TimeWarp

v. 0.4

1. Overview

To create a true sense of presence in a virtual reality experience, a so-called "time warp" may be used. This time warp corrects for the optical aberration of the lenses used in a virtual reality headset. It also transforms stereoscopic images based on the latest head tracking information to significantly reduce the motion-to-photon delay. Stereoscopic eye views are rendered to textures. These textures are then warped onto the display to correct for the distortion caused by wide angle lenses in the headset.

To reduce the motion-to-photon delay, updated orientation information is retrieved for the headset just before drawing the time warp, and a transformation matrix is calculated that warps eye textures from where they were at the time they were rendered to where they should be at the time they are displayed. Many people are skeptical on first hearing about this, but for attitude changes, the warped pixels are almost exactly correct. A sharp rotation will leave some pixels black at the edges, but this turns out to be minimally distracting.

The time warp is taken a step farther by making it an "interpolated time warp." Because the video is scanned out at a rate of about 120 scan lines a millisecond, scan lines farther to the right have a greater latency than lines to the left. On a sluggish LCD this doesn't really matter, but on a crisp switching OLED it makes it feel like the world is subtly stretching or shearing when you turn quickly. This is corrected by predicting the head attitude at the beginning of each eye, a prediction of < 8 milliseconds, and the end of each eye, < 16 milliseconds. These predictions are used to calculate time warp transformations, and the warp is interpolated between these two values for each scan line drawn.

The time warp can be implemented on the GPU by rendering a full screen quad with a fragment program that calculates warped texture coordinates to sample the eye textures. However, for improved performance the time warp renders a uniformly tessellated grid of triangles over the whole screen where the texture coordinates are setup to sample the eye textures. Rendering a grid of triangles with warped texture coordinates basically results in a piecewise linear approximation of the time warp.

If the time warp runs asynchronously to the stereoscopic rendering, then the time warp can also be used to increase the perceived frame rate and to smooth out inconsistent frame rates. By default, the time warp currently runs asynchronously for both native and Unity applications.

2. TimeWarp Minimum Vsyncs

The TimeWarp `MinimumVsyncs` parameter default value is 1 for a 60 FPS target. Setting it to 2 will cause WarpSwap to hold the application frame rate to no more than 30 FPS. The asynchronous TimeWarp thread will continue to render new frames with updated head tracking at 60 FPS, but the application will only have an opportunity to generate 30 new stereo pairs of eye buffers per second. You can set higher values for experimental purposes, but the only sane values for shipping apps are 1 and 2.

You can experiment with these values in a Native app by pressing right-trigger plus dpad-right in VrScene.apk to cycle from 1 to 4 MinimumVsyncs. For Unity apps, please refer to the Unity 30Hz TimeWarp SDK Example.

There are two cases where you might consider explicitly setting this:

If your application can't hold 60 FPS most of the time, it might be better to clamp at 30 FPS all the time, rather than have the app smoothness or behavior change unpredictably for the user. In most cases, we believe that simplifying the experiences to hold 60 FPS is the correct decision, but there may be exceptions.

Rendering at 30 application FPS will save a significant amount of power and reduce the thermal load on the device. Some applications may be able to hit 60 FPS, but run into thermal problems quickly, which can have catastrophic performance implications -- it may be necessary to target 30 FPS if you want to be able to play for extended periods of time. See *Power Management* for more information regarding thermal throttle mitigation strategies.

3. Consequences of not rendering at 60 FPS

These apply whether you have explicitly set MinimumVsyncs or your app is just going that slow by itself.

If the viewpoint is far away from all geometry, nothing is animating, and the rate of head rotation is low, there will be no visual difference. When any of these conditions are not present, there will be greater or lesser artifacts to balance.

If the head rotation rate is high, black at the edges of the screen will be visibly pulled in by a variable amount depending on how long it has been since an eye buffer was submitted. This still happens at 60 FPS, but because the total time is small and constant from frame to frame, it is almost impossible to notice. At lower frame rates, you can see it snapping at the edges of the screen.

There are two mitigations for this:

Instead of using either "now" or the time when the frame will start being displayed as the point where the head tracking model is queried, use a time that is at the midpoint of all the frames that the eye buffers will be shown on. This distributes the "unrendered area" on both sides of the screen, rather than piling up on one.

Coupled with that, increasing the field of view used for the eye buffers gives it more cushion off the edges to pull from. For native applications, we currently add 10 degrees to the FOV when the frame rate is below 60. If the resolution of the eye buffers is not increased, this effectively lowers the resolution in the center of the screen. There may be value in scaling the FOV dynamically based on the head rotation rates, but you would still see an initial pop at the edges, and changing the FOV continuously results in more visible edge artifacts when mostly stable.

TimeWarp currently makes no attempt to compensate for changes in position, only attitude. We don't have real position tracking in mobile yet, but we do use a head / neck model that provides some eye movement based on rotation, and games that allow the user to navigate around explicitly move the eye origin. These values will not change at all between eye updates, so at 30 eye FPS, TimeWarp would be smoothly updating attitude each frame, but movement would only change every other frame.

Walking straight ahead with nothing really close by works rather better than might be expected, but sidestepping next to a wall makes it fairly obvious. Even just moving your head when very close to objects makes the effect visible.

There is no magic solution for this. We do not have the performance headroom on mobile to have TimeWarp do a depth buffer informed reprojection, and doing so would create new visual artifacts in any case. There is a simplified approach that we may adopt that treats the entire scene as a single depth, but work on it is not currently scheduled.

It is safe to say that if your application has a significant graphical element nearly stuck to the view, like an FPS weapon, that it is not a candidate for 30 FPS.

Turning your viewpoint with the joypad is among the most nauseating things you can do in VR, but some games still require it. When handled entirely by the app this winds up being like a position change, so a low framerate app would have smooth "rotation" when the user's head was moving, but chunky rotation when they use the joypad. To address this, TimeWarp has an "ExternalVelocity" matrix parameter that can allow joypad yaw to be smoothly extrapolated on every rendered frame. We do not yet have a Unity interface for this.

In-world animation will be noticeably chunkier at lower frame rates, but in-place doesn't wind up being very distracting. Objects on trajectories are more problematic, because they appear to be stuttering back and forth as they move, when you track them with your head.

For many apps, monoscopic rendering may still be a better experience than 30 FPS rendering. The savings is not as large, but it is a clear tradeoff without as many variables.

If you go below 60 FPS, Unity apps may be better off without the multi-threaded renderer, which adds a frame of latency. 30 FPS with GPU pipeline and multi-threaded renderer is getting to be a lot of latency, and while TimeWarp will remove all of it for attitude, position changes including the head model, will feel very lagged.

Note that this is all bleeding edge, and some of this guidance is speculative.

4. TimeWarp Chromatic Aberration Correction

TimeWarp has an option for enabling Chromatic Aberration Correction. On a 1920x1080 Adreno 330 running at full speed, this increases the TimeWarp execution time from 2.0 ms to 3.1 ms per vsync, so it is a large enough performance cost that it is not the default behavior, but applications can enable it as desired.

5. TimeWarp Debug Graph

More detailed information about TimeWarp can be obtained from the debug graph. In native apps, the debug graph can be turned on with right-shoulder + dpad-up. This will cycle between off / running / frozen. In Unity, the plugin call OVR_SetDebugMode(1, 1) will turn the debug graph on, and OVR_SetDebugMode(0, 1) will turn it off.

Each line in the graph represents one eye on the screen. A green line means the eye has a new frame source compared to the last time it was drawn. A red line means it is using the same textures as the previous frame, time warped to the current position. An application rendering a steady 60 FPS will have all green lines. An even distribution of red and green lines means the application is generally slow. Red spikes means an intermittent operation like garbage collection may be causing problems. A pair of tall red lines means an entire frame was skipped. This should not happen unless the OS or graphics driver has some unexpected contention. Unfortunately this does still happen sometimes.

The horizontal white lines represent the approximately 8 milliseconds of time that the previous eye is being scanned out, and the red or green lines represent the start of the time warp operation to the completion of the rendering. If the line is completely inside the white lines, the drawing completed before the target memory was scanned out to video, and everything is good. If the line extends above the white line, a brief tear may be visible on screen.

In a perfect world, all the lines would be short and at the bottom of the graph. If a line starts well above the bottom, timewarp did not get scheduled when it wanted to be. If a line is unusually long, it means that the GPU took a long time to get to a point where it could context switch to the high priority time warp commands. The CPU load and GPU pipeline bubbles have to be balanced against maximum context switch latency.

The Adreno uses a tiling architecture and can switch tasks every so many tiling bins. The time warp is executed as a high performance task but has to wait for the last batch of tiling bins to be complete. If the foreground application is doing rendering that makes individual tiling bins very expensive, it may cause problems here. For the best results, avoid covering parts of the screen with highly tessellated geometry that uses an expensive fragment program.

Last Update: Nov 10, 2014