Global Software Engineering for Game Development

How the unique challenges of game development impact global engineering techniques.

Robert Grier
College of Computing and Digital Media
DePaul University
Chicago, Illinois, U.S.
rgrier@depaul.edu

ABSTRACT

Video game technology requires the combination of diverse creative expertise with strong software engineering practices. In a globally distributed software engineering environment, the interplay between subjective design and pragmatic engineering must be carefully managed to create a symbiotic relationship between teams who are potentially distributed in terms of geography, culture, and skillset. In this paper, I outline how global development impacts the software architecture, version control, and validation techniques used in game development to further understand the industry's deviations from standard software development [1]. In addition, I argue for improvements and considerations for global engineering in general by juxtaposing these game development practices with more widespread techniques.

At their core, video games are software products. Content aside, games are not so different from other consumer desktop applications. However, the content served by this software core has a dramatic impact on its design, performance requirements, and definition of quality. To be valuable as a product, the game must be fun to play. The way to accomplish a fun design is subjective and requires the genius of artists, audio specialists, writers, and gameplay designers.

The interaction between the software team developing the core application and the creative teams developing the content is at the core of what makes game development different from traditional software development. In a globally distributed environment, these teams are often distributed geographically or separated by organizational structure. Thus, the challenges of developing a creatively driven software product are compounded by the challenges of global engineering.

KEYWORDS

Global software engineering, globally distributed software development, games, software architecture, source control, playtesting

[1] Standard software development refers to pure software applications with traditional functional requirements.

1 Engineering Challenges in Game Development

The most common problem in the interaction between creative teams and engineering teams in game development is the issue of fluid and subjective requirements coming from the creative teams. Kasurinen et al [2] note that "changes to the product design during the development phase" is a key point in their review of what concerns game programmers. Wang and Nordmark [6] open their study of game architecture by stating that "there are no real functional requirements" for games. In other words, the testable, objective metrics that guide traditional software products are mostly absent. Instead, developers must accommodate a creative vision and the subjective experience of the end user.

The creative vision component is further complicated by the fact that there is a diverse set of creative teams, including the visual, audio, experiential, and narrative arts. These specialties may even be globally distributed and externally sourced to ensure high quality across all domains [1]. With this comes a diversity of culture and expectations for how the software might serve their creative process. Furthermore, these inputs must be balanced with the performance demands of a real-time application.

In their review of research on game development architecture, Mizutani et al [3] illustrate that there are "no standard or completely generic implementations" for games because of the fluid nature of the requirements. Compared to other domains where a particular algorithm might have a well-defined, optimal definition, elements of a game's behavior might need to be finely tuned to accomplish the desired effect.

2 Architecture for Creative Applications

The challenges laid out previously present an opportunity for strong software architecture. However, the research that has been done shows a lack of formality around software architecture in games, with some evidence showing that programmers put less emphasis on architecture "because there is such a high probability that parts of their code will be thrown away" [4]. This is anecdotal, but it helps to illustrate the extreme nature of fluid requirements.

In response to the idea that there are no functional requirements at play here, I propose that game programmers should aim to treat subjectivity as a functional requirement itself. With this as a guiding principle, the accommodation of design changes should be the core function of game software. The creative team's desire to change elements of the product should be seen as valid and as the defining requirement of the software. Game software that can be easily changed and iterated on will be more likely to produce a stronger product.

2.1 Data Driven Architecture

One specific technique that plays into this idea and is cited by the research is the **Data Driven Architecture**. Here, content can be represented as data and fed into a core executable to change the software's appearance and behavior. This architecture "offers a practical way of changing software behavior without modifying the codebase" [3], and it is analogous to patterns like **Configuration as Code** in other domains. The core executable here must be generalized in a sense, and more general code is usually seen as preferable to implementation specific code. However, **Data Driven Architecture** is generalized in a more targeted way because it must balance the potential arena of creativity being fed into the application with the performance demands of the real-time simulation in which that arena is presented. Like many other patterns, this architecture is the distillation of an abstract principle into a tangible, specialized solution.

This architectural pattern has some implications for global development. Chiefly, a purely data driven development tool can be distributed as a single executable to distributed teams. It can also enable the protection of proprietary source code from any number of external teams because they only needed to be provided with the pre-built application.

2.2 Product Oriented Architecture

The dynamism of data-driven design can be taken a step further with **Product Oriented Architecture**. In this paradigm, I propose that distributed game development studios could improve the architecture of creative applications by productizing the software components. This means reframing the software as a product or tool that is self-contained and can be completely driven by a creative team to create a video game. The product definition can be used internally by organizations to define a common language between the software engineering team and the creative teams. In addition, clear boundaries of the software's capabilities and possible extensions can be defined.

By treating the software as a product, teams can transition internal software tools into external tools. Internal interfaces and support mechanisms translate to external communication with distributed teams and even third-party customers. Support for external teams likewise benefits internal communications, as all creative users are interacting with the software as a single product.

Toftedahl and Engström's [5] taxonomy of game engines and tools tries to formalize the types of software products that exist in a modern game development pipeline. The breakdown of the pipeline into specific tools highlights the opportunity to create more standalone products out of the menagerie of software maintained in the support of creative applications. An example of this strategy is exemplified by commercial game engines, which treat the pipeline as an all-in-one tool that can be sold to other game developers. Often, engine developers are clients of their own tools and are therefore benefiting internally from treating their software as a product.

3 Version Control for Global Game Development

Using the correct version control solution is essential for effective distributed development. In game development, centralized version control systems seem to be strongly preferred over popular distributed solutions such as **Git**. Specifically, the **Perforce Helix Core** solution (commonly just referred to as **Perforce**) targets game developers with its features and is the most referenced centralized version control in the industry [8]. In this section, I will analyze the technical and cultural reasons why this is the case. The scholarly literature on version control for global game development is extremely limited beyond mentioning the fact that it is used, so documentation from the various projects, providers, and experts is used.

It is helpful to understand the differences between a centralized and distributed system, and the specific differences between the likely candidates: **Perforce** and **Git**. **Perforce** provides a single copy of a repository's history and configuration on the server, and users simply retrieve a snapshot of the version they want. **Git**, on the other hand, distributes all the information about a repository to each cloner, so it is replicated on each machine. This makes **Git** more flexible, but it might not scale as well as **Perforce** when large binary files are common, depending on how the project is implemented and maintained [8]. In another scenario, **Git** might scale better because there is no bottleneck of a central server for common operations performed by thousands of developers [7]. Another key difference is how projects are organized and managed. **Git** is repository oriented, meaning that repositories are semi-isolated environments and security is managed at a repository level. **Perforce**, on the other hand, allows managing security at different granularities irrespective of how the project is organized [8]. Lastly, **Git**'s distributed model allows for completely independent offline development, whereas **Perforce** is more connected. This connection provides some visibility into work in progress [8], but is less flexible.

3.1 Centralized Version Control Motivation

There are several technical and cultural reasons why the game development industry prefers the centralized version control of **Perforce** to enable their global, cooperative development. The first reason is that game projects contain large binary files alongside the source code. These binary files can be textures, 3D models, visual scripts, and other abstract objects that mean

something to the game engine, possibly playing a part in the **Data Driven Architecture** mentioned in the previous section. Because of the "single source of truth" [8] provided by the centralized history, there is less duplication of the large binary files and their diffs. This makes it faster for developers to fetch content and takes up less space on their machine.

An additional feature that games projects depend on is the "exclusive checkout" of these binary files. This means that one user can lock a file and has exclusive access to it to make changes. For a **Git** user, this might seem contrary to the purpose of good version control. However, it is essential for binaries that are edited as part of the development process because changes from multiple developers cannot be merged in a binary the same way they can in a plain text code file. If two developers were to make changes, one would have to override and nullify the other's work. **Git**'s **Large File Storage** solutions lack mature support for this type of workflow [7].

The emphasis put on large, diverse asset types in video game repositories is valid when discussing version control solutions because these assets contribute to the development process in a special way. In less specialized software projects, large files and binaries tend to be generated or independent of the source code. Here, however, it is helpful to treat binaries as first-class objects and have a symmetric workflow for working on code and content. By storing assets directly with the code in a centralized server, developers can avoid the complexity, maintenance, and dynamic costs of **Git LFS** or artifact servers.

The security benefits of the centralized **Perforce** product may also contribute to its success in this industry [8]. Game development managers working with distributed teams may find it beneficial to implement granular security controls for specific files, such as art files contributed by an external team. This can be done without changing the organization of files and potentially limiting the direct integration between assets and source code. In general, some software teams who work with **Git** are considering a transition from a many-repository structure to a mono-repository structure. The reasons are the ability to reuse code libraries across the organization, unify the build system and automation pipelines, and reduce the administration burden of repositories. One issue these teams might encounter is the inability to provide granular security in the mono-repository, especially when working with contractors and external teams. The options provided by centralized version control might be worth the transition for these specific teams.

In addition to the technical reasons why the centralized version control is preferred in video game development, there are cultural and historical reasons. As has been discussed, game projects require contributions from non-technical, creative teams. **Perforce** provided a GUI client from the start, and thus gained traction with these teams in terms of usability before the rise of modern **Git** GUIs. Additionally, **Perforce** has always been designed for **Windows** development, which benefits non-technical contributors and game developers in general [7]. In summary, **Perforce** had targeted the needs of game development long before modern Git solutions started making progress toward that end.

3.2 Version Control Conclusions

There are both technical and cultural reasons why centralized version control thrives in the game development world. Chiefly, large binary assets as part of the product source benefit greatly from the centralized models for many reasons. In general, version control can become mundane in the life of engineers and organizations, so the costs and benefits of various solutions are not often considered. However, it is important to understand that these engrained preferences arise from strong technical motivation.

A hypothetical, future version control system might be designed to accommodate the needs of game developers while preserving the dynamism of other systems like **Git**. **Perforce** provides a tool, **Helix4Git**, that makes progress toward this idea. Integration with the major source control hosting services could help a hybrid tool gain traction. There is room for providing standardized source control software and hosting that can bring programmers and non-technical contributors together under a common interface.

4 Global Testing and Validation of Games

The creative aspect of game development has consequences for testing and validation. The subjective nature of the final product makes testing difficult, because it is hard to declaratively say what the correct behavior should be. Instead, success is based on the amount of fun and the smoothness of the experience. Because the user can perform many different actions, there is often an uncountable number of situations, making it "harder to explore the state space in games" [4] with automated testing. Furthermore, randomness and emergent behavior in games do not lend themselves to automated testing. Because of these challenges, automated unit tests are not seen as important by game developers because the general feeling of the game is the key indicator of its success [4].

4.1 Playtesting and Outsourcing

According to research, the main avenue of testing game software is the concept of **Playtesting**. **Playtesting** allows the game to be tested by simply playing it, allowing the emergent behavior of the simulation to be evaluated for both technical performance and experiential quality.

Murphy-Hill et al suggest that the reason "human testing is so common is because it is relatively cheap" [4]. Cost is a major concern for game development companies, especially given the unpredictable timeframes implied by the challenges previously discussed. Fatima et al suggest that a major way to reduce costs is to embrace global software engineering and practice outsourcing [1]. **Playtesting** is an area where outsourcing might provide enough benefits without sacrificing the quality of the product. If

the **Playtesting** is left to an external team, their outside perspective might provide better feedback than the original developers interpretation of their own work. One consideration that should be made is whether these external teams are representative of the culture and community of the target audience. Secondly, although testing can be easily distributed because it requires less technical involvement than development, the testing team should still have a high-level understanding of the technology so that they can give meaningful feedback.

4.2 Unit Testing and Automation

Some research suggests that avoidance of automated unit testing in game development is based on fears that code might need to be thrown out when creative changes are requested [2]. Furthermore, because the technology might be designed for a product that is released once, testing is seen as waste because the code will not need to be maintained later [2].

The concept of **Product Oriented Development** could give a lifeline to unit testing practices in game development. By clearly defining the boundaries of internal software products, companies could find areas of their code base that can be tested as isolated products. This type of development would encourage smaller modules, which in turn creates opportunities to test those modules before they are used elsewhere. This also would create a culture where software products are expected to be reused, rather than expected to be thrown away, further justifying rigorous, automated tests.

5 Conclusion

This article has discussed the challenges presented by game development and the software behind creative applications. These challenges present significant complications, and some solutions, when introducing the practice of global software engineering. Firstly, software architecture can play a key role in reducing friction between teams distributed globally and teams of diverse expertise. Techniques like **Data Driven Architecture** and **Product Oriented Development** can be further formalized to accommodate the fluid requirements presented to software engineers in this space. Game programmers can evolve their approach to software architecture to better handle a globally distributed environment. Secondly, the version control divide between video game developers and standard software developers needs to be formally researched to understand how distributed teams can improve their code and asset collaboration. In addition, more research might reveal the next evolution of version control that combines the best qualities of the current solutions. Thirdly, testing and validation is uniquely challenging for games, but global development can be a boon to practices like **Playtesting**. Improvements in architecture can encourage more automated testing in the industry.

Creative input is the defining feature of game development and presents unique challenges to globally distributed development. This creates an extreme environment for architecture, asset and code management, and software validation. Successes in this environment should be researched because it could provide revelations for global development in every domain.

REFERENCES

[1] Alia Fatima, Tayyaba Rasool, Dr. Usman Qamar. 2018. GDGSE: Game Development with Global Software Engineering *2018 IEEE Games, Entertainment, Media Conference (GEM)* (2018) 1-9. DOI 10.1109/GEM.2018.8516498.

[2] Jussi Kasurinen, Maria Palacin-Silva, Erno Vanhala. 2017. What Concerns Game Developers?: A Study on Game Development Processes, Sustainability and Metrics. *IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)* (2017). DOI 10.1109/WETSoM.2017..3

[3] Wilson K. Mizutani, Vinícius K. Daros, Fabio Kon, 2019. Software architecture for digital game mechanics: A systematic literature review, *Entertainment Computing, Volume 38,* (2021), https://doi.org/10.1016/j.entcom.2021.100421.

[4] Emerson Murphy-Hill, Thomas Zimmermann, Nachiappan Nagappan. 2014. Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development? *ICSE'14* (2014) 1–11. DOI 10.1145/2568225.2568226.

[5] Marcus Toftedahl, Henrik Engström. 2019. A Taxonomy of Game Engines and the Tools that Drive the Industry. *Proceedings of DiGRA 2019* (2019). URN urn:nbn:se:his:diva-17706.

[6] Alf Inge Wang and Njål Nordmark. 2015. Software Architectures and the Creative Processes in Game Development *IFIP International Federation for Information Processing. ICEC, LNCS* 9353 (2015) 272–285. DOI 10.1007/978-3-319-24589-8_21.

[7] Atlassian Corporation. Perforce to Git: Why make the move. Retrieved from https://www.atlassian.com/git/tutorials/perforce-git.

[8] Perforce Software, Inc. 2019. Git vs. Perforce: How to Choose (and When to Use Both). Retrieved from https://www.perforce.com/blog/vcs/git-vs-perforce-how-choose-and-when-use-both.