Applying Designer-Led Game Engine Design to Visualization Software

Samuel J. Morris

Abstract—Visualizing a large dataset can be a difficult process, requiring niche software for the task, with a programmer generally tailoring a new program for the data to produce desired results. However, this could be mitigated if concepts from another area of software design was brought in; that area being game engines. These have grown in recent years from being engineered for a single product, into massive modular applications where the designer is primarily in control of what is created. By applying a game engine style asset pipeline, control is handed from the programmers into the hands of those with the data.

Index Terms—Open source software, Software design, Virtual reality, Visualization

1 Introduction

THE field of software development covers a broad range of purposes and design methodologies. While some design patterns and methods may be specific to one form of software, many could be studied and brought across to other forms of software to improve their design. In this instance, the designer-driven pipeline architecture of game engines has been studied and repurposed for a data visualization tool called *PhysVis*, requiring some degree of refactoring, but producing a more extendable and accessible tool in the process.

2 Evolution of Game Engine Software

When discussing game engines as software, it is pertinent to follow the evolution from humble beginnings to the modern era.

2.1 Early Video Game Production

Early video games were a largely electronic affair, using bespoke boards geared towards the games they were running. This evolved with the advent of the microprocessor, leading to platforms with the capability of running more than one title through an interchangeable storage medium. Titles produced during this time would not feature what is currently understood as an engine today, and would instead be largely bespoke programming for each release, possibly with some select components copied between games. This is usually thanks to the strict memory and processing budgets of these early systems, wherein highly optimized and bespoke solutions were the only real option for creating quality software. Generally, a game pipeline in this era was limited to some tools used internally by the production team, the results of which would be baked directly into the final game in a non-editable fashion. It's not that developers didn't want players to edit levels, titles such as *Excitebike* and *Lode Runner* featured an editor in-game, but without considerable effort it would simply not be viable on these early systems without compromising another aspect of the final product.

2.2 Invention of the 'Game Engine'

The term 'game engine' didn't widely exist before the release of 1993's *DOOM* from *id Software*. The lead programmer, John Carmack, designed their new title to decouple the relationship between game code, game behaviors and game assets. Primarily, this was for their own usage; allowing the design team at *id* to iterate upon gameplay and art without requiring a laborious rebuild of the game code. Carmack had been moving in this direction for some time: *Wolfenstein 3D* had some degree of separation between assets and code, and the team at *id* had been referring to their shared *Commander Keen* codebase as 'Keen engine' for a number of years, but *Doom* brought it to a new level. [1]

2.3 Platforms for Creation

After the success of *Doom* and the prevalence of user-authored modifications (mods), Carmack and the team at *id* went all-in for their next title *Quake*, designing the code to be even further removed from the assets as possible while improving the tools used to create them. The engine behind *Quake* was a major success – not just in the game that it initially powered, but in the wide range of player-created modifications it led to (some of which became major franchises in their own right, such as *Team Fortress*), and in licensing the technology to other companies (who would go on to create works such as *Half-Life* and *Call of Duty* using derivatives of the technology). Other developers would take note and follow the lead set by *Doom* and *Quake,* such as Ken Silverman's *Build Engine* which went on to power many games, including *Duke Nukem 3D*, *Shadow Warrior* and *Extreme Paintbrawl*. [2]

• *Samuel J Morris is with the University of the West of England, Bristol, UK. E-mail: samuel2.morris@live.uwe.ac.uk*

This continued into the new millennium, licensable and moddable game engines would become increasingly prevalent, including *Valve Software's Source Engine* (a *Quake Engine* derivative), [3] *Unreal Engine* from *Epic Games*, [4] and *idTech 3* from *id Software.* [5] With each major generation of engine, increasing degrees of control were passed from the programmers to the designers, with ingestion of content into the engine becoming known as the *'pipeline'.* [6]

The evolution of game engine technology has progressed a long way since the days of *Doom*. No longer is an engine expressly designed for an internal title first and allowed a wider release later, instead they are developed as productivity suites with the intention of allowing a wide range of people to develop on the platform. Modern engines such as *Unity 3D* and *Unreal 4* offer a degree of freedom that has not been seen in major engines before, with licensing terms that allow anyone from a major studio to a small independent team to work with the platform. The growth and prevalence of game engine technology has made game development accessible to audiences that would previously not have been able to take part in the field.

2.4 From Production Asset to Game: The Pipeline

The game development '*pipeline'* is the process of converting production assets into assets usable in the game engine. The concept of a pipeline is the main difference between a game and a game engine; an open and versatile approach to asset ingestion allows a codebase intended for one game to be quickly reused for another. Typically, the conversion process of a game asset pipeline is separate from the game engine itself, utilizing external tools and procedures to export game-ready materials. Modern platforms are becoming so versatile as to allow actual game code to be consumed by the pipeline (rather than, as was common, directly compiling it alongside the engine).

The process of converting a production asset into a game-ready one is generally performed to improve efficiency on the user's end, by providing compression or other engine-specific resources. [7] Games will quite frequently bundle large numbers asset files into packages, often with a proprietary format, in order to provide some measure of security against direct modification (usually the case in multiplayer titles), or to reduce the time taken to access files (operating-system dependent file operations tend to be slower than pulling data from a large file that is already open).

2.5 Modularity and Extendibility

As technology grows in scope, it too grows in complexity. This complexity is generally born of a combination of previous works, derived and adapted from what came before. As this practice becomes more commonplace, the reuse of technology becomes an intended effect of the creation of that technology, and its design is informed by this idea. W. Brian Arthur, in their book *The Nature of Technology* writes: "technology, once a means of production, is becoming a chemistry". [8] This is true too of software technologies, games included. Engines are being written in an ever-more modular fashion, allowing extension and remixing to meet new ends. The economical argument for this is obvious, that being it is cheaper to reuse components of software than reinvent the wheel each time, but the desire to improve the field as a whole by offering this modularity is also a motivational factor.

There are many methods of achieving modularity in software design, but an example employed by *Valve's Source Engine* adapts the *Bridge Pattern* via an abstract *DLL* interface *(Dynamic Link Library)* as detailed by the *'Gang of Four'* in their book *Design Patterns: Elements of Reusable Object-Oriented Software.* [9]

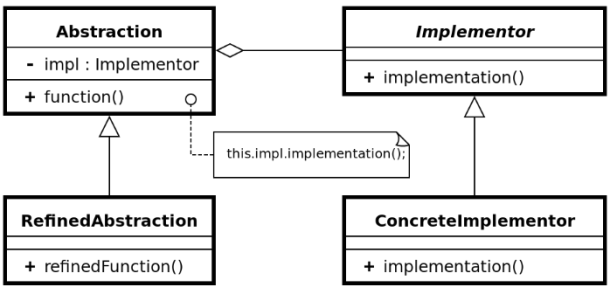

Fig 1 - UML diagram of the Bridge Pattern

This provides several benefits; primarily, developers at *Valve* are able to update implementations of various sections of the engine whilst leaving other areas untouched, but still able to make use of the newer modules. Second, this allows the many teams at the company to work on separate modules without stepping on each-others toes, as once an interface is authored, it will not be changed anywhere near as regularly as the implementation behind it. Finally, separating implementation into several dynamic link libraries allows the developers to reuse those modules for other software rather than copy or reimplement functionality found in those libraries (the most common usage of this is their filesystem interface library, which allows all their tools to access files in the same way).

3 PhysVis

3.1 First Phase

PhysVis is a visualization program originally intended to display the source-code of another program as a physics-based particle system, to work around the issue of software generally being unwieldy and intangible. [10] Source code is analyzed by an external tool and converted into a *JSON* dataset file by a tool called *MSE2JSON*. The software has been expanded to provide support for displaying molecular models also, supporting the *XYZ* format that is common in the field. Display data is provided by an accompanying JSON file. This work had been published in 2015.

While PhysVis was intended to be an open platform for authoring network-based visualizations, it wasn't geared towards offering wider support for more formats, or for being easily extended to add support manually. This was a key area of improvement highlighted in this phase of development, alongside wishing to author a true interface for editing the files ingested by the asset pipeline.

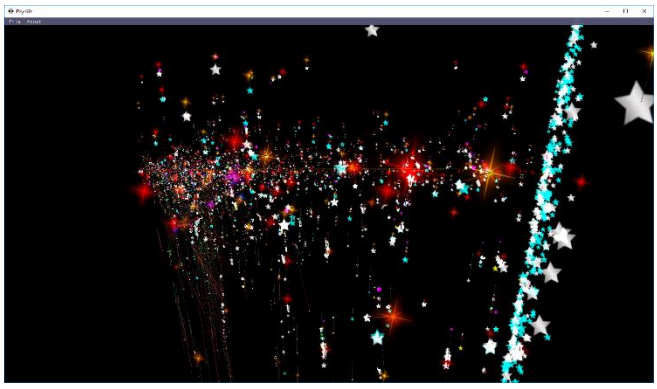

Fig 2 - Screenshot of PhysVis in action

3.2 Second Phase: Extending Modularity

The first step in improving the extendibility of PhysVis was allowing some method of adding support for additional formats without requiring the user to rebuild the main application. Upon application start, PhysVis scans a specific local directory for DLL files, mounting them and requesting a memory address for an exported function named *__GetImportModuleList*, which is uniquely defined in each DLL via user-friendly preprocessor macros. The generated function produces a list of importer modules which is returned to the main application to populate internal lists of currently supported file-extensions. Several extensions can be attributed to one importer, and several importers can be included in a single library, but it is considered good practice to keep different formats in different libraries without good reason.

The exported function makes use of a static function declared within each importer library by another macro, which is fed into a factory class. This avoids usage of C++ templates while offering similar dynamic functionality. This factory adapts the *Abstract Factory Pattern*, defined by the *'Gang of Four'* in their previously mentioned *Design Patterns* book. [9]

Fig 4 - Importer exposition macros and the code the preprocessor converts it into

Import module classes must be children of the *IImportModule* interface class and include the aforementioned macro which defines the creation function used by the factory objects. This method keeps all DLL exporting to a single function, resulting in easier cleanup on shutdown. The import process makes use of the *dirent* library in order to quickly scan the importer directory and find the needed files. [11]

Importer modules only require ingestion of a single library which defines the node graph classes used to build a scene within PhysVis, offering third-parties the opportunity to write their own importers for new formats without relying upon systems that may not have been intended for them. While implementation was not attempted, it is theoretically possible that any basic network dataset could be imported should an appropriate importer module be authored. PhysVis currently comes with two importer libraries, supporting the JSON and XYZ formats of the earlier hard-coded importer system.

3.3 Second Phase: Refining the Pipeline

The overall objective of the second phase of PhysVis development was to improve accessibility across the application, both for those creating visualizations and for those viewing them. To this end, a wide number of changes and additions were made beyond the modular loader system.

Taking direct inspiration from widely-known game engine pipelines has been key to improving the accessibility to PhysVis, and such an improved approach to asset management was implemented, involving a new directory structure following the style found in *Valve's Source Engine* titles.

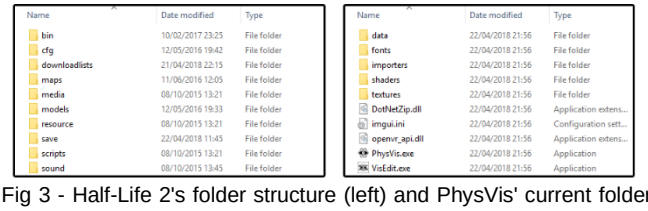

Fig 3 - Half-Life 2's folder structure (left) and PhysVis' current folder structure (right)

Further to this, a basic GUI (Graphical User-Interface) has been implemented, offering file loading from an explorer-style window as opposed to the previous iteration's entirely command-line based loading method. This interface has room for expansion, thanks to the use of the *DearImgui* library. [12]

PhysVis is built using the *Microsoft DirectX* graphics platform, and requires texture image data to be saved and loaded from the *DDS* (*DirectDraw Surface)* format. DDS is a relatively versatile format, offering S3 texture compression, storage of cubemap textures, and support for texture arrays; however, it is not a common format outside of 3D rendering, and isn't natively supported by most graphics applications. As such, game engine pipelines will use tools to convert common image formats to the more obscure one, occasionally hiding the underlying format

within another to allow for engine-specific data to be included, such as *VTF (Valve Texture Format)* utilized by *Source*. Instead of requiring conversion to DDS, it was instead decided that extended image support should be added to the application, allowing for the use of popular formats such as *JPEG*, *PNG*, *GIF* and *PSD*, among others. Support for this was achieved with use of the *stb_image* library by Sean Barrett, which is capable of decoding various image formats into raw pixel data that can then be loaded into a DirectX surface. [13] Assets found in the main texture directory are loaded and cached on renderer initialization to reduce overhead when opening a scene and attempting to fetch the needed assets.

Finally, an aforementioned feature of many engines is to bundle their many asset files into a smaller set of package files. This has been a common trend since *Doom*, which used the *WAD* (standing for *Where's All the Data?)* format; the *Build Engine* utilized the *GRP* format (short for *Group)*, *idTech 3* used *PK3* (meaning *Package format 3*, although internally this format is just a common *ZIP* file), and *Source* uses *VPK (Valve Package)* which is a fairly advanced format, with the ability to sideload newer packages atop of older ones for easier game patching. Something *Source* is also capable of is bundling assets inside its level files, allowing modders to package a map (in *BSP* format, standing for *Binary Space Parition)* and all required custom assets into a single file. This approach was selected as how PhysVis should handle a package format implementation, as it would allow a visualisation and all its resources to be bundled together and easily opened, rather than requiring the user place the package in a specific directory and modify loader scripts as many engine packages do.

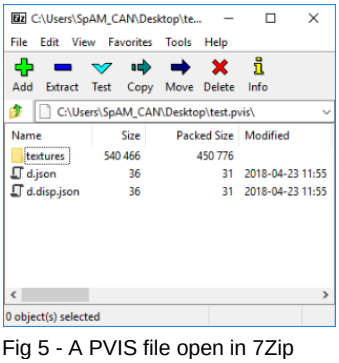

Fig 5 - A PVIS file open in 7Zip

Implementation of packages would make use of the widely ZIP format to avoid needlessly writing a custom binary format and parser library, with the custom extension *PVIS* for easier identification. The internal structure of the file is relatively simple, requiring the dataset and display info file be specifically named as *'d'* and *'d.disp'* respectively, and textures be included in either the root or a textures directory. Support for loading these files has been achieved through the use of the *PhysicsFS* library by Ryan C. Gordon, [14] which can read a wide range of package formats from many game engines alongside ZIP and related formats such as 7Z and ISO. This leads to a curious situation wherein asset package formats from old games could be used as a valid PhysVis file (this has been tested with the aforementioned *Build Engine GRP* format). Current implementation requires an importer library have package reading functionality specifically authored for it, but future development could unify file loading functionality across the application to require all filesystem calls go via PhysicsFS, as the library supports priority-based search-paths, offering transparent layering of data from different sources. For the time being, only JSON format datasets are natively supported within packages, due to an incompatibility between the stl datastream type used in the standard XYZ loader and the stl datastream type offered by PhysicsFS.

An idea implemented late in development is the ability to not only load visualization assets from a package, but program code also. Should a valid importer DLL be placed in a package and named *"importer.dll"*, it will give the user the option to use the bundled library instead of one already loaded by PhysVis. Using standard Windows functionality, this would require extracting the DLL to a temporary file, which isn't an ideal solution as it could leave junk data on the user's system. Instead, PhysVis loads DLL code directly from memory, achieved using a process known as *Reflective PE Loading* via the *MemoryModule* library by Joachim Bauch. [15] The resulting system allows the importer to be loaded ad-hoc from a package with no messy temporary files. There are security implications to be considered, however. The loading of program code from an untrusted source is ill advised, and Reflective PE Loading has been used in the past to slip malicious code past virus scanners. [16] In order to mitigate this somewhat, the user is informed of the risk before being asked to progress with the loading of any bundled DLLs, and have the opportunity to instead use the loaders already present in the directory structure of PhysVis.

Throughout development of the updated pipeline, the application underwent a number of changes to the core application loop, a major update to the Oculus VR library, and improved command like parsing, among smaller fixes and feature improvements.

3.5 A Future Third Phase

A third phase of development on PhysVis should focus on continued improvements to the accessibility of the software, improving import procedures for assets and adding a lower-level integration of PhysicsFS to enable support for reading other assets, such as shaders, from packages. A scriptable import system should be considered too, offering a safer method of extending format support, avoiding the security issues of loading arbitrary Windows DLL code and instead running in a stricter, sandboxed environment.

4 VisEdit

4.1 Overview

Accompanying the second phase development of PhysVis was the development of an editor tool, known as *VisEdit*, to enable easier editing of datasets and their accompanying display files, as well as bundle visualizations into the *PVIS* package format.

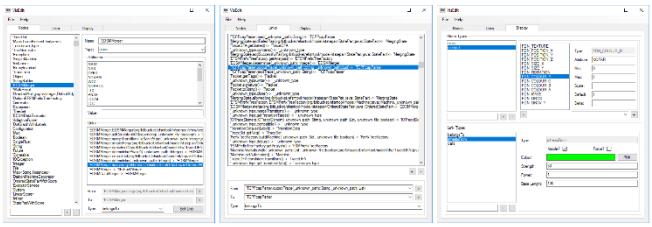

Fig 6 - VisEdit in its three states; Node View (left), Link View (middle), and Display Types View (right)

VisEdit has been authored in C# using WinForms and supports the creation, modification and saving of JSON datasets – XYZ is not currently supported, and no modular load/save interface is supported. The interface is divided into three separate editors for nodes, links and display types, crossing over and affecting each other in places where it was logical to do so, such as the list of display types populating type selection boxes for nodes and links. No preview of the visualization is currently offered, requiring the user to save and open their file in the main application each time they wish to view the results of a change.

PhysVis compatible package files can be exported from the application via the File menu. This process will save the currently opened dataset and display file to a desired package and offer the option of adding image files, an importer DLL or other miscellaneous assets.

4.2 Comparison to Game Engine Software

The VisEdit-PhysVis pairing is modeled closer on the previous generation of game engines, rather than the likes of *Unity* or *Unreal 4* which are more akin to a full *IDE (Integrated Development Environment)* than an engine-editor pairing. While VisEdit does perform the core task asked of it, that being allowing easier modification of datasets and display files, it is a fairly manual process. Even relatively old development tools, such as the *Hammer World Editor* that comes with the *Source Engine*, has various automated processes to make working with it more efficient, such as automatically seeking out and previewing resources needed by the current open file, which VisEdit lacks. VisEdit also does not encompass the functionality of the MSE2JSON tool used to process source code analysis into a structure usable by PhysVis, instead requiring the user to have already prepared such a file externally or entering the data into the editor by hand, which would be cumbersome and slow in the current UI – though arguably would be slower to create in a text editor by hand, and would require something closer to programming expertise.

The lack of a visual component is a fairly large flaw, but not an unexpected one, bearing in mind that the intention of PhysVis in the beginning was to find a way to get around the issues of visualizing such datasets in a traditional manner. [10] However, the biggest missing feature in comparison to many engine pipelines is a measure of automated interoperability to between the editor and engine applications. In the case of *Source*, *Hammer* can send the level load command directly to the engine upon compilation, and the engine can send entity state information back to *Hammer* to use as the default state of said entity in a level. This sort of direct link between the two applications gives the feeling that the two are part of the same family of applications, rather than entirely separate programs.

4.3 Evaluating VisEdit

User surveying has not been conducted for VisEdit as the software has only recently reached the level of maturity that would be required, however the methodologies for testing the tool have been considered for future use.

Ideally, the VisEdit-PhysVis pairing would be evaluated in turn by a number of users with a range of expertise and fields, wishing to produce a variety of visualizations. The users' session will be recorded with the user asked to vocalize their thoughts while using the software, and a questionnaire will be produced to be completed afterwards. This follows software testing practices set out by *Adobe.* [17]

One group will be asked to produce a visualization using the VisEdit software, and one group will be asked to produce a visualization without it (or potentially the same set of users tested twice to roughly emulate the two groups). Data from each group, such as the complexity of the final produced visualizations and the time taken to produce them, will be compared between the groups. The ideal result would be seeing an improvement through the use of VisEdit.

The questionnaire will ask users on which parts of the application they found frustrating, and offer them the chance to give suggestions on how to improve the interface of the editor. Session recordings will be studied afterwards to find possible subconscious sticking points in the interface, with the results combining to form a report detailing the best ways to progress with the development of VisEdit from the users' perspective.

4.4 Future Development

The software behind VisEdit could be improved considerably; the majority of the application has been quickly authored in a single class, with a large amount of duplicate code for certain actions. To better link with the newly developed modularity of the main application, support for importer/exporter modules should be integrated, rather than the current requirement of the use of JSON datasets. Ideally, these would be the same modules used by PhysVis itself, though there will likely be some C++/C# interoperability issues to be worked out. Furthermore, a stronger link between the two applications would also be desired, such as adding a *'Send to PhysVis'* button which would immediately load the visualization in PhysVis for viewing without needing to go through the process of manually saving and loading.

5 Conclusion

The overall goal was to adapt the PhysVis software to use a game engine style pipeline to improve accessibility for people wishing to create particle visualizations, rather than programmers with the ability to author their own software. Overall, this has been a success to some degree, with the implementation of modular importers, packaged visualizations, a GUI, and an editor application. The VisEdit utility allows users to build visualizations without manually editing the JSON datasets, attempting to emulate the editor-engine combination seen in many open-to-modification game engines, such as the *Hammer Editor-Source* pairing.

Methodolgy has been laid out for future user-testing of the application, but has not yet been conducted due to VisEdit only recently reaching a level of feature-completeness required to be appropriately evaulated.

While there is still further work to be done on implementing a fully matured game engine style pipeline and workflow, the second phase of development on PhysVis shows how taking lessons from other areas of software can improve a tool, offering more freedom to the end-user. Further development could aim the editor application even more toward those who want to author a visualization, regardless of programming ability, by enabling previews, interface changes based on user-testing and tigher integration with PhysVis itself.

Acknowledgment

The author wishes to thank Simon Scarle and Neil Walkinshaw for their work on the original PhysVis application. [10]

References

[1] H. Lowood, "Game Engines and Game History," *History of Games International Conference Proceedings,* 2014.

[2] K. Silverman, "Build Engine," 3D Realms, Garland, US, 1995.

[3] Valve Corporation, "Source Engine," Valve Corporation, Bellevue, Washington, 2004.

[4] Epic Games, "Unreal 4," Epic Games, Cary, North Carolina, US, 2014.

[5] id Software, "idTech 3," id Software, Richardson, US, 1999.

[6] B. Carter, The Game Asset Pipeline (Game Development Series), Massachusetts, United States: Charles River Media, 2004.

[7] J. v. Dongen, "The Game Asset Pipeline," Utrecht School of the Arts, Utrecht, Netherlands, 2007.

[8] W. B. Arthur, The Nature of Technology: What it Is and How it Evolves, London, UK: Penguin, 2009.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Boston, US: Addison-Wesley, 1994.

[10] S. Scarle and N. Walkinshaw, "Visualising software as a particle system," in *VISSOFT*, Bremen, Germany, 2015.

[11] T. Ronkko, "Dirent for Windows," 2006.

[12] O. Cornut, "DearIMGUI," 2014.

[13] S. Barrett, "stb_image library," 2006.

[14] R. C. Gordon, "PhysicsFS," 2001.

[15] J. Bauch, "MemoryModule," 2004.

[16] Deep Instinct Research Team, "Certificate Bypass: Hiding and Executing Malware from a Digitally Signed Executable," in *BlackHat USA*, San Francisco, USA, 2016.

[17] N. Babich, "Adobe Blog: The Top 5 User Testing Methods," Adobe, 23 February 2017. [Online]. Available: https://theblog.adobe.com/the-top-5-user-testing-methods/. [Accessed 23 April 2018].

[18] P. Ciszek, "3D Production Pipeline in Game Development," University of Jyväskylä, Jyväskylä, Finland , 2012.

[19] V. Broeren, "Producing with or without Game Engines," Utrecht University, Utrecht, Netherlands, 2014.

[20] A. Thomason, "Tools and middleware," Goldsmiths University, London, UK, 2015.

[21] "Designing Modular Software: A Case Study in Introductory Statistics," Iowa State University, Ames, US, 2016.

[22] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software,* vol. 23, no. 1, pp. 51-60, 2006.

[23] B.-J. Breitkreutz, C. Stark and M. Tyers, "Osprey: a network visualization system," *Genome Biology,* vol. 4, no. 3, 2003.

[24] D. Keim, H. Qu and K.-L. Ma, "Big-Data Visualization," *IEEE Computer Grtaphics and Applications,* vol. 33, no. 4, pp. 20 - 21, 2013.

[25] M. Krzywinski, "Hive Plots - Linear Layout for Network Visualization - Visually Interpreting Network Structure and Content Made Possible," 2011. [Online]. Available: http://www.hiveplot.com/. [Accessed 21 January 2018].

[26] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3 ed., Addison-Wesley Professional, 2003.

[27] R. Wallace, "Modding: Amateur Authorship and How the Video Game Industry is Actually Getting It Right," *BYU Law Review,* no. 1, p. Article 7, 2014.

[28] E. Hirvonen, "Improving the game with user generated content : an overview of Skyrim mod users," University of Jyväskylä, Jyväskylä, Finland, 2017.

[29] K. A. Moody, "Modders : changing the game through user-generated content and online communities," University of Iowa, Iowa City, US, 2014.

[30] T. Yeoh, M. Toh, S. Williams and R. Gasparini, "From Pacman to Pool: Mapping the evolution of User Generated Content," RMIT University, 2011. [Online]. Available: https://mediaindustries1.wordpress.com/. [Accessed 21 April 2018].

[31] W. J. Au, "Triumph of the mod," Salon.com, 16 April 2002. [Online]. Available: https://www.salon.com/2002/04/16/modding/. [Accessed 21 April 2018].

[32] W. Scacchi, "Modding as an Open Source Approach to

Extending Computer Game Systems," *Open Source Systems: Grounding Research,* pp. 62-74, 2011.

[33] T. Sihvonen, Players Unleashed!: Modding The Sims and the Culture of Gaming, Amsterdam, Netherlands: Amsterdam University Press, 2011.

[34] W. Scacchi, "Computer game mods, modders, modding and the mod scene," *First Monday,* vol. 15, no. 5, 2010.

[35] S. Agarwal and P. Seetharaman, "Understanding Game Modding through Phases of Mod Development," in *ICEIS 2015 Proceedings of the 17th International Conference on Enterprise Information Systems*, Barcelona, Spain, 2015.

[36] R. Hong, "Game Modding, Prosumerism and Neoliberal Labor Practices," *International Journal of Communication,* vol. 7, pp. 984-1002, 2013.

[37] M. Kolodnytsky and A. Kovalchuk, "Interactive software tool for data visualisation," in *International Workshop on Intelligent Data Acquisition and Advanced Computing Systems*, 2001.

[38] K. Gallagher, A. Hatch and M. Munro, "Software Architecture Visualization: An Evaluation Framework and Its Application," *IEEE Transactions on Software Engineering,* vol. 34, no. 2, pp. 260-270, 2008.

[39] R. Wettel, M. Lanza and R. Robbes, "Software Systems as Cities: A Controlled Experiment," *ICSE '11 Proceedings of the 33rd International Conference on Software Engineering*, pp. 551-560, 2011.

[40] Z. Sevarac, J. Tulach and A. Epple, "Patterns for Modularity II: Revenge Of the patterns," in *Oeracle OpenWorld*, San Francisco, US, 2011.

[41] H. Rajan, S. M. Kautz and W. Rowcliffe, "Concurrency by Modularity: Design Patterns, a Case in Point," in *Onward! Conference*, Reno, US, 2010.

[42] D. P. Simon, "The Art of Guerrilla Usability Testing," UX Booth, 25 July 2017. [Online]. Available: http://www.uxbooth.com/articles/the-art-of-guerrilla-usability-testing/. [Accessed 23 April 2018].

[43] J. Gregory, Game Engine Architecture, A K Peters/CRC Press, 2009.

[44] Valve Corporation, "Half-Life 2," Valve Corporation (digital), Sierra (retail), Bellevue, Washington, 2004.

[45] D. Griliopoulous, "A History of idTech," IGN, 28 April 2011. [Online]. Available: http://uk.ign.com/articles/2011/04/28/a-history-of-id-tech. [Accessed 25 April 2018].

[46] Unity Technologies, "Unity 3D Engine," Unity Technologies, San Francisco, US, 2005.