Project Proposal:

Adaptive Programming in the Context of Gameplay Programming in Unreal Engine 4

Joshua Pritchard

University of Huddersfield

N421: BSc (Hons) Computing Science with Games Programming SW

CHP2524: Final Year Project

Dr. Minsi Chen

October 30 2020

Project Proposal:

Adaptive Programming in the Context of Gameplay Programming in Unreal Engine 4

The problem

Background

Educating oneself in the world of programming is a trivial task. The wide array of resources and materials available across any preferred medium allow best practices to be learned from the very start. As a student progresses their understanding, 'specialism' routes begin to be followed. For example, a student may choose to focus their training from a broad, beginner level, to 'low-level programming'. From there, they may again choose to specialize further into 'Operating Systems Programming'.

At each level of focusing, the pool of relevant and (crucially) accessible resources grows smaller. For many disciplines of Programming, this is not an issue. The area of 'Games Programming', on the other hand, suffers at an amplified level (Epic Games, n.d.). Tutorials often fail to venture past basic introductory use cases; resources encourage bad practices; materials are hard to find and often hard to follow. Within such, '***Gameplay** Programming'*, particularly, experiences a worrisome and counterproductive lack of said resources. Consequently, students face issues which this project will explore.

The Problem

The process of learning the design, implementation, and testing of game mechanics is needlessly vague. Their conception/design is often a game designer's task, performed blind of any programmer's input, yet has significant consequences for the implementation – which itself requires performance and efficiency, yet is often taught as a prototyping task for designers (University of Huddersfield, 2020). Their testing is unique, yet all but ignored by the discipline of software testing.

A student learning gameplay programming will typically struggle along, accepting snippets rife with bad practices from multiple (sometimes conflicting) sources. Occasionally, they will, for example, follow outdated tutorials where syntax has changed and will fail at compilation, both hindering the progression of their learning, and creating the necessity for further research and time to be dedicated to comprehending complex compilation errors that just-so-happen to be poorly documented (Epic Games, n.d.). Where a student does succeed in crafting a game mechanic, it will often be plagued with *very* sub-optimal practices and be tightly coupled to the game or project they have created.

All of the above problems have already been solved, often with such comparatively simple concepts as OOP (object-oriented-programming). Yet, little resources exist on the application of these to gameplay programming. Even the possibility of learning from open-source or industry standard techniques seldom exists because of the Games Industry's protectivist approach to in-house technologies and isolationistic behaviour. A visible consequence of such behaviour is the fact that we, as members of the field, are *just now* seeing the release of source code for games that were developed in the *1990's* (Kerr, 2020).

The Aims

This brings us squarely to the question this project will attempt to answer: 'What if there was a framework, collection, library or tool of programmatic elements to make this easier?'

Adaptive programming is a programming paradigm by which pre-written, modular, blocks of code are combined by the leverage of higher level patterns: easy-to-understand, common, recurring 'use-cases' whose core implementations are identical, but differ in terms of details, or 'adaptations' (Cutumisu, et al., 2007).

One of the most recognizable applications of Adaptive Programming is in the form of *ScriptEase*, a scripting tool developed for *NeverWinter Nights*. With a visual interface, it applied the technique to

trivialize the process of creating and individualizing quests for the game, so much so that in a paper written by its creators, high school students with *no prior programming experience* were able to create unique quests in 1-2 hours (Cutumisu, et al., 2007). For Designers, this was a way to quickly and easily create content; for the Gameplay Programmers, full access to the generated[1] source code is given, to which further unique modifications can be made and, perhaps most importantly, understanding and learning of the techniques used could be gained.

Inspired by the advancements this tool made in the field of quest creation, the first aim of this project is to develop a prototype for a similar tool, to be used for the creation of game mechanics, instead of quests.

The second and third aims are based around the educational element of said tool prototype. The second aim is to make it as easy as possible to create a generic, high level, unique game mechanic; the third is to adhere to best practice principles both in the context of the Unreal Engine framework, and of programming in general.

From the interests of knowledge sharing can the fourth aim be derived: To prove that an open source, large scale implementation of this tool would be both feasible and useful to games education, and to the industry.

[1] This term is used sparingly here, as *Generative Programming* is a unique paradigm of its own, and is only a partial focus of this project.

The Product

As stated by the first aim, this project will produce a prototype to a potential, larger tool. This will be formed of 4 significant components: A visual interface, a generative programming frontend with parser, a collection of pre-made, generic, hierarchical game mechanic components, written in & for Unreal C++, and individualized usage guides.

The visual interface component will be a simple, easy to understand, easy to use, GUI based tool that will serve as a boundary interface to the generative programming frontend[2]. An initial use-case to which this will be designed, is of an 'installation wizard' (where that might be applied to constructing a gameplay mechanic).

The generative programming frontend will be JSON editing, for which a standard editor can be used to edit generic files. Their content will resemble a basic, high-level programming language for creating game mechanics, of which the syntax will encompass commonly created 'patterns' of game mechanics. Said patterns will then be parsed into calls to the Unreal Engine Editor Scripting API, such that the mechanics specified will be created within the editor instance, using the pre-written components.

Said collection of components will be pre-written classes. These will be written to adhere to, and force adherence of, Unreal Engine's 'Gameplay Framework' (Epic Games, n.d.). Each individual class will fall within the context of an abstracted, hierarchical, OOP approach to game mechanics as code, which itself will largely adhere to the design patterns detailed by Nystrom (2014). As this is a prototype tool, and incapable of dealing with every possible game mechanic, a small 'vertical slice' of mechanics will be developed. The specific vertical slice focused on will be that of commonly recurring mechanics within (usually top-down or isometric) dungeon crawlers.

[2] The tool will be designed in a way that does not mandate GUI usage. The GUI simply extends the generative programming paradigm used – from a constructive form, to an adaptive one.

The usage guides will be small information packets, created alongside the generation of the Unreal Engine content. Time constraints mean the packets produced will lack flow and potentially contain grammatical errors, but will provide informative 'metadata' like descriptions on: The Unreal components used, the Unreal framework they interface with, the programming patterns/techniques used (and why), and where adaptations can be made within the generated mechanic (out of code).

The goal use-case this prototype will satisfy will be as follows:

1. Specify generic mechanic pattern (User)
2. Create mechanic structure within Unreal (System)
3. Individualize mechanic with adaptations (User)

Potential Audience

The primary audience of both this prototype, and the proposed larger tool, is Trainee Gameplay Programmers. This can be applied to any group, provided they fit the defining quality: 'is learning how to create gameplay mechanics'. Due to the tool's proposed adherence to Unreal Engine's Gameplay Framework and Nystrom's Patterns (Nystrom, 2014; Epic Games, n.d.), the generated code will be considered 'correct' in the context of Unreal Engine's framework. Therefore, said code will have an inherent value to the education of this audience, teaching best practices first. As a note for the future, if the proposed larger tool was developed and the collection made open source, the generated code could eventually be considered peer-reviewed, qualifying it in the eyes of more doubtful critiques.

Additionally, the larger tool would have significant value in the discipline of prototyping. Assuming that a number of game prototypes are extended to produce full games, and that the prototyper(s) will typically produce 'less' correct code in favour of rapid iteration, it is reasonable to state that these 'flaws' get carried through to full scale releases in some cases. The larger tool could allow prototypers to produce more standardized, 'correct' frameworks upon which final games are then built.

Research

Unreal Engine's *Blueprints* visual scripting system can be considered a step towards making game mechanic creation accessible to non-programmers (Epic Games, n.d.; Salonen, 2019). The addition of a 'nativization' tool, by which blueprints can be converted to C++ code automatically, represents another big step, in this case bridging the gap between performance issues originating from the difference between interpreted visual scripting languages (blueprints), and compiled native languages (C++). Despite the advancements blueprints represents, it is still considered a constructive programming language by the works of Cutumisu et al. (2007), utilizing small blocks of pre-written native C++ code instead of the large, high level patterns this project will deal with. Examples of Unreal Engine demonstrating adaptive programming concepts in other places are discussed shortly.

Related Work

Many studies have encountered the same problems that this project will deal with. The development of *BPAIt* tackled an implementation of a tool to allow blueprint implementations to be switched between and merged easily, to solve the issue of iterating and prototyping over multiple choices in implementation (Chu & Zaman, 2021). The key difference here is that the burden of implementation (constructive) is still placed upon the user of the tool.

The works of Geisler (2019) in *GAMESPECT* identify the issues with domain specific languages and the re-writing of identical code across multiple games, while applying their solutions to solving the cross-cutting issue of game balance across different game engines.

A study by Saini & Guzidal (2020) deal with the issues of game programming difficulty and no-code mechanics creation by applying a rule-learning AI to implicitly infer a mechanic from 'frames' of a game illustrated to the AI by the user.

Work into a '*State Explorer*' tool by Volkovas et al. (2020) supports the notion of a lack of supporting game design tools, however presents a tool intended to be used as a support device, rather

than a starting point. In their case, it is already assumed that the user has knowledge of implementation details.

A very similar piece of work is done by Capasso-Ballesteros & De La Rosa (2020) in the area of automatic generation of video games using explicitly defined rules, leading to emergent gameplay. The study identifies many of the same issues with the lack of development in the field of mechanics and rules creation, however does not touch on the 'learning' issue that this project does. Therefore, the mentioned study focuses on collaboration between disciplines, rather than an individual learning tool.

A study citing the already mentioned ScriptEase paper, authored by Sarinho & Apolinário (2009), proposes a generative metaprogramming language applied to the generation of entire games. While the scope of this study is far out of the bounds of this project, the provably applied techniques of 'Game Specification Languages' parsed to generate real code are crucial to this project, and will be taken as direct inspiration and reference.

Finally, a study into implementing game mechanics with 'gang-of-four' programming patterns identifies the exact issue with reusability and standardization that this project does, even going so far as to directly quantify the lack of adherence within released games (Kounoukla, Ampatzoglou, & Anagnostopoulos, 2016). Within, the authors directly map the applicability of unique game mechanics to design patterns. This directly supports the works of Nystrom (2014), and as such the details of their recommendations will be studied and used wherever possible.

Literature Review

A significant technical aspect of this project is the collection-code's adherence to and correctness within Unreal Engine's framework. As such, relevant parts of Unreal Engine's documentation and source code will be studied, with the aim of forming 'rules' and 'tenets' by which to abide[3]. Similarly, the writings of Nystrom (2014) will be observed, adhering to the re-usable, extendable and standardized patterns the author puts forth. Both of these tasks fall under the broader scope of 'Software Engineering/Architecture' and Gameplay Programming research, which will be carried out to support and underpin the proposed 'correctness' of the developed code.

The greatest technical challenge to this project is applying the correct conventions of Generative and Adaptive programming with respect to gameplay programming. Few studies have been produced on this combination, therefore where directly applicable material is unavailable, 'bridges will have to be built' between the two. Additionally, some cursory research may be done into 3C's (Character, Controls, Camera) and Game Feel literature, however only in the context of gaining insight into typical adaptation options required by the game mechanics developed.

Some entry level UI/UX design may be reviewed as a consequence of the inclusion of a visual interface, but this is expendable and dependent on time constraints.

Finally, as this is a prototype project, an ongoing review of methods by which this tool can be proved feasible at a larger scale will be conducted. This will begin with a review of Software Metrics, and some investigation of code evaluation software, but will be non-exhaustive.

[3] this process could be considered fact finding within the documentation.

Visual Interface Design and Development

The visual interface will tackle the problem of creating a simple boundary interface with which the Unreal Engine Editor can be abstractly controlled. It's design and wording has to be simple enough that non-programmers can use and understand it easily, however rich enough that the user can reach their 'end goal mechanic' with a reasonable number of adaptations. Such considerations draw significant parallels to Cutumisu et al.'s discussions around ScriptEase's available list of patterns, and where the line is drawn between a constructive and an adaptive language (2007).

The second aim necessitates that this boundary interface tool be as easy to use as possible. Most of the design of this will be relegated to possible future work due to time constraints. However, in undertaking basic UI/UX research, existing guidelines and proven methods can be followed to completely minimize the training needed to use the tool. For instance, a brief thought into how a common 'installation wizard' UI might be applied to constructing game mechanics yields useful information as it would relate to a proposed use-case, such as the familiarity of its interface design. A hierarchical interface similar to ScriptEase's, detecting and keeping track of existing mechanics, allowing in-tool adaptations to be made will be pursued, but in the event of unfeasibility, will be dropped in favour of using Unreal Engine's exposure methods. This tool will seek to make cuts in the learning times cited by Cutumisu et al. in their case studies of ScriptEase (4 hours) (2007).

Providing *incredible* convenience, Unreal Engine 4.25 has added an experimental feature: 'Scripting the Editor using Python' (Epic Games, n.d.). Given Python's ease of use and wide range of applications (even within the generative languages to be discussed), this project will use said scripting tool for the visual interface. This will additionally serve to minimize the number of interlocking technologies and simplify the implementation (as best as possible).

Generative Programming Frontend Design and Development

This component of the project deals with creating a high level, human readable specification for a game mechanic, much in the same way that ScriptEase specifies 'Encounters' (Cutumisu, et al., 2007). The goal of this section is that a specification instance can be translated, by way of a python parser, into a list of API calls that can be passed to the Unreal Engine Editor. The order in which these commands are executed will be very important, so as to not create problems with the running Editor instance. It can be taken such that the order in which the commands are executed must mirror (to some extent) the workflow by which a human user would create the same end product. Take, for instance, the collision system in Unreal. While a very basic example, attempting to access a 'collision channel' before it has been defined (the editor is unaware of it), can lead to compilation issues. Having to resolve these manually would be entirely counterproductive to aims number two and four, therefore it is crucial that this sequence is generated in the correct order.

This project will seek to define a JSON specification, such that a mechanic's core code and most (if not all) of the adaptations desired can be generated from the information contained within a single instance. This is the application of generative programming that cannot be avoided in the context of this project, as for there to be any code to adapt, it must first be present. Seeing as to adhere to aim two, coding must not be necessary to create a game mechanic, this code must be generated. Such a specification will also make an instance of a single game mechanic platform agnostic, which for aim four is an important qualifier for scalability and expansion to multiple game engines. A crucial consideration of this project proposal is to understand that the JSON specification and parser produced will in no way match the quality of implementation that a dedicated team could. In order to work to time constraints, the work within this section will be prioritized towards the overall aim of producing the tool, not producing the most foolproof specification model.

Component Collection Design and Development

The most crucial component of this project is the code collection from which the mechanics are generated. This is a pre-written set of mechanics, within which adaptations can be easily made, preferably through generative methods, but definitely from within code. The abstracted, object-oriented, hierarchy of mechanics must be written in the best possible way[4]. To do this, it will closely follow Nystrom's patterns, Unreal Engine's best practices, and Unreal Engine's Gameplay Framework, ascertained from the literature review (Epic Games, n.d.; Nystrom, 2014). It is important to note that the created code can in no way adhere to every single possible best practice of: C++, Unreal Engine, Programming Patterns, etc. Such an undertaking comprises decades of experience, of which making the future tool's code open source would deal with.

The hierarchy of mechanics developed will be focused around that of a top down/isometric dungeon crawler. Instead of a single game, the set of gameplay mechanics developed for will encompass the *genre*. By doing this, the success of the tool will prove that the same base code can be adapted to make the mechanics seen in multiple games of that (dungeon crawler) genre.

Unreal Engine is quite unique in that it provides users with a large set of template resources upon which they are expected to build. Nowhere is this more obvious than with the `ACharacter` class. Epic Games recommend against re-writing this class or its components (such as the `UCharacterMovementComponent` class), instead directing users to adapt these given resources to suit their own needs, building on top of where necessary. This can be considered close to adaptive programming, given that the options provided are sufficient for making unique character axis movement. Because of this, the set of mechanics developed will be extensions to the framework already provided, such as additional movement mechanics (like a dash), or the framework for a combat system.

[4] The idea of 'best implementation' is loosely used, as the future work would make the collection code entirely open source, effectively peer reviewing the code in best practices.

Usage Guides Design and Development

This final component is the least technical of the project, and will simply aim to provide a small amount of written information alongside the generated mechanics code. Most of the reason for this component's inclusion is in aim two. Taking into account the non-programmers and learning students this tool is aimed at, the areas of Unreal Engine's framework that are utilized in the generated code (such as the collision system) must be explained, albeit briefly, so that the user understands how and why the code was written in that way. The code itself must also come with explanation regarding the decisions made, the techniques used, and where to make adaptations.

This will be implemented in two sections. Firstly, the code collection must be commented sufficiently so that readers can understand it (self-documenting code). The second is summaries or snippets of information from Unreal Engine's documentation, combined with specific usage details, into small .txt files that the user can read to understand what piece of Unreal Engine's framework is being used, and why. This can be done through metadata tags on the code files used, which can pull existing text information, and fill in blanks with specific details about the adaptations made.

It is worth noting that this component and the code collection are aimed at enforcing a didactic teaching method upon the students learning from this tool, in that it is trying to teach them what to do and what they cannot, in a very structured way (Leon-Henri, n.d.).

Testing

The implementation testing of this prototype tool is centered around assurance of functionality. Simply: the tool must work correctly for all use cases, implying the most basic testing acceptance of compilation success and use case acceptance. That being said, the tool is made of 4 distinct components, all of which must interface together correctly to form the product as a whole. Because of this, each of the components must be tested separately to ensure their isolated functionality.

The visual interface must be proven to generate the correct JSON instances in every case, and of that, parse a corresponding and valid Editor API-call sequence. From there it must be proven that the call sequence, were it processed, will generate the mechanics code correctly. The aforementioned must also take into account other mechanics already generated by the tool, creating checks to make sure existing generations or adaptations will not be clobbered. The code collection must be tested to ensure that all valid adaptations function correctly within the engine. Additionally, the information packets must be proven to be correct and factual for all adaptations. The scale of testing that must be done to prove the usability of an open-ended tool like this further necessitates the limited set of mechanics with which the prototype will deal.

User Testing

User testing is inherently inaccurate in small and non-diverse samples, however given its user oriented design philosophy, cannot be ignored for proving feasibility of this prototype's larger tool. Even a small sample of user testing will be invaluable in either proving this concept's effectiveness, or pointing out flaws in its methodology. Both of these outcomes would be valuable in a conclusion and recommendation for further work upon closure of this project.

The most appropriate form of user testing for this prototype will be in the form of single user use cases. Participants will be asked to use the tool to create, from a pre-written specification, a mechanic of the set dealt with by the prototype. The first metric here is whether the subject succeeds or not. They will then be asked to study the information generated, and to answer questions regarding the components of Unreal Engine's framework that were used, and about the decisions made within code. They will then be asked to make adaptations to the mechanic, to further its uniqueness. Depending on their level of programming skill, subjects will be prompted towards making more complex adaptations, in addition to using the adaptation tools developed.

Evaluation

User testing and proof of correctness are important parts of this prototype, however the technical side of the project is greatly hinged on whether or not the code produced is actually useful to anybody. For students, the code must represent examples of good practices, and serve as an example to follow when constructing their own mechanics. For industry, if any prototype is to build upon the generated mechanics, the quality of the code must at the very least be comparable to or useful to a commercial game.

The framework and pattern adherence discussed prior deals with the student's perspective, however even that also hinges on the industry's expectations that their junior hires must be able to produce the quality of code they expect. If a student learns practices that lead to objectively bad or severely inferior code, then the tool has failed.

Two separate methods will be used to evaluate the quality of the generated code. The first is with Software Metrics, 'A measure of some property of a piece of software or its specifications' (SQA, n.d.). While the area of software metrics is vast, a few individual metrics can be cherry picked by order of relative importance to this project. It is important to note that many simplistic software metrics are counterproductive to the end goal of software evaluation, as they don't measure what's important to said software (Kaner & Bond, 2004). With this reasoning, and to satisfy the educational element of this project, *Complexity* is the most relevant metric. Within that, both *Halstead* complexity and *Cyclomatic* complexity will be considered. Furthermore, *Cohesion* will be considered, both to ensure adherence to Unreal Engine's existing modules, and to consider the modules from which the generated code is sourced from (aivosto, n.d.). Microsoft's Visual Studio provides tools for calculating these metrics, of which the *Maintainability Index, Cyclomatic Complexity,* and *Class Coupling* will be crucial (Microsoft, 2018).

The second method of code evaluation can be considered to be more so a development tool, but has the potential to be used for evaluation. *Static Analyzers*, such as 'PVS Studio' and 'Klockwork' function

by analyzing source code repositories as a background process, alerting the user to any potential problems (PVS-Studio, n.d.). From there, the issues can be evaluated and addressed by human programmers, as is routinely carried out on Unreal Engine's own source code (Karpov, 2017; Greschishchev, 2019; Eremeev & Razmyslov, 2015). *ReSharper C++*, developed by *JetBrains*, also performs similarly (JetBrains, n.d.). On their own, these three pieces of software are simply development tools to achieve the quality desired for the collection codebase, an albeit just-as-crucial aim for this prototype. However, their evaluation potential for the feasibility-proving of this project is via *comparison*.

Unreal Engine's Blueprints and its nativization tool, already discussed, represent a major step forward for the accessibility of creating gameplay mechanics for non-coders. A crucial question to answer would be whether the project-generated code could match (or beat) the nativization-generated code, in terms of metric scores and problem detections. Subsequently, if it does not, then it must be asked whether this issue could be remedied with the peer-reviewing provided by the larger tool being open-source. Future work could involve analyzing an entire commercial game's source code with these methods, and comparing it in the same way to the same mechanics generated with the larger tool. For the purposes of this project, the methodology will concern replicating the code of the generated mechanic in blueprints, and then nativizing it to obtain the comparison code.

Overall Time Plan

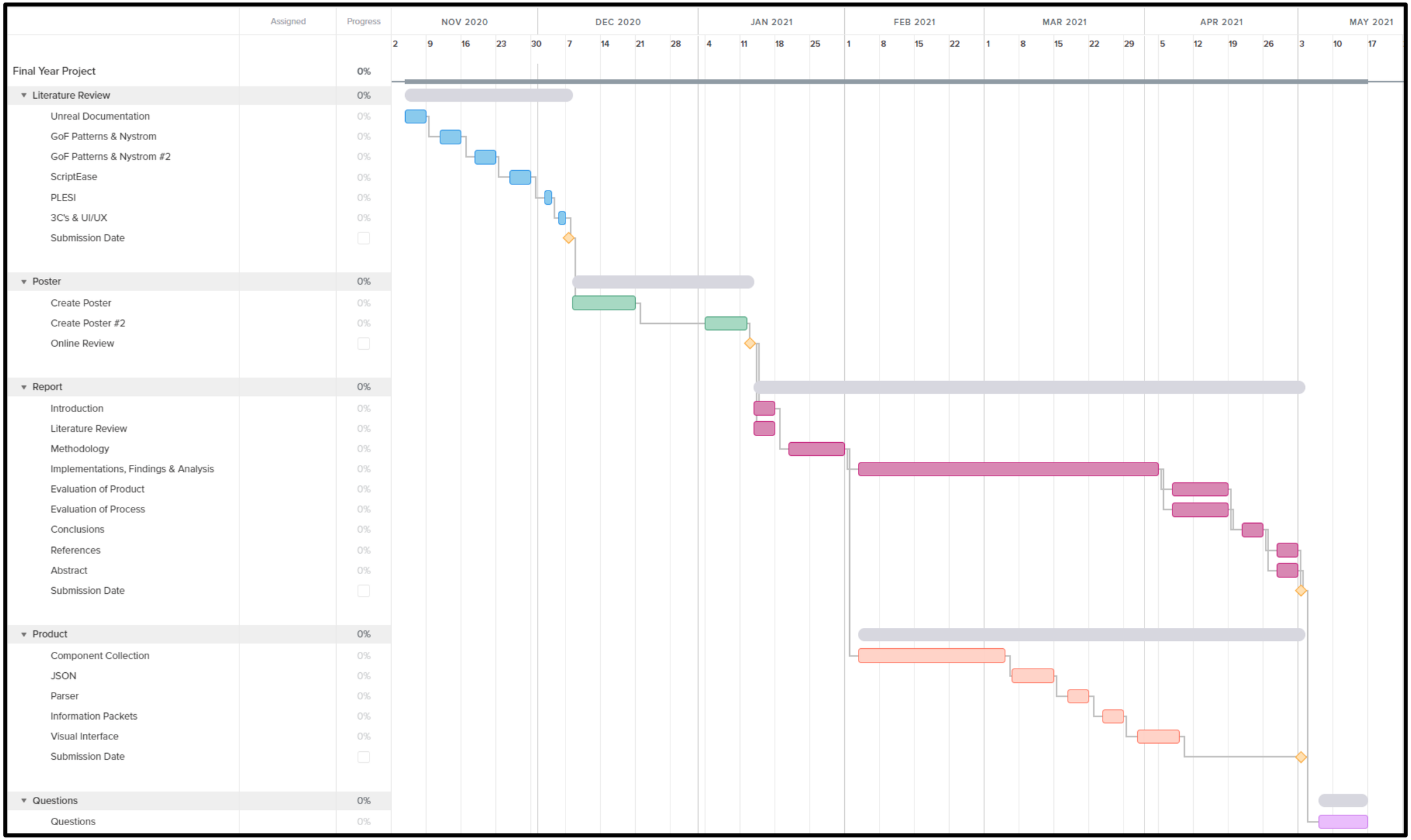

Figure 1: Overall Time Chart

Literature Review Time Plan

The literature review will cover other sources than the ones listed here, however these are some of the crucially important ones that will be covered.

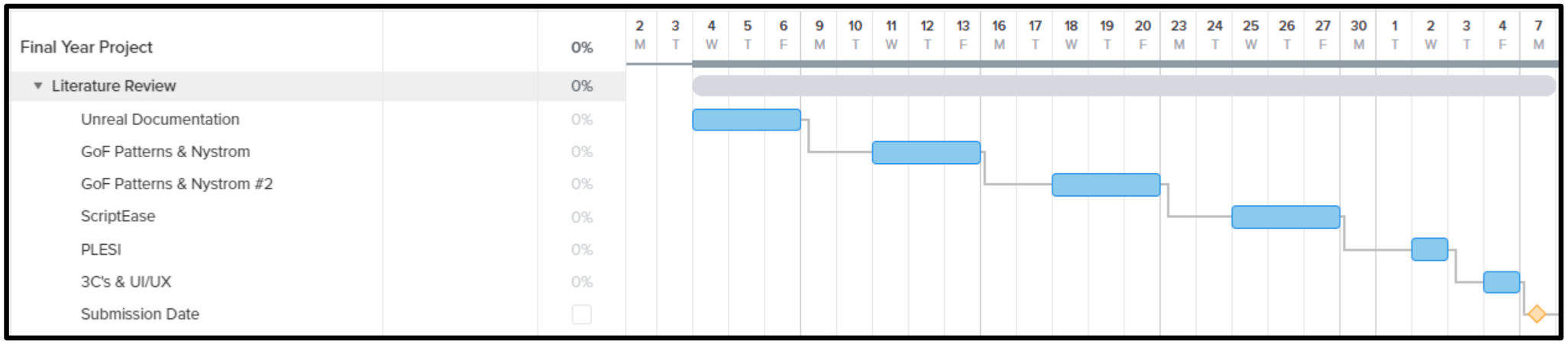

Figure 2: Literature Review Time Chart

Poster Time Plan

This period is subject to review based on available time, and may incorporate additional literature review work.

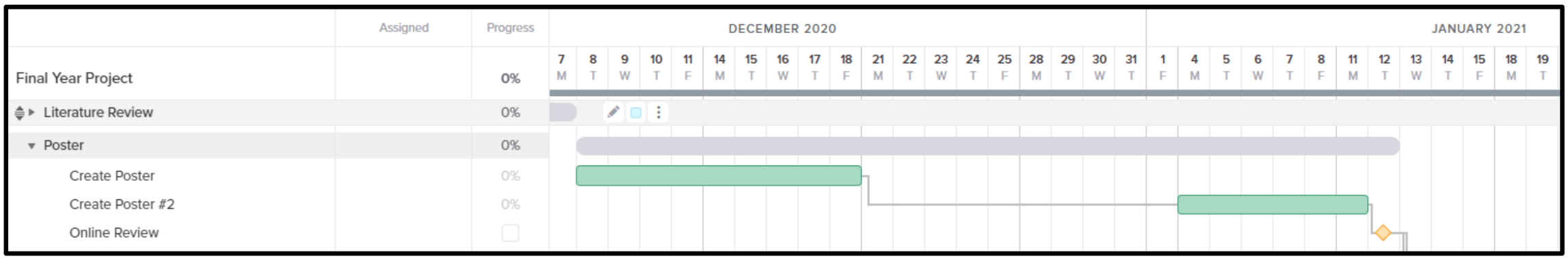

Figure 3: Poster Time Chart

Report and Product Time Plan

It is likely that preliminary tests will be done during the initial report time allocation. This entire period is subject to likely re-evaluation upon receipt of time tables for the *Team Project* module.

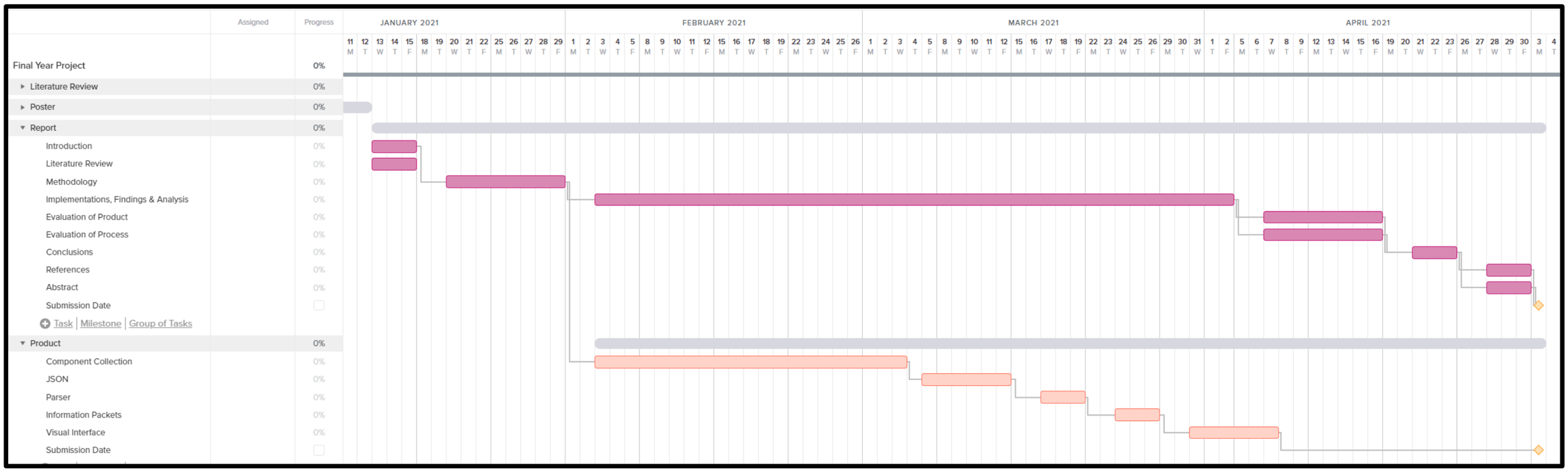

Figure 4: Report and Product Time Chart

References

aivosto. (n.d.). *Cohesion Metrics*. Retrieved from aivosto: https://www.aivosto.com/project/help/pm-oo-cohesion.html

Capasso-Ballesteros, I., & De La Rosa, F. (2020). Semi-automatic Construction of Video Game Design Prototypes with MaruGen. *Revista Facultad de Ingeniería Universidad de Antioquia*.

Chu, E., & Zaman, L. (2021). Exploring alternatives with Unreal Engine's Blueprints Visual Scripting System. *Entertainment Computing*.

Cutumisu, M., Onuczko, C., McNaughton, M., Roy, T., Schaeffer, J., Schumacher, A., . . . Gillis, S. (2007). ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 32-58.

Epic Games. (n.d.). *Gameplay Framework*. Retrieved from Unreal Engine: https://docs.unrealengine.com/en-US/Gameplay/Framework/index.html

Epic Games. (n.d.). *Introduction to Blueprints*. Retrieved from Unreal Engine: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html

Epic Games. (n.d.). *Multiplayer Programming Quick Start Guide*. Retrieved from Unreal Engine: https://docs.unrealengine.com/en-US/Gameplay/Networking/QuickStart/index.html

Epic Games. (n.d.). *Scripting the Editor using Python*. Retrieved from Unreal Engine: https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/Python/index.html

Epic Games. (n.d.). *Unreal Engine 4 Documentation*. Retrieved from Unreal Engine: https://docs.unrealengine.com/en-US/index.html

Eremeev, P., & Razmyslov, S. (2015, June 20). *How the PSV-Studio Team Improved Unreal Engine's Code*. Retrieved from viva64: https://www.viva64.com/en/b/0330/

Geisler, B. (2019). *GAMESPECT: A Composition Framework and Meta-Level Domain Specific Aspect Language for Unreal Engine 4.* Nova: College of Engineering and Computing Nova Southeastern University.

Greschishchev, M. (2019, July 22). *Unreal Engine Game Development Meets Static Code Analysis Tools*. Retrieved from Perforce: https://www.unrealengine.com/en-US/tech-blog/static-analysis-as-part-of-the-process?sessionInvalidated=true

JetBrains. (n.d.). *ReSharper C++*. Retrieved from jetbrains: https://www.jetbrains.com/resharper-cpp/

Kaner, C., & Bond, W. P. (2004). Software Engineering Metrics: What Do They Measure and How Do We Know? *International Software Metrics Symposium.* IEEE.

Karpov, A. (2017, June 26). *Static Analysis as Part of the Development Process in Unreal Engine*. Retrieved from Unreal Engine: https://www.unrealengine.com/en-US/tech-blog/static-analysis-as-part-of-the-process?sessionInvalidated=true

Kerr, C. (2020, May 21). *EA is releasing the source code for Command & Conquer: Red Alert and Tiberian Dawn*. Retrieved from Gamasutra: https://www.gamasutra.com/view/news/363396/EA_is_releasing_the_source_code_for_Command__Conquer_Red_Alert_and_Tiberian_Dawn.php

Kounoukla, X.-C., Ampatzoglou, A., & Anagnostopoulos, K. (2016). *Implementing Game Mechanics with GoF Design Patterns.*

Leon-Henri, D. D. (n.d.). *Difference Between Didactics and Pedagogy*. Retrieved from Reflective Teaching Journal: https://reflectiveteachingjournal.com/difference-between-didactics-and-pedagogy/

Microsoft. (2018, November 2). *Code metrics values*. Retrieved from Microsoft: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019

Nystrom, R. (2014). *Game Programming Patterns.* Genever Benning.

PVS-Studio. (n.d.). *PVS-Studio Analyzer*. Retrieved from viva64: https://www.viva64.com/en/pvs-studio/

Saini, V., & Guzidal, M. (2020). A Demonstration of Mechanic Maker: An AI for Mechanics Co-Creation. *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (pp. 325-327). AAAI Press.

Salonen, A. (2019). *User Experience Testing of Visual Scripting.* Turku: Turku University of Applied Sciences.

Sarinho, V. T., & Apolinário, A. L. (2009). A Generative Programming Approach for Game Development. *VIII Brazilian Symposium on Games and Digital Entertainment* (pp. 83-92). Rio de Janeiro: IEEE.

SQA. (n.d.). *Software Metrics*. Retrieved from SQA: http://www.sqa.net/softwarequalitymetrics.html

University of Huddersfield. (2020). *Games Development (Design) BA(Hons)*. Retrieved from University of Huddersfield: https://courses.hud.ac.uk/2021-22/full-time/undergraduate/games-development-design-ba-hons

Volkovas, R., Fairbank, M., Woodward, J. R., & Lucas, S. (2020). Practical Game Design Tool: State Explorer. *2020 IEEE Conference on Games (CoG)* (pp. 439-446). Osaka: IEEE.