What Went Wrong: A Taxonomy of Video Game Bugs

Chris Lewis, Jim Whitehead, Noah Wardrip-Fruin
University of California, Santa Cruz
1156 High St, Santa Cruz, California, USA
{cflewis,ejw,nwf}@soe.ucsc.edu

ABSTRACT

Video games are complex, emergent systems that are difficult to design and test. This difficulty invariably leads to failures being present in the game, negatively impacting the play experience of some. We present a taxonomy of possible failures, divided into temporal and non-temporal failures. The taxonomy can guide the thinking of designers and testers alike, helping them expose bugs in the game. This will lead to games being better tested and designed, with fewer failures when released.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging; K.8.0 [**Personal Computing**]: General

Keywords

video game, failure, fault, error, bug, taxonomy

1. INTRODUCTION

Most gamers think game bugs are bad. Bugs can cause a game to crash, disturb game balance, or ruin an engaging game session. However, there is a sub-community of gamers who actually *like* bugs. Not in the resigned sense of, "I like this game, bugs and all." Rather, in the more active sense of liking to find bugs, being motivated by the potential that a game bug might exist, and hence so motivated as to spend hours trying various quirky button and movement combinations in hopes of revealing a game bug. For these select few, game bugs are a passion, a hobby, with one imperative: videos showing bugs must be posted on YouTube.

YouTube's bug videos are a rich resource that mix creativity, subversiveness and pure chance. The videos available provide a startling amount of coverage; far more than any single research group could ever hope to expose personally. We became motivated to understand the variety of video game bugs, which could form the groundwork for unravelling the complexities of testing and verifying modern, emergent gameplay. YouTube presented a perfect starting point.

Figure 1: An image of a YouTube video showing the "Jesus Shot," a failure from *Tiger Woods '08* where Tiger Woods is able to walk on water.

After watching many videos demonstrating curious bugs across a wide range of games and genres, we began to notice patterns emerging. Several of the videos showed bugs pertaining to defects in collision detection, while others had unexpected emergent behavior of physics engines, and so forth. These similarities indicated that a categorization of these buggy gameplay experiences was possible, leading to new methods of analyzing and solving video game bugs.

In some areas, particularly that of computer science, taxonomies have been used to create an organized, systematic approach to understanding a wide variety of data. Vijayaraghavan & Kaner found taxonomies commonly used in the areas of security, computer system architectures and computer inputs [16]. They showed that the introduction of a bug taxonomy aided testers in focusing their thoughts, allowing them to generate new test cases and creating greater coverage of the system. Such taxonomies can also be used to validate test plan coverages and help newcomers understand the problem areas.

In this paper, we present a taxonomy of video game failures, which exists as a living document online, complete with

FDG 2010 June 19-21, Monterey, CA, USA

verifiable examples in a video archive [2]. This taxonomy will not only help guide current human testing, but also provide a framework to validate the coverage of new testing paradigms that may emerge.

2. TERMINOLOGY

This paper discusses a separation between a game design *specification* and its *implementation*, so it is important to understand the scope of these two terms:

Specification The game design specification, or game design document, is the game designer's formal specification for how the game and its systems should operate to create the intended gameplay experience.

Implementation The implementation of a game is any asset created in order to make the specification into a playable game: source code, art, level designs, etc.

Throughout our discussion, we will formalize the term "bug" into three layers: *fault*, *error* and *failure*. Let us start with an example, as seen in Figure 1 (Category: *Object out of bounds for any state*). This infamous bug is referred to as the "Jesus Shot," prompting a humorous video response from EA Sports showing the real-life Tiger Woods performing a shot out of a lake, claiming, "It's not a glitch. He's just that good." [11]

Fault The lake has a parameter incorrectly set that makes it a solid object that objects can land on, implicitly making it part of the course.

Error The collision detection system incorrectly believes the ball is in-bounds.

Failure Tiger appears to be walking on water when he takes the next shot.

This example shows the distinct layers of consideration we can give a "bug." The terms arise from a formal terminology for software bugs categorized by Avižienis & Laprie [3]:

Fault A *fault* is a phenomenon that leads to an *error* in the system. Faults can occur in computer hardware, but in this paper, we will focus on faults that are human-created: either as a mistake during the design or implementation of a video game.

Error An *error* is the manifestation of a fault during the software's execution that creates a state that could contribute to a failure.

Failure A *failure* is a user-observable deviation from the expected behavior of the system.

For consistency and clarity, we apply these terms equally to both the game implementation and the game specification. We understand these terms can sound unnecessarily pejorative, but they are the accepted terms in Software Engineering, and we use them here to maintain consistency.

Not all faults will inevitably lead to errors, as some dormant faults may never be exercised by the game during operation. Similarly, not all errors will lead to failures, as some errors in the game state might not have any adverse effect at a user-visible level.

Note that we are unable to categorically state how the failure occurred. Both the fault and error we posited in our *Tiger Woods '08* example are just one of many possibilities. We must treat faults and errors as a black-box. However, we do not comment on all the failures that we observe, focusing only on those that arise from a fault in the implementation, not a fault in the design document. Implementation failures are a *deviation* of the game's operation from the original game specification. These failures create unexpected "glitches" in the game world, or provide obvious ways of breaking the game from what was intended. Classic examples include escaping the boundaries of the game world (Category: *Object out of bounds for any state*), pathfinding leading characters into walls (Category: *Artificial stupidity*) or the camera pointing the wrong way (Category: *Lack of required information*). These are often very easy to recognize, and we are able to accurately pinpoint them as problems arising from the implementation.

We disregard failures that do not exhibit obvious implementation faults, as they may have been "faults" in the game specification itself. Issues in the game specification are often subjective, and, as such, have no reasonable means of being classified as a fault. For example, while some players consider bunny-hopping to attain greater speed in a first-person shooter as being a game design failure (Category: *Invalid position over time*), others believe it to be an intrinsic part of the gameplay. We discuss game design failures in more detail in Section 5.

3. RELATED WORK

There have been several uses of bug taxonomies. One of the most influential taxonomies was written by Beizer, who categorizes general software engineering bugs, such as "Requirements, Features and Functionality bugs," "Structural bugs" and "Data bugs" [5]. He comments on the nature of bug taxonomies, and notes that there is no one taxonomy that can hope to provide utility in all cases, stating that, "There is no universally correct way to categorize bugs," and that there are a potentially infinite number of ways to form a categorization. Beizer's taxonomy is written for programmers who have access to source code, which is commonly not the case for game testers. Instead, we must create our own taxonomy that black-box game testers can utilize, as well as create something that reflects the domain-specific problems we encounter in video games.

The largest repository of bugs is the Common Vulnerability Enumeration, containing over 40,000 issues [1]. However, the CVE is specifically focused on finding bugs that are stepping stones to security compromises, and our taxonomy requires the repository have a wider breadth to be useful in this domain.

Bainbridge & Bainbridge [4] have previously attempted to classify video game "glitches." They categorize video game bugs by perceived cause (i.e. the fault(s) that led to the failure), such as a hole in a level map, or undeleted test code. They note that their taxonomy is a "best-guess," as they are unable to inspect the source code and other game assets. This leads to a taxonomy that we find unsatisfying. For example, one category named "Bizarre maneuver," is classified as "requires... intentional behavior on the part of the player, involving several steps and going beyond what the game designers expected the player to do." We believe that such emergent (albeit perhaps subversive) play should be encouraged, and are hesitant to apportion blame to players themselves. As mentioned in Section 2, we do not attempt to

categorize causes, only observable failures, in order to limit the guesswork involved.

Trade publication *Game Developer* often carries postmortems of a game's development. Tschang surveyed these postmortems, counting the number of positive and negative references to aspects of game development. He notes that testing is often mentioned, in equal parts positive and negative [15], commenting that, "This confirms testing is very critical to video games."

Hind & Bell discuss establishing bug criteria so the testing team are able to correctly estimate their severity [6]. They take an operational view of bugs, classifying them by importance, ranging from a bug that blocks the release of a game, to one that is marked as not needing to be fixed. They note that their examples for the severity of a bug "suggest a graduation from obvious bug to subjective opinion."

4. TAXONOMY

Our taxonomy can be seen in Figure 2. We will describe it in more detail in this section.

4.1 Coverage

Lough's investigation into taxonomies resulted in eighteen different properties that taxonomies should satisfy [10]. Using his terminology, our taxonomy aims to be comprehensible, objective (unbiased), specific (does not contain categories too broad), unambiguous and useful (provides value to a community). We invite the academic and games communities to continue developing the taxonomy as a living document, hosted online [2].

We have designed our taxonomy to be as mutually exclusive as possible, but we do not believe that this is a property that can be realistically met in the video game domain. The complexity of the various sub-systems at work means various errors and failures will inevitably be cross-cutting, causing failures to be placed in multiple categories. For example, a Non-Player Character (NPC) blocking a doorway that a player needs to move through creates a failure of both "Artificial stupidity" and "Action not possible."

4.2 Methodology

To investigate the wide variety of possible failures, we turned to various Internet articles and communities dedicated to video game issues. The ability to upload evidence of the failures to YouTube allowed us to find many examples. Searching YouTube for "video game glitches" or "video game bugs" produces hundreds of related videos.

As we surveyed more and more videos, we began to form categorizations of bugs, steadily redrawing lines and generalizing categories as necessary. Failures in Figure 2 are categorized into the leaf nodes of the tree. Branch nodes are simply used as an organizational tool.

As a way of ensuring that our taxonomy aligned with the personal experiences of members of the games industry, we gave the taxonomy to two experienced game designers for review. The taxonomy was given to Damian Isla, Director of Technology of Moonshot Games, and Ken Hullett, PhD student at UC Santa Cruz, and former professional game designer at companies such as Activision and Novalogic.

4.3 Temporal and non-temporal failures

The first and main tier in the taxonomy is a split between temporal and non-temporal failures. Temporal failures are those which require some knowledge of previous game state in order to accurately categorize them. Non-temporal failures can be found by inspecting the game state at any point in time.

For example, consider two possible failures of Mario's jumping in *Super Mario Bros.* A non-temporal example would be Mario jumping so high as to fly off-screen: as soon as he is above a certain height, the game has exhibited a failure (Category: *Object out of bounds for any state*). Now consider Mario jumping at the right height, but is able to hover in mid-air (Category: *Invalid position over time*). Any single inspection of the game state will reveal that Mario is at a valid game height. It is only by collecting recordings of his height *over time* that it is possible to detect that Mario is hovering. As a result, we are able to categorize invalid *movements*, not just invalid *positions*.

4.4 Categories

Each category from the taxonomy in Figure 2 is elaborated on in Tables 1 and 2, providing examples and links to videos displaying the failures in action. More video examples can be found at our taxonomy web site.

4.4.1 Object out of bounds

The first non-temporal category, *Object out of bounds for any state* is a classification of an object being outside of the world boundaries. This category encompasses many common types of failures, such as escaping a map or falling through the floor. It also catches failures that are within world boundaries but simply non-sensical in the world fiction, such as the ability to walk on water or up the side of a building (if the game does not allow such functionality). Related to this is *Object out of bounds at a specific state*. This category is similar to being out of bounds for any state, except that the position is valid in certain contexts. Common examples include escaping a scripted sequence, such as an in-game narrative exposition or escaping a combat arena, such as a boss battle where you are supposedly locked in.

4.4.2 Invalid graphical representation

An *Invalid graphical representation* occurs when a certain aspect of the world state is being rendered incorrectly. Table 1 presents a failure where a character is performing a swimming animation when on land, but other failures include placing a certain item of clothing on a character in *Rock Band*, but that item not appearing during play.

4.4.3 Invalid value change

Invalid value change is a broad term that describes any game event that changes some form of counter in an unexpected way, such as a bullet that should remove health not doing so or collecting a coin that changes the score by 100 instead of 1.

4.4.4 Artificial stupidity

Artificial stupidity is another broad category that catches bugs related to an NPC performing some act that breaks the illusion of intelligence. Common examples include characters not responding to being shot at, blocking doorways or walking into walls. To be more specific, this category contains all bugs where the AI does not meet the expectation of the player. Player expectations for different NPCs in different games vary. In a first-person shooter, one expects

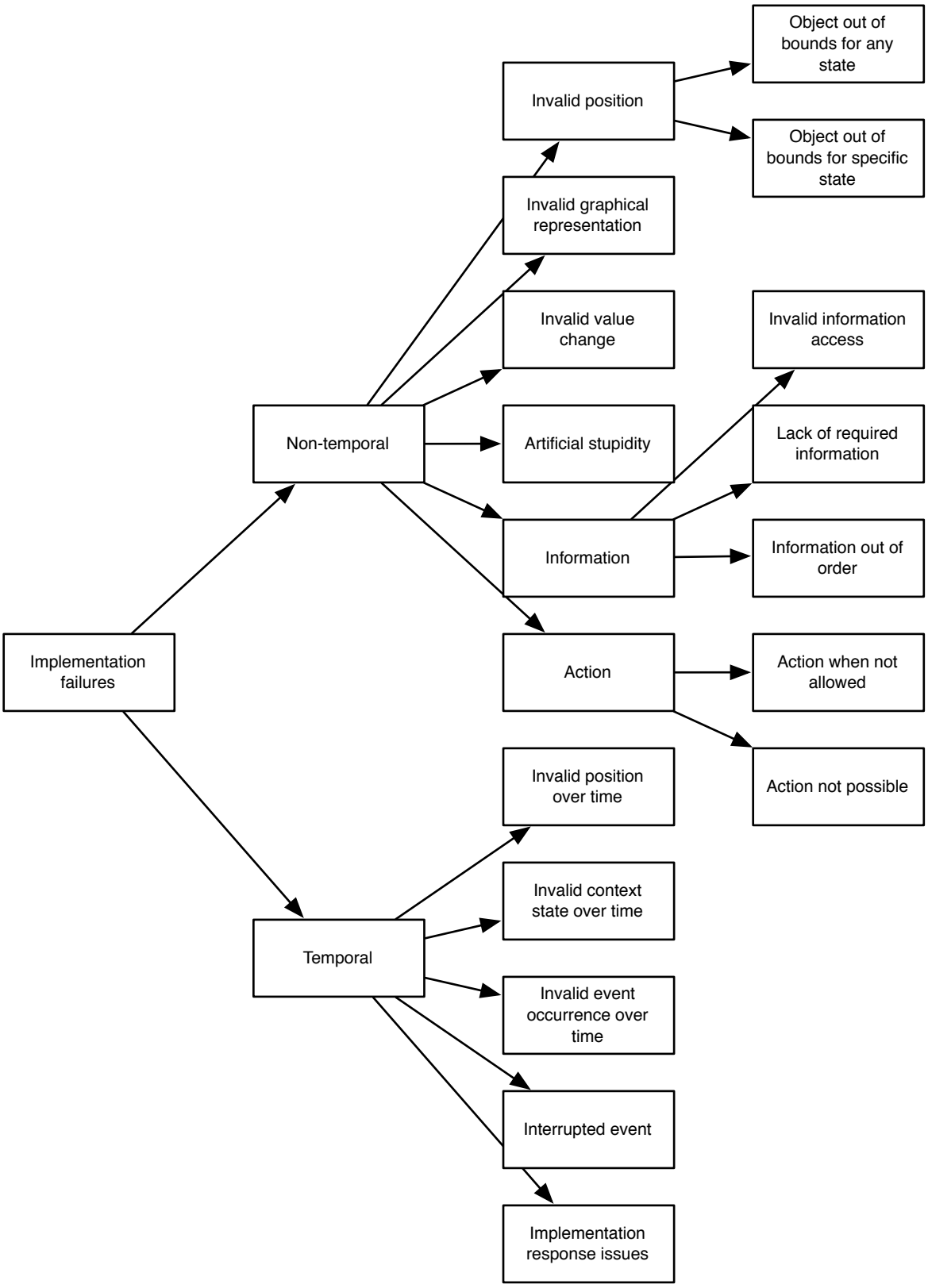

Figure 2: A taxonomy of video game failures. The taxonomy is ordered from left-to-right, beginning at "Implementation failures." Non-temporal failures can be inspected at any point in time and be classified as a failure, whereas temporal failures require knowledge of previous states before being able to confirm that a failure has occurred.

Category	Description	Game	Example	Reference ID
Object out of bounds for any state	For all game states, an object is at an invalid world position	*Left 4 Dead*	Falling through the floor of an elevator	`l4d_fall.mp4`
Object out of bounds at a specific state	Only during certain game states, an object is at an invalid world position	*Left 4 Dead*	Escaping the boundaries of a scripted sequence by using clipping errors	`l4d_sequence.mp4`
Invalid graphical representation	An object appears in the game world incorrectly	*Elder Scrolls IV: Oblivion*	Showing a swimming animation when on land	`oblivion_swimming.mp4`
Invalid value change	The intended internal value change by an event in-game is incorrect	*Grand Theft Auto IV*	Bullets not doing any damage	`gta4_bullets.mp4`
Artificial stupidity	AI displays poor reasoning	*Knights of the Old Republic*	AI ally walks onto a land mine	`kotor_mine.jpg`
Invalid information access	Player is afforded information that she shouldn't have	*Call of Duty: World at War*	Seeing through walls	`codww_seethrough.mp4`
Lack of required information	Player is not afforded information that she needs	*Mass Effect*	Camera pointing in the wrong direction	`me_camera.mp4`
Information out of order	Player acquires information about the game world in an unexpected order	*Knights of the Old Republic*	Characters request player to undertake already completed quest	`kotor_info1.mp4`
Action when not allowed	Object in game world can take action when action is supposedly paused (eg. cut-scene, game pause)	*Goldeneye*	Being able to shoot a character during a cut-scene	`goldeneye_fire.mp4`
Action not possible	Object in the game world cannot take action when it is allowed to	*Pokémon Gold*	Can't pick up item	`pokemon_action.mp4`

Table 1: Descriptions and examples of non-temporal failures. Reference IDs can find videos and images online at `http://www.zenetproject.com/video:[ID]` (without square brackets).

a certain degree of rashness from enemy NPCs in order to make them easy to defeat in combat, but not so much so that they are suicidal (unless this fits with the world fiction, such as if they are zombies). Conversely, one expects a certain degree of restraint by ally NPCs so they do not die quickly or steal kills from the player, but not so much that they are deemed useless in battle. In contrast, players do not expect such complexity of thought in the enemies of a platformer; moving backwards and forwards is intelligence enough, and thus would not be categorized as a failure. We would also not categorize a failure if the AI has access to perfect information, unless the player is able to spot the AI cheating, shattering the illusion of fairness. Any time the AI fails to meet expectations, we classify this as *Artificial stupidity.*

We acknowledge this category is quite broad, and may well be deserving of a taxonomy of its own. However, in the context of this taxonomy, we believe this is a suitable abstraction level for this subject.

4.4.5 Information

The *Invalid information access* category encompasses failures that allow the player to gain more information than is expected by the game design. This category includes seeing through walls or gaining complete information on a game map that should have a fog of war.

Category	Description	Game	Example	Reference ID
Invalid position over time	An object is moving in an invalid way	*Grand Theft Auto IV*	Being launched into the air, moving too quickly	`gta4_movement.mp4`
Invalid context state over time	An object is placed into a state for an incorrect amount of time	*Street Fighter II*	Character is stunned for entire round	`sf2_stunned.mp4`
Invalid event occurrence over time	An event is allowed to happen too often or too infrequently	*Left 4 Dead*	Cycling weapons allows melee hits to occur too quickly	`l4d_sequence.mp4`
Interrupted event	An event that was in action has now stopped before ending	*Call of Duty 4: Modern Warfare*	Characters stop moving in a cut scene	`cod4_stall.mp4`
Implementation response	Game does not function at the speed required	*Rainbow Six Vegas 2*	Lag when shooting particular NPC	`rainbow6_lag.mp4`

Table 2: Descriptions and examples of temporal failures. Reference IDs can find videos and images online at `http://www.zenetproject.com/video:[ID]` (without square brackets).

The counterpart to *Invalid information access* is *Lack of required information*, where the player should be privy to some information that she is not. This category is actually quite varied: it classifies a malfunctioning camera (not affording the player visual information required to play), audio not playing (not affording the player notification of events or story information) or some graphical information not appearing on-screen (this could also be categorized as *Invalid graphical representation*).

The final Information category is *Information out of order*, which classifies events when the player has received information in an unexpected order. This failure is most likely to occur in role-playing games, where players often have multiple methods of receiving the same information, such as interrogating an NPC, reading a note, or stumbling across an object. Once the information is acquired, it is possible that other aspects of the game world are unaware the player has that knowledge, and continue to act as if the player does not have that information. When related to an NPC specifically, such a failure can also be classified under *Artificial stupidity*.

4.4.6 Action

Action when not allowed includes actions being taken while the game is paused or playing a cut-scene, as well as scripts executing when they are not allowed to yet, such as a wedding scene in *Fallout 3* beginning when the bride and groom have not arrived at the church[1]. *Action not possible* is the realm of actions that the game should allow, but are not being executed, such as a light switch not activating the light, or a weapon being dropped on the floor that the player cannot pick up.

4.4.7 Invalid position over time

[1]`http://www.zenetproject.com/video:fallout3_action.mp4`

Moving on to temporal failures, the first category is *Invalid position over time*. Often, this category describes invalid movements, such as rapid accelerations or hovering in the air, but can also include objects teleporting around the world due to poor physics or faulty world updates. This category also describes the *lack* of expected movement, such as an NPC that is in a pathfinding mode but is stuck against a piece of geometry and hasn't been able to move for a period of time.

4.4.8 Invalid context state over time

Invalid context state over time applies to objects that stay in a state for too long or too infrequently. State is used only to mean the user-observable characteristics that an object is showing, not the actual flags used in the implementation. We include such failures as being stunned for too long, an NPC that stays in an alert mode when it should have switched to a patrol mode, or a character in an MMO that has remained dead and not respawned correctly.

4.4.9 Invalid event occurrence over time

Invalid event occurrence over time classifies discrete events that occur too frequently or too infrequently. Discrete events include incidents such as firing a gun, throwing a punch, sacking a quarterback and performing a jump. This category is curious in that there are probably far more examples in this category from a game design specification not working as intended than there are implementation faults. Examples of game design possibilities include any method of gaining more resources than intended in an RTS game, or flash knockouts in *UFC 2009 Undisputed* being too common. A concrete example of an actual implementation failure is provided in Table 2, where the ability to quickly switch weapons allows melee hits to occur too quickly.

4.4.10 Interrupted event

An *Interrupted event* is any action in game that was previ-

ously operating but has terminated abruptly against expectations, such as a sound effect partially playing or characters that were moving stopping suddenly. For example, if a character is speaking to you and the dialogue sound file is cut-off for no apparent reason, then this is an interrupted event. If the dialogue is cut-off because the character is killed half-way through speaking, then this meets the player's expectations of the game world, and thus is not a failure.

4.4.11 Implementation response issues

Implementation response is the category most closely aligned to how the game interacts with the base hardware. This category covers failures where some aspect of the hardware is not performing at an optimal speed. Such failures include network lag, input lag (time from pressing a button to something occurring in game) or frame rate fluctuations. Note that this category only covers issues due to errors in the software code; we do not include failures that are caused by errors in the hardware or network itself. Our example in Table 2, *Rainbow Six Vegas 2*, clearly shows lag due to some implementation failure. The player is able to move correctly before shooting a particular NPC which then causes the game to exhibit lag. Turning around and facing the other direction causes the lag to stop again, eliminating other, more transient explanations, such as network congestion.

5. VIDEO GAME DESIGN FAILURES

In the creation of this taxonomy we have taken steps to only consider failures caused by implementation faults, and not faults in the game design itself. Without the ability to consult the game design document, we cannot ascertain whether the on-screen behavior is intended, and we do not wish to make a value judgement as to whether some operation of the game design is negative. For example, snaking in *F-Zero GX* and rocket jumping in *Quake* appear to be failures. However, they have been claimed as intended tactics for advanced players [7, 8]. Whether these tactics were intentional before release is, as always, up for debate!

Even aspects of gameplay that appear to be important cornerstones of game design are often deliberately violated. Breaking such rules is, in itself, part of the creative expression of the game designer. For example, *September 12th* [12] cannot be "won" in any legitimate sense of the word. The game places you as a military commander who is able to launch air strikes against terrorists. However, the missiles have a lag before they land, and invariably hit civilians, radicalizing others. Doing nothing results in the terrorists walking freely around the city. Whether the player chooses action, or inaction, the terrorists cannot be removed and so the player cannot reach any satisfying win state. Similarly, other tenets of game design, such as not making games too difficult so as to cause anxiety rather than engagement [13], are violated in order to create unique experiences. *Ninja Gaiden* on the Xbox was deliberately very difficult, so much so that a subsequent release, *Ninja Gaiden Black*, had an easier setting.

By treating game design as an artistic expression, rather than a goal that can either succeed or fail, it becomes clear that it is an impossible task to place any aspects of a game design into a taxonomy that testers can use to evaluate against. This gets at the heart of the difference between the notion of correctness prevalent in most software engineering tasks versus in video games. The vast majority of computer software either performs a service (such as an operating system or a web server), is a tool (such as a word processor or spreadsheet) or is a controller (such as the software inside cars and planes). For such software, it is reasonable to talk about faults in their requirements or design, for if the service, tool, or controller does not provide necessary functionality, it is clearly not as useful, and should be changed. Games are different, as their goal is to evoke a certain emotional reaction, be it entertaining and/or meaningful. In contrast to most software that intends to be productive, the goal of game software is to be consumptive, converting electricity into an experience. Since people find enjoyment or meaning in a wide range of experiences, it is at best challenging, and at worst impossible, to objectively identify game design faults.

This is not to say that there is no overlap between the taxonomy categories with failures that could arise from game design missteps. Let us consider *Deus Ex*, which allowed players to climb up walls by jumping on mines placed on the wall (Category: *Object out of bounds for any state*)[2]. One could conceive that this a design fault, where the game design document specified that all solid objects could be jumped upon. It could be an implementation fault, where a designer accidentally specifies that the mine can be jumped upon. Regardless, we see that the line between design and implementation is blurry, and ideas within the taxonomy could be more applied to game design in the future.

That being said, there are game design failures that do not yet fit anywhere in the taxonomy, such as an asymmetrical map providing benefit to one team over another, dominant strategies, as well as our previous unwinnable game or difficulty examples. Such examples would need to be included should one wish to explore a game design issue taxonomy.

6. UTILITY

We envisage that this taxonomy will find the greatest utility in enabling testers to construct a more systematic approach to testing.

Vijayaraghavan & Kaner used a taxonomy to help structure the thoughts of testers for an e-commerce site [16]. They provided headings under which to think about problems, such as asking what possible failures could occur in regards to internationalization. They found this improved the number of relevant, focused failure cases produced.

This approach will also work well in the video game domain, but games are more complex than a common e-commerce site. We advocate utilizing a "divide-and-conquer" methodology, and applying the taxonomy to individual objects rather than using the taxonomy with the whole game at once. We could apply the taxonomy to the player character and generate a set of test cases, and then categorize failures related to specific NPCs and generate a different set of failure cases, and so on. This would be repeated for all objects of interest in the game world.

Let us return to the Jesus Shot, and imagine the failures that could have led up to this one, as well as all the possible descriptions of the Jesus Shot itself. What could those failures be? Using our methodology, for the ball we generated: ball can land in water without being placed out of play (Category: *Object out of bounds for any state*); ball is

[2] `http://www.zenetproject.com/video:deusex_mine.mp4`

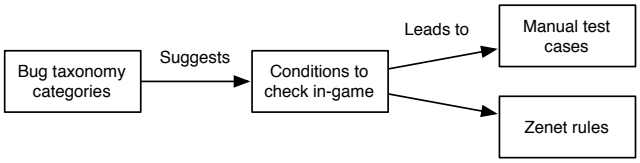

Figure 3: The possible uses of the failure taxonomy.

playable from water (Category: *Action when not allowed*); ball is not repositioned on grass after it lands in water (Category: *Action not possible*); ball can roll into water without being placed out of play (Category: *Invalid position over time*); ball is marked as out-of-play, but never returns to play (Category: *Invalid context state over time*); ball being placed on grass after hitting the water can be cancelled (Category: *Interrupted event*). For the Tiger Woods model, we generated: Tiger can stand on water (Category: *Object out of bounds for any state*); Tiger can swing when on water (Category: *Action when not allowed*).

We do not claim this list details all the possibilities, but thinking about individual objects seems to allow for much greater coverage: the Jesus Shot, in its description, implies that the failure is related to the player model being able to stand on water. However, our failure list indicates that it's more likely that the failure originates from some fault associated with the ball.

Up to this point, we have focused on using the taxonomy with today's approach to bug fixing: employing human testers, sometimes in the hundreds, to explore the game and try and expose failures [14]. However, we anticipate the taxonomy can serve other purposes in the future. We intend to use the taxonomy to generate failure conditions to be inserted into our tool, "Zenet," that can repair failures at runtime [9]. Figure 3 illustrates the similarities between the workflow required for human testers, and that for generating Zenet rules. The taxonomy suggests conditions that should be checked in-game. We can then either use human testers, or encode those conditions in Zenet which runs a rule engine to evaluate whether failure conditions are met. Simple non-temporal failures, such as *Object out of bounds at all times*, are easily expressed in the rule engine. Our rule engine is also designed to handle temporal failures, and can check *Invalid position/context state/event occurrence over time* failures by setting timers. The taxonomy not only helps us generate conditions to encode in Zenet, but also validates the tool's coverage, an important part of any future work into the automated detection or repair of video game failures.

7. CONCLUSION

This paper presents a taxonomy of video game failures, also available online in the form of a living document that can be improved by the collective experience of the gaming community. The taxonomy is validated with numerous examples, illustrating the wide breadth of failures in modern games. Not only will the taxonomy improve the effectiveness of pre-release testing, but it creates the vital groundwork to allow validation of exciting new research in video game failure reduction. Initiatives to investigate how to detect and repair bugs in games can begin, and Zenet is presented as one possible method.

We hope that the taxonomy will encourage others to take up the challenge of reducing game failures, ultimately leading to new technologies that ensure gamers around the world have positive, interesting and exciting experiences.

8. ACKNOWLEDGMENTS

The authors would like to thank Damián Isla for his input on the taxonomy, as well as the members of the Game Network mailing list for their input, including José P. Zagal, Andrew Armstrong, Jennifer Whitson, Louise Peterson and Annika Waern.

9. REFERENCES

[1] Common vunerabilities and exposures. `http://cve.mitre.org/`.

[2] Video game failure archive. `http://www.zenetproject.com/taxonomy`.

[3] Avižienis, A., Laprie, J. C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (January 2004), 11–33.

[4] Bainbridge, W. A., and Bainbridge, W. S. Creative uses of software errors: Glitches and cheats. *Social Science Computer Review 25*, 1 (February 2007), 61–77.

[5] Beizer, B. *Software Testing Techniques*, 2 ed. Van Nostrand Reinhold, 1990, pp. 33–34.

[6] Hind, C., and Bell, D. Setting the bar: Establishing bug criteria to save time, money, and sanity. January 2007.

[7] IGN. Fact or Fiction: The 10 Biggest Rumors on GameCube, August 2003. `http://cube.ign.com/articles/432/432558p3.html`.

[8] Killough, L. Doom level history. `http://www.rome.ro/lee_killough/history/doomqna.shtml`.

[9] Lewis, C. Zenet: Generating and enforcing real-time temporal invariants. In *Proceedings of International Conference of Software Engineering (ICSE 2010)* (May 2010). To be published.

[10] Lough, D. L. *A taxonomy of computer attacks with applications to wireless networks*. PhD thesis, Virginia Polytechnic Institute and State University, 2001.

[11] McElroy, J. See Tiger Woods actually make the Jesus Shot, August 2008. `http://bit.ly/joystiq-jesusshot`.

[12] Newsgaming. September 12th [online]. `http://www.newsgaming.com/games/index12.htm`.

[13] Salen, K., and Zimmerman, E. *Rules of Play*. MIT Press, 2004, pp. 350–353.

[14] Starr, K. Testing video games can't possibly be harder than an afternoon with Xbox, right? *Seattle Weekly* (July 2007).

[15] Tschang, F. T. Videogames as interactive experiential products and their manner of development. *International Journal of Innovation Management 9*, 01 (2005), 103–131.

[16] Vijayaraghavan, G., and Kaner, C. Bug taxnonomies: Use them to generate better tests. In *STAR EAST* (May 2003).