I. INTRODUCTION

Developing modern 3D video games is notoriously difficult. Games constantly try to push the envelope of what is achievable in soft real-time with current hardware, while having to meet strict release dates for commercial reasons. Given what modern hardware is capable of, it is hardly surprising that games are frequently on the forefront of advances in areas such as computer graphics, real-time physics simulation and artificial intelligence.

Because modern games are such complex beasts, developing an entire game from scratch is an immense endeavor [1]. It therefore makes sense to re-use previous work as much as possible. In the context of games, this usually means using middleware libraries or even entire game engines, which can be seen as pre-packaged collections of related middleware. It is widely known, however, that effectively re-using software is difficult. While games are perhaps more suited for re-use than some other types of software, games often have to modify and fit together several middleware libraries and integrate them with the core game logic. Additionally, game developers have to deal with all the common problems in software engineering, like handling design changes during development and debugging complex real-time systems.

In this paper we build a representative game engine using industry standard languages and best practices found from the related literature. We present our resulting architecture along with interesting implementation details of the engine, and analyze how modern games tackle the difficulties described above.

This paper is structured as follows. In section II we briefly introduce some middleware libraries for game development and review architectures of some existing game engines. In section III we present our architecture in detail. Section IV further elaborates on our use of a domain-specific language for game logic scripting. Finally, we discuss our findings and draw conclusions in section V.

II. RELATED WORK

- Languages
- Libraries
- Engines
- Other papers

III. COMPONENT-BASED ARCHITECTURE

- Implementation inheritance is bad.
- Big Ball of Mud and the Deadly Diamond of Death.
- has-a instead of is-a.
- Predefined, standard communication interfaces.
- Enables data-driven object creation, if necessary.
- How to optimize?
- Interesting implementation details.

IV. DOMAIN-SPECIFIC LANGUAGE

- Advantages
- Disadvantages
- Improvements
- Actors vs Component message passing: here or section III?

V. CONCLUSION

No results yet, boss!
Components perhaps not end-all-be-all. [2]
More sophisticated languages, especially more sophisticated static typing, both in the engine and scripting departments. [3]

REFERENCES