Temporal Super Resolution Frequently Asked Questions

Questions and answers to common questions related to Temporal Super Resolution.

This page contains frequently asked questions that relate to **Temporal Super Resolution** (TSR). Each section of this page corresponds to one in the TSR technical overview. For more detailed information and examples, see Temporal Super Resolution

TSR Scalability

TSR includes a lot of customization for its upscaling settings so you can tailor it to individual platforms based on your project's needs. For more information about TSR's scalability options and its function, see the "TSR Scalability" section of Temporal Super Resolution.

- **Why is TSR's runtime GPU cost in Stat GPU slower than expected?**

  TSR is running on the GPU close to the end of the frame in order to clean up aliasing before upscaling to display resolution. It is the last component of the renderer to allocate persistent GPU resources for its history used to accumulate data over frames. In scenarios where the GPU runs out of video memory, it can mean that TSR is often the first component to hit performance pitfalls of GPU resources needing to be allocated on host GPU memory.

  If you're encountering out-of-memory issues with your GPU, you can try adjusting the timeout detection and recovery (TDR) event that triggers a GPU crash to happen. See Dealing with GPU Crashes for more information.

- **What range does TSR support for Screen Percentage?**

  TSR shaders are designed to cope with a screen percentage in ranges between 25% up to 200% using the console command `r.TSR.History.ScreenPercentage` and have it work with `r.ScreenPercentage'. For instance, it would work like this:
  - `r.TSR.History.ScreenPercentage=200` could be fed `r.ScreenPercentage=50` to `r.ScreenPercentage=200`.
  - `r.TSR.History.ScreenPercentage=100` could be fed `r.ScreenPercentage=25` to `r.ScreenPercentage=100`.

  You'll want to pay close attention to the history of TSR feed and TSR 1spp to be mindful of how much information TSR has to work with each frame.
- **Does TSR support dynamic rendering resolution? What platforms are supported?**

  Yes, dynamic resolution is supported and enabled on console platforms. However, Windows and other desktop platforms are not. This isn't specific to TSR. This is an ongoing area of research and is currently being tested on Fortnite. These are some of the current problems:
  - The purpose of dynamic resolution is to max out the resolution, and not to underuse the GPU so that we have the highest TSR history convergence rate. This is unlike consoles where there are more programs than just the game running on the GPU, and each of those are running at their own frame rate. It's currently impossible with existing APIs to know how much GPU headroom should be left for other programs in order to complete frames at the desired frame rate.
  - There is a missing API on the D3D's swap chain to allow us to tear a little bit at the top of the screen with VSync enabled, which is common practice on consoles (this is specified with `rhi.PresentThreshold.Top`). It's important not to miss any frames when dynamic resolution ends up finishing a few microseconds late. This happens more often on desktop platforms, like Windows, more often than on consoles in our testing when the GPU can be preempted for the use of third party programs.
  - The GPU scales its clock for power saving reasons dynamically. This has a tendency to lower dynamic resolution as it tries to conservatively undershoot GPU costs. The problem here is we are missing official D3D APIs to query the GPU scaling. It's only using a drive backdoor (see Microsoft's article on D3DKMTQueryStatistics function) that we have been able to identify this problem shown in Fortnite when using `Stat TSR` with the stats for **GPU Clock** and **DynRes**.
- **Does TSR interpolate or generate intermediate frames?**

  It does not. TSR is Unreal Engine's built-in temporal upscaler / super resolution technology. Temporal upscaling / super resolution is different than tech that generates intermediate frames. Because there is a lot of interest in both these types of tech, information surrounding them can be confusing if it's not made clear.
- **Can TSR work with third-party frame interpolation / generation?**

  Yes it can. It works in the same way as a renderer does. This kind of frame interpolation tech has to handle many more features in the post processing chain after the temporal upscaler, such as motion blur, chromatic aberration, sharpening, film grain, bloom, lens dirt, local exposure, and post process materials.

The frame interpolation / generation plugin free to access depth and motion vector because of ISceneViewExtensions without needing to set up a ITemporalUpscaler to replace TSR. For instance, TV manufacturers do frame generation on any video feed, whether it's a broadcast channel, streaming service, physical media, or a game console.

- **What consequences does frame interpolation / generation have on TSR?**

  Frame interpolation / generation adds GPU cost, like many rendering technologies do. Increasing GPU cost can have an effect on frame rate when the GPU is bound while TSR or any other temporal upscaler is accumulating details.

  For instance, let's consider a frame interpolation tech where the frame is multiplied and doubles the number of frames being presented, taking a game's FPS from 55 Hz to an interpolated FPS of 88 Hz. The new frame rate of the game can be computed with frame interpolation enabled by having the interpolated game FPS is equal to

  For instance, let's consider a frame interpolation tech where FrameMultiplier=2 doubles the number of frames being presented, going from a GameFPS=55 Hz to InterpolatedFPS=88 Hz. The new game frame rate can be computed with frame interpolation enabled by having InterpolatedGameFPS=FPSInterpolatedFPS/FrameMultiplier=44 Hz. This is a drop of the game's framerate by a multiplication factor of InterpolatedGameFPS/GameFPS=0.8. This directly affects the MP feed to the temporal upscaler by *0.8 and increases history convergence rate by a factor of 1/0.8=1.25.

  You can check these results with `Stat TSR` that shows **TSR Feed** and **TSR** convergence rate. You can see the effect of enabling frame interpolation on the accumulation speed of details.

- **Are there plans to improve Temporal Anti-Aliasing (TAA) or Temporal Anti-Aliasing Upscaling (TAAU)?**

  No, TAA and its variant TAAU were developed with the limitations of last generation's consoles. Since their development, the only thing added was TSR's ability to have a R11G11B10 without serious color precision problems in the history. This saves more memory bandwidth performance on older platforms and is controlled with the command `r.TemporalAA.R11G11B10History`, which is enabled by default. For your own projects, you could still expose TAA on D3D11 or D3D12 for those users with lower end GPUs where TSR at low anti-aliasing scalability is still too costly. Fortnite is set up this way with these players in mind.

- **Is it possible to compare TSR with other third-party temporal upscalers?**

  While TSR is the default method in Unreal Engine 5, third-party temporal upscalers are readily available for your projects. A limitation of 5.3 with the ITemporalUpscaler interface that could cause a crash when switching between third party temporal upscalers has been addressed. You can use the **Show > Visualize** menu in the viewport to toggle on the **Temporal Upscaler** visualization. This visualizer works with any temporal upscaler, whether it's built into Unreal Engine or a third party one. This visualizer is a helpful way of comparing between built-in temporal upscalers and third party ones using the same rendering and display resolution.

- **Does TSR have a sharpening pass?**

No, it does not. We are only interested in matching references on the image produced by TSR. However, the tonemapper pass that runs after temporal upscaling has optional sharpening that can be enabled with `r.Tonemapper.Shapen`. You can verify its use with the **Temporal Upscaler** visualization found in the **Show > Visualize** menu.

> Fortnite sets `r.Tonemapper.Sharpen=0.5` for all temporal upscalers on all platforms due to the competitive nature of the game.

TSR History

TSR accumulates details over time. The aggregated integration of these rendered details over time is done in the history at display resolution. For more information about TSR's history and its function, see the "TSR History" section of Temporal Super Resolution.

- **Does TSR's History Sample Count affect the memory footprint of the history?**

  TSR's History isn't affected when setting a higher sample count with `r.TSR.History.SampleCount`. The pixels' details aren't kept raw in TSR, they are accumulated/aggregated in the history for performance reasons. The history sample count solely has an incidence of the minimum contribution of the current frame in the history. This is 1 / History Sample Count. The lower the sample count, the higher the minimum contribution of the current frame is in the history, which can cause higher temporal instability. Higher sample counts have a lower minimum contribution of the current frame making the image more stable. However, if there is ghosting, it will take time to disappear.

  The History Sample Count is per TSR output/display pixel, meaning the Nyquist-Shannon history uses a lower history sample count per history pixel, which is what is displayed when using the Temporal Super Resolution visualization mode (`show VisualizeTSR`).

- **How should the history weight be tuned for movement?**

  Setting the weight of the history while movement is happening is a tradeoff between image stability and sharpness. For example, games of a competitive nature, such as Fortnite, can benefit by sacrificing some image stability during movement in favor of sharpness by lowering `r.TSR.Velocity.WeightClampingSampleCount`. This console variable should be tuned with `r.TSR.History.ScreenPercentage` set to 100 since a higher history resolution automatically decreases both blurring and stability in movement.

- **Will running TSR on a 4K monitor on Epic anti-aliasing scalability render an 8K texture?**

  Yes, it will. Running TSR on a 4K monitor with Epic anti-aliasing scalability will effectively use an 8K texture. You may want to consider using High anti-aliasing scalability instead if your game doesn't need to have sharp visuals while in motion. Something to consider is if you're running on a recent powerful GPU but still on a 1080p monitor, detail loss due to history reprojection will be much more noticeable. For this reason, it would benefit your project to expose TSR anti-aliasing scalability for low, medium, high,

and epic quality levels is an important choice for players to tune the experience that best fits their hardware setup.

- **Should my project use a TSR History with a Screen Percentage greater than 200?**

  It's not necessary since twice as many pixels on each axis is enough to hide the interpolation between history pixels in display pixels. Using a screen percentage higher than 200 would also run into upscaling factors between the rendering resolution and history resolution that are far beyond what we test. The TSR History already scales the screen percentage arbitrarily between 100 and 200, and it's worth underlining that there is a shader permutation of the downsampling pass specifically for history screen percentage equal to 200 since that reaches Nyquist-Shannon's necessary condition to quick in and have some math that conveniently simplifies the downsampling. If needed, the TSR history screen percentage can be adjusted with `r.TSR.History.ScreenPercentage`.

- **Should my TSR History Screen Percentage match the primary Screen Percentage if I want it to be greater than 100%?**

  Yes, the TSR History Screen Percentage (`r.TSR.History.ScreenPerecentage`) should match the primary Screen Percentage (`r.ScreenPercentage`) if you want it to be greater than 100%, otherwise your super sampled rendering will suffer history rejection blur. If your project's frame budget can afford a higher primary screen percentage, it can afford a matching TSR history screen percentage.

- **Is TSR capable of rendering 8K?**

  TSR has been developed and designed to make the history update as fast as possible. For consoles rendering at 4K, this means having as much room in the budget at rendering resolution in parallax and shading heuristics for image quality. On PC, this means a budget for the Nyquist-Shannon history at 4K on high-end GPUs or lower display resolutions on decently fast GPUs. While we are not actively testing 8K, there shouldn't be a problem with higher anti-aliasing quality or setting lower TSR History Screen Percentage (`r.TSR.History.ScreenPerecentage`) to avoid 16K history.

  The engine has experimental 8K support to run part of the post processing chain at one-eighth of the resolution when using `r.PostProcessing.QuarterResolutionDownsample 2`. This can be combined with `r.Bloom.ScreenPercentage 12.5` for FFT to also run at this resolution. With these settings, TSR is capable of producing scene color in the history update at one-eighth size when `r.TSR.History.ScreenPercentage` is equal to 100 (or when using Epic anti-aliasing scalability)

- **What are the risks of the Nyquist-Shannon history running out of video memory to cause a crash?**

  If you have the video memory available, there shouldn't be an issue. However, if not, the drive starts allocating CPU memory, which can cause performance to drop. TSR checks GMaxTextureDimensions to avoid allocating a history larger than what the hardware can feasibly encode to its pixel coordinates (which is often 16384 pixels). TSR also computes all its texture UV coordinates with 32-bit floats to have the precision of 23-bit mantissa, even on 16-bit VALU shader permutations.

- **Does TSR's Nyquist-Shannon history have a performance impact on the post processing chain?**

  There is no impact to the post processing chain that follows the Nyquist-Shannon history because the downsample to display resolution happens within TSR. So, the passes after it see no difference in

resolution between Epic or High anti-aliasing scalability settings.

- **The Nyquist-Shannon history can be expensive to use. Should we provide other options for players?**

  Unreal Engine's scalability groups for low, medium, high, epic, and cinematic are used to scale quality, and Epic scalability is the highest quality it can offer to players. If you were to look at Steam's hardware survey, you could get an idea of what users' most common display resolutions are; not everyone has a 4K monitor. With this in mind, you might want to consider other options that offer more versatility with quality and performance trade offs of the Nyquist-Shannon history, such that they could use a lower anti-aliasing scalability offered with TSR. For instance, if a user has a 1080p monitor, the screen percentage could be 540p due to history reprojection blur on their 1080p monitor.

- **Consider exposing separate TSR settings for TSR's History Screen Percentage.**

  You can optionally add your own anti-aliasing scalability groups and expose them as separate settings. One way to differentiate TSR's settings from other anti-aliasing options is to add a parenthesis indicating the method being used. And with your own project's interface, you could differentiate this further. In Fortnite for example, multiple anti-aliasing methods are supported. The TSR settings have their own selections depending on low, medium, high, and epic quality settings. There are also some additional parameters exposed to the user interface, such as TSR's History Screen Percentage (`r.TSR.History.ScreenPercentage`).

  If you add TSR History Screen Percentage, consider how you name it. Something like "TSR Super Sampling" is effective but could be confused with Screen Percentage or other third-party temporal upscalers that have super sampling.

Shading Rejection

TSR's shading rejection heuristic is the process of deciding how much the current frame matches a frame that was previously rendered and whether it should be reused or rejected. For more information about TSR's shading rejection and its function, see the "Shading Rejection" section of Temporal Super Resolution.

- **TSR Ghosting when BlendFinal and ClampBlend are less than 1.**

  TSR development attempts to resolve this as much as possible but this is a known issue with TSR shading rejection.

- **What is the interest in history rejection using ClampBlend?**

  Temporal Anti-Aliasing in Unreal Engine 4 is mostly reliant on the technique of rejection using ClampBlend. It's useful for making sure the history colors aren't going too far off the input resolution, while preserving some anti-aliased edges. However, its downsides are that it causes ghosting when there is lots of noise, so it can't be used solely for history rejection to minimize ghosting. Also, because it blurs texture details, it can only be used when needed.

- **What is the interest in history rejection using BlendFinal?**

BlendFinal allows for dynamic control of the weight of the current frame compared to the history. A higher blend final means the current frame should dominate more of the output, up to the point where the history is no longer used. If BlendFinal is equal to 1, it disables the history reprojection to offset the cost of the spatial anti-aliaser. This particular method of using BlendFinal is useful to reject history on areas of the screen that are too noisy, and that would make the clamp have ghosting artifacts (like what can be seen with Temporal Anti-Aliasing). It has the downside of showing up in the rendering resolution noise, too.

TSR and Translucency

Translucency is a particular issue for TSR to handle because there can be any number of arbitrary layers blended on top of one another. Especially since they never draw velocities. For more information about how TSR works with translucent materials, see the "TSR and Translucency" section of Temporal Super Resolution.

- **Does TSR need translucent materials to use the Responsive AA setting?**

  TSR does not need translucent materials to use the Responsive AA setting. This setting is useful for Temporal Anti-Aliasing (TAA). This prevented TAA from losing at much detail with VFX that use translucent materials. TSR addresses this for translucent materials by setting the **Translucency Pass** to be **After DOF** by default.

- **Should After DOF translucency be disabled to draw directly to Scene Color to save performance?**

  You can use `r.SeparateTranslucency 0` to draw directly to Scene Color to save performance by removing the cost of the compositing pass. However, we recommend not to do this. TSR's GPU cost isn't any faster with it disabled. In fact, TSR does remove the need for a separate compositing pass in the process, and the only performance you could be saving is drawing translucencies to a float R11G11B10 (if configured with `r.SceneColorFormat`) instead of a float R16G16B16A16 pixel format. But, this has a tendency to have quantization problems when blending in very low frequency transparent effects, such as smoke clouds.

  There is a known issue where the BaseScalabililty.ini with low effects quality would set `r.SeparateTranslucency` to 0, making TSR ghost on translucencies in the scene. To get around this, you can use `r.TSR.ForceSeparateTranslucency` to force enable the separate translucency pass whenever using TSR.

- **Does translucency happening after depth of field add memory pressure?**

  There is no added pressure to memory for translucency that happens after depth of field because of how the engine reuses intermediary resource memory multiple times in a frame. It does this with Render Dependency Graph and the RHI with transient memory allocators. This can be set with `r.RDG.TransientAllocator 1`, which is enabled by default.

- **Is there a risk of ghosting if my After DOF translucency gets drawn Before DOF behind the focus distance?**

It is unlikely to happen since the depth of field's blur applied to your translucent material reduces noise that makes TSR less precise in the shading rejection. Instead, any noise could be the result of Nanite details.

TSR with Post Process Materials

Post-process materials afford TSR some amount of flexibility in rendering materials, but they come with their own limitations since TSR happens in the middle of the post-processing rendering chain. Materials are inserted in different locations in the scene color and after the depth of field translucency. For more information about how TSR handles post process materials, see the "TSR with Post Process Materials" section of Temporal Super Resolution.

- **What's the best way to handle cel-shading outlines with TSR?**

  When outlines are not working well — largely due to lack of motion vectors for these outlines — consider making them inline on the geometry it contours so that the geometry can drag the line around the object's motion vectors. The line should have at least the thickness of a pixel to keep the outline from looking discontinued.

  On dynamic objects, like a character that is running, the outline being camera view dependent means it has a tendency to move on the character's surface without feeding that movement to a motion vector. Use the post process material setting **Blendable Location** with **Translucency After DOF** to take advantage of special logic made for translucent materials to have them not ghost when no motion vector is provided.

- **How can I access an anti-aliasing depth buffer after TSR?**

  This may be impossible. The primary problem is the back buffer can't be anti-aliased. A way to think about this and why it is that it can't be anti-aliased is with a street light that is at 10 meters from the background that is 30 meters away. Anti-aliasing the depth between the two means you could end up with depth anywhere between 10 and 30 meters. The street light and the building are not connected. For this reason, we don't implement a depth temporal accumulation, and there are no plans to implement such a thing due to jittering caused by instability and aliasing of the depth buffer. It would be impossible to use this result concretely after TSR without introducing aliasing and instability back into the image. With this in mind, we recommend any depth-based effects happen before TSR or TAA.

Flickering Temporal Analysis

TSR's flickering temporal analysis analyzes the image for artifacts, such as Moire patterns and attempts to keep geometric details consistent and visible, even at far distances. It also attempts to stabilize the image as much as possible. For more information about how TSR's flickering temporal analysis works, see the "Flickering Temporal Analysis" section of Temporal Super Resolution.

- **Should flickering always be mitigated by TSR?**

There are currently two known sources of flickering that can be mitigated outside of TSR:

- When there is an object(s) in the scene with high intensity that causes Lumen Global Illumination to lose stability.
- Distant objects can cause flickering with Nanite simplification where geometry edges are not welded.

- **Does the material settings "Has Pixel Animation" force draw object velocity?**

  The material setting **Has Pixel Animation** forces this material to disable the render's heuristic that assumes animation is fully described with motion vectors. Due to the lack of available bits in the GBuffer, and for this feature to be available with Forward rendering, this setting is encoded into the velocity buffer that TSR already reads for reprojection. So, even if your opaque geometry with this material setting enable is static, it still needs to draw its static velocity just to encode this setting.

- **Does the "Has Pixel Animation" setting add material shader permutations on material instances?**

  The **Has Pixel Animation** material setting does not add any additional shader permutations to material instances using it. To avoid compiling more shader permutations, this setting is set automatically on primitives that have materials assigned that use it. However, this means that any primitive that has at least one material with the flag enabled in one of its material slots will have TSR's flickering temporal analysis disabled on all of its materials for this primitive. We recommend to avoid having primitives that have geometric complexity whereby they'll have multiple material slots but only part of the mesh uses material animation.

History Resurrection

TSR's history resurrection is the process of deciding whether to use the immediate previous frame or an earlier "resurrected" frame from the history that better matches details of the current frame. This process is an effort to stabilize the image to limit, or even prevent, noise and ghosting artifacts from happening. For more information on how frames are resurrected from the history, see the "History Resurrection" section of Temporal Super Resolution.

- **Why are some frames only kept persistently?**

  We only keep some frames persistently in order to save video memory. This is especially important with a Nyquist-Shannon history where TSR history screen percentage (`r.TSR.History.ScreenPercentage`) is equal to 200 on Epic and Cinematic anti-aliasing scalability settings.

- **Does TSR's history resurrection have a larger memory footprint?**

  Without history resurrection, TSR needs two histories: the current frame and the previous frame, whereby they flip roles every frame. When the resurrection is enabled (`r.TSR.Resurrection 1`), the number of histories becomes 2 + TSR resurrections persistent frame count (`r.TSR.Resurrection.PersistentFrameCount`). Because the persistent frame count is 2 frames, the history resurrection doubles the memory footprint of TSR by default. The memory footprint of the history can be visualized in the Temporal Upscaler visualization mode. You can toggle this visualization mode from the viewport under the **Show > Visualize** menu by selecting **Temporal Upscaler**.

- **Why only resurrect the oldest persistent frame?**

  We only resurrect the oldest persistent frame because of the base GPU cost of the resurrection since it still needs to reproject a rendering resolution version of the resurrected frame to compare with the previous and current frames. It must do this each frame, even no resurrection ends up being useful.

- **Why does TSR's resurrection persistent frame interval count have some constraints?**

  In order to optimize the history resurrection as much as possible without additional texture fetch instructions in the Update History pass happening at display resolution or higher, TSR makes use of Texture2DArrays. This makes it so that when a single shader needs to both read from and write to, there is some constraint imposed on the range of the texture slice with which the shader can read from at the same time.

- **Does the GPU perform additional copy of the history to keep persistent frames around?**

  No, it does not.

- **Why does the resurrection not happen everywhere I would expect it to?**

  The history resurrection happens under two conditions:
  - The resurrection frame is a closer match to the current frame than the previous frame is.
  - The previous frame doesn't match enough of the current frame to be considered.

  Under these conditions, the resurrected frame is then compared to the current frame. If it matches enough, it is then used. In scenarios where there is a lot of indirect lighting, the current frame might not have enough samples from Lumen, making the current frame look different to what it should actually be, or to what the resurrected frame looks like. This can prevent the frame resurrection from happening.

- **How complex would it be to merge TSR's latest changes to earlier versions of Unreal Engine?**

  Starting with Unreal Engine 5.4, TSR history is made of a Texture2DArray instead of Texture2D. To avoid extra GPU copy outputting Texture2D for the next post processing chain, some changes in the post processing passes have been made to accept SRV on the RDG texture instead with a new `FScreenPassTextureSlice` in the PostProcessing.cpp file. A lot of these changes are already in Unreal Engine 5.3, so you can cherry pick changes from 5.4 and later that are self-contained to the usual TSR files.

  For version of Unreal Engine prior to 5.3, the 5.3 changes in the post processing chain to allow the history resurrection can be found here:
  - https://github.com/EpicGames/UnrealEngine/commit/d03d2caf4e4e7b874ca0f672100ca35eeede5123
  - https://github.com/EpicGames/UnrealEngine/commit/0f284290394673adb307ba49aa5701f1dc5889be
  - https://github.com/EpicGames/UnrealEngine/commit/72ed81e4a0dff2f3d66ab062e1d1308f60f1706f
  - https://github.com/EpicGames/UnrealEngine/commit/25e8efb4a6f8775c2e73edd84d02d0ea6d1da3de
  - https://github.com/EpicGames/UnrealEngine/commit/6d3f7b6544feaec39ce88059bd31808fbd9da2cc

  Unreal Engine 5.4 required an additional change in the post processing chain, which can be found here:
  - https://github.com/EpicGames/UnrealEngine/commit/59a7373ea7bde36706a2f8601ff6d72979624672

- **Is the history resurrection able to be shipped on a project developed for console platforms?**

There have been enough optimizations implemented in TSR as of Unreal Engine 5.4 to compensate for the base cost of the resurrection to have the same GPU runtime cost to that of Unreal Engine 5.1. This was when TSR shipped on 4K consoles with Fortnite at 60 Hz. In tests we performed with performance replays locked at 60% dynamic resolution scale, we measured a 0.24 millisecond average cost overall, whereas Fortnite was running at an average of 51% screen percentage with dynamic resolution.

- **Can a moving object be resurrected?**

  Resurrection does its best to reduce how often details need to be accumulated from scratch again. In instances where an object is no longer on screen, like when it is occluded by something within the frame or it has moved outside of the frame, TSR will lose track of how it moves around and have to accumulate its details again. The lack of optical flow in TSR means it only reprojects using the current frame's depth buffer using the resurrected frame's view and projection matrices to see if their look lines up with the current frame.

  If your object hasn't moved, or has moved very little, TSR will resurrect those details except for when the colors don't line up. For example in Fortnite, TSR is able to resurrect the slow moving grass, even if it doesn't perfectly line up. This is sufficient to capture fresh details of the newly seen grass as they are accumulated. For details that move faster, it's harder to notice the lack of resurrection as they move on screen compared to static objects in the scene where it's easier to spot these dropped details.