Why You Should(n't) Build Your Own Game Engine

Andrés Rivela
Digital Bandit Studios
Vancouver, B.C., Canada
andres.rivela@digitalbanditstudios.com

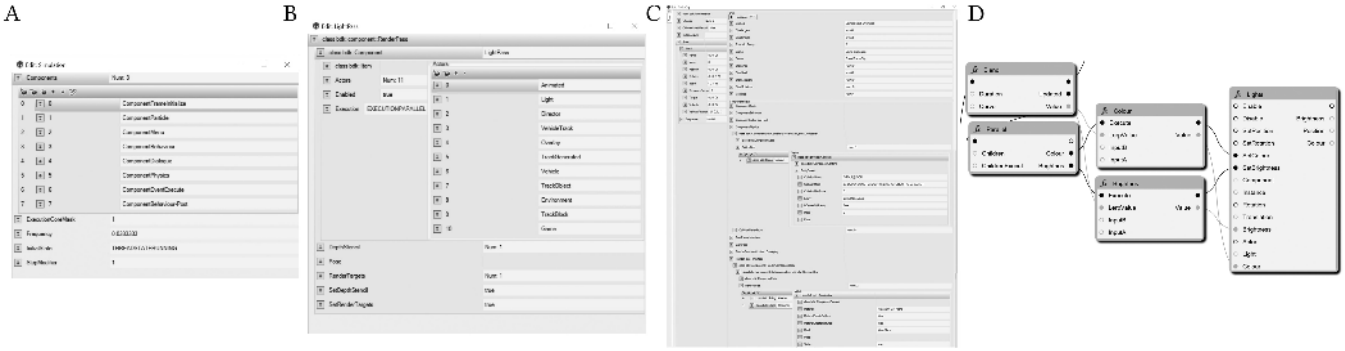

Figure 1: (A) A thread definition and its associated components. (B) A component definition and its associated actors. (C) An actor definition with instance data and attached component data. (D) An example behavior asset used for all logic and scripting.

ABSTRACT

Developing a modern game engine from the ground up has become an increasingly rare opportunity, and with good reason. It is a costly commitment and coupled with the existing technologies readily available at reasonable pricing models, it is a hard sell for any startup to take on such a burden.

This paper focuses on a few key issues when developing such a technology base to serve as both a guide and a warning. Rather than discussing the implementation details and features of the engine, the paper will delve into the importance of efficient workflows; the challenges of outsourcing, and finally the lessons learned from building the technology and a game that runs on it.

CCS CONCEPTS

• **Software and its engineering → Object oriented architectures**.

KEYWORDS

Game Engine, Software Architecture, Tools

ACM Reference Format:
Andrés Rivela. 2019. Why You Should(n't) Build Your Own Game Engine. In *Proceedings of SIGGRAPH '19 Talks.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3306307.3328180

SIGGRAPH '19 Talks, July 28 - August 01, 2019, Los Angeles, CA, USA

ACM ISBN 978-1-4503-6317-4/19/07.
https://doi.org/10.1145/3306307.3328180

1 WORKFLOWS ARE KING

Everyone knows the saying, "Work smart, not hard." The truth of an indie project is that you will have to do both, but it is imperative that the time spent on the project consistently pushes the product steadily forward. This is only possible with good tools and processes.

1.1 CODE WORKFLOW

Ideally, the engine should always be running. As the project matures, engine start-up times will increase respectively for various reasons. As such, shutting down the engine, compiling the code, and restarting becomes a prohibitive workflow. While updates to the core engine and tools will demand it, changes to the game code should not. Therefore, the core engine is a C++ DLL that houses all the critical runtime code and systems. On top of that sits a C# Editor that exposes engine functionality and provides various tools and widgets. Between them is a Managed DLL that governs data marshalling and interoperability between the managed and unmanaged systems. Finally, game specific code lives in a separate DLL that is loaded into a separate domain to allow it to be hot-loaded whenever updated.

1.2 ASSET WORKFLOW

Working with content requires the same quick turnaround as code. Assets in the engine are anything that requires an associated editor for previewing and/or editing. In addition, an asset can specify an external source that will be monitored for changes and updated in-engine automatically. A few surprising lessons surfaced from the continued use of certain engine tools. First, was the realization that can only come from mastery, that is, the extensive use of any given tool reveals its full capabilities. Many tools were predicated on an initial idea or requirement that I designed around meeting the said

use-case. It is only after implementing and using the tools for more than basic proof of concept that I became aware of the power of a tool and how much more functionality could be leveraged from them. It seems counterintuitive, but I had designed and built tools without fully realizing at the time how to use them. Secondly, after employing any tool for a significant amount of time, the need to address quality of life features becomes tantamount to productivity. The node-based behavior tool is a perfect example: trivial features like copy and paste within and across behaviors or cloning of inputs became major drivers to facilitate the creation of more complex logic. Initially I was hesitant of spending time on improving existing (and functioning) tools. Early on I adopted a "let the needs drive requirements" mentality to avoid getting sucked into developing engine features I wanted versus those I needed. This was a mistake, but it became apparent only after addressing quality of life issues with some of the heavily used tools. The easier it is to work, the more willing you are to be thorough, and the better your final product will be for it.

2 CONTENT OUTSOURCING

Engineering challenges aside, content outsourcing proved to be the biggest operational problem from a production point of view. The major drawback being that content creators were unfamiliar with the engine and were not provided a framework to simulate the final environment - for time constraints rather than technical ones. As a result, there was no testing or validation that could be done on their end prior to the delivery of an asset. In the case of 3D models, only FBX files are supported since Autodesk Maya is the only licensed product I own, and further, only a subset of FBX features were implemented to suit the needs of the engine as they arose. As a result, I was limited to artists working only with Maya and requiring them to ensure specific properties of the exported meshes were handled correctly - one such requirement was freezing transforms prior to export. Whilst none of the aforementioned issues was critical, they did add a level of frustration for the artists that felt constrained with their workflows. Consequently, I ended up creating a lot of the content myself, or at the very least had to touch every asset purchased, contracted, or found in free marketplaces. Postponing final bulk asset creation is a good idea, especially when authoring content for an ever-evolving technology base. Expect to go over asset types multiple times and sometimes entirely re-author them. Content creation felt like a perpetual case of two steps forward and one back throughout the entirety of development.

3 JACK OF ALL TRADES MASTER OF NONE

My journey led me to be the designer and artist, as well as the pipeline, game, and engine engineers. This firsthand comprehensive view of development highlighted, above all, that needs should dictate features. This became an increasingly frustrating paradigm as the development time grew. The breadth of work inevitably means that you have only so much time to dedicate to any given specific engine system. This often results in stripped-down implementations of some features or entirely avoiding them. When the project began, DX12 had not been released and so the renderer was written around a DX11-style API. When it was finally available, I could not justify going back and re-implementing the renderer. This also applies to rendering techniques that have become popular over the last few years including screen-space reflections and volumetric lights and fog. In years past, I would have implemented these features in a test-bed solely for research purposes on my own time. However, as an indie developer, the differentiation between your own time and work time is blurry. As a result, I feel somewhat behind the times with the latest rendering techniques and one of the things I look forward to the most is being able to get back to doing some implementation research.

4 CONCLUSION

Ultimately, my conclusion is that building your own engine is an academic exercise. Alone, you cannot hope to compete against existing well-established technologies, nor can you hope to build games of significant scope due to resource limitations. If your goal is simply to make games then you should use existing technologies. Only the requirement of some absurd feature that cannot be found elsewhere can ultimately justify taking such a burden on. Personally, I am a romantic of the old days and this was something I needed to do for myself.

5 BY THE NUMBERS

Approximately 4 years of development time and 180,185 lines of code in 1003 source files. 271 Behaviors, 284 Textures, 566 Meshes, 91 Materials (86 HLSL Shaders), and 126 Audio Files. 1 Developer.