

Annex A - Video Game Bad Smell Catalog

This Appendix describes the catalog of video game bad smells. Our catalog features on 28 video game bad smells grouped into five categories.

Several templates are available in the literature to document bad smells. In this paper, we use the following template:

- **Bad smell Name:** We describe in the title the full name to identify the bad smell.
- **Description of the smell:** Provides a general overview of the bad smell.
- **Consequences of the bad smell:** Reports the main consequences related to the bad smell.
- **Possible correction:** Presents solution(s) that could be applied to remove the bad smell.
- **Example of the bad smell:** When possible we added examples of the bad smells. In other cases, we did not add examples, especially where, the bad smell seems evident, or cannot be well illustrated with only one example.

1 Design and Game Logic

This category includes decision choices related to the overall game architecture (*e.g.*, what is on the clients, what 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).

1.1 Search by String/ID

- **Description of the smell:** This bad smell occurs when an object is identified (or searched) by its string identifier or tag. This occurs for example when determining whether a collision occurred with a specific object.
- **Consequences of the bad smell:** As consequences of this bad smell performance issues may be manifested. Indeed, searching objects by tags/strings (*e.g.*, in Unity through GetComponent), searching objects by their ID, or repeatedly comparing objects' tags may cause performance degradation.

- **Example of the bad smell:** A really simple example of this smell is discussed on Game Development site of StackExchange¹ where developers clearly 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*”
- **Possible correction:** 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`*” since this solution is robust against any change on the searched object and it also works in case of object setup is defined at run-time.

1.2 Creating components/objects at run-time

- **Description of the smell:** 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, where in some scenarios several objects (*e.g.*, bullets being fired) may need to be created in fractions of a second without degrading the performance and the user experience.
- **Consequences of the bad smell:** Repeatedly creating/destroying game objects at run-time may cause performance issues.
Example of the bad smell: Developers discuss on GameDev² 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*”.
- **Possible correction:** The typical solution of this smell is the use of an object pool from which pre-created objects are taken and then released. For example, developers discuss on GameDev³ 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*”.

1.3 Lack of separation of concerns

- **Description of the smell:** In principle, this bad smell can occur in any software application, not only video games. However, this is one of the cases for which we decided to retain the smell in our catalog, because,

¹<https://gamedev.stackexchange.com/questions/142546/in-unity-how-to-get-reference-of-descendant/142550/#142550>

²<https://gamedev.stackexchange.com/questions/101784/what-is-a-reasonable-way-to-avoid-gc-issues-in-unity>

³<https://gamedev.stackexchange.com/questions/101784/what-is-a-reasonable-way-to-avoid-gc-issues-in-unity>

also given the way game engines are conceived, some developers may be tempted to produce code exhibiting such a smell.

- **Consequences of the bad smell:** This bad smell results in having scripts and classes that are incohesive, mixing up controller handling, physics, animation, and rendering.
- **Example of the bad smell:** A typical example is represented by cases in which source code handling controller inputs is mixed with code producing object animations. This smell is discussed on Game Development⁴ 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.
- **Possible correction:** As a possible correction for this design smell, we recommend to simplify the code and reduce his 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”.

1.4 Prefer static classes instead of singletons for single instance entities in the game

- **Description of the smell:** Singletons, differently from static classes, can be derived from existing classes. Also, their usage is transparent to clients (as opposed to static classes). At the same time, when designing a singleton, care should be taken to avoid making it non-thread-safe.
- **Consequences of the bad smell:** Static classes could have no visible and controlled initialization time leading to dependency injection. Singleton could increase the code complexity.
- **Example of the bad smell:** Developers in the game forums also have different views on the usage of singletons. For example, a participant in the Java gaming forum⁵ mentioned that both singletons and static global variables are anti-patterns and this person recommended usage of dependency injection instead. On the other hand, one developer mentioned the importance of using singletons in the specific context of unity engines⁶.
- **Possible correction:** 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

⁴<https://gamedev.stackexchange.com/questions/172991/entity-component-system-implementation-choices/173601/#173601>

⁵<https://jvm-gaming.org/t/design-question/35647/4>

⁶<https://gamedev.stackexchange.com/questions/182053/is-it-bad-form-to-use-singletons-in-unity>

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 with the singleton one can leverage all features of object-oriented programming. With static classes, one cannot control when the constructor is called and no parameters can be passed⁷.

1.5 Dependencies between objects

- **Description of the smell:** This bad smell occurs when references to game objects of game object components are (unnecessarily) repeatedly retrieved.
- **Consequences of the bad smell:** The excessive presence of *dependencies between objects* creates, as in any other software systems, maintainability problems and even increased fault-proneness.
- **Example of the bad smell:** Developers on Game Development of StackExchange⁸ state “... *bad practises about gameObject/transform/components getters are connected with using them heavily, multiple times on every update*”
- **Possible correction:** In game development, an alternative would be to dynamically couple components instead. A possible solution would also be to fix the performance problem through “*caching mechanism*”.

1.6 Poor design of object state management

- **Description of the smell:** This bad smell concerns how a game object state is stored handled, *e.g.*, using simple state variables and conditioned code, or a state-strategy design pattern. For example, complex state management might suggest using a state design pattern instead of an if structure.
- **Consequences of the bad smell:** When the state of a game object is handled by an external component different from the game object itself, this creates unnecessary coupling.
- **Example of the bad smell:** For instance, on JVM Gaming forum⁹ developers face this problem. They debate on what is the best way to implement a state game manager, whether is better to use design patterns, *e.g.*, State or Strategy patterns [1], against if/switch statements. A developer

⁷<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors>

⁸[#74568](https://gamedev.stackexchange.com/questions/74566/using-this-gameobject/74568)

⁹<https://jvm-gaming.org/t/using-a-switch-statement-to-determine-state-of-game/56771>

advises to not “... *introduce design patterns and then try to force your application into them, but design your solution straightforward*”. This is due to the fact that design patterns “*really hardly make things easier*”, thus they have to be used only when needed, as pointed out by previous literature [2].

- **Possible correction:** For this smell, there is no proper fixing action. In general, a good solution can involve the use of design patterns wherever possible, eg State or Strategy patterns. However, if the introduction of these patterns will complicate the code a lot or they can not be introduced easily, a possible solution is to model the game state management through if / switch statements.

1.7 Static coupling

- **Description of the smell:** This smell 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.
- **Consequences of the bad smell:** This smell may cause maintainability problems in the source code as dependencies are not visible there.
- **Example of the bad smell:** 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 contained in a script attached to a game object has a public or *[SerializedField]* 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. On the one hand, this could be considered as a bad practice as 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 forum¹⁰).
- **Possible correction:** The alternative solution to static coupling is to create dependencies in the code through components’ names (*e.g.*, using *GetComponent*-like APIs)..

1.8 Bloated asset

- **Description of the smell:** This smell refers 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).
- **Consequences of the bad smell:** This may result in assets that are unnecessarily big especially when they contain assets that are rarely used.

¹⁰<https://forum.unity.com/threads/int-wont-decrease-to-zero.802566/#post-5331>

- **Example of the bad smell:** For example, developers discuss on Game Development of StackExchange¹¹ about “*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*”.
- **Possible correction:** A simple solution for this kind of smell is to remove unneeded assets to save space and keep the code clean.

1.9 Weak temporization strategy

- **Description of the smell:** This smell concerns wrong assumptions made on the time elapsed between subsequent game object updates.
- **Consequences of the bad smell:** This may result in weak temporization in the game (*e.g.*, physics simulation depends on the frame rate and a lack of synchronization between the updates of multiple game objects).
- **Example of the bad smell:** Developers discuss on Unreal Engine forum¹² 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. Running the game on several machines with different frame rates, while inputs are not scaled by delta time, leads to all input controls dependent on the frame rates.
- **Possible correction:** A common solutions imply the use of time-based updates (*FixedUpdate* in Unity) or making the movement/animation proportional to the time between two frames (*e.g.*, multiplying it by the `Time.deltaTime` in Unity).

2 Multiplayer

This category includes poor decision choices related to the design and implementation of a game’s multiplayer component.

2.1 Storing the game status on clients

- **Description of the smell:** This bad smell occurs in situations where a client can modify the status of another player’s object, making possible

¹¹<https://gamedev.stackexchange.com/questions/97712/do-i-have-to-commit-the-downloadable-assets-for-unity-to-the-repo-or-a-referenc/97726\#97726>

¹²<https://forums.unrealengine.com/t/different-physics-behavior-on-server-and-client-when-tested-on-weak-pc/101125/5>

attacks during games (*e.g.*, preventing the competitor to play/win the game).

- **Consequences of the bad smell:** Occurrences of this bad smell can cause out-of-sync of the gameplay among different players. Furthermore, the status could be potentially hacked by a player for cheating an online competition.
- **Example of the bad smell:** For example, when a new client connects, the game status has to be retrieved from another already connected client, then this state has to be sent to the new ones. Of course, this could cause synchronization problems, especially in case of unexpected disconnections from clients. In fact, a developer discusses this problem in the Game Development site of Stack Exchange¹³. They propose also a solution stating: *“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.”*
- **Possible correction:** The proposed solution for this bad smell is to serialize the data creating *“dots a function by which you save your data to a binary file”*.

2.2 Inefficient data transfer between a client and the host/server

- **Description of the smell:** This bad smell is manifested in the context of a multiplayer game, where there is an unnecessary transfer of data *e.g.*, either because more data than needed is transferred, or because data transfer is performed when not required.
- **Consequences of the bad smell:** This bad smell results in verbose transfer of data between different peers.
- **Example of the bad smell:** For example, in the Game Development site of Stack Exchange¹⁴ developers discuss to use sockets in client-server based game, one developer mentions that sending full information for every client repeatedly is a bad practice. For example, the WebSockets protocol does not have enough speed for a real-time 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”*.
- **Possible correction:** A possible correction to this type of bad smell could be to limit the transfer only to primitive and essential data.

¹³<https://gamedev.stackexchange.com/questions/108827/is-it-bad-practice-to-for-the-server-to-request-data-for-a-client-from-another-c>

¹⁴<https://gamedev.stackexchange.com/questions/124246/input-and-output-of-a-server-side-game-using-web-sockets/126556>

2.3 Reload entire game space when a new client connects

- **Description of the smell:** Every time a new client connects to a multiplayer game, the entire scene is reloaded. When a new player connects in an online game (with all game objects this player is concerned with: they can be a single object (*i.e.*, a humanoid or a vehicle, but they could also be multiple ones), this should not cause a complete reload of the game space for all players, but just a differential update.
- **Consequences of the bad smell:** Occurrences of this bad smell may result in lags and performance problems to all players.
- **Example of the bad smell:** For instance, developers discussed how to exchange game information between Clients and Server in Game Development site of Stack Exchange forum when addressing Box2d simulation problem¹⁵. They state that it is not needed to “... *to send updates for all objects in the game, to all clients.*”
- **Possible correction:** A possible correction for this bad smell would be to select only necessary objects that a client can see and sending their updates to it.

2.4 Synchronization of game objects featuring multiple textures

- **Description of the smell:** This smell is related to syncing multiple textures in a multiplayer environment.
- **Consequences of the bad smell:** Drawing and syncing many canvas/dynamic textures over clients/servers of multiplayer games causes lags or even out-of-sync glitches.
- **Example of the bad smell:** For instance, on HTML5 GameDev forum¹⁶ developers solve the problem of FPS dropping down due to draw canvas for dynamic texture, choosing to update the canvas “*just twice per second instead on every frame*”.
- **Possible correction:** The possible solution is to simplify the number of canvas and textures to draw and synchronize.

3 Animation

The category of animation-related smells is related to how game objects are animated.

¹⁵<https://gamedev.stackexchange.com/questions/62096/how-do-i-duplicate-a-box2d-simulation-mid-simulation>

¹⁶<https://www.html5gamedevs.com/topic/15564-help-with-dynamic-textures/\#comment-88321>

3.1 Improper use of rigging

- **Description of the smell:** There is an improper use of rigging in a composite object/model.
- **Consequences of the bad smell:** How game objects are connected together to form a complex, animated object (*e.g.*, by connecting the bones of a character) can result in problems in the animation.
- **Example of the bad smell:** For instance, we found an interesting discussion¹⁷ on Game Development site of Stack Exchange. Developers point out the importance of using rigging for animations. They highlight that “*it’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*”. In fact, 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*”.
- **Possible correction:** A simple solution for this smell is to introduce/modify the way in which an object is moved into animation. A good practice can be to attach the object to the skeleton of the mesh, *i.e.*, using rigging, in this way the animator will handle all the complexity needed to move a composed object.

3.2 Multiple animators over a model component

- **Description of the smell:** A game object uses (unnecessarily) multiple animator components (or in general components responsible to handle the animation of the game object, depending on the technology being used).
- **Consequences of the bad smell:** Managing multiple animators would be painful.
- **Example of the bad smell:** Developers on Game Development site of Stack Exchange¹⁸ face the possibility to use multiple animators to separately animate the head and the rest of the body. They assert that, from a performance perspective, there is no impact, if you use multiple animator components but they also advise to use this only whether it is needed.
- **Possible correction:** A recommended correction is to use a single animator, even with a complex state machine governing the animation, which would make things much clearer.

¹⁷<https://gamedev.stackexchange.com/questions/104096/how-do-i-get-the-position-and-rotation-of-an-animated-object-in-unity/105908/#105908>

¹⁸<https://gamedev.stackexchange.com/questions/129110/parent-and-child-with-different-animators>

3.3 Moving player vs moving the rest of the scene in an endless-world like game

- **Description of the smell:** The choice between moving the player game object vs. moving the rest of the scene has not been made properly.
- **Consequences of the bad smell:** To design moving a game object instead of the entire scene can lead to glitch in the animation or physic problems.
- **Example of the bad smell:** Developers discuss this smell on Unreal Engine forum¹⁹ and the problem is how to simulate the movement of a space ship while it is moving at a high speed. They suggest to move the entire scene around the ship and not the ship itself to avoid “wacky” physics and glitch due to high speed.
- **Possible correction:** In some games, it may be convenient to move the scene (*e.g.*, endless run) and not the player, or vice versa.

3.4 Continuously checking if position/rotation is within the boundary

- **Description of the smell:** When animating/moving an object, the game continuously checks whether the object is within a boundary (other mechanisms such as range clamping or colliders could be used instead).
- **Consequences of the bad smell:** This smell can cause performance problems, *e.g.*, a continuous check could lead to game lag.
- **Example of the bad smell:** Developers face this problem into a discussion²⁰ of Game Development site of Stack Exchange. They want to improve the performance of collisions detection. A proposed solution is the spatial hashing, *i.e.*, a technique aiming at spatially-dividing the scene in more than one part, and compute a list for objects for each part. When an object crosses the border between two parts, the collision detection algorithm will check whether there will be collisions against only the objects contained into the corresponding list, considerably reducing the checks to be performed.
- **Possible correction:** A compromise should be pursued (too much checking results in lagging, whereas not enough would compromise the behavior). A proposed solution to this bad smell is the spatial hashing, *i.e.*, a technique aiming at spatially divide the scene in more than one part and compute a list for objects for each part. When an object crosses

¹⁹<https://forums.unrealengine.com/t/walking-inside-a-space-ship-while-it-is-moving-at-high-speeds/42183>

²⁰<https://gamedev.stackexchange.com/questions/74858/how-can-i-improve-my-collision-detections-performance>

the border between two parts, the collision detection algorithm will check whether there will be collisions against only the objects contained into the corresponding list, considerably reducing the amount of checks.

3.5 Using objects over particle systems when not needed

- **Description of the smell:** The bad smell occurs when a game uses objects to create effects (*e.g.*, the presence of flies or butterflies in the sky) using game objects is unnecessary and heavyweight unless the latter need to interact with the game. Instead, particle systems—animated sprites projected to the cameras—are more recommended instead.
- **Consequences of the bad smell:** The use of game objects is heavyweight from a performance perspective, unless there is need to interact with the game.
- **Example of the bad smell:** For instance, on Unreal Engine forum we found an interesting discussion²¹ 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.
- **Possible correction:** Particle systems—animated sprites projected to the cameras—are more recommended instead.

3.6 Too many keyframes in animation

- **Description of the smell:** This bad smell occurs when an animation is designed manually, this can be done 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.
- **Consequences of the bad smell:** An animation contains too many keyframes, and this causes performance issues.
- **Example of the bad smell:** On HTML5 GameDevs forum²², there is an interesting example of this smell. Developers highlight that performance issues can be due to a high number of key frames into animation and

²¹<https://forums.unrealengine.com/development-discussion/content-creation/10301-efficient-rendering-of-physical-bullets-no-hit-scanning>

²²<https://www.html5gamedevs.com/topic/22796-problem-with-animation-in-blender-with-new-exporter/#comment-129973>

suggest “to remove most keyframes and just leave a few interpolations” to improve performance.

- **Possible correction:** A recommended correction is to remove keyframes when they are not mandatory.

4 Physics

This category includes smells related to situations where objects’ velocity is directly modified unnecessarily, instead of operating through Forces and Physics.

4.1 Improper mesh settings for a collider

- **Description of the smell:** This bad smell occurs when there is a sub-optimal choice of collider for an object (*e.g.*, too expensive mesh collider, or too-coarse collider instead).
- **Consequences of the bad smell:** A mesh-based collider could be perfectly aligned to a game object’s shape, though it can be the cause of performance degradation. At the same time, a coarser collider (*e.g.*, a box around a character or car) may be more performant, but can cause game glitches, *i.e.*, unwanted collisions.
- **Example of the bad smell:** Developers discuss on Game Development of Stack Exchange²³ about the bad practice of detecting collisions per pixels leading to expensive physic computation. The right way to handle collision is to use polygon shapes.
- **Possible correction:** A simple solution for this smell is to introduce/modify the way in which an object is moved into animation. A good practice can be to attach the object to the skeleton of the mesh, *i.e.*, using rigging, in this way the animator will handle all the complexity needed to move a composed object.

4.2 Heavy physics computation in update

- **Description of the smell:** The bad smell occurs where a game performs heavyweight physics computation at every update. That is, either at every frame (or with a fixed frequency, for the fixed updates) the game physics (*i.e.*, the forces acting on each object) is recomputed and then applied to produce animations.
- **Consequences of the bad smell:** This bad smell could lead to a performance drop, and the solution can be to update the velocity only when the

²³<https://gamedev.stackexchange.com/questions/17222/xna-platformer-collision-perpixel-vs-rectangle>

value of external velocity changes, introducing, for instance, an if statement checking this change.

- **Example of the bad smell:** On Game Development site of Stack Exchange²⁴ developers face the problem “*to set the velocity of an object every loop*”.
- **Possible correction:** 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’s physics only when necessary, or in general by optimizing such computations.

4.3 Setting object velocity and override forces

- **Description of the smell:** When this is not necessary, objects’ velocity is directly modified, instead of operating through Forces/Physics. This 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.
- **Consequences of the bad smell:** This bad smell may cause performance issues.
- **Example of the bad smell:** Regarding this smell we found two interesting discussions²⁵²⁶ on Game Development site of Stack Exchange: they mainly highlight that directly set the velocity on a rigidbody is a bad practice since it “*cancel 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 rigidbodies getting launched into orbit or getting pushed through solid walls*”.
- **Possible correction:** A possible solution is to avoid directly setting the object’s velocity, and apply any additional force instead.

5 Rendering

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

²⁴<https://gamedev.stackexchange.com/questions/51356/is-it-bad-practice-to-set-the-velocity-of-an-object-every-loop?r=SearchResults>

²⁵<https://gamedev.stackexchange.com/questions/153419/proper-way-to-set-a-rigidbody-maximum-velocity>

²⁶<https://gamedev.stackexchange.com/questions/186505/what-are-the-dangers-of-setting-rigidbody-velocity-directly-for-movement-and-wha>

5.1 Object drawing/rendering not properly optimized

- **Description of the smell:** This smell is related to a 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).
- **Consequences of the bad smell:** Occurrences of this bad smell may result in performance issues.
- **Example of the bad smell:** For example, on Unreal Engine forum ²⁷ 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.*”.
- **Possible correction:** 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*”.

5.2 Excessive number of elements in the scene

- **Description of the smell:** This bad smell occurs when we are using a high number of objects on the scene leading to rendering several objects simultaneously.
- **Consequences of the bad smell:** Using an excessive number of elements leads to performance issues, (*e.g.*, use an excessive number of meshes instead of combined ones).
- **Example of the bad smell:** A really simple example of this smell is discussed on Unreal Engine forum²⁸: developers underline that performance issues are related to 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 “*less meshes to process*”.
- **Possible correction:** The way this smell can be corrected 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.

²⁷<https://forums.unrealengine.com/t/ue4-applying-invisible-materials-to-some-bodyparts-improve-fps/149799>

²⁸<https://forums.unrealengine.com/t/beginner-performance-issues/96340>

5.3 Texture or polygon choices cause aliasing

- **Description of the smell:** This smell is related to a bad choice of object texturing or on the number or type of polygons composing game object causes aliasing.
- **Consequences of the bad smell:** The smell mostly results in visualization problems than in performance problems.
- **Example of the bad smell:** An example is discussed on Game Development site of StackExchange²⁹: developers ask a possible solution to “*no smoothing applied to the textures*”. Jagged and black lines are displayed around a figure.
- **Possible correction:** Presents solution(s) that could be applied to remove the bad smells.

5.4 Sampling and rendering to the same texture

- **Description of the smell:** This bad smell occurs in situations where we are writing to a texture while it is being rendered instead of creating a texture copy.
- **Consequences of the bad smell:** This bad smell may cause performance issues and undefined behavior on GPUs.
- **Example of the bad smell:** On Game Development forum of Stack Exchange, developers state “*...reading and writing to the same texture in a shader is bad practice (not surprisingly) and will cause undefined behavior on GPUs*”.
- **Possible correction:** A recommended practice is, instead, to create a texture copy, modifying, and then rendering it again. *e.g.*, “*...copy the target texture to a new texture and then read from the copy while still writing to the original target texture*”.

5.5 Sub-optimal, expensive choice of lights, shadows, or reflection

- **Description of the smell:** This bad smell occurs when some lights that can be baked are, instead, rendered in real-time, or when there is excessive usage of (unnecessary) shadows and reflections.
- **Consequences of the bad smell:** A sub-optimal choice of lights, reflections, or shadows may cause performance problems.

²⁹<https://gamedev.stackexchange.com/questions/110286/how-to-fix-texture-edge-artefacts>

- **Example of the bad smell:** An interesting example is discussed on Unreal Engine forum³⁰: developers ask if exist a way to “*[d]ynamically light up a big hall without too much performance lost*”. More precisely, when player comes into a room the light is turned on with a variable intensity and the attenuation radius causes performance issue.
- **Possible correction:** Recommended solutions to remove this bad smell are several, *e.g.*, try to use less light, make some lights static or try to split big mesh into smaller ones.

5.6 Issues in material rendering

- **Description of the smell:** This bad smell is manifested when a material is adopting wrong mesh types, or, for example, the presence of side-by-side materials could cause visualization issues.
- **Consequences of the bad smell:** The presence of side-by-side materials could cause visualization issues.
- **Example of the bad smell:** A discussion³¹ on Unreal Engine forum faces if to use 2 sided material against a non 2 sided affects performance. Developers point out that of course “*... it costs more performance to render a material 2 sided than not.*” but they also highlight “*everything in game design is about context so without context the answer is yes/no/maybe*”.
- **Possible correction:** A possible correction to this bad smell is avoid rendering in a precise way or not depends on the game context, *e.g.*, the game is either desktop project, Mobile, or VR.

References

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
- [2] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*, pages 77–84. IEEE Computer Society, 2001.

³⁰<https://forums.unrealengine.com/t/dynamically-light-up-a-big-hall-without-too-much-performance-lost/78721>

³¹<https://forums.unrealengine.com/t/is-there-any-hit-on-performance-with-2-sided-materials-vs-non-2-sided/76736>