# Video Game Bad Smells: What they are and how Developers Perceive Them

VITTORIA NARDONE, University of Sannio
BIRUK ASMARE MUSE, Polytechnique Montréal
MOUNA ABIDI, Polytechnique Montréal
FOUTSE KHOMH, Polytechnique Montréal
MASSIMILIANO DI PENTA, University of Sannio

Video games represent a substantial and increasing share of the software market. However, their development is particularly challenging as it requires multi-faceted knowledge, which is not consolidated in computer science education yet. This paper aims at defining a catalog of bad smells related to video game development. To achieve this goal, we mined discussions on general-purpose and video game-specific forums. After querying such a forum, we adopted an open coding strategy on a statistically significant sample of 572 discussions, stratified over different forums. As a result, we obtained a catalog of 28 bad smells, organized into 5 categories, covering problems related to game design and logic, physics, animation, rendering, or multiplayer. Then, we assessed the perceived relevance of such bad smells by surveying 76 game development professionals. The survey respondents agreed with the identified bad smells, but also provided us with further insights about the discussed smells. Upon reporting results, we discuss bad smell examples, their consequences, as well as possible mitigation/fixing strategies and trade-offs to be pursued by developers. The catalog can be used not only as a guideline for developers and educators but also can pave the way towards better automated tool support for video game developers.

CCS Concepts: • **Software and its engineering → Software design engineering**; *Software evolution.*

Additional Key Words and Phrases: Video games, Bad Smells, Q&A Forums, Empirical Study

**ACM Reference Format:**
Vittoria Nardone, Biruk Asmare Muse, Mouna Abidi, Foutse Khomh, and Massimiliano Di Penta. 2021. Video Game Bad Smells: What they are and how Developers Perceive Them. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 36 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

The video game industry is rapidly expanding, and it represents a significant share of the software development market [18]. It has been estimated that it will reach ≃ 257 billion dollars by 2025 with over 2.5 billion people playing games[1]. Developing video games requires very specific skills and knowledge, often going beyond the common knowledge of a developer working on conventional software. In particular, video game development is multi-disciplinary, and this requires to involve

[1]https://techjury.net/blog/gaming-industry-worth

Authors' addresses: Vittoria Nardone, vittoria.nardone@unisannio.it, University of Sannio, Benevento, Italy; Biruk Asmare Muse, biruk-asmare.muse@polymtl.ca, Polytechnique Montréal, Montréal, QC, Canada; Mouna Abidi, mouna.abidi@polymtl.ca, Polytechnique Montréal, Montréal, QC, Canada; Foutse Khomh, foutse.khomh@polymtl.ca, Polytechnique Montréal, Montréal, QC, Canada; Massimiliano Di Penta, dipenta@unisannio.it, University of Sannio, Benevento, Italy.


1049-331X/2021/1-ART1 $15.00
https://doi.org/10.1145/1122445.1122456

different types of competencies, related to artists, modelers, musicians, domain experts (as in the case of realistic simulator games), and programmers themselves. In terms of software development, the required skills vary from software architecture and design to computer graphics and artificial intelligence. On top of this, despite the increasing market share of this field, and the proliferation of specific curricula and courses on the topic, game development practices do not represent a "mainstream" in computer science education yet.

When developing a video game, programmers have to cope with different aspects including capturing players' inputs, reproducing/simulating the environment's physics, animating objects, rendering special effects, and even synchronizing different peers in an online, multiplayer game. All these elements, along with their combinations, require developers to make complex design and implementation decisions, that may have a positive or negative effect on different properties of the produced software. These include maintainability (*e.g.,* a game may become difficult to be extended with downloadable contents - DLC), security (online games could be hacked), and, above all, performance and game experience.

While previous literature on video game design [10, 45] and the application of design principles to video games [4–6, 16, 31, 48] exist, and while there have been attempts to identify video game development patterns [45] and some anti-patterns with related detection tools [9], there does not exist a consolidated body of good and bad practices, available for instance for object-oriented development [17]. Moreover, as pointed out by Khanve *et al.* [28], conventional code smells fail to capture all quality problems of video game source code.

To bridge such a gap, this paper empirically analyzes the bad video game development practices that are discussed by developers on forums, to systematically derive a catalog of bad smells to be avoided.

First, we identified data sources. Differently from other development practices, generalist Question & Answer (Q&A) forums such as Stack Overflow may only represent a niche of discussions. Therefore, we included in our set of sources a wide range of 13 forums (plus a direct search on the Google Search engine), including several video game development-specific ones, *e.g.,* the Unity forum, the Unreal forum, or other Q&A forums for game developers like Game Development Stack Exchange. Then, we defined a set of queries to retrieve candidate posts, executed them against forums and search engines, and performed a pruning and preprocessing of the results. We then extracted a randomly-stratified sample of 572 posts, using the forums as strata. Based on such a sample, we manually analyzed the posts using a card sorting [51] procedure, with the aim of (i) identifying video game-specific bad smells, and (ii) grouping them into meaningful categories. To validate the obtained bad smells, we have surveyed 76 video game professional developers, which provided us with (i) their agreement about the considered smells as well as additional comments about them, and (ii) pointed out possible smells that were not identified in our analysis of forum posts.

As a result, we obtained a taxonomy of 28 bad smells, grouped into 5 first-level categories, related to game design and logic, multiplayer, animation, physics, and rendering. For each bad smell, we provide a description, examples, possible consequences, ways to fix the problem, and possible trade-offs that can occur (*e.g.,* a fix may achieve better performance while degrading the game look-and-feel).

Other than serving as guidelines for developers, the collected bad smells can be used to develop better tool support for developers. While previous work has shown how it is feasible to develop video game-specific linters, the availability of the bad smell catalog presented in this paper would allow broadening the range of problems identified by such linters. Educators can also leverage this catalog to teach aspiring game developers about bad practices to avoid.

The work replication package is available online [43]. It comprises (i) the list of downloaded posts and the sampled ones, (ii) sheets from the various steps of the open coding, (iii) the survey questions, (iv) the survey results, and (v) the devised catalog of bad smells following a template that makes it usable by practitioners.

The paper is organized as follows. Section 2 describes the methodology followed to mine forums and devise the catalog of bad smells. The obtained catalog is presented and discussed in Section 3, while its implications are highlighted in Section 4. Section 5 discusses the threats to the study's validity. Then, Section 6 discusses the related literature, while Section 7 concludes the paper.

## 2 STUDY DEFINITION AND DATA EXTRACTION METHODOLOGY

The *goal* of this study is to identify bad smells in video game development and assess the perceived relevance of such bad smells. The *perspective* is that of researchers interested to create a catalog of video game bad smells and, for future works, to develop monitoring tools aimed at automatically identifying these bad smells, whenever this is possible. The *context* of the study is 572 discussions gathered from 13 Q&A forums, directly or indirectly related to video game development.

We address the following two research questions (RQs):

> **$RQ_1$:** *What kind of video game development bad smells do developers discuss on Q&A forums?* We are interested in studying if video game developers discuss video game development bad smells in discussion forums. We also want to investigate what kind of bad smells are debated because considered controversial. This research question aims to create a catalog of video game development bad smells.
>
> **$RQ_2$:** *How are those bad smells perceived by professionals?* We want to investigate if the identified bad smells are relevant for professional video game developers. This research question aims to assess the perceived relevance of the bad smells identified in $RQ_1$, and to perform an external validation of the defined catalog.

Fig. 1 depicts the study methodology. First, we identified forums where we should mine discussions related to game development bad smells. Also, we defined queries to search for candidate relevant discussions in such forums. Then, we retrieved candidate discussions. Since some forums have limited querying capabilities, we further refined them through local preprocessing, *i.e.,* we filtered discussion text considering only those that actually contained both selected keywords. After, we performed multiple rounds of open coding until we reached saturation. In each round, we (1) first sampled discussions, (2) then, four evaluators independently performed open coding, and then jointly discussed them by also resolving conflicts, and (3) refined the elicited smell catalog. After having devised a catalog of video game smells, we assessed their relevance through a survey with video game development professionals, also asking them to provide comments and suggest further smells that we may have missed.

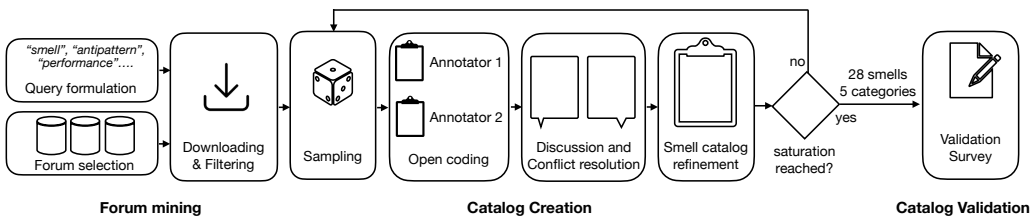


Fig. 1. Study methodology.

### 2.1 Study Context

For the context selection, we followed two subsequent steps. In the first step, we identified a set of queries aimed at retrieving candidate posts from developers' discussion forums, similarly to previous works that analyzed discussion forums for other purposes [24, 44, 46, 50, 52, 56, 62]. During a preliminary virtual meeting, we discussed possible queries, also starting from the investigation (and therefore its queries) performed by some of the authors in previous work [9]. Further, we revised the list through an iterative process in which the authors attempt to search for smell-related discussions through a search engine and by accessing some discussion forums. Our overall goal was to identify keywords meaningful for our study. It means that we were interested only in keywords that could point to discussions pertinent to our focus, *i.e.,* bad smells in video game development. The discussion ended up with the definition of six queries (of which the first three were considered initially, and then complemented by the last three): (i) "smell" (note we used "smell" to include cases such as "bad smell" or "code smell", although we were aware that this word alone could also produce some false positives in the context of video games), (ii) "antipattern", (iii) "bad practice", (iv) "maintainability", (v) "performance", and (vi) "technical debt". As it can be noticed, these queries are generic enough to ensure a good recall, even if they may produce some false positives, that can be, however, discarded during the manual analysis.

In the second step, we finalized the list of discussion forums to consider. Initially, we targeted general discussion forums, *e.g.,* Stack Overflow, and we manually submitted our queries defined in the previous step. Since the forums' field was related to development in general, we added to our queries keywords like "video game" or "game". Results gathered from general forums were few. Since our focus was specifically related to video game development, we changed our target towards Q&A forums related to video game development. In this case, the collected results were better than before. For example, querying "video game bad practices" on Stack Overflow produced only 24 results, while the same query on the Unity forum produced more than 9000 results. In addition, Stack Overflow results were mainly "how-to" questions.

To identify candidate forums, we leveraged (i) personal knowledge and (ii) lists of popular game development forums available on the Web. More precisely, to select video game-related Q&A forums, we searched online for the most popular game engines and, in a virtual meeting, we discussed and selected candidate forums. The website links used for selection are available in our replication package. In the end, we identified 13 forums, listed in Table 1. Note that, besides the 13 forums, we also included a direct search on the Google Search engine. Also, we included GitHub because we wanted to consider game-related Wiki pages. As the table shows, the selection includes forums related to specific pieces of technology (*e.g.,* Buildbox, Cocos, HTML5 Game Devs, Tigsource, Unreal, or Unity), hardware (NVIDIA), as well as generic forums (The Game Creators, Gamedev, Game Development Stack Exchange, or GitHub). From our initial list of candidates, we excluded forums (*e.g.,* those related to Indie games) that do not discuss development practices. A complete list of the forums is available in our replication package [43].

### 2.2 Data Extraction

To select the sample of discussions to be analyzed, we automatically ran the queries against the forums and downloaded the results. This was done using Selenium WebDriver for Python and the Beautiful Soup library. The Selenium WebDriver allows for automating the interaction with a website. Specifically, we used the Selenium APIs to submit the queries, and navigate the result pages.

Then, we leveraged *Beautiful Soup* to extract text from the HTML pages. Finally, we performed further automated filtering to check whether each candidate result page actually contained our

query. This was necessary because some forums do not support the "AND" search, therefore, for example, searching for "technical debt" would produce as a result all discussions containing either "technical" or "debt". As a result of the preprocessing, we collected a total of 3, 170 candidate discussion links.

### 2.3 Open coding of forum discussions

Given the large number of collected forum discussions (*i.e.,* 3, 170) and limited human resources, we extracted a randomly-stratified sample, using the forums in Table 1 as strata. More specifically, we considered a sample of 550 links, which ensures a ±5% margin of error with a confidence level of 99%. To perform a stratified sampling, links have been sampled proportionally (to the size of the stratus) from the results obtained for the different forums [22]. Precisely, the number of instance sampled from a stratus = sample size/population size * stratus size. Note that, during the manual analysis, we had to replace 22 links that turned out to be unavailable (likely removed in the time frame between our download and the manual analysis). This means that in total we considered 572 links.

Table 1. List of Forums/Sources for Sampling Developers' Discussions.

| Forum Name | Link |
|---|---|
| **Buildbox Official Forum** | https://www.buildbox.com/forum/index.php |
| **Cocos Forums** | https://discuss.cocos2d-x.org |
| **Game Development Stack Exchange** | https://gamedev.stackexchange.com |
| **GameDev.net** | https://www.gamedev.net |
| **r/gamedev** on Reddit | https://www.reddit.com/r/gamedev/ |
| **GitHub Support Community** | https://github.community |
| **Google Search** | https://google.com |
| **HTML5 Game Devs Forum** | https://www.html5gamedevs.com |
| **JVM Gaming** | https://jvm-gaming.org |
| **NVIDIA Developer Forum** | https://forums.developer.nvidia.com/c/visual-and-game-development/192 |
| **The Game Creators** | https://forum.thegamecreators.com |
| **TigSource Forum** | https://forums.tigsource.com/index.php |
| **Unity Forum** | https://forum.unity.com |
| **Unreal Engine Forum** | https://forums.unrealengine.com |

Then, we performed an open coding in which we manually analyzed the candidate discussion links to derive a categorization of video game bad smells. The coding has been conducted using online spreadsheets. Each coder used their independent spreadsheet, however, a separate (and common) sheet was used to list possible smell names. When tagging each link, the annotators first used a Boolean field to indicate whether the examined post was relevant or not (*i.e.,* whether it discussed or not video game bad smells), and then, if they answered positively to the previous field, they could select (using a drop-down menu in a cell) among the smells already available. If none of those smells fit, the annotator added a new smell name to the common sheet. Similarly, the coder could use a further drop-down cell (linked to another sheet) to indicate a possible high-level category to which each smell belongs.

As shown in Fig. 1, this process was performed iteratively over multiple rounds. Four authors were involved in the process, and each candidate link was assigned to two independent annotators. In the first round, we manually analyzed 340 links. This is a sample ensuring a ±5% margin of error

with a confidence level of 95%. The first 40 links of this first round were jointly examined, discussed, and categorized by all four annotators, to achieve a common understanding of the problem. Also, during this first phase, we defined possible high-level categories under which smells should be classified.

After the annotation round was completed, the annotators jointly discussed *all* links, including those where there was an agreement. This is because, on the one hand, given the open nature of the coding, it was not possible to compute a meaningful inter-rater agreement, and on the other hand, we wanted to avoid having cases where the annotators (wrongly) agreed by chance. After resolving disagreements, the annotators jointly analyzed, over multiple refinements, the list of smells, performing a merger where necessary. Also, the annotators finalized the set of high-level categories to be used for grouping the video game development smells. In the end, 5 categories were identified, *i.e.,* Game Design and Logic, Physics, Animation, Multiplayer, and Rendering. At the end of the first round, we identified 81 smells. Jointly we discussed the collected labels: we grouped and renamed similar ones (29 smells into 13 labels) and removed 13 smells (*e.g.,* those related to a specific technology used). Finally, our catalog contained 52 game bad smells and it has been used for the second round of validation.

After completing the first round, we performed two further rounds, in which the annotators inspected additional 102 and 130 smells respectively. The main goal of these rounds was to determine the extent to which we saturated the set of possible game bad smells. Those were the remaining from the 572 after having classified the first 340 ones. During these further rounds, the spreadsheet had an additional drop-down cell to classify the smell along with the five categories. After each round, as done for the first round, we performed a joint inspection of the classifications and a conflict resolution. At the end of both rounds, we added only 8 smells: 4 during the second round and 4 during the third.

Finally, we jointly, iteratively scrutinized the overall set of smells 60 (52 from the first round plus 8 from the remaining two rounds), merging them where appropriate, or moving them around categories where we realized the assigned category was not appropriate. More precisely, we merged 30 smells into 10 categories and we removed 12 very specific smells, *e.g.,* technology-related ones. One of the smells introduced in the second round and two smells introduced in the third round were merged into existing smells. In summary, the second and third rounds of labeling contributed with 3 and 2 new smells respectively. As a result, we obtained a total of 28 bad smells grouped into the 5 categories mentioned above.

The annotation activity took on average 5 minutes per post for each annotator, for a total of $572 \cdot 2 \cdot 5 = 5720$ min. $\simeq$ 96 hours. Clear negative cases took a few minutes, whereas relevant ones up to 15 minutes. This effort was complemented by the meetings we performed, *i.e.,* one agreement meeting, 4 discussion meetings, and two final smell catalog refinement meetings (one after the first round and one at the end), for a total of 13 hours. Also, note that part of the refinement activity was performed offline through spreadsheets.

### 2.4 Survey with Developers

After having identified possible bad smells in video game development, we wanted to investigate whether they are considered relevant by professional video game developers. We pursued this goal by surveying developers selected through LinkedIn. The survey structure is shown in Table 2. After a short description of the survey goals (as well as of the implications of the catalog we created) and a consent form, the survey is composed of seven sections. The first five sections allow the respondents, as shown in Fig. 2, to provide a relevance assessment to each smell using a 5-level Likert scale [14] where 1 star means "not critical", 3 stars means "neutral" and 5 stars means "highly critical". To facilitate the understanding of the problem, each smell is described by also providing

some examples. We explicitly instructed the respondents not to answer a question if they feel that they do not have enough expertise about the specific problem. Also, the respondent could use an open-text field to add additional comments.

Table 2. Survey structure.

| Sec. | Category |
|---|---|
| 1 | *Design and Game Logic*<br>Smell #1 relevance assessment (as in Fig. 2)<br>Smell #2 relevance assessment<br>... |
| 2 | *Multiplayer* |
| 3 | *Animation* |
| 4 | *Physics* |
| 5 | *Rendering* |
| 6 | *Additional Smells*<br>Would you like to suggest any other kind of smell not mentioned in the previous sections? |
| 7 | *Demographics*<br>Personal Information (Optional)<br>How many years of experience do you have in video game development?<br>How many years of experience do you have in software development?<br>Which languages do you use to develop video games?<br>Which game engine/framework do you use to develop video games? |

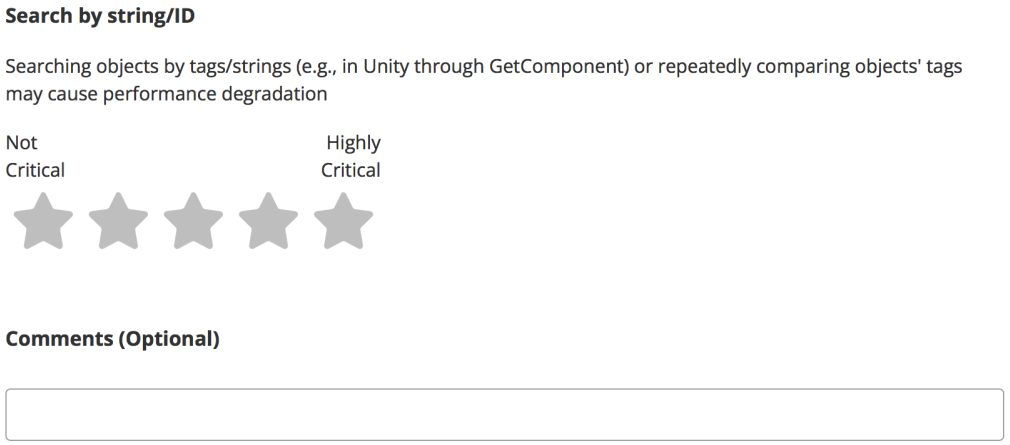


Fig. 2. Example of Question using 5 points Likert scale (from "Not Critical" to "Highly Critical") and an Opened-ended question for comments.

After the first five sections, a further section was used to allow respondents to suggest additional bad smells they feel were important, but that we did not mention in the previous sections. Finally, we had an optional demographics section featuring, other than fields to (optionally) provide contact and affiliation information, questions about (i) the number of years of professional development

experience, (ii) the number of years of video game development experience, (iii) the programming languages, and (iv) the game engines used for video game development. Note that we decided to avoid compulsory demographic questions. This is because, based on past experiences, such questions may discourage some respondents from completing the survey, and our goal was to maximize the number of answers. As it will be clearer later, we have mitigated this limitation by analyzing the effect of experience on the provided answers.

We tested the survey with a collaborator that had previous experience in video game development, and we also used his feedback to estimate the time it would take to complete the survey. Note that his answers were excluded from our results.

### 2.5 Participants' selection

We used LinkedIn as a research tool for the process of participant selection. LinkedIn is one of the largest professional social networks in the world and it has been used in previous studies surveying software developers, *e.g.,* [44]. During the selection process, as inclusion criteria, we targeted software developers with experience in video game development.

We organized a brainstorming session between the authors to define a set of keywords related to video games that could be used to gather the participants. Our research query contains the following keywords: "Game developer", "Game", "Game designer", "Game development", "Game engine", "Mobile gaming", and "Skeletal animation" [2].

We searched for participants, obtaining, for each keyword but "Skeletal animation" over 1,000 results, namely: 90,000 for "Game developer", 3,020,000 for "Game", 77,000 for "Game designer", 259,000 for "Game development", 11,000 for "Game engine", and 3,500 for "Mobile gaming". Then, we picked the top 200 (in terms of relevance) for each set, and two authors manually analyzed their profiles to make sure that they satisfy the selection criteria (*i.e.,* participants should have industrial experience in game development). The validation was based on their profiles, summary, and previous projects. Through this validation process, we ensured to reach only professionals with experience in game development. For the developers who passed the inclusion criteria, we sent a connection request. Then, for those who accepted we sent the survey link.

We invited a total of 642 professional developers and asked them to participate in the survey, by sending them an invitation message (available in our replication package) explaining the goals of our study, and clarifying that (i) the participation in the survey is voluntary, (ii) personal data will be treated as strictly confidential, (iii) the approx. time to answer the survey was estimated to be about 20 minutes, and a participant can withdraw at any time. The survey has been administered through SurveyHero[3]. The whole process of participants' selection, invitation, and survey response took about 2 months.

### 2.6 Participants' demographics

In the end, we collected 76 responses, *i.e.,* we achieved a return rate of 11.8%, which is in line with many software engineering surveys conducted outside specific companies [53, 55]. A total of 61 respondents provided answers to demographic questions. 30 of them had a Bachelor's Degree qualification, 13 a Master's Degree, 5 High school, 3 a Ph.D., and 9 other qualifications, such as a Diploma, *i.e.,* a certification awarded by colleges after two years of specialized training. A total of 36 respondents described themselves as Software Developers, 5 as Game designers, 4 as Technical Managers, 14 respondents mentioned other roles such as Production Manager or Game Developer teacher, while 2 did not provide an answer about their occupation.

[2]https://unity.com/how-to/beginner/game-development-terms
[3]https://www.surveyhero.com

The software development experience of the respondents (46 answers) ranges between 6 months and 24 years, with a median of 3 years. Their video game development experience (54 answers) ranges between 6 months and 24 years, with a median of 5 years. Note that we have also taken into account developers having less than one year of experience in development since we checked manually their background and years of experience on LinkedIn profiles.

In terms of game engines being used, the most adopted one among our respondents is Unity (44) followed by Unreal Engine (27), Blender (11), and CryEngine (2). In total, 17 respondents used other engines/frameworks, *e.g.,* proprietary engines or Cocos2d. Regarding the programming languages used to develop video games, the most widespread ones are C/C++ (45 preferences) and C# (45), followed by Java/JavaScript (16), Python (16), and PHP (4). Instead, 9 respondents used other languages to develop video games, *e.g.,* Swift or HTML. Note that the programming language is often related to the game engine being used (*e.g.,* C# for Unity, C/C++ for Unreal engine, or Python for Blender). Also, note that the sum of responses for game engines and programming languages goes above the number of responses as one could provide multiple answers.

## 3 RESULTS: THE CATALOG OF VIDEO GAME BAD SMELLS

Fig. 3 depicts an overview of the identified video game bad smells, grouped in the five categories (design and game logic, multiplayer, animation, physics, and rendering). In the following, we discuss in detail all the smells for the five categories. First, we describe each smell, also providing examples from the discussions we encountered. Then, we discuss possible ways to avoid it, reporting, and discussing the survey respondents' assessment and comments. For the five categories, results of the survey questionnaires are reported, using the chosen 5-level Likert scale, and using diverging stacked bar charts. The figures use different levels of colors to show different levels of the Likert scale, whereas percentages refer to aggregated negative, neutral, and aggregated positive values.

When discussing smells and their countermeasures, we keep traceability towards forums or survey responses. That is, a smell solution described while reporting an example comes from that example directly, whereas we use the notation *Rx* to trace discussions to survey responses. Note that the rationale for choosing examples among the classified discussion was mainly done with the purpose of favoring clarity, *i.e.,* we have chosen examples that better explain the problem (and possible solutions) in the paper. Therefore, the examples may not be fully representative of all problem instances. Finally, other discussions not traced explicitly are based on the authors' analysis and knowledge of the problem.

### 3.1 Design and Game Logic

This category includes smells concerning decision choices related to the overall game architecture (*e.g.,* what is on the clients, what is on the server), low-level design (*e.g.,* how different concerns of a video game are separated), the organization of game objects (*e.g.,* relations between similar objects, and object hierarchies), and implementation choices (*e.g.,* where the input handling goes and where the code handling actions and animation goes). The effect of poor design choices can be multiple, ranging from maintainability problems, *i.e.,* the video game becomes difficult to evolve, to performance problems, directly affecting the users' experience. The perceived relevance of design smells is reported in Fig. 4. All but three smells were perceived as critical/highly critical by the majority of respondents.

A smell that was perceived particularly crucial by developers is the one related to **Creating components/objects at run-time**, that has been considered as critical/highly critical by 83% of the respondents. The scenario in which this smell may occur is, to some extent, similar to what happens when one creates DBMS connections every time in a servlet without using a connection pool. This problem is even more relevant in video games, because, in some scenarios, several

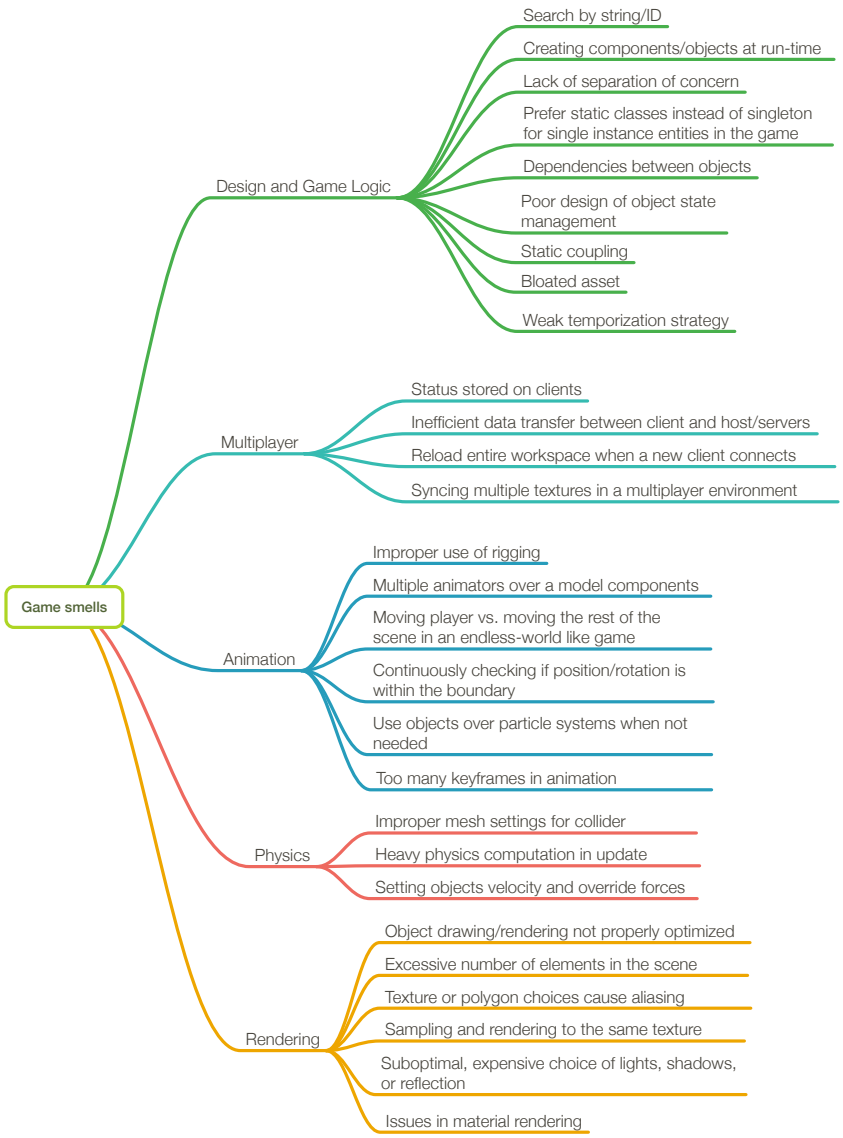


Fig. 3. Overview of Video Game Bad Smells Grouped into Categories.

objects (*e.g.*, bullets being fired) may need to be created in fractions of a second without degrading performance and user experience. The usual solution for this smell is the use of an object pool from which pre-created objects are taken and then released. For example, developers discuss on

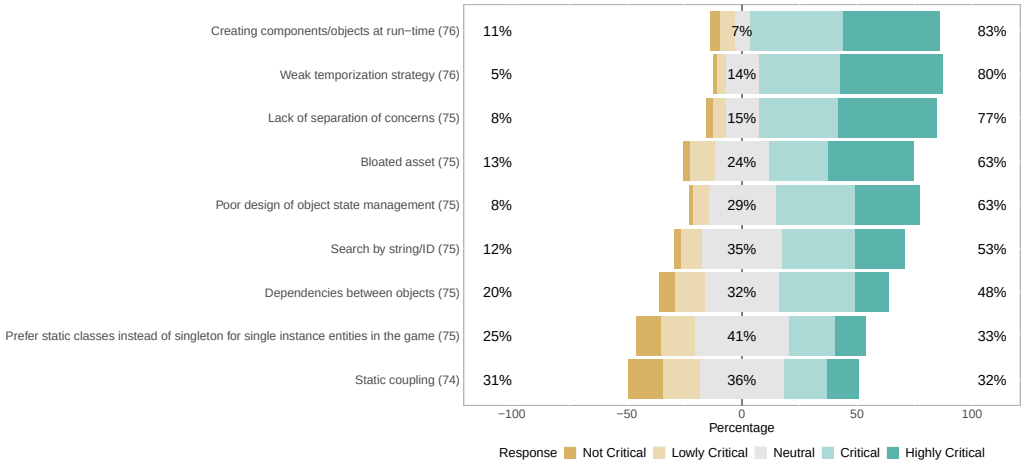


Fig. 4. Design and Game Logic Bad Smells: Developers' Perception (number of answers in parenthesis, the three percentages refer to the overall negative, neutral, and positive responses).

GameDev[4] about "*declaring many objects during the game's initialization and storing them in an object pool*" is the best way to avoid garbage collection of static objects: "*a static object pooler will be the best choice performance-wise*". Some respondents highlight how this issue is particularly important, *e.g.,* R3 "one of the major performance hindering things you can do is instantiating and destroying objects", while others like R2 despite agreeing mention that "there are cases of this that I would consider acceptable, such as spawning buildings in a tower defense game". Respondents that disagreed, *e.g.,* R25 mentions that this is "not that critical on small-scale projects and if used in moderation." Also, R39 mentions that creating/destroying objects at run-time could be acceptable while prototyping to avoid this becoming cumbersome.

**Weak temporization strategy** concerns wrong assumptions made on the time elapsed between subsequent game object updates, and was considered critical/highly critical by 80% of the respondents. A typical mistake is the game object update in a frame-based update (*e.g.,* a fixed movement is performed at every frame), making the animation speed dependent on the frame rate, and therefore varying on different devices, or even on the same device in different contexts. Common solutions imply the use of time-based updates (*FixedUPdate* in Unity, which is executed with a fixed frequency, whereas the conventional *Update* method is executed at every frame), as also mentioned by R37 and R48, or making the movement/animation proportional to the time between two frames (*e.g.,* in Unity, multiplying the movement magnitude by the `Time.deltaTime`, *i.e.,* the time elapsed between two subsequent frames) as also mentioned by R30. One respondent (R51) highlights how an early fix of this problem is highly desirable: "Grows more and more difficult to fix the longer the issue persists." Looking at a different technology, developers discuss this issue on the Unreal Engine forum[5] about the temporization strategy on a car multiplayer game. Simulating with two players, "*car response is slightly more "slow" or "sluggish""* on the client-side. To solve the problem, "delta time scaling" has to be taken into account when controls are handled since introducing this scaling will make your input less dependent on frame rate.

**Lack of separation of concerns** is the manifestation of bad design that, in principle, can occur in any software application, not only video games. However, this is one of the cases for

[4]https://gamedev.stackexchange.com/questions/101784/what-is-a-reasonable-way-to-avoid-gc-issues-in-unity
[5]https://forums.unrealengine.com/t/different-physics-behavior-on-server-and-client-when-tested-on-weak-pc/101125/5

which we decided to retain the smell in our catalog, because, also given the way game engines are conceived, some developers may be tempted to produce code exhibiting such a smell. A typical example is represented by cases in which source code handling controller inputs are mixed with code producing object animations. 77% of the respondents found this smell to be critical/highly critical. This smell is discussed on Game Development[6] site of StackExchange: developers state that "*[h]aving Logic and Data in the same object/class/structure is considered bad practice, and allows hackery that is likely to cause as many issues as it solves.* Thus, to simplify the code and reduce its complexity, it is good practice to separate game logic from game data: *e.g.,* "*[a] motion system, which updates position according to velocity, should not be able to read/change character data*". For example, one could attach to a game object multiple classes handling different concerns, such as movement, firing, defense, etc. Also, nowadays some game engines are trying to mitigate the problem, *e.g.,* Unity has developed a new input system that allows the developer to define logical actions bound to different input controllers. Where this is not available, developers should develop themselves virtual controllers.

**Bloated assets** refer to reusable assets (*e.g.,* complex game objects) bringing with them several elements (*e.g.,* various types of textures one can add to the object or various predefined animations). This may result in assets that are unnecessarily big especially when they contain assets that are rarely used. This smell was considered critical/highly critical by 63% of the respondents. For example, developers discuss this smell on the Stack Exchange Game Development[7]: "*example scenes with unnecessary art assets, scripts etc*" and recommend to remove unneeded assets "*not so much to save space, mostly to keep everything "clean"*", otherwise "*you will have classes with conflicting names, three sets of redundant animations for "jump" etc*".

**Poor design of object state management**, also perceived as critical/highly critical by 63% of the respondents concerns how a game object state is stored and handled, *e.g.,* using simple state variables and conditioned code, or a state-strategy design pattern [19]. R50 mentions that ". . . using if/else statement to manage the state of an object wouldn't allow easy expansion and would be rather hard to maintain." For instance, this is discussed in the JVM Gaming forum[8]. Developers debate on what is the best way to implement a state game manager, whether it is better to use design patterns, *e.g.,* State-Strategy patterns against if/switch statements. A developer advises to not ". . . *introduce design patterns and then try to force your application into them, but design your solution straightforward*". This is because design patterns "*really hardly make things easier*", thus they have to be used only when needed, as pointed out by previous literature [29, 60].

**Search by String/ID** occurs when an object is identified (or searched) by its string identifier/tag, for example when determining whether a collision occurred with a specific object. More specifically, if an object does not hold a reference to another (needed) object, it can access it through a search, which can be performed by name or by tag. This smell is perceived as critical/highly critical by 53% of the respondents. String comparisons, and even worse searching objects by their ID, could in principle cause performance lags. Also in mobile development (*e.g.,* Android development) this is discouraged and found to be energy greedy [58]. As R23 mentions "I would recommend comparing instances", R54 suggests using enums, and R1 mentions other side effects "Not only performance issues but prone to errors (tags/strings)". However, many other developers highlight that the performance degradation depends on how frequently such a comparison occurs (R3, R4, R13, R25, R39, R51,

[6]https://gamedev.stackexchange.com/questions/172991/entity-component-system-implementation-choices/173601#173601

[7]https://gamedev.stackexchange.com/questions/97712/do-i-have-to-commit-the-downloadable-assets-for-unity-to-the-repo-or-a-referenc/97726#97726

[8]https://jvm-gaming.org/t/using-a-switch-statement-to-determine-state-of-game/56771

R65). An example of this smell is discussed in the Stack Exchange Game Development forum[9] where developers state: "*beware that using Transform.Find or GameObject.Find should be avoided as much as possible, because it's quite slow*" and they also highlight that "*searching items by string arguments is a bad practice anyway*". A recommended solution is "*to use* `GetComponentInChildren<T>`*, and attaching to the game objects you want to be found a script (it can be empty) with name* `T`", because this limits the search space. As developers mention, the proposed solution is robust against any change on the searched object and it also works in case object setup is defined at run-time.

Further smells were considered as critical/highly critical by a minority of the respondents. The excessive presence of **dependencies between objects** creates, as in any other software systems, maintainability problems and even increased fault-proneness [7, 11]. When developing video games, an alternative would be to dynamically couple components instead. However, as R39 points out: "This come back to GetComponent being used to often unnecessarily. Other than this, it makes the code a little bit less readable than just having a variable containing the reference all the time." Therefore, developers prefer to favor performance over ensuring decoupling, also considering that such decoupling mechanisms also have a negative effect on source code readability. Similarly, this is discussed on the Stack Exchange Game Development forum[10] state "*. . . bad practices about gameObject/transform/components getters are connected with using them heavily, multiple times on every update*" and fix the performance problem through "*caching mechanism*", as R39 pointed out.

**Singleton vs. static** and **static coupling** deserve further discussion. The **singleton vs. static** choice is an endless question in object-oriented development [19]. Both static class and singleton as global variables should be used with moderation since they create a strong coupling with entities using them. However, the singleton design pattern allows for more flexibility in comparison to static classes. Static classes cannot be instantiated, cannot implement Interfaces or inherit a class, and can have only static members (constructor, fields, methods, properties, events), while a singleton can leverage all features of object-oriented programming. With static classes, one cannot control when the constructor is called and no parameters can be passed[11]. Several respondents, as expected, indicate that they make wise usage of both singletons and static classes, *i.e.,* the typical answer is "it depends" (R1, R2, R5, R32, R43, R48, R50, R76). Some explain when to use one and when to use others in the context of video games, *e.g.,* R43 "I use static classes for Data and Core methods. Singletons for managers/controllers. They serve different purposes", or R48 "Static classes should be used to store constants and some utility methods while singleton is used to reference objects universally (*e.g.,* Game Managers should be Singletons while Utility classes should be static)". Some advocate the use of singletons (R2, R5, R20, R30). For example, R20 says "Singleton has visible and controlled initialization time." Others point out their disadvantages, R38 "Prefer dependency injection.", R52 "singletons, especially when on monobehaviors, will increase complexity". Finally, some respondents point out solutions available in specific game engines, *i.e.,* R1 "Unreal has for example the GameInstance which is a singleton" or R47 "In Unity, even better to use Scriptable Objects". Developers in the game forums also have different views on the usage of singletons. For example, a participant in the Java gaming forum[12] mentioned that both singletons and static global variables are anti-patterns and this person recommended the usage of dependency injection instead. On the other hand, one developer mentioned the importance of using singletons in the specific context of unity engines[13].

[9]https://gamedev.stackexchange.com/questions/142546/in-unity-how-to-get-reference-of-descendant/142550#142550
[10]https://gamedev.stackexchange.com/questions/74566/using-this-gameobject/74568#74568
[11]https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors
[12]https://jvm-gaming.org/t/design-question/35647/4
[13]https://gamedev.stackexchange.com/questions/182053/is-it-bad-form-to-use-singletons-in-unity

**Static coupling** occurs when the coupling between game objects is enforced through the game engine IDE, *e.g.,* by dragging a game object on another game object's property. For example, one could drag a material towards a renderer to set the appearance of a game object. In some specific environments (*e.g.,* Unity) if a class attached to a game object (*i.e.,* a class inheriting from `MonoBehaviour`) has a public or *[SerializeField]* field, such a field appears in the IDE inspector, and it is possible to drag there any object compatible with the field type to create a dependency. That is, let us assume that a script attached to a game object named *Player* has to access another game object named *Ball*. Therefore, the script will have a [SerializeField] field of type game object, *e.g.,* named *myBall*. This field will be visible in the *Player* game object inspector, and dragging the game object ball into the field will create the dependency. On the one hand, this could be considered a bad practice, because dependencies are not stored (and visible) in the source code, yet they are encoded in the components' properties through the IDE (as discussed on Unity forum[14]). On the other hand, this is found to be very convenient by developers, and not considered as a bad practice in previous work [9]. The alternative to static coupling is to create dependencies in the code through component names (*e.g.,* using *GetComponent*-like APIs). In our study, only 21% found static coupling not to be a smell, and 23% disagreed with that. Some respondents (R2) indicated that this way of creating coupling is, instead, ideal, and the problem of "hidden dependencies" is mitigated by IDEs such as JetBrains Rider [1], which allows a developer to inspect game object variables. Also, R20 mentions that IDE-based coupling makes possible the use of the game engine dependency analyzer (not possible from code-based dependencies). Respondents who agreed this is a smell also mentioned (R38) that static coupling may be appropriate in some cases (*e.g.,* to couple scene objects with scripts), and less appropriate in other cases (*e.g.,* to couple scripts themselves). Nevertheless, some developers still explain why static coupling can be a bad practice, namely R44 "Soft dependencies are hard to search for."

**Key findings:** Bad choices in design and game logic are particularly important when they have a tangible effect on performances and users' experience in general. Very often, this is preferred also at the cost of reduced maintainability, *e.g.,* by preferring an increased coupling over solutions having negative effects on performance. Moreover, the magnitude/severity of a smell matters, *e.g.,* a smell potentially causing performance problems is still considered acceptable in circumstances where its effects are negligible.

### 3.2 Multiplayer

This category includes poor decision choices related to the design and implementation of a game's multiplayer component. In this case, as Fig. 5 shows, all smells were perceived as critical/highly critical by the majority of respondents. Multiplayer, online games are gaining popularity: a report by *GlobalWebIndex* [21] indicates how 26% of gamers leverage online games to create new contacts with others.

The first smell (and also perceived as critical/highly critical by the 91% of respondents) concerns **storing the game status on clients**. In general, status synchronization in distributed systems may be a problem for other applications too, *e.g.,* distributed data-intensive applications and in distributed systems in general. Video games have the peculiarity of requiring real-time synchronization to avoid disrupting the players' experience, and also may suffer from malicious attacks to cheat the game. Having the status stored on clients could cause synchronization problems, especially in case of unexpected disconnections from clients. Developers discuss this problem on

[14]https://forum.unity.com/threads/int-wont-decrease-to-zero.802566/#post-5331405

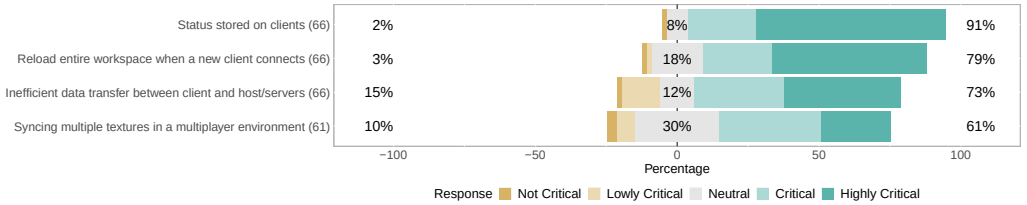


Fig. 5. Multiplayer Bad Smells: Developers' Perception (number of answers in parenthesis, the three percentages refer to the overall negative, neutral, and positive responses).

the Stack Exchange Game Development forum[15]. They also propose a solution: "*You should store the stack [game status] on the server and send it to the clients as they join. This will have also the advantage of preventing around with messing with the stack.*" Furthermore, the status could be potentially hacked by a player for cheating in an online competition. This problem was explicitly pointed out by seven respondents (R2, R4, R5, R27, R32, R39, and R48). An example of this security issue is discussed on the same forum[16] as above. Developers state: "PlayerPrefs" are "*VERY insecure*" location to store game data since they "*are extremely easy to edit, due to them being pretty much plain text and not encrypted in any way, so the values can be directly edited*". The proposed solution is to serialize the data creating "*dots a function by which you save your data to a binary file*". Some game engines have alternative solutions for state management. For example, Unity allows for handling the state locally and provides primitives/components to synchronize, but it has features to establish who has the authority to modify the status of a given object. For example, only the owner can modify the status of a player game object.

Another common bad smell is **Reload entire workspace when a new client connects**. This can occur when a new player connects in an online game and loads all his game objects; which can be a single object (*e.g.,* a humanoid or a vehicle), or multiple objects. This new connection should not cause a complete reload of the game space for all players, but just a differential update. 79% of the respondents agreed on this smell. Some respondents were more neutral, pointing out that this issue in some games could be avoided by design, *e.g.,* R2 explain that it "depends on if players can join while gameplay is running, or if they can only join while the rest of the players are in some kind of lobby/between match state." Other developers discuss the exchange of game information between Clients and Server [17], explaining that it is not needed "... *to send updates for all objects in the game, to all clients.*". They recommend selecting only the necessary objects that a client can see, and sending only their updates to the client. In conclusion, the problem can be addressed by design preventing its cause, *i.e.,* client connection during a game, happening at all, or by properly specifying (through the game engine networking APIs and facilities) a limited number of objects that need to be created and rendered when a new client connects.

A more generic problem (which can have different causes) is the **inefficient data transfer between a client and the host/server**, *e.g.,* either because more data than needed is transferred, or because data transfer is performed when not required. This problem was considered critical/highly critical by 73% of the respondents. For example, in the Stack Exchange Game

[15]https://gamedev.stackexchange.com/questions/108827/is-it-bad-practice-to-for-the-server-to-request-data-for-a-client-from-another-c
[16]https://gamedev.stackexchange.com/questions/124221/is-it-bad-practice-to-store-inventory-and-scores-in-playerprefs
[17]https://gamedev.stackexchange.com/questions/62096/how-do-i-duplicate-a-box2d-simulation-mid-simulation

Development forum[18] developers discuss on using sockets to let clients servers communicate. One developer mentions that sending full information to every client repeatedly is a bad practice. The *WebSockets* protocol does not have enough speed for an online game, developers point out "*[t]he technique is effective, but is not well suited for applications that have sub-500 millisecond latency or high throughput requirements*". Thus, a solution could be to limit the transfer only to primitive/essential data (as also R48 pointed out).

The least important problem (*i.e.,* perceived as critical/highly critical by 61% of the respondents) is related to problems of **Synchronization of game objects featuring multiple textures**. As pointed out by R48, to avoid this kind of problem, "syncing texture should be executed on client machine based on data received from server." For instance, on HTML5 GameDev forum[19] developers deal the problem of FPS (Frames per Second) dropping down caused by drawing canvas for dynamic texture. As a solution, they choose to update the canvas "*just twice per second instead on every frame*". In essence, continuous synchronization of properties such as textures may become excessively expensive. Therefore, a proper performance analysis is required to carefully tune the update frequency, balancing sometimes the realism of the objects' look and feel with performance.

**Key findings:** Multiplayer-related smells mainly concern how data is transferred and synchronized between clients and server/host. The aforementioned smells mainly have performance-related side effects, *e.g.,* lags when playing the game. Moreover, security issues need to be taken into account in multiplayer games, *e.g.,* by preventing a client to change the state of objects it is not owning.

### 3.3 Animation

Animation-related smells concern how game objects are animated. The six animation-related smells received, in general, a milder agreement than others. Indeed, as shown in Fig. 6, the percentage of agreements ranges between 42% and 60%, with, in general, a relatively high percentage of neutral responses (between 26% and 40%).

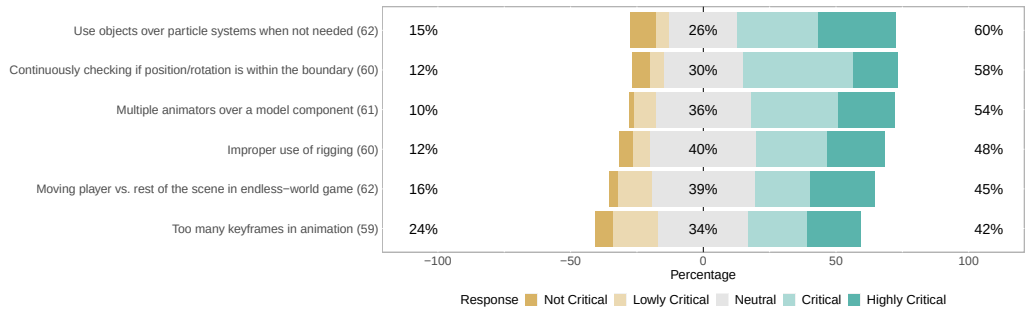


Fig. 6. Animation Bad Smells: Developers' Perception (number of answers in parenthesis, the three percentages refer to the overall negative, neutral, and positive responses).

The first smell is related to **using objects over particle systems when not needed**. Particle systems are sets of sprites, following certain animation patterns, projected towards the

[18]https://gamedev.stackexchange.com/questions/124246/input-and-output-of-a-server-side-game-using-web-sockets/126556
[19]https://www.html5gamedevs.com/topic/15564-help-with-dynamic-textures/#comment-88321

camera. They are typically used to create effects such as fire, smoke, water, etc. Creating some effects (*e.g.,* the presence of flies or butterflies in the sky) using game objects is unnecessary and heavyweight unless such objects need to interact with the rest of the game. Particle systems—animated sprites projected to the cameras—are more recommended instead. As pointed out by R5, "This should be pretty obvious, but if ignored will cause massive frames per second drop." For instance, in the Unreal Engine forum, we found a discussion[20] on how to render "*physical bullets*". Developers assert "*... for a realistic approach bullets are never visible, the only thing someone can notice are the vapor trails and the heated up bullets at night*", thus the best way to render them is using "*Hit-Scanning*" being "*less [performance] intensive*", *i.e., dots generating hundreds or even thousands of bullets in quick succession, you might experience some performance issues if each one is calculating velocity based on a 'ProjectileMovementComponent*". Note that *Hit-Scanning* is a programming technique casting a ray in the direction of the shot to determine where an object is pointing. In essence, particle systems alone are sufficient when the only goal is to create a visual effect. Whereas, when the game element needs to interact with other objects, and be subject to physics, then a game object, possibly complemented by particle systems, is required.

There is a smell related to how the movement of an object is handled in a confined region, *e.g.,* a car or a character on a track. In some games, instead of using colliders, this is implemented by **continuously checking if position/rotation is within the boundary.** As mentioned by R5, while this is necessary, a compromise should be pursued (too much checking would result in a lag, whereas enough would compromise the behavior). Also, R50 mentions how, in the end, this check does require a heavy calculation and in some circumstances, it can be preferred over colliders. Such a problem is mentioned, for example, in a Stack Exchange Game Development Forum discussion[21]. Developers want to improve the performance of collisions detection. The proposed solution is spatial hashing, *i.e.,* a technique to spatially divide the scene and keep into account objects belonging to each portion. When an object crosses the border between two parts, the collision detection algorithm checks for collisions only within objects belonging to that part, therefore reducing the required computation.

In some game engines (*e.g.,* Unity), the animator is a component that defines the animation behavior of a game object, for example through a state machine (**Multiple animators over a model component**). Such a state machine triggers different animations depending on whether a condition occurs. For example, if the vertical speed is greater than a given threshold, a character may start to walk, while it can turn left or right when the horizontal speed changes. On some occasions, developers may decide to attach multiple animators to the same object, *e.g.,* to handle different kinds of actions such as walking or firing, or to animate portions of the object, *e.g.,* moving its head. Some respondents (R5, R39, R51) explicitly mentioned not having experience with this concept (and in general this may be the reason for the high number of "Neutral"). However, some of the respondents, *e.g.,* R48 and R50, agreed that managing multiple animators would be painful, and, if needed, it would be much better to divide a single animator into layers, *i.e.,* sub-state machines. In a discussion of the Stack Exchange Game Development forum[22], developers debate the possibility of using multiple animators to separately animate the head and the rest of the body. They mention that, from a performance perspective, there is no impact, if one uses multiple animator components. Nevertheless, considering maintainability effects, they also advise opting for that only if needed.

When creating a third-person game in an endless world, *e.g.,* a racing track, one can decide to create a real model of the world and move the character with the camera, or else move the track

[20] https://forums.unrealengine.com/development-discussion/content-creation/10301-efficient-rendering-of-physical-bullets-no-hit-scanning
[21] https://gamedev.stackexchange.com/questions/74858/how-can-i-improve-my-collision-detections-performance
[22] https://gamedev.stackexchange.com/questions/129110/parent-and-child-with-different-animators

under the character (**Moving player vs. rest of the scene in endless-world game**). In the first case, the world model to be created may be very large, and possibly expensive, yet managing the game object's animation would be fairly simple, as it is simply an object, with a camera attached behind, that moves in a world. In the second case, it is not necessary to model the entire world nor to move the object, but there is a need for creating the environment behind the object as the latter moves. Some respondents (R30, R39, R50) mention how this design decision may depend on the game. Also, R30 mentions how, by using object pooling mechanisms mentioned in Section 3.1, it is possible to piece-wise create visible parts of the scene on the fly, giving the impression that the character is moving, and also, as explained in Section 3.1, mitigating the overhead due to continuous objects' creation. Developers discuss this smell on Unreal Engine forum[23], explaining challenges in simulating the movement of a space ship while it is moving at a high speed. They suggest moving the entire scene around the ship and not the ship itself to avoid "*wacky*" physics and glitches due to high speed.

The **Improper use of rigging**, *i.e.,* how game objects are connected to form a complex, animated object (*e.g.,* by connecting the bones of a character) can result in animation-related problems. For example, if an arm is not properly attached to the body of a character, it may be able to rotate towards unnatural positions. As R30 mentioned: "If the rigging is not properly done then the animation will not sync or give the feeling that we wanted it to give." However, the majority of respondents were neutral. One possible explanation (not mentioned though) is they are not directly involved in this problem, especially when models are mostly reused and adapted. For instance, a discussion[24] in the StackExchange Game Development Forum highlights the importance of using rigging for animations. In particular, developers point out that "*[i]t's very difficult to move an object independently to match an animation*" and recommend "*... to attach the object to the skeleton of the mesh*" since, although not perfect, "*it will almost always be good enough*". Through rigging, the animator will handle all the complexity needed to compute "*the precise dimensions of both meshes (character[/hand] and object) and the movements of the animation at every frame*".

Finally, when an animation is designed manually, this can be done by setting "key frames", *i.e.,* specific frames over the animation cycle in which the object assumes a given position or in general a given status (**Too many keyframes in animation**). That is, key frames set user-defined values for objects' position, rotation, or other properties, while values for frames in-between are obtained through interpolation curves. As pointed out by R39, it "would cause problems if it is done with sprite, but rigged animation shouldn't pose that much problem." In general, the controversial responses obtained for this smell indicate that in general it is not a big issue, and it really depends on the kind of animation (rigging model vs. sprites). In the HTML5 GameDevs forum[25], there is an interesting example of this smell. Developers highlight that performance issues can be due to a high number of key frames in an animation, and therefore suggest "*to remove most keyframes and just leave a few interpolations*" to improve performance.

[23] https://forums.unrealengine.com/t/walking-inside-a-space-ship-while-it-is-moving-at-high-speeds/42183
[24] https://gamedev.stackexchange.com/questions/104096/how-do-i-get-the-position-and-rotation-of-an-animated-object-in-unity/105908#105908
[25] https://www.html5gamedevs.com/topic/22796-problem-with-animation-in-blender-with-new-exporter/#comment-129973

**Key findings:** Animation-related smells are considered less critical than the other categories of smells by our survey respondents. A possible reason for that is because we mainly targeted developers, whereas some of the animation problems are handled by other specialists, *e.g.,* artists and experts in computer graphics, belonging to a video game development team. That being said, animation-related smells range from problems related to the general game animation architecture (*e.g.,* choice of particle systems vs. animated objects or design of endless games) to model-specific problems (*e.g.,* rigging or setting of key frames). The former may be of interest to developers too, while the latter may pertain more to a model designer.

### 3.4 Physics

We have identified three types of smells related to how the game physics is implemented. Their perceived importance is reported in Fig. 7, and two out of three were considered critical/highly critical by the majority of the respondents.

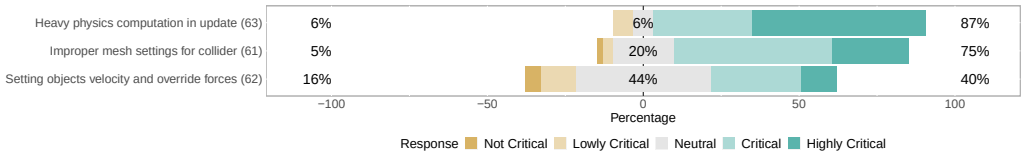


Fig. 7. Physics Bad Smells: Developers' Perception (number of answers in parenthesis, the three percentages refer to the overall negative, neutral, and positive responses).

The first smell, which received a very high level of agreement (87%), is related to the **heavy-weight physics computation in game object updates**. While this smell may appear very similar to bottlenecks in any conventional software, in this case, it is particularly important to distinguish what is executed only when a scene (or some objects) are created from what is executed continuously, *e.g.,* at every frame or with a fixed frequency, for the fixed updates. This smell occurs when the game physics (*i.e.,* the forces acting on each object) is continuously recomputed and then applied to produce animations. On the Stack Exchange Game Development Forum[26], developers discuss this problem "*to set the velocity of an object every loop*". This could cause performance drops, and a possible solution can be to update the object's velocity only when the external velocity value changes, introducing, for instance, an `if` statement checking this change.

While it is the norm to leverage update cycles to recompute physics and separate this from animations, attention needs to be paid to avoid this becomes a bottleneck, for example by updating game objects' physics only when necessary, or in general by optimizing such computations (see example on the Stack Exchange Game Development Forum discussed above[26]). Four respondents (R30, R37, R40, and R50) clearly stress the use of time-based updates (known as *Fixed Updates* in Unity) instead of updates performed at every frame, although this does not solve the problem. Others suggested updating physics every $N$ cycles (R37), or using optimized approaches such as lookup tables (R41) to fasten physics computation.

**Improper mesh settings for a collider** could also cause problems. 75% of the respondents pointed out this smell as critical/highly critical, and 20% were neutral. There exist different types of colliders. A mesh-based collider is composed of many (possibly small) triangles and fits a game

[26]https://gamedev.stackexchange.com/questions/51356/is-it-bad-practice-to-set-the-velocity-of-an-object-every-loop?r=SearchResults

object's shape, but it may negatively affect performances. Then, there are "coarser" colliders having the shape of boxes, cylinders, capsules, or spheres. Unless the object has a simple shape, the latter may not be able to properly fit the object, and can cause less precise collisions, yet being less resource demanding. Some respondents like R5 and R50 recommend always using the simplest collider, but, as explained above this can cause game glitches, *i.e.,* unwanted collisions. In the Stack Exchange Game Development Forum[27], there is a discussion about the bad practice of detecting collisions per pixel, leading to expensive physics computation. The right way to handle collision is to use polygon shapes. In essence, a lot depends on how collisions would occur in a game. For example, if a car can only impact against a wall or other cars, then a box collider may be sufficient. Instead, if objects may collide against its windshield, then a mesh collider would be needed.

**Setting object velocity and override forces** means that an object's speed is not determined based on forces applied to it, but, rather, programmatically set, *e.g.,* as a result of a player's input. Unsurprisingly, this smell has been considered quite controversial (44% neutral, and only 40% agreeing). This also depends on the fact that in most cases developers rely on predefined components to determine the movement of an object based on the received inputs. For example, Unity provides the *CharacterController* component for that. However, when the object's movement has to be determined based on multiple forces, to also allow some scenarios not contemplated by predefined components, then it may be necessary to directly modify the object's velocity. One reason is that some games intentionally have no physics, as one respondent (R50) indicated: "A game without physics shouldn't have to use forces to translate his units". A respondent (R25) who strongly disagreed indicates that the game engine s/he is using has poor physics management, therefore " homebrewed physics systems are usually a better option if there is the time, money and energy", although, in this case, a separation between physics and movement is still possible and desirable. Regarding this smell, we found two interesting discussions[28][29] on the Stack Exchange Game Development forum. They mainly highlight that directly setting the velocity on a rigid body is a bad practice since it "*cancels all other forces acting on the rigidbody*". Another side effect occurs in presence of "*other non-kinematic non-static rigidbodies blocking the object's path*" leading the engine to "*give it as much force as it needs to push them away without slowing down*" and consequently "*[y]ou might witness "fun" game mechanics like rigibodies getting launched into orbit or getting pushed through solid walls.*" A possible solution is to avoid directly setting the objects' velocity, and apply any additional force instead.

**Key findings:** Smells related to the game physics concern the way the game physics is being computed, through scripts or by leveraging existing components, and by setting-up colliders. Unsuitable choices may cause in some cases performance issues (*e.g.,* heavyweight updates or colliders), but also game glitches, *e.g.,* due to coarse collision handling or improper physics computations.

### 3.5 Rendering

Rendering smells concern issues related to the way objects are drawn/rendered, as well as the various visual effects in the games. Bad choices in this regard could not only cause performance problems but also result in visualization glitches, *e.g.,* aliasing or, in general, objects not properly

[27] https://gamedev.stackexchange.com/questions/17222/xna-platformer-collision-perpixel-vs-rectangle
[28] https://gamedev.stackexchange.com/questions/153419/proper-way-to-set-a-rigidbodys-maximum-velocity
[29] https://gamedev.stackexchange.com/questions/186505/what-are-the-dangers-of-setting-rigidbody-velocity-directly-for-movement-and-wha

drawn. As shown in Fig. 8, all but two smells were considered critical/highly critical by the majority of respondents.

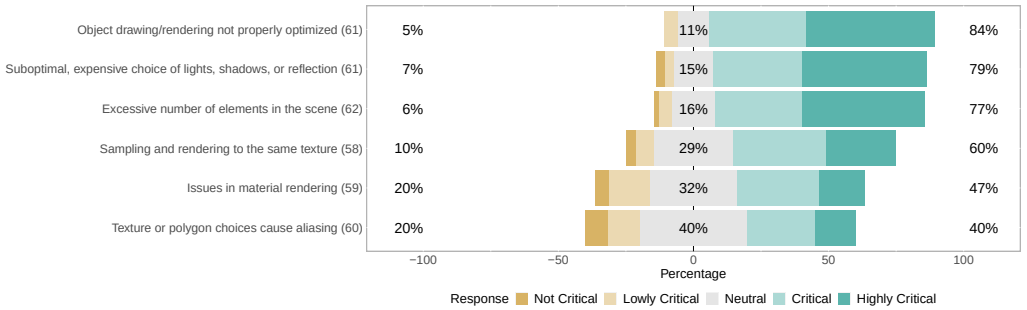


Fig. 8. Rendering Bad Smells: Developers' Perception (number of answers in parenthesis, the three percentages refer to the overall negative, neutral, and positive responses).

The most important problem here is the **LACK OF OPTIMIZATION WHEN DRAWING/RENDERING OBJECTS**. For example, too far, not visible objects are always redrawn, or all objects of a scene are redrawn at every frame (and not just those that have been changed). For example, on Unreal Engine forum [30] developers discuss whether to apply invisible material to some parts of the scene (body-parts) will improve the performance. They confirm that "*Applying a transparent material to a mesh will not make it render faster*" since "*there is a cost to translucency and masking. The vertices are still there, just deferring to render specific parts of the material*". A possible solution could be to use LOD (Level of Detail) generator "*to create new LODs that can look just as good as the full resolution mesh*". 84% of the respondents perceived this smell as critical/highly critical. R4 mentions that this is "one of the biggest sources of unnecessary framerate drop", while R37 argues that a clear solution for this smell and the other smells mentioned below is to "use occlusion culling to make sure GPU have to only compute objects at the camera's sight." Specifically, the occlusion culling determines the view angle of a camera and how far it can "see" objects. By reducing the occlusion culling, very far objects are no longer visible by the camera and, while in some cases this may reduce the level of detail, it can contribute to performance improvement.

Another problem, considered critical/highly critical by 79% of the respondents, is related to visualization effects, *i.e.,* to the **SUB-OPTIMAL, EXPENSIVE CHOICE OF LIGHTS, SHADOWS, OR REFLECTIONS**. For example, some lights whose behavior does not depend on the game dynamics could be "baked" statically instead of being rendered in real-time. In other words, when a shadow, a light, or a reflection is baked, it does not change when objects affecting it move. Therefore, it is suitable for effects produced by static objects only. Indeed, R5 mentions that "the concept of "baking" is quite important and should be taught early." That is, developers should perform an inventory of static and dynamic objects, as well as of their related effects, to properly plan where baked or real-time effects should be used. Also, the excessive usage of shadows or reflections could cause performance issues, as pointed out by R39, "shadow and reflection cost a lot of power and should be optimized." On the one hand, the aforementioned effects make the video game more realistic and improve its look and feel. On the other hand, they can be performance-demanding, especially on limited-resource devices. An interesting example is discussed on Unreal Engine forum[31]: developers ask if there exists a way to "*[d]ynamically light up a big hall without too much performance lost*".

[30] https://forums.unrealengine.com/t/ue4-applying-invisible-materials-to-some-bodyparts-improve-fps/149799
[31] https://forums.unrealengine.com/t/dynamically-light-up-a-big-hall-without-too-much-performance-lost/78721

More precisely, when a player comes into a room, the light is turned on with variable intensity and the attenuation radius causes performance issues. Several suggestions have been made to solve this problem, *e.g.,* try to use less light, make some lights static, or try to split big mesh into smaller ones.

Problems related to the presence of an **excessive number of elements in the scene** is considered critical/highly critical by 77% of the respondents. The way this problem can be solved is by (i) incrementally rendering elements only when they become visible, (ii) reducing the camera occlusion culling, as also mentioned above for another smell, *i.e.,* the distance over which objects are no longer visible, and (iii) simplifying the objects being shown. For example, R30 mentions that "if some meshes are not important, it's better not to put it or you can change the rendering setting to high". Some other respondents mentioned that this kind of problem is more on the engine optimization side than on the programmers' or computer graphics' side. R56 mentions that "the engine needs to be able to support multiple elements without hindering performances", while R54 clarifies the rendering capability of the Unreal Engine: "UE (Unreal Engine) 4 handles up to 1 million mesh and UE 5 20 million". A really simple example of this smell is discussed on Unreal Engine forum[32]: developers underline that performance issues are related to a high number of objects on the scene which leads to render a large number of objects on the same time. A possible solution is merging the objects to have "*fewer meshes to process*". The main issue is the number of objects (each one will impact performance). Hence, even if it is not a complex model (adds to the number of draw calls), it is better to merge objects that are very simple to lower the number of objects, starting from those using the same material.

**Sampling and rendering to the same texture** was considered critical/highly critical by 60% of the respondents, and it is related to scenarios when one is writing to a texture while it is being rendered, causing undefined behavior in the GPU. A recommended practice is, instead, to create a texture copy, modify it, and then render it again. On the Game Development forum of StackExchange, developers state that "*... reading and writing to the same texture in a shader is bad practice (not surprisingly) and will cause undefined behavior on GPUs*" and propose also a solution to that: "*... copy the target texture to a new texture and then read from the copy while still writing to the original target texture*". R5 mentions that "this is both very important and unlikely to happen (most people know not to do that)". In other words, while the behavior related to this smell can be indeed problematic, most game developers do not perform such advanced (run-time) texture modifications. R39 mentions that some game engines have already a solution for this problem, *e.g.,* "Unity seems to handle that automatically (cloning material when called directly)". In essence, the avoidance of this problem can be achieved on the one hand by developing static analysis tools able to detect the problem, similarly to race condition detectors. On the other hand, game engines could provide avoidance mechanisms, such as the cloning of Unity or, for example, allowing mutually exclusive access to textures.

Two smells were perceived as less important, with a higher percentage (the majority) of neutral responses and a minority of agreements. This is likely due to the background of our respondents. Most of them are game developers and not specialized in computer graphics, *i.e.,* they rely on expertise from others for such specific problems.

**Issues in material rendering** happen when a material is adopting the wrong mesh types, or, for example, the presence of side-by-side materials could cause visualization issues. As pointed out by R54, the severity of this smell can depend a lot on the engine. This smell was considered critical/highly critical by only 47% of the respondents, and 32% were neutral. This could be because the relevance of this smell highly depends on the context. In a discussion[33] on Unreal Engine

[32]https://forums.unrealengine.com/t/beginner-performance-issues/96340
[33]https://forums.unrealengine.com/t/is-there-any-hit-on-performance-with-2-sided-materials-vs-non-2-sided/76736

forum, developers asked whether using a 2 sided material against a non 2 sided affects performance. Developers point out that "*... it costs more performance to render a material two sided than not.*", but they also emphasize that "*everything in game design is about context so without context the answer is yes/no/maybe*". In this case, there is a trade-off: choosing or not an accurate rendering depends on the game context or targeted devices, *e.g.,*, whether the game is either desktop project, Mobile, or VR. Regarding two-sided material, developers advise to "*[not] avoid it any cost - just make sure not to use it on everything unless absolutely necessary*". For this smell, other than a suitable performance evaluation, automated tests able to identify the presence of visualization glitches, *e.g.,* by analyzing game videos, may be desirable.

Finally, **Texture or polygon choices cause aliasing** occurs when a bad choice of object texturing or number/type of polygons of a game object causes aliasing, *i.e.,* lines and surfaces that exhibit jagged edges. This smell was perceived critical/highly critical by 40% of the respondents, with 40% being neutral. Confirming what we conjectured above, R39 mentions that this is "more a problem on the 3D artist side". Also, R39 mentions that "gameplay and performance are mostly unaffected by this" (indeed the smell mostly results in visualization problems than in performance problems). An example is discussed on the Game Development site of StackExchange[34]: developers asked for a possible solution to "*no smoothing applied to the textures*". Jagged and black lines were displayed around a figure.

**Key findings:** While ensuring proper rendering should be a responsibility of computer graphics experts rather than developers, such problems are perceived as very important, because they often have tangible effects on the game's visual or its performance. In several cases, also depending on the employed hardware, there is a tradeoff between quality of detail and performance. From a software designer's perspective, it is important to allow the game to have a varying level of detail depending on the hardware characteristics.

### 3.6 Other smells the study respondents suggested us

In total, 13 respondents gave us additional suggestions about the relevance of other types of smells not considered in our study. Two of the authors performed an (independent) manual classification of such comments, after splitting those with multiple suggestions (in total, we ended up with 22 rows to classify) followed by a discussion. It was found that they belong to different categories:

- *Conventional code smells (5 suggestions):* in six cases, respondents pointed out (known) conventional smells, including abuse of references/coupling to static classes, abuse of logging statements, use of magic numbers, bad naming conventions, or incorrect usage of design patterns.
- *Issues more related to the game story or human-computer interaction (4 suggestions):* these are not strictly related to the game design or implementation. Instead, they mention problems related to easy-to-use/understand user interfaces, gameplay/game experience, design of the game story itself, and lack of proper communication/misunderstanding between different stakeholders involved in the video game development. While the former problems are beyond the scope of our work, the latter is particularly interesting. As the respondent explains, these are problems leading to reengineering "artists deciding to change the elbow of a character, breaking all the animations" or "sound designers adding Rigidbody to prefabs ... and the coders spend 3 days to figure out a problem that came from that ..."

[34]https://gamedev.stackexchange.com/questions/110286/how-to-fix-texture-edge-artefacts

- *Duplicates (3 cases):* in three cases respondents reported problems that could be a specialization of some of the bad smells that we already considered, namely (i) Recalculating object state inside Update (which is our **Poor design of object state management**); (ii) "excessive use of strings instead of constants", which, depending on the specific case, can be brought to our **Search by String/ID** or it can be a conventional magic number smell; and (iii) using game objects as folders, which is a case of **Bloated asset**.
- *Additional game smells (2 suggestions):* finally, we found two cases where respondents proposed two smells that we did not consider. The first one is about the use of the "Script Execution Order" occurring in game engines such as Unity, where scripts attached to all game objects are executed in an endless loop. This may cause unexpected behavior, because it is not clear whether actions happen in a certain order. The second one is the "wheel of death pattern of using Any State to transition anywhere in an Animator Controller". This is a specific issue occurring in Unity, where the *Animator Controller* is a component that governs a game object's animation through state machines. The *Any State*[35] is a special state used to specify that, should a given event occur, a transition towards a certain state must happen, regardless of the previous state (which is therefore modeled with this special state). However, in general, this could be also seen as a special case of **Poor design of object state management**.

### 3.7 Are the studied smells specific to video games?

While our study has focused on bad smells occurring during video game development, as also discussed in the previous sections, some of the smells may apply to other domains as well. Nevertheless, such smells are particularly relevant during video game development, and therefore we decided to keep them in our taxonomy.

Broadly speaking, smells related to the last three categories (Animation, Physics, and Rendering) are all specific to video games, or, possibly, can be applied to computer graphics, special effects in movies, or simulation. Instead, smells related to design and game logic or smells related to multiplayer (except for the one related to **Synchronization of game objects featuring multiple textures**) can also occur in other types of systems. To this aim, Table 3 provides, for each game smell applicable to other domains, an explanation of why it is particularly important in video game development, as well as scenarios where it can apply elsewhere.

## 4 LESSONS LEARNED

This section discusses the lessons learned for developers, educators, and researchers.

### 4.1 Lessons for developers

**Prioritize the removal and avoidance of smells having a negative effect on the game experience.** Our results, for example those related to design and game logic smells, but also those of other smell categories, such as animation and rendering, highlight how very often smells become problematic only if they have a tangible effect in terms of game performance, or on the gamers' experience, which is the highest priority during development. If this is not the case, the smell is not considered as particularly critical and therefore worthwhile to be removed. Also, smells having a negative impact on maintainability are considered as less critical. This is a substantial perspective shift from conventional software development and to the existing literature of code smells and antipatterns [12], where there is a high emphasis on program comprehension, maintainability and design for change [2, 25, 30, 61]. When focusing on the video game development domain, while

[35] https://docs.unity3d.com/Manual/class-State.html

Table 3. Game smells applicable to other domains

| Game Smell | Relevance for video games | Where applies elsewhere |
|---|---|---|
| | Design and Game Logic | |
| Search by String ID | Dependencies may be created by ID; use of tags to determine the objects' identity | Search of Android views [58] |
| Creating components/objects at run-time | In games, some types of objects (*e.g.,* bullets) are frequently created and destroyed in many instances | Database connections in client-server (*e.g.,* Web) applications |
| Lack of separation of concern | Game development framework may encourage mixing-up inputs, physics, and animation | Any cases where the structural [13] or conceptual [38] cohesion is low |
| Prefer static classes instead of singleton | (Static) Game objects must be unique to avoid access errors | Use of singleton vs. static classes largely discussed in object-oriented development [19] |
| Dependencies between objects | Need to achieve a tradeoff between decoupling and good performance and temporization | Excessive coupling is a general problem in software development [7, 11, 13] |
| Poor design of object state management | Some game behavior, *e.g.,* movement or animation may require complex, change-prone state models | Proper solutions for state-management advocated in object-oriented development [19] |
| Static coupling | Video Game IDEs entail the visual creation of dependencies | In principle, this might happen in other IDEs too |
| Bloated assets | Models may contain redundant/unlikely to be used elements, *e.g.,* textures or animations | The general concepts may apply elsewhere, *e.g.,* one can create bloated assents in software configuration management, or bloated containers |
| Weak temporization strategy | Perceived speed should be frame-rate independent | A similar smell may occur in any time-sensitive systems structured as an endless loop |
| | Multiplayer | |
| State stored on clients | May cause game cheating | May occur in other distributed systems, *e.g.,* online banking |
| Reload entire workspace when new clients connect | Game experience disruption caused by poor game lobby management | Some other types of applications, *e.g.,* online GPS navigators may exhibit similar issues |
| Inefficient data transfer between client and host/servers | Avoiding lags is essential in online games | In general, true for distributed systems |

designing for change is still important, *e.g.,* to allow releasing patches or downloadable content, the developers' main focus is on the gameplay, and therefore a coding practice is considered problematic only up to the point it may lead towards performance degradation effects. These considerations influence the way video game developers should prioritize quality assessment and improvement activities, *e.g.,* by combining smell identification with an assessment of their intensity and possible effect on performance.

**Smells are highly game/framework dependent.** Depending on the type of game under development, on the adopted technology, or the targeted hardware and software platforms, developers may or may not encounter a specific type of smell, or may consider it more or less important. Indeed, mitigation strategies adopted to deal with bad smells highly depend on the specific problem being solved. For example, as also discussed in Section 3.4, physics-related bad smells may not be an issue at all for games where physics does not apply, or they depend on the extent to which the game physics needs to be customized with respect to what is available out-of-the-box.

Similarly, some performance problems may occur only in a multiplayer context, *e.g.,* where multiplayer-specific smells such as **Reload entire workspace when a new client connects** may even interact with other smells, such as those related to an expensive rendering, *e.g.,* **excessive number of elements in the scene**. Moreover, as pointed out before, the performance impact of a smell is a key factor when assessing its relevance, therefore it is also important to determine the hardware and software characteristics of the targeted platforms, as some smells may affect more older consoles or mobile devices.

**Early fix of smells is highly desirable:** Some smells may become cumbersome to fix if they persist longer and developers have to prevent these smells at the design phase, *i.e.,* early in the video game development life-cycle. For instance, R39 states that "the concept of "baking" is quite important and should be taught early" when commenting about the rendering smell **sub-optimal, expensive choice of lights, shadows, or reflections**. In general, as discussed in Section 3.5, a proper and early planning of static vs. dynamic rendering of effects is highly desirable. Moreover, when commenting about the smell **Creating components/objects at run-time**, R40 mentions that it is "very important to start early while in production. During prototyping not so much, since it makes development cumbersome, but the infrastructure needs to be put early, because it is a nightmare to create later and this can impact performance by a lot on small devices."

**Heterogeneous teams:** Another emerging problem is the lack of communication between development team members leading to re-engineering. Video game development involves many skilled professionals [8], *i.e.,* Scripter, Game Designer, Graphics/Animation Programmer, etc. R48 highlights some problems derived from lack of communication between team members: "*artists deciding to change the elbow of a character, breaking all the animations, sound designers adding Rigidbody to prefabs because Wwise requires it and the coders spend 3 days to figure out a problem that came from that*". In this context, as discussed in Section 6, specific studies have analyzed video game-specific project management smells [57].

**Good practices from software development are still valid for video game development:** Some good practices coming from general software development, *e.g.,* clean up the project removing unnecessary code (dead code) or Single Responsibility Principle [40], are also valid for video game development. In some cases, developers are "encouraged" to introduce some smells by the IDE. For instance, Unity allows to statically couple objects through the game engine IDE using the [*SerializedField*] field attached to a game object, in this way the object will be visible to the IDE inspector (*i.e.,* the **Static Coupling** smell). However, as R1 points out "*. . . it is a real advantage for designers"* even though it is "*[n]ot visible and can be broken*" easily. This also calls for better tool support still aiding developers on the one side, while avoiding maintainability problems on the other side.

### 4.2 Lessons for educators

**On teaching design principles and goals for the video game domain:** more often than not, when teaching software design, there is a lot of emphasis on design for change, and in general on targeting maintainability goals. When designing video games, performance and user experience are the primary design goals. Therefore, developers should be instructed about design choices that could positively impact performance (*e.g.,* the use of object pools avoid searching objects through strings) and, at the same time, they should be careful with choices (*e.g.,* heavy usage of design patterns) that could have a positive effect on maintainability, but that may have a negative impact on performance.

**Elements of computer graphics are necessary also for software developers:** video game developers are not necessarily in charge of creating 3D models and animations, and will unlikely be artists. At the same time, when integrating these elements into their products, they should be able to grasp the basics of 3D modeling and computer graphics. As our study has shown, smells belonging to certain categories (*e.g.,* Animation, and above all Rendering) originate from wrong choices made when integrating graphical models into a game, even when developers mainly reused out-of-the-box assets produced by others.

### 4.3 Lessons for researchers and tool makers

This work paves different research work for researchers, as well as for companies developing IDEs or tools aimed at supporting video game development.

**Game-specific smell detectors.** Primarily, our study triggers the development of video game-specific bad smell detectors. Following what has been done in previous work [9] for five game smells which have been confirmed also in our study, it would be desirable to develop specific detectors for the smells we have identified in our study.

**Smell severity is relevant wrt. performance.** Especially for smells causing performance issues, the smell severity and resulting effect may vary from case to case. Therefore, static analyzers could be complemented with specific dynamic analysis tools, or testing generation tools, aimed at exercising specific components of a video game where a smell is likely to exist. Moreover, as discussed above, smell detectors should provide a smell severity and priority indicator based on the extent to which a smell type, and its intensity, is expected to have an effect of the game's performance.

**Smell removal recommendations.** & **Misuse game engine and facilities.** Smell detectors could be accompanied with recommenders that, based on the available knowledge or, even better, by learning from past changes, would be able to automatically suggest a refactoring or, in general, a solution for a given smell.

Ideally, the aforementioned tools could be integrated into popular video game development IDEs. As of today, some IDEs are able to identify some game-specific problems. For example, JetBrains Rider[36] [1] already tell developers if they are comparing object tags using the == operator, suggesting alternative solutions. Also, it reminds that the `Update()` method may become critical because it is executed at every frame (without checking its content, as *GameLinter* [9] does). Finally, it is able to identify too frequently-invoked APIs in a source code file. Therefore while some solutions for a limited set of problems are available, there is a room for a better integration of game smell detection and avoidance in the IDEs.

[36] https://www.jetbrains.com/lp/dotnet-unity/

## 5 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. The main threat of this kind of study (as other studies following a similar protocol [15, 23, 24, 44, 46, 52, 56, 62]) is that, by analyzing forum posts, one gets a perceived indication of what a bad smell/practice is. To some extent, this is also true for the survey, although the latter is less subject to our interpretation than the forum posts. Indeed, the relevance analysis performed through the survey helps to mitigate the threat that, by reading the posts, we could have misinterpreted the bad smells expressed there. That being said, once the catalog of bad smells is available, it would be desirable to conduct further studies, *e.g.,* field studies or analyses of existing video games. A further threat could be related to the misunderstanding of the smell description by the respondents. We mitigated this threat by having a comprehensive description including examples.

Threats to *internal validity* concern factors, internal to our study, that could have influenced our results. The selection of candidate posts could have been affected by the specific queries we performed on the forums. To mitigate this threat, we considered generic-enough queries. Moreover, we identified possible terms during an iterative process in which we used the query terms to search over game forums and a search engine (Google). Last, but not least, while missing some key terms could have resulted in an incomplete set of smells, we have mitigated this threat by asking the survey participants to list smells not considered in our study. A further threat can be due to subjectiveness or errors occurring when labeling the posts. We have mitigated this threat by (i) having multiple independent annotators (at least two for each post), (ii) performing a joint training on a subset of 40 posts, and (iii) jointly discussing all cases. Note that, since we performed an open card sorting in an exploratory context for which no predefined categories were available [51], computing an inter-rater agreement may not make sense (we have followed a similar protocol used in other exploratory studies [26, 47]). That being said, to avoid the risk of an agreement by chance, every single sample was jointly opened and discussed, even in the cases where there was an agreement.

Perception studies based on questionnaires could suffer from subjectiveness, also in establishing the suitable level in a Likert scale. We used a five-level Likert scale [14] also used in previous similar studies (e.g., [50, 62]). Moreover, to mitigate possible subjectiveness upon deciding between multiple positive or negative scores, the used representations also show the overall percentage of negative, neutral, and positive responses. Finally, there could be the fatigue or boredom effect experienced by participants in answering questions related to 28 bad smells. Participants had the option to skip smells they are not confident about.

Threats to *conclusion validity* concern the relationship between theory and outcome and are mainly related to the extent to which the produced catalog can be considered exhaustive enough. We have performed the labeling over three subsequent rounds and observed how the number of newly introduced smells in the last rounds decreased to four in both the second and third rounds. At the same time, we believe achieving a complete saturation may be nearly impossible as it is still possible to find further smells. We have mitigated this threat by asking the survey respondents to indicate smells that they believe are relevant and that were not part of our catalog.

Threats to *external validity* concern the generalization of our findings. We have analyzed posts from 13 popular video game development forums. Problems not discussed in those forums (*e.g.,* arising when using proprietary video game development technology) might not have emerged in our study. We have mitigated this threat by complementing the post classification with a survey with 76 professionals.

Threats to *Reliability validity* concern the possibility to replicate this study. We have attempted to provide all the necessary details needed to replicate our study. We share our full replication package [43].

## 6 RELATED WORK

This section discusses related work about (i) design principles in video games, (ii) video game-related patterns and anti-patterns, (iii) video game development practices, and (iv) studies on video games metadata and user reviews.

### 6.1 Design principles in video game development

Applying design principles in video game development has been extensively studied in literature [4, 6, 16, 31, 42, 45, 48]. Authors defined specific patterns or applied GoF design patterns (DPs) [19] in the context of video game development.

Nystrom [45] proposes a revisited version of some GoF DPs (namely command, flyweight, observer, prototype, singleton, and state), and a set of specific DPs for the video game domain. In particular, Nystrom defines thirteen design patterns grouped into four categories: sequencing patterns (related to time issues), behavioral patterns (to define and refine several behaviors in a way they are easy to maintain), decoupling patterns, and optimization patterns (to speed up a game).

Some of our identified video game smells can be related to Nystrom's DPs. For example, Run-Time Objects smell, *i.e.,* creating components/objects at run-time, is related to Nystrom's optimization patterns. Smells like Weak Temporization, Heavy physics, Heavy Drawing/Rendering, and Complex Scene are related to both sequencing and optimization categories. Instead, Search By ID and Lack of Separation of Concerns can be related to decoupling patterns and behavioral patterns respectively.

Similarly to Nystrom, in his book, Murray [42] proposes various kinds of design solutions specifically for Unity video games, including the use of object pools or virtual controllers that, as explained above, can be suitable solutions for some smells we have identified, *i.e.,* creating components/objects at run-time and lack of separation of concern.

Still on the line of proposing design solutions for video game development, Barakat *et al.* [6] propose to integrate creational and behavioral DPs (namely state, strategy, prototype, and observer) with a specific game design framework to provide the developers with some hints on what DP to use with the main game aspects. Such integration would ease reuse and maintenance tasks. Applying DPs to video game development requires concrete guidelines on how such DPs can be used to solve specific problems. To this extent, Qu *et al.* [48] reported an experience of application of DPs to solve different problems in video game development, including sprite and map management, or handling the game state.

Further work investigated the impact of DPs during game development. Figueiredo *et al.* [16] conduct a controlled experiment which results show how the adoption of GoF patterns has a positive impact on video game development productivity and also reduces the number of lines of code necessary to implement the required features. A broader set of studies on the impact of DPs on internal quality of video games source code has been conducted by Ampatzoglou *et al.* [4, 5, 31]. More precisely, they first study the impact of DPs on the maintainability of two open-source games [4]. Their results indicate that DPs help increasing software maintainability although—differently from the findings of Figueiredo *et al.* [16]—they increase the code base size. Such contrasting results seem to indicate how the impact of DPs highly depends on the context, and they may either contribute to increase or decrease the code size, and, possibly, have a different effect on developers' productivity. Furthermore, Ampatzoglou *et al.* leveraged DPs to implement game rules and logic [31], showing that the use of DPs in this context helps to avoid introducing undesired complexity and to increase reusability, maintainability, and flexibility. Finally, they conduct studies

on the correlation between DP application, software defects, and debugging rate [5]. Their results highlight that even if the overall number of DP instances does not correlate with defect frequency and debugging effectiveness, some specific DPs have a significant impact on the number of reported bugs and debugging rate.

In summary, previous work has studied the quality of video games' source code using conventional metrics, or proposed the use of DPs to solve video game development problems. Work related to smell definition and detection is complementary to studies on the use of DPs in video game development, and has an application when developers do not follow—or tend to deviate from—good design and implementation practice. In that case, the definition of smell catalogs, and, consequently, detectors, can help developers to avoid such bad practices and, where appropriate, refactor the code using the proper design principles defined above.

Other works analyze video game failures and categorize them through taxonomies. Lewis *et al.* [34] proposed a taxonomy of video game failures, capturing both temporal and non-temporal failures. The proposed taxonomy is illustrated with numerous examples, capturing the wide breadth of failures in modern video games. Differently from Lewis *et al.*, our focus is on bad smells rather than failures. Certainly, it is possible that, in some circumstances, bad smells can also induce failures, and therefore the two phenomena can be related and worthwhile to be jointly studied.

### 6.2 Studies on video game patterns and anti-patterns

Some previous works study, or propose, patterns and anti-patterns related to different phases of video game development, *e.g.,* the elicitation of the game ideas/concepts, project management, or coding.

Brandse and Tomimatsu [10] propose six rules that need to be followed when designing players' challenges in video games. The rules concern the deviations of the challenge from the core gameplay, technical implementations, player actions, information, effects on a future challenge, and the advantage of the challenge over the player. Differently from Brandse and Tomimatsu, our work does not deal with the game idea/concept, but rather with its development.

Ullmann *et al.* [57] investigate video game project management anti-patterns using 440 post-mortems problems involving 200 video game projects. The postmortems were collected from 1997 to 2019. They also mapped the identified anti-patterns with traditional software engineering anti-patterns collected from the literature, finding that the most frequent video game project management anti-patterns are *Project Mismanagement, Death March, Shoeless Children, Cover your assets and False Surrogate Endpoint.* Ullmann *et al.* also identify some anti-patterns that do not have a strict correspondence with conventional software anti-patterns. The most frequent ones among such video game-specific anti-patterns are *Feature Creep* (*i.e.,* continuously expanding the initial scope of the game) and *Feature cut* (*i.e.,* failing to prioritize features to be cut to accommodate for another desired feature). While our work focuses on design and implementation smells rather than on management smells, it may be useful, in future work, to investigate the interaction between the two types of smells.

Other work is related to implementation smells [3, 9, 28]. Khanve *et al.* [28] manually analyze the violation of game programming patterns in eight JavaScript games and argue that video games need to be handled differently than conventional programs in terms of code smells. Therefore, we can state that their findings further motivate our work, because they indicate that the detection of conventional code smells is not sufficient to assess the quality of video games' source code.

Vartika and Chimalakonda [3] analyze the commits, issues, and pull requests of 100 open-source GitHub game repositories to understand the implementation issues faced by game developers. Using topic modeling, they categorize these issues and propose a catalog of bad practices to support video game development. Contrary to our catalog which focuses on conceptual bad smells, their

proposed catalog reports implementation issues related to frameworks and libraries used, and faults (*e.g.,* faulty *Game controls*, *Invalid moves* leading to a violation of game rules, or *Wrong logic*). Moreover, their identified categories –because they have been obtained using topic modeling rather than through a qualitative analysis—are not directly linked to specific problems for which concrete solutions exist, contrary to our catalog as discussed in Section 3.

Borrelli *et al.* [9] propose *UnityLinter*, a static analysis tool to detect seven video game-specific smells. The smell types encompass performance, maintainability, and incorrect behavior problems. They validate the smells with a developer survey and detected smell instances using 100 unity open source projects. Their results show that the smells have a prevalence ranging from 39% to 97% and the tool has a precision between 86% and 100% and a recall between 50% and 100%. The main difference with Borrelli *et al.* is that their work is vertical towards the definition, assessment, and detection of seven specific smells, whereas our work aims at creating a broader catalog covering different aspects of video game development. Note that we take also into account smells already defined by Borrelli *et al.*, *e.g.,* Search by String/ID, lack of separation of concern, static coupling, and creating components/objects at run-time. Furthermore, we assessed the importance of the smells. To the best of our knowledge, except Borrelli *et al.*, no previous authors assessed the importance of video game bad smells.

### 6.3 Studies about video game development practices

Previous works empirically investigated video game development practices and identified commonalities and differences with canonical software development.

Specifically, Marklund *et al.* [39] conduct a literature review of empirical studies on game development practices. Their review covers 48 papers published between 2006 and 2016, and reveals the difficulty of accurately planning a game development project because of soft requirements. By conducting interviews and a survey, Murphy-Hill *et al.* [41] uncover substantial differences between video game development and software development. Their work also calls for more research on video game development to help developers cope with several challenges related to development and testing.

Other studies conduct surveys to investigate developers' motivation and interaction in video game development. For example, Giannakos *et al.* [20] conduct a study involving 78 students aged between 12 and 17 years old, to identify factors that motivate them to participate in creative game development activities. Their results indicate that expected effort and performance expectancy affect students' decision to participate in such development activities. Kamienski and Bezemer [27] examine how game developers use Q&A forums. They found that game developers tend to reduce their usage of Q&A Websites as they became more experienced. On summary, previous studies highlight peculiarities of video game development with respect to conventional software development. To that respect, our study shows that video game design and implementation choices may suffer for specific smells, coded in our catelog.

### 6.4 Studies on game metadata and reviews

Finally, there are studies about video game metadata, release notes, and user reviews on distribution platforms such as Steam.

Lin *et al.* examine the evolution of video games by mining data from the Steam platform [35–37]. More precisely, they investigate the mechanism for collecting feedback through early-access games. Their results indicate that developers update early access games more frequently but these early access games tend to get a small number of reviews [35]. Furthermore, they study the relationship between reviews and several game-specific characteristics. They conclude that the

feedback obtained from both positive and negative reviews should be considered by the developers [37].

Also by looking at metadata on distribution platforms, Lee *et al.* conduct empirical studies on game modifications (mods) [32, 33]. In particular, they analyze the metadata of mods and features characterizing the mods. Their results indicate that popular mods tend to have a high-quality description and promote community contribution [33], and that providing official support for mods is beneficial to improve the quality of the mods [32].

Other studies leverage platform metadata to investigate game quality. Vu and Bezemer [59] investigate the characteristics of game jams, hackathon-like events for games. Their results show that quality description has a positive contribution to popularity and game ranking.

Our work differs from these aforementioned studies, because they look at the video game (and its evolution) from a user (gamer) perspective, whereas we analyze bad smells occurring during video game design and implementation. We aim to understand the perceived relevance of identified bad smells by developers and not only from gamers' perspectives. Nevertheless, it might be interesting, as future work, to investigate the extent to which game smells have an impact on the quality of the game from a user's perspective.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we aimed at creating a catalog of video game development bad smells. To this aim, we first queried 13 popular game development discussion forums as well as the Google search engine to obtain candidate discussions. Then, we manually analyzed a statistically significant sample of 572 discussions and followed a cooperative card sorting approach [51] to elicit a catalog of bad smells. As a result, we derived a catalog of 28 bad smells, organized into 5 categories, covering problems related to game design and logic, physics, animation, rendering, or multiplayer.

Then, we have assessed the perceived relevance of such bad smells by surveying 76 game development professionals. Overall, the surveyed professionals agreed about the relevance of most of the identified bad smells, although at the same time they pointed out some cases of bad smells that, in their development context, were not considered particularly critical. In essence, there are a series of factors that determine the severity of a smell, and its need for removal:

- *Its effect on the performance and gameplay:* developers are often fine to accept maintainability problems, while smell effects visible from the player's side are considered as a priority;
- *The smell magnitude matters:* this concept, already true for traditional code bad smells [49] becomes even more important for video game bad smells, especially because performance-worsening smells create a tangible effect only if they are particularly severe.
- *The smell may depend on the type of video game:* some problems may occur only in multiplayer games, or games targeting certain devices (*e.g.,* mobile games).

Based on the study findings, we have formulated recommendations for developers, educators, and researchers, to help improve video game development and prevent the occurrences of the identified bad smells.

This work opens the road towards several pieces of future research. First, it might be worthwhile to develop detectors for the identified smells, extending a preliminary detector that has been already developed for five types of smells [9], but also tools, integrated into the IDE, able to automatically propose an alternative solution or a refactoring. Second, similarly to what has been done in the past, it may be interesting to perform studies to determine how these smells evolve over time, in terms of introduction and removal [54], how different types of smells interact (*e.g.,* game-specific smells with other smells such as conventional code smells or management-related smells), but

also to determine whether some of these smells may induce bugs or other (*e.g.*, non-functional) problems, such as performance degradation.

## ACKNOWLEDGMENTS

The authors would like to thank the participants to the survey questionnaire.

## REFERENCES

[1] Rider - Fast & powerful cross-platform .NET IDE. https://www.jetbrains.com/rider/. Accessed: 2022-02-22.

[2] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 181–190. https://doi.org/10.1109/CSMR.2011.24

[3] Vartika Agrahari and Sridhar Chimalakonda. A Catalogue of Game-Specific Anti-Patterns. In *ISEC 2022: 15th Innovations in Software Engineering Conference, Gandhinagar, India, February 24 - 26, 2022*. ACM, 8:1–8:10. https://doi.org/10.1145/3511430.3511436

[4] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Inf. Softw. Technol.* 49, 5 (2007), 445–454. https://doi.org/10.1016/j.infsof.2006.07.003

[5] Apostolos Ampatzoglou, Apostolos Kritikos, Elvira-Maria Arvanitou, Antonis Gortzis, Fragkiskos Chatziasimidis, and Ioannis Stamelos. An empirical investigation on the impact of design pattern application on computer game defects. In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, MindTrek 2011, Tampere, Finland, September 28-30, 2011*. 214–221. https://doi.org/10.1145/2181037.2181074

[6] Nahla H. Barakat. A Framework for integrating software design patterns with game design framework. In *Proceedings of the 2019 8th International Conference on Software and Information Engineering, ICSIE 2019, Cairo, Egypt, April 09-12, 2019*. ACM, 47–50. https://doi.org/10.1145/3328833.3328871

[7] Fabian Beck. Analysis of Multi-dimensional Code Couplings. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 560–565. https://doi.org/10.1109/ICSM.2013.96

[8] Markus Borg, Vahid Garousi, Anas Mahmoud, Thomas Olsson, and Oskar Stålberg. Video Game Development in a Rush: A Survey of the Global Game Jam Participants. *IEEE Trans. Games* 12, 3 (2020), 246–259. https://doi.org/10.1109/TG.2019.2910248

[9] Antonio Borrelli, Vittoria Nardone, Giuseppe A. Di Lucca, Gerardo Canfora, and Massimiliano Di Penta. Detecting Video Game-Specific Bad Smells in Unity Projects. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 198–208. https://doi.org/10.1145/3379597.3387454

[10] Michael Brandse and Kiyoshi Tomimatsu. Empirical Review of Challenge Design in Video Game Design. In *HCI International 2013 - Posters' Extended Abstracts - International Conference, HCI International 2013, Las Vegas, NV, USA, July 21-26, 2013, Proceedings, Part I (Communications in Computer and Information Science)*, Vol. 373. Springer, 398–406. https://doi.org/10.1007/978-3-642-39473-7_80

[11] Lionel C. Briand, Walcélio L. Melo, and Jürgen Wüst. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. *IEEE Trans. Software Eng.* 28, 7 (2002), 706–720. https://doi.org/10.1109/TSE.2002.1158285

[12] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons, Inc., 1998, USA.

[13] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895

[14] Peter M Chisnall. Questionnaire design, interviewing and attitude measurement. *Journal of the Market Research Society* 35, 4 (1993), 392–393.

[15] Mohamed Raed El aoun, Heng Li, Foutse Khomh, and Moses Openja. Understanding Quantum Software Engineering Challenges An Empirical Study on Stack Exchange Forums and GitHub Issues. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 343–354. https://doi.org/10.1109/ICSME52107.2021.00037

[16] Roberto Tenorio Figueiredo and Geber Lisboa Ramalho. GOF design patterns applied to the Development of Digital Games. In *Proceedings of SBGames 2015, November 11th - 13th, 2015, Teresina, Brazil*.

[17] Martin Fowler. Refactoring: Improving the Design of Existing Code. *Extreme Programming and Agile Methods–XP/Agile Universe 2002* (2002), 256.

[18] GameIndustry.biz. Global games market value rising to $134.9bn in 2018. https://www.gamesindustry.biz/articles/2018-12-18-global-games-market-value-rose-to-usd134-9bn-in-2018 (Last access: 01/01/2022).

[19] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[20] Michail N. Giannakos and Letizia Jaccheri. From players to makers: An empirical examination of factors that affect creative game development. *Int. J. Child Comput. Interact.* 18 (2018), 27–36. https://doi.org/10.1016/j.ijcci.2018.06.002

[21] GlobalWebIndex - GWI. The Gaming Playbook, https://www.gwi.com/reports/the-gaming-playbook (Last access: 23/08/2022).

[22] Robert M Groves, Floyd J Fowler Jr, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey methodology*. John Wiley & Sons, 2011.

[23] Alaleh Hamidi, Giuliano Antoniol, Foutse Khomh, Massimiliano Di Penta, and Mohammad Hamidi. Towards Understanding Developers' Machine-Learning Challenges: A Multi-Language Study on Stack Overflow. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021*. IEEE, 58–69. https://doi.org/10.1109/SCAM52516.2021.00016

[24] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395

[25] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. Mining the relationship between anti-patterns dependencies and fault-proneness. *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013* (2013), 351–360. https://doi.org/10.1109/WCRE.2013.6671310

[26] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John C. Grundy. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 432–443. https://doi.org/10.1109/MSR52588.2021.00055

[27] Arthur V. Kamienski and Cor-Paul Bezemer. An empirical study of Q&A websites for game developers. *Empir. Softw. Eng.* 26, 5 (2021), 115. https://doi.org/10.1007/s10664-021-10014-4

[28] Vaishali Khanve. Are existing code smells relevant in web games? an empirical study. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 1241–1243. https://doi.org/10.1145/3338906.3342504

[29] Foutse Khomh and Yann-Gaël Guéhéneuc. Do Design Patterns Impact Software Quality Positively?. In *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*. IEEE Computer Society, 274–278. https://doi.org/10.1109/CSMR.2008.4493325

[30] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empir. Softw. Eng.* 17, 3 (2012), 243–275. https://doi.org/10.1007/s10664-011-9171-y

[31] Xeni-Christina Kounoukla, Apostolos Ampatzoglou, and Konstantinos Anagnostopoulos. Implementing Game Mechanics with GoF Design Patterns. In *Proceedings of the 20th Pan-Hellenic Conference on Informatics, Patras, Greece, November 10-12, 2016*. ACM, 30. https://doi.org/10.1145/3003733.3003779

[32] Daniel Lee, Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Building the perfect game - an empirical study of game modifications. *Empir. Softw. Eng.* 25, 4 (2020), 2485–2518. https://doi.org/10.1007/s10664-019-09783-w

[33] Daniel Lee, Gopi Krishnan Rajbahadur, Dayi Lin, Mohammed Sayagh, Cor-Paul Bezemer, and Ahmed E. Hassan. An empirical study of the characteristics of popular Minecraft mods. *Empir. Softw. Eng.* 25, 5 (2020), 3396–3429. https://doi.org/10.1007/s10664-020-09840-9

[34] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *International Conference on the Foundations of Digital Games, FDG '10, Pacific Grove, CA, USA, June 19-21, 2010*. ACM, 108–115. https://doi.org/10.1145/1822348.1822363

[35] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. An empirical study of early access games on the Steam platform. *Empir. Softw. Eng.* 23, 2 (2018), 771–799. https://doi.org/10.1007/s10664-017-9531-3

[36] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Identifying gameplay videos that exhibit bugs in computer games. *Empir. Softw. Eng.* 24, 6 (2019), 4006–4033. https://doi.org/10.1007/s10664-019-09733-6

[37] Dayi Lin, Cor-Paul Bezemer, Ying Zou, and Ahmed E. Hassan. An empirical study of game reviews on the Steam platform. *Empir. Softw. Eng.* 24, 1 (2019), 170–207. https://doi.org/10.1007/s10664-018-9627-4

[38] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Trans. Software Eng.* 34, 2 (2008), 287–300. https://doi.org/10.1109/TSE.2007.70768

[39] Björn Berg Marklund, Henrik Engström, Marcus Hellkvist, and Per Backlund. What Empirically Based Research Tells Us About Game Development. *Comput. Games J.* 8, 3-4 (2019), 179–198. https://doi.org/10.1007/s40869-019-00085-1

[40] Robert C Martin, James Newkirk, and Robert S Koss. *Agile software development: principles, patterns, and practices*. Vol. 2. Prentice Hall Upper Saddle River, NJ, 2003.

[41] Emerson R. Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?. In *36th International Conference on*

*Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 1–11. https://doi.org/10.1145/2568225.2568226

[42] Jeff W. Murray. *C# Game Programming Cookbook for Unity 3D*. CRC Press, 2014, New York.

[43] Vittoria Nardone, Biruk Asmare Muse, Mouna Abidi, Foutse Khomh, and Massimiliano Di Penta. *Video Game Bad Smells: What they are and how Developers Perceive Them - Online dataset*. https://doi.org/10.5281/zenodo.6327678

[44] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. *Autom. Softw. Eng.* 29, 1 (2022), 8. https://doi.org/10.1007/s10515-021-00313-x

[45] Robert Nystrom. *Game Programming Patterns* (1st edition ed.). Lightning Source Inc., 2014.

[46] Moses Openja, Bram Adams, and Foutse Khomh. Analysis of Modern Release Engineering Topics : - A Large-Scale Study using StackOverflow -. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 104–114. https://doi.org/10.1109/ICSME46990.2020.00020

[47] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 29:1–29:30. https://doi.org/10.1145/3408302

[48] Junfeng Qu, Yinglei Song, and Yong Wei. Applying design patterns in game programming. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, 1.

[49] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using History Information to Improve Design Flaws Detection. In *8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceedings*. IEEE Computer Society, 223–232. https://doi.org/10.1109/CSMR.2004.1281423

[50] Shriram Shanbhag, Sridhar Chimalakonda, Vibhu Saujanya Sharma, and Vikrant Kaulgud. Towards a Catalog of Energy Patterns in Deep Learning Development. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022)*. Association for Computing Machinery, New York, NY, USA, 150–159. https://doi.org/10.1145/3530019.3530035

[51] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[52] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock A. Licorish, and Aiko Yamashita. A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites. *Inf. Softw. Technol.* 125 (2020), 106333. https://doi.org/10.1016/j.infsof.2020.106333

[53] Christoph Treude, Fernando Marques Figueira Filho, and Uirá Kulesza. Summarizing and measuring development activity. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 625–636. https://doi.org/10.1145/2786805.2786827

[54] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Trans. Software Eng.* 43, 11 (2017), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105

[55] Gias Uddin, Olga Baysal, Latifa Guerrouj, and Foutse Khomh. Understanding How and Why Developers Seek and Analyze API-Related Opinions. *IEEE Trans. Software Eng.* 47, 4 (2021), 694–735. https://doi.org/10.1109/TSE.2019.2903039

[56] Gias Uddin, Fatima Sabir, Yann-Gaël Guéhéneuc, Omar Alam, and Foutse Khomh. An empirical study of IoT topics in IoT developer discussions on Stack Overflow. *Empir. Softw. Eng.* 26, 6 (2021), 121. https://doi.org/10.1007/s10664-021-10021-5

[57] Gabriel Cavalheiro Ullmann, Cristiano Politowski, Yann-Gaël Guéhéneuc, Fábio Petrillo, and João Eduardo Montandon. Video Game Project Management Anti-patterns. *CoRR* abs/2202.06183 (2022). arXiv:2202.06183 https://arxiv.org/abs/2202.06183

[58] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 2–11.

[59] Quang N. Vu and Cor-Paul Bezemer. An Empirical Study of the Characteristics of Popular Game Jams and Their High-ranking Submissions on itch.io. In *FDG '20: International Conference on the Foundations of Digital Games, Bugibba, Malta, September 15-18, 2020*. ACM, 20:1–20:11. https://doi.org/10.1145/3402942.3402981

[60] Peter Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*. IEEE Computer Society, 77–84. https://doi.org/10.1109/.2001.914971

[61] Aiko Fallas Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 682–691. https://doi.org/10.1109/ICSE.2013.6606614

[62] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empir. Softw. Eng.* 25, 2 (2020), 1095–1135. https://doi.org/10.1007/s10664-019-09785-8