IX 3

Making it Large, Beautiful, Fast, and Consistent: Lessons Learned Developing *Just Cause 2*

Emil Persson

3.1 Introduction

Just Cause 2 is a sandbox game developed by Avalanche Studios for PC, Xbox 360 and PLAYSTATION 3. Its main visual traits are an enormous landscape with huge draw distance, an over-the top visual style, varied environments from forests to cities, different climate zones from desert to jungle, and a continuous day cycle. Developing *Just Cause 2* turned out to be a challenge in many ways. We had three different platforms with wildly varying abilities and performance characteristics. We also found that the platforms sometimes behaved differently even when running equivalent code.

This article covers some of the more interesting graphical techniques that give the game its distinctive look as well as how we dealt with performance issues and the efforts we made to keep the visual appearance consistent across platforms.

3.2 Making it Large and Beautiful

3.2.1 Dynamic Lights

The original *Just Cause* had some support for dynamic lights, but was limited to a few lights active at any given time for performance reasons. For *Just Cause 2*, we wanted to improve that. Instead of solving the problem by moving to a deferred renderer we came up with a technique for rendering a large number of dynamic lights with forward rendering without multi-passing or otherwise increasing the number of draw calls.

X	Y	Z	$1 / R^2$
R	G	B	—

Table 3.1. Light constants.

Light indexing. The fundamental problem with rendering multiple lights without multi-passing is how to feed the information about the lights to the shader, particularly when there are a large number of them. One solution is to provide that information in screen-space, such as in the *light-indexed deferred rendering technique* [Trebilco 09]. However, it is quite prone to artifacts to do so since chances are that several lights line up and occupy the same area in screen-space: e.g., looking down a line of street lamps, or just about any area where there are many semi-large lights around. Instead, we found that doing it in world space has much fewer overlaps. Furthermore, the overlap is not view-dependent, which means that we can design things to keep overlap within the technique's limitations.

We provide the light information to the shader through a 128×128 light index texture in RGBA8 format. This texture is mapped in the XZ plane around the camera position and is point-sampled. Each texel is mapped over a $4\text{m} \times 4\text{m}$ area and holds four indices to the lights that affect that square. This means we cover an area of $512\text{m} \times 512\text{m}$ where dynamic lights are active. The active lights are stored in a separate list, either in shader constants or as one-dimensional textures depending on the platform. Although with 8-bit channels one could index up to 256 lights, we limited the system to 64 simultaneous lights in order to fit the light information into shader constants. Each light takes two constant registers holding the position, reciprocal squared radius, and color (see Table 3.1).

Additionally, there is an extra "disabled" light slot with all these set to zero. This brings the total register count to 130. When one-dimensional textures are used, the disabled light is instead encoded in the border color. The position and reciprocal squared radius is stored in RGBA16F format and the color in RGBA8. In order to preserve precision, the position is stored in a local space relative the center of the texture.

The light index texture is generated on the CPU from the global list of lights. First its location is placed such that the texture area is fully utilized and as little space as possible ends up behind the camera. Among the lights that are enabled and fall within the area of the index texture, the most relevant lights are selected based on priority, approximate size on the screen, and other factors. Each light is inserted into available channels of the texels it covers. If the texel is covered by more than four lights we need to drop lights. If at insertion time a texel is full, we check whether the incoming light should replace any of the existing lights, based on maximum attenuation factor in the tile to reduce the visual error of dropped lights. These errors show up as lighting discontinuities around tile

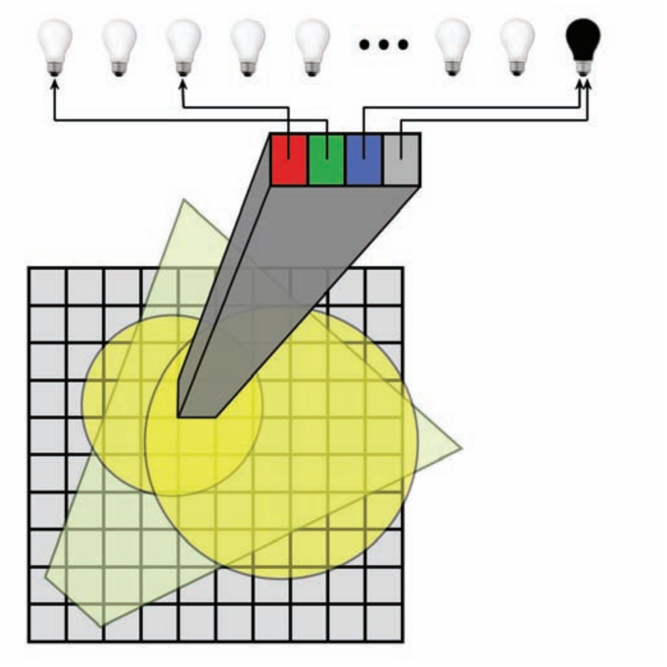

Figure 3.1. Light indexing in axis-aligned world space. Texture is placed such that as much of its area as possible is within the view frustum. The illustrated 4m × 4m area is intersected by two lights that are indexed by R and G channels. The unused slots reference the disabled light.

borders. Generally these errors are small, but they can be very obvious when they move around. To avoid that problem, we snap the index texture to texel-sized coordinates. In practice, dropped lights are quite rare, and where they occur it is usually hard to detect, even if you know what to look for. See Figure 3.1.

Lighting model. For performance reasons the lighting model for the dynamic lights is quite simple. It consists of diffuse lighting, specular lighting, and the attenuation factor. The specular lighting is used primarily for characters. The PC version of *Just Cause 2* also offers specular for the dynamic lights on a wider range of materials.

An important aspect of the lighting equation is that lights need to be range limited and fall to zero at the light radius. The traditional attenuation equation as used in the past by fixed function lighting was thus not an option, besides that it is very expensive. Initially we implemented probably the cheapest useful attenuation possible:

```
atten = saturate(1 - lightpos.w * dot(lVec, lVec));
```

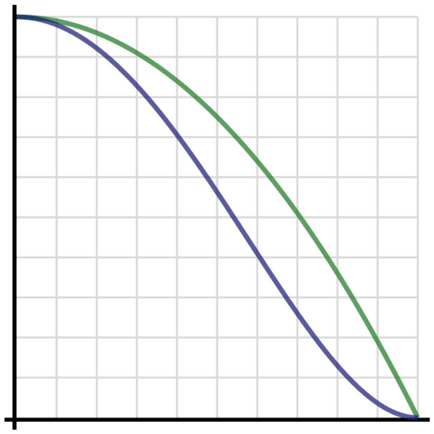

Figure 3.2. The new attenuation function (blue) compared to the old (green).

The `w` component of `lightpos` holds the reciprocal squared radius (see Table 3.1), which makes this equation fall to zero at the light radius. This amounts to a three component vector and a scalar instruction per light. Other than speed, the advantage of this equation compared to more realistic variants is that it preserves a lot of the energy quite far out in the light sphere, which means you can keep your lights smaller. The disadvantage is that it creates a quite blobby light, which often looks unrealistic. We also found that when applied to vertex lighting, it often became very obvious that lighting was done per vertex. After frequent artist nagging we looked for a better attenuation curve without compromising performance. We found that this simple modification tremendously improved the visual quality without sacrificing much speed or reducing the high energy attribute too much:

```
atten *= atten;
```

This creates an attenuation curve that is roughly a reversed s-curve (see Figure 3.2).

Optimizations. One of the first and most obvious optimizations we did was to add dynamic branching to avoid doing any lighting computations for the majority of the pixels where no dynamic light is needed. We found performance to be best when nesting two branches. So we either evaluate zero, two, or four lights. With this optimization we found that this much more flexible system generally performed at the same level as the old lighting system; sometimes a bit slower and sometimes a bit faster. Unfortunately, one of the consoles is rather lacking in the dynamic branching performance. The branching coherency is excellent though,

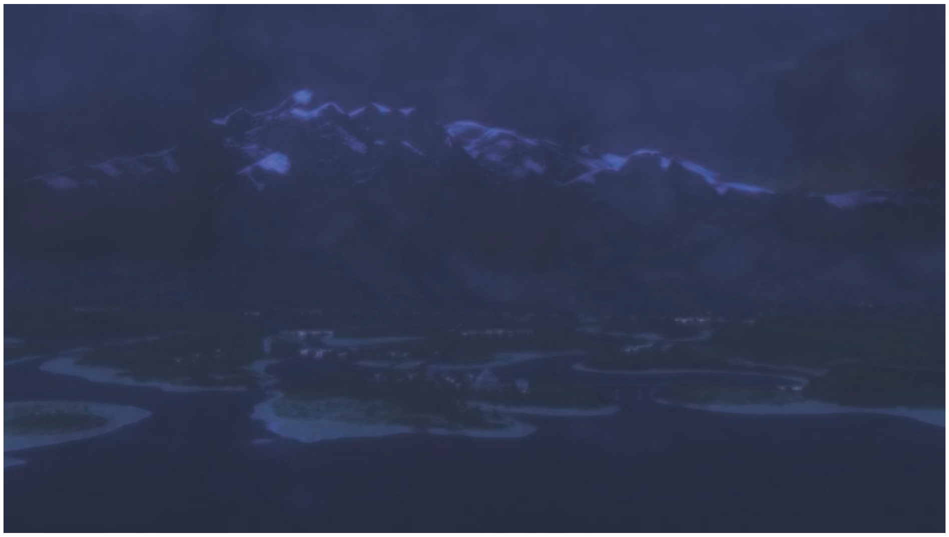

Figure 3.3. Distant lights in action.

so we do see a decent performance improvement where there are no lights. But there is a constant overhead of having the branching code in the shader in the first place, which makes the worst case performance drop noticeably. So we need other tools as well to maintain good performance on all platforms.

Many objects are limited in size such that they are normally not affected by more than four lights in the first place. We implemented a system to look up the best lights for such objects from the bounding box and instead pass the lights in a small array of constants and used different shaders for different light counts. For certain materials, such as for instance foliage, we found that it was good enough to use vertex lighting. For platforms that use lights in constants we also changed the index texture format to use integers instead of normalized values, which saved a few instructions.

Distant lights. Since the light index texture is 128×128 and each texel is $4\text{m} \times 4\text{m}$ the dynamic lights system covers a $512\text{m} \times 512\text{m}$ area. Given the size of our world, it is desirable to give lights beyond that which have a visual impact as well. We added a distant lights system that added a lot of life to night scenes (see Figure 3.3). We simply rendered all lights as point sprites. In DirectX 10, where point sprites are deprecated, we emulate them using two triangles.

In addition to the visual improvement, it really helped gameplay at night to have roads, settlements, and various interesting spots in the world light up and made player navigation easier and more interesting.

3.2.2 The Shadowing System

For shadows we use *cascaded shadow mapping* [Engel 06, Zhang 09]. We use three buffers in an atlas. The areas covered by the individual maps are adjusted dynamically depending on several factors. The most important factor is the height above the ground of the player. When the player is on or close to the ground the buffers are pulled in for better quality of the shadows at the expense of shorter range. When the player is flying far up in the air, the range becomes more important and the buffers are expanded. While all buffers are expanded, we usually keep the inner buffer fairly small even when flying high. This is to maintain a sharp shadow of the main character himself as he is skydiving or hang gliding. In the game, the shadow buffer parameters are handled automatically using logic that has been carefully tuned together with artists to provide the best use of the available buffers for most situations. In cut scenes the shadows are fully controlled by artists.

Each shadow map focuses on a point a distance in front of the player. A well-known problem with this is that as the camera view changes and shadow maps move around, ugly flickering artifacts occur along shadow edges. We solved this by aligning the shadow maps at the texel, taking out any frame-to-frame sub-texel offset. This solved the problem when looking around, but not when the player is traveling vertically and the shadow map ranges change. This was solved by using a threshold for applying a new range. If the new range differs by less than 7% from the current range, we simply don't change it. The 7% is a magic number derived from plain ocular inspection. There will still be a slight snap as you cross the threshold and ranges are adjusted, but having that happen once in a while is much more acceptable and discreet, unlike the really distracting large-scale flickering that was happening in every frame if one was up in the air.

By placing the three buffers into an atlas we only need to select the tile and sample the buffer once. The downside of this technique is that there is a noticeable seam where the switch to the next tile occurs, a phenomenon which can be observed in many recent games. One solution is to sample the next buffer as well and do a fade as one gets close to the edge. This eliminates the problem, but at the expense of requiring two samples. Although we support this technique in the engine, instead we most frequently dither, which allows us to sample only once. A pseudo-random number is derived from the screen position of the pixel and added to the fade value, which causes different pixels to swap to the next buffer at different times. It can look something like this:

```
float2 rc = { 0.782934f, -0.627817f };
float rnd = frac(dot(In.Position.xy, rc));
...
fade = saturate(fade + rnd);
```

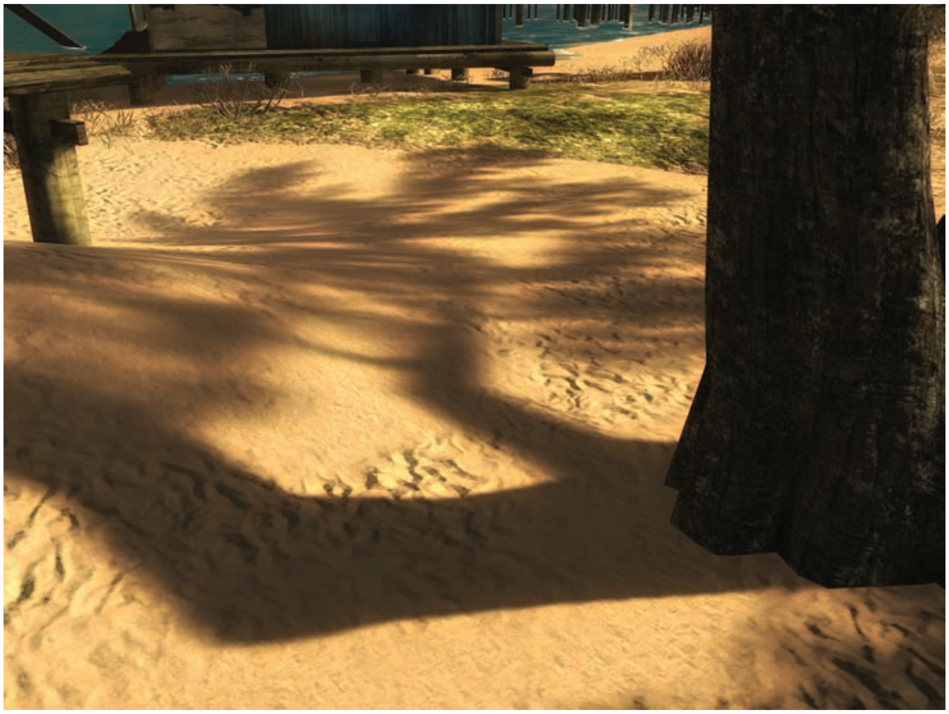

Figure 3.4. Soft shadows. Notice the sharp shadow at the base of the tree that gets progressively softer. Notice also that the foliage casts a very soft shadow.

The constants are chosen by entering a couple of random numbers, testing them in the game, repeating several times, and selecting the numbers that look best. The visual quality can differ greatly depending on the selected numbers.

Soft shadows. For the PC version of *Just Cause 2*, we offer a soft shadows option for those with powerful GPUs. While not physically accurate in any way, the algorithm does produce real soft shadows, rather than merely constant-radius blurred shadows as are used in many games (see Figure 3.4). In the first step, the algorithm (See Listing 3.1) searches the neighborhood in the shadow map for occluders. Samples that would cast a shadow counts as occluders. The average depth difference to the center sample among occluders is then used as a sampling radius in a second pass and multiple standard PCF samples are taken within this radius and averaged. In order to hide artifacts of a limited number of samples, the sampling pattern is rotated with a pseudo-random angle generated from the screen position.

```
// Setup rotation matrix
float3 rot0 = float3(rot.xy, shadow_coord.x);
float3 rot1 = float3(float2(-1, 1) * rot.yx, shadow_coord.y);

float z = shadow_coord.z * BlurFactor;

// Find average occluder distances.
// Only shadowing samples are taken into account.
[unroll] for (int i = 0; i < SHADOW_SAMPLES; i++)
{
    coord.x = dot(rot0, offsets[i]);
    coord.y = dot(rot1, offsets[i]);

    float depth = ShadowMap.Sample(ShadowDepthFilter, coord).r;

    de.x = saturate(z - depth * BlurFactor);
    de.y = (de.x > 0.0);
    dd += de;
}

// Compute blur radius
float radius = dd.x / dd.y + BlurBias;
rot0.xy *= radius;
rot1.xy *= radius;

// Sample shadow with radius
[unroll] for (int k = 0; k < SHADOW_SAMPLES; k++)
{
    coord.x = dot(rot0, offsets[k]);
    coord.y = dot(rot1, offsets[k]);

    shadow += ShadowMap.SampleCmpLevelZero(
        ShadowComparisonFilter, coord, shadow_coord.z).r;
}
```

Listing 3.1. Soft shadows shader stub.

3.2.3 Character Shadows

Characters can be quite problematic for a shadow mapping solution, in particular if you ever need to view characters up close. In *Just Cause 2* the main character is close to the camera at all times, and even more so in many cut scenes. To get proper shadow rendering on facial details you need very high resolution shadow maps, or you need to tighten up the inner shadow buffer very close around the face—in which case you probably need another shadow buffer just to maintain the same shadow quality elsewhere in the scene. We experimented with a separate character shadow buffer, but ultimately abandoned the idea due to additional memory requirement and implementation issues on some platforms. The

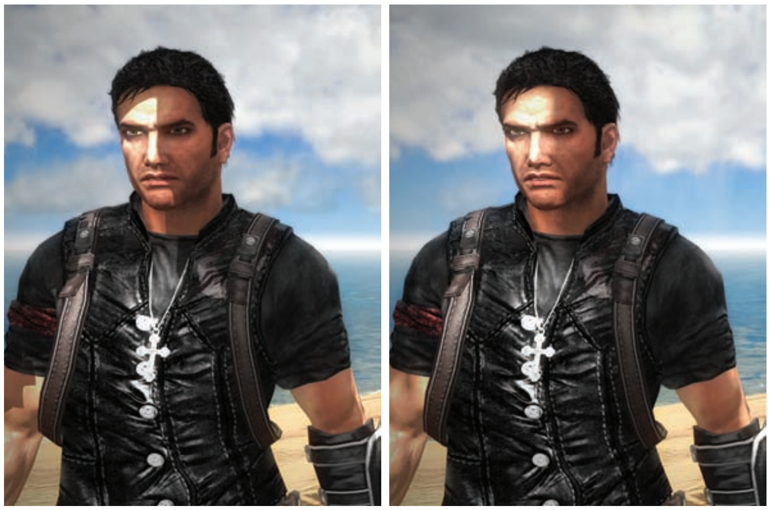

Figure 3.5. Linear interpolated depth (left); linear interpolated light bleed values (right).

separate character shadow buffer did not boost quality of character rendering as much as we wanted. Instead we settled for a *light bleed mapping technique* [Tatarchuk 05]. This is a basic extension of standard shadow mapping, but instead of an abrupt step it uses a soft falloff from light to shadow. Instead of the exponential falloff proposed by Tatarchuk we opted for a linear falloff, primarily because it was cheaper and sufficiently good. Unfortunately it does not work very well to just fetch one linear interpolated depth sample from the shadow map and use that in the light bleed computation. Instead we have to sample the four closest depth values, do the light bleed computation on the different samples, and then filter the result. Otherwise the result is very blocky and nasty artifacts show up on borders between shadow and light (see Figure 3.5).

On DX10.1 it is possible to use `Gather()` to get all samples in a single fetch. On the Xbox 360 where we already had to do four texture fetches due to lack of native PCF filtering, it came down to a few extra ALU operations.

3.2.4 Soft Particles

Soft particles vastly improve the quality of particle systems as it eliminates those nasty sharp edges where particles cut into the underlying geometry. Unfortunately, soft particles are also a good deal more expensive than regular particles. So the consoles use them only on snow smoke where the quality of plain particles was just not acceptable. On the PC version we made it a user option to turn it on for all kinds of particles.

The standard way to do soft particles is to fade away particles with the difference in depth between the stored depth value and the rasterized depth. This can be done either in linear space or with raw depth values, the latter being cheaper while the former is commonly viewed as "correct." However, for *Just Cause 2* using a linear distance difference did not cut it. The reason is that the view distance is huge and the particles vary wildly in scale. A burning car may look best if the fade is a decimeter or so, whereas clouds passing over mountains may need 50m or so. It is, of course, possible to tweak the fade range per particle system or use a reasonable heuristic based on the size of its bounding box. But it turns out that a reasonable fade close up does not necessarily look good at a distance. As you get farther away, and the particle system gets smaller on the screen, the fading range becomes smaller too, and its appearance ultimately approach that of the plain old depth buffer cut (see Figure 3.6).

We came up with a simple formula for the fade that worked well in pretty much every situation:

```
fade = saturate(poly_depth / sampled_depth * k - k);
```

This formula is perhaps more voodoo than science, but it is cheap, works well regardless of the scale of the particle system, and has a built-in adjustment of the fading distance as one moves away from the particle effect. This reason this works is because we use a reversed depth buffer, so Z at the far plane is 0.0 and 1.0 at the near plane. As objects get closer to the camera a smaller linear distance is needed to get the same ratio in Z values. The constant k is selected subjectively. For particle effects we used 8.0, for snow smoke 4.0, and for clouds we used 2.0.

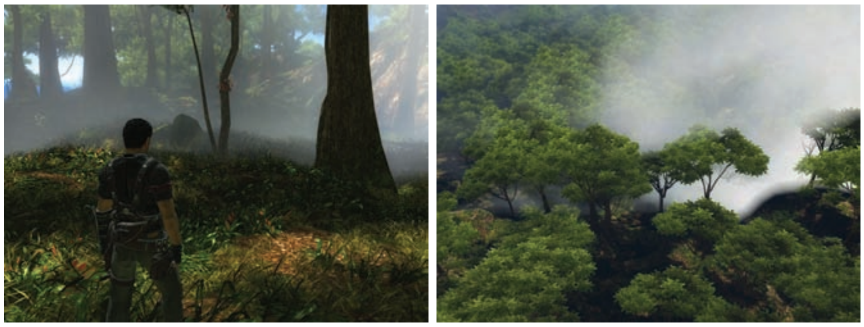

Figure 3.6. Linear fade of the cloud looks fine when viewed from ground level (left). The exact same fade looks very poor from the sky (right).

3.2.5 Ambient Occlusion

For ambient occlusion (AO) we use three different techniques: artist-generated AO, occlusion volumes, and SSAO [Kajalin 09]. The artist-generated ambient occlusion is for static models and consists of an AO channel in the material property texture. In addition, artists sometimes place ambient occlusion geometry at key spots. For dynamic objects we use occlusion volumes to cast an occlusion shadow on the underlying geometry, primarily the ground under characters and vehicles. SSAO is a user option for the PC version of *Just Cause 2*.

AO volumes. The idea behind this technique is that when objects approach other objects they occlude each other; e.g., a vehicle occludes the road beneath it. An approximate occlusion volume is placed at key locations on dynamic objects; e.g., a box under cars and an ellipsoid under each wheel and under the feet of characters. We support boxes and ellipsoids, but theoretically any convex shape could be used. The fragment shader samples the depth buffer and transforms the depth value into the local space of the occlusion volume; an occlusion value is computed based on the local position. Given that we use a forward renderer, the effect is applied after lighting, which is not entirely correct but works well enough for us.

SSAO. At the time of writing we use a fairly standard technique for SSAO, but more advanced options are currently being considered. Our technique is offered as a quality option for high-end GPUs on the PC and the focus is quality over performance. Since SSAO entered the project at a late stage in the project we did not want to make any radical changes to the engine, so we had to work with what we had around. Given that we have a forward renderer, we do not have scene normals available but instead generate them from the depth buffer.

Generating normals from the depth buffer is trickier than it may first seem. It is in fact impossible to make it robust in all cases; e.g., consider the case of a pixel-sized triangle generating a single depth value, whereas at least three depth values are needed to compute the normal. Using a few heuristics, one can usually generate “good enough” normals that eliminate the vast majority of the artifacts. The basic idea is to avoid crossing polygon edges when computing the normal. Depth buffer values are often considered nonlinear because of their distribution of precision across the depth range; however, they are linear in screen-space. This means that stepping the depth buffer we expect the pixel-to-pixel delta to be constant as long as the values belong to the same triangle. So horizontally we sample the two closest neighbors in both the left and right directions. Then we select the direction that has the smallest difference in delta between the three samples from center and to the left or right. The same is done vertically as well, giving us two tangent vectors, which crossed also gives us the normal. See Listing 3.2.

```
// Center sample
float center = Depth.Sample(Filter, In.TexCoord.xy).r;

// Horizontal and vertical neighbors
float x0 = Depth.Sample(Filter, In.TexCoord.xy, int2(-1,  0)).r;
float x1 = Depth.Sample(Filter, In.TexCoord.xy, int2( 1,  0)).r;
float y0 = Depth.Sample(Filter, In.TexCoord.xy, int2( 0,  1)).r;
float y1 = Depth.Sample(Filter, In.TexCoord.xy, int2( 0, -1)).r;

// Sample another step as well for edge detection
float ex0 = Depth.Sample(Filter, In.TexCoord, int2(-2,  0)).r;
float ex1 = Depth.Sample(Filter, In.TexCoord, int2( 2,  0)).r;
float ey0 = Depth.Sample(Filter, In.TexCoord, int2( 0,  2)).r;
float ey1 = Depth.Sample(Filter, In.TexCoord, int2( 0, -2)).r;

// Linear depths
float lin_depth = LinearizeDepth(center, DepthParams.xy);
float lin_depth_x0 = LinearizeDepth(x0, DepthParams.xy);
float lin_depth_x1 = LinearizeDepth(x1, DepthParams.xy);
float lin_depth_y0 = LinearizeDepth(y0, DepthParams.xy);
float lin_depth_y1 = LinearizeDepth(y1, DepthParams.xy);

// Local position (WorldPos - EyePosition)
float3 pos = In.Dir * lin_depth;
float3 pos_x0 = In.DirX0 * lin_depth_x0;
float3 pos_x1 = In.DirX1 * lin_depth_x1;
float3 pos_y0 = In.DirY0 * lin_depth_y0;
float3 pos_y1 = In.DirY1 * lin_depth_y1;

// Compute depth differences in screespace X and Y
float dx0 = 2.0f * x0 - center - ex0;
float dx1 = 2.0f * x1 - center - ex1;
float dy0 = 2.0f * y0 - center - ey0;
float dy1 = 2.0f * y1 - center - ey1;

// Select the direction that has the straightest
// slope and compute the tangent vectors
float3 tanX, tanY;
if (abs(dx0) < abs(dx1))
    tanX = pos - pos_x0;
else
    tanX = pos_x1 - pos;

if (abs(dy0) < abs(dy1))
    tanY = pos - pos_y0;
else
    tanY = pos_y1 - pos;

tanX = normalize(tanX);
tanY = normalize(tanY);
float3 normal = normalize(cross(tanX, tanY));
```

Listing 3.2. Deriving tangent space from depth buffer.

We found this technique to be virtually artifact-free without anti-aliasing. With multisampling enabled, some artifacts occurred. We use a resolved depth buffer for all effects that require depth input. To avoid artifacts elsewhere we resolve depth using the most distant value of the samples rather than the average. Unfortunately this breaks the pixel-to-pixel linearity for edge pixels, creating a few artifacts. It would have been desirable to use a specific sample instead, although that was not entirely artifact-free either. To be completely artifact free we would probably have to use a DirectX 10.1 sample frequency pixel shader, although that would have been massively more expensive.

We sample the depth buffer using Poisson distributed samples in a hemisphere. The computed tangent space is merged into the transformation matrix to make the inner loop of the shader as tight as possible.

3.2.6 The Jitter Bug: Dealing with Floating-Point Precision

Normally, 32-bit floats have sufficient precision for pretty much anything in graphics. But what about a game world that is over a thousand square kilometers, and yet game play takes place on a human scale? As it turns out, it can turn problematic in some cases. Close to the origin there are no problems, but as you approach the edges of the map, floating-point precision problems arise. Our coordinate system is in meters, ranging from -16,384 to 16,384 in x and z. Unfortunately, the lowest precision case where the magnitude of either the x- or z-coordinate (or both) is greater than 8,192 is true for 75% of the map, so it is essentially a universal issue. IEEE-754 has 23 mantissa bits, so the range (8192, 16384) has a precision of $8192 \ / \ 2^{23} = 1/1024$, which essentially gives us millimeter precision. That may not sound all that bad until you realize that all rounding errors from every floating-point operation add up, and millimeters turn into centimeters or even decimeters. For an overview of all the subtle ways floating-point can create problems, see [Ericson 07].

In practice what we observed was jittering on many levels. Instead of having trees swaying gently in the wind they would jump around in a jerky manner. Individual grass vertices would snap at different times causing very odd looking behavior. Shadow map texels would not line up very well with the scene geometry, and at the slightest move of the camera, the shadow could jump a decimeter in any random direction. We found jittering occurring both at vertex level and in animation. These phenomena were collectively referred to as the "jitter bug," a pun that still amuses us.

The key to maintaining floating-point precision is to avoid adding or subtracting numbers that vary greatly in magnitude. If you add two numbers together, where one is larger than the other by a factor of two, the smaller number will essentially see its least significant bit thrown away. If they differ by four times, two bits are lost, and so on. Computing $0.05+5000-5000$ will not result in 0.05 but in 0.0498, whereas $0.05+0.05-0.05$ comes back as 0.05 just fine. Multiplication and division generally behaves well regardless of scale of the operands though, $0.05 \times 5000.0/5000.0$ computes equally precise as $0.05 \times 0.05/0.05$.

In a traditional vertex pipeline, the input vertex goes through a world matrix, then a view matrix, and finally a projection matrix. Depending on what data we needed in the shader, we often did exactly the above. Sometimes view and projection were merged into a single matrix, but it was very common to first transform into world space because we needed that value in the shader. We then fed that result into a view-projection matrix. The view matrix generally consists of a rotation, scale, and translation. With a small object you could get values in the scale, and rotation part could return values of up to about 1.0. This then is added to the translation part, which could contain values in the 10,000+ range, thus stripping valuable bits out of the scale and rotation part. The view matrix has the same problem because it must bring a large world space position that down to the local space of the camera.

New transformation pipeline. To solve this problem, we rewrote the way we transformed vertices, then computed the transformation matrices to eliminate large translations. The model-view-projection matrix chain can be split into these submatrices:

$$[\text{Scale}][\text{Rotation}][\text{Translation}] \times [\text{Translation}][\text{Rotation}] \times [\text{Projection}]$$

This can be rewritten like this by merging the translation parts of the model and view matrices:

$$[\text{Scale}][\text{Rotation}] \times [\text{Translation}][\text{Translation}] \times [\text{Rotation}] \times [\text{Projection}]$$

The translation part of the model matrix is the world position of the model and the translation of the view matrix minus the eye position. So the translation matrix in the middle is just `world_pos - eye_pos`, or the position of the model relative to the camera if you will, which is much more manageable. If a world position in the shader is necessary, a world matrix will be provided. But instead of multiplying that through another matrix, we compute a merged model-view-projection matrix using the technique explained above, and transform the input vertex instead. This eliminates all vertex-level jittering. On the downside, more computations are necessary. Instead of using per-frame static view, projection, and view-projection matrices, we now have to compute the merged model-view-projection matrix per instance. It also results in having to set more shader constants.

Other floating-point precision issues. Similar solutions were applied to animation levels jittering and jittering in vertex skinning. We also had floating-point problems in the time dimension as well for systems that depend on any monotonically increasing value. One example is the water animation. For this reason we reset the water animation as soon as there was no visible water patch on the screen.

There were also precision issues with the depth buffer. We needed to have the near plane fairly close, but the far plane is very distant, which results in z-fighting in the distance. This was mostly solved by reversing the depth buffer and switching to `GREATER` depth test. This, together with a floating-point depth buffer, eliminated most of the problems. The floating-point buffer helps since its non-linear nature basically cancels the non-linear nature of the depth buffer value distribution. However, reversing the depth buffer helped significantly, even with a standard fixed-point depth buffer, because by reversing the depth buffer, the floating-point computations during transformations lost less precision. As a result, more precisely rasterized depth values are created.

3.3 Making it Consistent

3.3.1 Same-Same, but Different

Not all games studios spend significant time making the experience consistent across platforms. The result is that many games look very different depending on the machine, even when they are running essentially equivalent rendering code. At Avalanche Studios we have artists with very sensitive eyes who will cry blood at the slightest visual deviation from their carefully tweaked assets, so we have spent valuable time making sure our rendering code produces the same output on all platforms.

Cross-platform consistency. The sources of visual deviation vary. Many sources such as different output chips, different blending implementations, different handling of sRGB, different precision, etc. are hardware-related. But there are also software-related differences. Early in the development of *Just Cause 2*, each platform had its own implementation of many core rendering blocks. Much of this was a legacy from the organic growth of the original *Just Cause* code base. It became clear early on that having three different implementations of everything was a maintenance nightmare. A fix made on one platform was not always integrated to the other, features would be left unimplemented on a platform or two, and soon enough the different platforms were doing completely different things. In addition, this was hardly optimal from a productivity point of view.

Low-level API. The first thing we had to do to deal with these problems was to make a platform independent interface and use a single implementation of most stuff common to all platforms. Each platform has its own native graphics application programming interface (API): DirectX 10 for PC, DirectX 9 for our editor

```
union BlendState {
    uint32 Index;
    struct {
        int BlendEnable : 1;
        int SrcBlend    : 4;
        int DstBlend    : 4;
        // etc.
        ...
    };
};
```

Listing 3.3. Blend state.

and tools, a fancier version of DirectX 9 for Xbox360 and libgcm for PLAYSTATION 3. We designed a low-level graphics API that would sit on top of each platform's native graphics API. We wanted to keep the API as thin and simple as possible so that the compiler would be able to inline functions and execute the native API's code directly and as often as possible. For this reason much of the API became somewhat similar in style to LibGCM simply because it collapses many render states together that were hard to separate. One of the key elements of this design is that it is stateless; this is very important for being able to do multithreaded rendering using command buffers. Instead a context parameter is sent to each call. The context is just a pointer to a structure that is implementation dependent and hidden behind the API. The task of minimizing redundant state changes is placed on higher level code. For various practical reasons though, some states are sometimes mirrored in the context.

When we designed the API, we had not yet made the switch to DirectX 10 for the PC; otherwise it would have made sense to design it more like the state objects in DirectX 10 since they pack an even greater number of inseparable states together. However, since the PC is the most powerful platform it was decided to keep most of the original design to be friendly to the consoles. The DirectX 10 implementation had to be a good deal more stateful than the others. The other platforms, while having different style and capabilities, are similar enough that mapping the same API to each is relatively straightforward, with few performance concerns. With DirectX 10 being fundamentally different in many ways, mapping was not as easy. DX9 style render states had to be translated to DX10 state objects dynamically. In order to keep the overhead of this to a minimum, we packed states into a bit-field identifying the entire state of a certain state class; e.g., for blend states (see Listing 3.3).

Setting a render state simply boils down to updating the corresponding bits in the union. In the draw call the index value is compared to the previous index. Depending on the state class checked, this amounts to just a 32-bit or 64-bit integer comparison. If any of the states changed, the index value is used to look up and set the corresponding state object. The sampler state is bigger though, and there are 16. Fortunately, they are changed relatively rarely. So a special dirty bit was added to skip the entire check of them for most draw calls.

Shader constant management. Another major challenge was with handling shader constants. DirectX 9 hardware has a fixed set of constant registers whereas DirectX 10 has constant buffers. When used correctly, constant buffers can improve performance by reducing traffic between the CPU and GPU. When used incorrectly, the traffic can balloon up to massive amounts and reduce performance significantly. The latter is the common case for any naïve port where a single 256 register constant buffer is used, including our initial DirectX 10 implementation. Before optimization we typically uploaded 10–30MB of constant data every frame. This naturally turned the PCIe bus into the bottleneck. After optimization we typically uploaded 0.2–0.5MB constants per frame and observed substantially improved performance.

Coming up with a common interface for constants was not as straightforward as it initially seemed. It had to be fast on all platforms, safe and maintainable. The latter points cannot be emphasized enough. Littering the shader code with lots of `#ifdefs` or `mixing register()` and `packoffset()` declarations opens up for all kinds of nasty mismatches between platforms, which can create hard to track bugs or worse go undetected into the final product.

Our final solution is a hybrid between the register and constant buffer philosophies. On the shader side we have a few macros that set up constant buffers for us. It can look like this:

```
CB(0, PerFrameConsts , 0,  64);
CB(1, MaterialConsts , 64, 16);
CB(2, InstanceConsts , 80, 24);
```

After the expansion of the CB macro we get this on DirectX10:

```
cbuffer cb0 : register(b0) {
    float4 PerFrameConsts[64];
}
cbuffer cb1 : register(b1) {
    float4 MaterialConsts[16];
}
cbuffer cb2 : register(b2) {
    float4 InstanceConsts[24];
}
```

On other platforms we got this:

```
float4 PerFrameConsts[64] : register(c0);
```

```
float4 MaterialConsts[16] : register(c64);
float4 InstanceConsts[24] : register(c80);
```

Declaring shader variables are then done as static assignments from the provided arrays:

```
static float3 Offset   = InstanceConsts[0].xyz;
static float  Radius   = InstanceConsts[0].w;
static float3 Position = InstanceConsts[1].xyz;
static float4 Color    = InstanceConsts[2];
```

In the rendering code the constant buffer layout is specified with corresponding calls:

```
SetVertexConstantBufferSize(ctx, CB_0, 0,  64);
SetVertexConstantBufferSize(ctx, CB_1, 64, 16);
SetVertexConstantBufferSize(ctx, CB_2, 80, 24);
```

On DirectX 10 this selects the best fit constant buffer from a pool of pre-created buffers of various sizes. On the other platforms, it merely stores the offsets and sizes. Constants are then set by calls such as this:

```
SetVertexConstants(ctx, CB_2, 2, &Color, 1);
```

One great benefit of using this method is that we get a direct 1:1 mapping between the shader and rendering code on all platforms. We also eliminate the chance of any register mismatch in individual constants between platforms. If it is broken anywhere it is broken on all platforms. We can also place an `ASSERT()` in the `SetConstants()` calls to detect if we are writing outside any buffer, which helps catch errors.

For performance reasons it is not possible to do a partial constant buffer update in DirectX 10. So the `SetConstants()` calls store the constants into an array of `float4`, which essentially mirrors what would be in the constant registers on the other platforms. The actual constant buffer is later updated with the values in the array. An advantage of doing it this way is that any constant that was not updated after constant buffers changed will get the old value from the array. This way the DX10 path can maintain 100% compatibility with the other platforms even across constant buffer switches.

Most of our constant updates go through the above path, but we also support a more direct constant buffer interface for DirectX 10, where we can set a user supplied constant buffer to a slot instead of selecting one from the pool. This constant buffer can then be locked and filled manually where this provides a performance advantage. In some cases we also use pre-created `IMMUTABLE` constant buffers to eliminate the update all together.

3.3.2 Gamma

One of the decisions we made relatively early in development was to use linear lighting. The reasoning was simple: quality is better and transforming back and forth between gamma space and linear space comes for free on all our target hardware. Unfortunately, it turned out that hardware does not treat sRGB the same way. In hindsight, if we had known about what problems were ahead, we might have stuck to doing lighting in gamma space for this hardware generation.

The sRGB blending problem. In an unfortunate oversight, IHV's DirectX 9 level chips were designed to do render target sRGB transformation directly after the fragment shader. This works fine as long as blending is turned off. When blending is enabled, however, it occurs in sRGB space rather than linear space. The impact of this depends on the blending mode. For regular alpha translucency blend it is not much of a problem. For instance, if the incoming fragment is 0.3 and the render target has 0.7 in sRGB space and incoming alpha is 0.5 you get this:

$$(0.3^{1/2.2} \times (1 - 0.5) + 0.7^{1/2.2} \times 0.5)^{2.2} = 0.477$$

The result is not very far from the ideal linear result of 0.5. The story is quite different for additive blending. If the incoming fragment is 0.3 and the render target has 0.3 in sRGB space you get this:

$$(0.3^{1/2.2} + 0.3^{1/2.2})^{2.2} = 1.378$$

For alpha translucency blend, the non-linearities more or less cancel each other. For additive blending the error is instead amplified and you can get values as far off as in the example above. You do not expect 0.22 + 0.22 to saturate to full white.

Fortunately, the DirectX 10 specification is clear about blending happening in linear space, so for PC games targeting DirectX 10 there is consistent blending across hardware vendors. Unfortunately, there will be problems if one of the consoles has the problem, which was certainly an issue for *Just Cause 2*. For our online hunting title *theHunter* (PC only), this is also an issue because it uses DirectX 9, and all shader model 3 cards from this IHV are affected.

Dealing with this issue was troublesome. The blender is a small piece of fixed-function hardware with relatively few configurations and a minimal set of inputs. Also, one of the inputs—the destination color—is inaccessible from the fragment shader, which makes it impossible to correctly compensate for the error there. After a number of code and shader tweaks and careful side-by-side comparisons, we came up with a reasonable compromise that satisfied the artists. The first thing we did was move the multiplication of source color by alpha, from the blender into the shader, which at least made that part of the equation equal across platforms.

It also freed up one blender input. This turned out to be useful for additive particles where we could now apply the classic *soft add* blend mode. Soft add is like regular additive blending, except that the source color is multiplied with `INV_DSTCOLOR` instead of `ONE`. As the destination buffer gets brighter, the incoming values are dampened, which means more things can be added together before it saturates to full white. Meanwhile, it behaves much like a regular add when the destination is dark. Using this technique on additive particles helped control the tendency of incorrect adds to shoot through the roof too quickly, and it made the particles look more consistent across platforms.

Since our particles systems had been tuned by artist on the incorrect blending platform, we had to adjust the correctly behaving blends to mimic the incorrect behavior. While these adjustments were not exact, we did a number of tweaks through ocular inspection of our common effects. We ended up squaring the alpha and multiplying by 2.35 in the fragment shader for additive particles. While there are still visible differences in some effects, most looked similar enough to be acceptable. For alpha translucency we ended up raising alpha to the power of 1.18. This roughly compensated for the small difference mentioned earlier. A similar problem in our sun halo was dealt with by simply changing a constant to bring the intensity to the same range.

Other gamma issues. While one console had a problem with blending, we found that the other console was not very accurate when sampling sRGB textures. A simple piece-wise linear function was used, which was obvious when viewing a gradient. As a result, the rendering on this particular console was more prone to banding and other artifacts. In addition, we also had to add a compensation curve in one of our post-effects shaders to reduce the differences. Furthermore, the output to the monitor was not entirely correct. Given identical values in the front-buffer, different results were sent over the HDMI cable on the different platforms. Fortunately, we could solve this problem by loading a custom gamma compensation curve that eliminated the differences.

Lessons learned. One of our mistakes was letting artists tune particle effects on a single platform for a considerable amount of time. With thousands of effects, compensations on the artists' side to even out differences was not an option. Had this problem been caught earlier, there might have been a better solution.

Another blunder was not immediately fixing the sRGB code for our DirectX 9 path when first implementing linear lighting. Instead, there was a quick fix in the shaders. Rather than using the more correct power of 2.2, we used a power of 2.0, which was close enough for a programmer's eye and of course substantially faster. At this point DirectX 9 was only used internally for our tools so this did not seem important. Later when it work on *theHunter* started, it was decided to use DirectX 9, artists carefully tweaked lighting and assets for this code. When the code was later fixed to use actual sRGB texture sampling and sRGB writes, artists complained that it did not look the same anymore. Given the substantial performance gain we observed, it was not an option to leave the old shader compensation code. Instead, artists had to go over all the settings again and tweak for the new shader environment.

3.4 Making it Fast

3.4.1 Cloud Rendering Optimization

Clouds are an important component of the game world. The clouds in *Just Cause 2* consist of a layer of cirrus clouds with a cumulus cloud layer below it. The cirrus clouds are implemented as planar textured surfaces and a single draw call is used to render all cirrus clouds. Each cloud chooses its texture from an atlas of all cirrus cloud types. The cumulus clouds are implemented as view-aligned surfaces. As with the cirrus clouds, everything is rendered with a single draw call and individual cloud images are selected from a texture atlas.

When the player travels near the ground, clouds generally do not cause a performance issue. However, *Just Cause 2* is not a game where the player necessarily stays close to the ground: a player may very well decide to take a plane in the clouds. In fact, it is required to be far up in the sky to complete some missions. Cirrus clouds are usually not problematic, but cumulus clouds could easily take four to five milliseconds when the player is in the sky, and we got many layers of overdraw covering much of the screen. The clouds are rendered with a trivial fragment shader, so there were not many optimization opportunities there. Instead, the bottleneck was in the ROPs. Modern GPUs are increasingly getting more powerful in terms of ALU and texturing power, but ROP power has been lagging. So while the consoles were suffering from this problem, PC GPUs were not doing much better and the ROPs were in fact an even bigger bottleneck, relatively speaking.

The clouds were originally rendered as simple quads. However, we noted that many of the cloud textures contained a substantial amount of empty space where alpha was zero. So a lot of the rendered fragments did not contribute to the final results. To cut down on the fill requirements we decided to trim the clouds. We stuck to four vertices per cloud in order to not increase memory requirements, but depending on which atlas tile was used, an optimized quad was selected that closely enclosed the cloud (see Figure 3.7). The result of this optimization surpassed

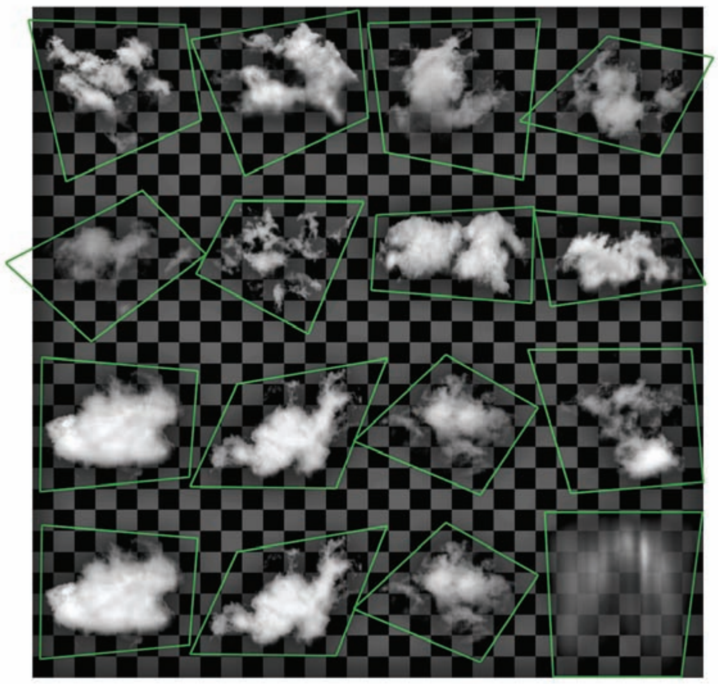

Figure 3.7. Cloud atlas with optimized quads.

our expectations, as the rendering time was now down to around two to three milliseconds at no visual loss.

To reduce flickering while players fly through clouds, the clouds were faded away as they come closer to the camera. Consequently, some of the most expensive clouds also contributed the least to the final result as they were close to the camera and covered substantial screen-space but were essentially faded away. Instead of just checking the vertex alpha to be greater than zero, we used a small but positive alpha threshold to allow us to reject the worst offenders. With a very conservative threshold we could still shave off a substantial amount of the remaining rendering time and landed at about 1.5 milliseconds, or nearly three times faster the speed of original implementation.

3.4.2 Particle Trimming

Given the success of our cloud optimizations, we decided to apply the same trimming technique to our particle systems. There are many objects that can blow up in *Just Cause 2*, and when an enemy grenade lands at your feet you do not want a low frame-rate added to your problems.

But there was just one problem: whereas the cloud atlas consists of 16 different clouds, the particle systems are composed of dozens of different textures, containing everything from a single particle image to 64 atlas tiles each. For the clouds we had tweaked the quad vertices manually by exposing some variables in the in-game debug menu. This was good enough for a small set of textures and a task that had to be done only once. But now we faced a task that was an order of magnitude greater. After first attempting to go along the manual tweaking route, we quickly discovered not only that this was inefficient, but also that four vertices were not enough since the optimal quad would often cut into particles in neighboring atlas tiles. Furthermore, with more vertices it was not as clear, just from looking at a particle texture, which polygon would be the most optimal.

In order to deal with this problem we came up with an algorithm for automating the process, and a tool was written that, given an input texture and an alpha threshold value, computed the smallest enclosing convex polygon of the desired number of vertices. At the core of the algorithm was a convex hull. Edge pixels of the particle are detected and the convex hull was updated iteratively. In the end we had a convex hull, typically consisting of a few dozen vertices. All permutations of the hull edges were then checked, and the one resulting in the smallest area while fully enclosing the particle was selected. This step was done using a brute force traversal over all permutations of the edges. This was usually not a problem, but the running time can quickly shoot through the roof if the number of vertices gets large. For this reason the hull is initially reduced to at most 50 edges by eliminating edges one by one, in each step removing the edge that results in the smallest expansion of the hull. With this tool, the task that previously took one programmer a day was now done instantly and more accurately, leaving us to cut and paste the result. The tool is open source and available online [Persson 09].

3.4.3 Memory Optimizations

Getting the most out of the available memory is very important, in particular on the consoles. One of the most important things we did to keep memory consumption in control was to allocate budgets for various subsystems. When the budget was exceeded the requested resources were replaced with dummy resources; e.g., a texture going over the budget was replaced with a green texture. When there were too many textures used in a particular location, green textures showed up in the game, giving visual feedback to level designers that they needed to rework the area to keep memory consumption in check.

Just Cause 2 needed a number of render targets. In addition to the back buffer, there is the multi-sampled color and depth buffer, as well as a non-multi-sampled depth buffer that the GUI requires, a shadow buffer, a reflection buffer, two half- and quarter-resolution temporaries for the post-effects, non-multi-sampled resolved back buffer, depth buffer, and a velocity buffer. In order to reduce the memory footprint of all this, we first did the obvious optimization—namely, reducing the shadow buffer precision from 24-bit to 16-bit. At 1024×3072, our shadow buffer required 12MB of memory. Changing this to 16-bit buffers got the cost down to 6MB. As it turned out, we did not need any more precision than that. The shadow bias value we used was a good deal greater than the contribution of the lower 8-bits. We did encounter a problem in that one of the consoles did not support 16-bit depth buffers natively. This was solved with an additional conversion pass using raw memory reads and writes to convert from a swizzled 32-bit depth buffer to the swizzled 16-bit format.

For the post-effects we used two copies of the temporary render targets because we were ping ponging between the render targets for some filters. On one console, however, we realized that this was not necessary due to its special memory architecture: in practice it was already ping ponging between EDRAM and the video memory, so we removed the redundant copy there. We also realized that several of the render targets had no overlap in time. For instance, the time from we rendered to the shadow buffer to the time we last used its content never overlapped with the life time of the post-effect temporaries. On the consoles where we had direct control over video memory, we took advantage of this fact to reuse some of the memory of render targets by placing other temporally non-overlapping render targets on top of it. The shadow buffer, which is the largest texture, gobbled up several of the smaller render targets. By carefully shuffling render targets around we were able to save about 5MB of render target space.

Another memory optimization we did was to replace 32-bit floats in vertex buffers with 16-bit fixed points together with a scale value for unpacking. One problem with this was that positions generally are three values, but there are no `SHORT3N` attribute types: only `2N` or `4N`. Instead of wasting space for a fourth value, we found that we could use a `SHORT4N` value but set the stride of the vertex to cut those last two bytes out of the vertex. For instance, if only `POSITION` is used, we would declare it as `SHORT4N` but use six bytes as our stride, as opposed to eight bytes. This trick worked on all platforms.

One of the more obvious ways to reduce memory consumption was to use compressed textures, which we did for nearly everything. A less obvious trick is to pack several textures into one compressed texture. Many materials are fairly uniformly colored, so you can get a long way by using just a luminance texture together with vertex colors. An artist can optionally place three different luminance textures into the channels of a DXT1 texture, which works well for generic textures with little uniquely identifiable detail, such as plain concrete. This essentially gives us textures at as low as 1.33bpp. For the texture fetch we configure the sampler to do the swizzle automatically so we can use the same shader for channel textures and regular full color textures at no extra cost. This is supported by most hardware, but unfortunately on the PC there is no such functionality in DirectX (OpenGL has the `GL_EXT_texture_` swizzle extension though), so we had to use a few ALU operations to sort out the swizzle.

3.5 Conclusion and Future Work

This article has presented a number of techniques we implemented in the Avalanche Engine for *Just Cause 2*. Various issues we encountered have been discussed as well as the solutions we came up with. We have also covered miscellaneous optimizations and memory savings.

At the time of this writing *Just Cause 2* is in beta stage and we are in bug stomping mode. Whether any significant new features will go into *Just Cause 2* before the release is anyone's guess, but looking forward we have a number of ideas of where to go next. With DirectX 11 around the corner there are many interesting we would like to use. The compute shader is something that would come to great use for our post effects, both as an optimization for existing post effect and an enabler for future techniques. Using the tessellator for our terrain seems like an obvious improvement, although we have concluded that it would be non-trivial. There are a number of other code driven systems that could use it with relatively small amount of effort, such as grass and water waves.

We still have not tapped the consoles on all the available power either. We do multi-thread our rendering on the consoles, and have good utilization of the available CPU cores (including SPUs), but there is still work to be done on threading all our systems. On the PC we utilize the available cores pretty well for general tasks, but we still do not multi-thread rendering. This is something we would like to do with DirectX 11 and its deferred contexts.

Over the years, deferred shading has come up numerous times. Still we stuck to forward rendering until the end. Chances are we will opt for some kind of deferred approach in future projects though. The main motivation would be to improve performance and flexibility for dynamic lights.

3.6 Acknowledgments

I would like to thank my fellow colleagues at Avalanche for the support in writing this article and for all their hard work in bringing the Avalanche Engine to world class quality as well as their performance and the intellectually stimulating environment they have created here at Avalanche Studios. Without the bright minds of all the people here, none of this would have been possible.

Our thoughts also go out to all former Avalanche employees that could not stay with us to the conclusion of this project. Thanks for the great work you did during your time with us. All of our continued friendship is deeply appreciated. Keep up the good work where you are now!

Special thanks to Viktor Blomberg for his endless flow of bright ideas; may the magic shader constants be with you! Alvar Jansson for all the Gothenburgian puns; work would not be half as fun without them! Christian Nilsendahl for the friendly leadership. Christian Murray for keeping a keen eye on performance and slapping us all once in a while when we submitted substandard code. John Fuller for proof reading. Christofer Sundberg for his ability to turn the very depths of hell into something ultimately positive. Ingela Hellqvist and Linda Bäcklund for all the candy.

Bibliography

[Trebilco 09] amian Trebilco. "Light-Indexed Deferred Rendering." In *ShaderX7*, edited by Wolfgang Engel. pp. 243–256. Boston: Charles River Media, 2008.

[Engel 06] olfgang Engel. "Cascaded Shadow Maps." In *ShaderX7*, edited by Wolfgang Engel. pp. 197–206. Boston: Charles River Media, 2008.

[Zhang 09] an Zhang, Alexander Zaprjagaev, and Allan Bentham. "Practical Cascaded Shadow Maps." In *ShaderX7*, edited by Wolfgang Engel. pp. 305–329. Boston: Charles River Media, 2008.

[Tatarchuk 05] atalya Tatarchuk. "Advances in Real-Time Skin Rendering." Available at http://developer.amd.com/media/gpu_assets/D3DTutorial05_Real-Time_Skin_Rendering.pdf, 2009.

[Kajalin 09] ladimir Kajalin. "Screen-Space Ambient Occlusion." In *ShaderX7*, edited by Wolfgang Engel. pp. 413–424. Boston: Charles River Media, 2008.

[Ericson 07] hrister Ericson. "*Physics for Games Programmers: Numerical Robustness (for Geometric Calculations).*" Available at http://realtimecollisiondetection.net, 2009.

[Persson 09] mil Persson. "Particle Trimmer." Available at http://www.humus.name, 2009.