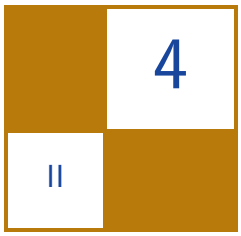


CryENGINE 3: Three Years of Work in Review

Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz

4.1 Introduction

For the game *Crysis 2*, the R&D team at Crytek created the acclaimed CryENGINE 3 (CE3) game engine. This latest engine iteration involved a large engineering team, took approximately three years to complete, and covered a number of technical areas including physics, AI, animation, graphics, and audio. For this chapter, we focus on a few key rendering topics.

Figure 4.1. A scene from *Crysis 2*.

We start with an introduction to the CE3 rendering pipeline and discuss the differences from previous iterations of the engine. We also cover individual topics, including gamma-correct HDR rendering, deferred lighting, shadows, batched HDR postprocessing for efficient multiplatform motion blur and bokeh DoF, stereo rendering, and advanced screen-space rendering methods. See Figure 4.1 for an example of a scene from *Crysis 2* rendered with CE3.

4.2 Going Multiplatform

In developing *Crysis 2* and CryENGINE 3, the team went out of its comfort zone. Virtually every team member had little or no prior console experience, most coming from a PC background.

Additionally, the design team decided early on to move away from Jungle-style environments, which Crytek has been so well known—for almost a decade—for creating. This time the challenge was to create a post-apocalyptic urban environment, an "Urban Jungle." This was a completely new experience for almost everyone on the team.

Our high-level goals were clear from the beginning: no lead platform, simultaneously shipping on all platforms (PS3, Xbox 360, and PC), and maximize quality on all platforms and all system specs, eliminating sloppy lower specs, as was common with *Crysis 1*. In *Crysis 1*, for example, we essentially disabled individual features for lower system specs, resulting in very poor quality results for the less capable systems. This time around, every single feature was re-engineered so that every platform configuration could support it to some degree.

Going multiplatform meant we had to pick our fights carefully and share as much base work as possible across platforms. This required us to generalize whenever feasible and to avoid custom techniques and optimizations for this or that platform—unless one proved to be a significant performance gain or visual improvement. Such an approach allowed us to minimize risks and time spent on quality assurance and debugging—though, of course, this was still a significant effort. This resulted in approximately three and a half years of learning and hard work for everyone at the company.

We essentially began the process by porting existing functionality across all platforms and checking that everything was in place and worked as expected. Very often we were shocked by how slowly certain processes were running. For example, we were surprised to find post processes taking up to 30 ms, particles taking up to 50 ms, and consoles running at 10 fps, with barely anything on the screen! (See Figure 4.2.)

After most of the key functionality was in place, we started a massive re-engineering and optimizing effort where our philosophy was in essence: "Every millisecond counts but so does visual quality."

Figure 4.2. Some of the very first CryENGINE 3 results on consoles PlayStation® 3 (left) and Xbox 360 (right). Notice the 10 fps on the PS3.

4.2.1 Multiplatform GPU High-Level Optimization Guidelines

Many lessons were learned during CryENGINE 3's development regarding GPU-side optimization. Even though these lessons are self-evident having gone through this process, they were not obvious just a couple of years ago. Here are some of the most important ones:

1. Generalize and always optimize for the worst case scenario.
   (a) Discover the biggest bottlenecks and address them by tackling the biggest time consumer. This means avoiding partial optimizations. An example from *Crysis 1* times: if the camera was static, then motion blur was disabled. If the camera was moving fast, then motion blur was enabled. This kind of bad optimization strategy resulted in big performance peaks and an inconsistent frame rate.
   (b) Once done, repeat *ad nauseam*!
2. Don't repeat work or do unnecessary work. For example:
   (a) Don't down-sample full-screen color targets or depth targets multiple times for different postprocessing functions.
   (b) Minimize the number of memory transfers, render target clears, and any redundant full-screen passes.
   (c) Such repeated or redundant work adds up very quickly. For example, a full-screen pass at 720 p costs ca. 0.25 ms on the Xbox 360 and ca. 0.4 ms on PS3. It is very easy to spend many milliseconds in a wasteful manner.
   (d) Batch as much as possible in a single pass.

3. Take advantage of interframe coherency. Amortize costs across frames:
   (a) This can provide a significant gain if done carefully, taking performance peaks and multi-GPU systems into account.
   (b) Distribute costs evenly. For example, if the HUD updates every nth frame, then every $n + 1$-th frame update some similar-costing render technique.
   (c) For screen-space ambient occlusion (SSAO) and the like, the cost can be distributed across frames.
4. In the end, the key words for most cases are: "share, share, share." Share as many computations and as much bandwidth as is reasonably possible in a single pass. We went through several steps, re-engineering several techniques so that certain dependencies would be eliminated and techniques could be batched.

4.2.2 Multiplatform Optimization: Best Practices

Streamline the optimization process; everyone in a company is responsible for performance and memory. Instead of programmers spending weeks or even months of work optimizing, sometimes it is much more efficient and productive for the art or design teams to spend a couple of hours optimizing the art resources to remove waste.

We achieved this kind of streamlining by helping each team keep within its budget by showing clearly where the cost of its work was located. We introduced a

Figure 4.3. Our GPU timers' view (left); lighting overdraw view (right).

few timers that showed where the GPU costs were located: e.g., shadows, lighting, post processes. We also employed other visualization tools such as lighting and scene overdraw visualization (see Figure 4.3).

Budgets are very important early on; they define clear limits for the project. It is important to ensure that the budgets are understood not only by programmers but also by art and design departments. Everyone must understand that budgets are guidelines that can be further adjusted, depending on a level's characteristics, since every level has its own requirements, for example an indoor or night level requires many light sources, while an outdoor daytime level requires mainly sunlight.

Another important step for us was to monitor performance regularly on all platforms, since on a very big team such as Crytek Frankfurt (300 colleagues), it is very easy for performance to run out of control. This meant that every week we made a couple of PIX/GPad captures at exact, repeatable locations as well as at any low-performance locations, and then we investigated the findings. By the end of the project this was done on almost a daily basis.

Last but not least, something new that we introduced at Crytek during *Crysis 2* production was visual regression tests. Whenever a programmer submitted code, an automated build/test was triggered that ran the new code changes at predefined camera locations. Performance was measured and a screenshot was captured. The results were uploaded to our automated test server and could be visualized through a user-friendly web interface. It was not fully automated; for example, no image comparison was done, but this simple measure allowed us to track visual or performance regressions much quicker. This is an area we will certainly continue to evolve for our next projects.

4.3 Physically Based Rendering

For CryENGINE 3 we wanted to take a step forward with more physically based rendering techniques, while still maintaining the minimal precomputation steps involved and also allowing for many dynamic light sources (a limitation of our previous engine).

We decided early on, that a certain degree of backward compatibility with our previous engine was required; hence, we opted for not moving to a fully deferred rendering engine, since current console hardware is still too limited for achieving the required material variety supported in CryENGINE. We used instead a variation of what is now commonly called deferred lighting, which was popularized by Naughty Dog's *Uncharted* series, allowing us to keep our material-variation flexibility, at the cost of an additional opaque rendering pass.

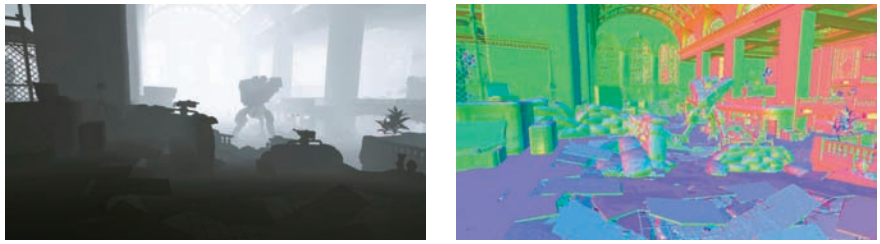
We use a minimal G-buffer setup, composed of depth/stencil (stencil used for tagging indoor areas), world-space best fit normals [Kaplanyan 10a], and material glossiness stored in the alpha channel. Additionally, we required an HDR

Target ID	R	G	B	A	Format (PC/Xbox 360/PS3)
DS	Depth + Stencil				D24S8
RT0	Normals + Glossiness				A8B8G8R8
RT0	Diffuse Accumulation				FP16, FP10, RGBK
RT1	Specular Accumulation				FP16, FP10, RGBK
RT0	Scene Target				FP16, FP10, RGBK

Table 4.1. G-buffer and render targets set up across different platforms (PC, Xbox 360, and PS3).

lighting buffer and an HDR scene target. Besides these, there are several other intermediate targets, used for many different purposes, including shadows, sprites, postprocessing, and other cases. This was one of the cases where generalization was not possible due to hardware limitations (see Table 4.1).

A typical frame starts with our G-buffer, rendering opaque geometry, then composing any alpha-blended layers on top of it (these can be decals, or deferred decals and terrain layers). Storing normals in world space helped minimize trouble with alpha blending since most of today's hardware still lacks programmable blending. This is still far from ideal since more complex normal encoding could help improve quality or storage costs but would not blend well on most hardware (see Figure 4.4).



Depth Normals

Figure 4.4. G-buffer visualization.

G-buffer generation is followed by lighting-accumulation stages. At this point, ambient lighting, environment lighting probes, SSDO or SSAO, GI, real-time local reflections, and all light sources are applied (see Figure 4.5).

Shading is then done in forward passes. Lighting buffers are fetched and when required, albedo and specular maps are applied along with a per-material Fresnel term. Different shading models, (such as deferred skin shading, hair rendering with anisotropic highlights, cloth) may be used for each material.

Figure 4.5. Diffuse lighting accumulation (left) and specular lighting accumulation (right).

After the opaque shading passes, alpha-blended geometry is processed. This can be any alpha-blended pass, like glass, deferred fog passes, water surfaces, or particles. A frame ends by performing final tone mapping, postprocessed antialiasing, and a couple of game-dependent postprocessing steps such as the game 3D HUD or similar (see Figure 4.6).

Figure 4.6. Shading passes (linear space; left) and final tonemapped and postprocessed image (right).

4.3.1 The Importance of HDR

High dynamic range (HDR) [Reinhard et al. 10] is important for two main reasons: precision and range. HDR rendering allows for more accurate lighting ranges and minimizes banding and color clipping. The human eye cells are sensitive to a wide range of light wavelengths and our brain is trained from birth to recognize objects through color. Thus, humans are particularly sensitive to any kind of incorrectness in images, any kind of color clipping, improper range, or tone reproduction. The naïve observer may not know why an image doesn't look or feel

Figure 4.7. LDR (left; notice color clipping) and HDR (right; notice the much wider range tone reproduction).

real, but will be unconsciously aware of the many and common contributors for unnatural-looking images.

Such underrated features are the keys to opening the door to physically based rendering, and consequently into achieving more realistic images by more accurately simulating how the human eye or cameras behave. HDR imaging also allows for physically based postprocessing, including depth of field and motion blur, camera CMOS grain simulation, the common bloom, flares, streaks, and other effects (See Figure 4.7).

Additionally, one large benefit for a high dynamic range rendering pipeline is the improved art workflow, since the artists don't have to spend as much time working around the limitations of low dynamic range (for example, not being able to use dark textures or darker/denser fog, and certain decals). Of course, hardware limitations still apply, particularly on consoles where range and precision are still limited compared to PCs, so a bit of care and common sense should be exercised.

4.3.2 The Importance of Being Linear

It might surprise many to learn that the lighting and rendering in video games have been incorrectly implemented for a long time. In fact, entire art teams have been tweaking their materials and light setups incorrectly. Lighting correctly in linear space means that all rendering computations should happen in linear space: lighting in linear space, linear blending operations, and ensuring gamma correctness at the end of frame output.

Movie industry professionals have been aware that such mathematical correctness is also one of the main contributors to the realism of the final image. Larry

Figure 4.8. Incorrect linear rendering (left) and correct linear rendering (right).

Gritz (Pixar/Nvidia/Sony Pictures Imageworks) popularized this extremely important topic in video games [Gritz and d'Eon 08] (see Figure 4.8).

The main source of linear incorrectness is the input albedo and specular maps, which most DCC tools store in sRGB space due to precision benefits. Luckily this gamma curve is now supported natively by most hardware, with some differences across certain platforms. For instance, Xbox 360 has a custom piecewise linear (PWL) sRGB curve, hence additional care must be taken to ensure linear correctness across a regular sRGB curve to PWL [Vlachos 08]. This conversion could be done in shader code, instead of relying on hardware functionality, but due to the lack of programmable texture filtering, this would give nonlinear filtered results. Another multiplatform limitation is that DX9-level hardware and the PS3 do not support gamma-correct alpha blending so alpha blending targets must be stored in linear space for correct results.

Since we are rendering using HDR, we can store our render targets in linear space without worrying about the loss of precision. Thus, all we have to worry about is using sRGB texture reads and then all the resulting shading and blending are implicitly linearly correct.

After tone mapping, a final gamma 2.2 curve is applied in the tone-mapping shader. This helps eliminate any hardware differences across multiple platforms; all platforms perform the exact same math.

However, the Xbox 360 requires an additional final step, since it is performing a custom TV gamma for its final image output. This final step effectively undoes the implicit TV gamma. Ideally, this should be done in a tone-mapping shader,

```
float DegammaFuncTV( float f )
{
  return (f <= 0.0812f) ? f / 4.5f :
                          powf(( f+0.099f ) / 1.099f, 1.0f/0.45f);
}
```

Listing 4.1. Undoing TV gamma, according to XDK.

before any quantization happens, but for performance reasons we did it through the hardware gamma ramp functionality.

Our individual steps to modify the gamma ramp are convert to linear space (remove gamma 2.2), remove TV gamma, and reapply the gamma 2.2 curve (see Listing 4.1 and Figure 4.9).

Figure 4.9. Color output equalization on Xbox 360 (left) and PS3 (right).

4.3.3 Deferred Lighting

Our lighting accumulation passes are done into HDR render targets or encoded in RGBK (also popularized as RGBM) for PS3, due to that platform's poor floating-point render target performance. Lower precision formats are insufficient due to lack of range and precision and should be avoided, unless very restrictive conditions can be imposed on the art department (such as avoiding albedo/specular maps below certain luminance thresholds, for example).

Since we are using an encoded format on PS3, hardware blending cannot be used. Luckily, the PS3's GPU allows for a workaround. Blending can be performed in the pixel shader itself by sampling from the same render target into which you are rendering. This feature requires that a pixel be written only once on the same drawcall (no overdraw). Such functionality allows for a very handy form of programmable blending that most hardware (including DX11-level hardware) lacks.

Z-buffer depth caveats. Unlike previous engine iteration, we don't output linear depth into a separate render target. On consoles and on DX10.1/DX11-level hardware (and on certain DX9 hardware) we can access the depth buffer directly. This saves performance, but with a few caveats: first, the depth's hyperbolic distribution needs to be converted back to linear space before it is used in our shaders. Second, first-person objects, like first-person arms and weapons on CryENGINE (and typically every engine), have different depth ranges to avoid intersections with the world and have additional art control, like different FOV (see Listing 4.2).

```
ProjRatio.xy=float2(zfar/(zfar-znear),znear/(znear-zfar));
float GetLinearDepth(float fDevDepth)
{
  return  ProjRatio.y/(fDevDepth-ProjRatio.x);
}
```

Listing 4.2. Device depth to linear depth.

This was problematic particularly because deferred lighting did not work properly for first-person objects. In addition, this resulted in extreme shadowing artifacts. We tried multiple approaches, such as stencil masking or outputting a few depth layers, but ended up simply adjusting our depth reconstruction function to convert hardware depth to a linear depth, using a different depth scale for first-person objects (see Listing 4.3).

```
float GetLinearDepth(float fDevDepth)
{
   float bNearDepth = step(fDevDepth, g_PS_NearestScaled.z);
   float2 ProjRatio= lerp(g_PS_ProjRatio.xy,
                          g_PS_NearestScaled.xy, bNearDepth);
   float fLinearDepth = ProjRatio.y/(fDevDepth-ProjRatio.x);
   return fLinearDepth;
}
```

Listing 4.3. Final linear depth reconstruction.

Reconstructing world-space/screen-space position from depth. In order to reconstruct a world space or shadow space position from a depth sample, we linearly transform the screen-space `VPos` basis directly to the target homogeneous space. This approach is used for all our deferred techniques, such as lights, shadows, fog, and stereo. This is also the simplest way to render light volumes, since it is not possible to use perspective correct interpolators.

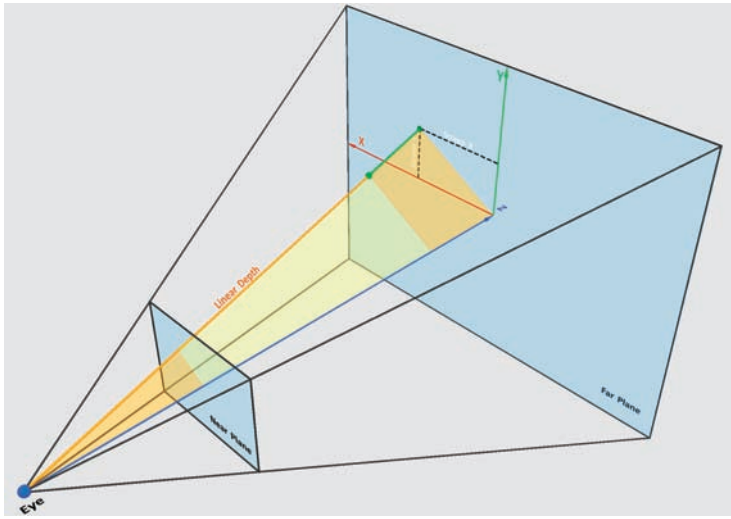


Figure 4.10. Position reconstruction visualization.

Geometric meaning of reconstruction. Essentially, position reconstruction from the depth buffer is achieved by rescaling the depth buffer's linear depth with the addition of three basis vectors. First vX and vY are scaled by screen coordinates (`VPos`) before addition. As the result of the addition of these three vectors, we have a view vector from the eye to the point on the far plane that corresponds to the specified X- and Y-screen coordinates. Simple scaling of this vector with the given linear depth from the depth buffer gives you the position in world space or in any homogenous space. This depends on the space in which the vX, vY, and vZ basis vectors are specified (see Figure 4.10 and Listing 4.4).

```
// projection ratio
float fProjectionRatio = fViewWidth/fViewHeight;
//all values are in camera space
float fFar = cam.GetFarPlane();
float fNear = cam.GetNearPlane();
float fWorldHeightDiv2 = fNear * cry_tanf(cam.GetFov()*0.5f);
float fWorldWidthDiv2 = fWorldHeightDiv2 * fProjectionRatio;
float k = fFar/fNear;
Vec3 vStereoShift = camMatrix.GetColumn0() * cam.GetAsymL();
// Apply matrix orientation
Vec3 vZ = (camMatrix.GetColumn1() * fNear + vStereoShift)* k;
// Size of vZ is the distance from camera pos to near plane
Vec3 vX = camMatrix.GetColumn0() * fWorldWidthDiv2 * k;
Vec3 vY = camMatrix.GetColumn2() * fWorldHeightDiv2 * k;
vZ = vZ - vX;
vX *= (2.0f/fViewWidth);
vZ = vZ + vY;
vY *= -(2.0f/fViewHeight);
```

```
// Transform basis to any local space (shadow space here)
vStoWBasisX = mShadowTexGen * Vec4 (vX, 0.0f);
vStoWBasisY = mShadowTexGen * Vec4 (vY, 0.0f);
vStoWBasisZ = mShadowTexGen * Vec4 (vZ, 0.0f);
vCamPos =  mShadowTexGen * Vec4 (cam.GetPosition(), 1.0f);

//Shader code:
float4 CalcHomogeneousPos(float SceneDepth, float2 WPos)
{
 float4 HPos = (vStoWBasisZ +
  (vStoWBasisX*VPos.x)+(vStoWBasisY*VPos.y) ) * fSceneDepth;
 HPos += vCamPos.xyzw;
 return HPos;
}
```

Listing 4.4. Computing the reconstruction basis.

Ambient passes. Lighting accumulation starts with hemisphere ambient-lighting passes. These can be the outdoor ambient light, which is rendered additively as a full-screen quad, or the indoor ambient light, which is rendered as stencil-tagged regions (which get set during the G-buffer passes), using an indoor volume bounding box. This functionality was helpful to designers, for setting up custom ambient lighting in an indoor area. Hemisphere lighting approximation is computed by warping the Z-component of the world-space normal and using this term to scale the ambient term, e.g., ambient * (N.z * 0.7 + 0.3) (see Figure 4.11).

Figure 4.11. Ambient passes: outdoor ambient lighting (left) and indoor ambient lighting (right). Notice the sharp transition between outdoor and indoor.

Environment lighting probes. Image-based lighting was one of the big novelties introduced in the latest version of our engine. It allows for more accurate diffuse and specular lighting, utilizing special light probe sampling locations that are carefully picked by the artists. At each sampling location an HDR cube map of the scene is captured. From this cube map we generate a low resolution (8×8) diffuse map and a specular cube map, encoded in RGBK. Such cube maps are preprocessed using the ATI CubemapGen tool [Sheuermann and Isidoro 05], which generates correct mip-map levels, taking into account cube map borders,

Figure 4.12. Environment lighting probes passes.

which avoids the noticeable seams that would otherwise result with a more naïve cube map generation.

These environment lighting probes are rendered as a light volume using linear blending and replacing existing ambient lighting. Because lighting artists can control the light multiplier, such lights can also act as "negative lights." G-buffer material glossiness is used for picking which mip level is sampled for the specular cube map to maintain consistency between image-based lighting and traditional specular lighting (see Figure 4.12).

Diffuse global illumination. Another novelty introduced for CryENGINE 3 is an approximation for real-time diffuse global illumination through light propagation volumes. This is an additive indirect ambient pass rendered into light buffers. We will not go into much detail on this feature since it has been covered in depth in previous publications [Kaplanyan 10b].

Contact shadows (SSDO). One of the greatest benefits of SSAO [Kajalin 09] is that it allows objects to appear more grounded by creating a slight drop shadow (sometimes called a "contact shadow") around them. However, since SSAO is meant to attenuate ambient light, it does not take into account any particular light direction. Screen-space directional occlusion (SSDO) [Grosch and Ritschel 10] improves upon this method by accumulating the lighting during the SSAO pass and discarding the light contribution if a given sample vector is occluded. This implicitly creates directional, colored shadows.

Sampling the lighting efficiently during the SSAO pass is not feasible for our deferred lighting pipeline, so we use a different more deferred approach that still gives us the directional and colored contact shadows for each light source. During the SSAO pass, in addition to the regular occlusion term, we also store the average unoccluded direction (usually known as a "bent normal"). The bent normal is just the normalized accumulation of all SSAO sample vectors weighted by each sample's occlusion amount. So if a sample is completely occluded, it does not contribute to the bent normal.

Figure 4.13. SSDO results. Notice the clear contact soft shadows (left). Bent normals target visualization (right).

During the lighting passes, the bent normal can be used to modulate the lighting intensity for each light source. We compute the directional lighting influence $N \cdot L$ as usual and then also compute $N' \cdot L$ using the bent normal (N'). The clamped difference of the two dot products is used to modulate the lighting intensity.

It is desirable that the SSAO shader use a large kernel radius so that the contact shadows are a reasonable size. Furthermore, it is essential that the SSAO implementation produces very clean results without any self-occlusion; otherwise lighting becomes incorrectly attenuated (see Figure 4.13).

Real-time local reflections (RLR). Accurate reflections are difficult to generate efficiently with rasterization-based rendering. The most common techniques for approximating reflections are to re-render the scene once for each planar reflection or create an environment map such as a cube map (which is only truly valid for computing reflections at a single position in space). Both of these techniques have in common that reflections are treated as a special case and will only work for a limited number of reflective objects.

For the *Crysis 2* DX11 upgrade we tried an old idea that we first considered at the beginning of the project: generating the reflections in screen space. The basic idea is simple. For each pixel on the screen, the reflection vector is computed using a per-pixel normal from the G-buffer. Then, in the pixel shader we ray march along the reflection vector to find out if and where the ray intersects the scene. Several samples are taken along the ray, and the z-coordinate of the ray is compared against the depth sampled from the depth buffer at the current step on the ray. If the difference is within a certain interval, we detect a hit and the computed world-space position is re-projected into the previous frame's back buffer to sample the reflection color.

The biggest benefit of this technique is that it can efficiently create accurate reflections on all kinds of visible geometry, even on complex curved surfaces.

Figure 4.14. RLR result. Notice reflections on most surfaces.

However, as the computation is performed in screen space, there are naturally a lot of limitations that need to be taken into account.

An inherent problem is that not all of the required information is available in screen space. The worst case occurs when the reflection vector points in exactly the opposite direction to the view vector. In this case, the back faces of the scene need to be reflected, and these are not visible from the camera's perspective. Instead of displaying an incorrect reflection color, we avoid this situation by smoothly fading out the reflections based on the dot product between the view and reflection vectors. A similar issue occurs when the reflected point is outside of the frame of the back buffer. To avoid hard popping when the perspective is changed, we smoothly fade out the reflections at the screen edges.

For ray marching, the number of samples makes a huge difference in the accuracy of the reflections, especially if a simple linear stepping scheme is used. In order to hide staircase artifacts, we apply some jittering to the length of the sampling steps. This also makes the reflections appear slightly glossy without requiring an additional blur step (see Figure 4.14).

Despite the numerous limitations, the technique can add a lot of visual quality if used carefully for nearby local reflections. It usually works fairly well on ground surfaces like a marble floor, where the reflection vector is pointing in the same direction as the view vector. Also, there are still several opportunities for future improvements. A more sophisticated ray-marching algorithm can be used to improve the reflection quality, and the glossiness of surfaces can be used to alter the sharpness of the reflections.

Light passes. We support point lights and projector light sources in CE3; the lighting model is limited to normalized Blinn-Phong lighting [Hoffman et al. 10].

Figure 4.15. Clip volume enabled (left) and disabled (right). Notice light leaking.

These lights are rendered as full screen quads with stencil volume prepasses if the viewer is inside the light source. Depending on the light's screen coverage, it might be converted into a convex light volume or simply a quad if the light is small enough, since using stencil volume prepasses can actually be slower in some cases. The light volumes are reconstructed in the vertex shader and the geometry used is a tessellated unit cube and frustum projection matrix depending on the light type.

We did several tests with different approaches such as interleaved light rendering or tiled rendering, but for our particular cases, such approaches did not pay off performance-wise and added additional complexity to further handling of lights and light leaking. Light leaking is particularly important, since we don't want a light source to bleed from a wall into an exterior area, but also it is not performance-friendly to have every light casting shadows. For such cases the lighting artists could specify a special clip box/volume shape to be attached to lights; such shapes are used for stencil culling (see Figure 4.15).

For performance reasons, the number of simultaneous shadow-casting lights was limited to four. On the Xbox 360 there was a constant cost penalty for each shadow map of about 0.5 ms due to EDRAM resolve/restore requirements, since all EDRAM regions were used during light accumulation stages. For high-end PCs, almost all lights cast shadows and there is no hard limit on the number of shadow casters, something that was impossible with our previous engine (where we limited the amount of shadow casters to 32).

Shadows. We have been using deferred shadows for the sun since CryENGINE 2 (back in the days of *Crysis 1*, we used just a shadow mask for the sun). Shadows for point lights and projectors are rendered directly into the light accumulation buffers. Shadows in our engine are shadow-map based, highly optimized, and fully dynamic. Point lights are supported by splitting them up into six projected light sources, which is both efficient and very simple to accomplish with our system (see Figure 4.16).

Figure 4.16. Visualizing the sun's shadow mask.

Cascaded shadow maps and point-light shadows. We switched to cascaded shadow maps during the development of *Crysis 1* since this solution better fit all our needs—our levels always consist of big environments with long view distances and everything cast shadows onto everything else. We tried to use different adaptive shadowing schemes that attempted to partially focus shadow maps on the specific areas that were visible to the camera, but this did not work well when the entire space around the camera both casts and receives shadows.

For local lights, a different approach must be used. We split omni-directional lights into six independent frustums. A shadow map for each projection was generated and its size was scaled independently of the other five projections and was based on its projected screen-space coverage. After scaling, a big texture atlas was used to pack all of the shadow maps for a given omni-light. This texture

Figure 4.17. Example scene with 10+ shadow-casting lights (left) and visualizing the shadow atlas (right).

atlas was allocated permanently, which helps avoid memory fragmentation (see Figure 4.17).

Cascade-splitting scheme. Our cascade frustum sizes are based on a logarithmic distribution and the orientation for each shadow frustum is fixed in world space (see Figure 4.18). Shadow frustums are adjusted also to cover camera view frustum conservatively.

More cascades allow better approximation of the logarithmical distribution and improve the density distribution of shadow map texels, thus improving shadow-acne problems for wide view ranges. Additionally, shadow space snapping helped us greatly reduce shadow flickering.

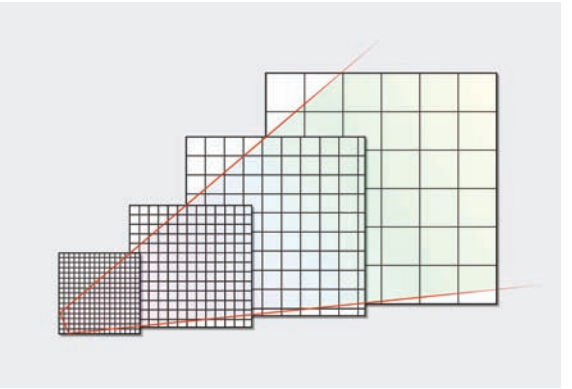
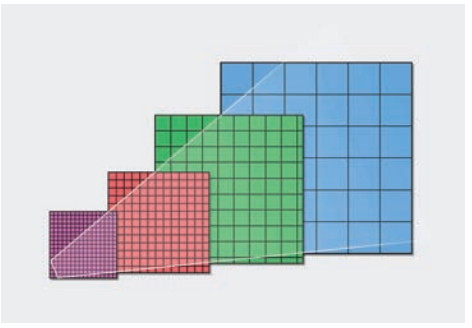


Figure 4.18. Visualizing the individual shadow cascades.

Deferred shadow passes. The shadow cascades are rendered in a deferred way, where potential shadow-receiving areas are selected by stenciling. Such a scheme



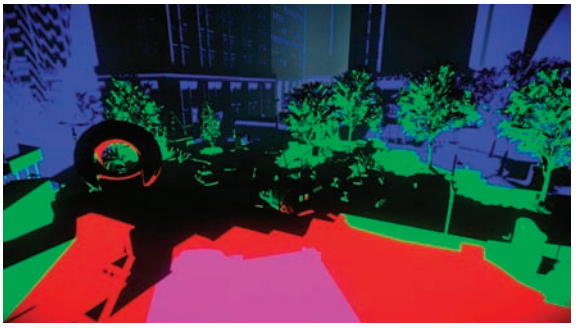


Figure 4.19. Visualizing the shadow cascade stenciling.

is better because the sophisticated splitting algorithm would make cascade selection in the shader too expensive. For the overlap regions of the cascade frustums, we always use the cascade with the highest shadow texels/world unit ratio. For optimizing the shadow-mask generation passes, stencil prepasses are used in order to avoid branching instructions in our shaders, which can be very expensive on consoles (see Figure 4.19).

Shadow cascade caching. For performance reasons, on consoles or on lower spec PCs, updates of shadow cascades are amortized over several frames. This allows us to have more cascades within the same performance restrictions, meaning better shadow-map density distribution. It works by updating more distant cascades less frequently. The most distant cascade uses VSM and is blended additively with the shadow mask. This allows us to have huge penumbras from huge distant objects.

Shadow acne. Shadow acne is typically caused by low shadow-map texel density and insufficient precision in shadow depth buffers (see Figure 4.20). We use different solutions for different cases to overcome issues due to low texel density during shadow-map rendering:

- *Sun shadows.* Render front faces only with slope-scaled depth bias.
- *Point light shadows.* Render back faces only (works better indoors).
- *Variance shadows for distant LODs.* Render both front and back faces to shadow maps due to very low shadow map texel density.

Additionally, a constant depth bias was used during deferred shadow passes to overcome precision problems.

Figure 4.20. Shadow acne example.

Figure 4.21. Particle casting transparent shadows.

Shadows and transparency. For transparent geometry, an additional forward rendering pass is performed for shadow-receiving geometry. For cascaded shadow maps, this means finding the affected geometry and figuring out which cascade frustums it uses. This is computed while processing shadow casters on the CPU. As it turns out, about 90% of the cases are affected by one cascade and the rest by two cascades.

For casting a transparent shadow, we generate a separate render target where we accumulate transparency alpha values (see Figure 4.21).

The steps for rendering translucency maps include:

1. Depth testing should be done using the regular opaque shadow map to avoid back projections.

2. Accumulate translucency alpha values. (Transparent objects can be sorted front to back to have proper accumulation of transparency alpha values.)

Finally the shadow depth map and the translucency alpha map are both used during a deferred shadow pass, computed as follows:

```
InShadow = max(translShadow, opaqueShadow).
```

4.3.4 Deferred Decals

Forward decals, especially projective forward decals, have quite a few drawbacks, which require additional memory (a big problem for consoles), mesh re-allocations (causing memory fragmentation), and CPU time for dynamic mesh creation. In

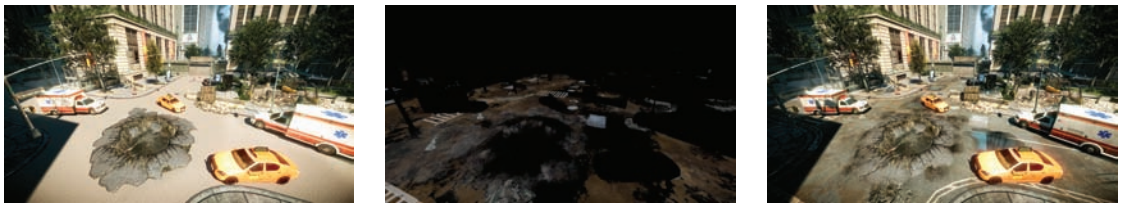


Figure 4.22. Deferred decals visualization. No decals (left), deferred decals layer (middle), and final composition (right).

addition, they are problematic with DX11 Tessellation + Displacement (intersections). Replacing the rendering of forward decals by deferred decals [Krassnigg 11] helped us to solve all these problems. Our deferred decals are rendered into a separate diffuse and normal buffer using a box volume shape. Such decals are applied to static geometry only, do not require any allocation for dynamic meshes, and reduce shader computation cost (see Figure 4.22).

All decals share the same shading and due to simplicity, they are very performance friendly, allowing designers to use a lot of them. They are shaded during forward shading passes by fetching the layer of accumulated deferred decals. Leaking through dynamic geometry and/or walls is the biggest problem with such an approach (see Figure 4.23). Besides only applying the approach to static geometry, we tried to further address this in several ways. For example, we used an adjustable decal volume and attenuation function based on a decal normal, and a world-space normal dot product.

Figure 4.23. Deferred decal leaking.

4.4 Forward Shading Passes

The biggest strength of deferred lighting (and weakness in performance terms), is that shading is done in an additional forward pass. This allows for a great deal of material variation and flexibility when compared to "vanilla" deferred rendering.

Lighting buffers are fetched as required (depending on material properties) albedo and specular maps are modulated as usual, and a per-material Fresnel term is applied (see Figure 4.24).

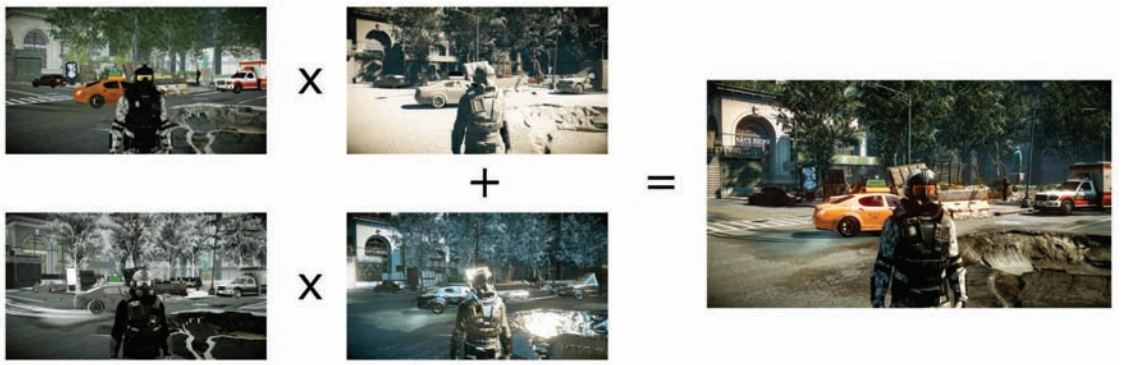


Figure 4.24. Shading composite overview.

To minimize the bandwidth impact due to so many full-screen targets being fetched, on the PS3, lighting buffers are stored in a 32-bit format using an RGBK encoding. On the Xbox 360, lighting buffers are resolved from A2B10G10R10F_EDRAM into A2B10G10R10 instead of A16B16G16R16F, since there is no corresponding FP10 texture format; this integer format is bitwise equivalent to FP10, and it is converted when sampled into a 7e3 floating-point format as proposed by Cook [Cook and Gould 08]. This allows us to minimize the resolve costs from 1.0 ms down to 0.5 ms and trade off bandwidth for ALU cost. Additionally, if a surface's material properties are mostly diffuse, then we only fetch from the diffuse lighting buffer to save on wasted bandwidth.

Deferred skin shading. The biggest strength of deferred lighting is material variation, and we tried to take advantage of it. One of our early ideas during development was to reuse all diffuse lighting accumulation for approximating subsurface scattering in UV, or even screen space. The idea was simply to project from screen space into UV space and to avoid redundant lighting computations, which are present in the vanilla approach [Borshukov and Lewis 03]. Although this worked, it still suffered from the limitations of the original algorithm: a need for a texture atlas into which all characters would be rendered, hence additional precious memory, additional geometry pass, unique UV texturing, and a constant

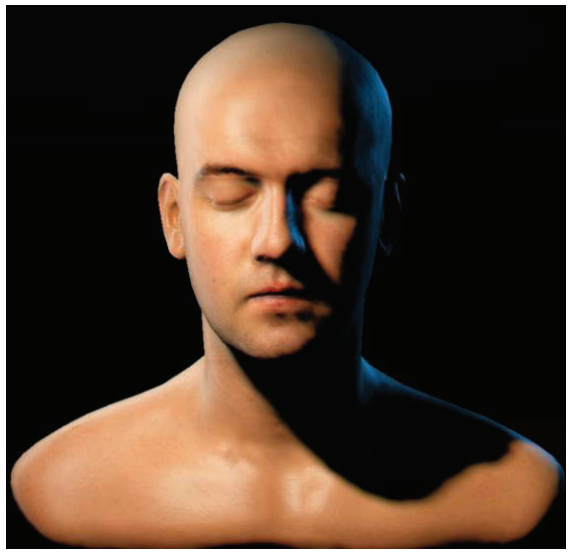


Figure 4.25. Deferred skin shading in action. (*Head model courtesy Infinite-Realities.*)

frame cost for each update. We were able to avoid UV space altogether by implementing our subsurface scattering approximation in screen space instead. This worked around most limitations and does not require additional render target memory, which was the biggest gain.

Screen-space subsurface scattering is implemented by performing a bilateral blur, using a Poisson distribution for taps, during the geometry pass. The kernel size is depth dependent. This might sound expensive, and it is, but because character faces and arms tend to be small on screen, it turns out that in practice, this approach is quite affordable since the overall cost of the technique is proportional to the number of screen pixels to which it is applied (see Figure 4.25).

Screen-Space self-shadowing. Self-shadowing still remains challenging for macro-geometry details and particularly for character faces. However, self-shadowing is an important visual hint and its significance should not be underestimated. Many games achieve high-quality self-shadows for their characters by generating a shadow depth map for each character—sometimes just for the closest few characters. This method works quite well but it requires additional memory for each shadow map and a custom code path specialized for handling character shadows.

One idea we had during production, of which we were initially very skeptical, was to approximate shadows in screen space for such cases. As it turned out, this method works relatively well and has no additional memory requirements. This approximation consists of ray marching through the depth buffer along the

Figure 4.26. Screen-space self-shadowing in action. Enabled (left) and disabled (right).

screen-space light vector during the geometry pass. As in other screen-space techniques, the runtime cost of this approximation is proportional to its size on screen. Fortunately, for characters, only a small amount of screen space is required and this technique performs quite well. We ended up using screen-space self-shadowing for our character's skin, hair, and eye rendering (see Figure 4.26).

Soft alpha test. Efficient hair rendering is a big challenge, particularly on the current generation of console hardware. This challenge is made even more difficult when using deferred rendering techniques such as deferred fog and deferred shadows. Most games avoid using alpha blending for hair, either because it is too

expensive or because it is incompatible with many deferred rendering techniques. Instead, most people chose to use alpha testing for hair. Unfortunately, alpha testing results in a cartoon-like look that spoils the realism of your characters.

One solution we came up with, again very early during the project, was inspired by existing image space motion-blur techniques. We attempted to smooth hair geometry along its per-pixel screen-space tangent vector. This is done during a separate geometry pass, which is still quite a bit cheaper than brute force alpha blending (which typically requires three passes for approximating back to front sorting). Once again, the cost of this screen-space technique is proportional to its size onscreen and for characters (who aren't extraordinarily hairy), this performs amazingly well (see Figure 4.27).

The exact same concept can be used for fur rendering but rather than smoothing along the tangent, for fur you should smooth along the screen-space surface normal.

Figure 4.27. Soft alpha test in action. Enabled (left) and disabled (right).

4.5 Batched HDR Postprocessing

The main goal for optimizing performance while postprocessing is to reduce, as much as possible, the amount of work that must be done and to share computations wherever possible.

Camera and object motion blur. One difference between our newest engine and the previous engine is that we rid ourselves of our old approach of using "a sphere around camera" to approximate camera motion blur. Our new approach computes camera motion blur in a single pass instead of several passes, which the old approach required.

For static geometry, reprojection is used since it is straightforward to compute how much a pixel has moved, compared to the previous frame. This is computed

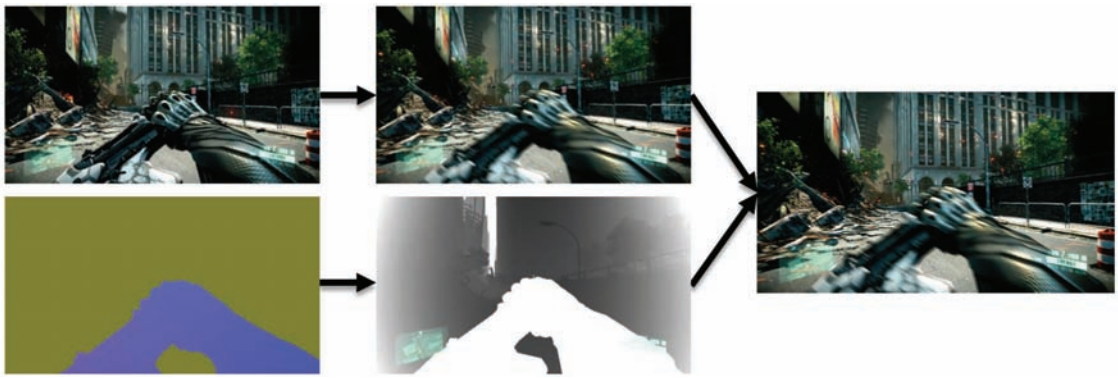


Figure 4.28. Visualizing individual stages for motion blur and composition.

using a per-pixel depth value and the previous frame's view projection matrix to find the previous frame's pixel location and then using that to compute a delta (velocity) to the current frame's position. To handle dynamic geometry, like skinned characters, we have no choice but to output velocity to a render target.

The velocity values are used to do a directional blur. Performing such a directional blur at full resolution means at least 3 ms spent for blurring an LDR render target (this cost is even higher if blurring an HDR image), and not really much can be done to speed up the worst-case scenario.

Inspired by the *Killzone 2* approach [Valient 09], we ended up using a half-resolution buffer instead of a full-screen buffer. To save bandwidth, we store 2 RGBA/8 targets, one that uses RGBK encoded colors with the final blurred result and the other with the composition mask. On a PC, we fall back to straightforward 16-bit floating-point render targets with the mask in alpha.

Object velocity buffer dilation is done, in a similar way to what we did in *Crysis 1* [Sousa 08]. This avoids unpleasant sharp silhouettes by slightly dilating the values stored in the velocity buffer. One difference from our earlier approach is that we now do this on the fly, in the same pass in which we apply the motion blur, which proved to be good enough and allowed us to avoid additional passes and the associated resolves steps that would be necessary on the Xbox 360 (see Figure 4.28).

The directional blur is done in linear color space, using HDR values, before tone mapping, which is more physically accurate and allows for the propagation of bright streaks [Debevec and Malik 97]. For consoles, we limited our kernel to nine taps and clamped to a maximum range to avoid undersampling. For high-spec'd PCs, we allow up to 24 taps.

Bokeh DoF: just another kernel and weights. It turns out that DoF, or any blur, is just a special case of the previous directional blur. It is essentially just another

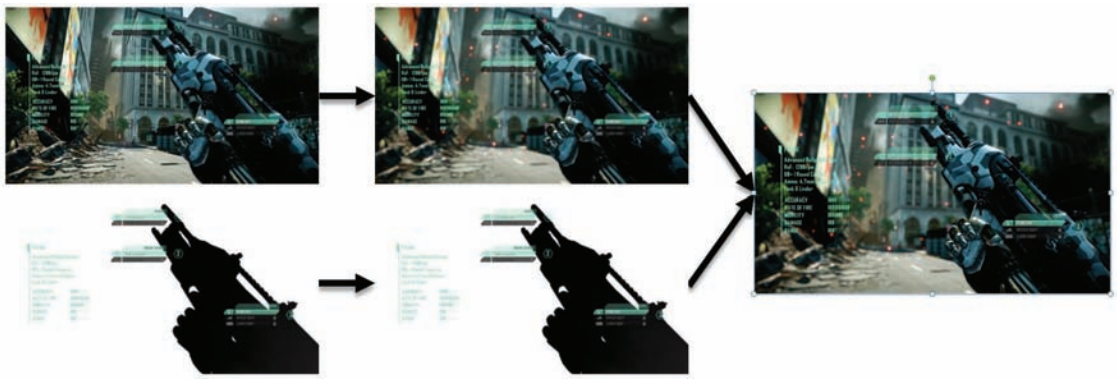


Figure 4.29. Visualizing individual stages for bokeh DoF and composition.

blur kernel with taps offsets in a disk-shaped pattern. The kernel size is scaled based on the circle of confusion (CoC) term—a term that is proportional to how "out of focus" a pixel is. In order to make this less expensive, we reuse the same nine taps (24 on PCs) mentioned above (see Figure 4.29).

Batching it all together. In the end, what we really want is to blur the image and to reuse all taps whenever possible in order to share as much work as possible. The kernel offsets are morphed based on the blur type: directional versus disk, masked blur, radial blur, distant DoF, and other cases.

The final composition is done in our tone-mapping pass. Note that this is not necessarily physically correct, since "final composite" should be done into an HDR target so that eye adaptation, bloom, and other stages are 100% correct. But we cheated to save performance at the cost of very minimal visual sacrifice. In the end, with one blur pass, we were able to achieve motion blur, depth of field, and many other blur-based effects for a constant frame cost of 1 ms on consoles.

Motion blur for ultra specs. For the DirectX 11 Ultra Upgrade, we added a very high-quality mode for certain effects. One of the effects that got a nice boost was motion blur (see Figure 4.30).

This mode is a single pass at full-screen resolution, using 12 taps (once again clamping the maximum range to limit undersampling). The alpha channel stores the objects' ID, and these are used to implement a directional blur masking scheme based on velocity and ID. For each tap we compute:

- If $||V|| >$ threshold then allow bleeding, else reject the tap.
- Early out if $||V|| <$ threshold.

Figure 4.30. Example of "ultra specs" motion blur (left); visualizing bleeding masking (center); and bleeding masking disabled (right).

Bokeh DoF for ultra specs. For the DirectX 11 Ultra Upgrade, we optimized the GPU-side so much that we had quite a bit of free performance to use (something in the order of 16 ms). One of the very high-quality modes with which we went a bit crazy was a technique commonly used in the movie industry to approximate high-quality DoF [Cyril, Sylvain, and Olivier 05]. Essentially, we render a quad/sprite per pixel in order to perform an arbitrarily sized masked blur (see Figure 4.31).

These quads are scaled by the circle-of-confusion factor using a geometry shader, and are additively accumulated into an intermediate render target. The quads can also feature a camera-aperture-shaped mask for different bokeh shapes (spherical, hexagonal, etc.). We employed dithering to minimize noticeable precision loss (see Figure 4.32).

Figure 4.31. DirectX 11 DoF with circular bokeh.

Figure 4.32. "Ultra Specs" DoF. Visualizing foreground and background layers.

There is no explicit depth masking, instead quads are sorted per background and foreground layers. Quad count is accumulated into a destination target alpha channel. During composite with the final scene, we normalize the accumulated color values by dividing by each pixel's alpha value. This technique provides a very high-quality, although physically incorrect, depth-of-field blur but it is also very expensive. The naïve implementation takes 100 ms for large defocus at 1080 p. Unfortunately, even high-end hardware is not currently fast enough to perform this technique at full-screen resolution.

Bokeh DoF for ultra specs: making it fast. In order to optimize our high-quality depth of field we first had to switch to performing it at quarter-resolution (half-width and half-height buffer). Additionally, quads were rejected using an interleaved pattern, which helped improve the fill-rate requirements of this technique. We also found that using "early out" conditions, both on the geometry shader and pixel shader side, and using a spherical aperture shape (which can be computed using ALU instructions in the shader instead of texture fetches) helped improve performance further. Although we did not do this, avoiding geometry shader usage could have helped performance as well, particularly on hardware that has weak geometry shader support.

After the front and back layers are generated, they are composited with the final scene as follows:

- Back layer is composited using full resolution CoC, computed based on full resolution depth target.
- Front layer is composited based on layer alpha, which works quite well, since the front layer should bleed into the background.

Merging quad-based bokeh DoF with motion blur. One idea that occurred to us late in the DX11 ultra upgrade project was to merge the DoF quad-based approach with motion blur in a single pass. This is achievable by simply extruding quad geometry and using geometry shader along the motion direction.

Figure 4.33. Screen space reprojection stereo in action (anaglyph output).

4.6 Stereoscopic 3D

With the recent success of 3D technology in the movie industry and a growing number of consumer devices that are capable of displaying 3D content, stereoscopic 3D is becoming increasingly important for video games. Although adding basic 3D support to an engine is straightforward in principle, in practice it can be challenging to get the performance right, and to ensure that the user can play the game comfortably over a longer period without eye strain (see Figure 4.33).

4.6.1 Generating the Stereo Image Pair

Stereo 3D (s3D) requires that the user be presented with two slightly different images, one for the left eye and one for the right eye. The standard way to generate this image pair is to render the scene two times from carefully shifted perspectives. While this approach is straightforward for a 3D engine and yields perfect results in terms of quality, it can be problematic for a graphics-heavy game that makes full use of the available hardware resources. If a game is GPU-bound, rendering the scene a second time will often mean that the frame rate will be cut in half—something that is usually unacceptable for a console title that must run at 30 fps. In order to overcome this performance issue, one must reduce the rendering resolution, the geometric detail, and the number of effects. Obviously, this can result in a very noticeable degradation of visual quality when entering 3D mode.

Reprojection. A completely different approach to achieving s3D is to generate the image pair from the existing nonstereo rendering data. When the depth for each pixel is available, it is possible to reproject each fragment into the space of the left and right eye cameras. Assuming that the scene was rendered from a regular "center" perspective, this is done by reconstructing the view space position for each pixel and applying the left and right camera view-projection matrices. The pixel is then plotted at the computed image-space location. Unfortunately, this point splatting approach requires data scattering, which is not efficient on current generation console GPUs. Furthermore, the generated images will contain holes at locations where no pixels were plotted. No information is available in the original image for these locations, so they need to be filled with some estimated color during a second pass.

Image displacement. As we required a fully gather-based approach that can be executed efficiently in a pixel shader, even on older console hardware, we ended up using a simple technique that can be considered a kind of image displacement. This technique is executed as a postprocessing step on the final back buffer, before the HUD is rendered.

We use two basic parameters for our s3D view setup: the maximum separation (`MaxSeparation`), which is the distance between the left and right eye cameras in normalized screen space; and the distance to the virtual screen plane, which we call zero parallax plane (ZPP). If the depth of a point is the same as the ZPP distance, it appears to be exactly on the screen, while a smaller or larger depth will make the point come out of or go into the screen.

The parallax is the distance on the ZPP between the positions of a point that was projected into the left and right views. Using Thales' theorem, the parallax is straightforward to compute (see Figure 4.34).

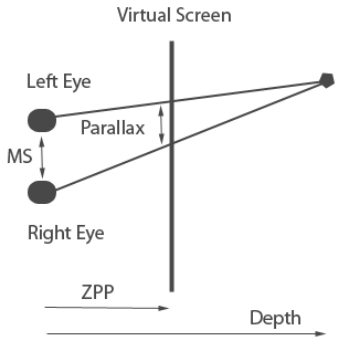


Figure 4.34. Parallax = MaxSeparation * (1 − ZPP / Depth).

The computed parallax is used as an offset to sample the back buffer. The parallax is added for the left eye and subtracted for the right eye. This creates two images that are displaced in a way that the brain expects; therefore, they will create the perception of 3D.

However, performing this displacement in image space reveals two basic problems: occlusion and disocclusion. Disocclusion occurs when a part of an object should be visible in the left or right eye view but is hidden in the original view. This issue is not handled explicitly or correctly with our solution. The center view is used for synthesizing the two images, so the error gets distributed for both eyes. Luckily, the brain seems to be quite forgiving of this kind of error if the stereo parameters are not too extreme.

The second issue, that of occlusion, can be more noticeable. It happens when a background object with a high parallax is occluded by a closer object with a lower parallax. In this case, the nearby object would be sampled by the background although the background should be sampled instead. Luckily this case is easy to detect by checking if the depth of the offset sample is closer. When this happens, the background can just be cloned. In our implementation, we simply search for a minimum depth in the MaxSeparation range and use that to compute the parallax. This works because a smaller depth results in a smaller offset for a positive parallax. A basic version of the s3D algorithm we use is shown in Listing 4.5.

```
const float samples[3] = { 0.5, 0.66, 1 };

float minDepthL = 1.0;
float minDepthR = 1.0;
float2 uv = 0;

for( int i = 0; i < 3; ++i )
{
    uv.x = samples[i] * MaxSeparation;
    minDepthL = min( minDepthL, GetDepth( baseUV + uv ) );
    minDepthR = min( minDepthR, GetDepth( baseUV - uv ) );
}

float parallaxL = MaxSeparation * (1 - ZPP / minDepthL );
float parallaxR = MaxSeparation * (1 - ZPP / minDepthR );

left = tex2D( backBuf, baseUV + float2( parallaxL, 0 ) );
right = tex2D( backBuf, baseUV - float2( parallaxR, 0 ) );
```

Listing 4.5. Basic s3D algorithm.

Although the algorithm works surprisingly well for carefully chosen stereo parameters, there are some artifacts that are inevitable due to the nature of this screen-space approach. As the background is getting cloned in the case of occlusion, some halos can appear around objects. These can be fairly noticeable

sometimes, but usually they do not disturb the perception of 3D. The screen edges need some special handling, as it is possible that the algorithm samples outside the image. This can easily be solved by cropping the image a bit at the sides. Finally, it needs to be noted that the technique does not work for transparent objects, as they usually do not write to the depth buffer. Transparent objects just get the depth of the background, which can look more or less wrong depending on the situation.

4.6.2 Viewing Comfort

Stereo 3D is essentially tricking the viewer's brain in order to produce the impression of depth. It is important that a few basic rules are not violated, otherwise the brain can get confused and the experience can result in uncomfortable side effects for the viewer.

For *Crysis 2* we decided that everything should have positive parallax and hence go into the screen, so that the screen is a window into the world. Not having any objects come out of the screen completely avoids window violations where an object with negative parallax would be cut by the screen borders, which would hurt the illusion. While negative parallax is considered as a "wow" effect in s3D movies, it can quickly fatigue the viewer as it requires strong refocusing and is hard to control in games.

One of the most important rules for keeping the s3D viewing comfortable is to avoid depth conflicts as much as possible. A depth conflict exists when the perceived stereo depth does not match the rendered depth. For example, if a crosshair is rendered on top of a wall, but the separation is higher than that of the wall, the brain will get confused by the wrong hints and can react with headaches and sickness after some time.

Avoiding depth conflicts. Giving a certain amount of depth to HUD elements and overlays can greatly improve the overall stereo experience. While static HUD elements can be placed carefully in s3D space to avoid intersections with the world, some more dynamic HUD elements may require additional effort. In *Crysis 2* the crosshair is located a few meters away from the weapon to look more convincing. However, when getting close to a wall, there will be an annoying depth conflict between the world and the crosshair. To avoid this, a ray-cast against the world is performed and in case of a hit, the crosshair is smoothly moved towards the viewer to avoid the intersection.

HUD elements usually get rendered after the scene without depth testing. Thus, there can be a depth conflict between the first-person weapon and the crosshair that is supposed to be in front of the gun. Although it is often just a few pixels that are conflicting, we wanted to get rid of that issue, as we found it to be disturbing enough while playing the game for a long time to warrant our attention. We ended up writing a mask for the weapon in the alpha channel of

the left and right eye buffers during the stereo image generation. When drawing HUD elements located in the scene, a blend mode is used that fades out, and hence occludes pixels that are covered by the weapon mask.

In order to get enough depth for the scene, a reasonably high distance for the zero parallax plane must be chosen. This is problematic for a first-person shooter where the weapon is always very close and would come out of the screen. Not only does this look unnatural, it also makes the viewer exhausted as the eye needs to refocus a lot more. In order to avoid this, the weapon is pushed into the screen during the stereo image synthesis by offsetting and scaling the depth values.

Unfortunately, pushing the weapon artificially into the screen is a serious source of depth conflicts when getting close to objects. This is especially annoying in a shooter where the player can take cover. To avoid these depth conflicts, we try to find out when the player is close to some world geometry. The most accurate way to achieve this would be to analyze the depth buffer, however, for simplicity we just do a few ray casts. In case a potential depth conflict is detected, the ZPP distance is smoothly reduced based on the distance to the closest object. This reduces the overall depth of the scene without being very obvious to the player.

4.7 Conclusion

We have covered here only a very small part of the immense work that went into our latest engine during a period of almost three and a half years. We hope that this chapter has effectively conveyed the massive team effort that went into the creation of CryENGINE 3 and *Crysis 2*.

4.8 Acknowledgments

For their support and helpful feedback, we thank in particular Vaclav Kyba, Michael Kopietz, Carsten Wenzel, Vladimir Kajalin, Andrey Konich, Ivo Zoltan Frey, Marco Corbetta, Christopher Evans, Chris Auty, Magnus Larbrant, Pierre-Ives Donzallaz, and Christopher Oat.

Last but not least, we thank the entire Crytek Team.

Bibliography

[Borshukov and Lewis 03] George Borshukov and J. P. Lewis. “Realistic Human Face Rendering for ‘The Matrix Reloaded’.” In *ACM SIGGRAPH 2003 Sketches & Applications (SIGGRAPH ’03)*, p. 1. New York: ACM, 2003.

[Cook and Gould 08] David Cook and Jason Gould,“Xbox Textures—Formats, Conversion, Fetching and Performance,” Microsoft. Gamefest 2008. Available online (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=1166).

[Cyril, Sylvain, and Olivier 05] Pichard Cyril, Michelin Sylvain, and Tubach Olivier. "Photographic Depth of Field Blur Rendering." Available online (http://wscg.zcu.cz/wscg2005/Papers_2005/Short/H13-full.pdf). 2005.

[Debevec and Malik 97] Paul E. Debevec and Jitendra Malik. "Recovering High Dynamic Range Radiance Maps from Photographs." In *Proceedings of SIGGRAPH '97, Computer Graphics Proceedings, Annual Conference Series*, edited by Turner Whitted, pp. 369–378, Reading, MA: Addison Wesley, 1997.

[Gritz and d'Eon 08] Larry Gritz and Eugene d'Eon, "The Importance of Being Linear," In *GPU Gems 3*. Edited by Hubert Nguyen, pp. 529–542. Reading, MA: Addison Wesley, 2008.

[Grosch and Ritschel 10] Thorsten Grosch and Tobias Ritschel. "Screen-Space Directional Occlusion." *GPU Pro*. Edited by Wolfgang Engel, pp. 215–230. Natick, MA: A K Peters, 2010.

[Hoffman et al. 10] Natty Hoffman, Yoshiharu Gotanda, Ben Snow, and Adam Martinez. "Physically-Based Shading Models in Film and Game Production." SIGGRAPH course 2010.

[Kajalin 09] Vladimir Kajalin. "Screen-Space Ambient Occlusion." *ShaderX^7: Advanced Rendering Techniques*. Edited by Wolfgang Engel, pp. 412–424. Hingham, MA: Charles River Media, 2009.

[Kaplanyan 10a] Anton Kaplanyan. "CryENGINE 3: Reaching the Speed of Light." Crytek. Available online (http://www.crytek.com/cryengine/presentations/CryENGINE3-reaching-the-speed-of-light). 2010.

[Kaplanyan 10b] Anton Kaplanyan. "Real-time Diffuse Global Illumination in CryENGINE 3." Crytek. Available online (http://www.crytek.com/cryengine/presentations/real-time-diffuse-global-illumination-in-cryengine-3). 2010.

[Krassnigg 11] J. Krassnigg. "A Deferred Decal Rendering Technique." *Game Engine Gems 1*. Edited by Eric Lengyel, pp. 271–280. Sudbury, MA: Jones and Bartlett, 2011.

[Reinhard et al. 10] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Second edition. San Francisco: Morgan Kaufmann, 2010.

[Sheuermann and Isidoro 05] Thorsten Scheuermann and John Isidoro. "Cubemap Filtering with CubeMapGen." AMD Developer Central. Available online (http://developer.amd.com/gpu_assets/GDC2005_CubeMapGen.pdf). 2005.

[Sousa 08] Tiago Sousa. "Crysis: Next Gen Effects." Crytek. Available online (http://crytek.com/cryengine/presentations/crysis-next-gen-effects). 2008.

[Tchou 07] Chris Tchou. "HDR The Bungie Way," Microsoft. Gamefest Unplugged (Europe) 2007. Available online (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=21523). 2007.

[Valient 09] Michal Valient. "Rendering Technology of Killzone 2." GDC 2009.

[Vlachos 08] Alex Vlachos,"Post Processing in The Orange Box," GDC 2008. Available online (http://www.valvesoftware.com/publications.html).