface (API). The DCC tool vendors prefer supporting the Software Development Kit (SDK) rather than publishing the details of their internal algorithms and allowing developers to directly process the proprietary format. The reason being the Software Development Kit abstracts and hides the internals mechanism of the DCC tool algorithms and data-structures. If a developer depended upon the internals of a DCC tool new releases of the tool would break any dependencies. The Software Development Kit exposed by the DCC tool provides stability and control in this respect.

The export plug-in integration provided by a DCC tool vendor generally exists in an ecosystem sandbox forcing a developer to use the provided Application Programming Interface (API) for security reasons. Common functionality exposed by the Software Development Kit (SDK) includes interfaces to query the internal representations of the scene data, write the data to a file and perform any conversion tasks as required. For applications manipulating complex data, such as three-dimensional modelling programs, this can be a complex task which requires detailed understanding of the underlying workflow of internal asset creation.

For example, Autodesk Maya stores graphical information in a Dependency Graph (DG) database (Autodesk Maya, 2017). The information in the Dependency Graph is stored in object called nodes. Nodes have properties called attributes that store the independent configurable characteristics of each node. Similar data types of nodes can be connected together to allow for the free flow of information and data from one node to another. The data flow internally is a mechanism for implementing construction history of a Maya scene. The complexity of the scene becomes apparent when one considers there are over six hundred built-in nodes in the ecosystem and nine hundred commands allowing for interaction and manipulation. Fortunately, the DCC tool vendors offer support in the way of example demonstration plug-in code that can be adapted for further means.

The export plug-in is not limited to directly reading the DCC tool source database and writing the data in a format applicable for further stages in the asset pipeline. It is common and desirable for an export plug-in to perform additional processing on the source data leveraging the algorithms and functionality provided by the Software Development Kit (SDK). Three-dimensional modelling tools use higher-order primitives such as Non-Uniform Rational Basis Splines (NURBS), however, rarely are these supported in the game engine runtimes. While it is possible to export these primitives as is and then tesselate them into triangles at a later stage in the pipeline it is more efficient to perform this transformation at the end point of export. At this stage the host DCC tool source database has the necessary support data available and the developer has access to a Software Developer Kit (SDK) providing the necessary functionality to provide transformation steps.

In an ideal world the DCC tool would be able to efficiently export all source assets automatically into the final asset format compatible with the game engine runtime. However, in practise this “single stage” asset pipeline is unachievable for all but the simplest of cases. Carter (2004) observes the following reasons:

Firstly, the majority of DCC tools incorporate both a high-level scripting language to automate common tasks and customize the user-interface, and a low-level Software Development Kit (SDK) for interrogating and manipulating the DCC tool source database internal representation as discussed above. However, very few DCC tools offer sufficient third-party integration to allow for the direct creation of the raw final assets as consumed by the game engine runtime.

Secondly, exporting the game-ready final asset from within the DCC tool tightly couples the export plug-in of the DCC tool to the format the game engine runtime requires. Any requirement changes to the format has the repercussion of invalidating the existing generated assets. Under this scenario the designer would have to re-export all of the source assets – a possibly time consuming and tedious task.

2.1.3 Pipeline Asset Format

For the scenario of three-dimensional source assets and video game specific data it is usually necessary to design a bespoke single file format as a foundation for the container for all subsequent asset pipeline processing stages. In only the simplest of cases is a single stage asset pipeline used whereby a particular format is derived for all stages of the pipeline and input directly into the game engine runtime. It is normally appropriate for such data to be encapsulated in an intermediate format which is ultimately converted to the final asset targeted to a specific game engine hardware runtime. There are several reasons as to why this additional step is necessary.

Primarily the video game is operating in near real-time conditions and maximum efficiency must be introduced in terms of final asset loading time and any additional processing required to be performed during the runtime start-up. This is especially apparent for those applications that require data to be read or streamed whilst the application is in operation. The final data format should be as close to what is actually contained in memory while the video game is executing. Any processing that must be performed on the data after or during the loading must be initiated whilst the runtime is operating causing significant impact on performance.

It is desirable to perform as much pre-processing as possible within the asset pipeline to ensure the minimum of impact to runtime performance. However, storing data in the final runtime format has a negative impact on compatibility. The closer the data format is to the final format as expected by the game runtime the more likely the format will change if an alteration is made to the algorithms or data-structures of the game engine runtime. During the early phases of development, the systems are experimental, and data-structures are changing rapidly to accommodate new features. If every change required all the source assets to be reexported from the DCC tools they were created in, it would generate a huge amount of time-consuming work for the designers.

Applying and exporting all source assets in an intermediate format that is not coupled to the final game engine runtime data-structures minimizes the impact on the designer. Any changes in the data-structures of the game engine runtime simply require propagation into the convertor that produces the game ready final assets. All affected intermediate files are processed rather than performing a complete export of all source assets. The final assets can then be regenerated in bulk using the build process, rather than impacting the artist. According to Lengyel (2010, p.23) the "[p]rimary advantage of providing a step between the source asset and the final asset format is decoupling engine development and asset creation."

The following scenario identified by Lengyel (2010, p.25) based on the development of the "MechAssault 2" video game highlights the advantages of storing source assets in an intermediate container:

> "A large studio shipped a game based on a new IP and cleverly stored all the source assets, source code, and build processed. It took many years, but eventually the sequel was ordered. The problem was that the source assets were not recognized by the new version of the DCC tools. Cleverly, they had also stored the tools used to create the assets. The problem was there was no way to get the license server working for those old tools. Intermediate assets were stored in a binary opaque format, so no help there. All the assets had to created again. Using a text/XML documented intermediate format in the archive would have saved a lot of time and money"

The requirements for the intermediate asset differ from those of the final game-ready assets: "The intermediate asset format should be very easy to read, parse, and extend. Intermediate assets should contain all the information that may be necessary downstream the pipeline, and be lossless wherever possible" (Ansari, 2011).

The above can be satisfied using any plain text file format, although XML provides distinct advantages. XML is a self-describing human-readable format that can be queried and manipulated using many common programming languages. It defines a hierarchical Document Object Model (DOM) that aligns with the hierarchical structure of the source scene graph (Lengyel, 2010).

In an ideal world a single intermediate asset format would be designed to contain the complete set of data necessary for all transformation stages in the asset pipeline. This approach introduces the flexibility of adding, removing and re-ordering the workflow steps whilst also standardising the use of common file processing.

Given the intermediate asset format acts as a "buffer between the source and final assets" (Ansari, 2011) the design of such a single common format is complex in practise. It must store sufficient source information for input into the build process but must also be capable of storing the final asset representation.

Creating a proprietary intermediate format has the disadvantage that a custom exporter must be developed as a plug-in for each DCC tool used by the artist to transform the internal representation of the source assets into the intermediate format. This can be a potentially daunting task considering each DCC tool exposes asset information through a separate Software Development Kit (SDK). Arnaud and Barnes (2006, p.vii) highlights the difficulty Tim Sweeney, the founder and CEO of Epic Games and the creator of the Unreal Engine, had within production:

> "in developing the Unreal Engine, my company has written importers for five scene file formats, and export plug-ins for the three major 3D applications. Each required significant development and testing effort. What's worse, every time we wanted to add a new feature, we have to update five import modules and three export plug-ins."

As a solution an intermediate format known as COLLADA (COLLABAborative Design Activity) was developed as a joint collaboration between Sony Computer Entertainment and the Khronos Group (Lengyel, 2010). The aim being to standardize a common intermediate representation for use by DCC tool vendors so that individuals do not have to implement custom exporters and importers. Technically COLLADA is an XML based file format supporting the transfer of common types of three-dimensional data between applications. Such types of data include three-dimensional models, polygons, vertices, textures, cameras, lights, and many more. Using XML as a container provides extensibility to support further three-dimensional features as they evolve. The uptake of native COLLADA is growing with a number of DCC tool and game engine manufacturers integrating the technology.

2.1.4 Asset Build Process

The intermediate assets must undergo a number of steps before they can be integrated into the final video game. The asset build process is the component tasked with performing the transformations. The exact nature of the steps is driven by a number of factors including the type of assets involved, the target game and the target platforms.

Comparable to the software development toolchain encapsulating the functionality of compiling and building source code for each target platform, the asset build process is responsible for compiling and optimizing the exported source assets for each target runtime platform. Each build node within the asset build process illustrated in figure 2.6 represents a single step that performs a transformation on one or more input assets into one or more output assets. Typically, the asset build nodes can be categorized into three distinct flavours: convertors, calculators and packers (Llopis, 2004).

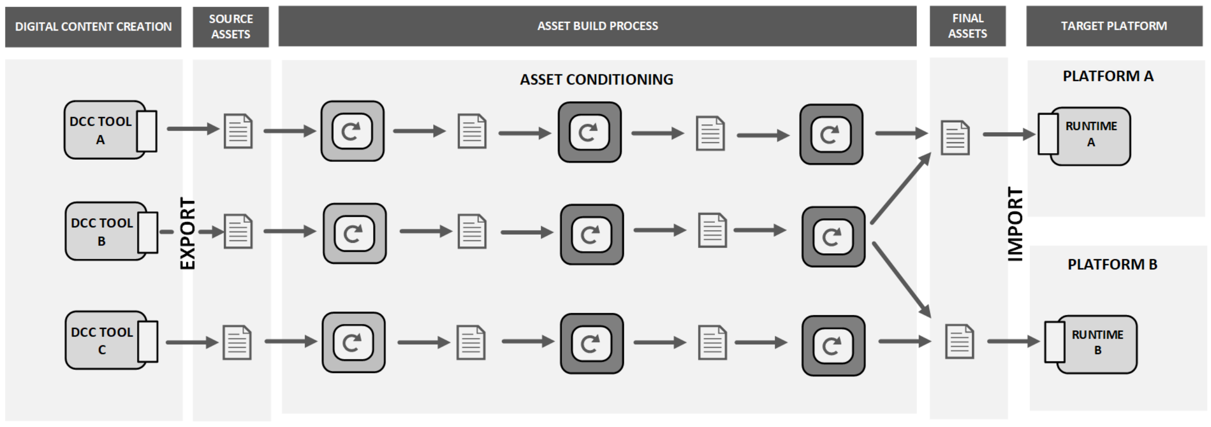


Figure 2.6: The asset build process

The convertor build node takes as input the asset data as arranged in a particular data structure format and transforms its representation into another set of data structures, which are often target game engine runtime specific. For example, taking as input a texture format stored in a Portable Network Graphics (PNG) format and converting the texture into form that can be directly loaded into the graphic memory of the target platform (DXTC). The calculator build node takes as input an asset or collection of assets, applies a set algorithmic calculation on the data set and outputs the result. Calculations performed comprise long running processes such as creating normal maps, baking scene lighting and using high resolution meshes to produce displacement maps. Whereas, the asset packer build node collects individual assets and packages them into data sets for use in particular instances of a game. Typical uses for the packages are to amass all assets used in a particular section of a video game or to collect chunks of data for data streaming purposes.

Day 1 studios (Lengyel, 2010) describe the functionality of their asset build process:

> "Format changes, mip-map generation, dithering, compression, lightmap generation and mesh optimizations. Some assets split different assets. For example, model XML file split into several smaller files containing vertex and index data. Sometimes multiple assets get combined into one larger asset (packing a set of textures into one larger texture)."

Owing to the common expectations of the game engine runtime functionality the basic algorithms used for handling asset processing are largely identical amongst projects. Predominantly the asset build process is concerned with manipulation of texture and geometric asset types.

Textures are two-dimensional arrays of colour information universally always stored as RGB (Red, Green, Blue) or RGBA (Red, Green, Blue, Alpha) colour channels in 32-bit precision. Modest texture sizes are 2K (2048x2048) pixel density, although higher resolution 4K (4096x4096) textures are increasingly common. There is a significant amount of research regarding image processing, however, the asset build process generally performs the following texture manipulation operations.

Texture Swizzling

The arrangement of pixels in Graphics Processing Unit (GPU) texture memory affects how efficiently the pixels can be accessed for rendering purposes. While traditional texture memory is stored as a linear array of rows data is rarely accessed in this manner. In order to improve the hit rate of the GPU cache the texture is swizzled whereby the texels are rearranged in memory ensuring adjacent texel fetches are more likely to find the data already in the GPU cache.

Texture Compression

Texture compression is a specialized form of image compression used to reduce the amount of storage space and bus bandwidth required by textures at runtime. In their paper on texture compression Beers, Agrawala and Chaddha (1996) specify four features that differentiate texture compression from other image compression manipulation techniques. These are decoding speed, random access, compression rate and encoding speed. Examples of practical texture compression systems are S3 Texture Compression (S3TC); Ericsson Texture Compression (ETC); Adaptive Scalable Texture Compression (ASTC) and PVRTC.

Texture Resizing

Texture resizing is performed as a pre-processing operation in two cases: reduction of texture size to conserve memory and to generate mipmaps (MIP maps) for textures. Mipmaps are a set of precalculated, optimized sequence of images at progressively lower resolutions for display at different texel densities. Primary used for level of detail (LOD) rendering within large environments they result in increasing rendering efficiency.

The majority of three-dimensional DCC modelling tools operate using higher order mathematical primitives to assist the designer with the creation of objects. The designer typically interacts with meshes constructed as a series of vertices, the points in three-dimensional space, and the triangles or polygons that connect them to form a surface. Consumer Graphics Processing Unit (GPU) hardware does not natively support the rendering of n-gons, or polygons with n sides. Owing to the unification of rendering formats such as DirectX, OpenGL and Vulkan a number of universal techniques for pre-processing three-dimensional mesh geometry has been developed.

Stripification

Modern Graphics Processing Unit (GPU) hardware operates using the data-structure of triangles as the input to the rendering process (Aguilar, Alexánder, and Saúco, 2020). A mesh or sequence of triangles can be submitted to the GPU pipeline as a simple list, each triangle comprising of three distinct vertices. However, this is inefficient since each mesh forms a continuous surface and each triangle shares vertices with its neighbours. Meshes form a continuous surface and each triangle thus shares vertices with its neighbours. Construction of a triangle strip, sharing common vertices, using the SGI algorithm and submission to the GPU increases efficiency of the rendering pipeline.

Mesh Subdivision

There are situations whereby it is desirable to iteratively subdivide the polygons of an existing mesh to provide a higher density mesh in order to improve the accuracy of both lighting and shadow calculations. Furthermore, an increased density mesh increases the precision of bounding volume and clipping issues.

Mesh Compression

The Graphical Processing Unit (GPU) of target hardware is bound by memory limitations; particularly with contemporary real-time video games where thousands of triangles are needed to make a surface look smooth. Geometry compression is used in situations where the meshes have a sufficiently high number of vertices and polygons that they represent a significant amount of memory. Mesh compression provides an efficiency saving twofold: the storage requirements for geometry within the application is condensed and the amount of geometric data sent to the GPU pipeline to render the model is reduced.

The actual implementation of the asset build process is dependent upon a number of factors including the target game engine runtime and the workflow of the designers and artists. The framework “Day 1 Studios” implemented consisted of a command line conversion tool taking as input the intermediate assets and producing as output the final runtime assets. The asset build node steps are executed as a set of conversion plugins, implemented as DLLs, each of which processes a certain type of resource identified by the file extension or the header information in the file.

2.1.5 Asset Dependencies

Efficiency within the asset pipeline is essential for the success of video game production. Lengyel (2010) explains this is especially true towards the latter stages of the development lifecycle:

> “Early in the process, little attention may be given to the optimization of the build process and the organization of the data, but the overall quality and success of the game will depend on the final editing that occurs when most of the assets have been created. In other words, a poorly designed build process is likely to directly impact the quality of the end product.”

One method of optimization in the asset build process is to perform incremental builds and only convert assets that have been subsequently modified. Within the asset build process an individual build node can only be executed when the requirements of its parent dependencies are satisfied. Thus, a build node can only be re-evaluated when its recursive parent dependencies differ in significant way. The order of the execution of build steps are determined by the overall set of dependencies.

Determination of the list of dependencies requires construction of a Directed Acyclic Graph (DAG) resulting from the union of each nodes dependencies. However, a major issue with this approach is the lack of information regarding the dependency hierarchy. Since source assets are stored in a DCC tool specific opaque binary format parsing the file and recursively collecting dependency information is difficult. Given a processing step is deterministic and for a given input the same output is always generated there are two approaches for determining dependencies.

Firstly, the dependency information is hard coded into a master script whereby the designer explicitly updates the script when a new asset is created. The asset build process sequentially executes the build nodes against the static dependency data. A disadvantage to this approach is it is prone to human error and is not flexible in approach. Deployment of the build process can utilize bath or shell scripts or simple “makefiles”.

Alternatively, each asset build node is abstracted as a templated description of the inputs that are required, the outputs that are generated and the parameters used in the compiler. A dependency graph is generated without knowledge of the asset contents by applying an algorithm that matches a set of input and output file types. The asset build process does not know in advance the dependency information and relies explicitly on dynamic dependency extraction. Implementation of such an approach uses a bespoke file scanner to read, parse and extract the relevant dependency information. The Open Source Scons software construction tool is based on such a framework.

2.1.6 Scenario: Game Engine Asset Pipeline

A game engine Is a reusable development environment originally established for use within the video game industry, however, recently found use in other domains such as real-time simulation visualizations. The core functionality of the game engine consists of reusable components to provide scene building, 3D graphics rendering, physics simulation, audio and behaviour scripting.

All source assets designed within a DCC tool for use within a game engine must undergo a sequence of steps to transform the asset into a final asset format for use in the runtime. A proprietary asset pipeline internal to the game engine is invoked to perform this processing. The following use case consists of the scenario of importing a three-dimensional "tree" asset modelled in the DCC tool 3ds Max for use in a real-time visualization, although in actual practice the scenario is applicable for any form of three-dimensional model. The game engine under consideration is the Unity game engine, a game engine popular amongst content creators owing to its modest licensing model, extensive functionality and wide variety of supported platforms. Please refer to table 2.1 for a feature comparison of Unity with other game engines.

The example workflow is illustrated in figure 2.7. The three-dimensional "tree" model is exported from the DCC tool 3ds Max into the Autodesk FBX standard format, step 1.

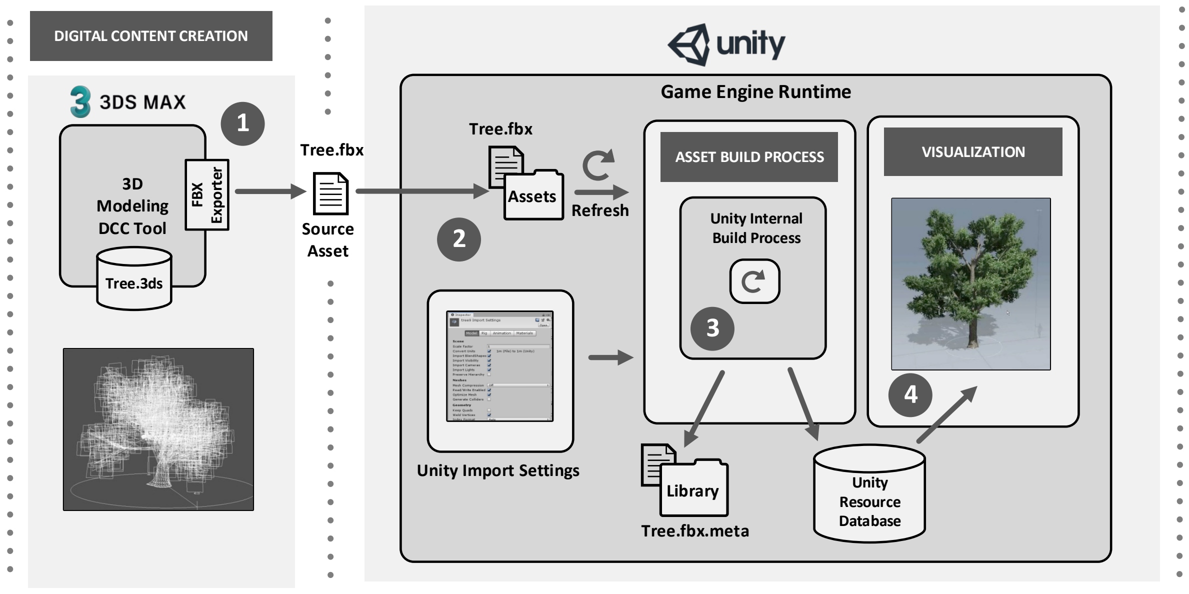
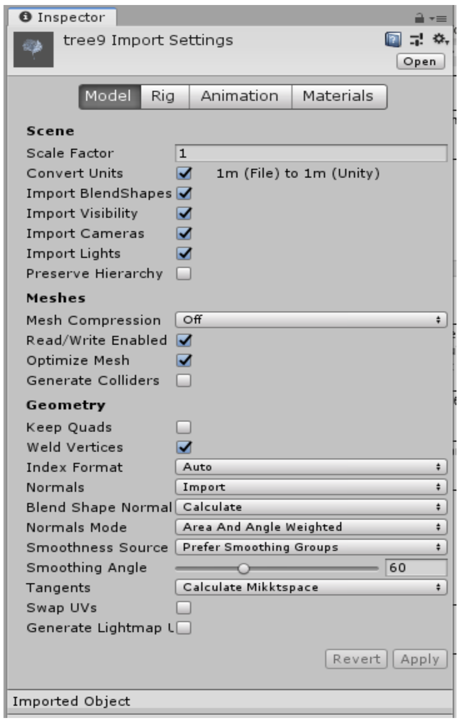


Figure 2.7: Unity asset pipeline – exporting a 3D model from a DCC tool into a game engine for real-time visualization

The designer copies the resulting asset into a public "Assets" folder within the Unity project and specifies the import settings for the three-dimensional model, step 2. The import settings provide a user interface for the designer to manipulate the internal game engine asset pipeline by dictating the build node steps performed and therefore alter the processing of the asset, figure 2.8.



Scene

Scale Factor	1	
Convert Units	✓	1m (File) to 1m (Unity)
Import BlendShapes	✓	
Import Visibility	✓	
Import Cameras	✓	
Import Lights	✓	
Preserve Hierarchy	☐	

Meshes

Mesh Compression	Off
Read/Write Enabled	✓
Optimize Mesh	✓
Generate Colliders	☐

Geometry

Keep Quads	☐
Weld Vertices	✓
Index Format	Auto
Normals	Import
Blend Shape Normal	Calculate
Normals Mode	Area And Angle Weighted
Smoothness Source	Prefer Smoothing Groups
Smoothing Angle	60
Tangents	Calculate Mikktspace
Swap UVs	☐
Generate Lightmap U	☐

Figure 2.8: Unity asset import settings categorized into scene, meshes and geometry

Although the individual low-level asset build node implementation stages are proprietary in nature the high-level functionality is exposed and can be inferred from the import settings. The documented feature set is detailed in table 2.2.

Category	Property	Function
Meshes	Mesh Compression	The mesh compression ratio determines the file size of the mesh. Increasing the value reduces the file size, although reduces the precision of the mesh.
	Read/Write Enabled	Determines how the mesh integrates with the GPU pipeline. When read/write is enabled Unity can access the mesh data at runtime and manipulate the mesh characteristics.
	Optimize Mesh	When enabled Unity will optimize the order of the triangles in the mesh by reordering the vertices and indices for better GPU performance.
	Generate Colliders	On import automatically generate collision meshes for the environment geometry.
Geometry	Keep Quads	Prevents Unity from directly converting polygons that have four vertices to triangles. A copy of the original mesh is created prior to triangulation.
	Weld Vertices	Combine vertices that share the same position in space to optimize the vertex count on meshes.
	Index Format	Specify the size of the Mesh index buffer.
	Normals	Determine how and if the normal are calculated upon import.
	Normals Mode	Specify the algorithm and weighting for calculating normals.
	Smoothness Source	Set how to determine the smoothing behaviour.
	Smoothing Angle	Determine whether the vertices are split for hard edges.
	Tangents	Determine how and if the vertex tangents are calculated.
	Generate Lightmap UVs	Generates a second UV channel for Lightmapping.

Table 2.2: Unity asset import settings and functionality

Unity monitors the public Assets folder for new file additions and upon recognition of a new asset file invokes the internal asset pipeline. The source asset file is converted into the internal runtime version using the import settings specified by the designer. This is illustrated as step 3 of the workflow in figure 2.7. Unity uses final versions of the assets within the runtime and preserves the unmodified source asset files in the “Assets” folder. The converted internal representation is cached within the “Library” folder and referenced in the resource runtime database. An import can sometime generate multiple final assets. For the situation of the 3D file, such as “tree.FBX”, that also defines materials or embedded textures, the textures are extracted and represented within Unity as separate assets.

Alongside the invocation of the asset pipeline, Unity assigns a unique ID to the asset. This ID is used internally by Unity to refer to the asset. The asset can therefore be renamed or moved without references to the asset breaking. A ".meta" file is created for each asset created in the "Assets" folder. Within the ".meta" file is the unique ID generated for the asset along with the import settings specified by the user. If such import settings are changed the asset is reimported and the final runtime asset will be updated in the "Library" folder of the project. Finally, on completion of the asset pipeline the asset is available for use in the runtime for visualization.

Unreal Engine 5 (2021) extends the philosophy of the internal asset pipeline as exhibited by Unity to potentially achieve photorealism on a par with computer-generated (CG) movies and real life. The technology termed Nanite allows a DCC artist or designer to produce fine-grained geometric detail comprising hundreds of millions or billions of polygons. Nanite virtualized micro polygon technology implements a real-time asset pipeline complete with a build process incorporating dynamic scaling and streaming of geometry. This technology, although not currently released, pushes the offline static asset build process into the game engine itself.

Although the example uses a game engine as the visualization runtime application, the asset pipeline pattern exists in many forms of visualization software where an asset has been designed using abstract high-level concepts and must be rendered using low-level runtime structures.

2.1.7 Scenario: XNA Asset Pipeline

Microsoft XNA or MonoGame is a toolset with a managed C# runtime that facilitates video game production. Within the suite of development tools is a component named XNA Build. XNA Build is a set of asset pipeline management tools which assist in the definition, maintenance, optimization and debugging of the game asset pipeline. There are two parts to the asset pipeline. One is the compile-time code which converts the source assets to the final loadable assets. When compiling the application, the compile-time code is executed once for every asset type being loaded. In effect the asset pipeline uses a templated processing scheme. The runtime code is executed for every object that is loaded. The diagram in figure 2.9 illustrates the steps that a source asset file takes to become a processed file or ".xnb" in XNA.

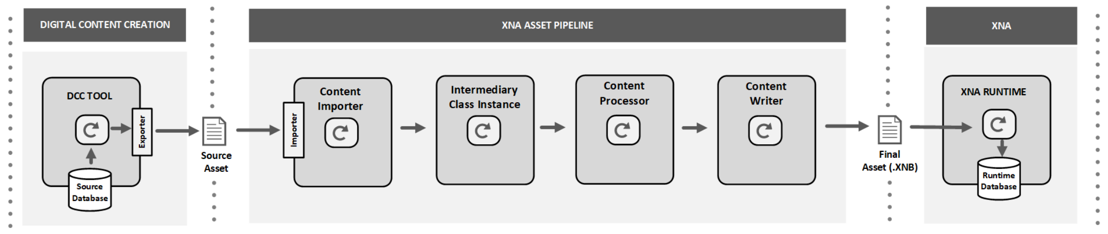


Figure 2.9: XNA asset pipeline stages

The source asset is a file that is created in a DCC tool such as Autodesk Maya. The compile-time code converts the source asset file to the final .XNB file using the asset pipeline. The content importer is a class that inherits from the "Microsoft.Xna.Framework.Content.Pipeline.ContentImporter<T>" responsible for loading the source asset file and creating the intermediatory class. There are several built-in importers and they are detailed in table 2.3. The output of the import process is an intermediary object that the next stage in the asset pipeline can progress.

Importer	Description
Autodesk FBX Importer	Imports .fbx 3D model files.
X File Importer	Imports .x 3D model files.
Effect Importer	Imports .fx files.
SpriteFont Description Importer	Imports fonts (.spritefont files).
XML Content Importer	Imports various .xml files.
MP3 Audio File Importer	Imports MP3 files.
Texture Importer	Imports a wide variety of images.
WAV Audio File Importer	Imports audio in the .wav file format.
WMV Video File Format	Imports video in the .wmv file format.
XACT Project Importer	Imports XACT audio projects.

Table 2.3: XNA built-in content importers

The intermediary class represents the intermediary state between the source asset and the final and processed asset. This class may include external references and dependencies to other files. For example, a file containing a three-dimensional model may contain references to the texture image files. The content processor is a class that takes an instance of the intermediary class and performs any processing. Such examples of processing include an effect processor, texture processor and model processor.

The content writer is an object which takes as input an instance of the intermediary class after it has been manipulated by the content processor and outputs the data into the final “.xnb” format. The “.XMB” processed final file is a binary file consisting of compiled content as read by the XNA game engine. The built-in content processors are details in table 2.4.

Content Processor	Description
Effect Processor	Processes effect files (.fx)
SpriteFont Description Processor	Processes spritefont descriptions from .spritefont files.
SpriteFont Texture Processor	Processes spritefonts from images/textures.
Model Processor	Processes .fbx and .x files.
Song Processor	Processes audio files as Songs.
Sound Effect Processor	Processes audio files as SoundEffects.
Texture Processor	Processes images as textures.
Video Processor	Processes videos.
XACT Project Processor	Processes XACT audio projects.

Table 2.4: XNA built-in content processors

Additional importers and processors can be created to transform bespoke formats. These can be added to the asset pipeline for processing.

2.2 Idioms, Patterns and Pattern Languages

2.2.1 Pattern Definition

During the period between the 1960s and 1970s the architect Christopher Alexander and his colleagues pioneered the application of design patterns in the context of the design and construction of buildings. Alexander captured his findings in a series of three volumes: "The Timeless Way of Building" (Alexander, 1979), "A Pattern Language: Towns, Buildings, Construction" (Alexander, Ishikawa and Silverstein, 1977) and "The Oregon Experiment" (Alexander, 1975). These seminal books present the view of the author as a language of the recurring problems he saw in cities and towns, neighbourhoods and buildings. Alexander describes the problems and their corresponding solutions using an expression he calls a pattern.

In his introductory discourse the "Timeless Way of Building" Alexander (1979) develops a philosophical argument focused on the design and construction of buildings using a pattern language achieving what he refers to as a "quality without a name." Alexander (1979) draws a parallel between the defining quality that we seek in a building with the quality a person seeks in their own life. A person can recognize the feeling this quality creates and likewise can also recognize this quality when it manifests in a building (Alexander, 1979). For Alexander (1975, p.53) this association is cyclical in nature and:

> "Places which have this quality, invite this quality to come to life in us. And when we have this quality in us, we tend to make it come to life in towns and buildings which we help to build."

Alexander (1979) views the world as having a definite structure because repeating pattern of events are rooted in a physical space. To Alexander the action and space are inseparable and form a unit known as a "pattern of events in space".

Observationally Alexander (1975, p.82) views, "On the geometric level, we see certain physical elements repeating endlessly, combined in an almost endless variety of combinations." A city is composed of roads, houses, gardens, shopping centres, places of work, parks ad infinitum. Whereas, at a lower level of granularity a building consists of rooms, walls, ceilings, windows, doors, stairs repeating.

Alexander, Ishikawa and Silverstein (1977, p.x) defines a pattern:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

For an element in the world, each pattern is a relationship between a certain context, a system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. The pattern is both an element in the real-world and the set of rules and instructions required to create it. Alexander (1979, p.247) summaries the pattern definition as:

> "a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

A language of patterns contains interconnected patterns and for "[e]ach one is an invariant field, needed to resolve a conflict among certain forces, expressed as an entity which has a name, with instructions so concrete that anyone can make one, and with its functional basis so clearly stated that everyone can decide for himself whether it is true, and when, and when not, to include it in his world." When correctly expressed the pattern defines an invariant characteristic that captures all the possible solutions to the given problem in the stated range of contexts.

The following Alexander pattern named "Window Place" illustrates a problem-solution pair operating within a context:

> "Window Place
> Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them … A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease…
> If the window contains no window which is a 'place', a person in the room will be torn between two forces:
> 1. He wants to sit down and be comfortable.
> 2. He is drawn toward the light.
> Obviously, if the comfortable places – those places in the room where you most want to sit – are away from the windows, there is no way of overcoming this conflict …
> Therefore: In every room where you spend any length of time during the day, make at least one window into a 'window place'."

After examining the work of Alexander and the field of building architecture software designers recognized recurring design problem-solution pairs in software. They discovered analogies between Alexandrian patterns and patterns in software architecture and design. From these humble beginnings there is now clear evidence in literature of patterns operating at all levels of software design; from the granularity of high-level architecture, object-oriented design to low level language specific idioms.

Gamma *et al.* (1995) established the lexicon for software design with their documentation of object-oriented design patterns as a catalogue of twenty-three patterns varying in granularity and abstraction. The design solutions provided by Gamma *et al.* (1995) are expressed in terms of objects and interfaces instead of walls and doors, however, as with the patterns of Alexander, the patterns of Gamma *et al.* (1995) follow the philosophy of providing a solution to a problem in context.

The patterns described are general purpose solutions to common object-oriented problems that have been discovered in more than one existing system. They are not low-level designs such as idioms and similarly they are not high-level complex, domain-specific designs for an entire software architecture. Gamma *et al.* (1995, p.13) details the patterns providing an object-oriented paradigm of "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." This follows the problem-solution dichotomy identified by Alexander and subsequently maintained by fellow pattern authors. Accordingly, Gamma *et al.* (1995, pg.3) provides the following functional definition:

> "A design pattern names, abstracts and identifies the key aspects of a common design structure that makes it useful for creating reusable … design."

Whereas the catalogue by Gamma *et al.* (1995) concerns design-level patterns, the patterns of "Pattern-Oriented Software Architecture Volume 1: A System of Patterns" (POSA1) concerns the wider context of software architecture; from high-level architectural patterns, through design patterns to low level idioms. Taking inspiration from the work of Alexander (1977) and Gamma *et al.* (1995), abstracting from specific problem-solution pairs and distilling out the common factors led to the generation of software architecture patterns (Buschmann *et al.*, 1996, p.3).

Buschmann *et al.* (1996, p.8) provides the following definition of a pattern in the domain of software architecture:

"A pattern for software architecture describes a particular recurring design problem that arises in specific design context and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."

Recurring in the discourse of patterns each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. According to Buschmann *et al.* (1996) this three-part schema underlies every pattern. The context extends the plain problem-solution dichotomy by describing situations in which the software architecture problem occurs; it provides a situation giving rise to a problem. All three parts of the schema are tightly coupled.

At a lower level in software architecture Coplien catalogued solutions to language specific problems called idioms and evolved two basic definitions of patterns. Coplien (1992) initially defines the pattern thusly, "A pattern is a solution to a problem in context." Whereas, Coplien and Schmidt (1995) elaborates the definition as "Design patterns capture the static and dynamic structure of solutions that occur repeatedly when producing applications in a particular context." The two definitions are similar; they both describe a solution to a problem in context, confirming the findings that came before them. The latter adds the idea that the solution has been found more than once in various applications. This implies that a solution cannot be a pattern unless it has been found repeatedly in a problem domain. Therefore, the more the application of a pattern can be found the more one can be sure of the quality of the solution it presents, the "quality without a name" (Alexander, 1975).

One agreed upon definition of a pattern as identified within literature and catalogued in the Pattern Languages of Programs collection is the Alexandrian canon that a pattern is the three-part rule of "a solution to a problem in a context" (Alexander, 1975).

2.2.2 Pattern Properties

Software design patterns were initially promoted to make the design of programs more flexible, modular, reusable and understandable (Gamma *et al.*, 1995); all perceived as positive characteristics of software design. A literature review has identified a fundamental set of themes as exhibited through the application of patterns. The themes as cited in literature are critically elaborated below.

Impact of design patterns on quality

Alexander puts forth the application of patterns within the context of building and architecture introduces a "quality without a name" (Alexander, 1975). In the practice of software engineering, application of design patterns encodes "good practices" guiding developers in solving recurring design problems operating within a certain context. Design patterns provide tested solutions saving the effort of designers and programmers from reinventing solutions. The process of reinventing solutions reduces efficiency and may result in solutions that are of less quality (Mawal and Elish, 2013). Nystrom (2014) whose background of industrial experience employed in the video game industry reenforces this viewpoint after observing co-workers struggling to reinvent good solutions where the solutions are already apparent within the same codebase. Design patterns are promoted to result in a more "flexible, modular, reusable, and understandable" design (Gamma *et al.*, 1995). Indeed, the subsequent application of the ethos of patterns within the domain of software engineering (Gamma *et al.*, 1995; Buschmann *et al.*, 1996; Coplien, 1992) as identified in classic literature has reenforced this common lore. Although researchers and practitioners have realized and emphasized the importance of design patterns in software design there has been a growing number of studies that critically investigate the role of design patterns with regard to the impact of software quality.

Wydaeghe *et al.* (1998) presented a paper on the concrete use of six design pattern identified by Gamma *et al.* (1995) when designing and implementing a simple editor. They studied the impact of these patterns on reusability, modularity, flexibility and the difficulty to instantiate the patterns into concrete implementations. The authors concluded that not all patterns have a positive impact on the attributes of quality. However, the study was conducted under the lens of the experience of the author and thus cannot be generalized to further contexts of development. Zhang and Budgen (2012) extended the study to a systematic literature review to identify primary studies utilizing the twenty-three patterns of Gamma *et al.* (1995). Their study could not identify firm support for any claims made for patterns in general although there is support for the usefulness of patterns for assisting novices learn about design.

Several researchers have studied the impact of design patterns on quality in the theme of software maintenance and evolution (Khomh and Gueheneuc, 2008; Wendorff, 2001). Khomh and Gueheneuc (2008) chose quality attributes such as expandability, simplicity, reusability, learnability and understandability. Application of patterns can result in overengineered designs whose maintenance cost of the change and removal of patterns can be high (Wendorff, 2001). The studies applied the design patterns of Gamma *et al.* (1995) whose pattern classification is within the middle ground of pattern granularity, above language specific idioms and below architectural patterns. Layering design patterns can introduce complexity. The area of this research, the asset pipeline, has a pattern classification of an architectural workflow and contains coarse grained abstract components mitigating the production of overengineered designs.

Khomh and Gueheneuc (2018) researched the impact of patterns on quality over a four-year period operating under a series of workshops. They highlighted two levels of problems that directly affect quality attributes. Firstly, the need to bridge research activities with industrial practice, thus promoting scientifically validated results to practitioners and to study the impact of such practices. Secondly, the need to support the teaching of patterns especially to students with little or no experience in the application of patterns through reflective review of pattern instantiated code. However, bridging the communication between academic research and industrial practice is difficult within the video game industry owing to the limited access to field sites (O'Donnell, 2014).

Capture expertise and make knowledge accessible to non-experts

A pattern addresses a recurring design problem that arises in specific design context and presents a solution to it (Buschmann, 1996). Patterns document and detail existing, well proven important empirical design experience. They distil and provide a means to reuse the design knowledge gained by experienced practitioners (Gamma *et al.*, 1995). Those familiar with an adequate set of patterns operating within a domain "can apply them immediately to design problems without having to rediscover them" (Gamma *et al.*, 1995). Concerning software architecture Buschmann (1996) succinctly describes the benefits:

> "Patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practise. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architectures with specific properties."

The power of patterns is that documenting and sharing experience draws attention to the salient design lessons learnt. Every domain has a wealth of accumulated knowledge that can be shared with both novices and experienced practitioners working in new domains that could benefit their application (Rising, 1998).

Likewise, within the video game industry the game developers “need to reinvigorate their ability and desire to write, talk and share details of their work that they take for granted” (O’Donnell, 2014). In the thirty years of video game development the video game developers have not managed to share more broadly the workflows used within their workplace despite the demands that people entering the workplace are already familiar with them (O’Donnell, 2014). This is especially true of the asset pipeline, a workflow not well communicated within the video game industry and especially not to wider domains of reach. Capturing such expertise using the paradigms of patterns goes some way into providing this conduit.

Although design patterns are powerful knowledge sharing tools excessive use of design patterns is likely to result in designs whereby it is difficult to recognize the structure of the participating design patterns. Agerbo and Cornils (1998) uses the term “tracing problem” to categorize this issue. The solution as identified by Agerbo and Cornils (1998) is to have central repository of patterns with an associated vetting process allowing only those patterns admittance if they have been experimentally proven to benefit software engineers. The research of Agerbo and Cornils (1998) highlighted a need for a signposted repository and some twenty years later The Hillside Group (2021) has fulfilled this requirement with an online database of categorized patterns.

Provide a common vocabulary and understanding of design principles

Patterns provide a new, common vocabulary and understanding for design principles (Gamma *et al.* 1995). A pattern name, if chosen correctly, becomes part of a widespread design vernacular, carrying with it all the details in the actual pattern description. Designers can discuss the design using the shared vocabulary of a pattern language, removing the need to explain the solution to a particular problem with a lengthy and complicated description (Coplien and Schmidt, 1995; Helm, 1995). Instead a designer can refer to a pattern name and explain further the parts of a solution that correspond to the components of the pattern (Buschmann *et al.*, 1996).

For example, the model-view-controller (MVC) pattern originally developed for desktop use (Fowler, 2003) is now widely adopted for use in web technology frameworks such as React. The React framework maps the following functional components onto the model-view-controller pattern; a presentation layer of Controller and View React Components and a UI Agnostic Data Model. Relating such underlying components to their corresponding pattern components can assist designers and implementers of contemporary React applications.

Patterns ease and improve design communication amongst designers on a team, amongst designers on different teams and between designer and themselves. Furthermore, a common vocabulary can improve communication amongst the roles of developers, managers, marketing strategists and others. The common vocabulary enables people in less technical roles to better understand the design architecture without becoming hindered in the technical details; different groups will have a better understanding of the design (Coplien and Schmidt, 1995; Helm, 1995).

Agerbo and Cornils (1998) claim the rapid proliferation of design patterns has hindered the benefit of a common vocabulary gained from using design patterns. They state the increase in the number of design patterns makes a common vocabulary unmanageable. This is certainly evident within the context of today. A systematic mapping of the literature referencing patterns has recognized over 2,775 papers (Bafandeh Mayvan *et al.*, 2017). Agerbo and Cornils (1998) propose mitigating this problem by further partitioning design patterns into the classifications of Fundamental Design Patterns, Language Dependent Design Patterns and Related Design Patterns. Although such elementary categorization benefited the body of patterns in 1998, a central repository has now been formulated. The Pattern Language of Programming (PLoP) conference was founded to create a new body of literature composed of solutions to ordinary yet by no mean simple problems found in software (Coplien and Schmidt, 1995).

Identify, name and specify abstractions above low-level implementation details

Patterns enable designers to communicate at a higher or more abstract level. Patterns identify and specify abstractions that are above the level of single classes and instances, or of components (Gamma *et al.*, 1995). Typically, a patten describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation. All components together solve the problem the pattern addresses.

The pattern abstraction simplifies and improves documentation. The use of pattern names in documentation carries associated information and produces a more approachable, simpler and maintainable document. Design detail is reduced since the pattern name associated with the solution indirectly provides it (Rising, 1998). Furthermore, the novice reader of such documentation understands the system more quickly when it is documented and referenced with the pattern it uses (Buschmann *et al.*, 1996; Vlissides, 1998).

Serve as a reusable building block for system development

Patterns support the construction of software with defined properties. Patterns provide a skeleton of functional behaviour and therefore help to implement the functionality of your application in the form of frameworks. The software framework provides generic functionality that can be adapted to the identified problem (Buschmann *et al.*, 1996). Generation of an asset pipeline pattern language is the first stage to such a framework implementation. Future research could extend the pattern language into a concrete framework suitable for specialization in a number of runtime visualization domains.

The choice of pattern to apply under a certain context may be a difficult process for a system developer. To assist in this process Gueheneuc and Mustapha (2007) devised a recommender system to help the user in choosing among the twenty-three patterns of Gamma *et al.* (1995).

Enable domain independent solutions

Patterns enable reuse of architecture and design patterns should be platform independent. This means that an architecture and design using patterns should also be independent of the platform and remain stable even with major changes to the underlying platform (Coplien and Schmidt, 1995; Helm, 1995). The asset pipelines as viewed in the video game industry are highly specific to each project because there may be particular aspects of the pipeline that are used for one project, however, not used by another (O'Donnell, 2014). This research proposes to formalize the asset pipeline into a pattern language, an organized set of patterns, whereby the designer makes a conscious decision on which component to include based on the context, problem and forces at play.

Provides a predefined schema or format for specifying the problem, solution and context under which they operate

Patterns deliver a common vocabulary and shared understanding for design concepts (Rising, 1998). A shared vernacular eases the reuse of architectural knowledge and artifacts. Every pattern has a name; naming a pattern immediately increases the design vocabulary of a practitioner. The pattern name is a handle which can be used to describe a design problem, its solutions and consequences in a word or two.

2.2.3 Pattern Form

Patterns are a literary form. The form serves a purpose: to introduce the reader to a problem, to describe the context of where the problem occurs and to analyse the problem and to present and elucidate a solution for the reader to apply (Coplien and Schmidt, 1995). Pattern authors vary the form their pattern material is presented and do not always write their patterns in the same styles. The pattern form is a mechanism to combine the same essential sections into a pattern structure for ease of understanding and comparison purposes. The choice of the style is high subjective and according to Vlissides (1998, p.147), "no one form suits everybody." The form depends upon both the target audience and the domain of application whereby certain sections are more applicable.

Discourse of pattern form is usually related to the definition of patterns. The prevailing definition: a solution to a problem in context evokes the primary elements of the form (Coplien and Schmidt, 1995). Some of these forms contain more elements than others; however, all contain the basic components establishing the pattern definition. That is the name, problem statement, context, description of forces and solution. It is important that the pattern is clear and complete, that it stands on its own, since the pattern author will not be available to answer questions.

Historically several pattern forms can be observed in literature: Alexandrian (Alexander *et al.*, 1977), Gang of Four (Gamma *et al.*, 1995), Coplien (Coplien and Schmidt, 1995) and the Portland form amongst others. In this section the most common classification schemes are described starting with the pattern forms, drawing directly on the insights of the originators.

Christopher Alexander created the first pattern form in the second of three books titled "A pattern language: towns, buildings, construction" (Alexander, Ishikawa and Silverstein, 1977). The sections of an Alexandrian pattern are not string delimited. The major syntactic structure is a "Therefore" statement immediately preceding the solution. The Alexandrian form usually contains the following elements: a clear statement of the problem, a discussion of the forces, the solution and the rationale.

Alexander (1977, pp.x-xiv) writes of the pattern form:

> "Each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern … after the picture… an introductory paragraph … sets the context … explaining how it helps to compete certain larger patterns. Then there are three diamonds … After the diamonds … a headline, in bold type … gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. It describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution – the heart of the pattern … in the stated context. This solution is always stated in the form of an instruction – so that you know exactly what you need to do, to build the pattern. Then … a diagram … [of] the solution …
> After the diagram another three diamonds … And … a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern….
> Each solution is stated in such a way that it gives the essential field of relationships needed to solve the problem, but in a very general and abstract way – so that you can solve the problem for yourself, in your own way … we have tried to write each solution in a way which imposes nothing on you. It contains only those essentials which cannot be avoided if you really want to solve the problem. In

this sense, we have tried, in each solution, to capture the invariant property common to all [solutions of] the problem"

The rationale behind the choice of form as described by Alexander (1977, p.xi):

"There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of all 253 patterns as a whole, as a language, within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it."

The canonical form used by Alexander and its constituent parts are detailed in table 2.5.

Section	Description
Photograph	Illustrates an archetypal example of the pattern.
Introductory Paragraph	Sets the context of the pattern within the scope of larger encapsulating patterns.
The Headline (bold type)	The essence of the problem in one or two sentences.
The Body	Often consisting of many paragraphs; describes the background of the pattern and range of ways the pattern manifests itself in the space.
The Solution (bold type)	Stated in the form of the instruction which is required to solve the stated problem in the stated context.
Encapsulated Patterns	A paragraph linking all the smaller patterns in the language which are required to complete this pattern.

Table 2.5: Alexandrian pattern form

The pattern reads as conventional prose; inviting a reading sequentially from start to finish. The overall structure is course and consists of narration with no fine implementation level detail.

The design patterns of Gamma *et al.* (1995) have a substantially different structure from the Alexandrian pattern form. By comparison the design patterns of Gamma *et al.* (1995) are highly structured and contain more information detail; with an average design pattern contained within ten pages compared to three pages for an Alexandrian counterpart. A Gamma *et al.* (1995) pattern has four essential elements: the pattern name, the problem, the solution and the consequences.

The name, consisting of a word or two, is extremely important as it succinctly describes the design problem, its solutions and consequences. According to Gamma *et al.* (1995) discovering a concise name is one of the most difficult stages of developing a catalogue of patterns. For Gamma *et al.* (1995) the name of a pattern is a noun phrase describing the central design decision, for example, "Chain of Responsibility"; the participants, for example, "Observer"; or the function, for example, "Abstract Factory.

The problem explains the specific design problem and the context within which it is found (Gamma *et al.*, 1995). The problem may take many forms: it may describe class or object structures leading to design inflexibility or it may describe how to represent algorithmic instructions as a set of objects. The problem often includes the preconditions that must be satisfied before application of the pattern.

The solution illustrates the configuration of elements in terms of their relationships, responsibilities and collaborations that compose the design. The solution provides detailed information on how to implement the pattern, including sample programming code – the Alexandrian pattern language seldom deals with construction details on a comparable level.

The consequences describe the benefits and costs of applying the pattern. They provide the designer with the necessary information to evaluate design alternatives. Gamma *et al.* (1995) highlights the following consequences: space and time trade-offs, language and implementation issues, impact on the flexibility of the system, extensibility and portability. Gamma *et al.* (1995) emphasize the importance of re-usability as one of the overriding forces in the creation of a common design structure.

The highly structured Gang of Four pattern form is detailed in table 2.6.

Section	Description
Pattern Name and Classification	Provides the succinct essence of the pattern. The name becomes part of the design vocabulary. The classification details the purpose of the pattern, either creational, structural or behavioural.
Intent	A description of the design issue or problem the pattern addresses and the rationale for its use.
Also Know As	The other well-known names for the pattern.
Motivation	A detailed use case scenario illustrating the design problem and a solution in the form of the structure of classes and objects involved.
Applicability	The situations in which the design pattern can be applied. Examples of poor designs the pattern addresses.
Structure	A static graphical representation of the classes and relationships involved in the pattern using a derivative of the Object Modelling Technique (OMT). Dynamic interactions are represented using interaction diagrams.
Participants	The classes and objects forming the structure of the design pattern.
Collaborations	Details of any collaborations required to fulfil the responsibilities of the participants.
Consequences	The results and trade-offs of applying the pattern. Details of the aspects of the system structure that can vary independently.
Implementation	A discussion of the drawbacks, recommendations and techniques that should be considered when implementing the pattern. Any language specific issues are highlighted.
Sample Code	C++ or Smalltalk code fragments illustrating how to implement the pattern.
Known Uses	At least two examples of real-world scenarios utilizing the pattern. The choice of scenarios must extend into differing domains.

Table 2.6: Pattern template format as identified by Gamma *et al.* (1995, pp.16-18)

The Coplien form is essentially identical to the Canonical Form. Coplien declares the essence of the Alexandrian form should be present regardless of style; there should always be a clear definition of the problem, the forces, and the solution (Coplien, 1995). The pattern form of Coplien (1995) delineates patterns sections with sections headings and includes the elements detailed in table 2.7.

Section	Description
Pattern Name	Coplien focuses on good names. Names suggest the problem, the solution, the resulting context or intent. Taking inspiration from the Alexandrian form, Coplien favours nouns for the pattern name, although short verb phrases can also be used.
Problem	The problem is stated as a question or design challenge, following the Alexandrian problem definition.
Context	The context in which the problem may arise, following the Alexandrian introductory paragraph.
Forces	Forces describe the pattern design trade-offs. This corresponds to the in-depth description of the Alexandrian pattern format.
Solution	The solution provides a mechanism to solve the problem. A diagrammatic representation of the pattern may accompany the solution.
Rationale	The important principles behind the driving force of the pattern.
Resulting Context	The pattern resolution resolves the forces stated, however, will manifest in a set of forces remaining unresolved.

Table 2.7: Coplien pattern form

The Pattern Languages of Program Design form extends the basic scheme of Alexander (1979) with additional aspects closely related to the one Gamma *et al.*, proposes (Gamma, 1995). The intention of the pattern form is to support the understanding, comparison, selection and implementation of patterns within a given design situation.

The pattern form of the Pattern Languages of Program Design is detailed in table 2.8.

Section	Description
Pattern Name	The name of the pattern, conveying its essence (Gamma et al)
Rationale	The motivation for the pattern; collectively the design issues and the problems it addresses.
Applicability	A rule stating when to use the pattern
Classification	Classification of the pattern according to the schema developed.
Description	Description of the participants and collaborators in the pattern as well as their responsibilities and relationships among each other.
Diagram	A graphical representation of the pattern's structure.
Dynamic Behaviour	A scenario-based illustration of the typical dynamic behaviour of a pattern.
Methodology	The methodological steps for constructing the pattern.
Implementation	Guidelines for implementing the pattern. When appropriate, the guidelines are illustrated with pseudocode and a concrete C++ code example.
Variants	Description of possible variants of the pattern.
Examples	Examples for the use of the pattern.
Discussion	The constraints of applying the pattern.
See also	References to related patterns.

Table 2.8: Pattern Languages of Program Design pattern form

2.2.4 Pattern Classification

The pattern community has converged on a layered granularity for pattern categories. The layered schema differentiates levels of abstraction for problem and scope. At the lowest level of granularity are the language specific patterns known as idioms. Idioms form solutions to the concrete realization and implementation of particular design issues. They have the narrowest scope and according to Coplien and Schmidt (1995, p.329):

> "An idiom describes how to implement particular components (parts) of a pattern, the components functionality or their relationships to other components within a given design. They are often specific for a particular programming language."

In 1992 James Coplien published the widely recognized book "Advanced C++ Programming Styles and Idioms" (Coplien, 1992). The idioms presented by Coplien are programming language specific solutions to a recurring functional problem. Coplien (1992, p.1) identifies idioms that "make C++ more expressive, and language styles that give software its structure or its 'character'." Although C++ code can be implemented without the use of idioms, they provide efficiency, expressiveness and aesthetic value. Recasting of idioms in pattern form often makes them more accessible to moderately experienced C++ programmers.

The approach Coplien practises is to use abstract data types, inheritance, object composition and object-oriented techniques to move beyond the orthodox idioms native to C++ to more advanced idioms that go beyond the functionality of the language. Although Coplien primarily concentrates on exposing object-oriented techniques other conventional paradigms are also embraced. One such idiom Coplien (1992, p.58) formulates is the "reference counting idiom", an approach not directly supported in the C++ language, that implements a form of reference counting manipulated through the handle class. The idiom encapsulates the memory allocation and reclamation of objects. Thus, introducing efficiency when copying shared objects. Often the same idiom looks different in distinctive languages, and sometimes an idiom that is useful in one language does not transfer into a different language flavour. The "reference counting idiom" for example is a technique incorporated in the Java language under the name of the garbage collector, and hence is incorporated into the core Java Development Kit (JDK).

The design patterns as catalogued by Gamma et al. (1995) are categorized within the middle level of the hierarchical structure. Design patterns are not language specific and can be implemented in a variety of languages. Gamma *et al.* (1995) provide a definition of their catalogue of design patterns:

> "A design pattern describes a basic scheme for structuring subsystems and components of a software architecture, as well as the relationships between them. It identifies, names, and abstracts a common design principle by describing its different parts and their collaboration and responsibilities."

Design patterns assist in the creation of components or small groups of components. Instances of design patterns from the Gang of Four catalogue include Singleton, Adapter, Bridge and Iterator (Gamma *et al.*, 1995).

An architectural pattern defines the structure of a design solution at the highest level. They provide the overall direction of the system and express a fundamental structural organization schema for software systems. An architectural pattern provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (Buschmann *et al.*, 1996). In effect the architectural pattern dictates the subsystems designed. The category of patterns known as "From Mud to Structure" includes patterns to solve the issue of decomposing an overall system into individual sub-components (Buschmann *et al.*, 1996).

2.2.5 Pattern Writing

Numerous metaphors have been proposed and used to describe the process of discovering and documenting patterns including fishing, hunting and harvesting a pattern (Rising, 1998). However, these phrases and nomenclature indicate too much randomness in the discovery process. According to Rising (1998, p.87) "it takes a great deal of effort to discover and document a pattern." In the fields of palaeontology and archaeology the notion of "mining" is used to define the process of digging to discover elements. The pattern community has settled on the mining metaphor and uses the term "pattern mining" or "patterns mining" to indicate the process of pattern discovery (Gabriel, 2010). "Great pattern writers," Richard Gabriel said, "are miners, they create nothing but the wonderful explanation" (Rising, 1998, p88).

Literature discussing the pattern mining field can be classified into two groups of approach: automatic programmatic mining techniques and manual application of pattern mining methodologies.

The automatic pattern detecting technique adopted by Zhang and Chen (2015) uses a source code pattern detection technique based on subgraph isomorphism to recover instances of the twenty-three patterns devised by Gamma *et al.* (1995). Although the approach detects pre-existing design patterns instances embedded in source code the research does not mine for new pattern abstractions. Nevertheless, recovery of the instances of design patterns from the source codes of software systems can assist the understanding of the systems and processes of re-factoring them. The work of Zanoni *at al.* (2015) applied a machine learning methodology to the problem of design pattern detection based on five patterns forming a subset of the catalogue identified by Gamma *et al.* (1995). The detection results benefit from a learned criteria which cannot be formally expressed as exact matching rules as used by Zhang and Chen (2015). Correct and incorrect examples are added to the training set over time.

FixMiner (Koyuncu *et al.*, 2020) is a pattern mining tool used to extract relevant fix patterns for automated program repair. With a goal of inferring separate and reusable fix patterns that can be leveraged in other patch generation systems the approach discovered patterns by analysing 11,416 patterns. The implementation of an automated repair pipeline mined patterns relevant for generating correct patches for twenty-six bugs in the Defects4J benchmark. These instantiated correct patches corresponded to eighty-one per cent of all plausible patches generated by the tool.

With the manual application of pattern mining methodologies, the patterns are discovered through the process of "pattern mining" existing successful designs (Rising, 1998). According to Buschmann *et al.* (1996, p.376) in the domain of use:

> "…find at least three examples where a particular recurring design or implementation problem is solved effectively by using the same solution schema. The examples could all be from different real-world systems, and all the systems should have been developed by different teams."

Subsequently the solution schema is abstracted from the specific details of its concrete applications. The problem and solution the schema addresses are described along with forces associated with the schema. At this point the solution schema is declared a pattern candidate (Buschmann *et al.*, 1996).

Although a pattern writer observes a solution many times in a domain Wellhausen and Fießer (2011) asserts it is difficult to write a pattern linearly in terms of the context, problem, forces, solution and consequences, figure 2.10.

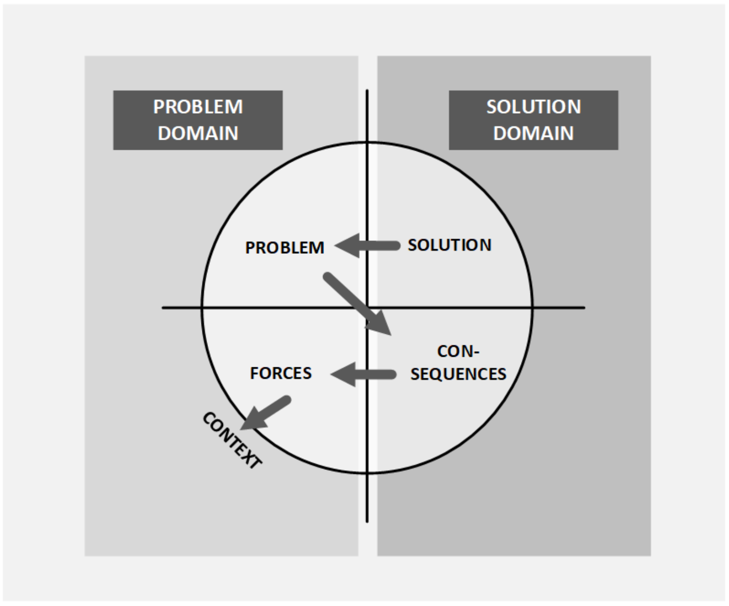


Figure 2.10: Pattern writing process according to Wellhausen and Fießer (2011)

Wellhausen and Fießer (2011) suggest starting with the solution and posing the following questions: What in particular makes it something special? What does it include and what does it not include? How can it be distinguished from similar ideas and what examples do you know? The pattern author must identify the aspects that describe the solution best. Subsequently the author transitions from the solution to the problem it solves, ensuring the problem statement matches the solution. The consequences of applying the solution, both in terms of the benefits and liabilities are defined. The forces restrict the available solutions to a single one and must match the consequences. The force is the characteristic that makes the problem difficult and the corresponding consequence determines how the force is resolved by the solution. Ideally a one to one (1:1) relationship between every force and consequence should exist to provide a balanced pattern. Finally, the context under which the problem exists is specified and a name is provide for the pattern (Wellhausen and Fießer, 2011).

To assist the pattern author with the process of pattern writing, a series of meta-patterns providing a pattern language for pattern writing has been developed (Meszaros and Doble, 1997). "Although there is no single right way to write patterns," Meszaros and Doble (1997) writes, "this pattern language describes and demonstrates a collection of writing practices that have been observed to be particularly effective." The pattern language is not prescriptive but rather describes the desired result providing the pattern author with the freedom to employ different techniques. The pattern language is grouped into five sections: context-setting patterns; pattern structuring patterns; pattern naming and referencing patterns; patterns for making patterns understandable and pattern language structuring patterns.

The pattern format and "adhering to a structure" is important since all patterns should be described in a uniform way (Coplien and Schmidt, 1995). As noted in section 2.3.4 several pattern formats have evolved since the creation of the Alexandrian pattern form and according to Vlissides (1998, p.147) "no one form suits everybody". There is a general agreement that a pattern is "a solution to a problem in context" (Alexander, 1979) and Vlissides (1998, p.147) expands this definition to "a pattern is a structured exposition of a solution to a problem in context". Vlissides emphasizes patterns have recognizable parts that guide their application and comparison. The parts include a name, a statement of the problem, the context and justification of its solution, and the concrete solution. Depending upon the domain of use and granularity of the solution these fundamental elements can be decomposed into more focused treatments. The proceedings of the Pattern Languages of Programs (PLoP) conferences include diverse variations on these themes.

An iterative approach to writing patterns is recommended. John Vlissides (1998) in his work "Pattern Hatching, Design Patterns Applied" describes "iterating tirelessly" as one of the seven habits of effective pattern writers. The pattern catalogue of object-oriented design patterns (Gamma *et al.*, 1995) went through such an iterative process with reviewers offering suggestions and improvements to the authors on the internet. Likewise, the authors of the Pattern-Oriented Software Architecture series (Buschmann *et al.*, 1996) setup an FTP site and internet discussion group for reviewing and discussing the pattern systems they proposed. An iterative approach provides new insights into the problem domain within each cycle (Wellhausen and Fießer, 2011).

A review process known as the writers' workshop was introduced by Richard Gabriel at the first Pattern Languages of Programs (PLoP) in 1994 and has been used thereupon. Borrowing ideas from general writing community, especially poetry, the writers' workshop provides a place for pattern authors to review and share comments on the writing of others in the group. In comparison to a direct document-review process the emphasis is on the communication of ideas. All the participants in a writers' workshop are fellow pattern authors. The workshop is led by a moderator in a highly structured format resulting in a safe and respectful place to have new patterns reviewed.

2.2.6 Games Technology Patterns

Taking inspiration from the patterns of Gamma *et al.* (1995) a single piece of literature devoted to a pattern-oriented approach to computer games development was produced in 2014. The book "Games Programming Patterns" (Nystrom, 2014) identifies patterns to solve the design challenges commonly encountered in computer game development. Those functional challenges as identified by Nystrom (2014) are:

1) Sequencing and time management – Computer games are real-time systems and must respond to external stimuli in both a timely manner and in the correct order of execution.
2) Short development times – The development life-cycle for computer game production is highly compressed and software developers must iterate over features sets without disruption to the core codebase and hindering other team members.
3) Behaviour interaction – Computer games consist of multiple entities and their interactions are derived from strict laws of behaviour. The simulated behaviour must be encapsulated in a design that is both easily understandable and can reflect future changes.
4) Performance – Computer games are soft real-time simulations (Gregory, 2014) and performance directly relates to usability and commercial success.

Nystrom (2014) identifies "good" software architecture as design that effortlessly accommodates future changes as if the entire system was crafted in anticipation of it. Decoupling components plays a key role in enabling maintainability and productivity since a loosely coupled system enables changes to one component without incurring the cost of changing other components. However, although decoupling facilitates rapid evolution of a software architecture, it incurs an impact in terms of the performance and real-time requirements of a game system. Flexibility in the form of abstraction is at odds with the optimization techniques relying on concrete limitations of a game system. Ultimately, Nystrom (2014) recognizes a careful balance must be achieved between incorporating the flexibility of object-oriented paradigms and realizing real-time performance and this entirely depends upon the context of the game system.

Nystrom (2014) identifies several issues surrounding the architecture of computer games whilst being employed at Electronic Arts and aimed to communicate their solutions through the provision of a catalogue of best practises in the form of a series of patterns. Six of the Gamma et *al.* (1995) patterns were revisited relating their context to that of game programming: command, flyweight, observer, prototype, singleton and state. Table 2.9 identifies the patterns along with their description.

Section	Description
Command	Encapsulates a request as an object, enabling a queue of commands. Examples include "undo" functionality in a game level editor and sending intelligent commands to a game entity.
Flyweight	Supports a large number of fine-grained objects using instancing. For example, the creation of a forest of similar trees.
Observer	Defines a one-to-many dependency between objects in a system. A state change of one object causes a notification to all interested parties. An achievement system in a game can be implemented using the observer pattern with little impact to the surrounding software architecture.
Prototype	Use a prototypical instance to specify the types of objects to create new objects by cloning the prototype. For example, creating multiple instances of a similar game entity
Singleton	Provide a global point of access to a single instance of a class. The singleton pattern can be used to access components of a game engine such as the audio system.
State	When an objects internal state changes allows an object to alter its behaviour. The object changes its class. Behaviour of a player or entity in a game can be implemented using a state machine.

Table 2.9: Gamma *et al.* (1995) patterns interpreted within the context of game programming

Nystrom (2014) presents thirteen original design patterns grouped into four categories: sequencing patterns, behavioural patterns, decoupling patterns and optimization patterns. The patterns, their category and description are detailed in table 2.10.

Category	Name	Description
Sequencing	Double Buffer	Causes a series of sequential operations to appear instantaneous or simultaneous.
	Game Loop	Decouple the progression of game time from user input and processor speed.
	Update Method	Simulate a collection of independent objects by telling each to process one frame of behaviour at a time.
Behavioural	Bytecode	Give behaviour the flexibility of data by encoding it as instructions for a virtual machine.
	Subclass Sandbox	Define behaviour in a subclass using a set of operations provided by its base class.
	Type Object	Allow the flexible creation of new "classes" by creating a class, each instance of which represents a different type of object.
Decoupling	Component	Allow a single entity to span multiple domains without coupling the domains to each other.
	Event Queue	Decouple when a message or event is sent from when it is processed.
	Service Locator	Provide a global point of access to a service without coupling users to the concrete class that implements it.
Optimization	Data Locality	Accelerate memory access by arranging data to take advantage of CPU caching.
	Dirty Flag	Avoid unnecessary work by deferring it until the result is needed.
	Object Pool	Improve performance and memory use by reusing objects from a fixed pool instead of allocating and freeing them individually.
	Spatial Partition	Efficiently locate objects by storing them in a data structure organized by their positions.

Table 2.10: The Game Programming Patterns as identified by Nystrom (2014)

Nystrom (2014) uses a consistent pattern format as detailed in Table 2.11.

Section	Description
Pattern Name and Category.	The name becomes part of the game design vocabulary. The category identifies the purpose of the pattern, either sequencing, behavioural, decoupling or optimization.
Intent	A description of the pattern within the context of the problem it solves.
Motivation	Provides an example use case problem that will form the foundation example for the remainder of the pattern definition.
Pattern	The formal essence of the pattern abstracted from the example problem.
When to Use It	Rules and scenarios for when to apply the pattern and when to avoid its application.
Keep in Mind	The consequences and risks of using the pattern.
Sample Code	The full C++ implementation of the pattern.
Collaborations	Details of any collaborations required to fulfil the responsibilities of the participants.
Consequences	The results and trade-offs of applying the pattern. Details of the aspects of the system structure that can vary independently.
Implementation	A discussion of the drawbacks, recommendations and techniques that should be considered when implementing the pattern. Any language specific issues are highlighted.
Design Decisions	Implementation of a pattern varies according to the exact context and the design decisions explore the different choices to consider when applying the pattern.
See Also	A list of related patterns along with examples of existing frameworks and components incorporating the pattern in their solution.

Table 2.11: Pattern format as identified by Nystrom (2014)

The patterns of Nystrom represent a set of functional patterns for use in the actual underlying design and implementation of a video game. A core set of the catalogue is based upon a collection of patterns published by Gamma *et al.* (1995). As a set of patterns, it is low-level and does not extend to encapsulate the workflow of the asset pipeline. However, their relevance in games technology remains and they are extensively used in academic institutions in the teaching and delivery of low-level video game development.

2.3 Summary and Evaluation

The video game industry is a young activity and is shrouded in secrecy (section 1.2). The amount of literature related to the theme of the asset pipeline and associated technologies is relatively small in comparison with other domains of research. The survey of literature specific to the asset pipeline is encompassed in the following main themes: game engines, game studies, game development tools, games technology surveys, video game post-mortems and the video game asset pipeline itself. The qualitative data analysis software NVivo (2021) was utilized as a central repository for such literature. Keyword searches using "asset pipeline" and the associated synonyms of "content pipeline" were used to codify and retrieve related literature. This automatic process was essential when querying the 209 issues of the industry focused Game Developer Magazine (2021) for video game post-mortems referencing issues with the asset pipeline. This is undoubtedly a methodology that will be transferred to future research.

The asset pipeline as described in literature provides an abstract framework overview of the workflow utilized in the video game industry. This is owing to two factors. Firstly, the specifics of the asset pipeline are adapted to the given requirements of the game content. Secondly, the video game industry is reluctant to divulge details of their internal toolsets for the reasons outlined in section 1.2. The workings of the asset build process and concrete examples of individual asset build nodes are obscured beyond examples found in the related literature of three-dimensional mesh optimization techniques (Aguilar, Alexánder, and Saúco, 2020).

Nevertheless, this level of abstraction is fortunately aligned to the formulation of a high-level pattern language. Manual pattern mining and writing requires at least three examples of where a particular recurring design or implementation problem is solved effectively by using the same solution schema (Buschmann *et al.*, 1996). This requirement has been satisfied with state of the art of the asset pipeline as represented in academic literature and the analysis of two concrete scenarios of the game engine asset pipeline (Unity) and the asset pipeline as found in the XNA build framework. The two use case scenarios provided a glimpse into the inner workings of the asset pipeline; the game engine asset pipeline functionality exposed through user-interface configuration and the XNA build framework directly manipulated by a developer using a set of C# interfaces and classes.

The asset pipeline "… is the path that all game assets follow, from conception until they can be loaded into the game" (Llopis, 2004). Video game content is designed using Digital Content Creation (DCC) tools and consists of source assets. Source assets being everything that is not the algorithm of the video game. Source assets are extracted from the DCC tool using a standard exporter or a custom export plug-in. The choice of export format is important; an intermediate asset format decouples the originating DCC tool proprietary format from the final format as expected by the final game engine runtime. Any changes in the runtime data-structures simply require propagation into the convertor that produces the final game ready assets. The asset build process is tasked with transforming and compiling the source assets into their final realized form as expected by the runtime.

The solution schema, as abstracted from the shared characteristics of the concrete examples identified in the literature review, is illustrated in figure 2.11.

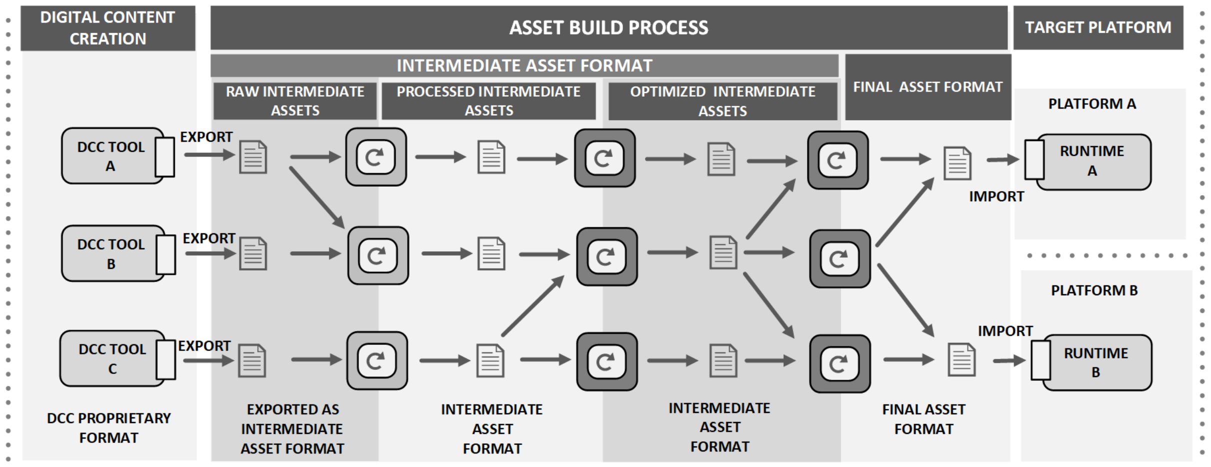


Figure 2.11: Asset pipeline workflow solution schema

Unfortunately, as with all research, the literature review represents a snapshot in time. The model of workflow identified in literature focuses on a "push" workflow whereby the build process is instigated by the designer. However, the video game industry moves at a fast rate. The technology cycle of a generation of video game hardware in the form of game consoles evolves approximately every five years. The required toolset software, including the asset pipeline, continually evolves to keep up with the demand for increased content as defined by source assets. Efficiency in the asset pipeline has dictated a move towards a "pull" model whereby changes are automatically pulled as and when they occur by the designer operating in the DCC tool. This approach has not been formalized in literature and consequently has not been propagated into the resulting asset pipeline pattern language - a limitation of this study that can be explored in future research.

Since their inauguration by the architect Alexander (1979) and their application in the world of software engineering (Gamma *et al.*, 1985) the recurring definition of a pattern is a solution to a common problem operating within a certain context. Likewise, within the video game industry the asset pipeline is a template solution to the common occurring problem of converting a large number of source assets into their final version as expected by the runtime.

Alexander (1975) states the application of patterns within the domain of building and architecture results in a "quality without a name." Subsequent application of the philosophy of patterns as found in literature in further fields such as software engineering (Gamma *et al.*, 1995; Buschmann *et al.*, 1996; Coplien, 1992) has reenforced this mantra. However, a growing number of studies have critically researched the role of design patterns with regard to quality with the result that design patterns do not always present themselves with such a quality attribute. Wydaeghe *et al.* (1995) concluded with their study of a subset of the patterns of Gamma *et al.* (1995) that not all patterns have a positive impact in the areas of reusability, modularity, flexibility and underlying implementation. Wendorff (2001) state the use of patterns in commercial systems can result in overengineering and high maintenance cost. Khomh and Gueheneuc (2008) cautioned the use of patterns within software maintenance and evolution since they may not necessarily promote reusability, expandability and understandability. Khomh and Gueheneuc (2018) highlight the impact of a pattern becomes apparent if the gap can be bridged between research activities and concrete industrial practice.

Patterns do capture expertise and make knowledge accessible to non-experts (Buschmann, 1996). Those familiar with an adequate set of patterns operating within a domain "can apply them immediately to design problems without having to rediscover them" (Gamma *et al.*, 1995). The proliferation of such knowledge exchange is prohibited in the video game industry and its associated technologies such as the asset pipeline (section 1.2). Agerbo and Cornils (1998) does however indicate such knowledge exchange is only accessible if a central repository of patterns exists with a quality gate for entry.

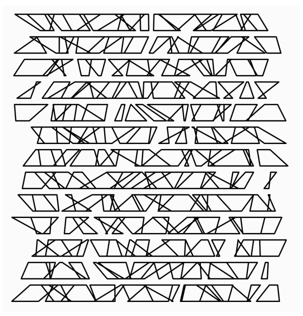
Patterns provide a new common vocabulary and understanding for design principles (Gamma *et al.* 1995). Something the video game industry has struggled with. The video game industry is relatively young, approximately fifty years old, and a set of agreed upon terms has been slow to develop. Furthermore, the asset pipeline requires interdisciplinary collaboration between the designers producing source assets and the software engineers developing the runtime. It is at this interdisciplinary fault line where communication barriers exist. Agerbo and Cornils (1998) claims the rapid uptake of design patterns counters this argument and hinders the benefit any such common vocabulary can provide. They indicate an increase in the number of patterns results in a vocabulary becoming unmanageable.

One attribute a pattern ethos provides is to provide a predefined schema or format for specifying the problem, solution and context under which they operate. Patterns take many literary forms and can read as conventional prose (Alexander, 1977) or can be highly structured in their layout (Gamma et al., 1995). One agreed upon characteristic is the inclusion of the primary elements of problem, context and solution based on the prevailing definition of a pattern: a solution to a problem in context. The asset pipeline is an architectural workflow and as such the derived format will not contain idiom level attributes such as source code.

Literature identifies two methodologies for mining patterns: automatic programmatic mining techniques and the manual application of pattern discovery. Pattern mining techniques as identified in literature (Zhang and Chen, 2015; Koyuncu *et al.*, 2020) require a source dataset of pattern instances. The mining tool of Koyuncu *et al.* (2020) analyzed 11,416 pattern instances. Access to concrete instantiations of the asset pipeline is not possible owing to the reasons outlined in section 1.2. The available asset pipeline instances are evident in the game engine and XNA build process, however, the underlying implementation source code is not available. These are in effect “black box" impenetrable to any automatic pattern mining algorithm. Thus, the available pattern mining approaches available for mining the asset pipeline are manual in approach. The spiral model is aligned to the Pattern Languages of Programs (PLoP) workflow to mine and derive the pattern language through an iterative sequence of shepherding and writers’ workshop.

Having surveyed the literature surrounding the domain areas of the asset pipeline and patterns the following chapter develops the asset pipeline pattern language incorporating the shared characteristics that all asset pipelines exhibit.

3 Asset Pipeline Pattern Language


Generative Pattern 3 – J Lear 2020

3.1 Introduction

Initially the methodology applied to formalize the asset pipeline pattern language is detailed. The spiral methodology is aligned to the framework of the Pattern Languages of Program (PLoP). Cycle one incorporates the derivation of pattern format, manual pattern mining based on the findings of the literature review and a phase of iterative pattern writing under the guidance of a shepherd. Subsequently, cycle two incorporates the feedback from the writers' workshop and refinement of the pattern language.

The outcomes present the findings based on the application of the methodology. The asset pipeline pattern language explores the transformation of digital assets from their basis in a Digital Content Creation (DCC) tool into a runtime environment within the general context of interactive real-time visualizations. The resulting four patterns derived are ASSET PIPELINE, DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS.

3.2 Methodology

The rigorous framework of the Pattern Languages of Program (PLoP) conference is followed resulting in the generation of an Asset Pipeline pattern language. The framework aligns with the iterative nature of the spiral model methodology. The PLoP process leads to the publication of the pattern language in the Association for Computing Machinery (ACM) International Conference Proceedings Series (ICPS) thereby communicating the pattern language for use within both academia and industry.

The PLoP framework involves three quality gates that must be fulfilled prior to the final pattern language being published. The acceptance of the first draft of the pattern language forms the first milestone. Consequently, the paper is assigned to a pattern shepherd who is an expert pattern writer and mentor. Over a period of four months and several iterations the shepherd asks questions and provides feedback on the quality of the pattern generation. A second draft, forming the second milestone, is reviewed for quality improvements and importantly the willingness of the pattern author to accept and incorporate the feedback of the shepherd. On acceptance the paper is discussed in the writers' workshop during the PLoP conference. During the writers' workshop the fellow pattern authors discuss the pattern providing constructive feedback for improvement. Based on the feedback during the workshop a final revised version is submitted, peer reviewed and published in the Association for Computing Machinery (ACM) proceedings.

The spiral model methodology, an iterative model, is aligned with the framework of the PLoP process encompassing two cycles as illustrated in figure 3.1. Cycle one incorporates the derivation of the pattern format; pattern mining for discovery of the individual patterns; pattern writing incorporating iterative feedback from a shepherd and finally completion of the cycle involving the milestone of acceptance to the writers' workshop.

Cycle two incorporates the feedback and findings from the writers' workshop and subsequent extensive refinement of the pattern language prior to acceptance for publication.

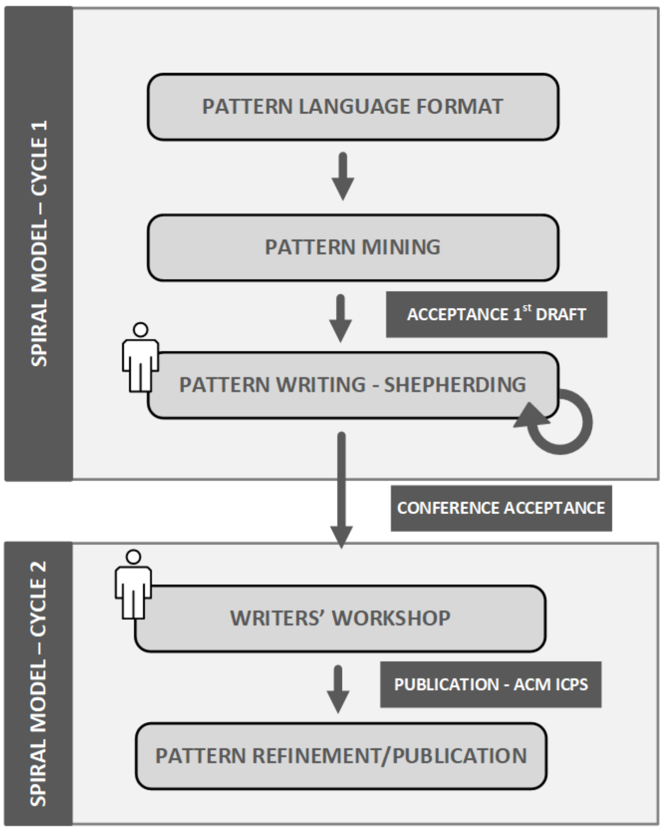


Figure 3.1: Pattern language formulation workflow

The PLoP conferences are international, and are sponsored by The Hillside Group. Development of the Asset Pipeline language was undertaken following the workflow of the European conference, EuroPLoP (2020), held in Kloster Irsee, Germany.

3.3 Outcomes

Application of the spiral model methodology detailed in section 3.2 results in the production of the outcomes detailed in the following sections; namely the pattern format and derivation of the pattern language over two iterative cycles.

3.3.1 Pattern Format

The pattern form provides the literary structure of the pattern and offers the reader consistency when viewing and comparing patterns within a pattern language. Alexander (1977) defines a pattern as a "solution to a problem in context" and complements this with a pattern form consisting of the pattern name, the problem statement, the context and the solution itself. The canonical form used by Alexander (1977) known as the Alexandrian form reads as conventional prose with an overall course structure. The pattern authors that followed took inspiration from the Alexandrian format and adapted the schema to suit their domain of use. Such styles include those of Gamma *et al.*, Portland, Coplien and Pattern Language of Program Design. A literary review of the primary pattern forms is provided in section 2.2.3 and is summarized in table 3.1.

Section	Alexandrian Form Alexander (1979)	Gang of Four Gamma *et al.* (1995)	Pattern Language of Program (PLoP)	Wellhausen and Fießer (2011) form
Name	Name	Name	Name	Name
Context	Introductory paragraph	Motivation	Applicability	Context
Problem	Headline in bold text	Intent and Motivation	Rationale	Problem in bold text
Forces	Between bold text	Motivation	Description, Rationale	Forces in bold text
Solution	After *therefore*	Applicability, Structure, Participants, Collaborations, Implementation	Dynamic Behaviour, Methodology, Implementation, Examples	Solution
Resulting Context	After the second row of three stars	Consequences	Discussion	Solution
Consequences	After the second row of three stars	Consequences	Discussion	Consequences, both benefits and liabilities
Rationale	Between bold text after the second row of three stars	Motivation	Rationale	Context, Problem
Known Uses	Between bold text after the second row of three stars	Known uses	Examples	Examples

Table 3.1: Pattern form comparison

As identified in section 2.3.5 patterns exist at various levels of abstraction; from language specific idioms, object-oriented design patterns to architectural patterns shaping the large overall structure of a system. Subsequently the level of granularity dictates the pattern form, inclusion and emphasis on individual sub-components.

The asset pipeline is a workflow consisting of a sequence of stages; an architectural pattern at a high level of abstraction. Implementation details, as found in idioms, are not be provided as these restrict the pattern to specific domains of use. Since one pattern form does not fit all (Vlissides, 1998) the pattern schema identified figure 3.2, was used during cycle one of the spiral model and outlined in table 3.2.

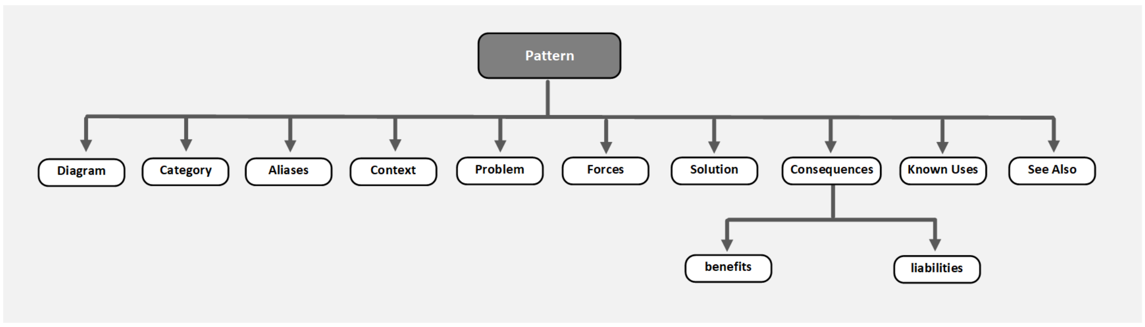


Figure 3.2: Asset Pipeline Pattern Schema

Section	Description
Pattern Name	The title of the pattern. The title enters the design vocabulary for the pattern.
Diagram	An illustration expressing the essence of the pattern.
Category	Pattern classification differentiating the different levels of abstraction. For example, architectural or component.
Aliases	A list of synonyms.
Context	The situation where the problem can be observed.
Problem	Explanation of the actual problem.
Forces	Describes why the problem is difficult to solve in terms of the conflicting forces. The opposing forces are delineated by the phrase *"However"*.
Solution	Outlines the recommended solution resolving the forces.
Consequences	Contains the associated benefits and liabilities of applying the pattern.
Known Uses	Application of the pattern in practice.
See Also	References to other associated patterns.

Table 3.2: Overview of the pattern format

3.3.2 Pattern Mining – Cycle 1

The first phase in the process of creating a pattern language is to mine best practices and knowledge. There are various methodologies to pattern mining. In "Holistic Pattern-Mining Patterns" Iba and Isaku (2012) propose a method in which members communicate in groups to write down rules or tips that are important about the subject onto sticky notes. In both "Mining by Interview" (Rising, 1998) and "Mining Interview Patterns" (Iba and Yoder, 2014) the research derived a method of mining by obtaining tips from the interviewee. In the approach identified by Akado *et al.* (2015) in "Five Patterns for Designing Pattern Mining Workshops" the pattern mining is performed under a workshop setting where participants reflect and recognize their lifestyle through dialogue. Akado *et al.* (2015) extended this process by creating a pattern language using a clustering method using the KJ method. Elements are mined and those that have similar attributes are clustered.

Patterns are "mined" based on abstracting the core elements of existing successful designs (Rising, 1998). According to Buschmann *et al.* (1996), "… find at least three examples where a particular recurring design or implementation problem is solved effectively by using the same solution schema." The literature review in section 2.2 forms the collective state of the art of the asset pipeline workflow as utilized in video game production and the details of which formed the seeds for pattern mining. In particular the asset pipeline workflow can be observed in the domains and scenarios of video game development (sections 2.1.1 to 2.1.5), game engine workflows (section 2.1.6) and build frameworks such as XNA (section 2.1.7). Regardless of location the asset pipeline process can be abstracted as "… the path that all the game assets follow, from conception until they can be loaded in the game." (Llopis, 2004, p.36).

Figure 3.3 illustrates the solution schema, the asset pipeline, abstracted from shared characteristics of the concrete workflows identified in the literature review. The source assets are exported from their Digital Content Creation (DCC) tool in an intermediate asset format. The assets are processed within the asset build process and converted into their final runtime format optimised for the target hardware platform in preparation for loading into the runtime game engine.

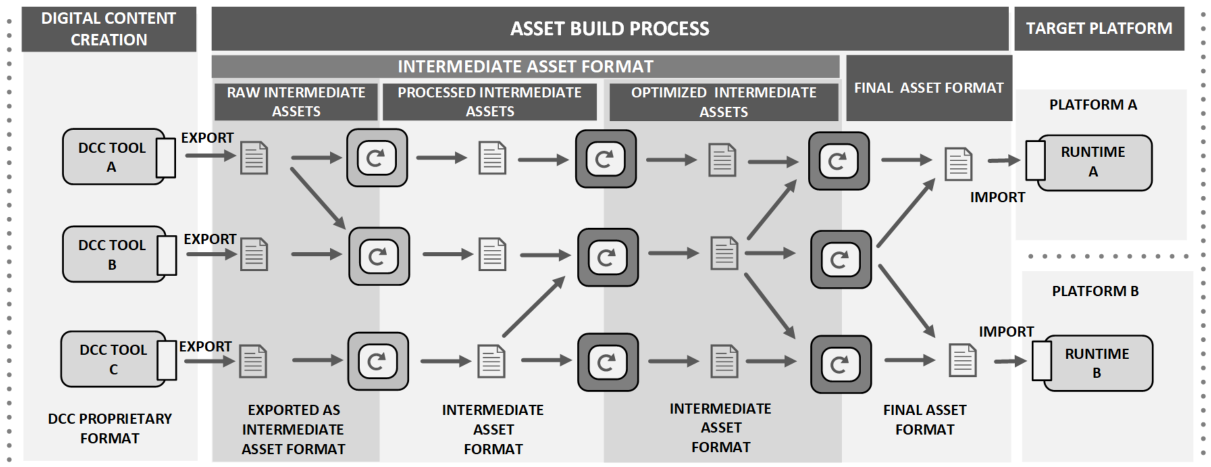


Figure 3.3: Asset pipeline workflow

Initially two abstract patterns were derived: the "asset pipeline" and the "intermediate asset format". The "asset pipeline" being the architectural level pattern and the "intermediate asset format" being a component pattern.

3.3.3 Pattern Writing – Cycle 1

The pattern writing method as observed by Wellhausen and Fießer (2011) was followed to elaborate the two patterns: ASSET PIPELINE and INTERMEDIATE ASSET FORMAT. The method incorporates the following sequential stages.

1. Initially the solution to the problem is documented.
2. The problem that leads to the solution is described answering such questions as: why is the solution relevant and what problem does the solution actually solve?
3. The results from applying the solution resolve into the consequences both in terms of benefits and liabilities.
4. The forces provide the reasons as to why the problem is difficult to solve and ideally the forces restrict the available solutions to simply a single one. A formal aspect to creating a formalized pattern structure is to ensure that every force is resolved by a consequence.
5. Finally, the context is derived containing the aspects and circumstances under which the problem appears.

The two resulting patterns are summarized in tables 3.2 and 3.3 and detailed in Appendix 3. The patterns were created during three months of the shepherding process forming one stage of the Pattern Languages of Program (PLoP) conference requirements. The shepherd guides the pattern author (sheep) into a more mature understanding of his or her pattern. Shepherds are individuals, with experience in pattern writing, assigned to the paper of the author who has an expressed interest in the domain of the pattern (Gabriel, 2010). To guide the shepherd through the process, the assigned shepherd follows the "The Language of the Shepherds" (Harrison, 1999). Although the shepherd is one stage of the process required for accepting a paper for the writers' workshop, the relationship is defined so the author is in full control of the paper. The recommendation of three iterations of communication involving comments to the author was undertaken.

The interaction between author and shepherd is by email. Given this, a novel approach of iterative development for assessing and incorporating the comments of the shepherd was performed. The approach took inspiration from the Scrum agile framework for delivering software products. The artifacts in this instance being the comments provided by the shepherd and the resulting action of incorporating changes into the pattern language. A Scrum backlog was instigated to ensure all comments were actioned and applied where necessary. The three iterations are detailed in Appendix 4.

Section	Overview
Pattern Name	ASSET PIPELINE
Category	Asset Pipeline Workflow
Also Known As	Content Pipeline
Pattern Summary	The asset pipeline solves the problem of digital content creators designing in DCC tools using high level concepts that require visualizing in a runtime requiring optimized versions.

Context	Designers require a workflow to assist them in viewing their digital content in an interactive real-time visualization.
Forces	Design feedback loop, incompatible source and final assets, source asset dependencies.
Problem	How to deliver interactive visualizations whereby the source assets are created in a high-level DCC tool and the visualization operates within an optimized runtime?
Solution	Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.
Consequences	Benefits: Design feedback loop efficiency, incompatible source and final asset format, source asset dependencies. Liabilities: Asset build process alternate path, flexibility of asset build node configuration.
Examples	Game engine asset pipeline, architectural visualization (ArchViz).

Table 3.2: Summary of the Asset Pipeline pattern

Section	Overview
Pattern Name	INTERMEDIATE ASSET FORMAT
Category	N/A
Also Known As	Intermediate File Format
Context	Designers are using a variety of DCC tools to create a visualization executing on multiple runtime environments.
Forces	Multiple source formats require identical asset conditioning, multiple runtime each expecting specific final formats, flexibility of asset build node configuration.
Problem	How to decouple the source format from the final asset format to introduce flexibility in the asset pipeline?
Solution	Use an intermediate asset format to decouple the source and final asset dependency.
Consequences	Benefits: Multiple source formats require identical asset conditioning, multiple runtime each expecting specific final formats, flexibility of asset build node configuration. Liabilities: Intermediate file format as a single container.
Examples	Video game entertainment industry.

Table 3.3: Summary of the Intermediate Asset Format pattern

3.3.4 Writers' Workshop

The collective Asset Pipeline patterns as formulated in cycle 1 were submitted to the writers' workshop and discussed in the European Conference on Pattern Languages of Programs (EuroPLoP, 2019). The pattern review follows a structured format, with the objective to acquire as much feedback for constructive improvement as possible. The writers' workshop follows the following format for the review of patterns (Buschmann *et al.*, 1996):

1. The pattern is discussed by a group of people that includes the pattern author and a group of reviewers familiar with the content of the pattern paper. The moderator is present to assist the participants follow the conventions of the workshop.
2. The author of the pattern selects and reads a summary paragraph of their choice from the pattern paper.
3. Two reviewers summarize the description presented by the author, from their personal understanding.
4. In three separate stages the positive points of the pattern are first discussed, followed by the deficiencies and finally other aspects of the pattern. A photograph taken during the proceedings is provided in figure 3.4.
5. During the review process the pattern author is listening and not interacting with the discussion group. The reviewers do not address the author directly. The reviewers discuss the pattern description as if the author is not present. The author may record or take notes about the discussion.
6. Upon completion of the discussion the author may clarify particular points made.
7. The author finalizes the session with a concluding comment on the discussion.

Figure 3.4: Writers' workshop discussing the proposed asset pipeline patterns

Following a two-hour review of the submitted Asset Pipeline patterns the following feedback was received from the fellow pattern writers.

Positive

- The two patterns, ASSET PIPELINE and INTERMEDIATE ASSET FORMAT, are comprehensive.
- The diagrams exhibit good graphic design and complement the pattern writing.
- The format of the pattern is easy to read.
- The style of the language used is clear.
- The problem and solution are detailed.
- Forces are extensive.
- The examples are comprehensive, interesting and illustrate pattern usage.

Deficiencies

- The Asset Pipeline pattern is extremely extensive in terms of technical detail.
- Decompose the Asset Pipeline pattern into further component patterns and create a sequence of patterns.

Although, the patterns were extremely well received the ASSET PIPELINE pattern was perceived to be too large and potentially required decomposition into further sub-components.

3.3.5 Pattern Mining – Cycle 2

The feedback from the writers' workshop was noted and the Asset Pipeline was mined to create three further patterns for the use in the design and implementation of interactive real-time visualization workflows. The patterns consist of an overall architectural workflow pattern, called ASSET PIPELINE, involving the application of three sequential component patterns referred to as DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS.

Together these patterns form the pattern language illustrated graphically in figure 3.5.

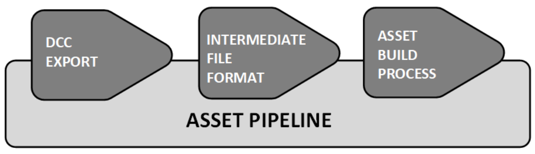


Figure 3.5: ASSET PIPELINE workflow pattern integrating the three component patterns

The ASSET PIPELINE provides visualization designers with a general-purpose efficient workflow for resolving the problem of the source asset data being in an incompatible format to the final visualization data format. A DCC EXPORT provides the conversion from the internal proprietary format of the Digital Content Creation (DCC) tool into a representation capable of further processing. An ASSET BUILD PROCESS is introduced whose responsibility is to optimize and transform the asset structure into the final format. An INTERMEDIATE ASSET FORMAT decouples the source asset format from the final format thus introducing flexibility in the ASSET BUILD PROCESS.

3.3.6 Pattern Writing – Cycle 2

The Asset Pipeline patterns explore the transformation of digital assets to a runtime environment within the general context of interactive real-time visualization. It is envisaged they will form a pattern of application in other domains where source data requires manipulation into a final format. The four patterns along with their category and intent are displayed in Table 3.4.

Pattern	Pattern Category	Intent
ASSET PIPELINE	Architectural	Provides designers with a workflow to assist them in viewing their digital content in an interactive real-time visualization.
DCC EXPORT	Component	A mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing within the ASSET BUILD PROCESS.
INTERMEDIATE FILE FORMAT	Component	Decouples the source and final asset dependency from the asset pipeline.
ASSET BUILD PROCESS	Component	Provides asset data conditioning to fulfill the requirements of the real-time visualization runtime.

Table 3.4: Asset Pipeline pattern language summarizing category and intent

The four patterns are summarized in tables 3.5, 3.6, 3.7 and 3.8 and detailed in sections 3.6.1 to 3.6.4.

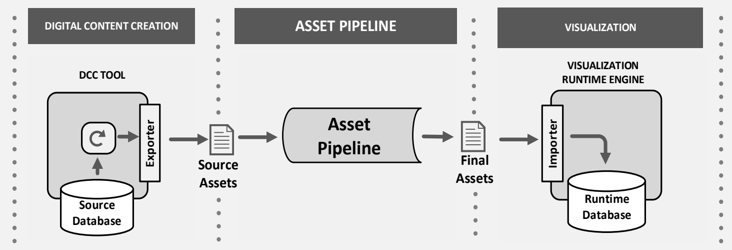
Section	Overview
Pattern Name	ASSET PIPELINE
Diagram	 DIGITAL CONTENT CREATION ASSET PIPELINE VISUALIZATION DCC TOOL Exporter Source Database Source Assets Asset Pipeline Final Assets VISUALIZATION RUNTIME ENGINE Importer Runtime Database
Category	Architectural
Aliases	Content Pipeline
Context	Designers require a workflow to assist them viewing their digital content in an interactive real-time visualization.
Problem	How to efficiently deliver interactive visualizations where the source asset are created in a high-level DCC tool and the visualization operates within a low-level optimized runtime?
Forces	Designers and content creators require an efficient process to assist them in the evaluation of their designs in the runtime. *However*, the process of transferring their designs from the DCC tool into the runtime in preparation for review incorporates many steps, often technically complex, that require careful orchestration.
Solution	Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.
Consequences	Benefits: Design feedback loop efficiency. Liabilities: Expensive initial setup cost.
Known Uses	Video game industry: Large number of source assets, often created with a variety of DCC tools, are transformed and converted into a format applicable for delivery into the game engine runtime.
See Also	DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS component patterns.

Table 3.5: ASSET PIPELINE pattern overview

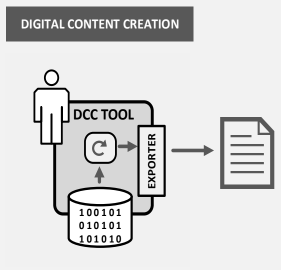
Section	Overview
Pattern Name	DCC EXPORT
Diagram	 DIGITAL CONTENT CREATION DCC TOOL EXPORTER 100101 010101 101010
Category	Component
Aliases	N/A
Context	Designers require a mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing.
Problem	How is data extracted from a DCC tool where the internal DCC source database is persisted in a proprietary format?
Forces	The DCC source data must be presented in a format satisfying the requirements of the ASSET BUILD PROCESS. *However*, the DCC source database is persisted in a proprietary internal format.
Solution	A DCC tool exporter is developed to permit the source asst data to be extracted from the source database into a more usable format.
Consequences	Benefits: Provides a mechanism to query the internal DCC source database and expose the data and data - structures in a format applicable for the ASSET BUILD PROCESS. Liabilities: Designing and implementing a custom DCC tool export plug-in requires software-engineering domain experience. Any future versions of the DCC tool and SDK may require an update of the exporter plug-in. Likewise, any changes in the requirements of the ASSET BUILD PROCESS may require modification to the DCC tool exporter plug-in. Difficult to arrive at a container format and schema to encapsulate all the data requirements of the ASSET BUILD PROCESS.
Known Uses	Extracting three-dimensional scene graph data from a DCC modelling tool. Extracting architectural data from a Building Information Management (BIM) system. Extracting scientific data from a source DCC tool in preparation for further processing.
See Also	Please refer to the INTERMEDIATE BUILD FORMAT Component pattern for a discussion of where standardization of the asset build format negates many of the above liabilities and can introduce flexibility in asset processing.

Table 3.6: DCC EXPORT pattern overview

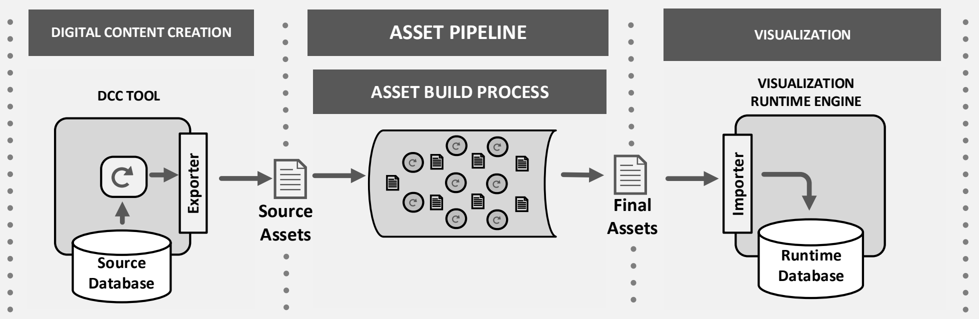
Section	Overview
Pattern Name	ASSET BUILD PROCESS
Diagram	 DIGITAL CONTENT CREATION DCC TOOL Exporter Source Database Source Assets ASSET PIPELINE ASSET BUILD PROCESS Final Assets VISUALIZATION VISUALIZATION RUNTIME ENGINE Importer Runtime Database
Category	Component
Aliases	Asset Conditioning Pipeline
Context	The source assets require data conditioning to fulfil the requirements of the real-time visualization runtime.
Problem	How can the source assets be transformed into final assets in preparation for importing into the runtime?
Forces	Source asset data is information rich and contained in a transparent, lossless format. *However*, the real-time visualization requires asset data cleansed, transformed and optimized to satisfy the requirements of the runtime and target platforms. Designers and content creators require an efficient design iteration workflow. *However*, the procedure of data compilation may be an internally complex process consisting of many dependent steps.
Solution	Integration of an Asset Build Process.
Consequences	Benefits: Incompatible Source and Final Asset Format, design feedback loop efficiency, source asset dependencies. Liabilities: Flexibility of asset build node configuration.
Known Uses	The video game development workflow requires the use of an asset pipeline to integrate the source assets into the game runtime. Real-time filmmaking allows rendering to be performed at run-time. An asset pipeline workflow provides a transparent mechanism for incorporating three-dimensional assets into film scenes.
See Also	Please refer to the INTERMEDIATE BUILD FORMAT Component pattern.

Table 3.7: ASSET BUILD PROCESS pattern overview

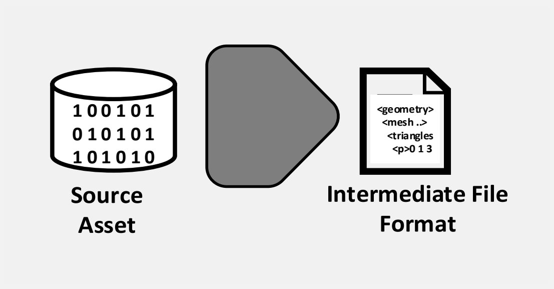
Section	Overview
Pattern Name	INTERMEDIATE FILE FORMAT
Diagram	 100101 010101 101010 Source Asset <geometry> <mesh ..> <triangles <p>0 1 3 Intermediate File Format
Category	Component
Aliases	N/A
Context	Designers are using a variety of DCC tools to create source assets, each with their own storage format.
Problem	How is the DCC tool file format decoupled from the ASSET BUILD PROCESS to introduce reusability and flexibility in the pipeline?
Forces	Source assets are created using multiple DCC tools, each with their individual export format. *However*, each type of source asset requires identical processing within the ASSET BUILD PROCESS. The ASSET BUILD PROCESS nodes are tailored to deal with specific input formats. *However*, this limits the reuse of build nodes and flexibility in their ordering in the ASSET BUILD PROCESS.
Solution	Introduce a unified intermediate file format to increase flexibility and reusability within the ASSET BUILD PROCESS.
Consequences	Benefits: Where multiple source formats require identical asset conditioning the introduction of an intermediate file format allow common processing to be shared in the ASSET BUILD PROCESS. A unified intermediate file format introduces flexibility of asset build node configuration. Liabilities: Arriving at a single intermediate file format may be difficult since it must hold a superset of all data required for the transformations in the build asset pipeline.
Known Uses	The COLLADA intermediate file format has been widely accepted within the videogame development workflow.

Table 3.8: INTERMEDIATE FILE FORMAT pattern overview

3.3.6.1 ASSET PIPELINE

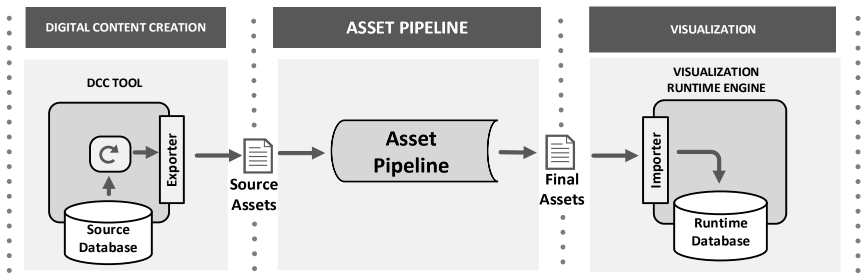


Figure 3.6: ASSET PIPELINE architectural pattern diagram

Category: Architectural.

Aliases: Content Pipeline.

Context:

Designers require a workflow to assist them viewing their digital content in an interactive real-time visualization.

Consider the development of a three-dimensional interactive real-time visualization targeted at the entertainment industry. Designers of the visualization use Digital Content Creation (DCC) tools to create several source assets such as three-dimensional models, materials and textures representing real-world objects. The three-dimensional geometric models are created using a DCC hard surface modelling tool such as AutoDesk 3ds Max and the complementary materials and textures, essential for a high-quality visualization, are designed using a digital painting DCC tool such as Adobe Photoshop.

The visualization operates in a runtime environment targeted for a specific platform. For the scenario of the visual entertainment industry a game engine such as Unity is typically used to inherently provide the runtime capability of real-time rendering. The visualization is targeted to operate on several hardware platforms such as the Windows desktop environment and the Oculus Rift virtual reality (VR) platform to provide a sense of immersion.

Source assets, for instance the three-dimensional model and textures, require optimization for the runtime in question to facilitate fast loading into the runtime and optimal rendering performance. The designers require a method to review their content, or source assets, within the environment of the runtime to ensure they are aesthetically identical to their design intent. Any such changes require adjustment in the DCC tool and the iterative design cycle proceeds until the source asset is considered complete.

Problem:

How to efficiently deliver interactive visualizations where the source assets are created in a high-level DCC tool and the visualization operates within a low-level optimized runtime?

The workflow of creating an interactive real-time visualization consists of the design process of creating the source assets within a DCC tool, transforming the source assets into a compatible form utilized by the visualization runtime and subsequently reviewing the produced visualization. After visual inspection often source assets require modification and hence the feedback loop continues, Figure 3.7. Any form of automation in this respect will increase the efficiency of the designer and reduce the iterative cycle of the design-review process.

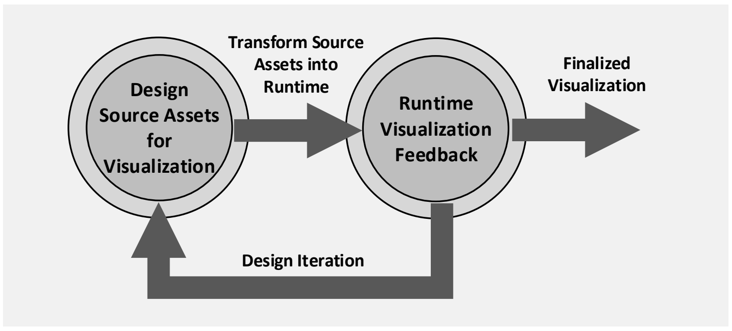

Transform Source Assets into Runtime
Finalized Visualization
Design Source Assets for Visualization
Runtime Visualization Feedback
Design Iteration

Figure 3.7: Design-review process

Forces:

Designers and content creators require an efficient process to assist them in the evaluation of their designs in the runtime.

However, the process of transferring their designs from the DCC tool into the runtime in preparation for review incorporates many steps, often technically complex, that require careful orchestration.

Solution:

Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.

The workflow is called the ASSET PIPELINE and is the mechanism that interfaces the designers of digital content to the visualization runtime. It is the workflow path that all source assets follow, from conception until they are loaded into the runtime engine.

The ASSET PIPELINE forms a composite pattern consisting of the component patterns: DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS. The solution to the ASSET PIPELINE is formed by following the sequential application of the component patterns detailed in figure 3.8 and described below.

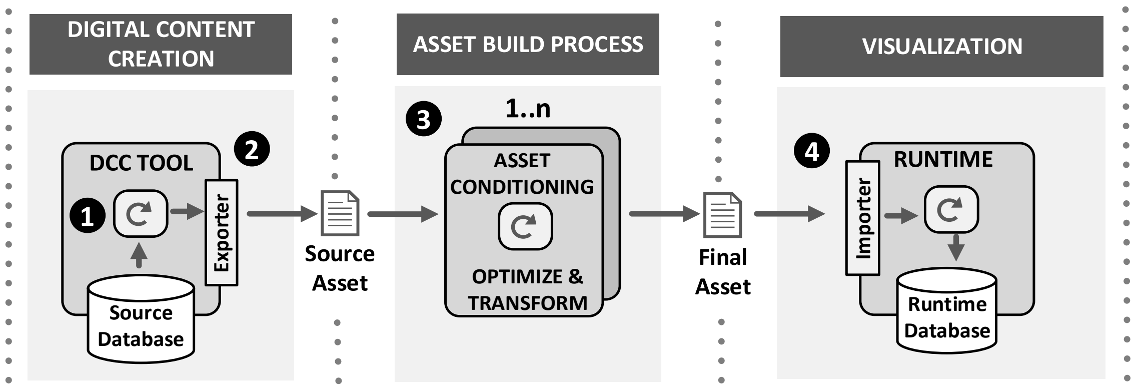


Figure 3.8: Block diagram representing the stages of the asset pipeline

Step 1

One or more DCC tools are used by the designer to create the source assets. The source assets are persisted in the proprietary file format of the DCC tool and are often subject to version-control using an asset-management system.

Step 2

The exporter of the DCC tool is an integral component of the asset pipeline, since it exposes the complex and proprietary internal representation of the source asset data into a more usable format. DCC tool vendors provide the facility for end-users to extend their application and the DCC EXPORT component explores this mechanism in further detail.

Given the source asset data is the front end to the asset build process the designer must ensure they choose an export format where the data container has the capability to hold all the data necessary for the data conditioning pipeline stage. A single unified container file format designed for use within the asset build process provides many advantages, including reusability and flexibility within the pipeline, and the INTERMEDIATE FILE FORMAT component explores the various forces at play and possible solutions.

Step 3

The build process is responsible for orchestrating a series of stages to transform the source asset data into a final format optimized and acceptable for the runtime target platform. The ASSET BUILD PROCESS component provides a pattern for instantiation to deal with this often complicated and technical process.

Step 4

The result of the ASSET BUILD PROCESS is the optimization of source assets into their final asset form acceptable for the target runtime and hardware platform. The pre-existing importer of the runtime can then be used to import the final assets and persist them in their optimized form in the internal runtime database in preparation for execution of the real-time visualization.

Consequences:

Benefits:

- Design feedback loop efficiency: The asset pipeline provides a pattern for creating an optimized build framework. Automating this process increases the efficiency of the designer by reducing the iterative cycle of design and review.

Liabilities:

- Expensive initial setup costs: The asset pipeline process automates and resolves the steps required to transform source assets into their final asset format expected by the runtime.

Known Uses:

- Video game industry: Large numbers of source assets, often created with a variety of DCC tools, are transformed and converted into a format applicable for delivery into the game engine runtime.

See Also:

- DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS component patterns.

3.3.6.2 DCC EXPORT

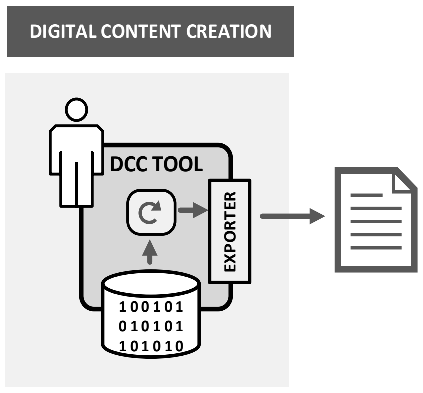


Figure 3.9: DCC EXPORT component pattern diagram

Category: Component

Aliases: Content Pipeline

Context:

Designers require a mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing.

Designers of content for use in a real-time visualization use one or more DCC tools to create the source assets. The DCC tool provides the designer with the capability to operate with efficient workflows using enhanced data modelling and manipulation techniques. Through the process of incremental design, the designer regularly saves and persists their work in the proprietary file format of the DCC tool. At certain stages in the design process the designer needs to review the source assets in the real-time visualization runtime.

However, the source assets require further processing and optimization prior to integration into the real-time visualization runtime. The designer requires a mechanism to extract the data necessary from the DCC tool into a format expected by the ASSET BUILD PROCESS.

Problem:

How is data extracted from a DCC tool where the internal DCC source database is persisted in a proprietary format?

The source assets are often stored in an opaque proprietary format within the source database of the DCC tool. Even when the format is published it may be difficult to parse and require technical knowledge of the underlying DCC tool algorithms that are not available to consumers (Carter, 2004). The source assets are information rich, containing the complete set of semantic data required to reconstruct the DCC tool user-interface in preparation for future editing and modification. However, further stages in the ASSET PIPELINE workflow require the data to be in a transparent, usable format for subsequent processing.

Forces:

- The DCC source data must be presented in a format satisfying the requirements of the ASSET BUILD PROCESS.
  However, the DCC source database is persisted in a proprietary internal format.

Solution:

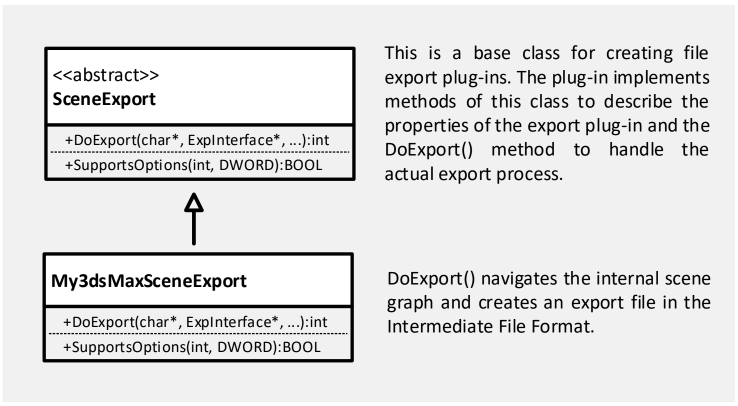
A DCC tool exporter is developed to permit the source asset data to be extracted from the source database into a more usable format.

The DCC tool exporter is an integral component of the workflow, since it exposes the complex and proprietary internal representation of the source asset data in a more usable format. With the recognition that source assets are now designed and manipulated within a variety of complementary tools, commercial manufacturers of DCC tools attempt to widen their market value by offering the capability to export content to many different standard file formats. This extends the source asset interoperability between DCC tools made by different manufacturers.

Frequently commercial DCC tool vendors provide end-users with a Software Development Kit (SDK) to extend the capability of their products and incorporate additional functionality. Catering for common programming languages such as C++, Python and other scripting languages the vendor allows end-users to query and process the internal representation of asset data persisted in the source database. Such an Application Programming Interface (API) can be utilized by an end-user software developer to create a bespoke file exporter plug-in to transform and process the internal representation of the source asset to a format acceptable by the ASSET BUILD PROCESS (Carter, 2004).

For instance, consider the development of a real-time interactive architectural visualization. The scene is designed using the three-dimensional modelling DCC tool Autodesk 3ds Max and comprised of source assets of a landscape and building. Prior to integration into the runtime the source assets require further processing using the ASSET PIPELINE to reduce scene complexity and increase rendering performance.

To integrate with the format required by the ASSET PIPELINE a custom 3ds Max file exporter plug-in is developed, Figure 3.10.



<<abstract>>
SceneExport
+DoExport(char*, ExpInterface*, ...):int
+SupportsOptions(int, DWORD):BOOL
This is a base class for creating file export plug-ins. The plug-in implements methods of this class to describe the properties of the export plug-in and the DoExport() method to handle the actual export process.
My3dsMaxSceneExport
+DoExport(char*, ExpInterface*, ...):int
+SupportsOptions(int, DWORD):BOOL
DoExport() navigates the internal scene graph and creates an export file in the Intermediate File Format.

Figure 3.10: Creation of a DCC export plugin

The source assets are organized within the source database of Autodesk 3ds Max in the form of a scene graph. A commonly used paradigm within three-dimensional modelling, the scene graph provides a hierarchical tree structure where each branch represents the relative position of each object in relation to each other. Autodesk 3ds Max supplies an SDK and corresponding C++ API to allow an end-user to interrogate the scene graph and export the scene in a tailored format. Programmatically this is achieved by creating a plug-in that exposes its functionality to 3ds Max by defining a class that derives from a specific base-class (SceneExport). Implementations of key virtual functions (DoExport) are developed to navigate the scene graph and persist in the required file format. The designer exports the source asset from 3ds Max using the familiar export dialog, selecting the custom export type.

Consequences:

Benefits:

- Provides a mechanism to query the internal DCC source database and expose the data and data-structures in a format applicable for the ASSET BUILD PROCESS.

Liabilities:

- Designing and implementing a custom DCC tool export plug-in requires software-engineering domain experience.
- Any future versions of the DCC tool and SDK may require an update of the exporter plug-in. Often designers utilize one version of a DCC tool for the duration of the visualization development to eliminate the associated maintenance cost of updating the plug-in (Carter, 2004).
- Likewise, any changes in the requirements of the ASSET BUILD PROCESS may require modification to the DCC tool exporter plug-in. Where several tools are used in the production of the visualization content multiple DCC tool plug-ins will require updating.
- Difficult to arrive at a container format and schema to encapsulate all the data required by the ASSET BUILD PROCESS.

Known uses:

- Extracting three-dimensional scene graph data from a DCC modelling tool.
- Extracting architectural data from a Building Information Management (BIM) system.
- Extracting scientific data from a source DCC tool in preparation for further processing.

See Also:

- Please refer to the INTERMEDIATE BUILD FORMAT Component pattern for a discussion of where standardization of the asset build format negates many of the above liabilities and can introduce flexibility in asset processing.

3.3.6.3 ASSET BUILD PROCESS

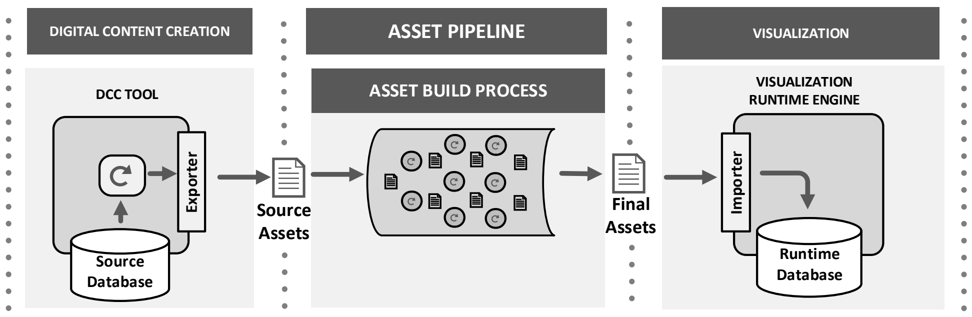


Figure 3.11: ASSET BUILD PROCESS component pattern diagram

<u>Category:</u> Component

<u>Also Known As:</u> Asset Conditioning Pipeline

<u>Context:</u>

The source assets require data conditioning to fulfil the requirements of the real-time visualization runtime.

A DCC EXPORT has been performed to decouple the source assets from their associated DCC tool into a non-proprietary transparent file format. The source assets are now in an INTERMEDIATE FILE FORMAT that is a candidate for further processing. Prior to integration into the real-time visualization the source assets require compilation and optimization to satisfy the requirements of the runtime.

<u>Problem:</u>

How can the source assets be transformed into final assets in preparation for importing into the runtime?

Consider, for example, an asset conditioning pipeline of geometry processing; the designer creates individual scene models based on three-dimensional polygon meshes, constructed as a set of vertices and the associated edges connecting them. Current rendering hardware does not natively support the processing of n-sided polygons. As part of the asset build process stage of the asset pipeline the geometry primitives must be converted to triangles acceptable by the runtime hardware. Such geometry may form the basis for further forms of processing, for instance geometry compression of the high density meshes into lower density forms producing different levels of detail, figure 3.12.

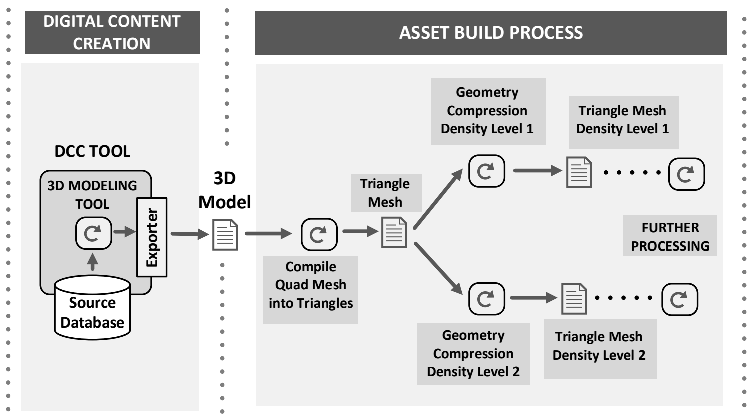

DIGITAL CONTENT CREATION
ASSET BUILD PROCESS
DCC TOOL
3D MODELING TOOL
Exporter
Source Database
3D Model
Compile Quad Mesh into Triangles
Triangle Mesh
Geometry Compression Density Level 1
Triangle Mesh Density Level 1
FURTHER PROCESSING
Geometry Compression Density Level 2
Triangle Mesh Density Level 2

Figure 3.12: Asset build process – Processing of high density meshes into lower density meshes

Forces:

- Source asset data is information rich and contained in a transparent, lossless format.

  However, the real-time visualization requires asset data cleansed, transformed and optimized to satisfy the requirements of the runtime and target platforms.

- Designers and content creators require an efficient design iteration workflow.

  However, the procedure of data compilation may be an internally complex process consisting of many dependent steps.

Solution:

Integration of an Asset Build Process

The exported source assets must undergo several processing steps before they can be integrated into the final runtime. The asset build process is the component tasked with performing this source data conditioning. In its simplest form the asset build process is executed at the workstation of the designer and is the main workflow for pushing exported source content from the DCC into the visualization runtime (Gregory, 2014).

In a similar manner to a software development toolchain consisting of compiling and building source code for each target platform, the asset build process consists of compiling and optimizing the exported source assets for each target runtime platform.

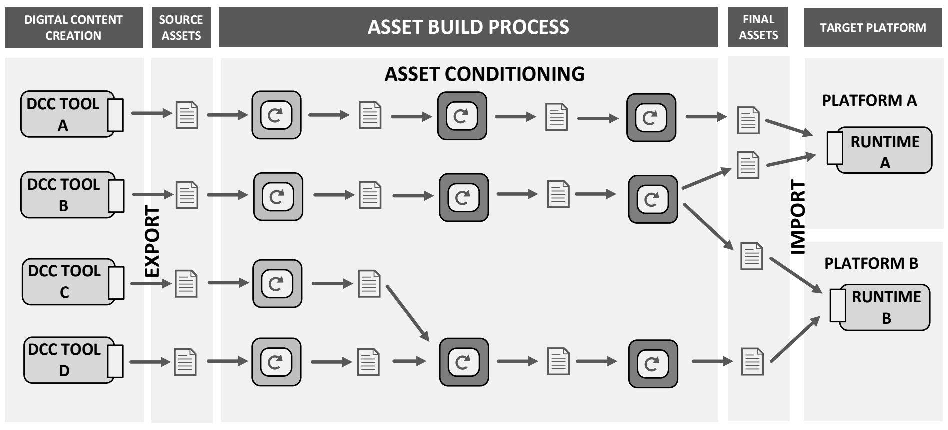


Figure 3.13: Asset build process

Each build asset node within the asset build process in figure 3.13 represents a single step that performs a transformation on one or more input assets into one or more output assets. In practice this transformation represents an act of either invoking a conversion script, a data compiler or a tool resulting in gradual refinement of the source assets into their final representation as consumed by the runtime environment. The exact nature of the steps is driven by several factors including the type of assets involved, the visualization domain and the target platforms.

Each input asset is a direct dependency for an individual asset build node, and the build node itself becomes a direct dependency for each of its outputs, figure 3.14. That is, a many to many relationships exists whereby one output asset file may depend upon many source assets and one source asset may produce many output asset files.

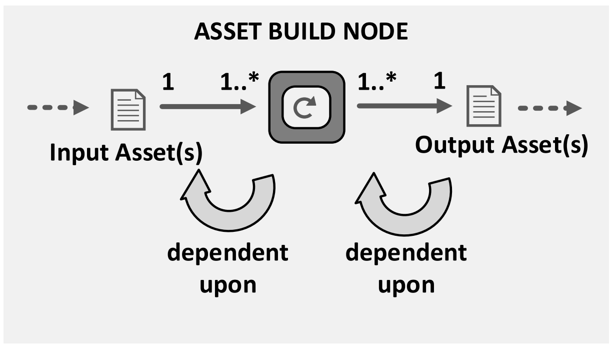


Figure 3.14: Dependencies of an asset build node

In actual practice where visualizations consist of a large amount of content the hierarchy can be both deep and wide. Automating the asset build process depends upon determining the execution order of the individual asset build node steps so a set of tasks can be scheduled and executed.

A build node can only be executed if all its child dependencies are satisfied. Theoretically, determining the correct order of build dependencies is a straightforward process since the entire hierarchy is represented as a directed acyclic graph (DAG) (Gregory, 2014). Considering a software build process, the dependency information is inherently specified within the source code written in text format. However, source assets are often binary files, in proprietary formats, from which parsing to determine dependencies is technically challenging. For an asset build process, consisting of binary source assets, there two differing solutions that can be implemented: a static and declarative asset build process.

Firstly, the dependency information and the execution order can be hard coded in the build script. This static asset build process introduces a maintenance overhead since the build script requires updating every time a designer adds an asset. In software development the static approach is identical to building source code using a batch file or traditional make file. Several off the shelf solutions can be extended to incorporate the compilation and build of assets including Scons, CCNet, TFS, Perforce and Cmake.

Secondly, and a more flexible approach, is to produce a framework where each asset build node is represented as a template specifying the expected source input type, a set of parameters to bind to the build script and the outputs that are produced. The asset build framework gathers the source assets and therefore the set of input types, matching those to the set of build asset node templates in order to generate the dependency graph for the assets that require building. This declarative approach to the asset build process minimizes the amount of maintenance changes that must occur when a new source asset is added or modified. The content pipeline in the MonoGame open source implementation of Microsoft XNA framework uses such a declarative approach for transforming source assets into their final realized form.

Consequences:

Benefits:

- Incompatible Source and Final Asset Format: The asset build process automates and resolves the steps required to transform source assets into their final asset form expected by the runtime.
- Design Feedback Loop Efficiency: The asset build process provides a solution for creating an optimized build framework. This workflow increases the efficiency of the designer by providing an automated mechanism to reduce the time spent transferring the source assets from the DCC into the runtime for review.

- Source Asset Dependencies: The process of determining the source asset dependency chain required for the asset build process is solved using a dependency strategy, either static or dynamic.

Liabilities:

- Flexibility of Asset Build Node Configuration: Use of multiple asset formats within the asset build process results in an inflexibility in the configuration of the build steps. The build steps cannot be easily reordered and reused.
  Please see INTERMEDIATE ASSET FORMAT for a solution.

<u>Known Uses:</u>

- The video game development workflow requires the use of an asset pipeline to integrate the source assets into the game runtime.
- Real-time filmmaking allows rendering to be performed at run-time. An asset pipeline workflow provides a transparent mechanism for incorporating three-dimensional assets into film scenes.

<u>See Also:</u>

- Please refer to the INTERMEDIATE BUILD FORMAT Component Pattern.

3.3.6.4 INTERMEDIATE FILE FORMAT

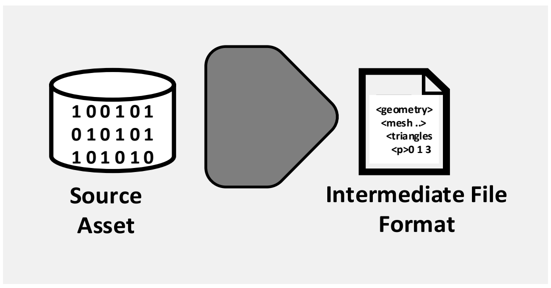


Figure 3.15: INTERMEDIATE FILE FORMAT component pattern diagram

Also Known As: Intermediate Asset Format

Context:

Designers are using a variety of DCC tools to create source assets, each with their own storage format.

Consider the development of a large and rich interactive real-time visualization requiring a variety of three-dimensional models. Historically designers used one all-encompassing modelling package such as Autodesk 3ds Max to create their content. However, with the proliferation of new three-dimensional modelling tools, each with their own specialized function, designers are increasingly using many tools in their day to day operation. For example, designers use Autodesk 3ds Max and Maya for hard surface modelling, Pixologic Zbrush for digital sculpting and Trimble SketchUp for industrial design. Each DCC tool has its own proprietary file format with partial support for exporting to other formats.

Designers of the visualization DCC EXPORT their source assets from the three-dimensional modelling tool and execute the ASSET BUILD PROCESS expecting the workflow to understand the different content formats.

Problem:

How is the DCC tool file format decoupled from the ASSET BUILD PROCESS to introduce reusability and flexibility in the pipeline?

Many of the source assets, particularly those of a similar category such as three-dimensional geometry, may be subject to the same set of asset build node transformations prior to integration within the runtime. Consider a visualization involving source assets based on three-dimensional models textured with a material as shown in Figure 3.16. Designers create three-dimensional model assets using a variety of DCC tools, each of which are exported in different file formats. The build steps for the three-dimensional models are algorithmically identical, consisting of geometry processing, however, since they do not expect a common format the asset build nodes must be individually tailored for each input format. An identical situation arises for the material textures.

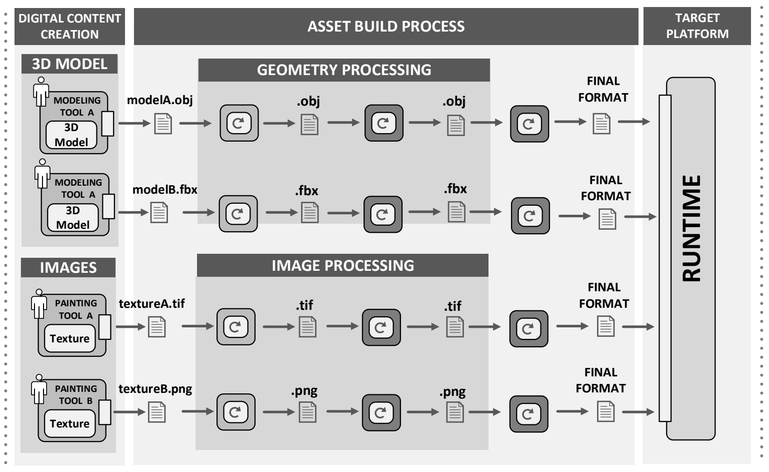


Figure 3.16: Asset build nodes performing algorithmically identical processing on different asset formats

Use of multiple asset formats within the ASSET BUILD PROCESS results in an inflexibility in the configuration of the build steps. The build steps cannot be easily reordered and reused between subsequent projects.

Forces:

- Source assets are created using multiple DCC tools, each with their individual export format.
  However, each type of source asset requires identical processing within the ASSET BUILD PROCESS.
- The ASSET BUILD PROCESS nodes are tailored to deal with specific input formats.
  However, this limits the reuse of build nodes and flexibility in their ordering in the ASSET BUILD PROCESS.

Solution:

Introduce a unified intermediate file format to increase flexibility and reusability within the ASSET BUILD PROCESS.

Initially DCC EXPORT is applied to convert the source assets from their native DCC tool proprietary format into an intermediate file format not tied to the final presentation expected by the runtime. It is the intermediate file format that is used as the input to the ASSET BUILD PROCESS tasked with generating the runtime ready final assets, figure 3.17.

The requirements for the intermediate file format differ from those of the final format expected by the visualization runtime engine. The intermediate file format must be information rich and lossless where possible since it must contain all the necessary information for processing further in the pipeline.

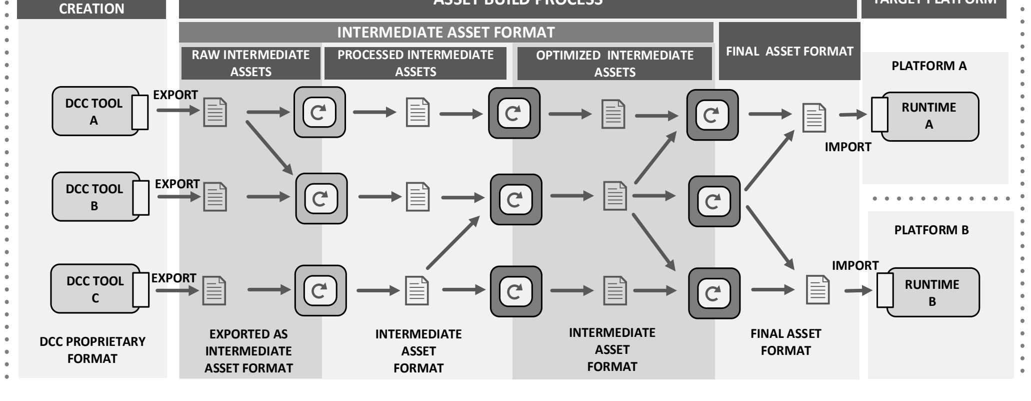


Figure 3.17: Use of intermediate file format within the asset build process

Structurally the intermediate file format must be easy to read, parse and extend to fulfil any future requirements expected of the runtime. Given this, the intermediate file format can be satisfied using a plain text file as a container, although any self-describing human-readable format such as XML is suitable. XML is considered a data interchange format that can be parsed, queried and manipulated using a set of common programming languages. Furthermore, the hierarchical structure of the embedded document object model (DOM) aligns with the hierarchical scene graph structure commonly representing a three-dimensional scene in a visualization. In this instance, an XML schema must be designed and implemented to contain a superset of all the data required for processing within the ASSET BUILD PROCESS.

In most cases a single intermediate file format should be used during all stages of the asset build process. The implementation of a unified container format permits common file parsing and writing functionality to be abstracted and reused within the algorithms of individual asset build node steps. Subsequently, since all build node steps operate on a single intermediate format the input and output parameter format of the build step is identical with only the specific core transformation algorithm altering. A powerful benefit resulting in an introduction of flexibility in the build process whereby asset build steps can be reused and reordered where necessary.

There are two possible integration points within the ASSET PIPELINE where source assets can be converted into an intermediate file format: either at the point of DCC EXPORT from the DCC tool (figure 3.18) or within the first stage of the ASSET BUILD PROCESS (figure 3.19).

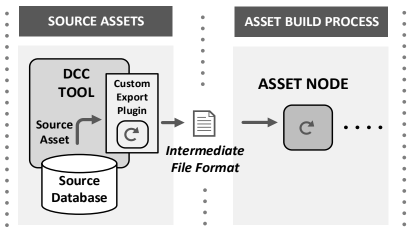


Figure 3.18: DCC Export provides conversion

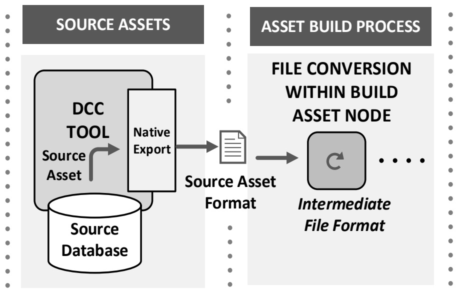


Figure 3.19: Conversion within the first stage of asset build process

Consider the scenario of integration within the asset export process, as in Figure 3.18. An export plug-in must be developed for each distinct DCC tool used to transform the internal representation of the source assets into the intermediate format used by the build asset pipeline. Please review DCC EXPORT for a discussion of the consequences. A more favourable approach is to perform the conversion from source asset to intermediate asset format within the first stage of the ASSET BUILD PROCESS, figure 3.19. This has the distinct advantage of loosely coupling the proprietary format of the DCC tool from the final runtime format. Subsequently, any changes to either the final runtime format or the DCC tool proprietary format requires only a change to the asset build node tasked with performing the export.

For visualizations consisting of three-dimensional models an existing solution in the form of COLLADA (COLLABAborative Design Activity) may be used. Developed as a collaboration between Sony Computer Entertainment and the Khronos Group the format aims to standardize a common intermediate representation for use by DCC tool vendors so that individuals do not have to implement a custom DCC EXPORT and asset build node importers. Initially developed for interchange of assets between DCC tools, the format is sufficiently rich in functionality to allow it to be used directly in the asset build process. Furthermore, COLLADA uses XML as the container and is human readable allowing for human inspection during the build process. The uptake of native COLLADA is growing with several DCC tool and runtime vendors supporting the format.

Consequences:

Benefits:

- Where multiple source formats require identical asset conditioning the introduction of an intermediate file format allows common processing to be shared in the ASSET BUILD PROCESS where source assets are of a similar type.
- A unified intermediate file format introduces flexibility of asset build node configuration. The asset build nodes may be reordered because they all share the same input and output file types.

Liabilities:

- Arriving at a single intermediate file format may be difficult since it must hold a superset of all data required for the transformations in the build asset pipeline. Use of COLLADA and other interchange formats provide a good starting point in solving this.

Known Uses:

- The COLLADA intermediate file format has been widely accepted within the video game development workflow.

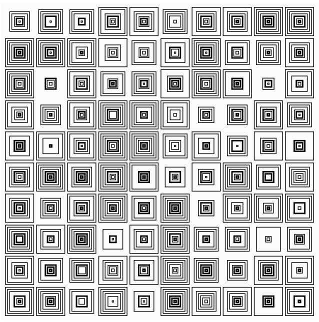
3.4 Summary

The spiral methodology was adopted to derive the asset pipeline pattern language. Initially manual pattern mining resulted in the production of the two abstract patterns of ASSET PIPELINE and INTERMEDIATE ASSET FORMAT. These were refined using an iterative approach under the supervision of a Pattern Languages of Program (PLoP) shepherd. Subsequently, after review and constructive feedback at the Pattern Languages of Programs (EuroPLoP, 2019) conference the ASSET PIPELINE pattern was decomposed into the component patterns of DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS. The ASSET PIPELINE forms an architectural pattern proving designers with a workflow to assist them in viewing their digital content in an interactive real-time visualization. The ASSET PIPELINE involves the application of the three sequential component patterns. The DCC EXPORT provides a mechanism to transform and extract the DCC source asset data into a format appropriate for further workflow processing within the parent ASSET PIPELINE. The container of the INTERMEDIATE FILE FORMAT decouples the source and final asset format from the asset pipeline thus introducing flexibility in the pipeline.

The process of the Pattern Languages of Programs (PLoP) is rigorous in its approach encapsulating both a shepherding process and a writers' workshop. However, incorporating feedback from domain experts such as those infrastructure specialists of the technical artist or the software tools engineer would have strengthened the functionality of the pattern language. Unfortunately owing to the problems outlined in section 1.2 such collaboration was not possible and out of the scope of this research study.

This chapter formalized the asset pipeline into a pattern language for general purpose instantiation and use in the domains of interactive visualization. As a consequence, the following two chapters will evaluate the derived pattern language via application and construction of two visualizations: a real-time architectural visualization (ArchViz) and a real-time graph network visualization.

4 Use Case – Architectural Visualization


Generative Pattern 4– J Lear 2020

Architectural visualizations are used in architecture, engineering and construction (AEC) projects to communicate designs and building details to clients and other stakeholders such as onsite management and contractors. Traditionally the intentions of the architect are conveyed using two-dimensional technical drawings, either by hand or using Computer Aided Design (CAD) software. Plans, elevations and details are constructed based on the three-dimensional mental model visualized in the mind of the designer. During a design review phase, the interested parties are tasked with interpreting and transforming the two-dimensional design into a three-dimensional spatial model. A difficult task considering a small project can easily contain over one hundred architectural drawings (Smith, 2012). A three-dimensional medium overcomes the limitation of two-dimensions as it represents an intuitive common language for visual interpretation amongst all interested parties (Hughes, 2014). Historically this has taken the form of a scale model of the structure, known as a Physical Mock-Up (PMU). However, with the advancement and popularity of three-dimensional modelling software, computers are increasingly being utilized as a medium for generating architectural visualizations.

Architectural visualizations may take many forms: from static computer-generated images (CGIs), fixed path walkthroughs to truly interactive, real-time experiences. Contemporary visualizations are realistic in their appearance, sometimes photo-realistic, incorporating materials, lighting and landscaping. However, this realism does not come without a cost. Existing approaches use a workflow, a number of manual steps, which are both heavyweight and require expertise to produce (Smith, 2012).

4.1 Existing Architectural Visualization

A still render image is a single computer-generated image of an AEC project from a fixed point of view. The focus of which is an interesting visual aspect of the construction. Figure 4.1 illustrates an example of a still render – the Faculty of Business and Law Building at the University of the West of England (2016).

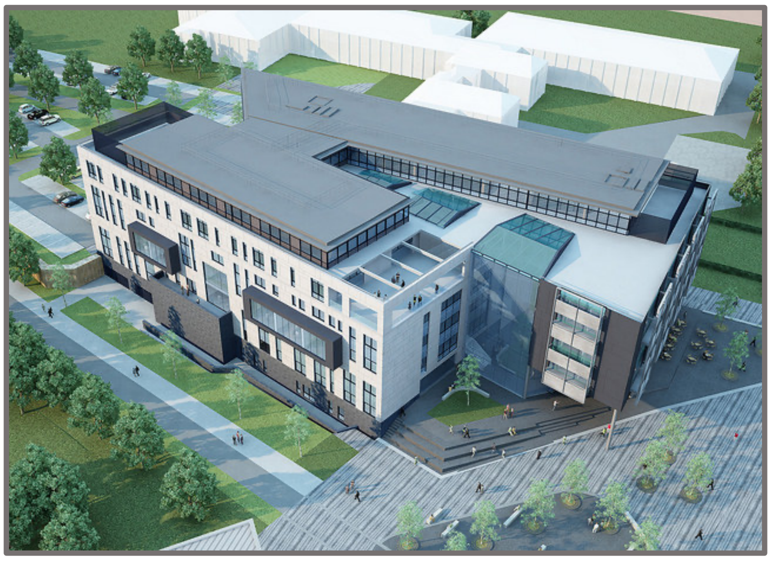


Figure 4.1: An example of a still render – the Faculty of Business and Law Building (UWE)

The still render was produced for marketing purposes, prior to the construction of the building; the aerial view illustrates the attributes of the proposed architectural design. Hence, the observer gains an understanding of the colour of the building, materials used, interaction of natural light and the placement of the building within its surroundings. Indeed the render provides an immediate understanding of the sense of place or Genius Loci (Norberg-Schulz, 1980).

4.2 Architectural Visualization Workflow

Production of a still render requires a collaboration between the architect and designer. The exchange of information and processing is performed using the traditional workflow illustrated in figure 4.2 and described below.

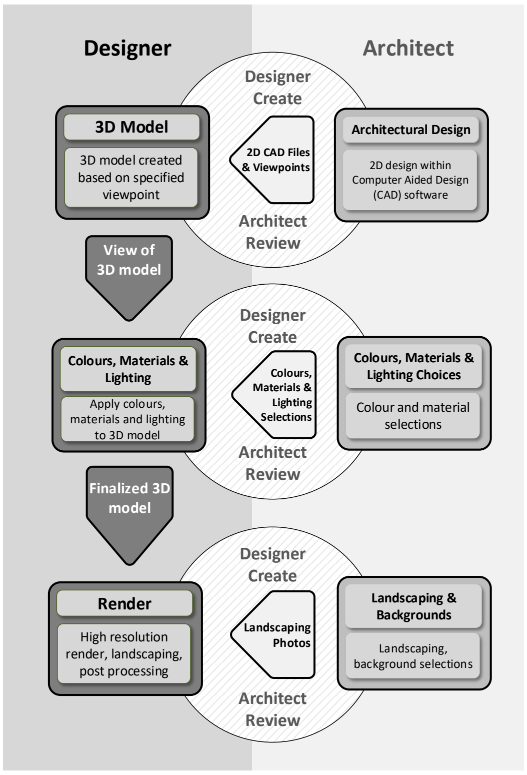

Designer
Architect
Designer Create
3D Model
3D model created based on specified viewpoint
2D CAD Files & Viewpoints
Architectural Design
2D design within Computer Aided Design (CAD) software
Architect Review
View of 3D model
Designer Create
Colours, Materials & Lighting
Apply colours, materials and lighting to 3D model
Colours, Materials & Lighting Selections
Colours, Materials & Lighting Choices
Colour and material selections
Architect Review
Finalized 3D model
Designer Create
Render
High resolution render, landscaping, post processing
Landscaping Photos
Landscaping & Backgrounds
Landscaping, background selections
Architect Review

Figure 4.2: The traditional workflow for the production of a still render

1. *Three-dimensional Model:* Initially, the architect defines the requirements for the visualization in terms of a number of views and camera angles. Taking the plans and elevations as a visual guide the designer constructs a three-dimensional model within a Digital Content Creation (DCC) tool, focusing on the specified views.
   For example, studying figure 4.1, the focus of the camera shows a raised frontal view of the building. For this particular render, the hidden rear aspect of the building need not be modelled or modelled with a lower level of detail.
   This is an essential stage since any changes to the focus of the view obviously has repercussions "downstream". In fact altering view angles completely may require extensive re-modelling to be performed. Thus, drafts of the proposed model are sent to the architect for approval and rework if necessary.
2. *Colours, Materials & Lighting:* Once the structure of the model is complete the architect provides the colour and material selections. The finer elements of additional furniture, textures and lights are created and placed into the scene. Again, the approval process is repeated to ensure the final model, viewpoint, materials and colours are correct.
3. *Render:* A final high-resolution render is produced and landscaping elements – backgrounds, trees, grass, plants – are incorporated in a post-processing phase.

4.3 Architectural Visualization Workflow Problems

The workflow for the production of the visualization follows a linear sequence of tasks and is known as a "pipeline". Although the pipeline ultimately achieves the desired result it has two closely related limitations:

1. *Uni-directional pipeline*: The pipeline flows in one direction – downstream. Any changes made in a task upstream cannot be propagated through the dependency chain without invalidating or losing existing changes.
2. *Time consuming creative iterations*: Each step in the pipeline requires a process whereby the architect specifies the requirements, the designer implements those requirements in the model and the changes are approved by the architect. This creative iteration may be destructive in nature requiring rework by the modeller.

The pipeline limitations are further compounded when more advanced forms of visualizations are required, such as building walkthrough animations. Instead of a fixed render a "fly-by" animation is created with the camera travelling on a pre-defined route. To achieve this an additional stage in the workflow is required whereby a scene is setup with one or more camera paths. Subsequently the scene is rendered frame by frame, a slow task, making it problematic to incorporate any changes to the scene.

In practice, production of an architectural visualization requires proficiency in a DCC tool targeted at creating and rendering three-dimensional models. Autodesk's 3ds Max (AutoDesk, 2016a) and Maya (AutoDesk, 2016b) are widely used in this regard. These inherently complex types of software require such expertise to master that visualizations are often outsourced to a dedicated external company. The process is a time consuming and costly task. Consequently, they are primarily used to visualize a finalized design, rather than becoming a tool in which to provide timely informed design decisions throughout the lifecycle of a project.

Many of the issues raised remain open and have not been solved in the field of architecture. However, a process that exists within computer games development parallels the architectural process; with the additional benefits of efficiency and flexibility.

4.4 Integration of Asset Pipeline and Architectural Visualization Workflow

A correlation exists between the architectural visualization workflow and the game asset pipeline; both take a three-dimensional model and manipulate the asset into a form desired for their output. An architectural visualization may be presented as a render and the game asset pipeline creates a runtime object to be used by the game engine.

An opportunity exists to extend the architectural visualization pipeline already in place to integrate ideas from the game asset pipeline to:

1. Produce rapid design iterations pre-fabrication, allowing for a better-informed design and flexibility.
2. Use enhanced visualizations leveraging the rendering capabilities of game engines. That is the features of rendering in three-dimensions, applying textures, lighting and shadows.
3. Provide designers flexible control over the production of a visualization.
4. Increase the usability and impact for various audiences.

The aforementioned architectural visualization pipeline uses two-dimensional Computer Aided Design (CAD) files as its foundation. Based on the plans and elevations a three-dimensional model is created. However, an exciting development within the AEC industry has the potential to change this existing workflow; the uptake of Building Information Modelling (BIM).

4.5 Architectural Models and Building Information Modeling (BIM)

Building Information Modeling (BIM) is an emerging technology in the AEC industry. BIM extends the CAD paradigm to incorporate both a three-dimensional representation of the building components and the data that describes how they behave. This key concept is known as parametric modelling. A parametric BIM object consists of its geometric definition along with associated semantic data and rules. For example, a rule may specify the height of a windowsill above floor level.

The National BIM Standard-United States (NBIMS-US) provides the following definition for BIM: "BIM is a digital representation of physical and functional characteristics of a facility. As such, it serves as a shared knowledge resource for information about a facility, forming a reliable basis for decisions during its life cycle from inception onward." (Fernandez, 2015).

With the above emphasis on collaborative working, the BIM repository is shared among all interested parties, designers, subcontractors and clients, for the lifecycle of the project. That is from the initial design phase through to construction, occupation and into the building maintenance and aftercare phase.

In practise architects, structural engineers and other contributors from the AEC industry design and collate the building information using a single vendors BIM product such as Autodesk Revit (Autodesk, 2016c). This provides a large, living central repository of information that can be used during the project life-cycle. Having such a building model repository enables the impact of changes to be identified immediately. This allows for the effective identification and reporting of interferences in a BIM model, known as clash detection.

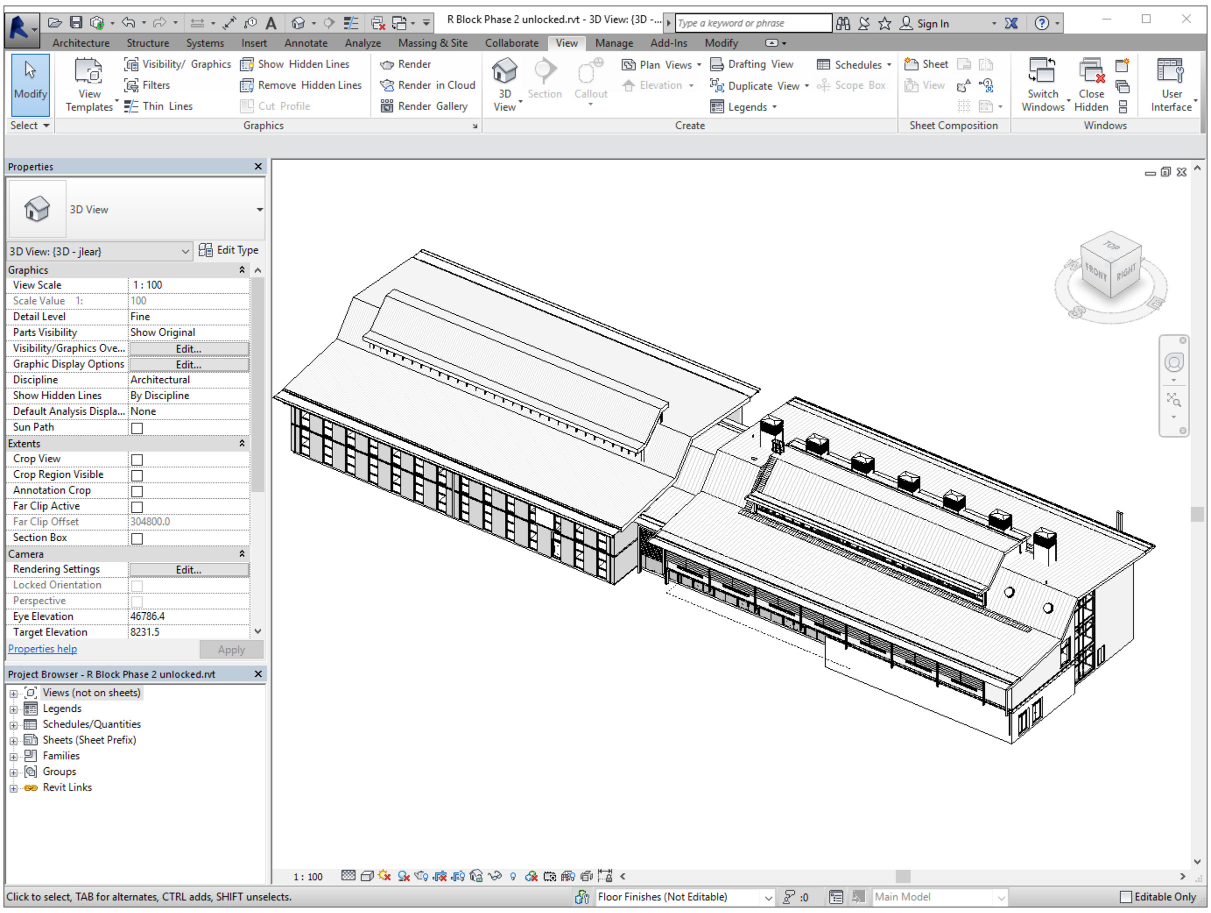
BIM is considered of such importance by the UK Government that by 2016 all AEC project information must adhere to a fully collaborative three-dimensional BIM (Parliament, 2012).

4.6 Use Case Prototype

An initial prototype was developed to produce a BIM visualization tool providing real-time three-dimensional navigation of a building in first-person view. The semantic information contained within the model is used to enhance the workflow and visualization.

Autodesk Revit 2015 was chosen for the BIM Digital Content Creation (DCC) tool owing to its reputation within the AEC industry and also the high level of support available at the University of the West of England. The product is fully compatible with the latest BIM standards and offers a high level of interoperability amongst other products.

The source model used in the study was a BIM of the Architecture and the Built Environment block as supplied by the University of the West of England. Developed using Autodesk Revit, the model contains the full geometry of the building along with a limited subset of the semantic information, figure 4.3.



Properties

3D View

3D View: {3D - jlear}   Edit Type

Graphics	
View Scale	1 : 100
Scale Value 1:	100
Detail Level	Fine
Parts Visibility	Show Original
Visibility/Graphics Ove...	Edit...
Graphic Display Options	Edit...
Discipline	Architectural
Show Hidden Lines	By Discipline
Default Analysis Displa...	None
Sun Path	☐
Extents	
Crop View	☐
Crop Region Visible	☐
Annotation Crop	☐
Far Clip Active	☐
Far Clip Offset	304800.0
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	☐
Perspective	☐
Eye Elevation	46786.4
Target Elevation	8231.5

Properties help   Apply

Project Browser - R Block Phase 2 unlocked.rvt

- Views (not on sheets)
- Legends
- Schedules/Quantities
- Sheets (Sheet Prefix)
- Families
- Groups
- Revit Links

1 : 100

Click to select, TAB for alternates, CTRL adds, SHIFT unselects.   Floor Finishes (Not Editable)   :0   Main Model   ☐ Editable Only

Figure 4.3: Architecture and the Built Environment Building Information Model

Unity 3D was chosen as the game engine since it focuses on intuitive and fast project development speeds, facilitating quick prototyping. The render system is well suited to the field of visualization owing to its advanced graphical capabilities such as real-time global illumination and physically based shading. Unity supports native interoperability with three-dimensional modelling tools, such as 3dsMax, allowing the product to import the Autodesk FBX file format. In terms of software development the engine consists of an object-oriented framework programmable in the C# programming language. The distinct advantage Unity has over other games engines is its provision for multi-platform development and deployment to desktops, consoles, browsers and mobile. Finally, Unity receives excellent community support, essential when performing research.

The following sections detail the instantiation of the asset pipeline pattern and component patterns in the production of such a visualization. Specifically referring to the ASSET PIPELINE pattern definition in section 3.6.1 and aligning the pattern sections to the problem.

4.7 ASSET PIPELINE

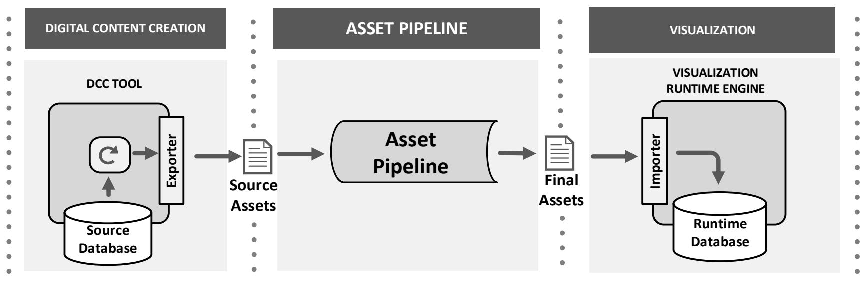


Figure 4.4: ASSET PIPELINE architectural pattern diagram

Context

Designers require a workflow to assist them viewing their digital content in an interactive real-time visualization.

Architects use the BIM Digital Content Creation (DCC) tool Autodesk Revit to design and specify buildings. However, the tool does not incorporate interactive real-time rendering capabilities afforded by runtime game engines. The architects require an efficient workflow to assist them in viewing their designs within an interactive real-time visualization to evaluate their designs and increase the usability and impact to various stakeholders and audiences.

Problem

How to efficiently deliver interactive visualizations where the source assets are created in a high-level DCC tool and the visualization operates within a low-level optimized runtime?

The architects operate within the confines of the Digital Content Creation (DCC) tool Autodesk Revit. The DCC tool is rich in functionality, allowing for the design and specification of the building. A central living repository is maintained during the lifecycle of the project used by all stakeholders operating within the AEC industry. The Autodesk Revit DCC tool operates using high-level design principles necessary to allow the stakeholders to create and amend building objects. The Autodesk Revit files are binary and proprietary in the nature. Whereas, the architectural visualization is operating in the low-level game engine runtime Unity and requires data presented in a different form.

Solution

Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.

Instantiation of the ASSET PIPELINE pattern is formed by following the sequential application of the component patterns: DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS as illustrated in figure 4.5 and described below.

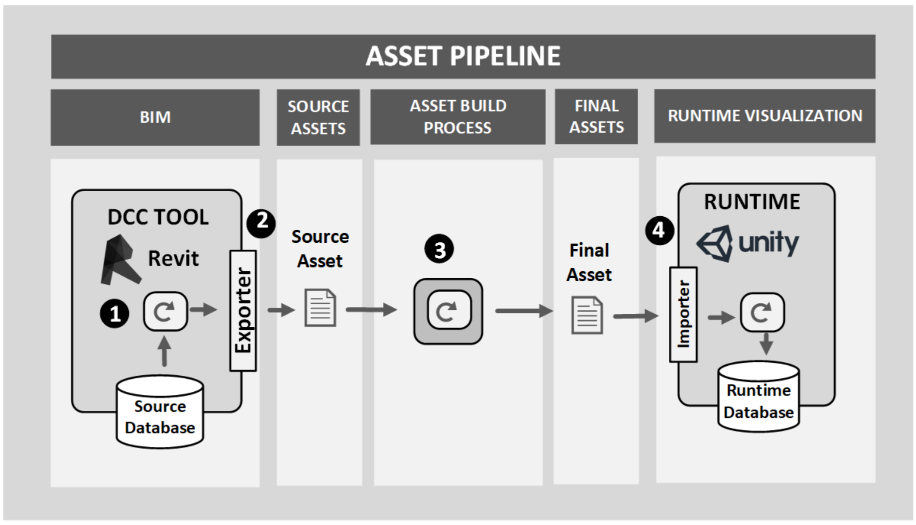


Figure 4.5: BIM Visualization Asset Pipeline

Step 1

The BIM model is held in a central repository for access by all interested AEC stakeholders. The complete dataset is the Architecture and the Built Environment block within the University of the West of England. The source asset project file is persisted in proprietary binary format with file extension of ".REVIT".

Step 2

The DCC tool Autodek Revit supports exporting to a wide range of industry standards and file formats. Please refer to the component patterns DCC EXPORT and INTERMEDIATE FILE FORMAT for a further discussion.

Step 3

The BIM Revit source database contains information rich architectural semantic data. Whereas, a runtime visualization requires three-dimensional geometric data with the associated semantic properties. The build process manipulates the source asset data into a format acceptable by the runtime visualization.

Please refer to the ASSET BUILD PROCESS for a complete explanation.

Step 4

The result of the ASSET BUILD PROCESS is the optimization of the source assets into their final asset form acceptable for the Unity runtime game engine. A custom importer designed and implemented within the Unity runtime loads the final assets. The importer generates the runtime objects necessary for viewing with the three-dimensional building in first-person view.

4.8 DCC EXPORT

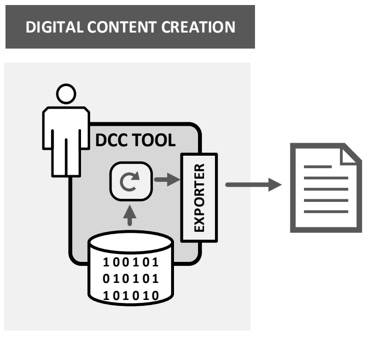


Figure 4.6: DCC EXPORT component pattern diagram

Context:

Designers require a mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing.

The Autodesk Revit project file encapsulating the building geometry, semantics and Revit Graphical User Interface (GUI) editing history is persisted in binary format. The ASSET BUILD PROCESS requires the source asset to be structured in an opaque format suitable for further processing.

Problem:

How is data extracted from a DCC tool where the internal DCC source database is persisted in a proprietary format?

The Autodesk Revit format ".RVT" is proprietary in nature. Although Autodesk offer a Software Development Kit (SDK), the Application Programming Interface (API) operates only within the bounds of the operating application process; primarily used for macro purposes. The bespoke format cannot be interpreted by the ASSET BUILD PROCESS.

Solution:

A DCC tool exporter is developed to permit the source asset data to be extracted from the source database into a more usable format.

In this use case scenario, the recommendation of exporter development is not required. Autodesk Revit exports the project file in the formats identified in table 4.1.

File Formats	Description
DWG, DXF, DGN, ACIS SAT	Computer Aided Design (CAD) formats.
DWF/DWFx	For use with Autodesk Viewer.
ADSK	Autdoesk Exchange File format used to exchange CAD models between Inventor to Revit to Civil 3D.
FBX	Autodesk FBX is a 3D data interchange format utilized between 3D editors and game engines.
NWC	Cached version of model geometry created and used by Autodesk Navisworks.
gbXML	Contains the XML semantics for over 500 types of building elements and attributes.
IFC	Industry Foundation Classes (IFC) format used by BIM DCC Tools.
JPEG, TIFF, BMP, TARGA, PNG	Static image formats.

Table 4.1: Autodesk Revit export file formats.

Please refer to application of the INTERMEDIATE FILE FORMAT for a discussion of the common container format derived for subsequent processing in the ASSET BUILD PROCESS.

4.9 INTERMEDIATE FILE FORMAT

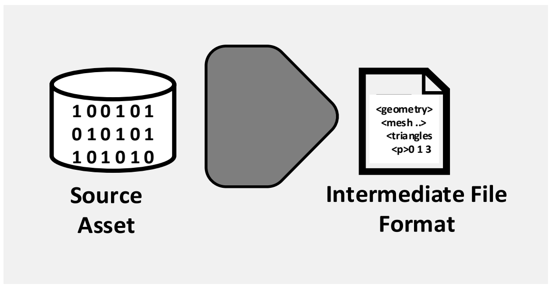


Figure 4.7: INTERMEDIATE FILE FORMAT component pattern diagram

Context:

Designers are using a variety of DCC tools to create source assets, each with their own storage format.

Autodesk Revit is utilized by the primary project stakeholders in the AEC industry. The data is stored in a shared, central repository for all stakeholders to access over the lifetime of the project. Data is proprietary in nature. Designers of the BIM visualization DCC EXPORT their source assets and execute the ASSET BUILD PROCESS expecting the workflow to understand the format.

Problem:

How is the DCC tool file format decoupled from the ASSET BUILD PROCESS to introduce reusability and flexibility in the pipeline?

The ASSET BUILD PROCESS cannot extract the necessary three-dimensional and semantic data from the native format of the DCC Tool Autodesk Revit. The Software Development Kit (SDK) operates within the confines of the application process only.

Solution:

Introduce a unified intermediate file format to increase flexibility and reusability within the ASSET BUILD PROCESS.

An Architectural Visualization (ArchViz) operating in first-person view must allow for the viewer to traverse through doors unobstructed and all hard surfaces such as walls must be impenetrable. Derivation of the individual building components cannot be achieved using the three-dimensional geometry alone. By leveraging the knowledge rich BIM meta-data, the process of creating the final visualization within Unity can be automated, since distinct behaviours can be assigned to categories of objects. For example, all walls must appear solid and all doors must allow a user to pass through them unobstructed. This of course requires access to the BIM semantic information within the visualization application, the game engine. The key to this is in the correct choice of intermediate asset format which must both contain the building geometry and semantic information. Secondly, it must also provide a high level of interoperability between the source DCC tool, Revit, and the game engine, Unity.

The chosen format for the intermediate assets is the Autodesk FBX data exchange format. The container is a three-dimensional asset exchange format that facilitates transfer of data between DCC Tools such as Autodesk 3dsmax, Autodesk Maya, Pixologic ZBrush and other third-party software. The format provides a high level of interoperability between the source DCC tool, Revit, and the game engine, Unity. Furthermore, upon inspection the file also contained embedded BIM semantic data for all architectural components, a discovery Bille *et al.* (2014) missed. This data is stored as a set of "user properties" against each element in the form of name, value pairs.

4.10 ASSET BUILD PROCESS

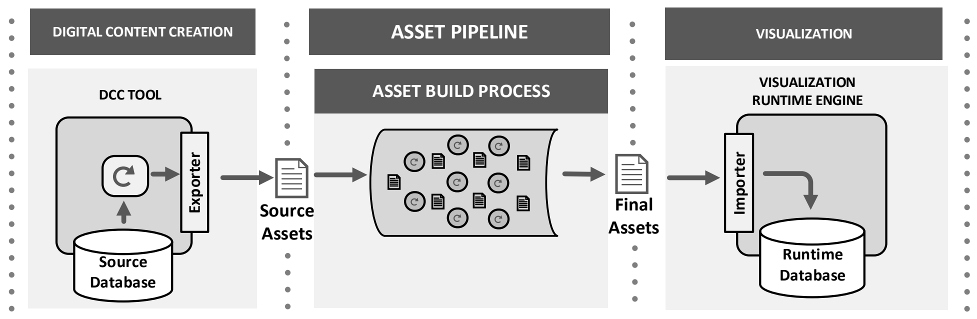


Figure 4.8: ASSET BUILD PROCESS component pattern diagram

Context:

The source assets require data conditioning to fulfil the requirements of the real-time visualization runtime.

The transformation steps required to transfer an Autodesk Revit BIM file into an architectural visualization are numerous and complex. Autodesk Revit is a high-level DCC tool operating against a central working repository. The Revit data source contains a large amount of data that is not required for visualization purposes.

The source asset requires transformation and processing prior to integration with the Unity architectural visualization tool.

Problem

How to efficiently deliver interactive visualizations where the source assets are created in a high-level DCC tool and the visualization operates within a low-level optimized runtime?

The Autodesk Revit format ".RVT" cannot be directly imported into the Unity game engine. Unity supports importing three-dimensional meshes from two different types of files. Firstly, exported three-dimensional formats such as FBX, DAE (COLLADA), DXF, OBJ and SKP files. Secondly, proprietary three-dimensional DCC application files such as MAX and BLEND file formats from 3dsMax and Blender. The chosen INTERMEDIATE FILE FORMAT is Autodesk FBX. However, a process is required to ensure all BIM semantics, properties and materials are converted appropriately for finally loading into the Unity game engine.

Solution

Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.

Initially the source BIM model is exported from Revit as the FBX file format. This is directly imported into Unity and the resulting building is placed in a scene. This workflow is illustrated in figure 4.9. Although the three-dimensional geometry of the building is present, the materials and colours are absent. The single stage pipeline would not suffice.

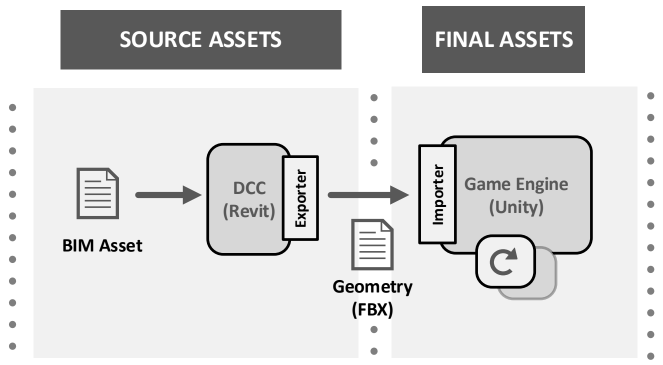


Figure 4.9: Single stage asset pipeline

The solution provided is to use a workflow whereby the FBX geometric file was is passed through an intermediate conversion stage in the form of 3dsMax. Being from the same vendor, Autodesk, 3dsMax can reference the materials and colours of the geometry in the Revit file. Subsequently, the scripting language of 3dsMax, MaxScript, is used to design and create a tool to perform the conversion. That is, for each AutoDesk material a Standard material is instantiated as illustrated in figure 4.10. The corresponding properties such as colour, texture and bump maps are copied across and any dependent objects are updated to reference the new Standard material.

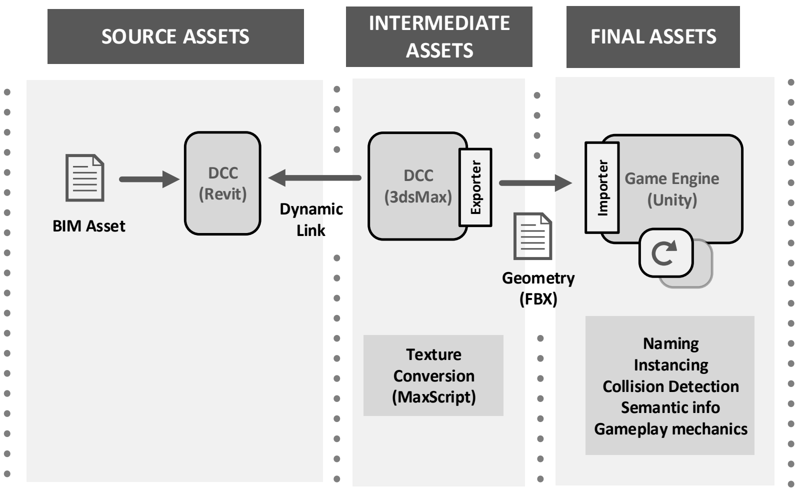


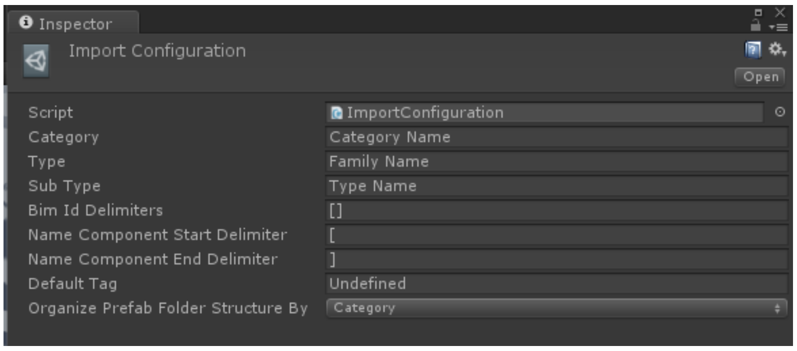
Figure 4.10: Intermediate processing of materials

Unity provides a mechanism to custom process the "user properties" during an import of the FBX file. This functionality is contained within the "AssetPostprocessor" and is leveraged within the prototype to extract the BIM metadata for model elements. The metadata is used to provide the functionality described in the following sections.

Object Nomenclature

The BIM representation defines objects in a hierarchy of categories, families and types (Eastman, 2011). Categories are the broadest classification and consist of a fixed set, examples being walls, doors and roofs. Each "category" has a class of elements associated with it known as "families" and within those "types" define the overall behaviour and the characteristics of a BIM object. Based on this knowledge the imported architectural elements were renamed for ease of identification within the Unity editor so "undefined2door273" becomes "[Doors][Internal][730 x 2110][273]".

A Unity tool was developed to allow the visualization designer flexibility in the naming configuration prior to import. The structure of the name along with the component delimiters can be specified, figure 4.11.

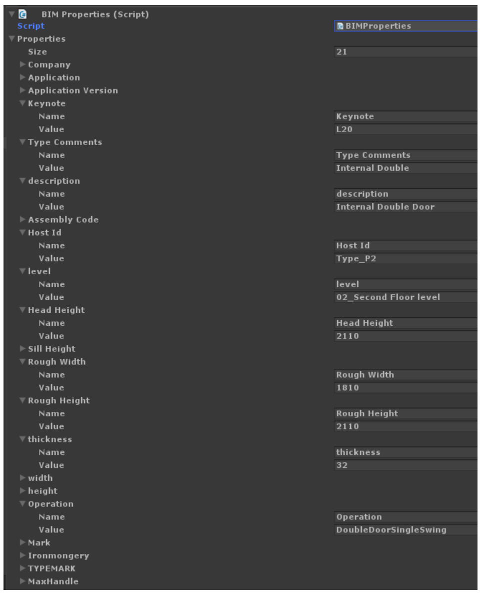


Script	ImportConfiguration
Category	Category Name
Type	Family Name
Sub Type	Type Name
Bim Id Delimiters	[]
Name Component Start Delimiter	[
Name Component End Delimiter	]
Default Tag	Undefined
Organize Prefab Folder Structure By	Category

Figure 4.11: Configuration of object names

BIM Properties

Significantly the semantic information for each architectural element is imported into Unity and stored within the Resource Manager of the game engine. This information can be used in the future to drive enhanced visualizations or simulations. The Unity property visualizer, figure 4.12, illustrates the parametric building information associated with a single door, imported from the BIM test model.

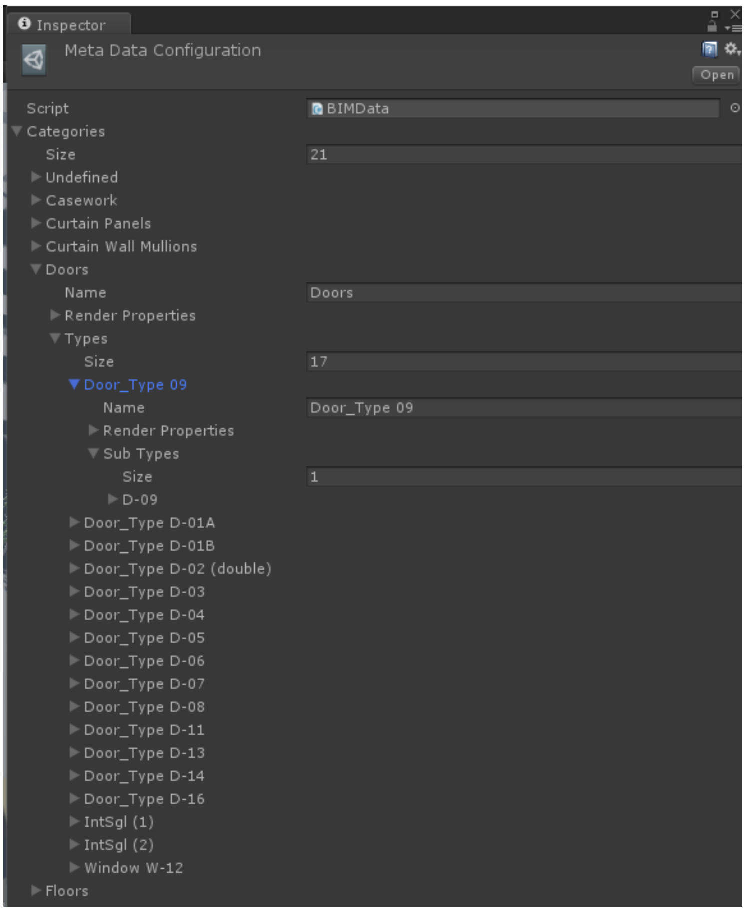


```
▼ BIM Properties (Script)
  Script                              BIMProperties
▼ Properties
    Size                              21
  ▶ Company
  ▶ Application
  ▶ Application Version
  ▼ Keynote
      Name                            Keynote
      Value                           L20
  ▼ Type Comments
      Name                            Type Comments
      Value                           Internal Double
  ▼ description
      Name                            description
      Value                           Internal Double Door
  ▶ Assembly Code
  ▼ Host Id
      Name                            Host Id
      Value                           Type_P2
  ▼ level
      Name                            level
      Value                           02_Second Floor level
  ▼ Head Height
      Name                            Head Height
      Value                           2110
  ▶ Sill Height
  ▼ Rough Width
      Name                            Rough Width
      Value                           1810
  ▼ Rough Height
      Name                            Rough Height
      Value                           2110
  ▼ thickness
      Name                            thickness
      Value                           32
  ▶ width
  ▶ height
  ▼ Operation
      Name                            Operation
      Value                           DoubleDoorSingleSwing
  ▶ Mark
  ▶ Ironmongery
  ▶ TYPEMARK
  ▶ MaxHandle
```

Figure 4.12: BIM semantic properties

BIM Class Hierarchy

The BIM semantic class hierarchy, containing the relationships between Category, Family and Type, is automatically constructed on import of the model and stored in the resource manager database of Unity. It is proposed this pre-computed hierarchy would be represented within a visualization as a tree structure to assist the user in performing tasks such as enabling and disabling building elements. The representation of the BIM class hierarchy, including an expansion of a door type, is shown in figure 4.13.



```
Inspector
Meta Data Configuration                                  Open

Script                                BIMData
▼ Categories
    Size                              21
  ▶ Undefined
  ▶ Casework
  ▶ Curtain Panels
  ▶ Curtain Wall Mullions
  ▼ Doors
      Name                            Doors
    ▶ Render Properties
    ▼ Types
        Size                          17
      ▼ Door_Type 09
          Name                        Door_Type 09
        ▶ Render Properties
        ▼ Sub Types
            Size                      1
          ▶ D-09
      ▶ Door_Type D-01A
      ▶ Door_Type D-01B
      ▶ Door_Type D-02 (double)
      ▶ Door_Type D-03
      ▶ Door_Type D-04
      ▶ Door_Type D-05
      ▶ Door_Type D-06
      ▶ Door_Type D-07
      ▶ Door_Type D-08
      ▶ Door_Type D-11
      ▶ Door_Type D-13
      ▶ Door_Type D-14
      ▶ Door_Type D-16
      ▶ IntSgl (1)
      ▶ IntSgl (2)
      ▶ Window W-12
  ▶ Floors
```

Figure 4.13: BIM class hierarchy

Object behaviour

In a process that occurs post-import the behaviours of the building components are automatically assigned based on their BIM category. This novel approach has not been implemented before in research. Walls, floors and other solid surfaces were assigned mesh colliders ensuring the physics engine prevents a user from navigating through them. Whereas, doors were assigned spherical colliders whose event was triggered when a user passed within a certain radius. The event triggered the temporary removal of the door to allow the user passage through.

Object instances

A potential performance optimization that was incorporated into the prototype is that of geometry instancing (Zink, Pettineo and Hoxley, 2011). Importing the BIM FBX model into Unity natively creates multiple instances of the same geometric object, albeit at different translations and rotations. The test BIM model contains over 170 instances of the same chair each consisting of the geometric mesh along with the position in world space. Utilizing the "AssetPostProcessor" duplicate instances of the same object are recognized and a Unity prefab is created forming a template game object. From this prefab individual instances of the objects are created with their individual rotation and position. A technique reserved for video game production; further research is required to understand whether this approach provides performance gains in this scenario.

Asset Build Process

Given the above transformations, the following sequential asset build node stages form the asset build process:

1. Intermediate processing of materials using the foundation of Autodesk 3dsMax. The intermediate file format of FBX is updated to include corresponding FBX standard materials with corresponding colour, texture and bump maps.
2. Update of building component nomenclature to include the BIM classification, families and types of an object.
3. Transferral of BIM semantic information for each architectural element. The Unity property visualizer provides a mechanism for viewing the parametric building information.
4. Assigning relevant object behaviour to the BIM game object runtime objects.
5. Instancing common geometry into Unity game object prefabs for general instantiation.

4.11 Summary and Evaluation

Instantiation of the ASSET PIPELINE architectural pattern and component patterns gives rise to the asset pipeline workflow illustrated in figure 4.14.

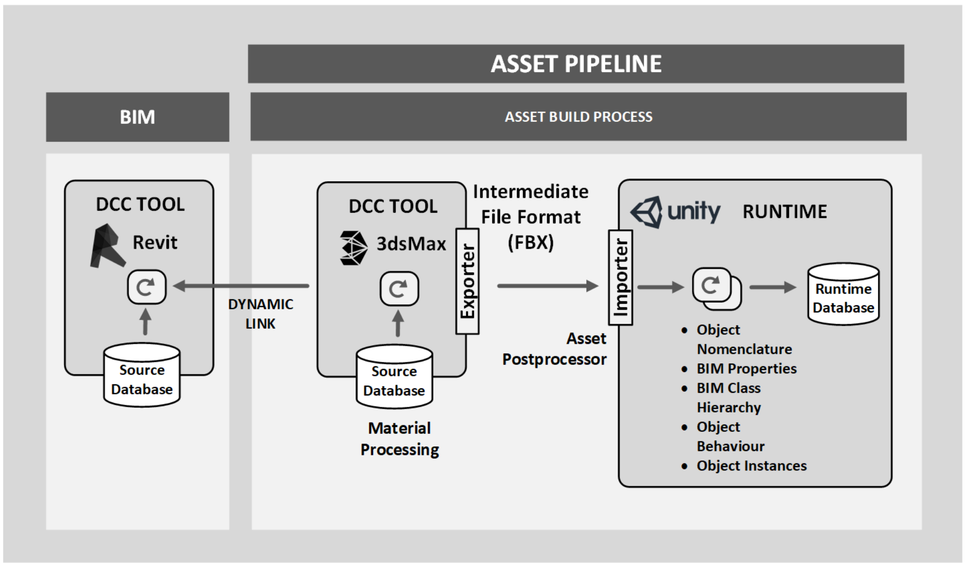


Figure 4.14: BIM Visualization ASSET PIPELINE

A dynamic link associating the DCC tools 3dsMax and Revit is initiated. This data conduit enables 3dsMax to reference the geometry, materials and semantic data of the BIM model. The first stage of the ASSET BUILD process converts all materials using MaxScript into standard materials interoperable with Autodesk FBX. A DCC EXPORT is performed to extract the geometric model and properties from 3dsMax into the chosen INTERMEDIATE FILE FORMAT of Autodesk FBX. An import of the model data into the Unity runtime invokes an import postprocessor initiating further stages of the ASSET BUILD PROCESS. The following asset build node steps are sequentially performed: object nomenclature, BIM properties transferal, BIM class hierarchy generation, assignment of object behaviour and finally instancing of common geometry. This concludes the ASSET PIPELINE.

The example BIM visualization is shown in figure 4.15. The visualization uses a custom generated first-person controller allowing an end-user to navigate the scene and interrogate the properties of the BIM objects. It is envisaged the ASSET PIPELINE workflow will allow architects and other stakeholders the flexibility to produce visualizations targeted to their area of need.

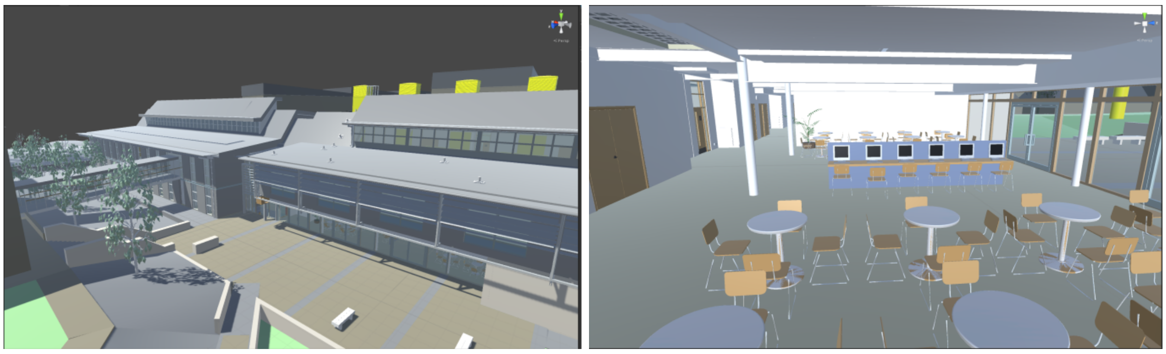


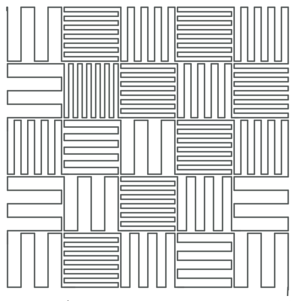
Figure 4.15: BIM Visualization generated from ASSET PIPELINE instantiation

The novel approach of applying the ASSET PIPELINE pattern to architectural BIM source asset data has resulted in a visualization that is both rich in semantic BIM data and can leverage the advanced capabilities of the game engine runtime.

The building components in the form of the three-dimensional geometry and their associated BIM semantic data are mapped to the component-based architecture inherent in the Unity game engine. This powerful mapping is important since it exposes the BIM semantic data for further levels of research and interrogation. An opportunity now exists to extend the research reach into the further area of building simulations. Possible simulations to consider are the consequences of fire escape location, building flow and building performance. The architectural visualization (ArchViz) produced in this research leveraged the rendering and physics engine encapsulated within the game engine. Further research incorporating simulations may leverage further components of the game engine runtime such as the artificial intelligence module.

Although application of the ASSET PIPELINE workflow in the instantiation of an architectural visualization (ArchViz) provides many benefits it is not without its limitations. The asset pipeline as defined consists of a unidirectional push model. The AEC designer instigates the asset pipeline workflow resulting in the rich architectural visualization. Any changes performed within the game engine editor are not propagated back into the originating source asset data residing in the DCC tool, Autodesk Revit. For example, changing characteristics of the building interior within the game engine editor such as position of furniture or the materials associated with a surface does not update the associated BIM data. Such functionality would require a bi-directional asset pipeline, a workflow outside the focus of this research.

5 Use Case – Graph Visualization



Generative Pattern 5 – J Lear 2020

Graphs can be utilized across a number of domains and they differ from the standard relational model of data. A graph, also known as a network, is a model of data that contains nodes that are discrete data elements and edges which represent the relationships between the nodes (Lanum, 2016; Vehkalahti, 2008). The edges take one of two forms; either directed or undirected. Both nodes and edges can contain properties persisted as key-value pairs (Lanum, 2016). The number of links from a node is known as the degree of the node (Lanum, 2016). The graph model highlights the relationships that may be hidden in a relational, tabular view of the same data. The relationships form the core part of the data structure thus providing the ability to recognize patterns in the data that would not otherwise be apparent (Lanum, 2016). Much of the big data revolution has been in understanding trends in the aggregate data it is important to understand the previously unknown connections and relationships between the individual data elements; and network graph visualization provides such a lens.

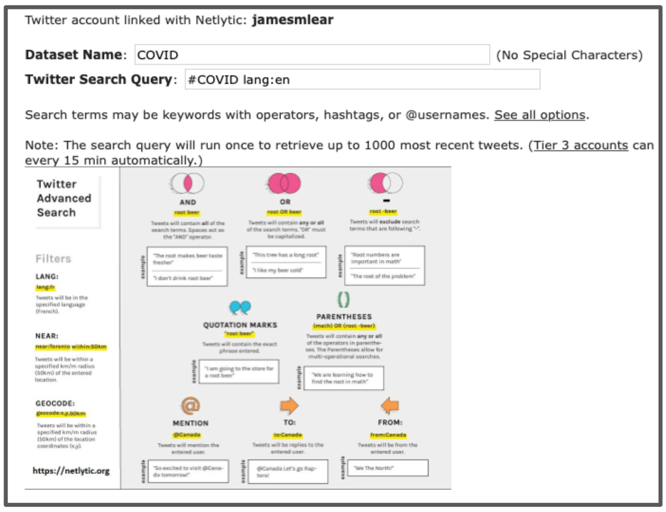
This use case provides an exploratory graph visualization based on the sub-category of data visualization known as Social Network Analysis (SNA). Social network analysis is a research field that aims to use analytic algorithms to understand social dynamics within a group. The use case scenario maps and visualizes the social network Twitter. Twitter is now established as the second most important social network in the world after Facebook (Bruns, 2012). The 140-character updates are designed for succinct messaging and its structure is a flat format; the messages are either public and visible to all or are private and visible to approved followers of the sender. Twitter does not encapsulate further degrees of connection such as family, friends or friends of friends. Instead the end-users of Twitter have developed two effective mechanisms for circumventing these limitations. The first, known as the "#hashtag" which provides manual and automatic collection of all tweets using the Twitter application and facilitating an end-user to subscribe to a particular content feed. The second, is the "@replies" semantic which allows a sender to direct public messages to users, regardless of whether they follow them (Huberman, Romero and Wu, 2008).

Twitter Graph Visualization

This use case instantiates the asset pipeline pattern language to produce a real-time exploratory Twitter graph visualization. The graph visualization concerns the discussion of the Coronavirus (COVID-19) identified with the specific hashtag "#COVID" (2020). In the context of social media, the exploratory visualization seeks to understand how individuals interact within loosely connected information networks through the exchange of digital artifacts such as Twitter tweets (Rainie, 2014). The analysis and visualization concerns the "@reply" networks existing between participating users as a time-constrained static network. Study of "@reply@ patterns within all tweets marked with a specific hashtag provides a framework for the identification of the most central users within the topic and provides a catalyst for further exploratory evaluation.

5.1 Data Acquisition

Data acquisition is performed using the cloud-based research tool Netlytic (2020). Under a research license "Tier 2" account an end-user is permitted access to ten thousand twitter records over a maximum of five datasets. The Netlytic importer uses the Twitter public RESTful API v1.1 for authentication and data retrieval. The first step in dataset acquisition is the specification of the search query, #COVID, as illustrated in figure 5.1.



Twitter account linked with Netlytic: **jamesmlear**

Dataset Name: COVID (No Special Characters)
Twitter Search Query: #COVID lang:en

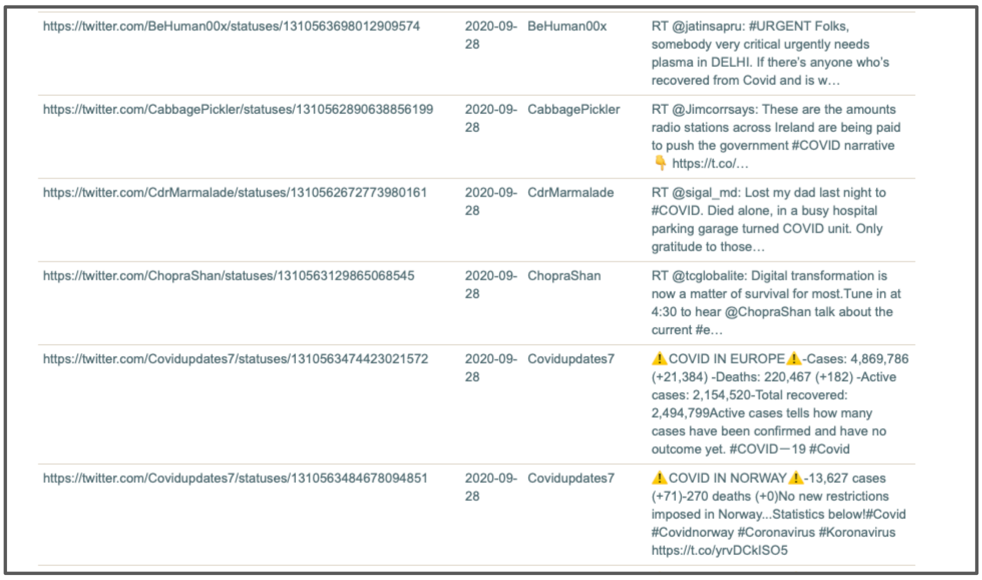
Search terms may be keywords with operators, hashtags, or @usernames. See all options.

Note: The search query will run once to retrieve up to 1000 most recent tweets. (Tier 3 accounts can every 15 min automatically.)

Twitter Advanced Search
Filters
LANG:
lang:fr
Tweets will be in the specified language (French).
NEAR:
near:Toronto within:50km
Tweets will be within a specified km/m radius (50km) of the entered location.
GEOCODE:
geocode:x,y,50km
Tweets will be within a specified km/m radius (50km) of the location coordinates (x,y).
https://netlytic.org
AND
root beer
Tweets will contain all of the search terms. Spaces act as the "AND" operator.
example
"The root makes beer taste fresher"
"I don't drink root beer"
OR
root OR beer
Tweets will contain any or all of the search terms. "OR" must be capitalized.
example
"This tree has a long root"
"I like my beer cold"
-
root -beer
Tweets will exclude search terms that are following "-".
example
"Root numbers are important in math"
"The root of the problem"
QUOTATION MARKS
"root beer"
Tweets will contain the exact phrase entered.
example
"I am going to the store for a root beer"
PARENTHESES
(math) OR (root -beer)
Tweets will contain any or all of the operators in parentheses. The Parentheses allow for multi-operational searches.
example
"We are learning how to find the root in math"
MENTION
@Canada
Tweets will mention the entered user.
example
"So excited to visit @Canada tomorrow!"
TO:
to:Canada
Tweets will be replies to the entered user.
example
@Canada Let's go Raptors!
FROM:
from:Canada
Tweets will be from the entered user.
example
"We The North!"

Figure 5.1: Netlytic (2020) data acquisition using the hashtag #COVID search query

A preview provides the last one hundred records to ensure the import accuracy, figure 5.2.



https://twitter.com/BeHuman00x/statuses/1310563698012909574	2020-09-28	BeHuman00x	RT @jatinsapru: #URGENT Folks, somebody very critical urgently needs plasma in DELHI. If there's anyone who's recovered from Covid and is w…
https://twitter.com/CabbagePickler/statuses/1310562890638856199	2020-09-28	CabbagePickler	RT @Jimcorrsays: These are the amounts radio stations across Ireland are being paid to push the government #COVID narrative 👇 https://t.co/…
https://twitter.com/CdrMarmalade/statuses/1310562672773980161	2020-09-28	CdrMarmalade	RT @sigal_md: Lost my dad last night to #COVID. Died alone, in a busy hospital parking garage turned COVID unit. Only gratitude to those…
https://twitter.com/ChopraShan/statuses/1310563129865068545	2020-09-28	ChopraShan	RT @tcglobalite: Digital transformation is now a matter of survival for most.Tune in at 4:30 to hear @ChopraShan talk about the current #e…
https://twitter.com/Covidupdates7/statuses/1310563474423021572	2020-09-28	Covidupdates7	⚠️COVID IN EUROPE⚠️-Cases: 4,869,786 (+21,384) -Deaths: 220,467 (+182) -Active cases: 2,154,520-Total recovered: 2,494,799Active cases tells how many cases have been confirmed and have no outcome yet. #COVID—19 #Covid
https://twitter.com/Covidupdates7/statuses/1310563484678094851	2020-09-28	Covidupdates7	⚠️COVID IN NORWAY⚠️-13,627 cases (+71)-270 deaths (+0)No new restrictions imposed in Norway...Statistics below!#Covid #Covidnorway #Coronavirus #Koronavirus https://t.co/yrvDCkISO5

Figure 5.2: Netlytic (2020) table of tweets containing #COVID displayed in relational format

The data set consists of flat relational data, whereas, a graph network requires connections between the data elements. Twitter has a concrete and well-defined relationships based on user interactions. The following relationships exist:

- ***Users follow users.*** Each user has a list of users they follow.
- ***Users tweet tweets.*** Each generated tweet can be directly associated with the user who created it.
- ***Tweet references a user.*** Optionally, a tweet contains a direct reference to another user.

- ***Tweet responds to tweet.*** Optionally, a tweet is a direct response to another tweet.
- ***Tweet retweets tweet.*** Optionally, a tweet encapsulates a previous tweet known as a retweet.

The entities and relationships give rise to the semantic conceptual model illustrated in figure 5.3.

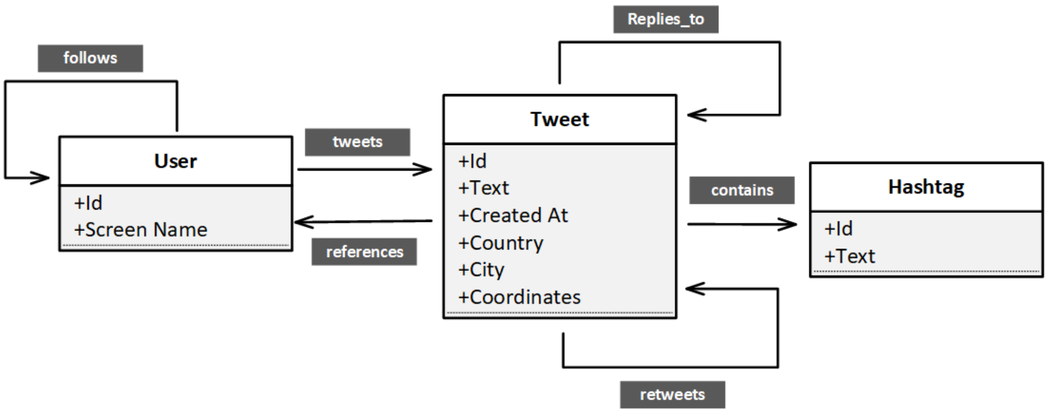
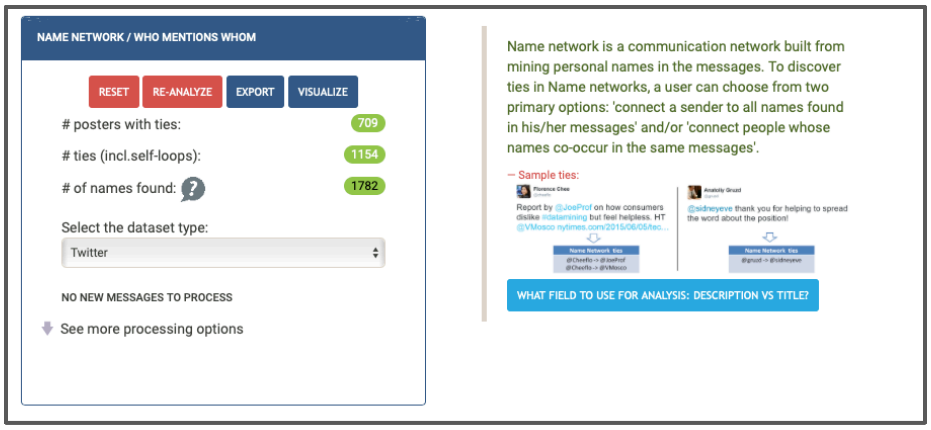


Figure 5.3: Conceptual model of Twitter semantics

Given the underlying semantic relationships the initial data acquisition using Netlytic (2020) is further processed to generate a network graph. The process reviews each tweet in the dataset encapsulating the poster and all referenced nodes. Each account translates to a ”Üse” node with associated properties and each tweet that mentions other user accounts is represented as a link between the poster and the other accounts. The choice of options for which is displayed in figure 5.4.



NAME NETWORK / WHO MENTIONS WHOM

RESET RE-ANALYZE EXPORT VISUALIZE

posters with ties: 709

ties (incl.self-loops): 1154

# of names found: 1782

Select the dataset type: Twitter

NO NEW MESSAGES TO PROCESS

See more processing options

Name network is a communication network built from mining personal names in the messages. To discover ties in Name networks, a user can choose from two primary options: 'connect a sender to all names found in his/her messages' and/or 'connect people whose names co-occur in the same messages'.

– Sample ties:

Florence Chee

Report by @JoeProf on how consumers dislike #datamining but feel helpless. HT @VMosco nytimes.com/2015/06/05/tec...

Name Network ties

@Cheeflo -> @JoeProf
@Cheeflo -> @VMosco

Anatoliy Gruzd

@sidneyeve thank you for helping to spread the word about the position!

Name Network ties

@gruzd -> @sidneyeve

WHAT FIELD TO USE FOR ANALYSIS: DESCRIPTION VS TITLE?

Figure 5.4: Netlytic (2020) network analysis

5.2 Graph Layout

An initial survey of graph visualization tools was undertaken as detailed in table 5.1. The open source graph visualization tool Gephi (Bastian, Heymann and Jacomy, 2009) was chosen for initial graph layout of the Twitter dataset. In essence Gephi is the Digital Content Creation (DCC) tool augmenting the Twitter dataset with design elements such as the graph layout.

Graph Visualization DCC Tool	URL	Description
Cytoscape.js	https://cytoscape.org	Open source JavaScript library; primarily used for biological network visualization and analysis (Lotia et al., 2013).
D3.js	https://d3js.org	Open source JavaScript library; graph visualization and HTML, SVG and CSS rendering. Data driven approach.
Gephi	https://gephi.org	Open source DCC tool; graph and network visualization. Exploratory data analysis (Bastian, Heymann and Jacomy, 2009). Provides plugin layout functionality.
GraphViz	https://graphviz.org	Open source DCC tool; legacy graph and network visualization for software documentation.
KeyLines	https://cambridge-intelligence.com/keylines/	Commercial DCC library; Graph visualization for JavaScript and React developers.

Table 5.1: Graph visualization tool survey

A number of different layout algorithms are built into Gephi. The class of graph drawing techniques most useful for data visualization are force directed techniques (Vehkalahti, 2008). Initial layout of the data set uses the spatial layout algorithm, "Force Atlas 3D" (Levallois, 2013). Gephi provides a visual representation of the layout, however, does not associate the node properties and does not provide three-dimensional real-time exploratory navigation of the data set, figure 5.5.

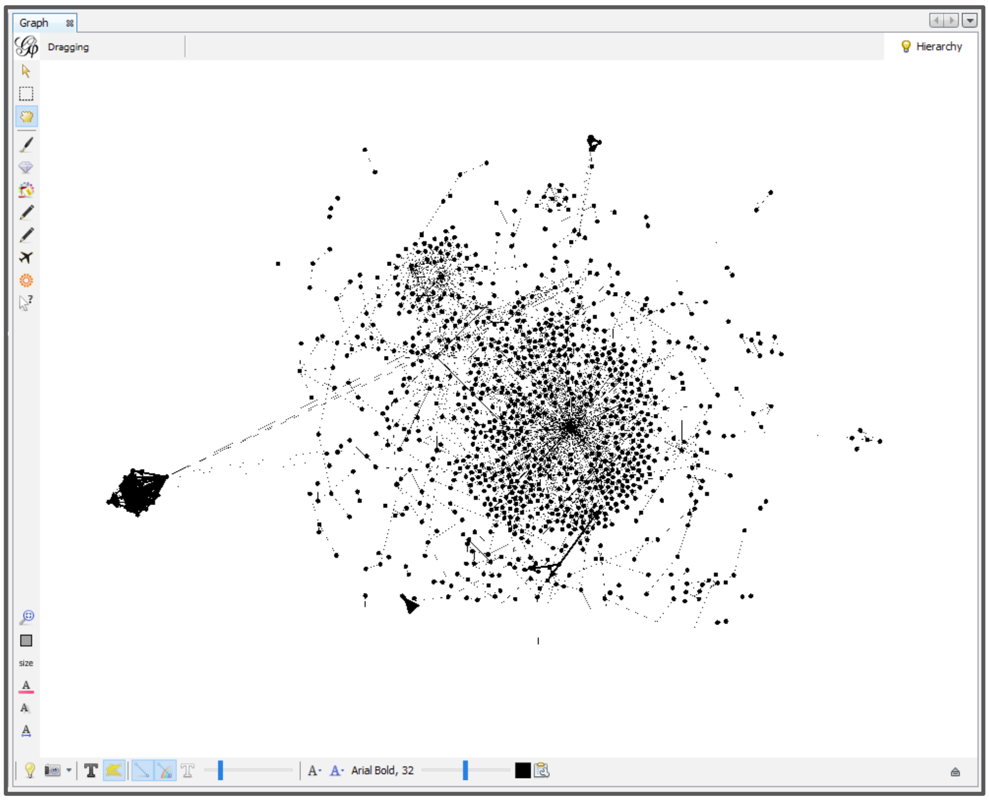


Figure 5.5: Gephi - Initial graph visualization using Force Atlas 3D.

5.3 Use Case Prototype

This novel approach uses a Gephi to Unity pipeline in the production of an exploratory runtime graph visualization providing detailed node attributes and rich three-dimensional navigation functionality. A correlation exists between the exploratory runtime graph visualization and the game asset pipeline; both take source assets in the form of a three-dimensional model and manipulate the assets into a form acceptable for executing in the game engine runtime.

An opportunity exists to enrich the exploratory visualizations of graph visualization tools, such as Gephi, to integrate ideas from the game asset pipeline to:

1. Produce rapid design iterations to allow runtime exploratory visualizations.
2. Use the enhanced rendering capability of runtime game engines. That is the software features of three-dimensional rendering, navigation and detailed information overlays and the hardware technological stack of XR capabilities.
3. Increase the usability and impact for various audiences.

The following sections detail the instantiation of the asset pipeline pattern and component patterns in the production of such a visualization. Specifically referring to the ASSET PIPELINE pattern definition in section 3.6.1 and aligning the pattern sections to the problem.

5.4 ASSET PIPELINE

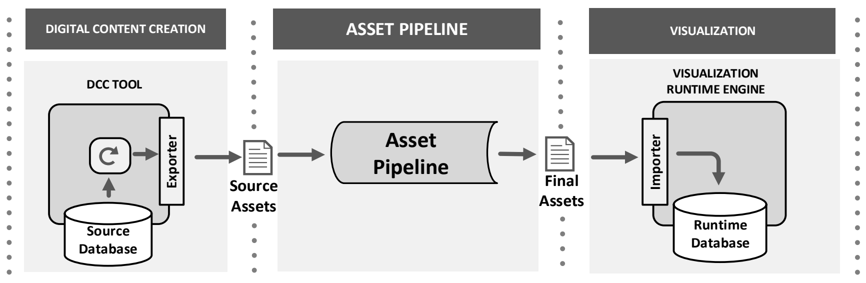


Figure 5.6: ASSET PIPELINE architectural pattern diagram

<u>**Context**</u>

Designers require a workflow to assist them viewing their digital content in an interactive real-time visualization.

The open-source data visualization tool Gephi (Bastian, Heymann and Jacomy, 2009) provides a mechanism to visualize network graph data using a number of layout algorithms. However, firstly the Netlytic (2020) to Gephi workflow lacks the node attributes; essential for understanding the dataset. Secondly, Gephi does not afford the three-dimensional real-time rendering and navigation capability of a game engine such as Unity. Designers require a workflow to assist them in viewing and exploring such layouts.

<u>Problem</u>

How to efficiently deliver interactive visualizations where the source assets are created in a high-level DCC tool and the visualization operates within a low-level optimized runtime?

Twitter data acquisition and network analysis has been processed using Netlytic (2020). The high-level Digital Content Creation (DCC) tool in this instance is the data visualization tool Gephi. Initial layout of the graph visualization will be achieved using the "Force Atlas 3D" algorithm (Levallois, 2013). The exploratory visualization runtime is operating within the low-level game engine Unity.

The workflow of creating the visualization consists of creating the source assets within the Digital Content Creation Tool Gephi, transforming the source assets into a compatible form utilized by the Unity runtime game engine and subsequently performing exploratory data analysis using the visualization. After such analysis either further slices of the Twitter data set require investigation (Netlytic) or a visual layout change is required (Gephi). Any form of automation increases the efficiency of the designer or data-scientist.

<u>Solution</u>

Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.

Instantiation of the ASSET PIPELINE pattern is formed by following the sequential application of the component patterns: DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS as illustrated in figure 5.7 and described below.

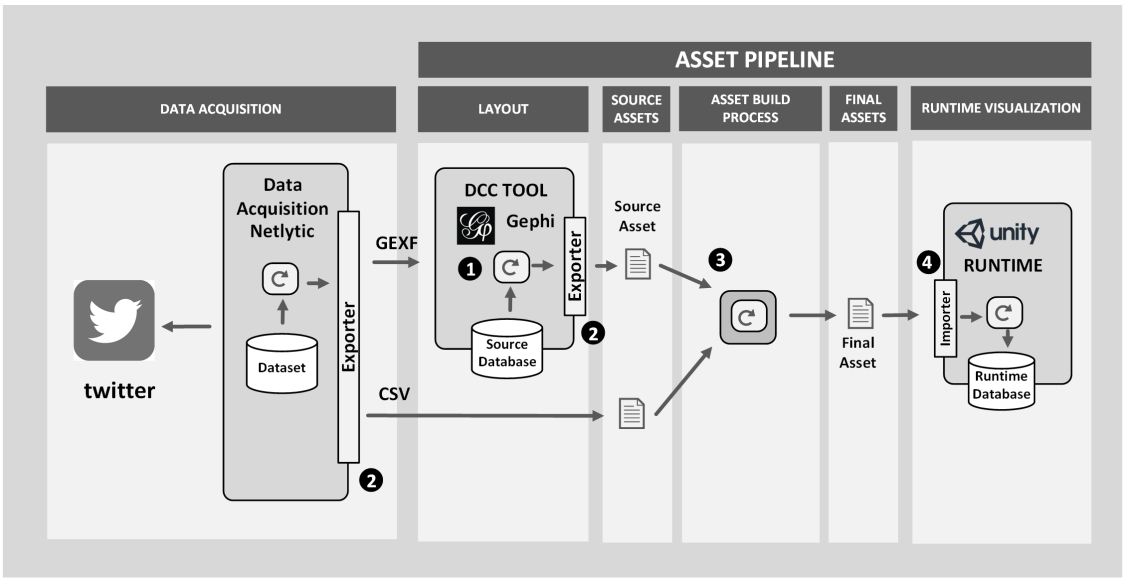


Figure 5.7: Graph Visualization (GraphViz) Asset Pipeline

<u>Step 1</u>

The high-level Gephi Data Visualization tool is used to layout the Twitter dataset, initially applying the "Force Atlas 3D" algorithm (Levallois, 2013). The source asset is persisted in a proprietary binary format with file extension of ".gephi".

Step 2

Two architectural endpoints exist for data export. Firstly, the binary proprietary format of Gephi is exposed to further processing within the asset pipeline via the use of an exporter. Secondly, the properties associated with individual Twitter user accounts is exported from Netlytic. Please refer to the component patterns DCC EXPORT and INTERMEDIATE FILE FORMAT for a further discussion.

Step 3

The build process manipulates both the Gephi exported source asset and Twitter user properties into a format acceptable for loading into the Unity runtime game engine. Please refer to the ASSET BUILD PROCESS for a complete explanation.

Step 4

The result of the ASSET BUILD PROCESS is the optimization of the source assets into their final asset form acceptable for the Unity runtime game engine. A custom importer is used to load the final assets. A Unity exploratory tool is created to navigate the data set.

5.5 DCC EXPORT

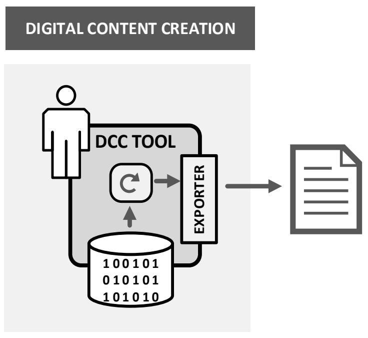


Figure 5.8: DCC EXPORT component pattern diagram

Context:

Designers require a mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing.

Gephi (Bastian, Heymann and Jacomy, 2009) natively supports multiple layout options and provides the capability to extend these with the support for a plugin mechanism. The designer uses the visualization capability of Gephi to review and refine the choice of layout options, and ultimately settles on the “Force Atlas 3D” algorithm (Levallois, 2013). At this stage the designer needs to review the visualization within the runtime Unity game engine for further exploratory navigation. However, the Gephi source asset requires further processing prior to integration into the runtime Unity game engine. The designer requires a mechanism to extract the proprietary network graph data in a format expected by the ASSET BUILD PROCESS.

Problem:

How is data extracted from a DCC tool where the internal DCC source database is persisted in a proprietary format?

The source asset structure is persisted in an opaque proprietary format within the source database of the Gephi data visualization tool. In its native format the source format structure is difficult to parse. Further stages in the ASSET PIPELINE workflow require the data to be in a readable format suitable for subsequent processing.

Solution:

A DCC tool exporter is developed to permit the source asset data to be extracted from the source database into a more usable format.

In this use case scenario, the recommendation of developing an exporter is not required. Gephi supports exporting the file formats identified in table 5.2.

File Extension	File Format	Description
DL	UCINET	UCINET is a software package for the analysis of social network data. Gephi supports the full matrix and edgelist1 sub-formats.
GDF	GUESS	GUESS is an exploratory data analysis and visualization tool for graph networks.
GEXF	Graph Exchange XML Format	Extensible and open XML format for describing complex network structures.
GML	Graph Modeling Language	Text file format supporting network data used by the visualization tools Graphlet, Pajek, yEd, LEDA and NetworkX.
GRAPHML	GraphML	XML structured supporting attributes for nodes and edges, hierarchical graphs and is flexible in approach.
NET	Pajek NET	Supports representation of network topology, however, lacks attribute support.
VNA	Netdraw VNA	Similar to Pajek format, however, also supports node attributes.

Table 5.2: Gephi export file format

The properties associated with individual Twitter tweets can be exported from Netlytic using the container format Comma-Separated Values (CSV). The Comma-Separated Value (CSV) format, although not fully standardized, can be parsed by a large number of programming languages.

Please refer to application of the INTERMEDIATE FILE FORMAT for a discussion of the common container format derived for subsequent processing in the ASSET BUILD PROCESS.

5.6 INTERMEDIATE FILE FORMAT

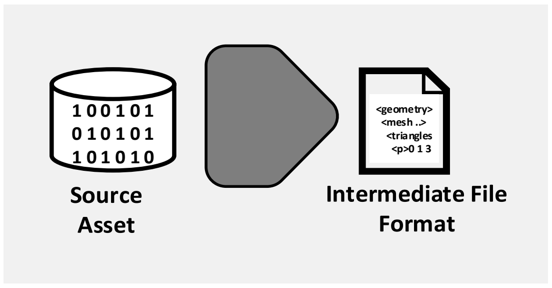


Figure 5.9: INTERMEDIATE FILE FORMAT component pattern diagram

Context:

Designers are using a variety of DCC tools to create source assets, each with their own storage format.

The graph visualization is sourced from two sets of assets; Gephi supplies the network graph layout and Netlytic supplies the Twitter tweet properties associated with the graph nodes. Designers of the graph visualization DCC EXPORT their source assets and execute the ASSET BUILD PROCESS expecting the workflow to understand the different content formats.

Problem:

How is the DCC tool file format decoupled from the ASSET BUILD PROCESS to introduce reusability and flexibility in the pipeline?

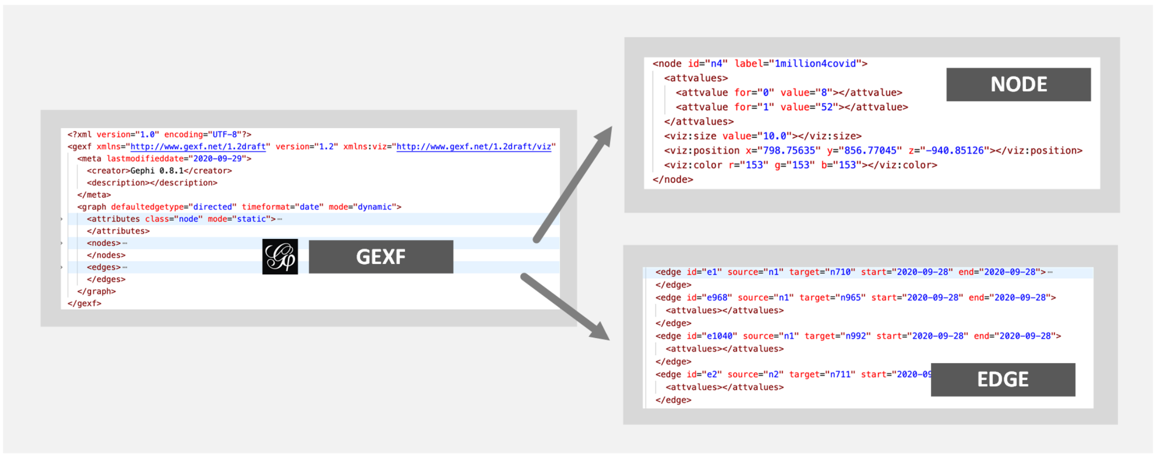
Use of multiple asset formats within the ASSET BUILD PROCESS results in an inflexibility in the individual build steps. Processing steps required to be applied to both sets of assets cannot be easily shared.

Solution:

Introduce a unified intermediate file format to increase flexibility and reusability within the ASSET BUILD PROCESS.

The graph Exchange XML Format (GEXF) is chosen as the common intermediate file format. Initiated in 2007 the Gephi file format is an established language based on XML for encapsulating complex network structures and their associated data. The format is both extensible and open consisting of an XML schema for validation purposes. Libraries exist for many popular programming languages including C++, R, Java, Python and JavaScript.

The layout information including three-dimensional spatial information is exported from Gephi in the GEXF format. A graph of individual nodes and related edges is illustrated in figure 5.10. Individual node properties relating to the Twitter tweet are not encapsulated.



```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2" xmlns:viz="http://www.gexf.net/1.2draft/viz"
  <meta lastmodifieddate="2020-09-29">
    <creator>Gephi 0.8.1</creator>
    <description></description>
  </meta>
  <graph defaultedgetype="directed" timeformat="date" mode="dynamic">
    <attributes class="node" mode="static">⋯
    </attributes>
    <nodes>⋯
    </nodes>
    <edges>⋯
    </edges>
  </graph>
</gexf>
```

```
<node id="n4" label="1million4covid">
  <attvalues>
    <attvalue for="0" value="8"></attvalue>
    <attvalue for="1" value="52"></attvalue>
  </attvalues>
  <viz:size value="10.0"></viz:size>
  <viz:position x="798.75635" y="856.77045" z="-940.85126"></viz:position>
  <viz:color r="153" g="153" b="153"></viz:color>
</node>
```

```
<edge id="e1" source="n1" target="n710" start="2020-09-28" end="2020-09-28">⋯
</edge>
<edge id="e968" source="n1" target="n965" start="2020-09-28" end="2020-09-28">
  <attvalues></attvalues>
</edge>
<edge id="e1040" source="n1" target="n992" start="2020-09-28" end="2020-09-28">
  <attvalues></attvalues>
</edge>
<edge id="e2" source="n2" target="n711" start="2020-09
  <attvalues></attvalues>
</edge>
```

Figure 5.10: Exported Gephi graph format GEXF

The tweets associated with the nodes are exported from Netlytic in Comma-Separated Value (CSV) format. Each record in the file represents an individual node and thus a message tweet. The attributes or properties contained in the record are detailed in table 5.3.

Property	Description
ID	The unique graph node ID for the tweet.
Link	The public URL of the tweet.
PubDate	Publication date or timestamp of the tweet.
Author	Twitter account name and author of the tweet.
Title	Title of the tweet or retweet.
Description	Main body of the tweet, in 140 characters or less.
Source	Name or IRL of the tool used for tweeting (Android, iPhone, WebApp).
ProfileImage	URL of the tweet profile picture of the sender.
UserFriendsCount	Number of friends the user is associated with.
UserFollowersCount	Number of followers the user is related to.
UserLocation	Human readable form of the location of the user.
Lang	ISO language code of the senders default language.

Table 5.3: Netlytic tweet format

The intermediate file format GEXF provides a mechanism to add data and meta-data to topology elements. The Netlytic tweet properties associated with a node are merged into the intermediate file format as XML attributes. This manipulation is performed in the first stage of the ASSET BUILD PROCESS. Subsequent processing may then take advantage of the common intermediate file format.

5.7 ASSET BUILD PROCESS

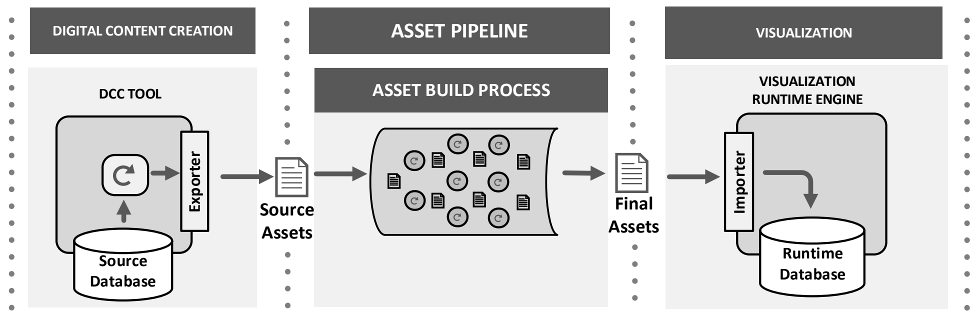


Figure 5.11: ASSET BUILD PROCESS component pattern diagram

Context:

The source assets require data conditioning to fulfil the requirements of the real-time visualization runtime.

A DCC EXPORT has been performed within Gephi and Netlytic to extract both the graph layout configuration and the graph node attributes. The corresponding file formats of GEXF and CSV are used as the source asset format. Prior to integration into the Unity runtime game engine the source assets require data cleansing and transformation into the final asset format.

Problem:

How can the source assets be transformed into final assets in preparation for importing into the runtime?

The Unity primary scripting API is the language C# and this language has native support for parsing and manipulating the JavaScript Object Notation (JSON). The final format as chosen for the use case is JSON. However, the original source asset formats (GEXF, CSV) are incompatible with the final runtime format.

Solution:

Integration of an Asset Build Process.

The Asset Build Process is implemented in the Python programming language owing to its extensive package support. The following sequential asset build node stages form the asset build process:

1. Merge of the Netlytic attributes stored in CSV file format into the INTERMEDIATE FILE FORMAT of the Graph Exchange XML Format (GEXF). The attributes are transformed into XML properties within the common graph format. Subsequent asset build node stages operate on the common intermediate file format.

2. Data cleansing of node attribute values and removal of XML namespaces.

3. A final transformation of the GEXF XML into a JSON equivalent in preparation for loading into the Unity runtime.

Correspondingly, a real-time three-dimensional visualization tool is implemented in the Unity game engine runtime. The tool imports the JSON graph format, deserializing the data into corresponding C# objects. The graph nodes and edges correspond to underlying runtime Unity game objects. The node objects have associations with the name-value Netlytic properties. The edge objects are represented as a geometric mesh connecting individual nodes. The resulting visualization is navigable within the Unity runtime using a bespoke navigation script.

5.8 Summary and Evaluation

Instantiation of the ASSET PIPELINE architectural pattern and component patterns gives rise to the asset pipeline workflow illustrated in figure 5.12.

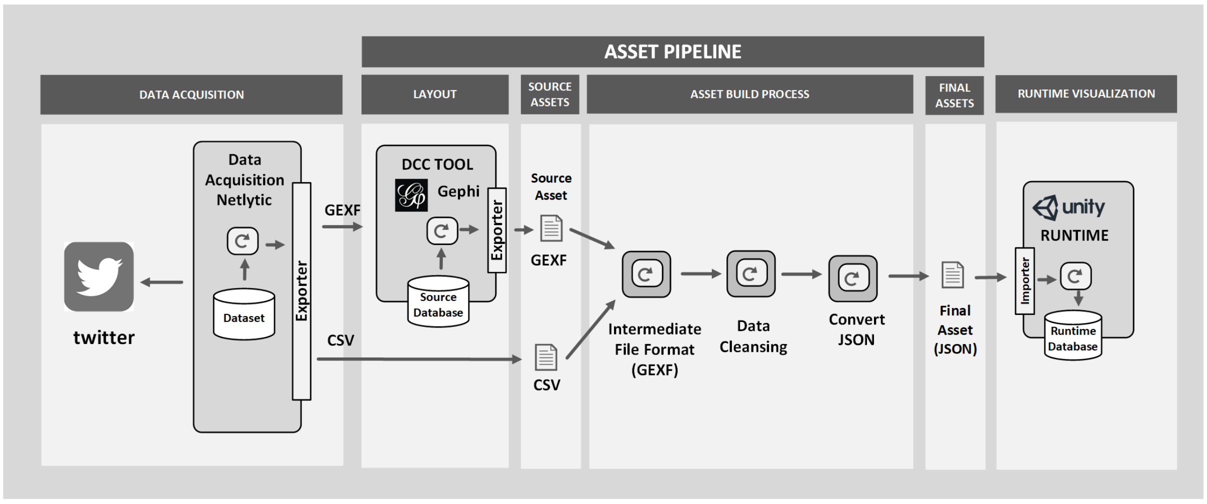
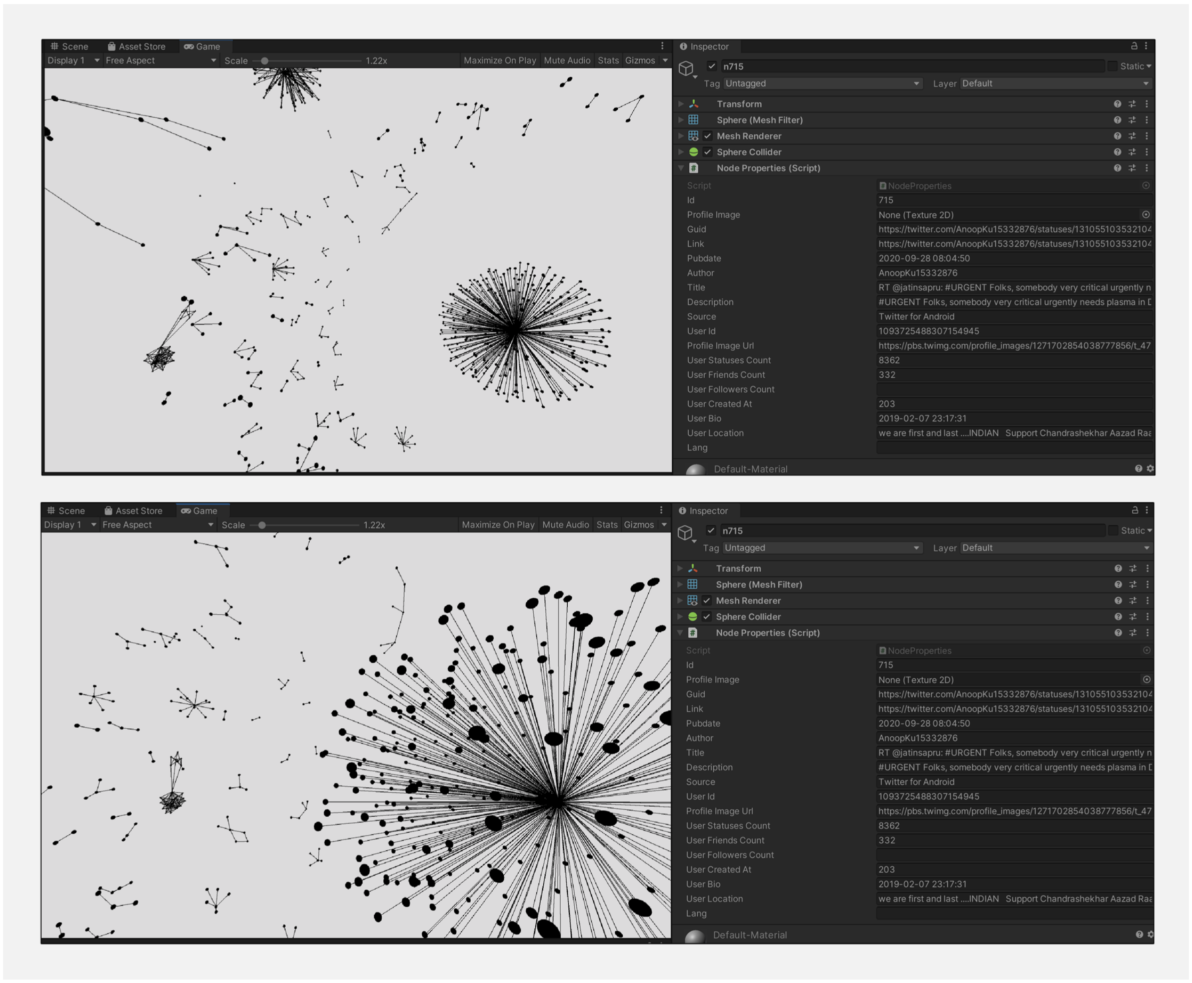


Figure 5.12: Graph Visualization (GraphViz) ASSET PIPELINE

Social Network data in the form of a Twitter dataset representing conversations specified with the #COVID hashtag is obtained using Netlytic cloud service. The data is transferred to Gephi, the open source visualization tool. Subsequently, the graph data is visualized with the Force Atlas 3D (Levallois, 2013) algorithm. A DCC EXPORT is performed to extract the graph layout including three-dimensional spatial data. A further DCC EXPORT from Netlytic extracts the properties for individual tweets. Through the ASSET PIPELINE the data is combined into an INTERMEDIATE FILE FORMAT. Further stages in the asset pipeline perform data cleansing and asset conversion into its final format of JSON in preparation for loading into the Unity runtime. A three-dimensional navigation tool implemented utilizing the Unity runtime and underlying rendering capability imports the final asset data. The resulting visualization facilitates enhanced navigation and visualization capabilities. Furthermore, Unity supports through a technological stack XR capabilities supporting Virtual Reality (VR), Mixed Reality (MR) and Augmented Reality (AR).

An example of the Unity runtime visualization is demonstrated in figure 5.13.



```
Inspector
n715                                   Static
Tag Untagged            Layer Default
Transform
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
Node Properties (Script)
Script                  NodeProperties
Id                      715
Profile Image           None (Texture 2D)
Guid                    https://twitter.com/AnoopKu15332876/statuses/13105510353210
Link                    https://twitter.com/AnoopKu15332876/statuses/13105510353210
Pubdate                 2020-09-28 08:04:50
Author                  AnoopKu15332876
Title                   RT @jatinsapru: #URGENT Folks, somebody very critical urgently n
Description             #URGENT Folks, somebody very critical urgently needs plasma in
Source                  Twitter for Android
User Id                 1093725488307154945
Profile Image Url       https://pbs.twimg.com/profile_images/1271702854038777856/t_47
User Statuses Count     8362
User Friends Count      332
User Followers Count
User Created At         203
User Bio                2019-02-07 23:17:31
User Location           we are first and last ....INDIAN  Support Chandrashekhar Aazad Ra
Lang
Default-Material
```

```
Inspector
n715                                   Static
Tag Untagged            Layer Default
Transform
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
Node Properties (Script)
Script                  NodeProperties
Id                      715
Profile Image           None (Texture 2D)
Guid                    https://twitter.com/AnoopKu15332876/statuses/13105510353210
Link                    https://twitter.com/AnoopKu15332876/statuses/13105510353210
Pubdate                 2020-09-28 08:04:50
Author                  AnoopKu15332876
Title                   RT @jatinsapru: #URGENT Folks, somebody very critical urgently n
Description             #URGENT Folks, somebody very critical urgently needs plasma in
Source                  Twitter for Android
User Id                 1093725488307154945
Profile Image Url       https://pbs.twimg.com/profile_images/1271702854038777856/t_47
User Statuses Count     8362
User Friends Count      332
User Followers Count
User Created At         203
User Bio                2019-02-07 23:17:31
User Location           we are first and last ....INDIAN  Support Chandrashekhar Aazad Ra
Lang
Default-Material
```

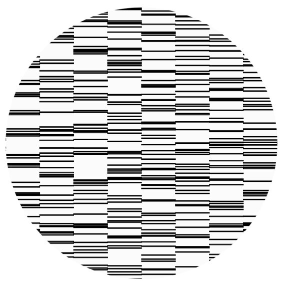
Figure 5.13: Graph Visualization (GraphViz) implemented in Unity.

The novel approach to graph visualization draws direct parallels with video game production. Within video game production a DCC tool is used by a designer to layout and design source assets such as a three-dimensional model. Comparably, the graph visualization incorporates the use of the Gephi DCC tool to three-dimensionally layout the social network data using a force directed layout algorithm. Subsequent stages of the asset build process are automated using Python to process the data into its finalized form as expected by the game engine.

A limitation of the approach is the instantiation of the asset pipeline produces a static visualization of the Twitter Social Network data; a snapshot in time of the source data. A subsequent refresh of the data requires the designer to push the data and instigate the asset pipeline workflow.

A dynamic real-time visualization is attainable by migrating the functionality and processing contained in the source stages of the asset pipeline into the runtime visualization. The approach would use the Twitter public RESTful API v1.1 to stream live twitter data representing source assets delivered to the asset build process. The physics engine middleware component of the game engine would replace the Gephi DCC tool to dynamically perform the force directed layout calculations in real-time.

6 Conclusions and Future Work


Generative Pattern 6 – JL 2020

This chapter concludes the thesis with critical reflection, a discussion of the research outcomes significance and a presentation of future research. The chapter initially reflects on the research questions whose outcome provides a conclusion to the research hypothesis. The contribution to knowledge and significance of the research is discussed. The limitations of the research are outlined, providing final input into the future directions for the domain of research.

6.1 Research Questions Revisited

In order to evaluate the research hypothesis, this section critically analyses responses to each research question.

RQ1. *How does an asset pipeline approach assist in the design and production of visualizations within the video game industry?*

To answer the first research question a literature review in the area of the video game production asset pipeline, the game engine and the XNA build framework was carried out (section 2.1). Owing to the problems outlined in section 1.2, primarily secrecy within the video game industry and a reluctance to impart knowledge that would harm intellectual property, a literature review of the asset pipeline was problematic. The majority of literature attempts to be too encompassing, offering an un-nuanced, general account of a dynamic industry (Kerr, 2006). Other literature delves into the technical details of the game engine, the final destination of the asset pipeline, with little detail on the underlying technical production workflow of the asset pipeline itself (Anderson *et al.*, 2008; Lengyel, 2010; Lewis and Jacobson, 2002). An abstract picture of the asset pipeline is provided within literature detailing game development tools (Ansari, 2011); these abstractions provide ideal sources for pattern mining. A more focused representation of the asset pipeline is illustrated in industry trade publications detailing video game production post-mortems (Game Developer Magazine, 2021). Any available web resources reflect external bias and so does the analysis of the production workflows used by the associated video game studios.

The amount of literature related to the theme of the asset pipeline is therefore relatively small in comparison to other domains of research. Any such literature provides an overview of the components and does not divulge low-level functional and implementation details. The lack of public knowledge related to the asset pipeline was unforeseen prior to commencing the research study. Nonetheless, the exhaustive literature review in section 2.1 provides a valuable contribution to knowledge. Future research could publish a literature survey of asset pipeline abstractions and implementations; thus providing a basis for further knowledge.

Video game content, in terms of source assets, is now the distinguishing factor used to differentiate video games within the marketplace. The amount of content required for a Triple-A (AAA) video game doubles every year (Llopis, 2004). Since a real-time preview of the source assets within the game engine runtime is cited as essential by artists (O'Donnell, 2014), and whose working practice follows the basic design cycle methodology (Koomen, 1991), an efficient workflow is required to assist the designer. The asset pipeline provides such a workflow and is the path that all game assets follow from their original conception within the Digital Content Creation (DCC) tool until they are loaded into the game engine runtime itself (Gregory, 2014).

As found in the survey of literature the asset pipeline assists a designer in the production of a visualization by fulfilling the requirements of:

i) Given a set of input source assets automatically determine the files that need to be processed and the order of processing.
ii) Handles the amount and complexity of transformations required to process each file.
iii) Produces assets for a number of target hardware runtime platforms.
iv) Is extensible and scalable in design.
v) Provides the granularity to handle dependency evaluation.

As an architectural workflow the asset pipeline as seen in video game production, the game engine and the XNA build framework can be abstracted into the common components of:

i) A DCC tool export plug-in initiated to extract the source assets from the DCC tool (section 2.1.2).
ii) An intermediate file format utilized to decouple the source asset file format from the asset build process to introduce reusability and flexibility in the pipeline (section 2.1.3). Correspondingly, decoupling components within an object-orientated system introduces flexibility and reusability (Gamma *et al.*, 1985).
iii) An asset build process tasked with transforming and processing the source assets into their final destination asset format. (sections 2.1.4 and 2.1.5).

The asset pipeline as identified in literature provides a snapshot in time; a limitation of research in an area of rapid technological development. In recognition of the difficulty Digital Content Creation (DCC) artists and designers face when attempting to transfer their source assets into the final runtime game engine manufacturers are now attempting to streamline the process. The upcoming Unreal Engine (2021) game engine encapsulates the external asset pipeline functionality as viewed in current literature. The technology termed Nanite allows a designer to import fine-grained geometric detailed models comprising hundreds of millions of polygons. The Nanite virtualized micro polygon technology performs a real-time asset build process to dynamically scale and stream geometry. The technology, not currently released for general use, pushes the offline static asset pipeline as detailed in this research, into the game engine itself. Future research in the area of the asset pipeline could incorporate this new model of workflow when available for analysis.

The focus of the research is the domain of video game technology, specifically the asset pipeline workflow. However, it became apparent the asset pipeline can be observed in many applications and domains outside of the video game industry, where source data requires transformation into a final state appropriate for visualization. For example, a number of television and film studios have expanded their programming teams to leverage a data-orientated approach to image production and address the inherent complex design challenges. Many have built proprietary asset pipelines to feed third-party visualization engines. In 2017 DreamWorks Animation introduced MoonRay, a high speed rendering architecture, integrating a wide variety of Digital Content Creation (DCC) tools for three-dimensional visualization – taking advantage of an asset pipeline incorporating open source components and differentiated APIs to produce an intuitive workflow (Lee *et al.*, 2017). These findings developed the realization that the asset pipeline is a generalizable solution, a pattern language, for application and instantiation into further domains.

RQ2. What characteristics of a pattern language make it a suitable approach for the communication of nomenclature, good design and development practices within the context of an asset pipeline workflow?

In answering the first research question a literature review was performed to identify the set of characteristics that an asset pipeline provides to assist a designer in the production of a visualization as seen in the video game industry. The characteristics are delivered under the implementation of the asset pipeline workflow and its identified underlying sub-components (sections 2.1.2 to 2.1.5).

To answer the second question a literature review of idioms, patterns and pattern languages was performed as covered in section 2.2. The asset pipeline is a general reusable solution to the commonly occurring problem of converting a large number of source assets into their final assets as required by the game engine runtime. The specifics of the asset pipeline are adapted to the context of its use. The asset pipeline is simultaneously an entity sui generis and motely (O'Donnell, 2014). Likewise, in the discourse of patterns, Alexander (1977, p.x) provides the definition, a "…pattern describes a problem with occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

In terms of format a pattern is "a three-part rule, which expresses a relation between a certain context, a problem, and a solution" (Alexander, 1979, p.247). The asset pipeline is a pattern, forming a recurring solution to a problem occurring within the context of video game production. That is how to deliver interactive visualizations where the source assets are created in a high-level Digital Content Creation (DCC) tool and the visualization operates within a low-level optimized runtime, for example a game engine.

The asset pipeline as utilized in the video game industry is subject to a number of problems, as identified in the research motivation, section 1.2. These are a lack of common nomenclature for games technology, such as the asset pipeline, and a difficulty communicating solutions to those within the industry and to those outside operating in domains of context that could benefit. The characteristics of a pattern language as identified in literature aspire to solve the problems identified in section 1.2.

Application of patterns is cited to result in a measure of increased "quality" (Alexander, 1975). The quality is evaluated in terms of increased flexibility, reusability and modularity in the design, implementation and maintenance of software systems (Gamma *et al.*, 1995). Indeed, the outcome of introducing attributes of quality is a mantra reenforced in classic literature (Gamma *et al.*, 1995; Buschmann *et al.*, 1996; Coplien, 1992). Likewise, the asset pipeline as found in video game production was evolved to increase flexibility, reusability and modularity within the production process, a process subject to rapid change during early video game development cycles (O'Donnell, 2014). The source asset format of the Digital Content Creation (DCC) tool is decoupled from the final format of the runtime using an intermediate file format. This introduces flexibility and reusability within the asset build process. It could be argued the asset pipeline exhibits many of the positive characteristics extoled by the pattern movement.

The use of design patterns provides a common vocabulary and understanding of design principles (Gamma *et al.*, 1995), something the video game industry has struggled with. A pattern identifies, names and specifies abstractions above low-level implementation details (Gamma *et al.*, 1995; Rising, 1998; Buschmann *et al.*, 1996; Vlissides, 1998). Although the existence of an asset pipeline is inherent in an established video game development studio, for those new to the industry or for those independent developers not owned by a video game publisher, the use of a pattern approach will increase their shared understanding. O'Donnell (2014) voiced concern about outsourcing art asset creation, identifying the problems arising from a lack of shared vocabulary and understanding of the content driven approach utilized by the asset pipeline. The predefined format for specifying the problem, solution and context under which a pattern operates (Rising, 1998) will assist in this process.

Although design patterns capture expertise and make knowledge accessible to non-experts (Buschmann, 1996; Gamma *et al.*, (1995) such knowledge exchange can only occur if the pattern language is accessible. Agerbo and Cornils (1998) asserts that such a knowledge exchange can only occur if the patterns are accessible in a central repository and the patterns have been vetted for acceptance. The Pattern Languages of Programs (PLoP) process aligns to this requirement. The asset pipeline pattern language was subject to the rigorous steps of shepherding and a writers' workshop. However, although the patterns are published and within the public domain their usefulness will only become evident when they enter into the design vernacular of visualization design and production. Khomh and Gueheneuc (2018) emphasize the positive impact of a pattern becomes apparent if the conduit can be bridged between research and industrial activities; although resistance from the video game industry may be evident in the domain of research.

Design patterns serve as reusable building block for system development (Buschmann *et al.*, 1996; Gamma *et al.* 1995). The choice of pattern to apply under a certain context may be a difficult process for a developer to apply. Gueheneuc and Mustapha (2007) devised a recommender system to assist in the process although their research focused on the twenty-three patterns of the design catalogue (Gamma *et al.*, 1995). The asset pipeline is formalized into a high-level architectural pattern and three component child patterns. It is envisaged such a problem will not arise given the smaller set of patterns.

RQ3. To what extent can an asset pipeline be formalized into a pattern language?

Section 3 details the successful creation of the asset pipeline pattern language. The framework of the Pattern Languages of Program (PLoP) conference aligned to the spiral model methodology was followed. Cycle one of the spiral model incorporated:

i) **Derivation of the pattern format:** The pattern form provides the literary structure of the pattern and offers the pattern reader with a consistency when viewing and comparing patterns within a pattern language.

   The original Alexandrian form (Alexander, 1977) was adapted and extended by fellow pattern authors such as Gamma *et al.* (1995)*,* Portland (Cunningham, 2020), Coplien (1994) and Pattern Language of Program Design (section 2.2.3). According to Vlissides (1998, p.147), "no one form suits everybody." The asset pipeline is an architectural workflow pattern, and as such does not require implementation details at the code level. The formulated asset pipeline pattern language uses an extension of the pattern schema identified by Wellhausen and Fießer (2011).

ii) **Pattern mining for discovery of the individual asset pipeline patterns**: There are two methodologies for mining patterns: automatic pattern mining derived from concrete solution instances in the field and manual pattern mining based on observable abstractions of solutions. Beyond those abstractions viewed within the game engine and build frameworks (XNA) publicly accessible asset pipeline source code instances do not exist. The seeds for manual pattern mining the asset pipeline pattern language were derived from the literature review in section 2.1. In particular the asset pipeline workflow as observed in video game production, the game engine and the XNA build framework. The solution schema was abstracted from the shared characteristics of such workflows.

iii) **Pattern writing**: Following the pattern writing method of Wellhausen and Fießer (2011) and under the supervision of a shepherd the two asset pipeline patterns were elaborated: ASSET PIPELINE and INTERMEDIATE ASSET FORMAT.

   The ASSET PIPELINE pattern solves the problem of digital content creators designing in DCC tools using high level concepts that require visualizing in a runtime requiring optimized versions. Whereas, the INTERMEDIATE ASSET FORMAT decouples the source format from the final format to introduce flexibility in the asset pipeline.

Cycle two of the spiral model composed:

i) **Writers' workshop:** The writers' workshop highlighted the ASSET PIPELINE pattern was extremely extensive in terms of technical detail and would benefit from further decomposition.

ii) **Pattern mining:** The ASSET PIPELINE pattern was promoted to an overall architectural workflow pattern involving the application of three sequential component patterns referred to as DCC EXPORT, INTERMDIATE FILE FORMAT and ASSET BUILD PROCESS.

iii) **Pattern writing:** The ASSET PIPELINE provides designers with a workflow to assist them in viewing their digital content in an interactive real-time visualization. DCC EXPORT provides a mechanism to transform and extract DCC source asset data into a format appropriate for further workflow processing within the ASSET BUILD PROCESS. The INTERMEDIATE ASSET FORMAT decouples the source and final asset dependency from the asset pipeline. Finally, the ASSET BUILD PROCESS provides asset data conditioning to fulfil the requirements of the real-time visualization runtime.

A conceptual representation of the ASSET PIPELINE architectural pattern and component patterns is illustrated in figure 6.1.

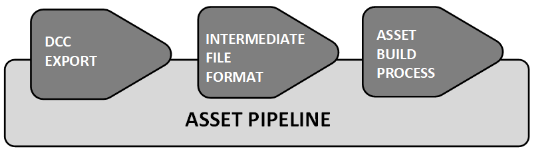


Figure 6.1: ASSET PIPELINE architectural and component patterns

Formalization of the asset pipeline into a pattern language encapsulates the core functional details of the asset pipeline as observed in literature. However, a limitation of deriving the requirements of the pattern language from literature, rather than from either industrial practitioners or automatic mining of concrete instantiations, is a lack of access to the nuanced characteristics of the pipeline. Future research could investigate such approaches.

Nevertheless, formalization of the asset pipeline into a pattern language provides a significant contribution to knowledge for two reasons. Firstly, the world of patterns and video games has not been researched beyond the low-level game design idiom of Nystrom (2014) and these did not extend to the asset pipeline workflow. Secondly, the asset pipeline has not been communicated outside of the video game industry for application into further domains that could benefit from such application. As found in baseline literature the positive properties of patterns provide an ideal conduit for such communication.

RQ4. *Can an asset pipeline pattern approach be applied to assist in the design and production of a visualization?*

Under the test scenario piecemeal application of the asset pipeline pattern language formed the derivation of two visualizations. The asset pipeline pattern language was instantiated and applied in two domains: an architectural visualization (ArchViz) and a graph visualization (GraphViz) - section 4 and section 5 correspondingly. In this respect functionally the pattern language passes interrogation. However, in the test scenarios the pattern was applied by the author of the pattern language and this is not truly representative of the use of the pattern language in the field. The author is familiar with the problem, the context under which the problem is observed and the conflicting forces at play.

Ultimately, determination of the success of a pattern really depends upon the pattern becoming part of the design vernacular in the context of the video game industry and other domains that could benefit. Only when the pattern language is communicated in verbal or written form can we determine the widespread acceptance of the pattern. That is, not only within the pattern community but more importantly in domains that incorporate such visualizations and workflows and would benefit from a pattern approach to their application. Communication and publication of the pattern language at the European Conference on Pattern Languages of Programs Proceedings (EuroPLoP’19) has part filled this requirement.

It took years before the patterns of Gamma *et al.* (1995) were recognized, taken out of their context of the insular pattern community and into the software engineering industry at large. However, the design patterns of Gamma *et al.* (1995) are now taught in Universities around the world and are directly referenced in industry and the latest software frameworks.

To conclude, RQ1 investigates how an asset pipeline assists in the design and production of visualizations in the video game industry as viewed in literature. The asset pipeline provides a flexible, efficient method for transforming a set of source assets as defined by DCC tools into a format appropriate for a target runtime visualization. The asset pipeline architecture can be abstracted into the components of a DCC tool export plug-in, an intermediate asset container format and an asset build process. Given the problems associated within the domain of the video game industry, specifically a lack of game development language and the difficulty communicating solutions, RQ2 identified the characteristics of a pattern approach that can potentially mitigate such problems. Namely patterns provide a common vocabulary; they provide a predefined format for specifying the problem, solution and context under which they operate; they capture expertise and importantly they enable domain independent solutions.

RQ3 formalized the asset pipeline into a pattern language using manual pattern mining and pattern writing. Adopting the rigorous framework of the Pattern Languages of Program (PLoP) workflow aligned to the spiral model methodology the following patterns were formulated: ASSET PIPELINE, DCC EXPORT, INTERMEDIATE FILE FORMAT and ASSET BUILD PROCESS. RQ4 evaluated the instantiation of the pattern catalogue in two domains of visualization: an architectural visualization and a graph visualization. Using a sequential approach initially the ASSET PIPELINE pattern was applied, followed by the application of the underlying component patterns.

The research hypothesis asserts the application of a pattern language approach to the asset pipeline can assist both artists and designers in the communication. In answering the research hypothesis, one must initially consider the integrity of the pattern language. Pattern writing was achieved using the rigorous workflow of shepherding and a writers’ workshop. However, the patterns were mined using the findings from the literature review. Owing to the issues outlined previously, concrete low-level examples of the asset pipeline were not available for analysis. If such examples existed application of an automatic pattern mining technique would potentially result in fine grained patterns at the design level of classification.

Validation of the derived pattern language was achieved exploring two scenarios operating in domains outside of the video game industry with comparable workflows. The pattern author sequentially applied the pattern language to achieve the resulting visualizations. The success of the application of the pattern language proves the research hypothesis at a functional level. However, one must be mindful of the familiarity of the pattern author with the formalized pattern language. A wider study incorporating practitioners in the field would provide valuable input into the quality of the pattern language. Such an evaluation was outside the scope of this research.

6.2 Contributions to Knowledge

The following are the key contributions to this research study:

(i) **Asset Pipeline Workflow:** The asset pipeline as seen in video game production has not been critically investigated. Literature with regard to the asset pipeline and games technology production workflows in general are extremely limited owing to the problems outlined in the research motivation (section 1.2). Nevertheless, the unique literature review in section 2.2 provides a contribution in this field.

(ii) **Asset Pipeline pattern language:** A novel asset pipeline pattern language has been created within the framework of the Pattern Languages of Program (PLoP) workflow. This research heralds an interdisciplinary approach to games technology and pattern languages beyond the low-level idioms identified by Nystrom (2014). The asset pipeline pattern language has been published in the ACM after their rigorous peer review process for communication to a wider audience.

(iii) **Architectural Visualization:** The ASSET PIPELINE pattern language was instantiated to create an architectural visualization (ArchViz). The novel approach generates a Unity project rich in BIM semantic data from which a real-time navigation application is generated to allow exploratory investigation. Further development can leverage the semantic data for further purposes.

(iv) **Network Graph Visualization:** A novel approach to exploratory graph visualization (GraphViz). The visualization uses the DCC tool Gephi to layout the initial source asset data. Application of the ASSET PIPELINE pattern language is used to instantiate a real-time interactive exploratory tool facilitating rich navigation. Utilizing the Unity game engine initially for its rendering capability the prototype can be extended to incorporate XR input devices in the guise of virtual reality, augmented reality and mixed reality.

6.3 Research Limitations

As highlighted in the literature review (section 2.2.5) access to the source code of proprietary asset pipelines is currently impossible owing to the reluctance of the video game industry to divulge intellectual property. The remaining avenue of investigation for the researcher is to review academic texts devoted to games technology in general, games toolset creation and trade publications detailing concrete asset pipelines as detailed in video game post-mortems. Further asset pipelines were evaluated in the games engine and build pipeline (XNA), however, these provided only a "black box" view or skeleton framework of the asset pipeline. This level of generalization is appropriate for manual pattern mining techniques since high-level abstractions of solution schemas are required. However, the lack of concrete implementations prevented the use of automatic pattern mining approaches and undoubtedly compromised the breadth of the asset pipeline language.

Given further asset pipeline source code solutions an automatic process of pattern mining could be developed. It is envisaged this process would uncover patterns at the lower-level design and idiom level of pattern classification. That is patterns approaching the level as identified by Nystrom (2014), however, in relation to the asset pipeline workflow. Future research could seek to gain access to underlying concrete asset pipeline instantiations as used in the video game industry.

Refinement of the derived pattern language was achieved through the iterative shepherding process and writers' workshop mechanism. Although feedback from experienced pattern writers undoubtedly raised the quality of the pattern language, communication and evaluation with video game industry practitioners would have provided more domain specific knowledge. Such collaboration is cited by Khomh and Gueheneuc (2018) as having positive impact on pattern quality. O'Donnell (2014) recognizes limited access to field sites and small sample sizes restricts research in the domain of the video game industry. Future research in this area could seek to obtain such collaboration.

The formalized asset pipeline pattern language was tested through application of the patterns to assist in the creation of two visualizations. The process of which was performed by the researcher who derived the original pattern language and is familiar with their application. Patterns introduce a common vocabulary, understanding and capture expertise and make knowledge accessible to non-experts. In this instance the researcher is familiar with the intricacies of the domain area and application of the pattern language was relatively straightforward. Reduction of such bias in pattern choice and application could be reduced by extending the study to industrial practitioners involved in the creation of interactive visualizations who are not acquainted with the technology of the asset pipeline and the related pattern language. With concrete use and instantiation into domains of visualization further patterns will emerge and become part of the design vernacular. The identified asset pipeline patterns are to be considered an evolving catalogue.

The existing asset pipeline patterns consist of patterns considered architectural in scope. Further expansion of the catalogue should explore finer grained patterns operating at the design level in scope. For example, the ASSET BUILD PROCESS pattern comprises the application of individual asset build nodes responsible for transforming, converting and compiling assets. These may form a candidate for such a design pattern.

Instantiation of the ASSET PIPELINE pattern and its sub-component patterns resulted in the development of two asset pipelines for the construction of the visualizations in architecture (ArchViz) and graph (GraphViz). These software artifacts were bespoke in nature, however, the common entities should be abstracted into a software framework in line with the asset pipeline catalogue. The asset pipeline constructed for the generation of the graph visualization part fulfilled this requirement having a Python based framework.

The resulting architectural and graph visualization were limited in their user experience (UX). Given time the semantic information persisted in the architectural BIM and graph model could be further exposed to the user. Although this is undoubtedly out of scope of this study; research into the asset pipeline production workflow.

6.4 Future Directions

There are several avenues for further research in both the asset pipeline, patterns in games technology and expansion of the use case scenario functionality.

The following outline the areas of potential research in the domain of the asset pipeline:

(i) **Push and Pull Model:** The asset pipeline as described in this research is a push model in which the end-user invokes the asset build process, and traditionally this has been the case. An alternative is whereby the runtime game engine monitors when a source asset has changed and pulls the latest changes automatically. Indeed, both Unity and Unreal Engine employ a simplified version in their asset pipeline (section 2.2.6). The game engine polls for changes in a folder structure and instigates an automatic import. This workflow is a candidate for further pattern expansion.

(ii) **Fast Path:** A separate path, named the fast-path, is often created in parallel to the main asset build process. To maintain the highest possible performance only the data that is modified traverses the fast path; the remaining data is loaded in the optimized final runtime format. During the production phases of video game development, it is necessary for the game engine to be capable of loading both the optimized data and also the intermediate file format. The two alternative paths are illustrated in figure 6.4. However, one possible risk is the preview version of the source asset is not representative of the final quality. There is a delicate balance to be achieved; it is desirable to allow the designers the freedom to quickly iterate their designs, however, it is essential they do not manifest in unrepresentative previews. The alternative workflow of the fast path is a candidate for pattern mining.

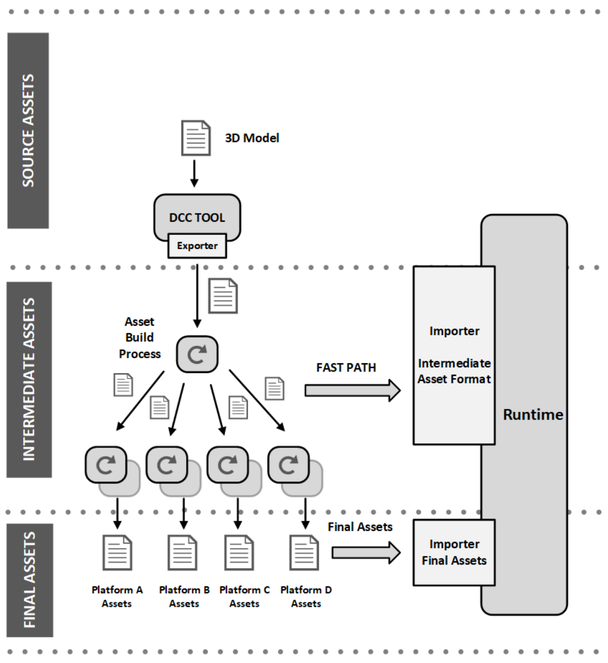


Figure 6.4: Alternative fast-path

(iii) **Asset Build Node:** Where the asset pipeline is an architectural pattern, the asset build node is a candidate for a lower level of granularity design pattern. The asset build node performs a transformation on one or more input assets into one or more output assets. Initial thoughts are the transformation may be specialized into the following types: exporter, importer and compiler. The distinctions being made for the following reasons. The exporter type is responsible for transforming the source assets from a Digital Content Creation (DCC) tool format into the intermediate file format expected by the asset build process. Within the asset build process the compiler performs the necessary transformations on the intermediate file format. Finally, the importer creates the final asset format as consumed by the game engine.

(iv) **General purpose framework:** The pattern language may be implemented as a general-purpose software framework. Since frameworks are written in a specific computer language and often depend upon a certain operating system they do not solve all problems. A software engineer has to understand the underlying rationale for the design of the framework before they can utilize its functionality. For frameworks that are based on an underlying pattern language, the language can assist the designer in understanding this rationale. A general-purpose asset pipeline pattern framework is the next evolutionary step for research in this area.

In the broader sense there is opportunity to discover patterns in further areas of games technology. One such area is virtual reality navigation. Whilst there is a substantial body of Human-Computer Interaction (HCI) research exploring navigation in two-dimensional and three-dimensional interfaces, many of the approaches are unsuitable for use in XR, specifically augmented, virtual and mixed reality technologies. Poupyrev *et al.* (2017) compartmentalizes the navigation task into the components of wayfinding and travel. Travel within a virtual reality environment involves the movement of the viewpoint from one place to another. The techniques for navigating within three-dimensional virtual spaces remains an active research problem. Although many such navigation techniques, such as teleportation, have been employed in protype and real-world applications human factors such as motion sickness have resulted in difficulty for a designer to choose one such technique over another. The ethos of patterns provides an ideal fit for further exploration into virtual reality navigation techniques.

The use case scenarios of Architectural Visualization (ArchViz) and graph visualization (GraphViz) can be further researched in the areas of:

Architectural Visualization (ArchViz):

i) **Virtual Reality path generation:** The BIM semantic data available for the building encompasses the notion of floors, rooms, entrances and exits (doorways). This information may be used to automatically deduce all possible navigation routes through a building, and these form the pathways for virtual reality navigation.

ii) **Gameplay mechanics for simulation:** The BIM geometric model and object semantics can be used to automatically generate assets to solve simulation problems. For example, the architectural model of the Architecture and the Built Environment block as supplied by the University of the West of England contains the semantic information such as fire alarm locations and emergency exit points. A simulation of the effectiveness of exit routes under an emergency structural fire situation may be performed with actual real-world players.

iii) **Automatic generation of game assets:** Environmental game assets such as buildings are time intensive for a content creator to design and create. Real-world fully specified BIM models can be used as a basis for three-dimensional content creation. The semantic information associated with building components can be used for procedural exchange of materials and for placement of additional gameplay components.

Graph Visualization (GraphViz):

i) **Layout functionality:** A game engine such as Unity contains an underlying physics engine middleware component responsible for the approximate simulation of rigid body dynamics. The functionality provided by the Digital Content Creation (DCC) graph layout tool, such as Gephi, can be migrated into the real-time exploratory graph visualization tool operating in the game engine runtime. That is, the layout functionality can exploit the physics engine component to perform operations such as force directed layout techniques.

ii) **Data acquisition:** The graph visualization scenario uses a static visualization to perform Social Network Analysis (SNA). The cloud-based service, Netlytic, is used to acquire the data. This stage of the pipeline can be migrated into the graph visualization tool. The visualization would utilize the Twitter public RESTful API v1.1 to stream live twitter data; becoming dynamic in appearance.

iii) **Advanced rendering capability:** The rendering capability of the game engine, specifically the use of physically based shaders, can enhance the fidelity and appearance of the graph visualization. Research into creating graph visualization with differing appearances is a candidate for exploration.

6.5 Conclusion

Video games consist of virtual worlds modelled as an approximation of either a real or imaginary environment. The expectation of the end-user and video game publisher in terms of immersive and detailed environments has risen drastically and the amount of content required for Triple-A (AAA) video games doubles every few years (Llopis, 2004). The art discipline responsible for creating such environments now comprises the majority of those working in video game studios. Artists and designers utilize a suite of development tools known as Digital Content Creation (DCC) tools to generate digital content in the form of assets. The DCC tools were not originally designed for the use of video game asset creation (Blow, 2004) and offer very little interoperability support in this respect.

The creative process of video game production is iterative in approach and follows the design cycle methodology. Ultimately an artist requires a real-time preview of the content within the rendering engine as provided by the game engine to ensure accuracy under runtime conditions. However, a barrier exists preventing the artist from achieving this goal in a fast and efficient manner. That is, the requirements of the source asset are very different from those of the final game asset. The source asset is often held in a binary proprietary format of the DCC tool and is information rich allowing for future edits and modifications. Whereas, the final game asset is lightweight and optimized for fast and efficient loading in the runtime environment.

To overcome this challenge the video game industry has invested heavily in the production of a workflow to streamline game asset production. This is known as the asset pipeline. The asset pipeline forms a general reusable solution to the problem identified in video game production of converting a large number of source assets into their final runtime versions. There are elements of the asset pipeline that are used for one project and not another. The pipeline is simultaneously an entity sui generis and motley (O'Donnell, 2014).

Likewise, within the discourse of patterns a pattern is defined as a general reusable solution to a problem operating within a certain context. The study claims the asset pipeline is a template solution applied within the domain of video game production that can equally be extended into other domains with comparable workflows to assist in their production.

A single piece of literature devoted to the pattern-oriented approach to video games technology was produced by Nystrom (2014), however, the patterns identified by Nystrom are low level idioms and do not extend to that of the overall video game production workflow of the asset pipeline. No academic study has specifically looked at the asset pipeline and certainly no study has formalized the asset pipeline into a pattern catalogue for wider use. This is owing to a number of reasons. Firstly, the video game industry is relatively young, and a set of agreed-upon terms has not arisen. Secondly, general secrecy pervades the industry preventing the natural dissemination of information owing to compulsory non-disclosure agreements (NDAs). For this reason, proprietary technology, including such workflows as the asset pipeline are not communicated both within and outside of the industry. Thirdly, the asset pipeline is situated at the interdisciplinary fault line between the software engineers working on the underlying technology and the artists designing the content. It is at the interdisciplinary fault line where a language barrier exists, and a common set of nomenclature does not exist. Finally, the resources available for new developers wanting to implement such a pipeline are lacking, especially when it comes to those documenting data-oriented designs.

The aim of the thesis was to evaluate whether the application of a pattern-based approach to the asset pipeline workflow can assist both artists and designers with the creation of interactive real-time visualizations. Initially a literature review of the asset pipeline and how the workflow assists in the design and production of visualizations within the videogame industry was undertaken. Given the secrecy and reluctance to disseminate technical information the literature review was problematic. An exhaustive and prolonged search for information was undertaken incorporating academic texts detailing video games and associated technologies, industry focused post-mortems of video game production and publications. From these observations the asset pipeline was abstracted into a common set of functionalities.

Subsequently, the properties of a pattern language that make it a suitable approach for the communication of such nomenclature was achieved through a literature review of patterns. Patterns are promoted to make the design of programs more flexible, modular and reusable. They capture expertise and make knowledge accessible to non-experts, provide a common vocabulary for communication and enable domain independent solutions. Patterns are noted for introducing quality attributes to a design and solution. However, there is now an increasing amount of research that indicates otherwise. Such research is primarily based on application of the patterns of Gamma *et al.* (1995) and cannot possibly include all domains of application.

Given these inputs the asset pipeline was formalized into a catalogue of patterns. There are two distinct pattern mining methodologies: automatic programmatic mining of patterns based on code instances and manual pattern mining techniques. Asset pipeline code instances are not available for academic inspection beyond the asset pipelines as viewed in the games engine and build framework of XNA. The asset pipeline is an integral component of video game production and as such is proprietary intellectual property. An automatic pattern mining technique was not permissible, a limitation of this research study. Future research in this area should seek to collaborate with industry to obtain such code instances.

Manual pattern mining using the spiral model methodology aligned to the framework of the Pattern Languages of Program (PLoP) was applied. Under the iterative process of shepherding a catalogue of patterns was mined and documented. At the Pattern Languages of Program (PLoP) writers' workshop the asset pipeline pattern catalogue was extensively critiqued. Subsequently after another cycle of pattern mining, decomposition and refinement the pattern catalogue was published. The catalogue consists of the ASSET PIPELINE architectural pattern and the component patterns of DCC EXPORT, INTERMEDIATE ASSET FORMAT and ASSET BUILD PROCESS. The catalogue of patterns are presented using a derived common format facilitating a pattern reader to compare patterns and understand the context, problem, resolution of forces, solution and consequences. Formalization of the asset pipeline into a catalogue of patterns assists in overcoming the problems identified. That is, a common set of nomenclature and vocabulary, a method of selecting and applying such a workflow depending upon the context of use and communication of the pattern catalogue to the wider community of practitioners. Khomh and Gueheneuc (2018) emphasize the need to bridge research activities within industrial practice.

The sequential application of the asset pipeline pattern catalogue was successfully instantiated into two concrete visualizations. The first, the use case of an architectural exploratory visualization based on Building Information Modeling (BIM) and the second the use case of an exploratory graph visualization tool based on a sub-category of data visualization known as Social Network Analysis (SNA). At a functional level, application of the pattern language successfully assisted in the design and production of two visualizations. However, the asset pipeline pattern language is perfectly understandable to the author of a pattern who is familiar with the context, problem and solution. Further evaluation of the asset pipeline pattern catalogue with industrial practitioners would enhance the study, however, the restriction of industrial collaboration and access to field sites prohibited such involvement. The reluctance for academic and industrial collaboration within the video game industry has been cited by previous researchers (O'Donnell, 2014). Furthermore, the video game industry is content driven and undoubtedly those established within the industry will have a mature integrated asset pipeline workflow. Certainly for those new to the industry, in academic learning, or independent developers the pattern language will be of assistance. Indeed, the patterns of Nystrom (2014) are extensively used within academia. Where the real value and contribution to knowledge lies is the application of the asset pipeline outside of the video game industry through the conduit of the communication of patterns.

According to Perron and Wolf (2009) video game studies has now moved into a second phase in which having already set its foundations as an academic field of study must now codify its tools and terminology and organize its findings in a coherent discipline. This study has satisfied these requirements in one aspect of video games technology. That is the production workflow, known as the asset pipeline. The asset pipeline has been formalized into a published pattern catalogue for general consumption and application in further areas of study. The asset pipeline pattern catalogue has provided a mechanism of raising the level of discourse about the video game production workflow from a concrete to a new abstract level of description. Although this is one aspect of video games technology it is nonetheless a starting point for future researchers to study and extend; the integration of patterns and emerging areas of games technology.

7 Bibliography

Aarseth, E. (2001) *The international journal of computer game research - computer game studies - year one.* Available from: http://gamestudies.org/0101/editorial.html [15 July 2020].

Activision (2009) *Call of Duty: Modern Warfare 2 sets all-time entertainment industry record grossing an estimated $550 million worldwide in first five days.* Available from: https://investor.activision.com/news-releases/news-release-details/call-duty-modern-warfarer-2-sets-all-time-entertainment [Accessed 1st April 2020].

Agerbo, E. and Cornils, A. (1998) How to preserve the benefits of design patterns. *SIGPLAN Notices*. 33 (10), pp.134-143.

Aguilar, E., Alexánder, I., Saúcog, I.A. (2020) An Evolutionary Mesh Compression Algorithm for Video Games. *Virtual Games Group, University of Informatics Sciences (UCi), Havana City, Cuba.*

Akado, Y., Kogure, S., Sasabe, A., Hong, J., Saruwatari, K., Iba, T. (2015) Five Patterns for Designing Pattern Mining Workshops. *Proceedings of the 20th European Conference on Pattern Languages of Programs*. Kaufbeuren, Germany, Association for Computing Machinery.

Alexander, C. (1975) *The Oregon Experiment*. United States: Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M. (1977) *A pattern language: Towns, Buildings, Construction.* New York: Oxford University Press.

Alexander, C. (1979) *The Timeless Way of Building*. New York: Oxford University Press.

Anderson, E., Steffen, E., Comninos, P., McLoughlin, L. (2008) The case for research in game engine architecture. *Proceedings of the 2008 Conference on future play, 2008*, pp. 228-231.

Andrade, A. (2015) Game engines: A survey. *EAI Endorsed Transactions on Game-Based Learning, 2*(6), 150615-6.

Ansari, M. Y. (2011) *Game development tools*. Natick, Mass: A K Peters.

Arnaud, R., Barnes, M. (2006) *COLLADA: Sailing the Gulf of 3D Digital Content Creation*. Massachusetts: CRC Press.

AutoDesk (2016a) *3ds Max - 3D Design Software for Modelling, Animation and Rendering.* Available from: http://www.autodesk.co.uk/products/3ds-max/overview [Accessed 12 June 2016].

AutoDesk (2016b) *Maya - Comprehensive 3D Animation Software.* Available from: http://www.autodesk.co.uk/products/maya/overview [Accessed 14 June 2016].

AutoDesk (2016c) *Revit API.* Available from: http://forums.autodesk.com/t5/revit-api/bd-p/160 [Accessed 12 June 2016].

Autodesk Maya. (2017) *Maya Architecture Overview.* Available from: https://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=__files_GUID_F584322E_4A12_4995_9F4E_A38D4331808F_htm [Accessed 10 July 2020].

Baek, N., Yoo, K. (2015) Emulating OpenGL ES 2.0 over the desktop OpenGL. *Cluster Computing*. 18 (1), pp. 165-175.

Bafandeh Mayvan, B., Rasoolzadegan, A. and Ghavidel Yazdi, Z. (2017) *The state of the art on design patterns: A systematic mapping of the literature.* The Journal of Systems and Software. 125 pp.93-118.

Ballhaus, W. (2018) *PwC global entertainment & media outlook 2018 – 2022.* Available from: https://www.pwc.com/gx/en/industries/tmt/media/outlook.html [Accessed 10 July 2020]

Bastian, M., Heymann, S., Jacomy, M. (2009) Gephi: An Open Source Software for Exploring and Manipulating Networks.

Beers, A., Agrawala, M., Chaddha N. (1996) Rendering from compressed textures. *Proceedings of the 23rd annual conference on computer graphics and interactive techniques*, 08/1996.

Bille, R., Smith, S., Maund, K., Brewer, G. (2014) Extending Building Information Models into Game Engines. *Proceedings of the 2014 Conference on interactive entertainment*, 1-8.

Blender (2020) *About Blender.* Available from: https://www.blender.org [Accessed 26 October 2020]

Blow, J. (2004) Game development: Harder than you think: Ten or twenty years ago it was all fun and games. Now it's blood, sweat, and code. *Queue*. pp. 28-37. doi:10.1145/971564.97159

Blythe, S. (2000) Readings in Information Visualization: Using Vision to Think / Information Design / Information Architects: TCQ. *Technical Communication Quarterly*. 9 (3), pp. 347.

Boehm, B.W. (1988) A spiral model of software development and enhancement. *Computer (Long Beach, Calif.)*. 21 (5), pp. 61-72.

Bogdanowicz, M., Prato, G. d., Nepelski, D., Simon, J., Lusoli, W. (2010) *Born digital / grown digital: Assessing the future competitiveness of the EU video games software industry* Joint Research Centre (Seville site).

Bowker, G. C., Star, S. L. (1999) *Sorting things out: Classification and its consequences*. London; Cambridge, Mass: MIT.

Boyle, E. A., Hainey, T., Connolly, T. M., Gray, G., Earp, J., Ott, M., et al. (2016) An update to the systematic literature review of empirical evidence of the impacts and outcomes of computer games and serious games. *Computers and Education, 94*, 178-192.

Bruns, A. (2012) How long is a tweet? Mapping dynamic conversation networks on twitter using Gawk and Gephi. *Information, Communication & Society*. 15 (9), pp.1323-1351.

Burnham, V. (2003) *Supercade: A Visual History of the Videogame Age 1971-1984*. Cambridge, Mass: MIT Press.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Chichester; New York: Wiley.

Carter, B. (2004) *The Game Asset Pipeline*. Massachusetts: Charles River Media.

Carver, J.C. and Prikladnicki, R. (2018) *Industry–Academia Collaboration in Software Engineering.* IEEE Software. 35 (5), pp.120-124.

Caves, R. E. (2000) *Creative industries: Contracts between art and commerce*. London: Harvard University Press.

Chandler, H. (2019) *The Game Production Toolbox*. London: CRC Press.

Charrieras, D., Ivanova, N. (2016) Emergence in video game production: Video game engines as technical individuals. *Social Science Information, 55*(3), pp. 337-356.

Coplien, J. O. (1992) *Advanced C++ programming styles and idioms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Coplien, J.O., Schmidt, D.C. (1995) *Pattern Languages of Program Design*. United States.

Cunningham, W. (2020) About the Portland Form. Available from: https://c2.com/ppr/about/portland.html [Accessed 20 September 2020].

DeLoura, M. (2009) *The engine survey: General results.* Available from: https://www.gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php [Accessed 14 July 2020]

Deterding, S., Dixon, D., Khaled, R., Nacke, L. (2011) From Game Design Elements to Gamefulness: Defining "gamification". *15th International Academic MindTrek Conference: Envisioning Future Media Environments.* pp. 9-15. doi:10.1145/2181037.2181040

Digital Games Research Association (2019) *About DiGRA.* Available from: http://www.digra.org/ [Accessed 26 October 2020]

Eastman, C.M. (2011) *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers Designers, Engineers, and Contractors*. 2nd ed. Hoboken, N.J: Wiley.

EuroPLoP (2019) *The Conference.* Available from: https://europlop.net/content/conference [Accessed 26 October 2020]

Fellmann, M., Koschmider, A., Laue, R., Schoknecht, A., & Vetter, A. (2019) Business process model patterns: State-of-the-art, research classification and taxonomy. *Business Process Management Journal, 25*(5), pp. 972-994.

Fernandez, D. (2015) National BIM Standard-United States. *National Institute of BUILDING SCIENCES*.

Fowler, M. (2003) *Patterns of Enterprise Application Architecture*. 1st ed. London; Boston, Mass: Addison-Wesley.

Gabriel, R. (2010) *Writers' Workshops As Scientific Methodology.* Dream Songs, Inc.

Galison, P.L. (1999) Reflections on Image and Logic: A Material Culture of Microphysics. *Perspectives on Science*. 7 (2), pp. 255-284.

Gamasutra the art & business of making games (2020) *About Gamasutra.* Available from: https://www.gamasutra.com/features/postmortem/ [Accessed 14 July 2020]

Game developers conference (2020) *About GDC.* Available from: https://gdconf.com/ [Accessed 14 July 2020]

Game Developer Magazine (2021) *GDC Vault Complete Archives.* Available from: https://www.gdcvault.com/gdmag [Accessed 15 Feb 2021]

Game Studies (2020) *About Game Studies.* Available from: http://gamestudies.org/2003/about [Accessed 26 October 2020]

Games and Culture (2020) *Journal Description.* Available from: https://journals.sagepub.com/description/gac [Accessed 26 October 2020]

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

Garlan, D., Shaw, M. (1993) An introduction to software architecture. Paper presented at the *Advances in Software Engineering and Knowledge Engineering,*

Gephi (2020) *The Open Graph Viz Platform.* Available from: https://gephi.org [Accessed 26 October 2020].

Graham, L. (2009) *The NPD group: More Americans play video games than go to the movies.* Available from: https://www.npd.com/wps/portal/npd/us/news/press-releases/pr_090520/ [Accessed 10 July 2020].

Gould, P. (2003) *Doing science + culture: How cultural and interdisciplinary studies are changing the way we look at science and medicine*. Norwich: Cambridge University Press.

Gregory, J. (2014) *Game engine architecture, second edition* 2nd ed. Wellesley, Mass: A K Peters.

Gueheneuc, Y., Mustapha, R. (2007) A simple recommender system for design patterns. *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories (EPFPR).*

Harrison, N. (1999) *The Language of the Shepherds – A Pattern Language for Shepherding.* Materials Science.

Helm, R. (1995) Patterns in practice. *SIGPLAN Notices*. 30 (10), pp.337-341.

Hesmondhalgh, D. (2013) *The cultural industries* (3rd ed.). Los Angeles: SAGE.

Huberman, B., Romero, D.M. and Wu, F. (2008) Social networks that matter: Twitter under the microscope. *First Monday*. 14 (1).

Hughes, J.F. (2014) *Computer Graphics. Principles and Practice*. 3rd ed. Upper Saddle River: Addison-Wesley.

Iba, T., Isaku, T. (2012) *Holistic Pattern-Mining Patterns A Pattern Language for Pattern Mining on a Holistic Approach.* In: 19th Conference on Pattern Languages of Programs Proceedings. New York: ACM.

Iba, T., Yoder, J. (2014) *Mining interview patterns – patterns for effectively obtaining seeds of patterns.* In: 10th Latin American Conference on Pattern Languages of Programs. New York: ACM.

Id Software. (1993) *Prerelease: Doom (PC, 1993).* Available from: https://tcrf.net/Prerelease:Doom [Accessed 10 July 2020]

International Journal of Gaming and Computer Mediated Simulations (2020) *About IGI Global.* Available from: https://www.igi-global.com/about/ [Accessed 26 October 2020]

International Journal of Game-Based Learning (2020) *About IGI Global.* Available from: https://www.igi-global.com/about/ [Accessed 26 October 2020]

Kerr, A. (2006) *The business and culture of digital games: Gamework/gameplay*. London: SAGE.

Kent, S.L. (2001) *The Ultimate History of Video Games: From Pong to Pokemon and Beyond*. Roseville, Calif: Prima.

Khomh, F., Guéhéneuc, Y. (2018) Design patterns impact on software quality: Where are the theories? *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 15-25, doi: 10.1109/SANER.2018.8330193.

Khomh, F., and Gueheneuc, Y. (2008) Do Design Patterns Impact Software Quality Positively? *2008 12th European Conference on Software Maintenance and Reengineering*, Athens, 2008, pp. 274-278, doi: 10.1109/CSMR.2008.4493325.

Knorr-Cetina, K. D. (1999) *Epistemic cultures: How the sciences make knowledge*. London; Cambridge, Mass: Harvard University Press.

Koomen, C. J. (1991) The basic design cycle. *The design of communicating systems: A system engineering approach*. pp. 3-10. Boston, MA: Springer US. doi:10.1007/978-1-4615-4020-5.

Koyuncu, A., Liu, K., Bissyande, T. F., Kim, D., Klein, J., Monperrus, M. and Le Traon, Y. (2020) FixMiner: Mining relevant fix patterns for automated program repair. *Journal of Empirical Software Engineering*. 25(3), pp. 1980.

Kushner, D. (2003) Masters of doom: How two guys created an empire and transformed pop culture. Austin: Kirkus Media LLC.

Lange, D., Nakamura, Y. (1995) Interactive visualization of design patterns can help in framework understanding. *Proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications.* ACM Press.

Lanum, C.L. (2016) Visualizing Graph Data. 1st ed. Manning Publications.

Lear, J., Scarle, S., McClatchey, R. (2019) Asset Pipeline Patterns: Patterns in Interactive Real-Time Visualization Workflow. *24th European Conference on Pattern Languages of Programs Proceedings (EuroPLoP'19).* Irsee, Germany, 3-7 July 2019. New York: ACM.

Lee, M., Green, B., Xie, F., Tabellion, E. (2017) Vectorized production path tracing. *Proceedings of High Performance Graphics.* New York: ACM.

Lengyel, E. (2010) *Game engine gems, volume one*. United States: Jones and Bartlett Publishers.

Levallois, C. (2013) Force Atlas 3D: an open source plugin for Gephi to visualize networks in 3D. Simply install this plugin in Gephi, then explore any network in 3D.

Lewis, M., Jacobson, J. (2002) Game engines in scientific research. *Communications of the ACM, 45*, 27-31.

Llopis, N. (2004) Optimizing the content pipeline. *Game Developer.* pp. 36-44.
Lowood, H. (2014) Game Engines and Game History. *History of Games International Proceedings.* Kinephanos, ISSN 1916-985X.

Mawal, M., Elish, M. (2013) A Comparative Literature Survey of Design Patterns Impact on Software Quality. *2013 International Conference on Information Science and Applications (ICISA).*

Marchand, A., Hennig-Thurau, T. (2013) Value creation in the video game industry: Industry economics, consumer benefits, and research opportunities. *Journal of Interactive Marketing, 27*(3), pp. 141-157.

McGrath, M. B., Brown, J. R. (2005) Visual learning for science and engineering. *IEEE Computer Graphics and Applications, 25*(5), 63. doi:10.1109/MCG.2005.117

Meszaros and Doble (1997) A Pattern Language for Pattern Writing. *In 3rd Pattern Languages of Programming Conference.*

Netlytic (2020) *Overview.* Available from: https://netlytic.org [Accessed 20 September 2020]

Norberg-Schulz, C.. (1980) *Genius Loci: Towards a Phenomenology of Architecture.* 1st ed. New York: Rizzoli International Publications.

NVivo (2021) *Unlock insights in your data with powerful analysis.* Available from: https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/home [Accessed 15 Feb 2021]

Nystrom, R. (2014) *Game programming patterns*. Great Britain: Genever Benning.

Object, G. M., Meszaros, G., Doble, J. (1996). MetaPatterns: A pattern language for pattern writing. Paper presented at the *3rd Pattern Languages of Programming Conference,* pp. 4-6.

O'Donnell, C. (2014) *Developer's Dilemma: The Secret World of Videogame Creators*. Cambridge, Massachusetts: The MIT Press.

Orland, K., Steinberg, S., Thomas, T. (2007) *The videogame style guide and reference manual*. Lulu.com.

Parliament, ed. (2012) *Building Information Modeling*. London: The Stationery Office.

Paravizo, E., Braatz, D. (2019) Using a game engine for simulation in ergonomics analysis, design and education: An exploratory study. *Applied Ergonomics, 77*, pp. 22-28.

Pattern Languages of Programs (2020) *Conferences.* Available from: https://www.hillside.net/conferences [Accessed 26 October 2020]

Perron, B., Wolf, M. J. P. (2009) *The video game theory reader 2*. Abingdon: Routledge.

Pixologic (2020) *ZBrush at a Glance.* Available from: http://pixologic.com/features/about-zbrush.php [Accessed 26 October 2020].

Polygon (2020) *ZeniMax, Facebook settle $250M suit over Oculus VR tech.* Available from: https://www.polygon.com/2018/12/12/18137706/zenimax-facebook-lawsuit-john-carmack-palmer-luckey [Accessed 26 October 2020].

Purewal, J. (2010) *The Problem with Non Disclosure Agreements.* Available from: https://www.gamasutra.com/blogs/JasPurewal/20100513/87319/The_Problem_With_Non_Disclosure_Agreements.php [Accessed 14 July 2020].

Rainie, L. (2014) Networked: The New Social Operating System. Cambridge, MA: The MIT Press. 2014.

Rao, N. S., Larson, J. C., Griffin, B. N. (1992) Lunar Rover Simulator Development Study Based on a Modular Simulation Architecture. *Sage Journals, Simulation.*

Rising, L. (1998) The patterns handbook: Techniques, strategies, and applications. *Computers & Mathematics with Applications (1987)*. 36 (8), pp. 118.

Ritterfeld, U., Cody, M. J., Vorderer, P. (2009) *Serious games: Mechanisms and effects*. London; New York: Routledge.

Salen, K., Zimmerman, E. (2004) *Rules of play: Game design fundamentals*. London; Cambridge, Mass: MIT.

Sanglard, F. (2018) *Game engine black book: Doom*. London: Create Space Independent Publishing Platform.

Schmidt, D. C., Stal, M., Rohnert, H., Buschmann, F. (2000) *Pattern-oriented software architecture, patterns for concurrent and networked objects* (1st ed.). Chichester: John Wiley & Sons, Incorporated.

Shreiner, D., Kessenich, J., Sellers, G. (2016) *OpenGL programming guide: The official guide to learning OpenGL.* Addison Wesley.

Simpson, J. (2002) *Game engine anatomy 101.* Available from: https://www.extremetech.com/computing/50938-game-engine-anatomy-101-part-i [Accessed 14 July 2020]

Simulation & Gaming (2020) *About this journal.* Available from: https://journals.sagepub.com/home/sag [Accessed 26 October 2020]

Siwek, S. (2017) Video games in the 21st century the 2017 report. *Entertainment Software Association.*

Smith, B.L. (2012) *3ds Max Design Architectural Visualization*. 1st ed. Focal Press.

Smith, M. (2007) Postmortem: Resistance: Fall of Man. *Game Developer Magazine.* 14 (2): pp. 28-36.

The Computer Games Journal (2020) *Aims and Scope.* Available from: https://www.springer.com/journal/40869/aims-and-scope [Accessed 26 October 2020]

The Hillside Group (2021) *Patterns Catalog.* Available from: https://hillside.net/patterns/patterns-catalog [Accessed 19 Feb 2021]

Thier, D. Forbes (2020) *Animal Crossing: New Horizons Sold Even More Than You Think It Did.* Available from: https://www.forbes.com/sites/davidthier/2020/08/06/animal-crossing-new-horizons-sold-even-more-than-you-think-it-did [Accessed 1st April 2020].

University of the West of England (2016) *New £50 Million Building for UWE Bristol Faculty of Business and Law.* Available from: https://info.uwe.ac.uk/news/UWENews/news.aspx?id=2892 [Accessed 15/05/2017].

Unreal Engine 5 (2021) *A first look at Unreal Engine 5.* Available from: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5 [Accessed 10/02/2021].

Vehkalahti, K. (2008) Handbook of Data Visualization edited by Chun-houh Chen, Wolfgang Härdle, Antony Unwin. *International Statistical Review*. 76(3), pp. 442-443.

Vlissides, J. (1998) *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Professional.

Ware, C. (2012) *Information visualization: Perception for design*. 3rd ed. US: Morgan Kaufmann Publishers Inc.

Wellhausen, T. and Fießer, A. (2011) How to write a pattern? A rough guide for first-time pattern authors. *Proceedings of the 16th European Conference on Pattern Languages of Programs*. Irsee, Germany, ACM.

Wendorff, P. (2001) Assessment of design patterns during software reengineering: lessons learned from a large commercial project. *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal. pp. 77-84, doi: 10.1109/CSMR.2001.914971.

Wise, R. (2000) The Media Reader: Continuity and Transformation (London: Sage in Association with the Open University, 1999), 425pp. ISBN 0 761 96 2506. *Convergence: The International Journal of Research into New Media Technologies*. 6 (2), pp.129-131.

Wydaeghe, B., Verschaeve, K., Michiels, B., Van Bamme, I., Arckens, E. and Jonckers, V. (1998) Building an OMT-editor using design patterns: an experience report. *IEE*. pp.20-32.

Yu, D., Zhang, Y. and Chen, Z. (2015) A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *The Journal of Systems and Software*. 103 pp.1-16.

Zackariasson, P., Wilson, T. L. (2012) *The video game industry: Formation, present state, and future*. London: Routledge.

Zanoni, M., Arcelli Fontana, F. and Stella, F. (2015) On applying machine learning techniques for design pattern detection. *The Journal of Systems and Software*. 103 pp.102-117.

Zhang, C., Budgen, D. (2012) What Do We Know about the Effectiveness of Software Design Patterns? *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213-1231, Sept.-Oct. 2012, doi: 10.1109/TSE.2011.79.

Zink, J., Pettineo, M. and Hoxley, J. (2011) *Practical Rendering and Computation with Direct3D 11.* 1st ed. Hoboken: CRC Press.

Appendix 1 – Video Game Development

In order to further situate this research, it is important to have an appreciation of the key terminology and structure of the video game industry, the value chain involved and the relationship between the participants. The introductory contextualization is offered below to provide academic readers with the necessary scaffolding.

The Video Game Value Chain

The functionaries within the video game industry are similar to those of other publishing industries such as film, television and literature (Caves, 2000; Hesmondhalgh, 2013). The value network typically includes the steps of content creation, content publication, content distribution, content retail and content consumption (Bogdanowicz *et al.*, 2010). Collectively these roles are fulfilled with video game and middleware developer, publisher, distributor, retailer and end-user customer, figure 8.1.

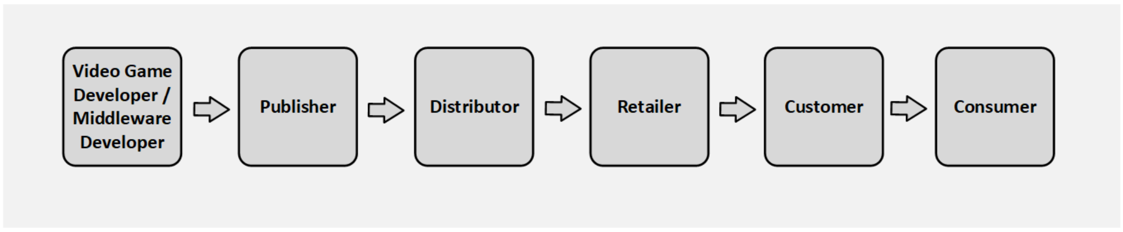


Figure 8.1: The video game industry value network

The video game developer is the participant that designs and implements the video game comprising of art, design, software engineering, audio, user experience (UX) and quality assurance (QA) testing (Chandler, 2019). A video game developer may target a specific video game console (Sony PlayStation, Microsoft Xbox, Nintendo Switch) or may target a variety of hardware platforms including personal computer (PC), tablet or mobile (Bogdanowicz *et al.*, 2010). Embodied in a studio the developer employs a large number of multidisciplinary professions working to develop individual components of the video game, ensuring cooperatively they form a coherent immersive experience for the end-user. Typical studios range in size from very small independent teams to very large studios employing several hundred developers producing multiple video games in parallel. In the United States alone the video game software industry directly employs some sixty-five thousand people and indirectly greater than two hundred and twenty thousand (Siwek, 2017). Pivotal to the software development process is the central role of middleware (Bogdanowicz *et al.*, 2010). Middleware is the software layer that is situated between the Operating System, device drivers and the end-user application, the video game (Zackariasson and Wilson, 2012). The complexity of game design and the performance requirements of a video game has grown to unmatched levels in the last twenty years (Lengyel, 2010). To handle this complexity an assemblage of reusable software components has consolidated into a single reusable entity called the game engine (Charrieras and Ivanova, 2016).

For those Triple-A (AAA) video games of high quality and expected high sales the development costs can consume tens or hundreds of millions of dollars (DeLoura, 2009). One such video game, “Call of Duty Modern Warfare 2” developed by the game studio Infinity Ward, cost $265 million (Activision, 2009). Game studios typically lack the financial capability to fund, promote and bring their video game to market. It is the role of the publisher to finance the development process, acquire the intellectual property rights for new games and provide marketing to distributors, retailers and end-users. Presently in 2020 the most successful publishers are international companies with strong bargaining power (Kerr, 2006). A publisher will rarely specialise in one platform but elect for diversification to achieve economies of scale (Bogdanowicz *et al.*, 2010). Major publishers have headquarters located in North America (Activision/Blizzard, Take Two, Sony, Electronic Arts, Microsoft, THQ) and Europe (Ubisoft France).

The distributor operates as the middleman between the publisher and the retailer. The responsibilities are concerned with the sales agreements as well as the logistics of the distribution chain. Retailers exist either in physical or online site form. The general distribution trend for video games follows that as for all consumed media, such as books and music. That is a move away from physical form to digital. Major hardware platform manufacturers such as Microsoft, Sony and Nintendo have an online shop presence offering direct digital distribution with other third-party distributors such as Valve following suite. This is especially prevalent in the mobile video game application arena with both the App Store and Google Play providing digital store fronts (Zackariasson and Wilson, 2012).

The Video Game and Development

Video games consist of virtual worlds modelled as an approximation of either a real or imaginary environment. These worlds are often expansive, populated with characters, architecture and realistic environments. The video game operates as a mathematical simulation, where distinct entities or agents interact. End users or players navigate and explore the virtual world using a Human Interface Device (HID), such as a game controller, and the simulated world responds with appropriate reactions. Visual feedback to the user is provided through a graphical view into the world. This view must be rendered at a sufficiently high frame rate to achieve the illusion of motion for the end user (Perron and Wolf, 2009). To add to this complexity, all the above conflicting steps must be performed in near real-time to invoke the level of immersion that games aspire to. For the reasons outlined above Gregory (2014, p.9) succinctly defines video games as “soft real-time interactive agent-based computer simulations.”

Design and implementation of a video game requires a number of distinct skillsets or roles from the development team. The roles of the developer vary, however, in general there is a requirement for software engineers, artists, audio engineers, designers and producers (figure 8.2).

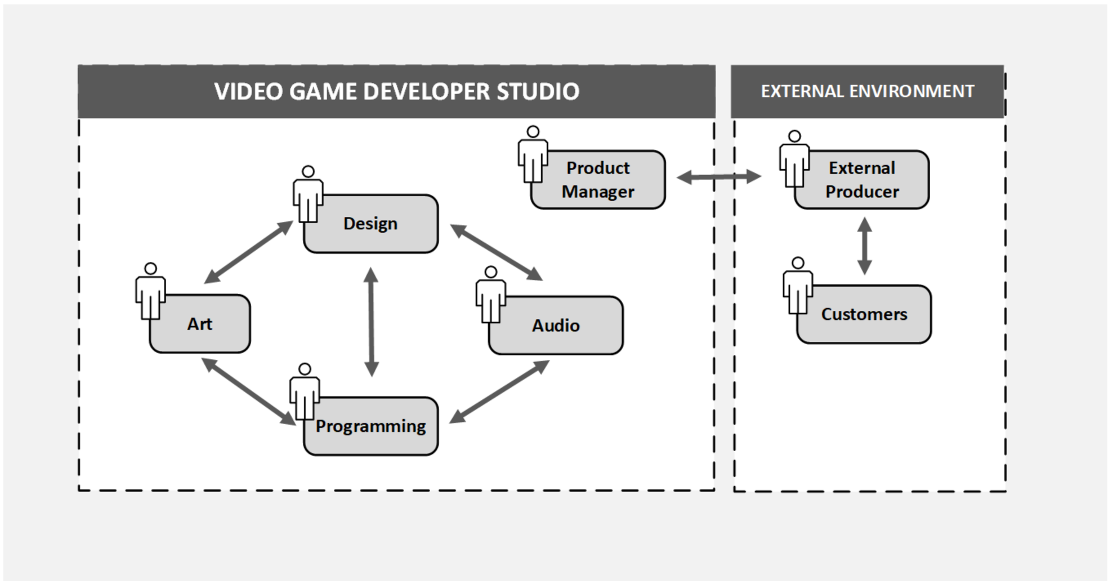


Figure 8.2: The roles of the video game developer

Software engineers or programmers develop and modify the software modules and technical assets that enable video games to be compiled, built, executed and tested (Chandler, 2019). Although historically considered the galvanizing force behind video game development, programmers now frequently only compose thirty to forty per cent of the workforce (Zackariasson and Wilson, 2012). Programmers are responsible for developing software that consumes the art content produced by creatives along with design data produced by game designers into the game engine. The programmer either extends and integrates with an existing game engine, a proprietary development environment or a combination of the two. The duties of the programmer can involve both high-level scripting and low-level programming. Depending on the studio size programmers may take on speciality roles such as game engine development, tools development, visual effects (VFX) development, network development or artificial intelligence (AI). Owing to the specialization the programmer integrates and communicates closely with one or more of the other disciplines (Chandler, 2019).

Appendix 2 – Games Technology Surveys

Game Engine	Game Gender & Renderer	Target Platform	Starting Price for Commercial Use	Scripting Language	Supported Platforms
Unity 2020	Generic – 2D/3D	Windows; macOS; iOS; Android; Linux; WebGL; PS4, Xbox One; 3DS, Oculus Rift; Steam VR; PS VR; Windows Mixed Reality; Nintendo Switch; Apple ARKit; Google Daydream	Free for companies generating less than $100,000 annually.	C#	Windows; macOS; Linux
Unreal Engine 4	Generic – 2D/3D	Windows; macOS; iOS; Android; Linux; HTML5; PS4; Xbox One; Oculus Rift; HTC Vive; HoloLens 2; Nintendo Switch; Google Daydream	Free initially and pay 5% when product succeeds.	C++	Windows; macOS; Linux
Amazon Lumberyard	Generic – 2D/3D	Windows; PS4; Xbox One; iOS; Android	Free	C++ or Lua	Windows
CryEngine	Generic – 3D	Windows; Linux; Nintendo Switch; PS3; PS4; Wii U; Xbox 360; Xbox One;	5% revenue sharing model	C++ or Lua	Windows; macOS; Linux
GameMaker Studio 2	Generic 2D	Windows; macOS; Ubuntu; HTML5; Android; iOS; Amazon Fire TV; Android TV; PS4; Xbox One; Nintendo Switch	Variety of licensing models ranging from free to console licenses.	GameMaker Language	Windows; macOS; Android; iOS; FireTVl PS4; Xbox One
RPG Maker XP	Role-Playing Games (RPG)	Super Famicom; Windows; Sega Saturn; PlayStation; GameBoy Color; PS2; GameBoy Advance; Nintendo DS; Nintendo 3DS; Linux; Nintendo Switch; PS4; macOS	$18.99	Not Required	Windows

Table 9.1: Game engine survey updated and extending survey of Andrade (2015)

DCC Tool	Asset Type	Source Format	Starting Price for Commercial Use	Extension Language	Supported Platforms
Autodesk Maya 2020	3D models	Proprietary – Binary - .mb scene and ASCII .ma scene	$1,872 / year	Maya Embedded Language (MEL) – Scripting	Windows; macOS
Autodesk 3ds Max 2021	3D models and animations	Proprietary – Binary - .3ds scene	$1,872 / year	MAXScript; 3ds Max SDK (C/C++)	Windows
Blender 2.83.4	3D assets and animation	Proprietary – Binary - .blend	Free and Open Source	Python add-on	Windows; Linux; macOS
Pixelogic Zbrush 2021	3D digital sculpts	Proprietary – Binary - .zbr	$895 single user perpetual license	Zscripting	Windows; macOS
Autodesk Mudbox	3D digital sculpts	Proprietary – Binary - .mud	$96 / year	Mudbox SDK	Windows; macOS; Linux
Adobe Photoshop	2D raster graphics	Proprietary – Binary - .psd	$19.97 / month	Photoshop Plugin Extension Builder	Windows; macOS, iPadOS
Substance Painter	Procedural Texture	Proprietary – Binary - .sbsar	$99.90 / month	Integrated add-ons for various DCC Tools	Windows; macOS; Linux
Substance Designer	Node based texturing application	Proprietary – Binary - .sbs	$99.90 / month	Integrated add-ons for various DCC Tools	Windows; macOS; Linux

Table 9.2: Survey of Digital Content Creation (DCC) tools

Appendix 3 – Cycle 1 Shepherding Process

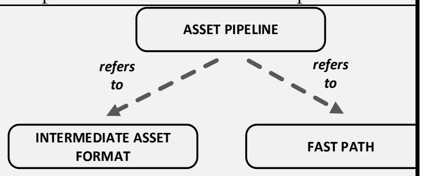
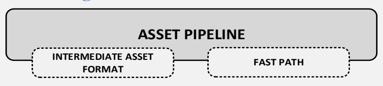
Iteration 1 – Feedback and Resolution

Item	Page Number	Section	Feedback (Cesare Pautasso)	Resolution (James Lear)
1	2	1. Introduction	I am not sure this is a pattern language, given the very small number of patterns involved. Also in the figure you should indicate how the patterns are related.	• Updated to refer to the set of three patterns as a three-pattern catalogue rather than a pattern language. • Updated figure to include the "uses" relationship.
2	3	2.1 Asset Pipeline - Context	How coupled is the design <-> runtime? Can you have more than one target runtime for the same design tool? Or can you target the same runtime from multiple design tools?	• Yes, multiple DCC tools can produce content for a runtime. Furthermore, the content may execute on multiple runtime environments. • The context section has been updated to reflect this. Specifically, the architectural visualization example explains this with tangible examples.
3	4	2.1 Asset Pipeline – Source Asset Dependencies (figure 2.3)	Just wondering if you must use the same representation for stateful components also to show the semantics of what essentialy are data files? If you would add a legend to the diagram what would you label each shape with? Is there a need to include each shape? and are these shapes referring to the same model elements as in the previous figure 2.2?	• The diagram represents the dependency graph for a scene containing multiple objects (landscape, trees and building). • Changed and simplified the diagram. Added a legend to define the symbols.
4	4	2.1 Asset Pipeline – Source Asset Dependencies (figure 2.3)	Are the edges of the graph always of the same kind?	• For the example scenario of an architectural visualization the leaf nodes consist of the texture images. These must be transformed into their final format prior to being mapped onto the corresponding 3D object. I have changed the text in this area. • The edges of the graph may not always be of the same type. For the example this is the case, however, if environmental sound was also included in the visualization the sound files would require processing.
5	5	2.1 Asset Pipeline - Problem	This problem statement sounds like a constraint that needs to be satisfied. So maybe it is another force? If you would turn the problem into a question, what would it be?	• The asset pipeline pattern solves the problem of digital content creators designing in DCC tools that use high level concepts that require visualizing in a runtime requiring optimized versions. • **QUESTION: Do you think the following are better problem statements?** • **"How can interactive visualizations be created that rely on both the design of source assets using high-level DCC concepts and the final optimized runtime assets?"** • **"What is the workflow for creating interactive visualizations?"** • **"What is the workflow for creating interactive visualizations whereby the source assets are created in a high-level DCC tool and the**

				visualization operates within an optimized runtime?”
6	5	2.1 Asset Pipeline - Solution	Is this optimization always required?	• Yes, the optimization is always required.
7	5	2.1 Asset Pipeline - Solution	What is the role of standard formats?	• Reworded and added an explanation of standard file format interoperability.
8	5	2.1 Asset Pipeline - Solution	The figure only shows one stage - maybe you have some concrete known uses where you can show multiple stages and why they are needed?	• Updated the figure to include multiple stages (1..n). Concrete example is provided in figure 2.7.
9	6	2.1 Asset Pipeline – Solution – Figure 2.5	I find it difficult to picture two parallel processing steps writing into the same final output artifact. Is this what you are trying to show?	• You are correct. There is not a situation where two processes write to an identical output artefact. I have now changed the diagram. A build node may take as its input multiple input files and may also output multiple output files (n..m).
10	6	2.1 Asset Pipeline – Solution – Figure 2.6	Is this always 1 input asset transformed into 1 output asset, or do you also have N-M transformations?	• n..m transformations also exist. I have updated figure 2.5, figure 2.6 and the text to state this. I have incorporated cardinality in the diagrams (1..*).
11	6	2.1 Asset Pipeline – Solution – Figure 2.7	Nice example, the more specific and concrete the better. Also what happened to the artifacts shapes between the processing nodes? The caption is a bit generic.	• Changed the diagram to incorporate artifacts. Diagram is now hopefully clearer. • Changed the caption to detail the functionality.
12	7	2.1 Asset Pipeline – Solution – Asset Dependency Evaluation	Sounds like there are two pattern variants here? Hard-coded asset pipeline vs. Declarative asset pipeline? Have you considered separating them within your “pattern language”? Maybe it would be useful if they have fundamentally different consequences and if they would solve slightly different problems. Right now they are kind of buried in this paragraph.	• The two approaches were mentioned as possible solutions to asset dependency evaluation. • TO DO: Either expand/change description or separate.
13	7	2.1 Asset Pipeline – Solution – Benefits	Before going into the details, please double check these are connected with the forces you opened the pattern with.	• Added “Design Feedback Loop” to ensure all forces satisfied by the consequences – benefits.
14	7	2.1 Asset Pipeline – Solution – Example	A good pattern should have at least 3 known uses. So this example is a good starting point but you may want to expand it and look for real-world rendering systems which use the asset pipeline as you have described it.	• Added “Game Engine Asset Pipeline” as an initial example. • TO DO: Add further examples. Most likely Architectural Visualization.
15	8	2.2 Intermediate Asset Format – Forces – Figure 2.8	Not sure I can see the intermediate format in the figure above	• The intermediate format should not be represented in this diagram. The diagram reflects a build process prior to the incorporation of the intermediate file format. The figure was incorrectly labelled. The label has now been changed to reflect this.
16	8	2.2 Intermediate Asset Format – Forces – Problem	This one sounds like a consequence of the context, i.e., a force you are trying to avoid, a challenge which makes the problem hard. But I am not sure it is stated as a well defined problem. I suspect the coupling extends beyond the final runtime format and also involves the intermediate processing steps of the pipeline? Like before: think of which question your solution is trying to answer.	• The asset pipeline pattern solves the problem of digital content creators designing in DCC tools that use high level concepts that require visualizing in a runtime requiring optimized versions. • **QUESTION: Do you think the following is a better problem statement?** • **“How to introduce flexibility in the asset pipeline?”**

17	8	2.2 Intermediate Asset Format – Forces – Solution	Can you always directly export into such intermediate format? what if the DCC tool only exports into a proprietary format? in the next page you write about “converting into the intermediate format”, please check it is consistent with this sentence.	• Changed the phrasing of the first sentence to “initially converted”. The next page explores the mechanics behind this conversion; either exported directly from the DCC tool or converted within the first build step.
18	8	2.2 Intermediate Asset Format – Forces – Solution	Is this efficiency concern part of the problem? or maybe another force?	• Within this context the efficiency concern is not part of the problem. It is used in this context to explain the intermediate format should contain all data necessary for all steps in the pipeline.
19	9	2.2 Intermediate Asset Format – Forces – Solution	Here’s another force! Also, dealing with errors appears somewhat as a surprise, so it should be mentioned somewhere also in the previous pattern?	• TO DO: Investigate whether transparency of format is another force or a side-effect.
20	10	2.2 Intermediate Asset Format – Forces – Solution	Known use?	• TO DO: Add further examples. COLLADA and other formats.
21	10	2.2 Intermediate Asset Format – Forces - Consequences	So do you really need a single container format?	• A single container format resolves the forces indicated. Multiple formats requires a more complicated build process and potentially multiple asset pipelines.
22	11	2.3 Fast Path – Problem	Also in this case, the problem is a constraint that is not 100% related to the solution. What about something like “How to give a fast preview of the assets without having to wait for the final assets to be built?”	• Your problem statement is much better than in the original paper. I have changed the statement to yours.
23	11	2.3 Fast Path – Solution	This is surprising, nowhere until here it was written that the runtime can display intermediate assets as well.	• TO DO: Add further explanation.
24	11	2.3 Fast Path – Consequences	Example with known uses is missing	• TO DO: Add example use cases.

Iteration 2 – Feedback and Resolution

Item	Page Number	Section	Feedback (Cesare Pautasso)	Resolution (James Lear)
1	2	1. Introduction – Figure 1.1	Thank you for adding the relationships! I think there is a bit more potentially left to uncover regarding those.	
2	2	1. Introduction – Figure 1.1	Is the intermediate asset format a stand-alone pattern? Can it work outside the context of an asset pipeline? And viceversa, can you have an asset pipeline without “using” the I.A.F. ?	• The optional pipeline can be applied in isolation or may be optionally complemented with the Intermediate Asset Format and FAST PATH. The two component patterns cannot exist without the parent ASSET PIPELINE pattern.
3	2	1. Introduction – Figure 1.1	By following the arrows and reading the caption, one would understand that the asset pipeline is built by using the other two patterns. However, I wonder, as these necessary? Can you build an asset pipeline without fast path? And to introduce the fast path, do you require the asset pipeline as context? Could fast path be seen as a specialised form of asset pipeline?	 ASSET PIPELINE refers to refers to INTERMEDIATE ASSET FORMAT FAST PATH • Changed the above diagram to the following:  • **QUESTION: If you feel the arrow relationships were more informative, I can revert the diagram, however, I can include a key?**

				• Also added the following sentence in the description, "The base ASSET PIPELINE pattern can be applied in isolation or may be optionally complemented with the two additional component patterns" • Added pattern category column to table 1.1.
4	3	2.1 Asset Pipeline - Figure 2.1	Looks nice!	
5	5	2.1 Asset Pipeline – Figure 2.4	The legend helps to understand the diagram better. Is export the only possible case/value written inside the white box? What is DCC? (One may not necessarily read the text before looking at the figures) - consider moving the Painting Tool and 3D Modeling Tool inside the box instead of DCC	• The individual asset files are produced using the mechanism of export. • Therefore, the individual nodes represent the DCC tools, processing and then exporting. • I have now added the little "processing" icon (box with circular arrow). • I moved the Painting Tool and 3D Modeling Tool inside the box as suggested.
6	5	2.1 Asset Pipeline – Figure 2.4	The legend helps to understand the diagram better. Is export the only possible case/value writen inside the white box? What is DCC? (One may not necessarily read the text before looking at the figures) - consider moving the Painting Tool and 3D Modeling Tool inside the box intead of DCC	• Removed DCC from diagram. As above, moved the DCC type inside the box.
7	7	2.1 Asset Pipeline – Figure 2.7	Cardinalities can be mis-interpreted. Does this mean that 1 input asset can be fed into 1 or more build nodes? But that each build node only consumes one input asset?	• Cardinality was incorrect on this diagram. Thanks for finding this. I have reversed. One or more input assets can be processed by an asset build node. This may result in one or more output assets.
8	7	2.1 Asset Pipeline – Figure 2.7	Also on this side, do you mean that 1 output asset is produced by 1 or more build nodes? Or are you trying to say that 1 build node can produce one or more output assets? Please make sure this is also mentioned in the corresponding text (to avoid ambiguity)	• Changed diagram cardinality. Description above diagram indicates, • " Each square node within the asset build process in figure 2.6 represents a single step that performs a transformation on one or more input assets into one or more output assets"
9	10	2.1 Asset Pipeline – Examples	Is this connected to Fig 2.4. where you show dependencies between assets?	•
10	10	2.1 Asset Pipeline – Examples	The example reads really well to illustrate the abstract pattern description. I would just mention a few more application domains (architectural visualisation as you suggested makes sense) and maybe mention and give references some of the tools used to build such pipelines specific for that domain. Also VR/AR comes to mind.	• Added a further example, "Architectural Visualization" as you suggest.
11	16	2.3 Fast Path	If you have no negative consequences, then the question would be: what would be the reason not to use the fast path? why is the normal (slow) path still needed? Answering this may also help improve the force section	• TO DO: Add negative consequences. Custom Intermediate Format Importer required for visualization runtime (next iteration)

Iteration 3 – Feedback and Resolution

Item	Page Number	Section	Feedback (Cesare Pautasso)	Resolution (James Lear)
12	7	2.1 Asset Pipeline – Solution – Asset Dependency Evaluation	Sounds like there are two pattern variants here? Hard-coded asset pipeline vs. Declarative asset pipeline? Have you considered separating them within your "pattern language"? Maybe it would be useful if they have fundamentally different consequences and if they would solve slightly different problems. Right now they are kind of buried in this paragraph.	• Changed description. Currently this is considered an implementation discussion. Separating into two patterns is an interesting idea. Consider separating after feedback from Writer's Workshop.
19	9	2.2 Intermediate Asset Format – Forces – Solution	Here's another force! Also, dealing with errors appears somewhat as a surprise, so it should be mentioned somewhere also in the previous pattern?	• Removed this reference to build errors, as did not want the patterns focus and emphasis to be at this level.
20	10	2.2 Intermediate Asset Format – Forces – Solution	Known use?	• Added example. Use of COLLADA in the Asset Pipeline.
23	11	2.3 Fast Path – Solution	This is surprising, nowhere until here it was written that the runtime can display intermediate assets as well.	• Changed description within Solution.
24	11	2.3 Fast Path – Consequences	Example with known uses is missing	• TO DO: Iteration 4 add an example.
11	16	2.3 Fast Path	If you have no negative consequences, then the question would be: what would be the reason not to use the fast path? why is the normal (slow) path still needed? Answering this may also help improve the force section	• Added to Liabilities section.
12			The asset pipeline pa‹ern solves the problem of digital content creators designing in DCC tools that use high level concepts that require visualizing in a run‹me requiring op‹mized versions. This is a great pattern summary, please write it at the top of Section 2.1 (after the AKA before the Context).	• Added the pattern summary as mentioned. TO DO: Add pattern summaries for the other patterns for consistency in pattern template (Iteration 4).

Appendix 4 – Cycle 1 Asset Pipeline Patterns

<u>ASSET PIPELINE</u>

<u>Category:</u> Asset Pipeline Workflow

<u>Also Known As:</u> Content Pipeline

<u>Pattern Summary:</u> The asset pipeline pattern solves the problem of digital content creators designing in DCC tools using high level concepts that require visualizing in a runtime requiring optimized versions.

<u>Context:</u> ***Designers require a workflow to assist them in viewing their digital content in an interactive real-time visualization.***

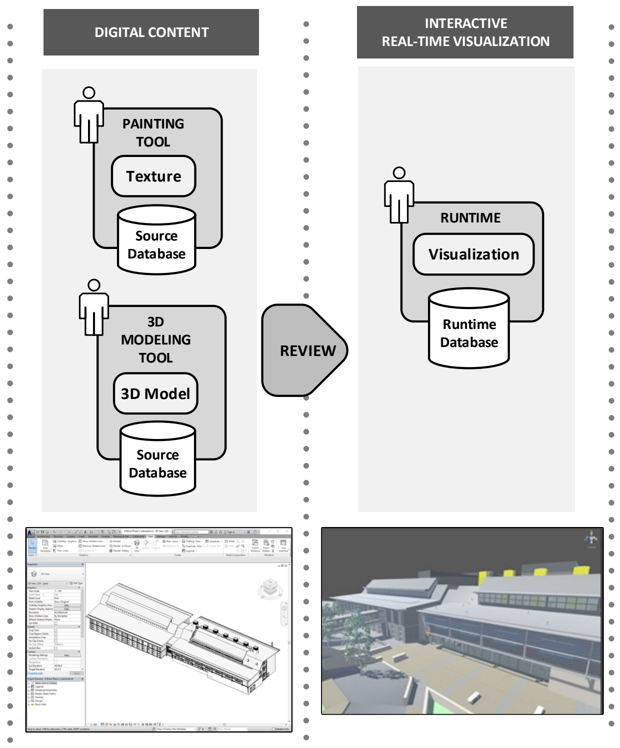


Figure 11.1: Digital Content Creation and Interactive Real-Time Visualization

Consider the design and development of a real-time interactive visualization such as an architectural visualization as shown in figure 11.1. Designers use Digital Content Creation (DCC) tools to create a number of source assets in the form of three-dimensional models and textures. Typically, the architecture, engineering and construction (AEC) industry design and specify the structure using a Building Information Modelling DCC tool such as AutoDesk Revit. An output of the process is the three-dimensional geometric model of the building. Further detail and textures essential for a high-quality real-time visualization are created using a painting tool such as Adobe Photoshop. Ultimately many DCC tools may be required to produce such a visualization, table 11.1.

DCC Domain	Artifact	Example DCC tools
Building design software	Building Information Modeling (BIM) model incorporating 3D model and document management.	Autodesk Revit, Graphisoft Archicad.

3D modeling and animation	3D mesh incorporating materials and animations.	Autodesk 3ds Max, Autodesk Maya, SketchUp, Blender (open source).
Image manipulation and graphic design	Bitmap image, Vector graphic.	Adobe Photoshop, Adobe Illustrator, Gimp (open source).
Material and texture authoring	Physically Based Rendering (PBR) material.	Adobe Substance Designer, Substance Painter, Substance B2M, Quixel SUITE.

Table 11.1: Example DCC tools used in the production of an architectural visualization

The visualization operates in a runtime environment targeted for a specific platform. For the scenario of a real-time architectural visualization a game engine such as Unity provides the runtime capability of real-time rendering. Target platforms include the Windows desktop environment and Oculus Rift virtual reality (VR) platform.

Assets are optimized for the runtime in question and differ significantly from the source asset structure whose container is in a proprietary format. The designers require a method to review their content, or source assets, within the environment of the runtime to ensure they are aesthetically identical to their design intent. Any such changes require adjustment in the DCC tool and the iterative design cycle proceeds until the source asset is considered complete. This workflow is further complicated when multiple DCC tools are involved in the creation of source assets and the visualization is targeted to multiple runtimes.

<u>Forces:</u>

Design Feedback Loop

The workflow of creating an interactive real-time visualization utilizes the design process of creating the source assets within a DCC tool, transforming the source assets into a compatible form utilized by the visualization runtime and subsequently reviewing the produced visualization. After visual inspection often source assets require modification and hence the feedback loop continues, figure 11.2. Any form of automation in this respect will increase the efficiency of the designer and reduce the iterative cycle of the design-review process.

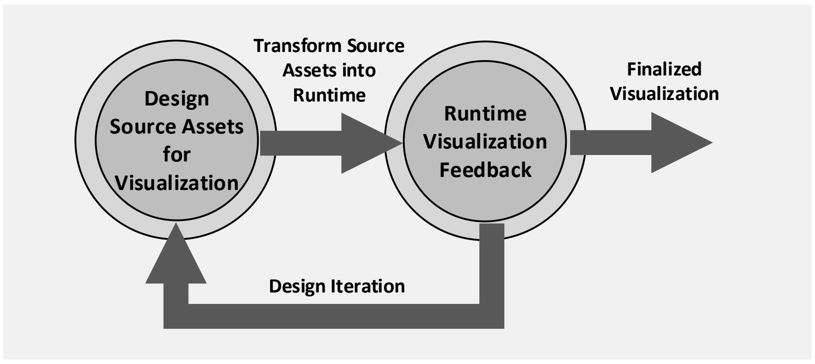

Design Source Assets for Visualization
Transform Source Assets into Runtime
Runtime Visualization Feedback
Finalized Visualization
Design Iteration

Figure 11.2: Design Feedback Loop

Incompatible Source and Final Asset Format

The requirements of the source asset are very different from those expected by the runtime engine. DCC tools allow a designer to work with efficient workflows using enhanced data modelling techniques. The resulting source asset is stored in a proprietary format within the source database of the DCC tool. It is information rich, containing the complete set of semantic data required to reconstruct the DCC tool user-interface in preparation for future editing and modification by the designer. In comparison, the final visualization asset format is lightweight and optimized for loading efficiently into the runtime environment. For instance, three-dimensional model geometry must be presented in a format applicable for submitting to the graphics pipeline on the destination hardware. Likewise, any textures must be formatted to a size appropriate to operate within the limitations of any hardware constraints. Often the visualization designer cannot dictate the format of the final asset since the runtime target platform or third-party runtime engine imposes such a format, and this may not be known at design time.

Traditionally this has been solved by incorporating any custom processing required to massage the source asset into its final format at the specific point of exporting the asset from the DCC tool. To facilitate this, commercial DCC tool vendors provide end-users with the capability to extend the product to incorporate additional functionality. The internal proprietary representation of the data stored in the source database is exposed through a public Application Programming Interface (API). Often catering for common programming languages such as C++, Python and other scripting languages the vendor allows end-users to query and process the data persisted in the source database. Such a system can be exploited by the designer to transform and process the internal representation of the source asset to a format applicable for executing in the runtime. Referring to figure 11.3, a custom export plug-in developed leveraging the DCC tool API transforms and exports the source asset into a final format suitable for the target runtime.

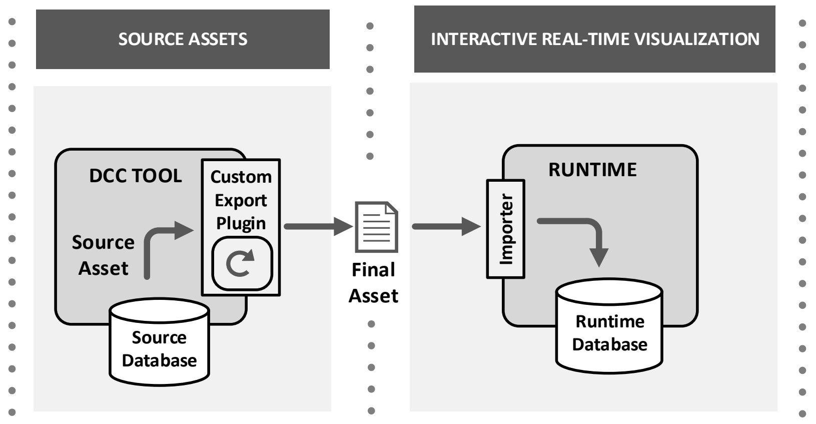


Figure 11.3: Custom export plug-in transforming internal source representation to a final asset

Unfortunately, this approach has several limitations. Firstly, the plug-in interface of DCC tools are not designed to handle such potentially long running processes and complex transformations. Secondly, source asset data is rarely self-contained and normally is dependent on one or more other assets. Therefore, the mismatches between the format of the source and final runtime assets has become a barrier that prevents a designer from achieving their goal of reviewing their designs in the runtime in a fast and efficient manner.

Source Asset Dependencies

It is unusual for a visualization to be composed of a set of completely self-contained source assets. Often dependencies exist where assets reference or use one another. For instance, considering an architectural visualization of a scene consisting of a landscape with a building and many trees as shown in figure 11.4. The overall scene is dependent upon the three-dimensional models of the landscape or terrain, the architectural building and the set of trees. These in turn are dependent upon several texture images giving substance to the three-dimensional models.

For the scenario of where the format of the source assets are not aligned to the final expected runtime format, any processing or manipulation must be performed on the child assets prior to integration with any parent asset. For example, the textures must be processed prior to inclusion and mapping onto the three-dimensional scene models.

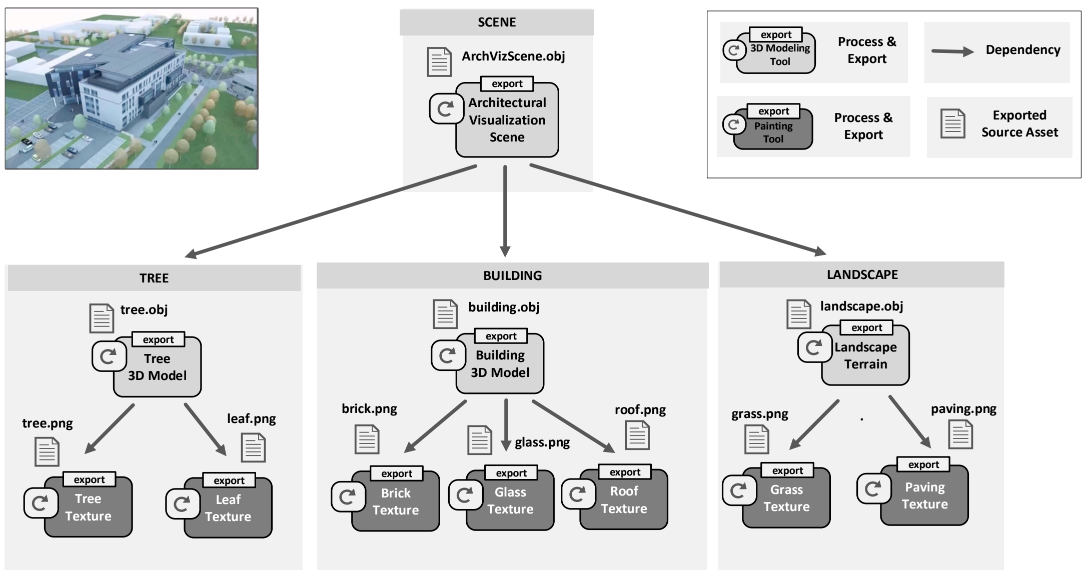


Figure 11.4: Simplified scene graph and dependency hierarchy of an architectural visualization

Source assets are often binary files, held in opaque proprietary formats, from which parsing to determine dependencies is difficult if not impossible. There needs to be a mechanism to manage this dependency problem during any processing that is required.

<u>**Problem:**</u>
How to deliver interactive visualizations whereby the source assets are created in a high-level DCC tool and the visualization operates within an optimized runtime?
<u>**Solution:**</u> ***Integrate an efficient workflow enabling designers to transform their source assets into the final runtime form.***

The workflow is called the asset pipeline and is the mechanism that interfaces the designers of digital content to the visualization runtime. It is the workflow path that all source assets follow, from conception until they are loaded into the runtime engine.

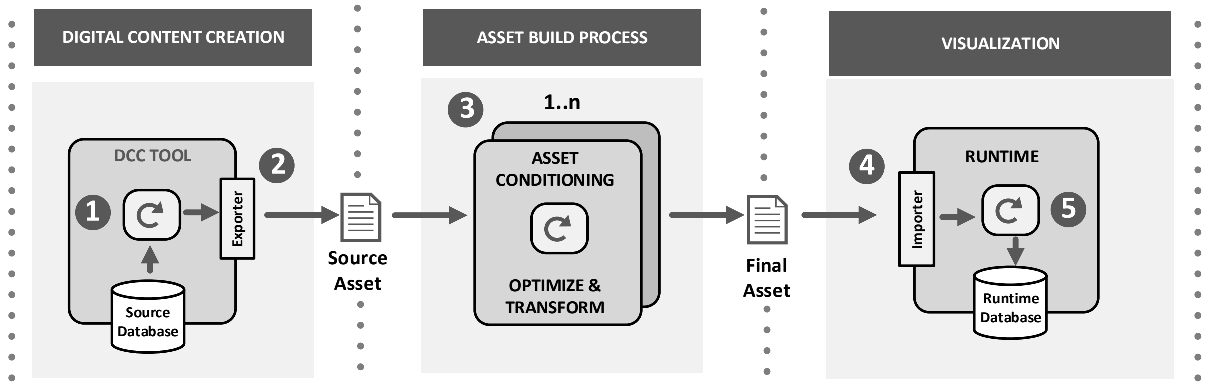


Figure 11.5: Block diagram representing the three stages of the Asset Pipeline

The asset pipeline, illustrated in figure 11.5, is composed of the following components:

1) A DCC tool used by the designer to create the source asset.
2) The DCC exporter permitting the source asset data to be extracted from the source database for further processing.
3) An asset build process whose responsibility is to orchestrate a series of stages to transform the source asset data into a final format optimized and acceptable for the runtime target platform.
4) The runtime importer accepting the final optimized asset and persisting in the runtime database.
5) The visualization runtime incorporating the runtime database, targeted for a particular hardware platform.

DCC Tool Exporter

The exporter of the DCC tool is an integral component of the asset pipeline, since it exposes the complex and proprietary internal representation of the source asset data in a more usable format. With the recognition that source assets are now designed and manipulated within a variety of complementary tools, commercial manufacturers of DCC tools attempt to widen their market value by offering the capability to export content to many different standard file formats. This extends the source asset interoperability between DCC tools made by different manufacturers.

Given the source asset data is the front end to the asset build process the designer must ensure they choose a format where the data container has the capability to hold all the data necessary for the data conditioning pipeline stage. A designer must export the data source in a lossless format in preparation for further processing.

Considering three-dimensional model data, a designer has many possible formats from which to choose. One such container is the OBJ file format, developed by Wavefront Technologies, and is an open source format adopted by many vendors of DCC graphics tools. This simple format stores three-dimensional geometry as a series of positional vertices and has the capability to reference any dependent materials or substances.

Please refer to the INTERMEDIATE BUILD FORMAT Asset Pipeline Component pattern for discussion of where the choice of asset format determines the flexibility of asset processing.

Asset Build Process

The source assets must undergo a number of steps before they can be integrated into the final runtime. The asset build process is the component tasked with performing the source data conditioning.

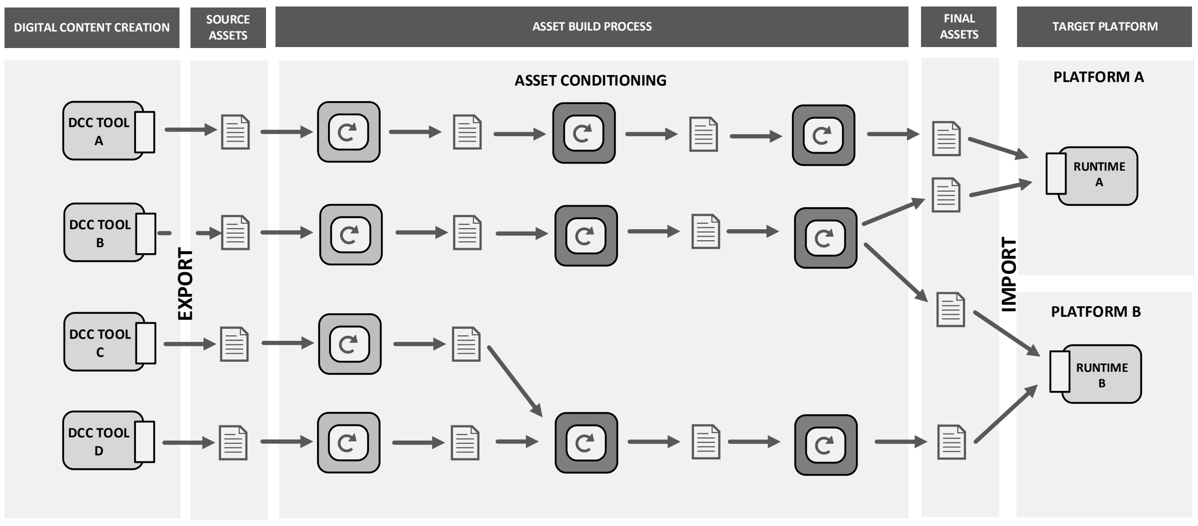


Figure 11.6: Asset Build Process

In a similar manner to source code the source asset must be compiled and optimized for each target runtime platform. Each square node within the asset build process in figure 11.6 represents a single step that performs a transformation on one or more input assets into one or more output assets. This transformation represents an act of either executing a conversion script, a data compiler or a tool resulting in gradual refinement of the source asset into its final representation as consumed by the runtime environment. The exact nature of the steps is driven by several factors including the type of assets involved, the visualization application and the target platforms. Each input asset is a direct dependency for an individual asset build node, and the node itself becomes a direct dependency for each of its outputs, figure 11.7. That is, a many to many relationship exists whereby one output asset file may depend upon many source assets and one source asset may produce many output asset files.

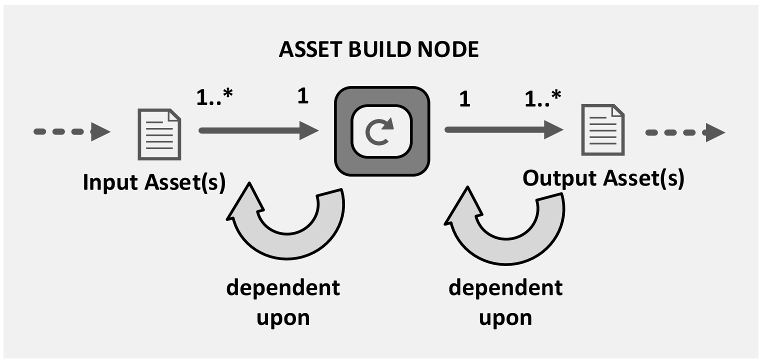


Figure 11.7: Dependencies of an asset build node

Consider for example an asset conditioning pipeline of geometry processing; the designer creates individual scene models based on three-dimensional polygon meshes, constructed as a set of vertices and the associated edges connecting them. Current rendering hardware does not natively support the processing of n-sided polygons. As part of the asset build process stage of the asset pipeline the geometry primitives must be converted to triangles acceptable by the runtime hardware. Such geometry may form the basis for further forms of processing, for instance geometry compression of the high density meshes into lower density forms producing different levels of detail, figure 11.8.

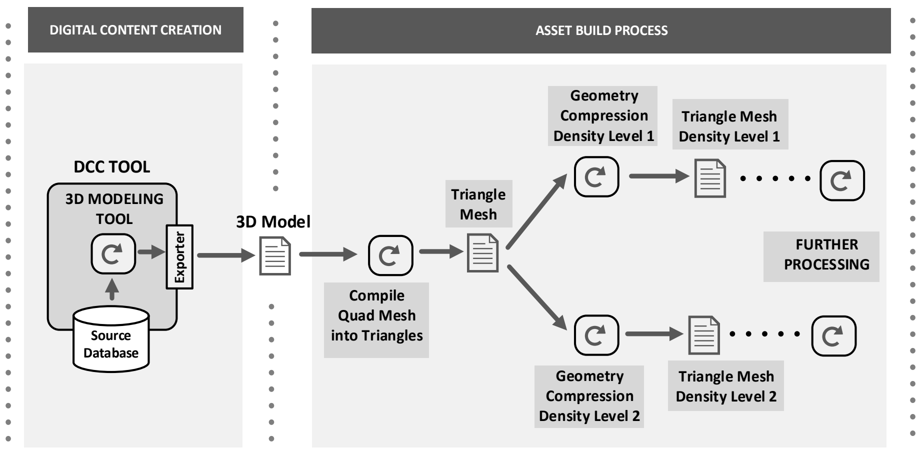

DIGITAL CONTENT CREATION
ASSET BUILD PROCESS
DCC TOOL
3D MODELING TOOL
Exporter
Source Database
3D Model
Compile Quad Mesh into Triangles
Triangle Mesh
Geometry Compression Density Level 1
Triangle Mesh Density Level 1
FURTHER PROCESSING
Geometry Compression Density Level 2
Triangle Mesh Density Level 2

Figure 11.8: Asset Build Process – Processing of high density meshes into lower density meshes

Asset Dependency Evaluation

In conjunction with dependencies of asset build nodes in the asset build process, dependencies also exist within the assets themselves. Figure 9.4 illustrates the asset dependency hierarchy within a scene composed of a landscape, building and trees. In actual practice where visualizations consist of a large amount of content the hierarchy can be both deep and wide. Automating the asset build process depends upon determining the execution order of the individual asset build node steps so a set of tasks can be scheduled and executed.

A build node can only be executed if all its child dependencies are satisfied. Theoretically, determining the correct order of build dependencies is a straightforward process since the entire hierarchy is represented as a directed acyclic graph (DAG). Considering a software build process, the dependency information is inherently specified within the source code written in text format. However, source assets are often binary files, in proprietary formats, from which parsing to determine dependencies is technically challenging.

For an asset build process, consisting of binary source assets, there are two differing approaches that can be used. Firstly, the dependency information and the execution order can be hard-coded in the build script. This static asset build process introduces a maintenance overhead since the build script requires updating every time a designer adds an asset. In software development this approach is identical to building source code using a batch file or traditional make file.

Secondly, and a more flexible approach, is to produce a framework where each asset build node is represented as a template specifying the expected source input type, a set of parameters to bind to the build script and the outputs that are produced. The asset build framework gathers the source assets and therefore the set of input types, matching those to the set of build asset node templates in order to generate the dependency graph for the assets that require building. This declarative approach to the asset build process minimizes the amount of maintenance changes that must occur when a new source asset is added or modified.

<u>**Consequences:**</u>

Benefits:

- ***Design Feedback Loop Efficiency:*** The asset pipeline provides a pattern for creating an optimized build framework. Automating this process increases the efficiency of the designer by reducing the iterative cycle of design and review.

- ***Incompatible Source and Final Asset Format:*** The asset pipeline process automates and resolves the steps required to transform source assets into their final asset format expected by the runtime
- ***Source Asset Dependencies:*** The process of determining the source asset dependency chain required for the asset build process is solved using a dependency strategy.

Liabilities:

- ***Asset Build Process Alternate Path:*** A source asset must undergo all asset conditioning steps prior to viewing in the runtime. There is not an advanced preview mechanism allowing the designer to bypass these steps to view assets in the runtime. Please see FAST PATH for a solution.
- ***Flexibility of Asset Build Node Configuration:*** Use of multiple asset formats within the asset build process results in an inflexibility in the configuration of the build steps. The build steps cannot be easily reordered and reused.

  Please see INTERMEDIATE ASSET FORMAT for a solution.

Examples:

1. Game Engine Asset Pipeline

A game engine is a reusable development environment originally established for use within the video game industry, however, recently found use in other domains such as real-time simulation visualizations. The core functionality of the game engine consists of reusable components to provide scene building, 3D graphics rendering, physics simulation, audio and behaviour scripting.

All source assets designed within a DCC tool for use within a game engine must undergo a sequence of steps to transform the asset into a final asset format for use in the runtime. An asset pipeline internal to the game engine is invoked to perform this processing. The following use case consists of the scenario of importing a three-dimensional tree modelled in the DCC tool 3ds Max for use in a real-time visualization, although in actual practice the scenario is applicable for any form of three-dimensional model. The game engine under consideration is the Unity game engine, a game engine popular amongst content creators owing to its modest licensing model, extensive functionality and wide variety of supported platforms.

The example workflow is illustrated in figure 11.9. The three-dimensional tree model is exported from the DCC tool 3ds Max into the Autodesk FBX standard format, step 1.

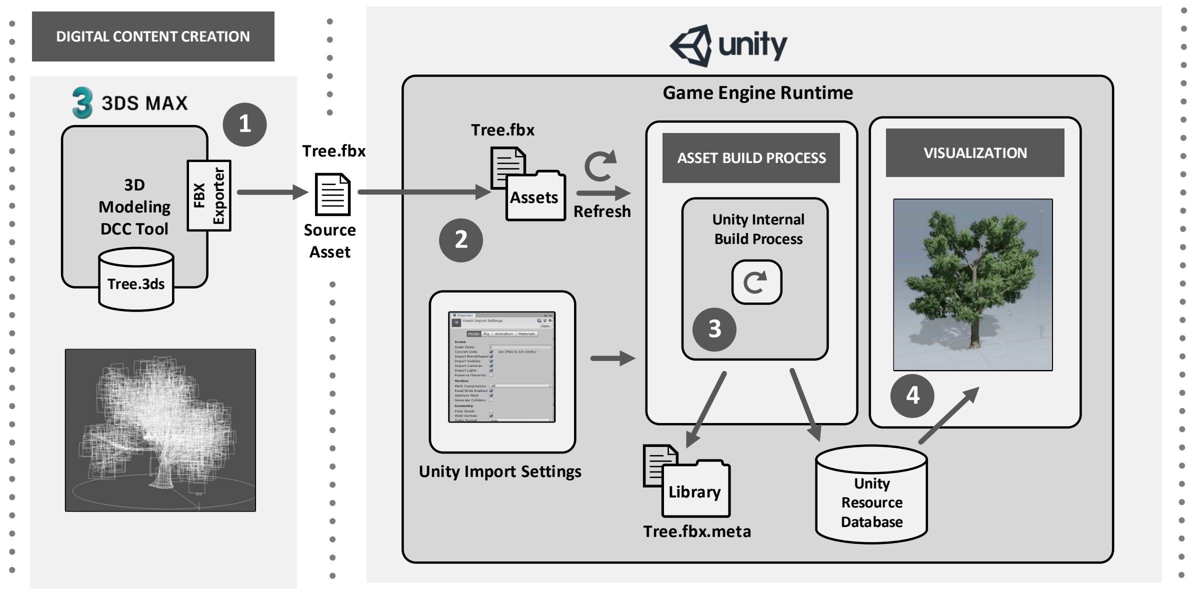
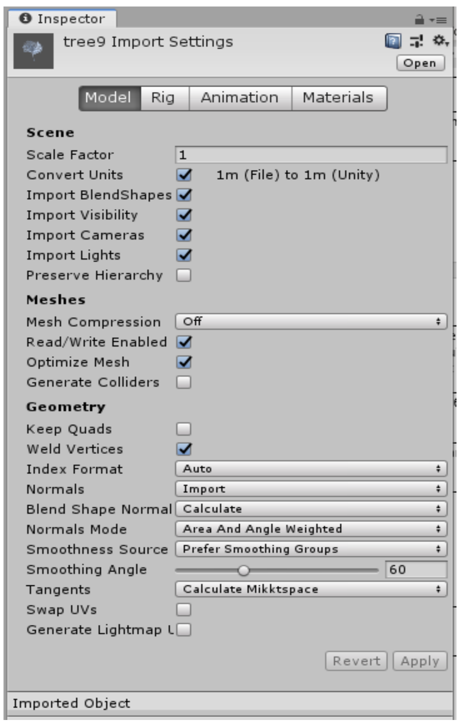


Figure 11.9: Unity Asset Pipeline – exporting a 3D model from a DCC tool into a game engine for real-time visualization

The designer copies the resulting asset into a specified "Assets" folder within the Unity project and specifies the import settings for the three-dimensional model, step 2. Import settings provide a user interface for the designer to manipulate the internal game engine asset pipeline by dictating the build steps performed and therefore alter the processing of the asset, figure 11.10.



Scene

Scale Factor	1
Convert Units	✓ 1m (File) to 1m (Unity)
Import BlendShapes	✓
Import Visibility	✓
Import Cameras	✓
Import Lights	✓
Preserve Hierarchy	☐

Meshes

Mesh Compression	Off
Read/Write Enabled	✓
Optimize Mesh	✓
Generate Colliders	☐

Geometry

Keep Quads	☐
Weld Vertices	✓
Index Format	Auto
Normals	Import
Blend Shape Normal	Calculate
Normals Mode	Area And Angle Weighted
Smoothness Source	Prefer Smoothing Groups
Smoothing Angle	60
Tangents	Calculate Mikktspace
Swap UVs	☐
Generate Lightmap U	☐

Figure 11.10: Unity asset import settings categorized into scene, meshes and geometry

Although the individual low-level asset build node implementation steps are proprietary in nature the high-level functionality is exposed and can be inferred from the import settings. The documented feature set is detailed in table 11.2.

Category	Property	Function
Meshes	Mesh Compression	The mesh compression ratio determines the file size of the mesh. Increasing the value reduces the file size, although reduces the precision of the mesh.
	Read/Write Enabled	Determines how the mesh integrates with the GPU pipeline. When read/write is enabled Unity can access the mesh data at runtime and manipulate the mesh characteristics.
	Optimize Mesh	When enabled Unity will optimize the order of the triangles in the mesh by reordering the vertices and indices for better GPU performance.
	Generate Colliders	On import automatically generate collision meshes for the environment geometry.
Geometry	Keep Quads	Prevents Unity from directly converting polygons that have four vertices to triangles. A copy of the original mesh is created prior to triangulation.
	Weld Vertices	Combine vertices that share the same position in space to optimize the vertex count on meshes.
	Index Format	Specify the size of the Mesh index buffer.
	Normals	Determine how and if the normal are calculated upon import.
	Normals Mode	Specify the algorithm and weighting for calculating normals.
	Smoothness Source	Set how to determine the smoothing behaviour.
	Smoothing Angle	Determine whether the vertices are split for hard edges.
	Tangents	Determine how and if the vertex tangents are calculated.
	Generate Lightmap UVs	Generates a second UV channel for Lightmapping.

Table 11.2: Unity asset import settings and functionality

When Unity reads and processes any files that are added to the Assets folder the game engine invokes the internal asset pipeline and converts the contents to internal runtime versions using the specified import settings. This is illustrated as step 3 of the workflow in figure 9.9. Unity uses final versions of the assets within the runtime and preserves the unmodified source asset files in the "Assets" folder. All runtime assets are persisted within the "Library" folder of the project. For example, the FBX format is convenient for a content designer to work with and can be saved into the "Assets" folder. Whereas, hardware such as GPUs cannot accept that format directly to render as geometry and therefore requires converting. The converted internal representation is held within the "Library" folder, similar to a cache folder. An import can sometime generate multiple final assets. For the situation of the 3D file, such as "tree.FBX", that also defines materials or embedded textures, the textures are extracted and represented within unity as separate assets.

Alongside the invocation of the asset pipeline, Unity assigns a unique ID to the asset. This ID is used internally by Unity to refer to the asset. The asset can therefore be renamed or moved without references to the asset breaking. A ".meta" file is created for each asset created in the "Assets" folder. Within the ".meta" file is the unique ID generated for the asset along with the import settings specified by the user. If such import settings are changed the asset is reimported and the final runtime asset will be updated in the "Library" folder of the project. Finally, on completion of the asset pipeline the asset is available for use in the runtime for visualization.

Although the example uses a game engine as the visualization runtime application, the asset pipeline pattern exists in many forms of visualization software where an asset has been designed using abstract high-level concepts and must be rendered using low-level runtime structures.

2. *Architectural Visualization (ArchViz)*

Architectural visualizations are used in architecture, engineering and construction (AEC) projects to communicate designs and building details to clients and other stakeholders such as onsite management and contractors. Architectural visualizations may take many forms: from static computer-generated images (CGIs), fixed path walkthroughs to truly interactive, real-time experiences. Contemporary visualizations are realistic in their appearance, sometimes photo-realistic, incorporating materials, lighting and landscaping. Figure 11.11 illustrates an example of a still render – the Faculty of Business and Law Building at the University of the West of England.

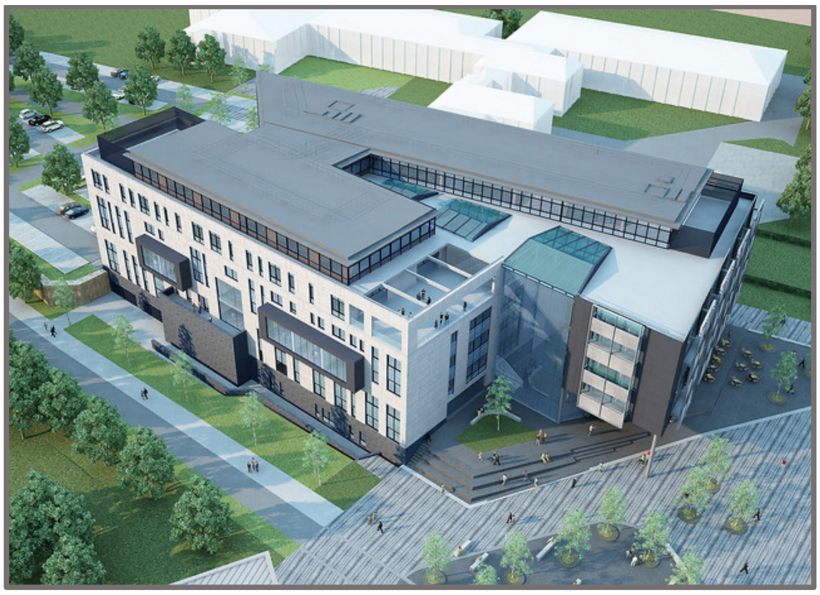


Figure 11.11: An example of a still render – the Faculty of Business and Law Building (UWE)

Building Information Modeling (BIM) is an emerging and prevalent technology in the AEC industry. BIM extends the CAD paradigm to incorporate both a three-dimensional representation of the building components and the data that describes how they behave. This key concept is known as parametric modelling. A parametric BIM object consists of its geometric definition along with associated semantic data and rules.

With an emphasis on collaborative working, the BIM repository is shared among all interested parties, designers, subcontractors and clients, for the lifecycle of the project. That is from the initial design phase through to construction, occupation and into the building maintenance and aftercare phase. In practise architects, structural engineers and other contributors from the AEC industry design and collate the building information using a single vendors BIM DCC tool such as Autodesk Revit. This provides a large, living central repository of information that can be used during the project life-cycle.

The BIM DCC tool Autodesk Revit does not provide a sophisticated mechanism for interactive real-time visualization of the building. Whereas, a game engine provides such a capability, incorporating real-time rendering of complex 3D geometry, a physics engine to handle any collision detection and an evolved mechanism to navigate 3D environments. The render system of a game engine is well suited to the field of visualization owing to its advanced graphical capabilities such as real-time global illumination and physically based shading. However, there exists a barrier between the production of such a visualization; the source proprietary format is not compatible with the expected format required for realization in the game engine. A constraint that can be resolved using an asset pipeline workflow.

The following scenario considers the asset pipeline necessary to produce an architectural 3D real-time visualization with the functionality to navigate a scene. The BIM design tool Autodesk Revit provides the source

BIM model along with the full 3D geometry and the game engine Unity provides the runtime environment. The asset pipeline workflow is illustrated in figure 11.12.

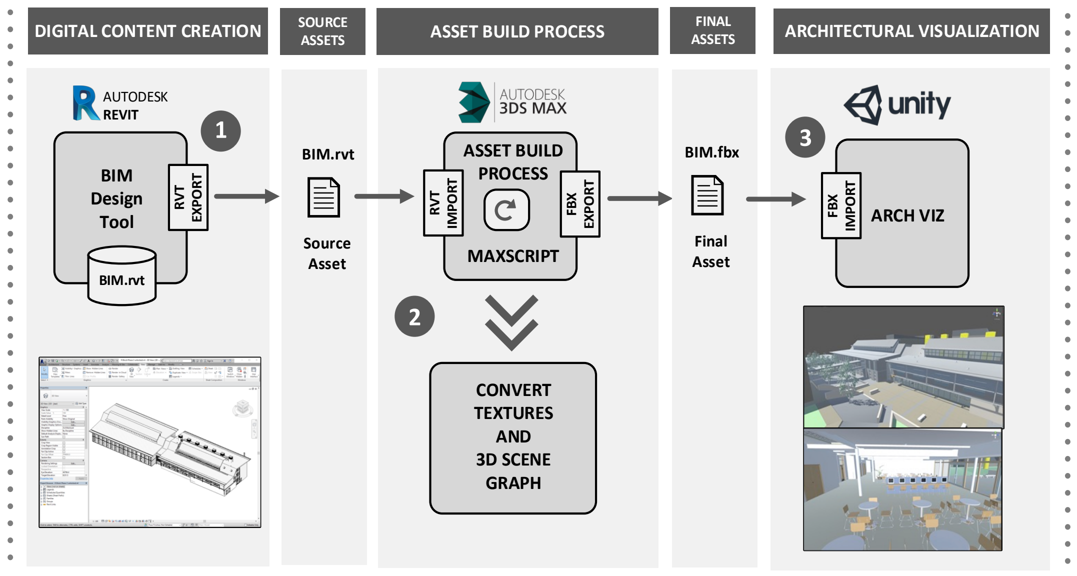


Figure 11.12: ArchViz Asset Pipeline – exporting a BIM model for real-time visualization in a game engine

The entire architectural BIM model is initially exported from the DCC tool Autodesk Revit in its native file format, RVT, step 1. This format encapsulates both the three-dimensional model and the semantic data providing information related to the model components. In its proprietary format the model cannot be directly imported into the game engine Unity for real-time visualization. The 3D model information, its geometry and textures, require conversion and extraction prior to the creation of the final assets.

The conduit to achieve the asset build process is the application Autodesk 3ds Max, step 2. Being of the same vendor as Autodesk Revit, 3ds Max understands the proprietary BIM format. Furthermore, Autodesk 3ds Max provides a built-in scripting language called MAXScript with the functionality to navigate and understand the BIM 3D scene graph and any assigned textures. For example, the 3D geometry of a building consists of walls, roof, glass and the materials of concrete, tiles and glass.

The asset build process consists of two steps. Initially, a custom MAXScript asset build node converts the Autodesk specific materials assigned to the BIM geometry into Standard materials compatible with the Unity game engine. That is, for each Autodesk material a Standard material is instantiated with the properties of colour, texture and height maps. Subsequently the converted scene graph, geometry and materials are exported from 3ds Max into the standard Autodesk FBX format, a structure the Unity game engine understands.

Finally in step 3 the fully resolved 3D model asset is imported into the Unity game engine for visualization using the runtime engine.

INTERMEDIATE ASSET FORMAT

Also Known As: Intermediate File Format

Context: ***Designers are using a variety of DCC tools to create a visualization executing on multiple runtime environments.***

Consider the development of a large and rich interactive real-time visualization requiring a variety of content types. A team of professional designers are often specialized in individual domains of content. Although they create assets in a similar area, such as three-dimensional geometric models, over their professional career designers have become specialists in a subset of DCC tools, some of which output to different and incompatible formats. The proliferation of different hardware devices, such as mobile and virtual reality, has resulted in the desire to view the real-time visualization on a variety of platforms each of which have their own set of hardware limitations. For multi-platform visualizations the final assets must be optimized to the runtime environment of each platform. That is each individual source asset type requires data conditioning to satisfy the runtime. The visualization designers submit the variety of assets to the asset pipeline and expect the final visualizations to be optimized for all target runtimes.

Forces:

Multiple source formats require identical asset conditioning

Many of the source assets, particularly those of a similar type, may be subject to the same set of asset build steps prior to integration within the runtime. Considering a visualization involving source assets based on textured three-dimensional models as shown in figure 11.13. Designers create three-dimensional model assets using two different DCC tools, each of which are exported to separate formats. The build steps for the three-dimensional models are algorithmically identical, consisting of geometry processing, however, since the formats are not identical, they must be individually tailored for each source format. An identical situation arises for the textures source assets.

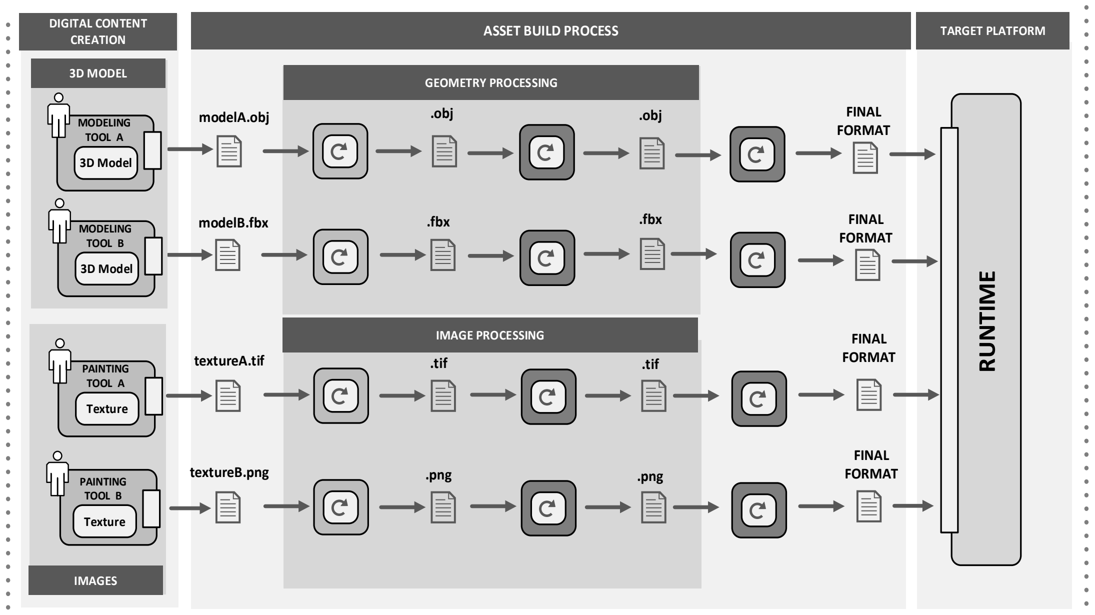


Figure 11.13: Asset Build Nodes performing algorithmically identical processing on different asset formats

Multiple runtimes each expecting specific final formats

The final step in the asset build process consists of transforming the processed source assets into their final format ready for use in the runtime. For each type of source asset format an individual build process must be created to perform the transformation.

Flexibility of Asset Build Node Configuration

Use of multiple asset formats within the asset build process results in an inflexibility in the configuration of the build steps. The build steps cannot be easily reordered and reused between subsequent projects.

<u>Problem:</u> ***How to decouple the source format from the final asset format to introduce flexibility in the asset pipeline?***
<u>Solution:</u> ***Use an intermediate asset format to decouple the source and final asset dependency.***

The source assets are initially converted from their native DCC tool proprietary format into an intermediate file format that is not tied to the final format expected by the runtime. It is the intermediate format that is used as the input to the asset build process tasked with generating the runtime ready final assets (see figure 11.14).

The requirements for the intermediate file format differ from those of the final format expected by the visualization runtime engine. The intermediate file format must be information rich and lossless where possible since it must contain all the necessary information for processing further in the pipeline. The asset build process transforms the intermediate assets as they travel through the pipeline, refining them, until they eventually are optimized to the final format expected by the runtime. The asset build process should not over-optimize and trim the intermediate assets too early in the pipeline since it is more efficient to discard data further down rather than having to re-export all the source assets as more information is required.

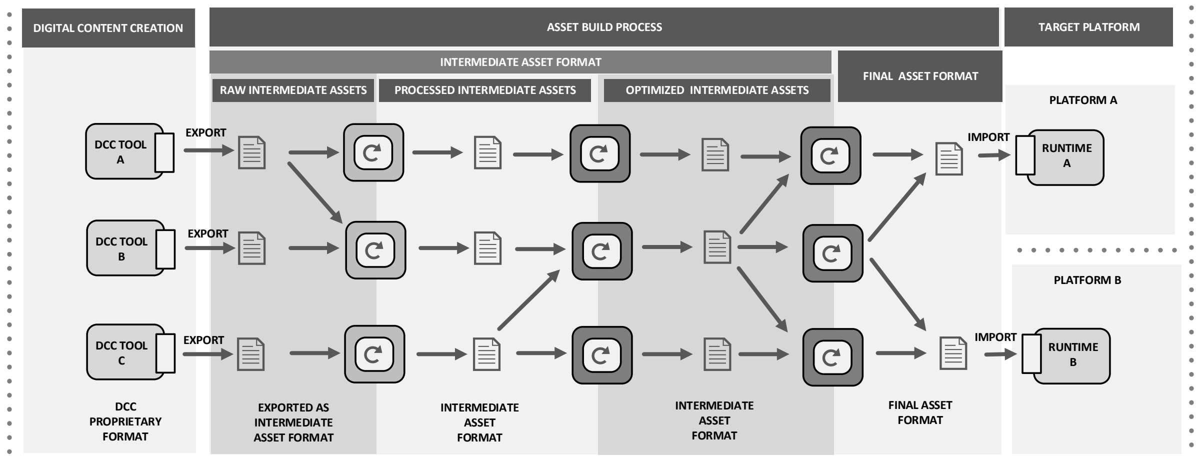


Figure 11.14: Use of Intermediate File Format within the Asset Build Process

Structurally the intermediate file format must be easy to read, parse and extend to fulfil any future requirements expected of the runtime. Given this requirement, the intermediate file format can be satisfied using a plain text file as a container, although any self-describing human-readable format such as XML would be suitable. XML is considered a data interchange format that can be parsed, queried and manipulated using a set of common programming languages. Furthermore, the hierarchical structure of the embedded document object model (DOM) aligns with the scene graph commonly representing a visualization. In this instance, an XML schema must be designed and implemented to contain a superset of all the data required by the asset build process.

In most cases a single intermediate file format should be used during all stages of the asset build process. The implementation of a single container format permits any common file parsing and writing functionality to be reused within the algorithms of individual asset build node steps. Subsequently, since all build node steps operate on a single intermediate format the input and output format of the build step will be identical with only the core transformations altering. A powerful result being an introduction of flexibility in the build process whereby asset build steps can be reused and reordered where necessary.

There are two possible integration points within the asset pipeline of where source assets can be converted into an intermediate format: either at the point of asset export from the DCC tool or at the first stage in the asset build process. Consider the scenario of integration within the asset export process, as in figure 11.15. An export plug-in must be developed for each distinct DCC tool used to transform the internal representation of the source assets into the intermediate format used by the build asset pipeline.

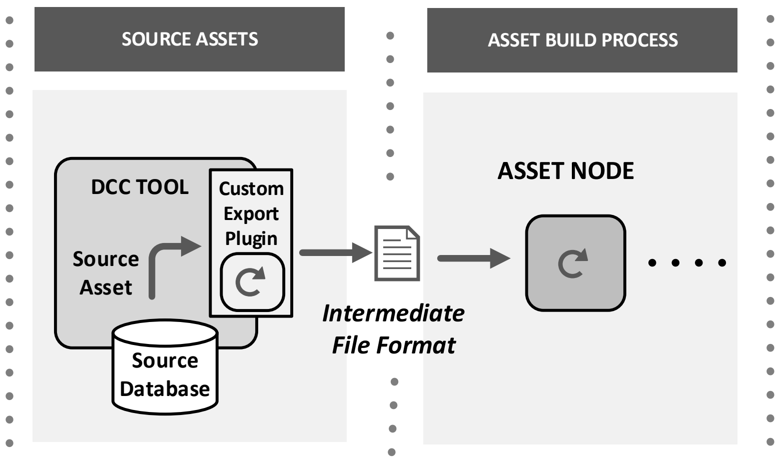


Figure 11.15: Conversion of Source Asset format to Intermediate Asset Format within an Exporter Plug-in

This can be a potentially daunting task considering each DCC tool exposes asset information through a separate API. The development does not stop once the plug-in is used in production since any changes to the DCC tool, for instance owing to an upgrade, may require additional maintenance development to integrate with a new API. A more favourable approach is to perform the conversion from source asset to intermediate asset format within the first stage of the asset build process, figure 11.16. This has the distinct advantage of loosely coupling the proprietary format of the DCC tool from the final runtime format. Subsequently, any changes to either the final runtime format or the DCC tool proprietary format requires only a change to the asset build node.

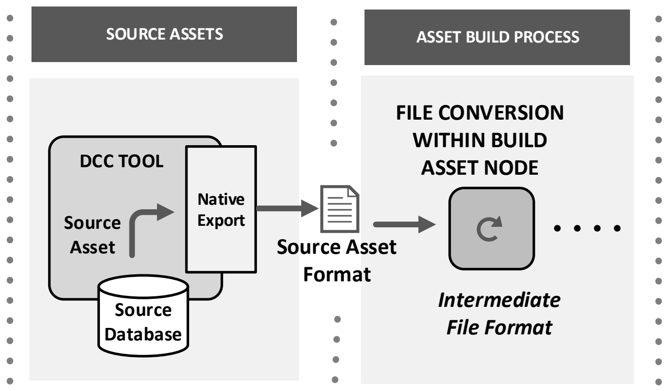


Figure 11.16: Conversion performed within the build asset node

For visualizations consisting of three-dimensional models an existing solution in the form of COLLADA (COLLABAborative Design Activity) may be used. Developed as a collaboration between Sony Computer Entertainment and the Khronos Group the format aims to standardize a common intermediate representation for use by DCC tool vendors so that individuals do not have to implement custom exporters and importers. The uptake of native COLLADA is growing with several DCC tool and runtime vendors supporting the format. Initially developed for interchange of assets between DCC tools, the format is sufficiently rich in functionality to allow it to be used directly in the asset build process. Furthermore, COLLADA uses XML as the container and is human readable allowing for human inspection during the build process.

Consequences:

Benefits:

- ***Multiple source formats require identical asset conditioning:*** Introduction of an intermediate file format allows common processing to be shared in the asset build process for source assets of a similar type.
- ***Multiple runtimes each expecting specific final formats:*** Conversion from an intermediate file format to a final runtime format is easier than converting multiple source formats to the final format.
- ***Flexibility of Asset Build Node Configuration:*** A single intermediate file format in the asset build process provides the facility to reorder asset build nodes since they all share the same input and output file types.

Liabilities:

- ***Intermediate File Format as a single container:*** Arriving at a single intermediate file format may be difficult since it must hold the data for all the transformations in the build asset pipeline. Use of COLLADA and other interchange formats provide a good starting point.

Examples:

1. Video Game Entertainment Industry

Consider the scenario of real-time interactive applications in the video game entertainment industry where the source assets are three-dimensional, and the runtime is a game or a related application. As consumers demand more immersive and detailed environments, the amount of content created for commercial game applications is doubling every year.

Teams of specialist designers and content creators use DCC tools to model the three-dimensional source assets using advanced concepts such as mathematical Bezier curves and Non-uniform Rational Basis Splines (NURBS). The DCC tools use proprietary formats, binary in nature, that do not encourage interoperability amongst DCC tool vendors. For example, the commercial three-dimensional DCC tool Autodesk 3ds Max and the free and open-source three-dimensional tool Blender persist their assets using native file formats.

In the video game industry, the interactive real-time visualization runtime is referred to as the game engine. Popular generic game engines include Unreal Engine and Unity. The game engine provides functional components to render three-dimensional graphics, simulate physics and collision detection, process and play sound and handle the underlying gameplay using a scripting language.

Producers within the video game industry compete for overall total market share, and to survive in this arena they must ensure their products operate on as many target runtime platforms as possible. The target platforms consist of disparate technical constraints and processing power. Ultimately the source assets must undergo a series of conversions, compilations and optimizations into their final form acceptable by the runtime. Common asset processing tasks that may be performed on the content include processing of textures, geometry and the environment.

Figure 11.17 illustrates the common video game production asset pipeline. Many source assets are created, often designed using a number of disparate DCC tools. The assets require conditioning into their final designated format suitable for multiple target runtimes. Given these constraints provision of an intermediate asset format decouples the source asset format from the format expected by the runtime introducing flexibility into the asset pipeline. Given a common intermediate format, build node steps may be shared amongst the build targets, and even reused across future projects.

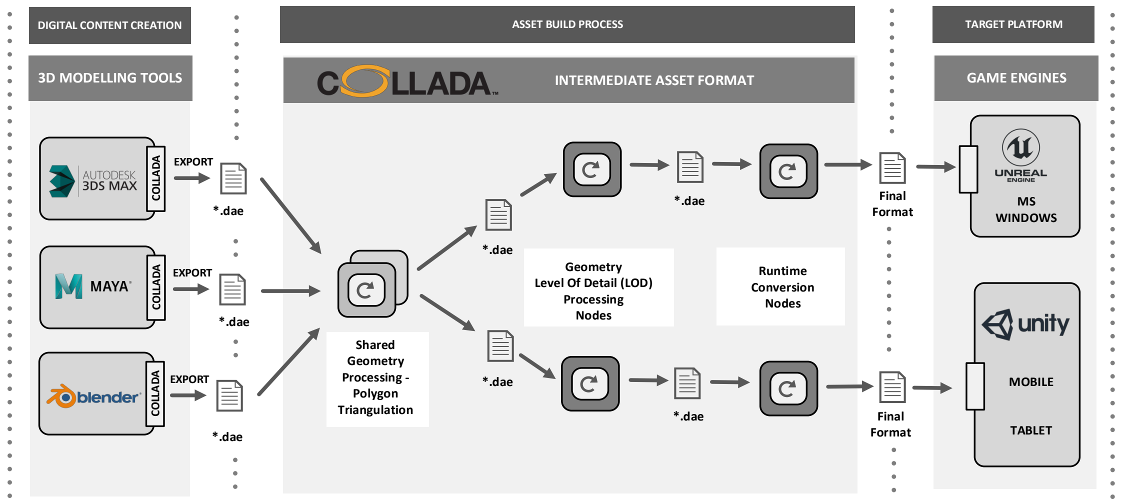


Figure 11.17: Video game asset pipeline illustrating use of Intermediate Asset Format

Although a variety of common formats have been adopted by DCC tool vendors, such as Wavefront OBJ and Autodesk FBX, one such format named COLLADA was designed with standardization of a common intermediate representation at its core. The COLLADA format encodes both the content and its semantics using the Extensible Markup Language (XML) container format. This intermediate format satisfies the requirements of being able to hold three-dimensional geometry and has the distinct advantage of allowing developers to extend its capability with any specific parameters required by the game engine. COLLADA has now reached a point of ubiquity, adopted by both a wide range of tool vendors and generic game engines. Applications such as the DCC modelling tools Autodesk 3ds Max, Autodesk Maya and Blender provide facilities to both import and export the intermediate COLLADA format. Likewise, the Unity game engine provides native support for COLLADA import.

FAST PATH

Also Known As: Alternate Build Path

Context: ***Designers need to view an isolated set of source assets in the runtime environment without performing a full build.***

Consider a visualization consisting of many source assets resulting in the asset build process being long running. An initial full build of the set of assets has been performed. A designer working on a subset of the source assets requires a mechanism to review their designs in the runtime in an efficient manner. The process of design is iterative and so this workflow will most likely occur multiple times. Any saving in time during a single such review cycle will accumulate during the creation of the content leading to further efficiency.

Forces:

Asset Build Process Alternate Path

A restriction of the asset pipeline is the entire set of asset build process steps must be performed on an asset that has changed prior to viewing within the runtime. Depending upon the number of asset build nodes and their level of processing the asset build process may be lengthy in operation. If the designer is required to initiate a build process for the changed asset whenever they wish to review their content in the runtime the design will progress slowly.

Trade-off between asset build time and accuracy of the result

The alternative path allows a designer to review a subset of their source assets within the runtime environment in a timely manner. Although the asset has not been subjected to the full build process and will therefore be compromised in terms of quality, during early design cycles this trade-off may be justified. After a number of design iteration cycles where the coarse design choices have been finalized the designer may wish to perform a full build and produce production quality assets.

Problem: ***How to provide a fast preview of the assets without having to wait for the final assets to be built.***
Solution: ***Asset build process supplemented with an alternate path allowing intermediate assets to be viewed directly in the runtime.***

The alternate path is known as the fast path. This provides the facility for an asset to be loaded into the runtime without having to go through the complete build process, figure 11.18. For the scenario of where a designer is working on a subset of the project assets and requires a preview under runtime conditions the designer instigates a minimal build. The asset build process creates an intermediate asset file that is loaded into the runtime using a custom importer. All other assets of the visualization may be loaded from a previous full build from the final build path.

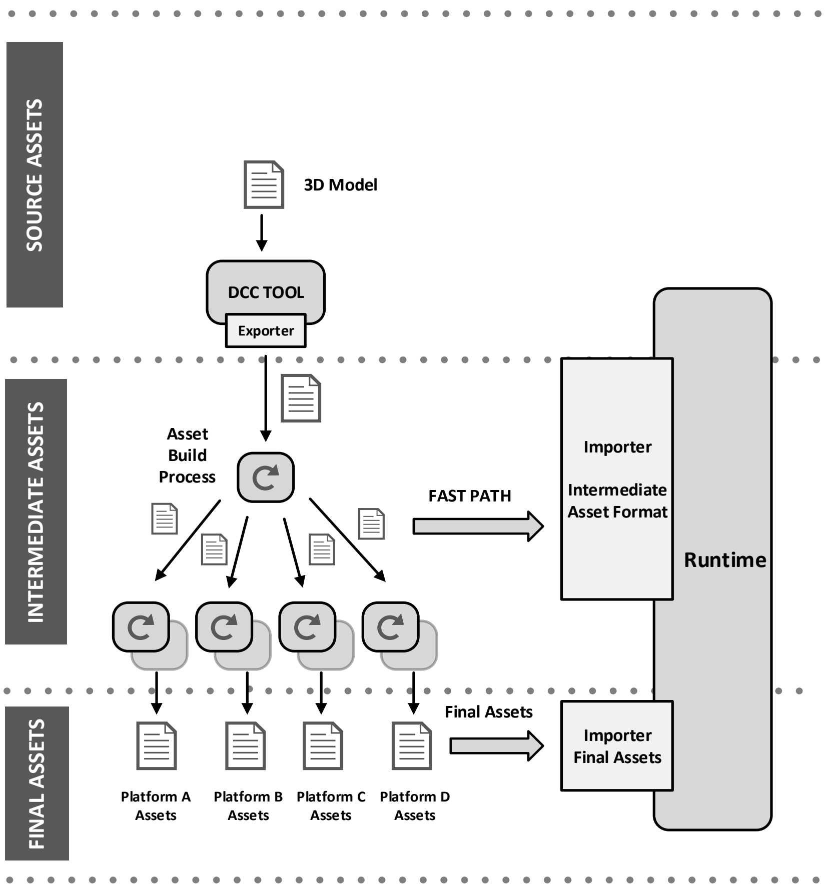


Figure 11.18: Integration of the Fast Path in the Asset Pipeline

Although, the intermediate asset may not be fully optimized and of production quality the fast path allows an artist to view their changes in the runtime within the context of the surrounding visualization. This mechanism increases artist efficiency by allowing faster iterations of the design process during early stages of the production workflow. Ultimately, when the source asset design has been finalized a complete asset build process may be undertaken to view the asset in its actual processed final state.

Consequences:

Benefits:

- ***Efficiency***
  The fast path provides a route for the source assets to be directly viewed within the runtime without having to pass through the complete asset build process. This scenario is particularly applicable for use cases where teams of designers work independently on areas of a runtime visualization.

Liabilities:

- ***Integration of Intermediate Asset Format within the Runtime:*** Support for the Fast Path requires additional development effort to extend the runtime importer functionality to read, parse and process the Intermediate File Format. The importer therefore is tightly coupled to the Intermediate File Format and any such changes in the format will have to be propagated to the importer.
- ***Asset preview quality compromised:*** Assets built using the alternate fast path are not subject to the complete asset build process. Therefore, the quality and accuracy of the final asset is not of production standard.

Examples:

1. Video Game Entertainment Industry

Contemporary video games are large and complex in nature. They are developed using teams consisting of designers, artists, programmers, level designers and sound engineers. For three-dimensional orientated video games, source assets are created using a variety of 3D modeling DCC tools. Designers require a mechanism to view and approve their designs within the context of the underlying video game engine.

The traditional design feedback loop is employed; that of incrementally making changes to a design and reviewing the changes within the video game. To achieve this the designer must build the source assets using the workflow of the asset pipeline. Where the video game consists of many source assets independently worked on by teams of designers this process may be potentially long running.

To achieve efficiency in the workflow, isolated assets may be subject to the alternate build path. In this instance a designer initiates a build using the fast path and previews the changes to the source asset within the framework of the video game. Although the quality of the asset is not of production quality, during the early stages of development where an asset is being continually refined the time saving in the rapid design-review cycle outweighs any loss in quality. At the point of where the asset is considered a candidate for approval in production the final full asset build pipeline is instigated.