UNIVERSIDADE FEDERAL DE PERNAMBUCO

Universidade Federal de Pernambuco

Centro de Ciências Exatas e Naturais

Centro de Informática

Ciência da Computação

João Lucas Mendes de Lemos Lins

An Engine For The Hobby Game Dev: Simplifying game creation through application of data-driven design an modularity on the engine level

Trabalho de Graduação

Recife

15 de Dezembro de 2017

João Lucas Mendes de Lemos Lins

An Engine For The Hobby Game Dev: Simplifying game creation through application of data-driven design an modularity on the engine level

Trabalho de Graduação apresentado ao Programa de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como parte dos requisitos necessários à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Giordano Cabral Eulalio

Recife

15 de Dezembro de 2017

João Lucas Mendes de Lemos Lins

An Engine For The Hobby Game Dev: Simplifying game creation through application of data-driven design an modularity on the engine level/ João Lucas Mendes de Lemos Lins. – Recife, 15 de Dezembro de 2017-

48 p. : il. (algumas color.) ; 30 cm.

Orientador: Giordano Cabral Eulalio

Trabalho de Graduação – **Universidade Federal de Pernambuco**
Centro de Ciências Exatas e Naturais
Centro de Informática
Ciência da Computação , 15 de Dezembro de 2017.

IMPORTANTE: ESSE É APENAS UM TEXTO DE EXEMPLO DE FICHA CATALOGRÁFICA. VOCÊ DEVERÁ SOLICITAR UMA FICHA CATALOGRÁFICA PARA SEU TRABALHO NA BILBIOTECA DA SUA INSTITUIÇÃO (OU DEPARTAMENTO).

João Lucas Mendes de Lemos Lins

An Engine For The Hobby Game Dev: Simplifying game creation through application of data-driven design an modularity on the engine level

IMPORTANTE: ESSE É APENAS UM TEXTO DE EXEMPLO DE FOLHA DE APROVAÇÃO. VOCÊ DEVERÁ SOLICITAR UMA FOLHA DE APROVAÇÃO PARA SEU TRABALHO NA SECRETARIA DO SEU CURSO (OU DEPARTAMENTO).

Trabalho aprovado. Recife, DATA DA APROVAÇÃO:

Giordano Cabral Eulalio
Orientador

Professor
Convidado 1

Professor
Convidado 2

Recife

15 de Dezembro de 2017

To all the unsung stories in the night. We will listen!

Agradecimentos

To all that talked to me, advised me and shared with me their stories, this project could not be done without you.

Special thanks go to Leo Falcão, multi-media storytelling researcher for the Universidade Católica de Pernambuco, for sharing his inordinate amount of experience in storytelling with us, and to all the users of the GameDev chat, for sharing with us their experiences. Though the concepts here are my own, the user interfaces you see throghout the project were created by Pedro Barroca; without him, this document would be a lot drabber.

Important to note is that, though my name is in the cover, this work is a culmination of years of work by people way more dedicated and talented than me, and what I am, I owe to them.

Finally I would like to thank all my family and friends, for without their support, I would not be here today.

Epígrafe

Resumo

Abstract

Lista de ilustrações

Lista de tabelas

Sumário

1 Introduction

The games industry is one of the fastest growing in the world.(ENTERTAINMENT SOFTWARE ASSOCIATION, 2014) Coming from humble beginnings, it long surpassed film and music(NATH, 2016; CHATFIELD, 2009) and become probably the most important media industry of today. In doing so, we've created vast support structures that favor the development of games as a profession, ranging from university courses to specific middleware that greatly ease the strain of development for specific teams. Despite that we never really think about how such developments affect the hobby developer.

The hobby developer has long been a tricky subject. While they are an essential part of the industry, nobody seems to pay much attention to them. Solution developers targetting game development seem to expect solving a problem for the larger industry will eventually have the same effect for non-professional developers. We've found that is not always true, as the circumstances surrounding the developer greatly influence the effectiviness of such solutions. While studying for this project, we've found several resources created for solving the problems that ail game development, but very few that focus on the specific needs or the hobby game developer. Probably because of their status as products, most of these solutions tend to focus the professional side of game development where the exdenditure of money can be better justified. Meanwhile, there exist problems that are endemic to the hobby developer background, like lack of training in the skills necessary for game development, or the need for a better collaboration framework, and these remain largelly unaddressed. The result is that while game development has evolved into a more and more diverse and in-depth area throughout the years, the accessibility of the hobby developer to this depth has remained limited.

Which is odd, if we stop to think about it. From the beginning creating games was a leasure project, before it became a commercial one. Games used to be made for friends and family, for the fun of creating them, way before they've become a means for earning a living(KENT, 2001). This unassuming place is where the practice began, way before that was any way to make money out of them. In a historical sense, the games industry wouldn't be here today if it were not for the work of hobby developers in establishing the imporance of games to our society.(FLOYD et al.,)

It is easy to forget given the huge industry games have become, but most game developers today are actually hobbyist, not processionals. Verifying such a suspicion may be hard since these are the people who go largely unseen in surveys and census, but we are able to make some good guesses. ESA's 2016 survey of the industry counted 65,678 direct employees working in the games industry in the United States(the largest gaming market worldwide)(ASSOCIATION, 2017). While that might seem like a big number, a single game developer focused site(gamedev.net) has over 3 times that many

users at 240,029(GAMEDEV. . . ,). Itch.io, a 5 year old platform focused on the hobby game developer has a catalogue of 83,457 free games created non-professionally with no intention of profit(ITCH.IO,). That is over 4 times the amount of games released on Steam(the major distribution platform on PC) in it's 13 years of market, with a catalogue of 19,845 games(LLC,). That doesn't even account for the possible hundreds of thousands more in the dark corners of the internet, or that are simply never shared.

These represent people making games for their children, their friends, or just for their own amusement with no interest of every making any money out of the endeavour. They might be often regarded as a lower grade of developer, the ones that don't put on the effort to go pro, but that is not fair representation. Much as your cousin that plays on a sports team might never go play for the major leagues or your husband that does glassblowing on weekends might never drop his job to become a professional glass-blower, these are the people who are interested enough in games to make something for themselves without the incentive of monetary gain, and that love always shows through in their projects.

It is important to remember the importance of non-professional developers for the industry in general so that we can be better motivated to support them. These reasons may not be obvious at a first glance, but it is directly related to quite a few aspects that make the games great(FLOYD et al.,):

- First, it is important to remember that non-professional game development is part of the road towards professional game development. While not all non-professional game developers will eventually end up in the industry, it is safe to say that all professional game developers were hobby developers at some point. Creating games non-professionally represents an important stepping stone in an industry such as this that puts so much value on hands-on experience.(FOERTSCH et al., 2017; CHIRONIS, 2015)

- Second, without the pressure of having to turn on a profit, hobby developers are more open to explore crazy new concepts that might be too risky for a commercial venture. No matter how strict or lax the company rules, all commercial games are first and foremost products, and this results a(maybe even unconscious) pressure to better weight your risks. Making crazy new stuff is a lot easier when no one's job is in the line.(ĐÔNG, ; ROHRER,)

- Third, hobby game developers can go into incredibly niche subjects. For the same reason as outlined above, non-professional developers are able to create games based on really specific and often personal subjects. The experiences enabled by these games are not perfect, and often not even fun, but they represent points of view that are seldom explored in other media.(KHONSARI,)

With this work we propose we give those people more attention, creating a solution that focuses on fulfilling their specific needs towards game development. To achieve that, we've decided that we must first look towards forming a better understanding of who the hobby game developer is, and how specific problems of game development manifest in their unique environments. Doing so is hightly difficult not only due to the scarcity of good papers on game development in general(FLOYD et al., 2016; HAGGIS, 2017), but also because this low profile public has largely gone unnoticed by most researchers in the area. As such, this project relies heavily on surveys and direct contact with the many non-professional development communities around the net in its effort for identifying the problems that particularly affect them.

In our search to better understand the needs of the hobby developer, we first need to understand how they relate the needs of the games development in general. To do that, in Chapter 2 we establish a context and revision the most common problems that affect the process of professional video game development, as documented by the academia. Then, on Chapter 3 we come back to the hobby developer, establish how those larger problems affect them specifically. These problems are then further explored on Chapter 4, in which we discuss them in more specific terms. Following this exploration of the problem and the context surrounding it, we then move on to looking for a solution to address it. Chapter 5 defines how we've achieved that through the use of a new paradigm in game engine architecture, and Chapter 6 goes into further detail in regards to its implementation. Finally, Chapter 7 describes the results of our solution so far, Chapter 8 discusses possible future projects that could be explored in this architecture, and Chapter 9 closes off the paper with our thoughts on how the engine architecture impacts the game development landscape.

2 Context

2.1 Making games is hard

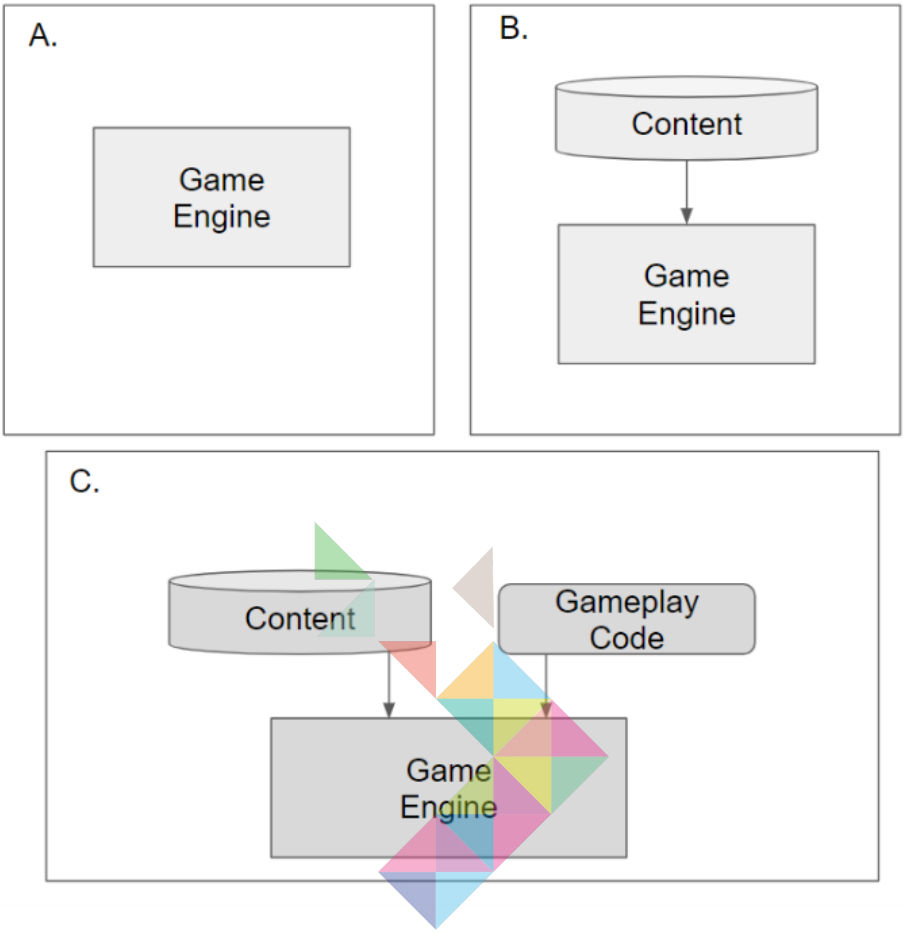
A long time ago, in times when people still thought digital watches were a great idea, and cellphones didn't dream of existing, computers were a pretty niche hobby. In those times, you had to struggle to get a message to print on a screen(that you assembled yourself!) and you had an even harder time making it fit in a microchip. Yet, even at those trying times, people were already making games.

Even with rustic technology, people were always driven by the ludic splendor of games. The first computer game ever released was Spacewar, in 1962, and it was made primarily by a single developer(RUSSELL, 1962). Indeed most games developed in the early days, like Mrs Pacman and Adventure, were made by small teams of programmers and a couple artists(GIANTBOMB, a; GIANTBOMB, b).

As time went on and the industry started to mature, making a game became an increasingly larger(and expensive) development. Developers wanted to push the envelope of what was possible with their simple hardware, and gamers came to expect each year to bring better and more complex gaming experiences. Games quickly became an market to rival film and music, quickly growing from a niche market to a billion dollar industry. By 2017 the gaming industry was expected to gross 108.9 billion dollars, way surpassing the movies and music industry.(NEWZOO, 2017)

By then making games was no longer a single man project. In the search for more for better gameplay and ever increasing sales, making games became a company endeavor involving hundreds of people and millions of dollars. Specialists from all related areas were brought in to paint, code, and write the newest AAA game in an effort to make it certain it would succeed in the it's launch week.

**Figura 1 – Figure 1: On the left, the entire development team of Squaresoft on the release of Final Fantasy in 1987.
On the right is the production team of Square Enix Europe in 2014.**



http://jpninfo.com/21060 e http://www.tombraidercollection.com/single-post/2014/09/11/Square-Enix-Europe-leaves-its-offices-of-Wimbledon

Regardless of that investment, making games has become harder, not easier. Sure, with years of experience and millions of dollars behind us, we've solved some of the problems that plagued the early days. We've created specialized hardware made exclusively for attending the demands of the gaming market and we've developed a library of techniques and good practices to make creating games more efficient. And yet still, we fail.

Unseen64.net serves as a memorial to all the games that have been cancelled after a public release, containing records of thousands of games that will never see the light of the day(UNSEEN64.NET,). Even then, those account only for the games which had open publicity before their cancellation, many more probably lie in the darkness, never to be seen or heard from but for the developers who worked on it.

Certainly, corporate decision higher up on the hierarchy of the behemoths that game companies now have become might be to blame for this tragedy(and don't mistake the loss of any game for anything else), but not all of it.

In February of 2004, Jonathan Blow, a famous programmer and video game designer, posted an often cited article entitled “Game Development: Harder Than You Think” (BLOW, 2004). In it, Blow outlines what he considers to be the major problems hampering modern video game development, mostly from the view of a game programmer, though the views in the piece extend to much wider trends in the gaming industry. The article divides the problems into two major categories, that while not entirely not entirely distinct(as admitted by the author himself), does serve as a good didactic structure for discussing the problem.

Not to repeat the the article, here we briefly summarize both categories:

- Problems due to project size and complexity: These relate to the increase in knowledge necessary to make a modern game. While a game in the early days was made from very simple components the demands of modern game design mean that dozens of interlocking parts are necessary. This relates not only to the way programmers need to know graphics programming, audio programming, multi-platform development, etc. but also to how many new people with different expertises need to be integrated into a modern project, and how this affects productivity.
- Problems due to highly domain-specific requirements: These relate to how over the years the art of making games has grown not only in width but also in depth. Programmers these days are required to know highly specialized techniques, built over decades of iteration on how to build AI, physics simulations and graphics. Once again, these ideas relate not only to programmers, as writers, and artists have come to found in games uniquely specialized challenges that require consideration.

While great strides have been made to remedy these issues since Blow‘s article, these problems still represent a large problem in game making.

In this article we intend to present our own attempt at addressing these problems, with one particular twist. The gaming industry is a deeply secretive one, especially when it comes to their interactions with the academia. That said, in our research for solutions for these problems, we noticed a general lack of attention given(even for industry standards) to a particular subset of the developer community that is greatly harmed by these problems; the hobby developers.

3 How does these problems affect hobby developers?

In trying to reach out to this demographic we've created a survey questioning about past experiences with game development and released it on forums, chats and mailing lists. The full survey can be found in Appendix 1, along with the full list of webpages where it was posted and it's results. The survey ran for 21 days, and collected 661 responses during that time. We know that while this is nowhere near close to the escope of the whole demographic group, the extensive breadth of the survey should better represent the vast array of experiences through many insular communities inside the main demographic. This makes us confident this is a trustworthy representation of the group, even though our numbers were limited. To compliment the surveys, we've also reached out to individuals of those communities online, through chats, forums posts and skype interviews. This was a great resource in better understanding them, helping us to get a feel for the community more than numbers on a survey ever could, and motivating us in our quest to try and them. For any researcher intending to work in this area, we greatly recommend they do their own field research, as the community is very approachable, and very willing to help us better understand their plight. The following section outlines the results of these forays into the hobby game developer community.

The survey showed that only about 65% of interviewed developers actually ever got a game project to a release state. Of the ones that failed to do so, about one third have never even got time to take the project out of paper. When approached, the developer's complaints led us to think back to Blow's problems once again.

Large Complexity. No developer is good at everything, and these in particular struggle to work without the wealth of insider expertise the industry keeps to themselves(though platforms like GDC Vault are starting to change that). More importantly, with little time available to study even their own area, the lone developer is in even deeper trouble when it comes to learning other areas. While talented individuals capable of doing everything from programming to pixel art exist, that is a rare exception. Most hobby developers will either collapse under the weight of trying to learn the disciplines on their own, or they will look for external help, be that through a partner, or a contract.

A few developments in recent years have been of great use in easing this problem for the hobbyists:

- Major community sites have become a great help in providing training, contacts and encouragement.(gamedev.net, gamekodo, /r/gamedev, gamedev, stack)
- Online teaching platforms have helped disseminate the necessary knowledge, enabling self-teaching no matter where in the world.(Pluralsight, GDCVault)

- Looking For Group is now a feature supported by most major sites, that allows users to advertise their projects and skills to those interested.(/r/INAT, gamedev.net, itch.io)

These platforms alleviate the problem, but do not solve them. The very nature of hobby projects as discussed above often goes against this help. The niche nature of some of these projects also make it hard to find a partner sometimes and their weird and innovative nature will often require a deeper understanding of programming and game design than general tutorials and how-tos can provide. Third party tools also help with these problems, but they are usually expensive and create a new problem of their own which we will explore next.

Specialization Requirements. Let's not make fools of ourselves saying that the availability of reasonably "open" third party general engines wasn't one of the greatest developments in the history of game-making. Given how complicated a simple 3D render pipeline can be, it is easy to see the worth of not having to create one from the ground up yourself. The amount of time saved by these engines(and other third party middleware) is enough to be worth thousands of dollars, gladly paid by game producers in exchange for a quicker release cycle. The use of these tools greatly reduces the time spent on tech.(BLOK-ANDERSEN, 2015)

However it is important to understand that "greatly reducing" is not the same as "totally removing". Even if we disconsider the engine design requirements that puzzle even profissional developers, we can't be too quick to dismiss the time required to learn to use third party tools. While reducing a time to deployment on a game from 2000 to 800 man-hours might sound amazing for a studio, 800 hours remain too expensive for a lone developer.

Becoming accustomed with a large engine like Unreal(EPIC GAMES, INC,) or CryEngine(CRYTEK GMBH,) is not a feat for one weekend. These are complex systems that require a lot of dedication to master. Each modern engine contains their own unique graphics pipeline, scripting language, support library(not counting the thousand and one quirks we must get used to for proper functioning). Understanding such engines has become such a large endeavour that there are even commercial textbooks devoted entirely to teaching their systems.(BUSBY; PARRISH; WILSON, ; GUNDLACH; MARTIN,)

A great beam of hope in recent years has been the Unity Engine(UNITY TECHNOLOGIES, c). Released in 2004 and putting a heavy focus on accessibility, the engine has had great success in "democratizing game development"(ROWLAND; CLARKE, 2011)(a title we would gladly have stolen for the title of this project, had they not used it first). Since its first release, the engine has grown to become the most widely used

game development site worldwide, being used in 34% of games released(UNITY TECHNOLOGIES, a). More importantly, Unity's practices and focus on accessibility have had great impact in the industry, being one of the contributing factors for the Indie Boom of 2008(COBBETT, 2017). It wouldn't be fair to not admit that this same care for accessibility and "democratizing game development" has been one of our major inspirations in creating this project.

Despite Unity's focus on making game development easier, and it's huge success in doing so, it is still not perfect. Its popularity helps to lower the barrier of adoption, as it potentializes the facilitating points on the previous section, but it remains a software you must take time to learn, time that is often very precious.

In conclusion, we propose our own categorization for Blow's problems that better fits the issues of hobby game development. We propose that the problems discussed can be understood as a *Problem of Depth* and a *Problem of Breadth*. The Problem of Depth can be defined as the time expenditure necessary to learn new skills related to game development, and how detrimental this expenditure of time can actually be for the hobby developer. The Problem of Breadth can be defined as the need to work in teams, and the difficulties that arise from trying to coordinate an heterogenous work team without the proper understanding of the processes involved in team managment.

Finally, we found that the tools that currently exist are not enough to fully address these problems, and decided to give our own try in tackling the issues. To do that, we had to further analyze the problems, and look for a way that they could be resolved in terms of technology and processes.

4 Exploring The Problems

In this section we relate how we explored the problem and the hypotheses made and tested through direct interaction with the community. Finally we summarize our conclusions before moving on to our solution on the next section.

When dealing with complicated issues such as these it is often a good practice to break them down into smaller components. Here, once again Blow's categorization comes in handy.

4.1 The Depth Problem

The information on the surveys first instigated us to take a closer look at the tools available. More specifically, how they were being used and how they are seen by our public.

To achieve that we interviewed community members in online chats and looked at discussion boards in community sites on technology. What we found out was that for a great number of users, games were their first contact with programming. This view is coherent with previous surveys attempting to profile the interests of computer science studies.(RANKIN; GOOCH; GOOCH, 2008)

It is rewarding to see that games are actually a major force in driving people towards our field, but we cannot forget that not all of these developers will ever see a formal programming course in their lives. Actually, most won't. When interviewing users, we saw that most of their learning was based on the resources mentioned in the previous chapter(mostly written tutorials and how-to videos). This creates a problem of specificity.

While "programming" might be seen as a singular area for the gaming culture, it actually involves dozens of specific disciplines, each with their own well of knowledge. It is difficult to find a single man capable of working on all areas of programming required to make a modern game. Studios solve this by hiring multiple personel under the "programmer" role with different specialties, but finding(and specially, managing) such a team is impractical for the hobby developer.

Then comes the solution of "programming-free"(most are actually "programming-light") game makers. Gamemaker, RPG Maker, Stencyl and StoryNexus (STENCYL, LLC, ; YOYO GAMES LTD., ; ENTERBRAIN, INC, ; FAILBETTER GAMES,) are good examples of software friendly to the non-professional developers. Their interfaces allow the user to create commercial-grade games with little programming. Perhaps more important than that, most are free to use and publish games in.

However, when asked about such tools, users often said that they didn't use such tools. When questioned on why, the most common responses were either:

1. Couldn't find the time to learn them.

2. They didn't fit the game they wanted to make.

This brought to light shortcomings of such software as they exists right now.

The first seems to relate to a failing in user experience. Let's not be mistaken that these are powerful engines, capable of the hundred of functions necessary to make a commercial-grade game. The problem then becomes: *"How do you tuck all of these neat features away in a way that is unobtrusive to the user, but lets them find what they want?"*

Then comes the second problem. To provide a better user experience, engine developers will start to favor certain aspects of their software, creating a more narrow, and specific kind of software made to attend a certain genre. This business decision would be enough to solve the problem if there were enough engines being created to cover all genres(which there are not)(ANDERSON et al., 2008). And then again, there still would be the problem of games that transcend genre; those masterpieces that typify their own new genres.

This creates something of a see-saw problem. Do you provide a better, more directed experience, like with RPG Maker, or do you provide a more general, but more difficult to learn one like Stencyl?

In our vision, neither. Upon analysis, we found that this isn't really a problem of finding the right balance. If we look further down the line we will see that this is actually a problem of encapsulation.

In trying to balance flexibility with information density, we've created a program that satisfies neither one, nor the other. When thinking about this problem we asked ourselves: "*What is would the best program to create my game be like?"*. The solution we reached was: **"One that has exactly the things I need and nothing else"**. Then it becomes clear how this conundrum can be solved through modularity. If we break all tools that make those programs into their own modules, we become able to assemble the perfect editor for the creation of any game.

This modularity also enables us to be more flexible. Instead of trying to come up with a software that fulfills all possible permutations of what constitutes a "platformer" we can now work on specific features that make up a game. We can create a module for health, a module for physics, a module for side scroller-graphics, etc. This idea is related to the component design pattern(NYSTROM, ; FOLMER; GROUP,), but taken on an engine level, where the features of the engine itself are components.

As we will see later, this approach will introduce certain problems that will be solved in our discussion of the prototype in Chapter 6.

4.2 The Breadth Problem

When we approached people, instead of finding a large amount of generalists, we instead discovered that most of them are actually very centered in specific areas. We've found writers that couldn't code, artists that couldn't create narratives and programmers that didn't know the first thing about art. Their projects would then end up in one of two categories:

Either the would create projects that focused on their strong points and had less or no requirements towards their weak points.

Or they would collaborate with other members of the communities they could find or outsource freelance work.

The first is is a clear limitation, and while those are not always good, it introduces an unforeseen risk into the project. Some people we interviewed have indeed been incentivized to learn new skills through this limitation and ended up falling in love with them. However, the opposite was also true. When interviewing developers, some admitted to entirely dropping projects because they lost interest once they've reached the parts of development they were not good at. This is not necessarily a failing in their resolve. It isn't fair to expect people to love all parts of game development after all.

The second option becomes a quest to find collaborators. This is where the "Looking for group" resources mentioned in Chapter 3 come in. If we look at those sites right now, we can see several posting of people looking for writers, programmers and artists. What we do not find, however, is a listing looking for producers.

It might be easy to overlook the importance of a producer in a game. After all, they don't actually produce anything that goes into the final game, right? But that is a mistake. A game is a very complicated product, requiring collaboration of people from very different background working together on the same (hopefully)holistic project. A producer is the one responsible for making sure all those parts fit just right.(CHANDLER, 2013)

Producers over time have created an entire science relating to the organization of the processes of game development itself. What is worked on when. How to maximize efficiency for the team. How to maintain focus. How to avoid feature creep. These are all very important answers for getting any game out of the door, and failing to properly address them is often the reason hobby projects fail.

From what we saw, it just isn't common for hobby developers to think in these

terms. Indeed for small teams, some of these questions can be overlooked. The problem comes once the development grows in scope. Without training on how to work together, adding people to the project of a loner developer represents more of a liability than a strength.

If people know how to work together and can depend of each other, everything is peachy. But if that is not the case, each person on a team actually becomes a point of failure for the whole project. This truth is represented community-wide as a feeling that it is “difficult to work with others”. We believe this, along with the very personal nature of these projects to begin with, are the main reasons why this community has so many one or two men projects.

This represents a problem of dependencies on the human level, and as far as we’ve looked, we’ve found no system specifically designed to address it. Sure there are project management software and web applications out there that could be perfect for managing this kind of problem, but the existence of those(and sometimes even the need for them) might not be obvious for a developer that doesn’t come from a project centered background.

It is of our opinion that the solution to this problem is to embed project management knowledge and tools into the platform itself, so as to become obvious for the user the need for them. If the intention of a game-making platform is indeed to see projects succeed, we need to look past providing the tools for creating software and start looking at the tools to make teams.

Here is the point we have to mention the importance of independence in projects. While games remain a deeply collaborative effort, reducing dependencies is good(KANODE; HADDAD, 2009; GREGORY, 2014). Probably the greatest breakthrough in this department in the last 20 years comes to the shift towards data-driven engines.(BLOW, 2004; GREGORY, 2014) Within this framework, programmers can focus on working with engine features, while designers and artists can create gameplay code directly that is then interpreted by the engine dynamically. This frees time from both sides by allowing them to work independently, maximizing time use.

This characteristic is doubly as important in the hobby game-dev circles, where a steady level of engagement is essential for the continuation of the project.

5 The Solution

Given our exploration of the subject, and our interpretation of the problems that afflict them, we've come up with a few changes to the modern paradigm of engine design that we believe should greatly improve completion rates of hobby games.

In looking for solutions for the problems that ail the hobby community we've come across two concepts that we believe will be essential in solving this problem: modularity and data-driven design. The first part of our solution relates to using modularity to its fullest extent, creating an engine that is completely modular with exception of some very small integration modules. We further free up this framework through the use of data-driven concepts. By allowing modules to be interpreted as data to be consumed, it enables us to interpret the game editor environment dynamically, thus making modules more than simple libraries.

Through this approach, we combine the strengths of both approaches to create a very flexible, very powerful engine, that remains easy to comprehend, keeping only the necessary tools for the game you are currently making. The data-driven aspects also enable us to load or reload these tools at runtime, accounting for the case in which the design of the game changes in the middle of a development session.

It amazes us that such a project was never tried in the past. Modularity has been an essential aspect of computer science for a very long time, and data-driven technologies have been very present in most modern game technologies, yet, there has never been a fully modular data-driven engine.

The reason to shy away from this might to be due to the focus of game programmers on performance. One of the drawbacks of such a framework is that it becomes very hard to control the control flow within the application. This means that there is no way to absolutely guarantee a return within the delta-time required to churn out those 60 frames per second(GREGORY, 2014).

There also comes the problem of integration. Without knowledge of what parts are online at the time, it becomes very difficult to enforce well behaved communication between modules. Solving this problem also causes more overhead that further exacerbates realtime problem.

We do not believe that has to be the case, but even if it is, we believe it to be a worthwhile trade-off in this instance. Not all games are played in real time, and we believe the benefits brought in by this framework would be greatly advantageous for the development non-realtime games. Turn-based strategy games, classic RPGs, management sims and text adventures are a few examples of the genres that would not suffer from the delta-time problems.

This modularity also gives us an opportunity to let people work the in a language that is natural to their task. Since all modules are independent and focus on very specific features, they can be programmed with a design language that fits their charge, instead of fitting to overarching design decisions made for the whole program. This will hopefully make working with them even more intuitive, and will open the doors to what we believe to be the greatest innovation of this project; a shift in the way we build tools.

6 The Prototype

In trying to keep to small steps, we decided to focus our prototype into the creation of the modules necessary for a general text adventure engine(and support systems) following the framework described on the last section. We choose text adventures for two primary reasons; first, it has historically been one the simplest forms of games to create, used by many game schools worldwide as a freshman project(SAIL, ; DIGIPEN,); second it happens to fit well with our non-realtime limitation. Another reason we choose a text adventure engine is that most games created today have some form of story engine, so our work could serve further projects through modularity.

Given that, from now on all assertion towards the project will come from the point of view of building a text adventure engine, which we dubbed Writ. Accompanying the engine are a content editor and a website, together creating the Writ Platform.

But of course, our project isn't about a platform. The platform is actually the result of our work. The objective of this paper is to describe a new paradigm to working with engines, which in turn, created Writ.

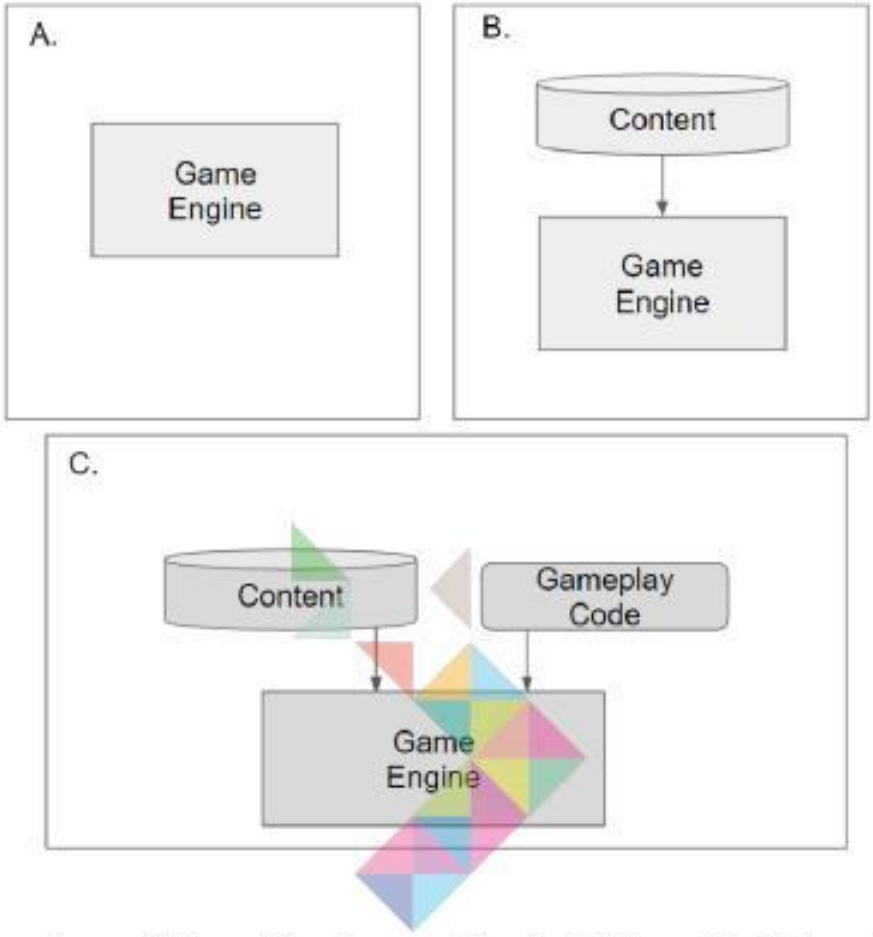
This could be paradigm can be better explored by looking at a practical example, our Writ Platform, and what sets it apart from most game makers in the market today. We will do this in three steps. First we will discuss the Writ model, and how it leverages modularity and data-driven design to solve the specificity problem discussed in the previous sections. Then we will have a look at the Architecture of Writ, and how it solved a few of the problems inherent of a completely modular system. Finally, we will look at the Culture we are trying to create with the Writ Platform, and why it is essential for the survival of the module going forward.

We will now go into each in detail.

6.1 The Model

Going back to the old days for a bit again, we can have a look at how games were divided back then. As we see in Figure 2a, the first games created thought as graphics as an extension of the engine, perhaps due to how simple they were. Those were the Asteroids and the Spacewars, where most of the graphical interface was generated at runtime.

Figura 2 – Figura 2: a. Engine de um jogo antigo, todos os dados fazem parte do proprio código da engine.
b. Engine separada de conteúdo c. Engine data-driven



Then when artists and fancier graphics started to get into the mix, we begun creating a separation between engine, and content. Games were their code, but they also had content they would have to load, so the separation now became as in Figure 2b.

Then we started making data driven engines, and now we've finally struck a separation between gameplay rules, and code. Sure, gameplay and engine code are still very close, but we can all see how an engine with a scripting language could be seen as in Figure 2c.

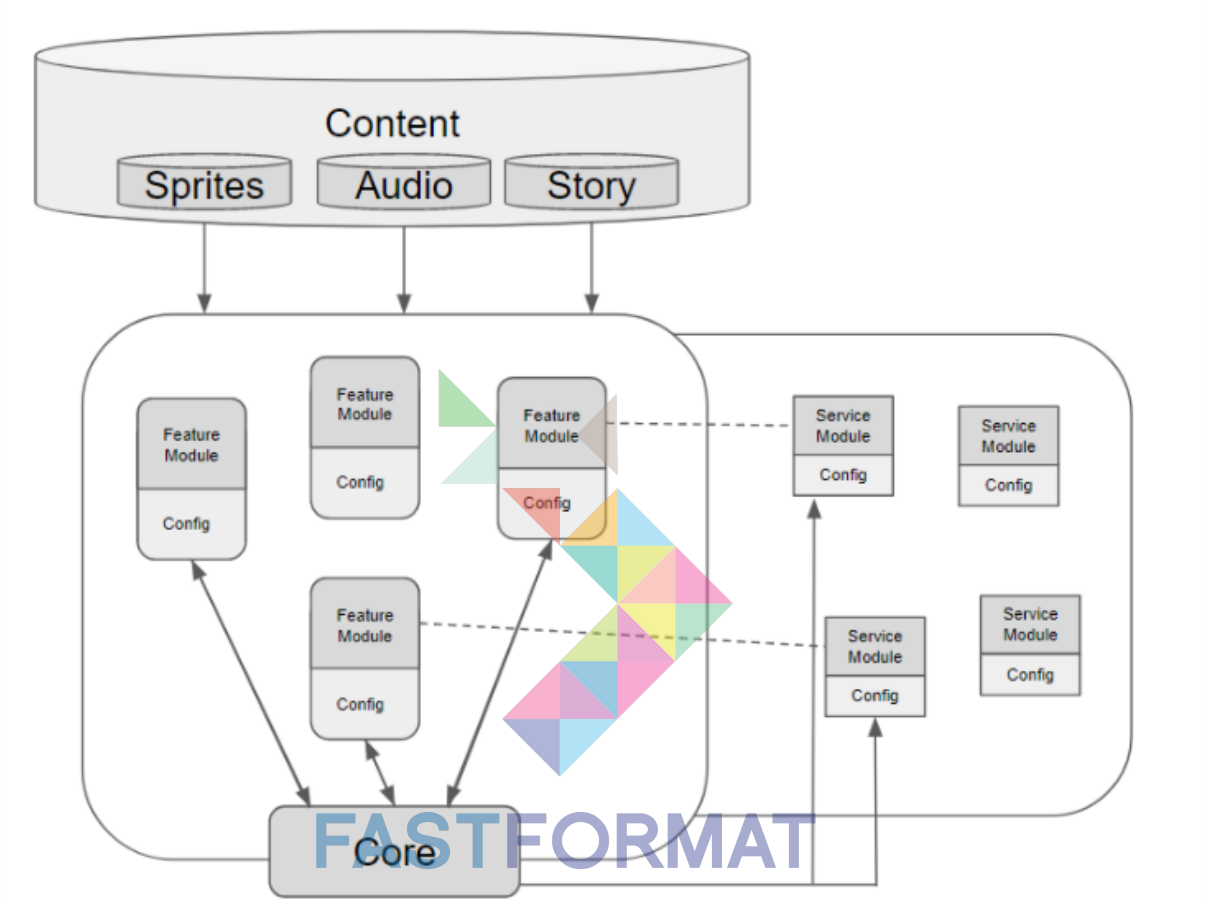
Now we are even proposing breaking the engine into several individual modules, so this diagram might look even more different in the future.

The point I am trying to get at here is that this is a system focused on the engine. A data-driven engine consumes content and consumes game rules. The engine is not only the base but also the main consumer, therefore all parts of the game are made to fit the engine, and not the contrary.

Once we break the engine we can see a shift occur(Figure 3). In our framework, the gameplay is the main point of the architecture. Feature modules contain a unity of

game feature. These feature modules comprise a combination of the code required that allows such game features to exist and the rules programmed into them by the designers. These modules consume data, much like in the old architecture, but more importantly, they use the engine, not the contrary.

Figura 3 – Figure 3: Our modular architecture.



This shift not only represents a more game-centric approach to engine design, but more importantly it also enables us to change the way engine editors are presented. If we look at Figure 3, we can see that "content" isn't really just one kind of data. This content includes sprites, animations, AI automata, story events, etc. So far all of these resources had to be formatted to fit the engine, but now that we decoupled the way the engine functions from the way the game runs, we are free to do with them what we will.

Content production has long been known as the bottleneck of this industry. In a game course, more than 90% of development man-hours are actually spent on (SMITH,)content. With recent attempts at procedural generation failing to reach expectations, we present this freedom as another solution to this problem.

If in the past content creation had been made to fit with engine development,

now that we've broken the monolithic engine into discreet independent blocks, we can enforce the opposite, that engine development is made to fit content creation.

The way we enforce this is through the editor itself. As feature modules are loaded into it according to the necessities of the game design, they load into our (very flexible) editor all the interface aspects required to work their features. In our model all features available in a module must have graphical representation. This is what we call the *What You Get is What You See Rule*(**WYGIWUS Rule** for short). This ensures that the people who came up with the feature can choose the best representation to teach it, and also creates an user experience that is completely code free.

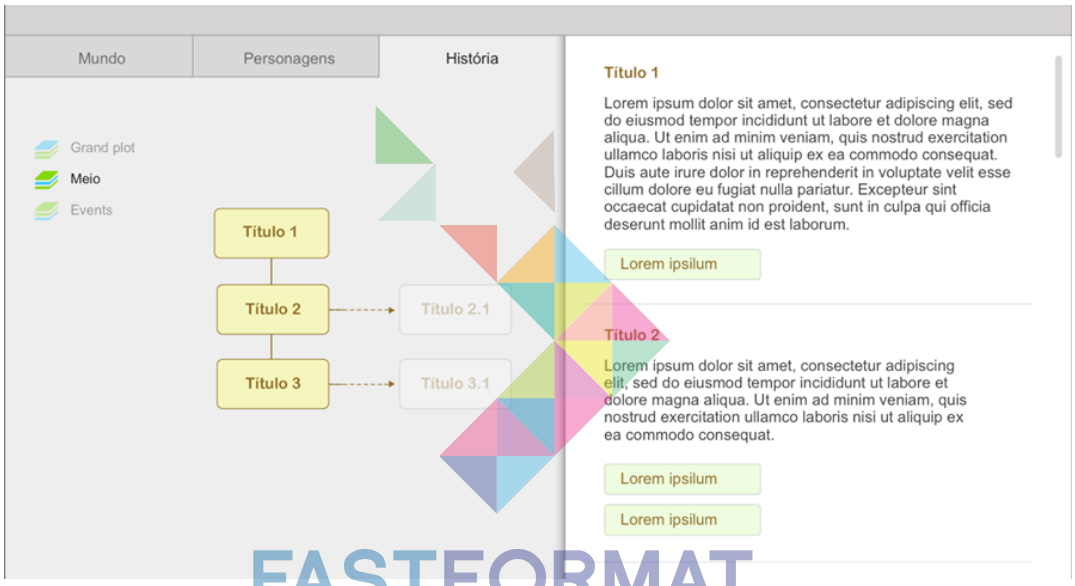
Figura 4 – Figure 4: An example of WYGIWUS, here the user has a "WorldManager" module loaded, the user can then open a library and drag a "codeblock" into the content. This "codeblock" is also edited through the interface, without having to write a single like of code.

Once finished configuring the modules, the editor writes a configuration file that will then be opened by the module on engine launch. When being loaded into the Writ

engine proper, a different part of the module is activated which will consume content and enact the feature on the game itself as specified.

One such example from our prototype is our EventGraph module. For it's creation we worked closely together with Multimedia Narrative professor and Storytelling Researcher Leo Falção. With his help we were able to identify best practices for visualization and authoring of branching storylines. With this knowledge in hand we were then capable of creating mock-up prototypes of how the user experience of this tool would be, and only then we worked on creating the actual implementation.

Figura 5 – Figure 5: The EventGraph module was created with the help of specialists to make sure it is ideal for the task. This is the kind of quality we expect from any new feature module.



This top-bottom approach to software creation isn't new. Indeed it is the recommended approach for games(FULLERTON, 2008). But due to the necessity of a base engine to work on usually meant this approach was only applicable to game design aspects. Now that each part of an engine is it's own self sufficient module, we hope to remove this limitation, and see more top-bottom development of features.

Low-Level Modules. Of course an engine wouldn't be complete without low level modules. These represent memory allocators, data-base managers, render modules, etc. These are the modules that offer services necessary to make features work.

Since these modules are very intricate and usually unseen for most relevant top level choices, we've decided to leave them inaccessible to the regular user. This ensures two things; first it ensures good use of these modules, since the user can only interact with them through a regulated interface(the feature modules); second, it ensures that the interface remains clear for the lay user, hiding most of its complexities through a layer of abstractions.

Once coupled with the Writ website discussed on the Culture section, this also ensures that these modules will only get better and more efficient with time. With revision after revision.

Don't be mistaken, though. A major focus of Writ is on being a didactic experience, so that people can use it as a platform for bigger projects. As such, low-level modules are not exempt from the module creation rules discussed on the Culture section. It is specifically due to their complexity that it is imperative that they have good documentation.

On our Writ prototype we also removed the need for feature modules to be coupled with specific modules. We do this by making all feature modules create a list of "commands" that need to be executed by someone with their current configuration. Once that is done, at run-time, our dependency injection module is then capable of making sure some module is capable of executing the command.

Leveraging Reusability. When working with a third party data-driven engine, each game project will usually yield dozens if not hundreds of scripts related to making it work along with your game rules. The lack of a model to look at game feature engineering means those scripts end up incredibly coupled. This hampers the reusability of such code.

By providing a clear way to look at game feature development, we hope to mitigate this coupling effect. Our model enforces a modular abstraction on the feature level. This reminds people that "a full module in the Writ model is an auto-sufficient unit". With this we hope to incentivize code reuse and sharing, which will be further discussed on section 6.3.

Conclusion. As we can see, the modular framework affords us great freedom in the way we conduct development. The most important part of this freedom, however, is what it can do to the priorities of game development..

On our Writ model we propose to put content creation and tool development on the spotlight, counting on reusability and open source development to ease the strain this puts into software development.

This is of course, not the only way to leverage the freedom offered by modularity, and once this platform is released we look forward to seeing all the innovative ways people choose to work.

6.2 The Architecture

It is clear from the beginning that this framework creates quite a few problems from the architecture perspective.

- **Dealing with dependencies.** First we see from (GREGORY, 2014) that dependencies are an extremely complicated part of engine design. A regular game engine can be a deeply coupled system, and if we do not take care to properly isolate their modules, it can lead to some serious problems.(NYSTROM, ; GREGORY, 2014) And yet here we are defending the use of an unknown amount of modules, on an unknown amount of configurations.

- **Inter-module Communication.** Whenever we encapsulate a module, along we it we must also define the interfaces that will be used to communicate with it. This presents a challenge not only for figuring out how to access these modules, but also to how data is going to be passed between them.

- **Control Flow.**The problem discussed on Chapter 5 about controlling the flow of the application makes a return here. Not only this problem impedes us from maintaining a regular deltatime, but it also creates risk of an update method never actually stopping. In such a flexible system, how do we guarantee a call actually returns?

We will move through these problems one at a time, following the engine's boot sequence and solving them each on their own time. By the end we will have a final architecture diagram for our completely modular Writ engine.

Modules as data. First, we would like to explain how loading Writ works. We've gone through many iterations on how to dynamically load modules. In some of them, each module was a separate application, in others they were directly injected into the code of the Game Loader by the editor application. The version we went with in the end is a bit more flexible.

Taking cues from data-driven design again, each module is loaded into the engine and interpreted as a python script. This approach is good for it allows us the flexibility of running live code, but also bad in that it could lead to performance problems if not dealt with properly. To do that, we use a memory allocator module to allocate python bytecode directly into the memory, and enforce the existence of a liveReload method both on the main engine and on all modules. Once a reload is called on the engine, it will call the reload method of the module, giving it a last chance to save its current state, before being reloaded by the allocator.

This choice to maintain modules as live-data is also very useful for debugging, as it allows live coding.

Initialization. For proper initiation, most modules will need to read data regarding their configurations. This is done through reading their configuration files and serializing them into an Config object that is part of each module's initialization routine.

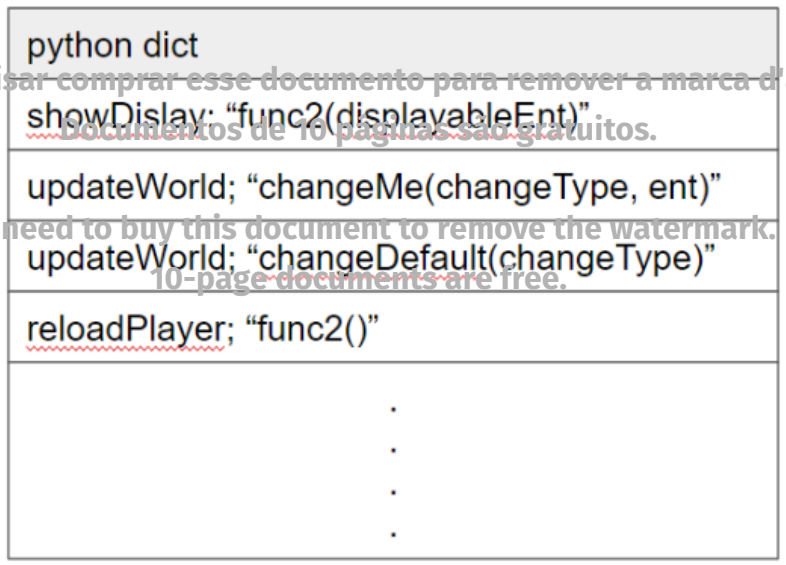
Though initialized and loaded, the modules aren't ready to be started yet. Before that can happen we have to resolve our dependencies.

Dependency Injection. From the very beginning we knew that if we wanted a truly modular system, we would end up having to put some form of dependency injection at the core of the engine.

In software engineering, dependency injection is practice of using one module to satisfy the dependencies of another module. The way this is done tends to vary, but DI is a great way to ensure decoupling between a service provider and a service consumer. The way we use DI in Writ is thought a module we call Caretaker. The Caretaker is one of the core modules initialized on boot and it serves a few important functions to the engine.

The Caretaker ensures all functions required for the running of the game are executed by someone. It does this by reading a file generated by the editor upon commit that lists services that need to be fulfilled. At this point in the boot sequence, all required modules should already have been loaded by the GameLoader, so the Caretaker first looks if a command has a suggested module associated, if doesn't, the Caretaker asks every module in the list of loaded modules if anyone is capable of taking responsibilities for that service through an offer("service", "signature") mandatory function. If no module is capable of fulfilling that service the Caretaker throws an error asking the user whether it would like the engine to download the missing module, or whether they would prefer to abort. After all dependencies are resolved the Caretaker fills a dictionary with all commands associations(Figure 6).

Figura 6 – Figure 6: The dictionary maps function names to first order functions.



The Caretaker ensures all unexpected functions are forwarded to their proper

services. Any module, upon failing to execute a function that is not on it's namespace, forwards the command to the Caretaker. The Caretaker then looks in it's dictionary for an entry containing that function. If it finds an entry, it forward the function to the responsible module, and waits for the response to send back. If it doesn't find any entry, it once again goes through the module list and asks if any can take the responsibility. If no module takes the responsibility, it throws an error.

Data Management. At about this point, modules might want to start receiving data. One of the core modules initiated was an I/O module that carries file associations from the editor. These file associations link which files belong to which service, and forward them using the newly set up Caretaker.

To guarantee this process, any module capable of receiving data must implement a data-loading function. From there on the data is responsibility of the module, though it can be requested again from the I/O module if lost.

This is a good point to talk about data types in the Writ engine. As mentioned previously, type mismatching is a real risk in a system that dynamically associates tasks with modules with different implementations. This means we have two possibly solutions. We enforce data types we think are appropriate for the task through an standard protocol. Alternatively, we could also enforce abstract data types instead, and leave the implementation up to the module creators. What we come up with is a middle ground between these two approaches.

Data types in Writ are not final, instead they can be augmented with module specific data-types. Though communication between modules is all done through standard data types(to ensure module compatibility), we allow each module to define their own additional data types, and store and retrieve them from a common data pool. Along with this, all data moving through modules is enclosed in a content container structure figure 7, that contains an unique sourceid that identifies the origin point of the data. If the original data was read from the hard-drive, it is assigned an unique id when read by the I/O module. When receiving a standard type that lacks information to perform their work, the module can then ask for any supplementary file associated with that id. This means modules can supplement information for their specific need provided by the editor, while maintaining a standard minimum for compatibility. If the data was created by a module, it's id contains a reference to the creator module. If the module knows some modules that consume their data need extra information beyond the regular data type, it can enclose this data in a container and send it to a shared pool, where the other module can retrieve it.

Figura 7 – Figure 7: Therepresentation of a “data” container.

Through this means, data types can be extended to supply the individual needs of specific implementations, but it is also worth noting that standards aren’t set in stone. While it may be rare; with time, new revisions to the standard protocol might be released, allowing us to better supply the needs of the community. For the meantime in which some practice is not widely accepted, these supplementary data types should be enough.

Ensuring a proper update order. Another problem of uncertain dependences is in regards to how they affect the main game loop. In a monolithic engine, a single game loop will be enforced in the game’s main.(NYSTROM,)In this loop, it is imperative that all udate() functions are called in precisely the right order to their respective modules.

This problem was a head scratcher for us for quite some time. We tried defining update phases, like in Unity3D(UNITY TECHNOLOGIES, b), but we have no way to ensure which modules should update first in case there are multiple supposed to run in the same phase. Then we tried strong typing a dependency tree into each of them, but since dependencies are so fluid and we enable the community to create new modules, we couldn’t ever be certain that our dependency tree would be final.

To remedy that loaded a hierarchy tree for each phase unto our web platform, and every time an unknown module is added into the engine or into the editor, it asks the online platform for an update. To do that we also must enforce that each new released module is carefully tested to find out where it fits in the dependency tree. As a side-effect of this, any new version of a module is also considered a new unknown module.

This is far from a perfect solution, but we’ve found no other way to ensure a proper update loop is maintained. As such, checking this dependence tree is the last part of our boot sequence. Regardless, since we had the trouble to include a networking module, we’ve used it for the Auto-publish function discussed on the next section,

Maintaining stable game loops. At this point we have all modules properly set up, communicating properly between themselves, and with all the data they need to work. We've also ensured a proper update order, but how do we ensure those updates actually return to our main?

The problem here is that since flow within the program can go from one module to another, there is no way to know what other modules will be called. In such conditions, it is very easy to create a program where two modules just keep looping back and forth between themselves and the flow never actually returns. In one such tests, the program become completely unresponsive to the user, which is hardly an acceptable effect. One of the suggested solutions were threads, but those would open a whole different can of worms we didn't have time to deal with in time for this project.

To solve this issue in a timely manner for the delivery of the project we've added a call stack and a timer to the Caretaker. This means that as a call is made, the Caretaker records the time and if the current call exceeds the time limit(5000ms) it propagates an error back when the next call to it is made.

Hopefully in the future we will be able to develop a threaded implementation of Writ and we will be better prepared to cope with this issue.

It is worth noting, however, that threads do not necessarily solve the deltatime issue. While threads can guarantee a timely return, they can not ensure the processes actually finish in that time. This means that even with threads, Writ might suffer from missed frames. In modern engine design, the only way to ensure this doesn't happen is through careful crafting of the engine components down to the lowest level, which is something we cannot guarantee.

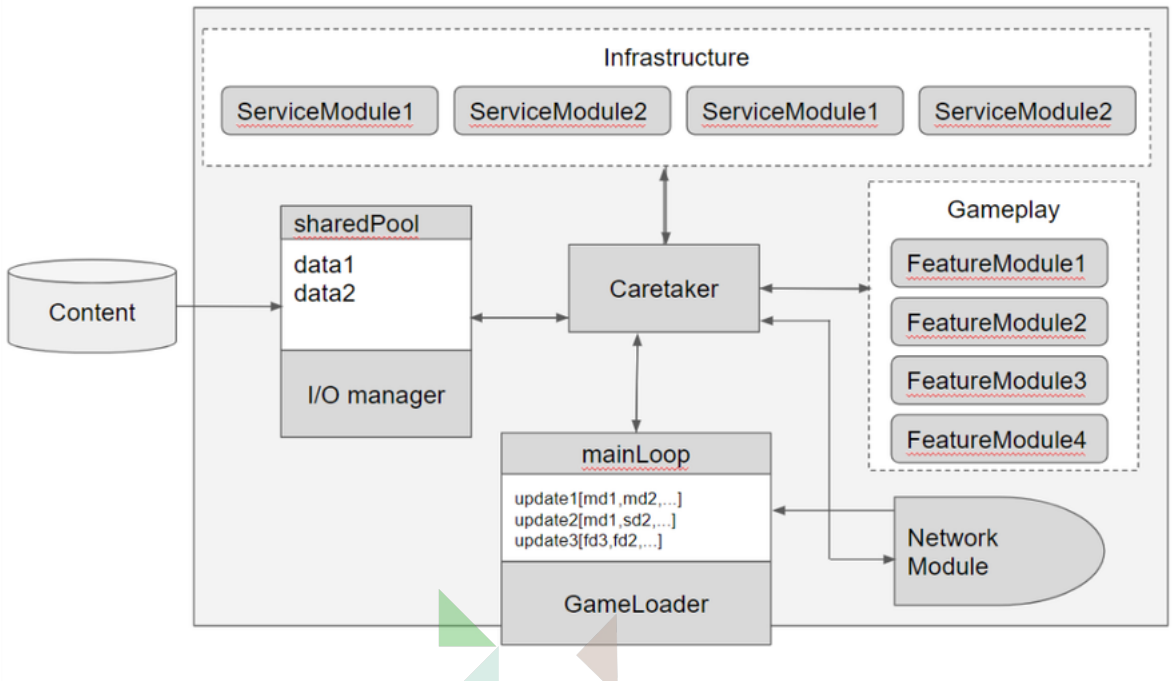
A proposed solution is to make use of certification to guarantee only fine tuned modules are allowed for real-time games, but the effects of such a decision on the hobby game community and our architecture would have to be studied before making any commitment.

Conclusion. In following the boot sequence we've not only discovered all the minimal aspects required by such an engine, but also proved that most of the superficial problems can indeed be overcome with a little application of software engineering, Below in [figure number] is a final diagram of the core parts of the engine.

A more complete diagram including the modules created for the text adventure features can be found in the addendum along with a complete set of requirements.

Further improvements to this architecture are discussed on Chapter 8, regarding future projects.

Figura 8 – Figure 8: The core architecture diagram for the engine.



6.3 The Culture

After all the discussion on models and architectures, what is left if to discuss how these ideas affect people, and how this should be supported by the platform.

First and foremost, this was a project for hobbyists, by a group of hobbyists. Therefore it is only natural that the entire project will be entire free and open source. But we don't think this is enough.

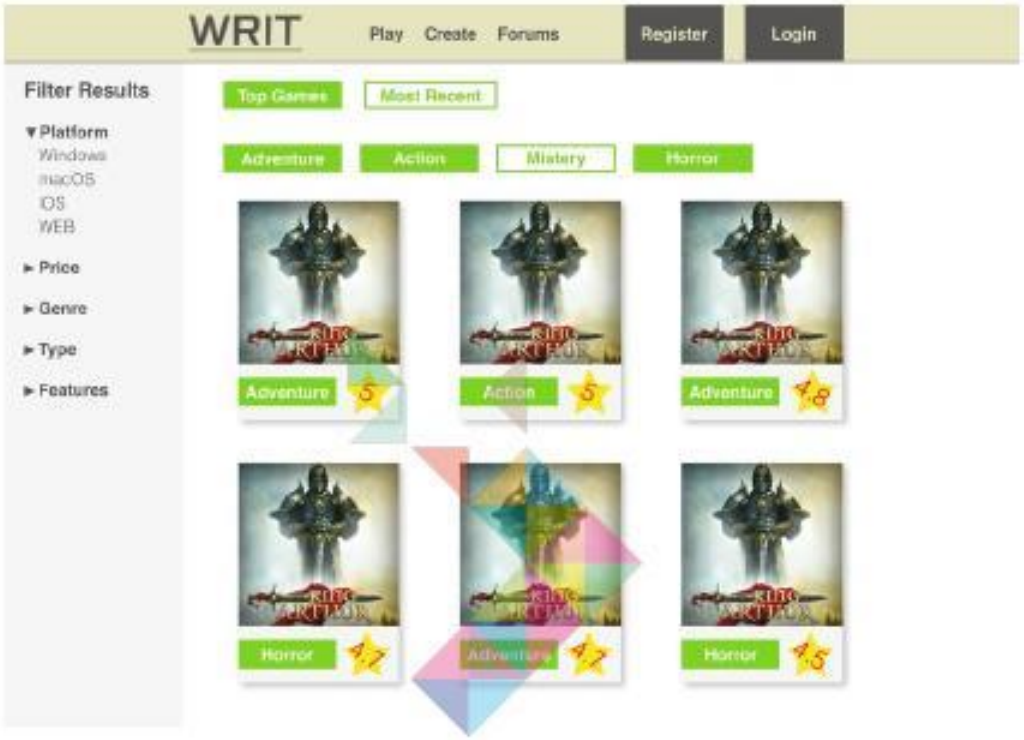
When we considering helping the non-professional dev community, it became clear that not all problems could be solved by software alone. There are problems, such as teaching developers how to properly work together, that cannot be solved by modularity. What we need to solve those is a culture of free good spirited collaboration, which is where the open source community comes in.

But we couldn't just release Writ and leave people to sort themselves out. We've noticed that this system put us in a perfect position to foster this newfound Writ Community from its infancy. Grabbing hold of this opportunity we created the Writ website, and designed it to direct people towards responsible game production habits.

Using (TEXTADEVNTURES.CO.UK,) as inspiration, our site stands as a central platform for users to find games and modules, release their own and discuss what has been going on. Taking advantage of the networking module on the core modules, we've implemented an auto-publish feature that can be accessed through the engine itself or editor. With it users can publish their newly created games to our website with a single

click. This feature is particularly useful since hobby developers usually have very few ways to distribute their games. Putting all games for a platform in a single locations greatly helps with that.(ALEX, 2006) We've also decided to allow people to charge for their hard work, as well as choose a "Pay What You Want" model similar to the one provided by (ITCH CORP,).

Figura 9 – Figure 9: The site for the Writ Platform



Finally, we've decided to take collaborative game creation to a brand new level. When publishing their games to our site, developers are asked if they want to leave the game open for the community. If they choose to close the game, Writ compiles and uploads an encrypted file with all the required modules and content. If they don't the game is uploaded as open source files, that can then be accessed and modified by all community.

Our commitment to open development goes so deep actually, that all Writ content files are human-readable, which greatly helps with sharing the files and allows people to make quick fixes without needing to open the whole editor.

Our site also sports a project hierarchy much like github's, that allows users to create branches of existing projects and work on them. With a single click, Writ downloads the files, sets up the environment, and launches the editor. What this creates is a modding culture that doesn't depend of heavily restrictive APIs deep programing knowledge to create content for. The way we understand modding suddenly changes when anyone is able to create content for their favorite games with minimum effort.

While modding features is nice, we particularly hope to see a lot of writers and artists out there using this feature to mod new content into games they like,.

Ownership. This of course raises the question of ownership. Who really owns the content when it trades hands so freely?

When dealing with open software, ownership becomes more than a matter of whether an downloaded item represents a property. In such situations the authorship itself becomes a matter of dispute. Therefore we've put down a few rules into the design of the site to prevent against plagiarism and violations to fair use.

First, we've made the project branches into a control hierarchy. This way, anyone who creates a branch of your project is subject to your discretion. In cases of merges, or collaborative projects, control defaults to the same level as the highest member involved, but decisions only go through by joint decision. Second, we've made the our system check for old content whenever creating a new project hierarchy, to prevent plagiarism.

However, we've chosen to not extend the same systems towards modules. Game mechanics are tools, and as fall under Fair Use.(STIM,)

Certification. Finally, if we want be certain Writ remains a beginner friendly platform, we must enforce quality control when it comes to the submission of new modules. We've put great care into crafting good, reusable, and well documented code, and we expect module creators to maintain the same level of quality.

First, all modules must come with great documentation. Anything you do in your module must either be explained in code comments or in documentation. The existence of code comments also does not forgo the existence of a documentation, however. Every single function available on your code must have an entry in the documentation, and you must also include a flow diagram for your outermost functions. Finally, a layman tutorial is encouraged, but not required.

Then, all features of your module must be accessible through graphical user interface, as in accordance to our WYGIWUS Rule. Our editor has a very flexible interface using the intermediary language native to Qt, and you have complete access to it from any module by using the editing functions. Modifications to the editor interface must be always additive, and you can never remove interface elements owned by other modules.

At last, once you release, you become responsible for your module. User requests and complaints should be responded to within a time limit, or it's code will become open for any willing to fix it.

Conclusion. At no point this whole system was ever intended for anyone other than the hobby developer. We made it from the ground up thinking on the hobby

community and it wouldn't be right to exclude them from anything.

This is a community based on solidarity and friendship. The willingness to help each other and work together. And the happiness of seeing each other succeed.

This project will ultimately become what the community makes makes of it, and we are under no misgiving that it's fate is in good hands.

7 Resultados e Discussão

This project represents a shift in the way game engines are built. We hope that it's characteristics will be enough to solve or alleviate the problems of Breadth and Depth, but whether or not that actually happens yet remains to be seen.

To trully be able to measure the effects of the introduction of such a tool in an established market requires time that escapes the scope of this paper. The community will need time to grow and to get used to the new model, and it is uncertain how the platform will be accepted by the larger community.

So far we've only openned the platform to a few chosen alpha testers from the community, seeking to further refine our tools before any major releases are attempted. Listening to the feedback of these individuals, we see that our current offerings are favorable to their productivity. Users cite the ease of working on their projects without the need to code as a major boon of the platform. Users also lament the current scarcity of modules.

As noted in Chapter 6, this first foray into our new architecture was limited to a text adventure engine and supporting systems. This means that other than our UIDesigner module and a rudimentary MusicManager module, most interactions with the player are completely text driven. This limits the kinds of games that can be created right now, though not as much as you'd think. Users on the platform already managed to create simple puzzle games like chess and tic-tac-toe through clever uses of the UIDesigner, and one of the current ongoing projects is a tavern management simulator. Regardless, we intend to develop more modules before a full release of the platform, if only to make obvious that this platform is not only limited to text games, but could be used to develop all kinds of games in the future.

8 Future Developments

As mentioned previously, we are not entirely convinced non-realtime applications is the limit of this architecture. There are several angles through which we can possibly solve this problem. One option would be to increase the granularity of each module, the flow inside individual modules easier to predict. Another suggested solution is to better curate a selection of real-time ready modules with greatly optimized code. However at this point these possibilities remain unexplored.

We also hope to soon bring multithreading to the system. We know for certain that it would help with a few of the problems discussed on Chapter 6, and it might bring about other benefits relevant to finally making the jump to real-time. Multithreading will probably also introduce a cadre of new dependence problems. To deal with those we plan to introduce an oversee module to ensure business rules between modules and ward against deadlocks and starvation. This however requires deeper study on how what these business rules will be and how they will be enforced within the architecture.

Finally, we feel like our approach to game development is different enough from what is out there to warrant further study. One particular topic we are interested in is on how this new paradigm will affect the community one year from now, once the platform has gained more traction. It should also be interesting to see which aspects of these results will be due to the implementation of the platform, and which will relate directly to the paradigm presented here.

9 Conclusão

Hobby game development was always at the heart of this project. We feel that for too long this aspect of game making has gone unnoticed and unloved. With this project, we've made an effort to reach out to them and to understand the particularities of their position. We did so not with the intention of creating a product or selling an idea, but of helping those who have little to get a little more out of their effort.

We firmly believe this is the place of academia in the games industry. To look out for the little guys and develop solutions without eyes on profit, and in doing so, better the community as a whole. That is why even if all else proposed in this project fail, we at least managed to bring awareness to the importance of the hobby developer, and that is a reward in its own right.

We see this project not as a great innovation in engine design, but as the culmination of several trends that have been going on for a very long time. In it, we've taken the ideas of modular design, data-driven architecture, and user-centered design and applied them to their fullest extent in the area of engine design, creating the world"s first completely modular and data-driven engine.

More than a creating a software, however, this paper proposes a new way to look at engines. And with this proposition, we hope to cure two major problems that greatly affect the hobbyist game development community: the large expenditure of time in learning new skills and the difficulty to work in teams.

We hope to soon bring the platform live, so we can marvel at the things people are capable of doing together.

More than that, we hope to bring people together, in working towards their dreams, for this is what gaming is all about.

Referências

ALEX. *textadventures.co.uk – the new Quest Games Archive*. 2006. Disponível em: <https://blog.textadventures.co.uk/2007/02/09/new-quest-games-archive/>. Acesso em: 10/12/2017.

ANDERSON, E. F. et al. The Case for Research in Game Engine Architecture. In: FUTURE PLAY '08, 2008, Toronto. Toronto: ACM, 2008. p. 228 – 231. ISSN 978-1-60558-218-4. Disponível em: <https://dl.acm.org/citation.cfm?id=1497031>. Acesso em: 10/12/2017.

ASSOCIATION, E. S. *Analyzing the American Video Game Industry 2016*. 2017.

BLOK-ANDERSEN, R. The Benefits and Challenges of Supporting Third-Party Developers in Eve Online. In: GDC 2015, 2015, San Francisco. *UBM Tech*. San Francisco, 2015.

BLOW, J. Game Development: Harder Than You Think. *Magazine Queue*, ACM, New York, NY, USA, v. 1, n. 10, February 2004. Disponível em: <http://queue.acm.org/detail.cfm?id=971590>. Acesso em: 10/12/2017.

BUSBY, J.; PARRISH, Z.; WILSON, J. *Mastering Unreal Technology, Volume I*. [S.l.]: Sams Publishing.

CHANDLER, H. M. *The Game Production Handbook*. 3. ed. [S.l.]: Jones & Bartlett Learning, 2013. ISBN 1449688098.

CHATFIELD, T. Videogames now outperform Hollywood movies. *The Guardian*, 2009. Disponível em: <https://www.theguardian.com/technology/gamesblog/2009/sep/27/videogames-hollywood>. Acesso em: 10/12/2017.

CHIRONIS, K. From Student To Designer/Writer. In: HTTPS://WWW.GDCVAULT.COM/PLAY/1021994/FROM-STUDENT-TO-DESIGNER-WRITER, 2015, San Francisco. *UBM Tech*. San Francisco, 2015.

COBBETT, R. From shareware superstars to the Steam gold rush: How indie conquered the PC. *PC Gamer*, 2017.

CRYTEK GMBH. *CryEngine*. Disponível em: <https://www.cryengine.com/>. Acesso em: 10/12/2017.

DIGIPEN. *Digipen Course Catalog*. Disponível em: <https://www.digipen.edu/coursecatalog/>. Acesso em: 10/12/2017.

ENTERBRAIN, INC. *RPG Maker*. Disponível em: <http://www.rpgmakerweb.com/>. Acesso em: 10/12/2017.

ENTERTAINMENT SOFTWARE ASSOCIATION. *Games*: Improving The Economy. 2014. Disponível em: <http://www.theesa.com/wp-content/uploads/2014/11/Games_Economy-11-4-14.pdf>. Acesso em: 10/12/2017.

EPIC GAMES, INC. *Unreal Engine*. Disponível em: <https://www.unrealengine.com>. Acesso em: 10/12/2017.

FAILBETTER GAMES. *StoryNexus*. Disponível em: <http://www.storynexus.com/s>. Acesso em: 10/12/2017.

FLOYD, D. et al. *Non-Professional Game Dev - The Joy of Making - Extra Credits*. Disponível em: <https://www.youtube.com/watch?v=m4p7T9O_tqg>. Acesso em: 10/12/2017.

FLOYD, D. et al. *Integrating Academia - Experimenting for Better Games*. 2016. Disponível em: <https://www.youtube.com/watch?v=b6Y6YNhyxOI>. Acesso em: 10/12/2017.

FOERTSCH, G. et al. Killer Portfolio or Portfolio Killer. In: GDC 2017, 2017, San Francisco. *UBM Tech*. San Francisco, 2017.

FOLMER, E.; GROUP, G. E. R. Component based game development: a solution to escalating costs and expanding deadlines? In: CBSE'07 PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON COMPONENT-BASED SOFTWARE ENGINEERING, Medford, MA, USA. Medford, MA, USA: Springer-Verlag. p. 66 – 73. ISSN 978-3-540-73550-2. Acesso em: 10/12/2017.

FULLERTON, T. *Game Design Workshop*: A Playcentric Approach to Creating Innovative Games. 2. ed. [S.l.]: Morgan Kaufmann, 2008. ISBN 0240809742.

GAMEDEV.NET. Disponível em: <https://www.gamedev.net/>. Acesso em: 10/12/2017.

GIANTBOMB. *Colossal Cave Adventure Credits*. Disponível em: <https://www.giantbomb.com/colossal-cave-adventure/3030-19301/credits/>. Acesso em: 10/12/2017.

GIANTBOMB. *Ms. Pacman Credits*. Disponível em: <https://www.giantbomb.com/ms-pac-man/3030-6332/credits/>. Acesso em: 10/12/2017.

GREGORY, J. *Game Engine Architecture*. 2. ed. [S.l.]: A K Peters/CRC Press, 2014. ISBN 1466560010.

GUNDLACH, S.; MARTIN, M. K. *Mastering CryENGINE*. [S.l.]: Packt Publishing.

HAGGIS, M. Studying the value of games at GDC 2017. *Gaming Horizons*, 2017. Disponível em: <https://www.gaminghorizons.eu/studying-the-value-of-games-at-gdc-2017/>. Acesso em: 10/12/2017.

ITCH CORP. *Itch.io Docs - Pricing*. Disponível em: <https://itch.io/docs/creators/pricing#pay-what-you-want-pricing>. Acesso em: 10/12/2017.

ITCH.IO. *The List of Free Games on Itch.io*. Disponível em: <https://itch.io/games/free>.

KANODE, C. M.; HADDAD, H. M. Software Engineering Challenges in Game Development. In: 2009 SIXTH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS, 2009. *IEEE Conference Publications*. [S.l.], 2009. p. 260 – 265.

KENT, S. L. *The Ultimate History of Video Games*: From Pong to Pokemon–The Story Behind the Craze That Touched Our Lives and Changed the World. 1. ed. [S.l.]: Three Rivers Press, 2001. ISBN 0761536434.

KHONSARI, N. *1979 Revolution*: Black Friday. Disponível em: <https://en.wikipedia.org/wiki/1979_Revolution:_Black_Friday>. Acesso em: 10/12/2017.

LLC, S. *Steam Platform games catalog*. Disponível em: <http://store.steampowered.com/search/?sort_by=Released_DESC&tags=-1&category1=998>. Acesso em: 10/12/2017.

NATH, T. Investing in Video Games: This Industry Pulls In More Revenue Than Movies, Music. *NASDAQ.com*, 2016. Disponível em: <http://www.nasdaq.com/article/investing-in-video-games-this-industry-pulls-in-more-revenue-than-movies-music-cm634585>. Acesso em: 10/12/2017.

NEWZOO. *Global Games Market Report*. 2017. Disponível em: <http://resources.newzoo.com/hubfs/Factsheets/Newzoo_Global_Games_Market_Report_Factsheet.pdf?hsCtaTracking=fb39be06-d968-42e5-9671-d0522fc3421f|3fbe5531-c71b-417d-9b11-749d0ccb14f7>. Acesso em: 10/12/2017.

NYSTROM, R. *Game Programming Patterns*. 1. ed. Genever Benning. ISBN 0990582906. Disponível em: <http://gameprogrammingpatterns.com/component.html>. Acesso em: 10/12/2017.

RANKIN, Y.; GOOCH, A.; GOOCH, B. The impact of game design on students' interest in CS. In: GDCSE '08, 2008, Miami, Florida. *ACM*. Miami, Florida, 2008. p. 31 – 35.

ROHRER, J. *Passage*. Disponível em: <https://en.wikipedia.org/wiki/Passage_(video_game)>. Acesso em: 10/12/2017.

ROWLAND, A.; CLARKE, C. Unity Technologies Lands $12 Million in Series B Funding Led by WestSummit Capital and iGlobe Partners. *Market Wired*, 2011. Disponível em: <http://www.marketwired.com/press-release/unity-technologies-lands-12-million-series-b-funding-led-westsummit-capital-iglobe-partners-1540593.htm>.

RUSSELL, S. *SpaceWar!* 1962. Disponível em: <https://en.wikipedia.org/wiki/Spacewar!> Acesso em: 10/12/2017.

SAIL, L. F. *FullSail Course Catalog*. Disponível em: <https://www.fullsail.edu/resources/brochure-file/full-sail-catalog.pdf>. Acesso em: 10/12/2017.

SMITH, Z. *In game development, what's the ratio of time spent programming to non-programming tasks like asset creation? - Quora*. Disponível em: <https://www.quora.com/In-game-development-whats-the-ratio-of-time-spent-programming-to-non-programming-tasks-like-asset-creation>. Acesso em: 10/12/2017.

STENCYL, LLC. *Stencyl*. Disponível em: <http://www.stencyl.com/>. Acesso em: 10/12/2017.

STIM, R. *Fair Use Overview*. Disponível em: <https://fairuse.stanford.edu/overview/fair-use/>. Acesso em: 10/12/2017.

TEXTADEVNTURES.CO.UK. *Textadevntures.co.uk*. Disponível em: <Textadevntures.co.uk>. Acesso em: 10/12/2017.

UNITY TECHNOLOGIES. *Unity Fast Facts*. Disponível em: <https://unity3d.com/public-relations>. Acesso em: 10/12/2017.

UNITY TECHNOLOGIES. *Unity Manual - Execution Order of Event Functions*. Disponível em: <https://docs.unity3d.com/560/Documentation/Manual/ExecutionOrder.html>. Acesso em: 10/12/2017.

UNITY TECHNOLOGIES. *Unity3D*. Disponível em: <https://unity3d.com/pt/>. Acesso em: 10/12/2017.

UNSEEN64.NET. *An archive of lost and cancelled games*. Disponível em: <https://www.unseen64.net/beta-and-cancelled-videogames/>. Acesso em: 10/12/2017.

YOYO GAMES LTD. *GameMaker*. Disponível em: <https://www.yoyogames.com/gamemaker>. Acesso em: 10/12/2017.

ĐÔNG, N. H. *Flappy Bird*. Disponível em: <https://en.wikipedia.org/wiki/Flappy_Bird>. Acesso em: 10/12/2017.