Dynamic Resolution Rendering

Doug Binks

Introduction

The resolution selection screen has been one of the defining aspects of PC gaming since the birth of 3D games. In this whitepaper and the accompanying sample code, we argue that this no longer needs to be the case; developers can dynamically vary the resolution of their rendering instead of having a static resolution selection.

Dynamic resolution rendering involves adjusting the resolution to which you render the 3D scene by constraining the rendering to a portion of a render target using a viewport, and then scaling this to the output back buffer. Graphical user interface components can then be rendered at the back buffer resolution, as these are typically less expensive elements to draw. The end result is that stable high frame rates can be achieved with high quality GUIs.

We'll be presenting performance results and screenshots in this article taken on a pre-release mobile 2nd generation Intel® Core™ i7 processor (Intel® microarchitecture code name Sandy Bridge, D1 stepping quad core 2.4 GHz CPU with 4GB DDR3 1333MHz RAM) with Intel® HD Graphics 3000.

This article and the accompanying sample were originally presented at the Game Developers Conference (GDC) in San Francisco 2011, and a video of the presentation can be found on GDC Vault [GDC Vault 2011], with the slides for that presentation available on the Intel website [Intel GDC 2011]. Since the presentation, the author has discovered that several game companies already use this technique on consoles; Dmitry Andreev from LucasArts' presentation on Anti-Aliasing is the only public source, though with few details on the dynamic resolution technique used [Andreev 2011].

Figure 1: The sample scene viewed from one of the static camera viewpoints.

Motivation

Games have almost always had a strong performance variation with resolution, and the increase in shader complexity along with post-processing techniques has continued the trend of per-pixel costs dominating modern games. Increasing resolution also increases texture sampling and render target bandwidth. Setting the resolution appropriately for the performance of the system is therefore critical. Being able to vary the resolution dynamically gives the developer an additional performance control option which can enable the game to maintain a stable and appropriate frame rate, thus improving the overall quality of the experience.

Rendering the graphical user interface at the native screen resolution can be particularly important for role playing, real time strategy, and massively multiplayer games. Suddenly, even on low-end systems, the player can indulge in complex chat messaging whilst keeping an eye on their teammates' stats.

Finally, with the increasing dominance of laptops in PC gaming, power consumption is beginning to become relevant to game development. Performance settings can cause a reduction in CPU and GPU frequency when a machine goes from mains to battery power, and with dynamic resolution rendering, the game can automatically adjust the resolution to compensate. Some games may want to give the user the option of a low power profile to further reduce power consumption and enable longer gaming on the go. Experiments with the sample have found that cutting the resolution to 0.5x reduces the power consumption of the processor package to 0.7x normal when vertical sync is enabled so that the frame rate is maintained.

Basic Principles

The basic principle of dynamic resolution rendering is to use a viewport to constrain the rendering to a portion of an off-screen render target, and then to scale the view. For example, the render target might be of size (1920, 1080), but the viewport could have an origin of (0, 0) and size (1280, 720).

Figure 2: using a viewport to constrain rendering

By creating render targets larger than the back buffer, the dynamic resolution can be varied from subsampled to supersampled. Care needs to be taken to ensure the full set of required render targets and textures fit within graphics memory, but systems based on Intel® microarchitecture code name Sandy Bridge processor graphics usually have considerable memory, as they use system memory.

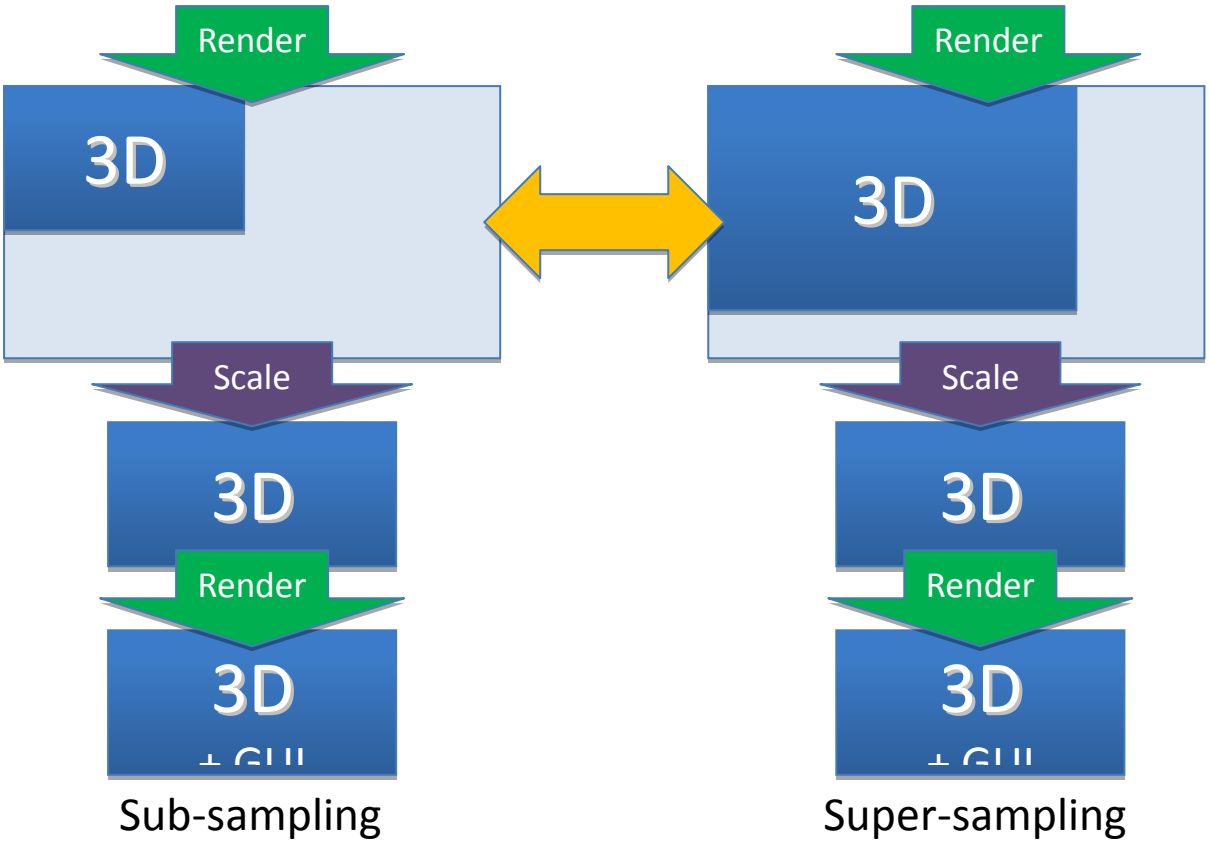

Figure 3: dynamic resolution can be varied from subsampling to supersampling

When undertaking normal rendering to the dynamic viewport, there are no changes that need to be made—the rasterization rules ensure this is handled. However, when reading from the render target, care needs to be taken to scale the coordinates appropriately and handle clamping at the right and bottom edges.

The following example pixel shader code shows how to clamp UVs. This is mainly used when doing dependent reads (i.e., when there are per-pixel operations on a UV, which is subsequently used to sample from a dynamic render target).

```
// Clamp UVs to texture size
// PSSubSampleRTCurrRatio is the fraction of the render target in use.
float2 clampedUV = min( unclampedUV,  g_PSSubSampleRTCurrRatio.xy );
```

In the case of motion blur—a common post-process operation that uses dependent reads from a render target—the extra math required has little effect on the performance, as the shader is texture-fetch bound.

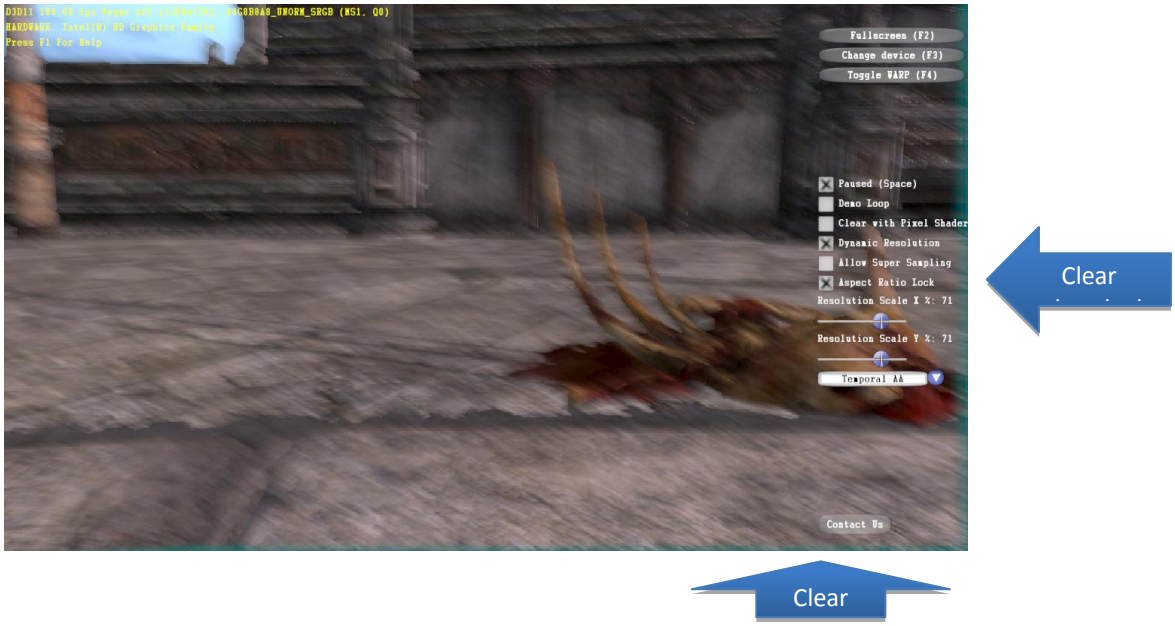

Figure 4: Colour leak on edges of screen due to motion blur, which can be solved by using clamping

In addition to clamping, it's also important to ensure that the resolution ratios used in shaders is representative of the actual viewport ratio, rather than just your application's desired ratio. This is easily obtained by recalculating the ratio from the dynamic viewport dimensions. For example, in the sample code function DynamicResolution::SetScale, the following is performed after ensuring the scale meets boundary criteria:

```
// Now convert to give integer height and width for viewport
m_CurrDynamicRTHeight    = floor( (float)m_DynamicRTHeight * m_ScaleY / m_MaxScalingY );
m_CurrDynamicRTWidth     = floor( (float)m_DynamicRTWidth *  m_ScaleX / m_MaxScalingX );

// Recreate scale values from actual viewport values
m_ScaleY = m_CurrDynamicRTHeight * m_MaxScalingY / (float)m_DynamicRTHeight;
m_ScaleX = m_CurrDynamicRTWidth *  m_MaxScalingX / (float)m_DynamicRTWidth;
```

Scaling Filters

After rendering the 3D scene, the viewport area needs to be scaled to the back buffer resolution. A variety of filters can be used to perform this, and the sample implements several examples as described here.

Point Filtering

Point filtering is a fast basic filter option. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~0.4ms.

Bilinear Filtering

Bilinear filtering is almost as fast as point filtering due to hardware support, and it reduces the aliasing artifacts from edges by smoothing, but also blurs the scene. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~0.4ms.

Bicubic Filtering

Bicubic filtering is only noticeably better than bilinear for resolutions of 0.5x the back buffer, and its performance is 7x slower even using a fast bicubic filter [Sigg 2005]. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~3.5ms.

Noise Filtering

Adding some noise to point filtering helps to add high frequencies, which break the aliasing slightly at a low cost. The implementation in the sample is fairly basic, and improved film grain filtering might artistically fit your rendering. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~0.5ms.

Noise Offset Filtering

Adding a small random offset to the sampling location during scaling reduces the regularity of aliased edges. This approach is common in fast filtering of shadow maps. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~0.7ms.

Temporal Anti-aliasing Filtering

This scaling filter requires extra support during the initial rendering path to render odd and even frames offset by half a pixel in X and Y. When filtered intelligently to remove ghosting artifacts, the resulting image quality is substantially improved by sampling from twice as many pixels. This filtering method is described in greater depth in its own section below. Scaling from a 0.71x ratio dynamic viewport to 1280x720 takes ~1.1ms, and has almost the same quality as rendering to full resolution.

Temporal Anti-aliasing Details

Temporal anti-aliasing has been around for some time; however, ghosting problems due to differences in the positions of objects in consecutive frames have limited its use. Modern rendering techniques are finally making it an attractive option due to its low performance overhead.

The basic approach is to render odd and even frames jittered (offset) by half a pixel in both X and Y. The sample code does this by translating the projection matrix. The final scaling then combines both the current and previous frames, offsetting them by the inverse of the amount they were jittered. The final image is thus made from twice the number of pixels arranged in a pattern similar to the dots of the five side on a die, frequently termed a quincunx pattern.

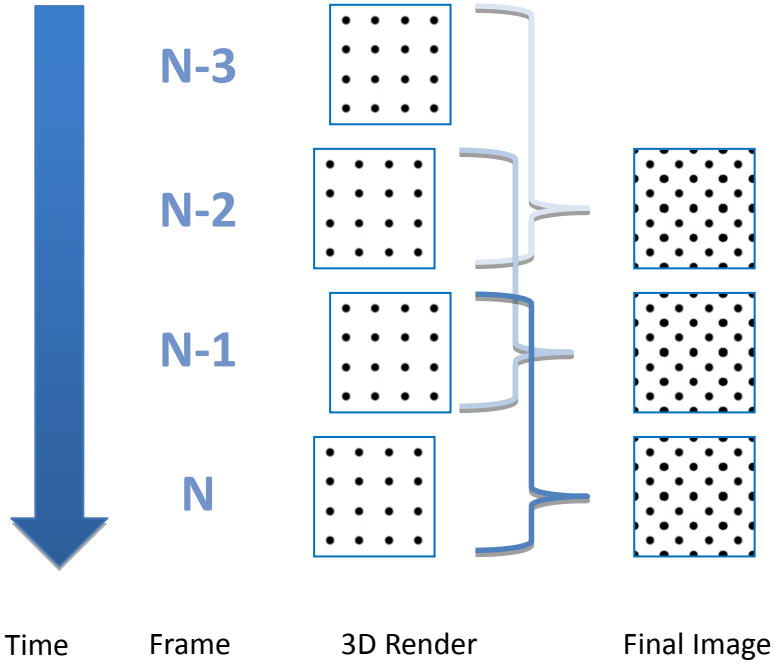

Figure 5: Temporal Anti-Aliasing basic principle

Used along with dynamic resolution, this approach gives an increased observed number of pixels in the scene when the dynamic resolution is lower than the back buffer, improving the detail in the scene. When the dynamic resolution is equal or higher to the back buffer, the result is a form of anti-aliasing.

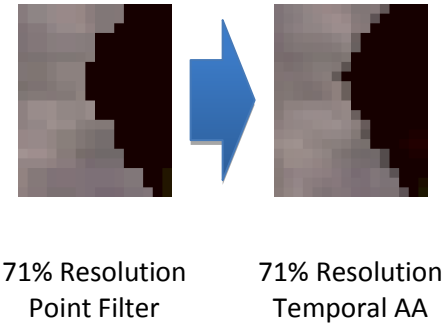

Figure 6: Result of Temporal AA when dynamic resolution is lower than that of the back buffer

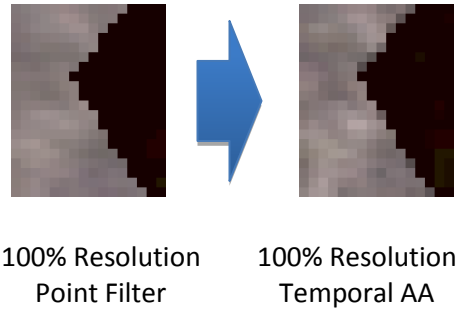

Figure 7: Result of Temporal AA when dynamic resolution is equal or higher to that of the back buffer

In order to get increased texture resolution, a MIP LOD bias needs to be applied to textures. In Microsoft Direct3D* 11, use a D3D11_SAMPLER_DESC MipLODBias of -0.5f during the 3D scene pass. Additionally, the sampler used during scaling needs to use bilinear minification filtering, for example: D3D11_FILTER_MIN_LINEAR_MAG_MIP_POINT.

In order to reduce ghosting, we use the velocity buffer written out for motion blur. Importantly, this buffer contains the velocity for each pixel in screen space, thus accounting for camera movement. A scale factor is calculated from both the current and previous frame's velocity and applied to the previous frame's colour to determine its contribution to the final image. This scales the contribution based on how similar the sample location is in real space in both frames.

$$S = \frac{1}{1 + K \times (\mathbf{V}_n \cdot \mathbf{V}_n + \mathbf{V}_{n-1} \cdot \mathbf{V}_{n-1})}$$

$$\mathbf{C}' = \frac{(\mathbf{C}_n + S \times \mathbf{C}_{n-1})}{(1 + S)}$$

Where $\mathbf{C}'$ is the final color output.
$\mathbf{C}_n$ the current and $\mathbf{C}_{n-1}$ the previous color buffers.
$\mathbf{V}_n$ is the current and $\mathbf{V}_{n-1}$ the previous velocity buffers.
The constant K is typically sized to be $\sim 1/\text{width}$.

The sample has K tuned to give what the author considers to be the best results for a real time application, with no ghosting observed at realistically playable frame rates. Screenshots do expose a small amount of ghosting in high contrast areas as in the screenshot below, which can be tuned out if desired.

For games, transparencies present a particular problem in not always rendering out velocity information. In this case, the alpha channel could be used during the forwards rendering of the transparencies to store a value used to scale the contributions in much the same way as the velocity is currently used.

An alternative to this approach for ghosting removal is to use the screen space velocity to sample from the previous frame at the location where the current pixel was. This is the technique used in the CryENGINE* 3, first demonstrated in the game Crysis* 2 [Crytek 2010]. Intriguingly, LucasArts' Dmitry Andreev considered using temporal anti-aliasing, but did not due to the use of dynamic resolution in their engine [Andreev 2011]. The author believes these are compatible, as demonstrated in the sample code.

Figure 8: Temporal Anti-Aliasing with velocity scaling and moving objects

The effect of motion blur

Motion blur smears pixels and reduces observed aliasing effectively, hence a lower resolution can be used when the camera is moving. However, the sample does not exploit this in its resolution control algorithm. The following screenshots show how reducing the resolution to 0.71x the back buffer results in higher performance, but roughly the same image. Combined with varying motion blur sample rates, this could be a way to reduce artifacts from undersampling with large camera motions whilst maintaining a consistent performance.

Figure 9: Motion blur with dynamic resolution off

Figure 10: Motion blur with dynamic resolution on at 0.71x resolution. Note the decreased frame time yet similar quality end result

Supersampling

Supersampling is a simple technique where the render target used to render the scene is larger than the back buffer. This technique is largely ignored by the current real-time rendering community—multi sampled anti-aliasing and other anti-aliasing techniques have replaced its usage due to their better memory consumption and performance.

Using dynamic resolution significantly reduces the performance impact of adding supersampling, as the actual resolution used can be dynamically adjusted. There is a small performance impact to enabling supersampling, mainly due to the extra cost of clearing the larger buffers. The sample code implements a 2x resolution render target when supersampling is enabled, but good quality results are observed for relatively small increases in resolution over the back buffer resolution, so a smaller render target could be used if memory were at a premium. Memory is less of an issue on processor graphics platforms, as the GPU has access to a relatively large proportion of the system memory, all of which is accessible at full performance.

Once dynamic resolution rendering methods are integrated, using supersampling is trivial. We encourage developers to consider this, since it can be beneficial for smaller screen sizes and future hardware which could have sufficient performance to run the game at more than its maximum quality.

Render Target Clearing

Since dynamic resolution rendering does not always use the entire render targets surface, it can be beneficial to clear only the required portion. The sample implements a pixel shader clear, and on the Intel® HD Graphics 3000-based system tested, the performance of a pixel shader clear was greater than that of a standard clear when the dynamic ratio was less than 0.71x for a 1280x720 back buffer. In many cases, it may not be necessary to clear the render targets, as these get overwritten fully every frame.

Depth buffers should still be cleared completely with the standard clear methods, since these may implement hierarchical depth. Some multi-sampled render targets may also use compression, so should be cleared normally.

Performance Scaling

The sample code scales well with resolution, despite the heavy vertex processing load due to the large highly detailed scene with no level of detail and only very simple culling performed. This gives the chosen control method significant leverage to maintain frame rate at the desired level.

Most games use level-of-detail mechanisms to control the vertex load. If these are linked to the approximate size of the object in pixels, the resulting performance scaling will be greater.

-

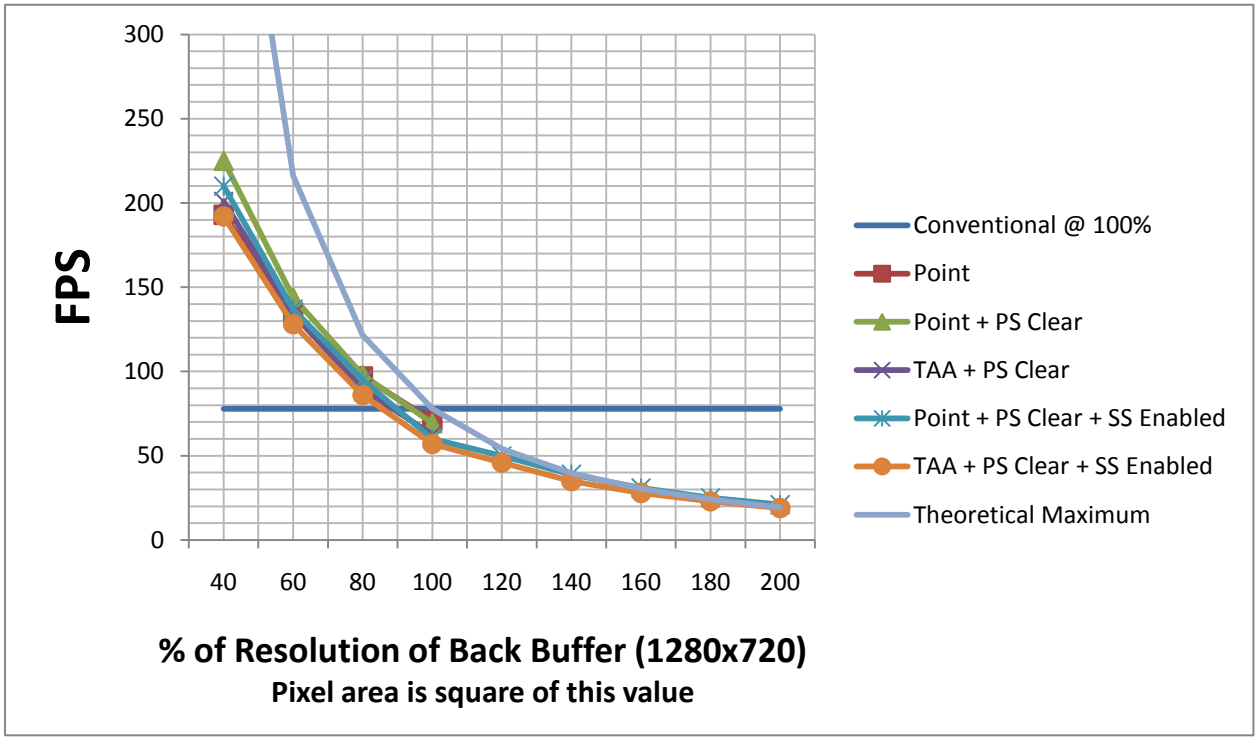

Figure 11: Dynamic Resolution Performance at 1280x720

Resolution Control

The sample implements a resolution control method in addition to allowing manual control. The code is in the file DynamicResolutionRendering.cpp, in the function ControlResolution. The desired

performance can be selected between the refresh rate (usually 60Hz or 60FPS) and half the refresh rate (usually 30FPS).

The control scheme is basic: a resolution scale delta is calculated proportionally to the dimensionless difference in the desired frame time and the current frame time.

$$\Delta s = k \times S \times \frac{(T - t)}{T}$$

$$S' = S + \Delta s$$

Where $S'$is the new resolution scale ratio, S is the current resolution scale ratio, Δs is the scale delta, k a rate of change constant, $T$the desired frame time, and t the current frame time.

The current frame time uses an average of the GPU inner frame time excluding the present calculated using Microsoft DirectX* queries, and the frame time calculated from the interval between frames in the normal way. The GPU inner frame time is required when vertical sync is enabled, as in this situation the frame time is capped to the sync rate, yet we need to know if the actual rendering time is shorter than that. Averaging with the actual frame rate helps to take into account the present along with some CPU frame workloads. If the actual frame time is significantly larger than the GPU inner frame time, this is ignored, as these are usually due to CPU side spikes such as going from windowed to fullscreen.

Potential Improvements

The following list is by no means complete, but merely some of the features which the author believes would naturally extend the current work:

- Combine the dynamic resolution scene rendering with a similar method for shadow maps.
- Use this technique with a separate control mechanism for particle systems, allowing enhanced quality when only a few small particles are being rendered and improved performance when the fill rate increases.
- The technique is compatible with other anti-aliasing techniques that can also be applied along with temporal anti-aliasing.
- Temporal anti-aliasing can use an improved weighted sum dependent on the distance to the pixel center of the current and previous frames, rather than just a summed blend. A velocity-dependent offset read, such as that used in the CryENGINE* 3 [Crytek 2010], could also be used.
- Some games may benefit from running higher quality anti-aliasing techniques over a smaller area of the image, such as for the main character or on RTS units highlighted by the mouse.

Conclusion

Dynamic resolution rendering gives developers the tools needed to improve overall quality with minimal user intervention, especially when combined with temporal anti-aliasing. Given the large

range of performance in the PC GPU market, we encourage developers to use this technique as one of their methods of achieving the desired frame rate for their game.

References

[Sigg 2005] Christian Sigg, Martin Hadwiger, “Fast Third Order Filtering”, GPU Gems 2. Addison-Wesley, 2005.

[Crytek 2010] HPG 2010 “Future graphics in games”, Cevat Yerli & Anton Kaplanyan. http://www.crytek.com/cryengine/presentations

[GDC Vault 2011] http://www.gdcvault.com/play/1014646/-SPONSORED-Dynamic-Resolution-Rendering

[Intel GDC 2011] http://software.intel.com/en-us/articles/intelgdc2011/

[Andreev 2011] http://www.gdcvault.com/play/1014550/Anti-aliasing-from-a-Different [PPT 4.6MB]