2.6 Graphics Techniques in *Crackdown*

HUGH MALAN, REALTIME WORLDS

INTRODUCTION

Crackdown was a four-year-long open-world action-adventure game project, initially developed for the original Xbox. The project was shifted to the Xbox 360 halfway through.

Since the rendering time budget was allotted, finding the render time to allow new features had to be done by optimizing the existing code for time or memory. Clutter, vehicle reflections, water reflections, car drop-shadows, the parallax window shader, and many lighting tweaks were added in this way.

As a result of the hard render time and memory restrictions, we often had to look for solutions in places that probably would not be considered with a larger budget. Our solutions for clutter and vehicle reflections are examples of bottomdollar technology that worked unexpectedly well.

The techniques covered in this article are the sky, clutter, outlines, deferred lighting, and vehicle reflections.

SKY

Figures 2.6.1 and 2.6.2 are in-game captures of *Crackdown's* sky, demonstrating the features of the shader.

FIGURE 2.6.1 Daytime sky.

FIGURE 2.6.2 Sky at dusk

The shader provides self-shadowing and rim lighting, and the clouds animate slowly over time.

The sky uses the atmospheric scattering approximation described by [Neil05] for sky color; our only addition is the animated cloud layer.

The cloud animation and lighting is effectively a 2D effect. It is controlled by three texture layers; each holds a tilable pattern of cloud density. The maps hold the density and the two partial derivatives of that density, packed into the RGB components.[1] The maps are sampled and the densities summed to find the final cloud density (and its partial derivatives) for the current pixel. The three texture layers are not all comparable: One contains large-scale details

(coarse map); the other two contain small-scale details (detail maps). For *Crackdown*, the two detail maps used the same texture as a space-saving measure. It was sampled twice, once at position *XY* and the second time for location *YX,* to effectively rotate and flip the map to try to hide the fact it had been used twice.

The maps are scrolled slowly in different directions and with different speeds to produce the effect of clouds forming and evaporating. To make the movement of the cloud border more complex, the resulting density was scaled and offset, so 0.2 was remapped to 0 (1 was left unchanged). This is equivalent to allowing slightly negative values in the density maps. The change produces large areas with density 0. Without it, the underlying cross-scroll effect is obvious. This change adds a single instruction (`mad_sat`) and is intended to make the effect a little more difficult to see.

The cloud lighting is a function of cloud density and derivative. The lighting function is very simple, but bears a faint resemblance to a simple physical model of light scattering in a cloud. It is described in [Rost06].

The most important features of cloud lighting that we can simulate easily in the pixel shader are that:

- Wispy clouds are completely brightly lit.
- Dense clouds are brightly lit on the side from which the light is coming; they are dark on the other side.

In the shader, the final color and opacity are computed as follows.

- Opacity is simply proportional to cloud density.
- Color is more complex. It begins with the calculation of the large-scale cloud brightness, implementing the two cloud lighting bullet points above:

$$saturate\ (base\_brightness + dot(coarse\_surface\_normal$$

$$light\_direction) \times cloud\_density$$

The base cloud color is quite bright by default (*base_brightness* is roughly 0.8).

If *cloud_density* is low, then the result will be bright. Only if the surface normal of the coarse cloud map points away from the light and the cloud is dense will the result be dark. This emulates self-shadowing and rim lighting.

The cloud lighting in *Crackdown* is based on this equation, but has been extended with artist-controllable colors. The lit cloud color, shadowed cloud color, detail color change, and other such colors can all be set separately, and are animated as part of the day-night cycle. Also, the partial derivatives of the sum of all three layers (with a disproportionate contribution from the detail layers) is used to pick out the small cloud features.

Implementation Notes

Since the effect is 2D, it would be easy to render it by applying it to a plane at a constant height, but this looks completely wrong. Since the world is a sphere, a layer of clouds will also be a sphere. This may sound academic, but there is a dramatic difference in appearance near the horizon, and a cloud plane looks noticeably wrong.

Since a spherical surface patch is used instead of a plane, then the tangent-space light direction will not be constant. The light direction has to be transformed into tangent space in the same way that a light vector might be transformed into tangent space for normal-mapped lighting.

The UV offset of the cloud layers was also based on the camera position, so the clouds scroll past overhead. This was set up so it was noticeable but not attention-grabbing at top speeds.

The sky shading (Sean O'Neil's method) gave excellent realistic results, but not the dramatic "graphic novel" look. Setting up the sky parameters so sunrise and sunset were fiery required a dense atmosphere, which made midday dull and reddish. The solution was to animate the atmosphere parameters during the day-night cycle, providing both the dramatic sunrise and the deep blue skies at midday. This was complicated by the fact that some of the parameters changed by several orders of magnitude between sunrise and midday, and so the interpolation between keyframes for those parameters had to be done in log space. Even the sun direction had to be keyframed to avoid unsightly results during the morning and afternoon. In the final version, sunrise and sunset last disproportionately long for maximum drama, and the transition to the midday blue sky happens relatively quickly.

Lastly, the color tone of distant buildings needed to match up with the sky at the horizon. The full calculation for the sky shader was too expensive to be used even just for the distant buildings, so a much simpler color blend was used for them based on distance and height. The sky shader was extended with this simple color blend so very distant buildings were tinted with the same color as the sky near them.

Clutter was added near the end of *Crackdown's* development, and is responsible for adding a great deal of detail to the scene. Geometric detail provides city features down to roughly a few feet, and the detail maps provide features a few inches in size. Conceptually, the goal of clutter was to provide interesting features at scales in between. As can be seen from the screenshots in Figures 6.2.3 and 6.2.4, the city looks simple and much less interesting without clutter.

FIGURE 2.6.3 Clutter off

FIGURE 2.6.4 Clutter on.

Since the aim was to provide a large variety of objects, and the budgets for block size were set, the per-instance size of each piece of clutter had to be as small as possible, and the render cost had to be minimized.

Each clutter instance had to contain position, orientation, and clutter type; to minimize render cost we couldn't afford a separate draw call for each kind of clutter, so some kind of instanced geometry method was required. [Dudash04] describes the standard method for implementing instanced geometry.

The instanced geometry was implemented in a slightly unusual way: Each type of clutter was a single quad, and the vertex position data for each of the different types of clutter was stored in the shader constants. The positions of the vertices were packed into the shader constants with three bytes to a float, so each four-component vector held the geometry for a single clutter type. The UVs were not user-editable. The texture for each clutter type was 64×64 and packed into a 1024×2048 map (see Figure 2.6.5), so the vertex program computes UVs for the corners of the region of the texture map belonging to this kind of clutter.

The placement, orientation, and type of each clutter instance were stored in a vertex buffer; each instance required 8 bytes (the format is described below).

At render time, the only vertex buffer read is the one containing the list of clutter instance data. Indexed quads are drawn, with one quad per clutter instance. The index buffer is a dummy; it contains 0,1,2,3,..., and needs to be four times larger than the maximum number of possible clutter pieces in a block (a dummy index buffer was the simplest solution on the XBox360, where `SetStreamSourceFreq()` is not available).

For index *k*, the vertex program reads entry *k/4* from the vertex buffer to find the position, orientation, and type of the current piece of clutter. The value of *k* module 4 specifies which corner of the quad is being processed.

The 8 bytes for each instance are interpreted as follows.

The first DWORD specifies the position of this clutter instance. It is a 3D vector in 11-11-10 format, with each coordinate being a fraction within the bounding box of the relevant block, which is usually about 150–200 meters across.

This was not quite accurate enough. The 11-bit components were used for the horizontal coordinates, but a 1-bit change meant a movement of about 7–10 cm. The artists placing the

clutter had been having suspicions that some of the clutter was getting lost; it turned out that half of it was hiding behind walls and under floors because the position had been rounded the wrong way. The precision problems were fixed by carefully rounding the coordinates of position the appropriate way to bring it into view and adding some depth bias to avoid z-fighting problems.

FIGURE 2.6.5 Clutter texture map used in *Crackdown*.

The second DWORD is a 4D vector in 8-8-8-8 format, and provides orientation and clutter type. The orientation was defined by the first three 8-bit values (call them *a*, *b*, and *c*) with the fourth value *d* controlling clutter type.

The three basis vectors were set up as follows. The values *a* and *b* are scaled to the range [−1, +1] and used to form the vector (*a*, 1, *b*). It is normalized, and the result is the vector *q* (in *Crackdown's* coordinate system, *Y* is up).

The rotation around that vector is controlled by *c*. It is scaled to the range (0, 2π), and the vector (cos *c*, 0, sin *c*) is computed. This vector is guaranteed not to be parallel to *q*, so the cross-product with *q* can be taken and normalized without risks of degeneracy. The result is vector *p*. The third vector *r* is the cross-product of *p*, and *q*.*p*, *q*, and *r* form an orthonormal basis, which is used to transform the raw vertex positions of the clutter type to the correct orientation (*q* is local up, *p* is forward, and *r* is right expressed in world space).

This setup allowed a maximum rotation of 45 degrees from horizontal, which was enough to accurately align clutter to the streets and most slopes in *Crackdown*. Posters, graffiti, and stains on walls were accomplished by modeling that sort of clutter vertically; the 45-degree restriction on rotation meant that a poster cannot be rotated to lie on the ground.

For clutter type *c*, the four-component vector in shader constant *c* contains the encoded vertex position. Depending on the value of *k* module 4 the X, Y, Z, or W component of that vector is read. The resulting float is unpacked to give the clutter vertex position with frac(*v*, *v* × 256.0, v × 65536.0). The divisor of 256 (2^8) was chosen because it yields 24 bits of precision (3 × 8), which is only slightly more than the 23-bit mantissa, so at most one bit of precision is expected to be lost from one of the components (it would be possible to squeeze a few more bits of precision out by extracting data from the exponent too, but this was already more precision than we required).

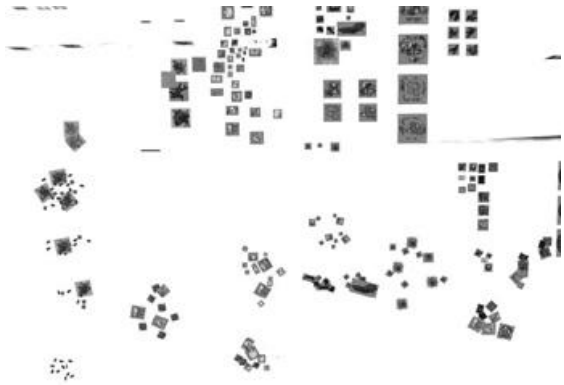
The unpacked vector is then transformed into the space defined by the three basis vectors *p*, *q*, and *r* and offset to the correct position in world space.

This technology might sound quite basic and inflexible, but since the vertices of each kind of clutter can be placed without restriction, in principle it is possible to model anything by breaking it down into quads. The tools were set up to allow collections of quads to form larger structures that would be placed together as a group, and to automatically break down quads using larger textures to 32×32 pieces. The artists then built a huge variety of simple items such as crushed cups, coffee cup lids, food wrappers, leftover kebabs, faint stains, vomit, bloodstains, graffiti, bullet holes, and so on (see Figure 2.6.6).

The individual pieces were collected into "clumps." For instance, a crushed cup or bottle on a stain, or food wrappers with assorted stains and leftover food, or two crossed quads forming a small weed (see Figure 2.6.7).

FIGURE 2.6.6 Sample clutter images

FIGURE 2.6.7 Clutter clumps: groupings of clutter pieces.



The artists then placed the clumps throughout the city. MaxScript procedures were built to spray clutter of a certain category into the world (such as party remains, firefights, greenery, stains, posters). This allowed the artists to quickly place large amounts of clutter in a category appropriate to the environment. So near a bin there might be a lot of waste paper, plastic bags, and other such items crushed into the pavement. Near Guerra's nightclub there are flyers dropped on the ground. In some of the back alleys are shell casings, bloodstains, graffiti, and bullet holes.

This feature is similar in some ways to texture bombing, in that it allows decals to be scattered across the surface. However, since real geometry is used, the decals can stand out from the surface (to provide weeds, bushes, and boxes). It is relatively easy to set up tools to allow the artists to place and edit the clutter, and there are no hard limits on the number of overlapping items that can be placed at a particular point. The downsides stem from the use of a second render pass; no lighting calculations can be shared, so some arithmetic and logic unit (ALU) has to be repeated. Also, it becomes difficult to match clutter items precisely to texture features.

The use of geometry opens up a lot of interesting possibilities that we did not have time to explore for *Crackdown*. The memory and render time it used were minimal. Most blocks used only a few hundred clutter items. If the budget was lifted to allow 10 or 100 times more items, the clutter density might have been high enough for it to embellish the surface texture, rather than just provide decals. One simple idea is to add non-periodic features to surfaces, such as a scattering of discolored or protruding bricks in a wall, or using a parallax shader to provide the appearance of missing or damaged bricks in the wall without paying the price for the parallax shader for the entire surface. Edge fins may be able to provide the appearance of jagged, non-periodic edges on man-made constructions such as concrete structures or brick walls.

Clutter ended up being a very successful feature. It added a great deal of graphical detail to the game for a low render and memory cost, and relatively little artist time. Of all the features described here, clutter probably improved the perceived quality of *Crackdown's* graphics the most.

OUTLINES

Outlines were implemented as a post-process effect, using the per-pixel normal and depth

buffer (see [Decaudin96] for a description of the use of per-pixel normals and the depth buffer to detect edges and creases to create the outlines). Figure 2.6.8 shows the outline effect.

Figures 2.6.9 to 2.6.11 show the source per-pixel normal texture and the effect on the final image.

FIGURE 2.6.8 Outline opacity.

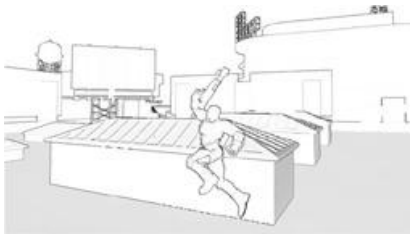


FIGURE 2.6.9 Per-pixel normals.

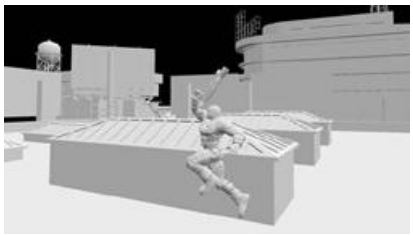


FIGURE 2.6.10 Image without outlines.

FIGURE 2.6.11 Final composite.

To achieve the desired “graphic novel” look, we wanted very thick anti-aliased outlines, but since render performance was a major concern for project management, if the render time was anything more than a handful of milliseconds, we were in danger of it being cut.

Thicker or smoother outlines result in a larger sample area and more texture reads, but render time is roughly proportional to the number of texture reads. We went through myriads of different variations; in the end we used the resolved result from the 2×MSAA (multisampling anti-aliasing) normal buffer and the depth buffer, reading five samples in a cross-pattern.

The normal texture is read using bilinear filtering. The samples are offset by half a texel, so four texels are sampled and averaged with a single read. This means a 20-pixel neighborhood in the normal texture is used to calculate the outlines.

The depth texture cannot be sampled with the same finesse since the format is floating

point. The same five-point sample pattern is used, though this means that some texels are skipped and there is the danger of artifacts due to undersampling.

The per-pixel normals and depths are averaged, and the darkness of the outline depends on the difference between the averaged result and the values at the current pixel.

For a planar surface the per-pixel normal will be constant, so the average normal will match the current pixel. The depth will similarly match, so planar surfaces will not be darkened. Pixels near the edge of an object will have some samples with a depth value substantially different from others, so the averaged depth will be quite different from the current pixel depth and an outline will form. (The per-pixel normal may be constant. Imagine looking down at the edge of a table onto the floor. All nearby pixels have normals pointing "u.")

Pixels near an internal crease on an object may have quite similar depth values, but abruptly different normals, so an outline will appear there too. The contribution of these two different effects had to be carefully balanced to provide consistent outlines.

Since the source per-pixel normal texture used is anti-aliased, it helps provide the (somewhat) anti-aliased edge to the outlines.

Since the outlines only appear for objects included in the depth-normal pass, objects that suddenly appear in or disappear from the depth-normal pass will have their outlines appear or disappear. This is most noticeable for the way the player character and props fade away when they get too close to the camera. The outline change is masked by dropping the opacity from 100% to 60% in one frame (so the object becomes quite transparent at the same time as the outlines disappear) and fading down to 0% smoothly after that.

The outlines fade away in the distance, and the fade depth was set up so the outlines would completely disappear before the first environment levels of detail (LODs) transition happens.

One other problem was due to tiling. A full 1280×720 2×MSAA image cannot fit into EDRAM on the Xenon. It has to be rendered in two sections. The first tile fills in the top half of the normal and depth buffers, and the second tile completes them. Since the outlines sample three pixels in each direction around a pixel, there were problems when rendering the bottom three rows of the first tile because the outline shader would be sampling off the bottom of that tile into uninitialized or stale areas of the normal and depth buffer. The solution we used was to overlap the two tiles by four pixels: the second tile redraws the bottom four rows of the first tile.

Outlines were difficult and time-consuming to develop, and required a great deal of tweaking of many other graphics components to avoid or reduce the quality problems. The final version required little render time and provided the thick anti-aliased outlines around everything, which was the goal.

Deferred Rendering

Deferred lighting has the greatest advantage over standard lighting when there are numerous localized lights, and the frequently changing set of relevant objects that each interacts with ([Hargreaves04] has an introduction covering how deferred lighting can be implemented on the GPU).

The goal of looking out from the top of a building at night and seeing myriads of streetlights (and to a lesser extent, car headlights) was a strong argument for the use of deferred lighting. The fact that the per-pixel normal buffer and depth buffer would also be used by the outlines and no other screen-sized buffers would be required made the proposition a lot more palatable. The standard problem with deferred lighting and other deferred effects is that they do not benefit from MSAA. In *Crackdown's* case, the lighting would only affect a fraction of the pixels onscreen, so only a corresponding fraction of the edges would suffer, and the outlines would help cover up artifacts on those edges too.

Since there were so many differences between standard deferred lighting and the approach we used, we gave the name "afterlights" to the deferred lights in *Crackdown* to avoid confusion. Figures 2.6.12 and 2.6.13 demonstrate the effects provided by the afterlights.

The deferred lighting used in *Crackdown* is relatively simple. As mentioned previously, the only per-pixel quantities stored are the normal and depth resolved out after the depth-normal prepass. Although surface color is not stored, its effect is simulated in the following way.

FIGURE 2.6.12 Deferred effects disabled.

The afterlights are applied after the main color pass, and while rendering opaque surfaces in the main color pass the alpha channel is set to a value proportional to the brightness of the surface color. When the afterlights are blended into the frame buffer, they are modulated by the destination alpha, so their effect varies depending on surface color. Lights are usually modulated by surface color by multiplying each component of the light by surface color.

Here, the dest-alpha modulation means a single scalar is used for all three channels, but it was vital. Without it, surfaces tend to look very flat when lit by afterlights.

FIGURE 2.6.13 Deferred effects enabled: headlights, streetlights, and undercar shadows.

Afterlights were used for streetlights, car headlights, searchlights on the light-house and Agency tower, and in effects such as muzzle flashes and explosions. To provide as many lights onscreen as possible, all the lights of a single species are drawn in a single call using instanced geometry. A cube is rendered for each light; back-face culling is enabled, so each pixel within the volume is drawn once rather than twice. Z-testing is enabled too, so pixels of the light cube that are behind geometry will be culled early by the hardware, and will not have the costly pixel shader executed for them at all.

The vertex program does the standard transformation into homogeneous clip space, and sets up the screen texture coordinate (i.e., texture coordinates corresponding to the vertex position onscreen, used to read from the screen-sized depth and normal textures). The pixel shader requires this texture coordinate to match the position of the pixel currently being shaded.

The obvious choice is to pass the homogeneous position down to the pixel shader and use a projective texture map mode, but unfortunately this failed in some cases.

The solution we used in *Crackdown* was for the vertex program to rescale the homogeneous position of the vertex so the Z and W components are constant for all vertices making up the bounding geometry of a particular afterlight. The values chosen are found by taking the cube center point, moving toward the camera by the light radius, and projecting the resulting position. This provides geometry guaranteed to be outside the light radius, but as far back as possible so as many pixels as possible are culled by the depth buffer.

This approach fails in some cases. For instance, if the camera is within the light sphere, the resulting point is behind the camera, and so the resulting geometry would not be rendered as it was behind the camera too. In this case, the vertex program generates a screen-covering quad.

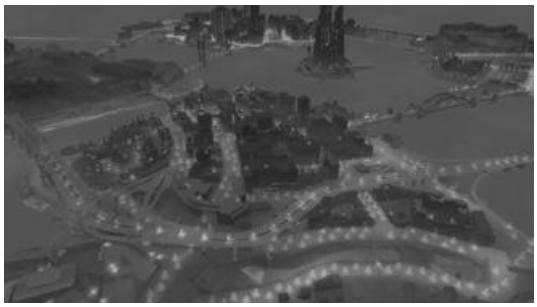
The pixel shader samples the depth buffer and per-pixel normal textures, and computes the normal and position for the pixel currently being shaded. Spherical lights simply compute the dot product between the vector from the light to the pixel and the surface normal, and attenuate the effect based on distance.

Headlights and searchlights were more complex. As well as the direct lighting effect, an in-scatter effect was simulated. This was simulated by defining an ellipse in screen-space, with the major axis along the headlight direction. If the current pixel was within the light volume, the in-scatter effect was reduced based on depth. In practice, this did not work very well. For instance, at the top of the Agency tower the searchlights produce phantom glows when they point away from you. Headlights of cars going the other way produce similar phantoms just after they pass you. A better solution would probably have been to simulate in-scatter using a group of crossed quads.

The most successful use is probably for the streetlights. At night, all 3,000 (or so) streetlights are rendered. From a sufficiently high vantage point, hundreds of lights can be

seen. In principle, the lights could be individually moved, recolored, and turned off at no extra cost. No extra CPU work such as per-light culling was required to achieve this. Even the first time the streetlights were hooked into the afterlights, the render time was acceptable. The streetlight effect also included a glare card, not provided by afterlights (see Figure 2.6.14)

FIGURE 2.6.14 An early afterlight test.



To avoid frame rate dips when many car headlights cover the entire screen (such as in a multi-car pileup at night), the afterlights are rendered to a half-size off-screen buffer. This made it quite unlikely that there would be afterlight fill-rate problems, but it meant that the quality of the lighting suffered. The worst artifact is the strobing effect seen in the pools of light under highway streetlights when driving at high speed. The pool is only a few pixels high so it suffers from undersampling. Some undersampling artifacts can be seen on the streetlight post in Figure 2.6.13.

A few other deferred effects were implemented in *Crackdown* such as soft shadows under cars and the cracks on the ground that appear after a hard landing. We hoped to add soft underfoot shadows and some localized darkening on walls close to the characters and perhaps props to simulate ambient occlusion effects, but these features did not make the cut.

Vehicle Reflections

The question, “Can we have vehicle reflections?” came up many times throughout the development of *Crackdown*. It kept coming up despite the fact that the answer was always a unanimous “no.” Code to render a dynamic cube map had been added, but the render time required was unaffordably high, so the feature was cut and considered dead.

In the final months of the project, we came up with a very cheap and simple hack that gave the impression of dynamic reflections on vehicles. I do not know the full story of how it was approved (the little I have heard is unprintable), but it was allowed in. Here is how it works.

The sharp transition between light to dark in a reflection is the quality that implies that a surface is shiny. In comparison, smudged, dirty, or matte finishes have blurred transitions. The dynamic part of *Crackdown’s* car reflections is a very sharp transition between light and dark, in a pattern that roughly matches the car’s surroundings.

Conceptually, the pixel shader ray-marches through a height field of the city to determine whether a pixel on the car is reflecting light or dark.

For both practical and political reasons, we simply could not afford to increase GPU time at all, so we used the absolutely cheapest possible implementation we could find. The “raymarch” is just a single sample of the height field with a carefully chosen position. To fit our memory budget, the city-wide height field is a single static 768×768 16-bpp map with no mipmaps, which meant each texel is roughly 10 feet across. (DVD transfer rate was a very limited resource, so streaming anything was also not an option. All data had to remain in memory.) The resulting color is chosen from two constant colors. There is no attempt to vary the color from building to building, no change in color with height, no distance fogging, and no antialiasing of the reflection edge.

The height field is packed in a way that accurately reproduces the abrupt height changes of buildings in a city. The texture map holds three channels of information; two (*A* and *B*) specify height fields, and the third (*C* —“choice value”) selects which of the two height fields to use at that point. The choice is made depending on whether the sampled value is over or under 0.5; the values of each texel are set up so the interpolated value crosses the 0.5 threshold as close as possible to the building edge. The *A* and *B* values are set up to be the height of the ground and top of the building.

In the pixel shader, the texture is sampled with bilinear filtering. If the *C* value is >=0.5, then the height to use is based on the *A* channel; otherwise, the *B* channel is used.

The pixel shader for the car computes the reflected eye-surface vector (which is also used to sample the reflection cube map), and steps a specific distance from the reflection point in that direction to find the point *p*. The height field is sampled at that location; if the height of the height field is lower than the height of *p*, then the reflection color is bright; otherwise, the

reflection color is dark.

This means that the samples are taken in a roughly hemispherical shape around the car. The step distance was 0.005 units in UV space, which is 3.84 texels—roughly 40 feet. The distance was chosen so that the sample would not step through an entire row of buildings. However, features smaller than this distance will be undersampled. A thin wall would show up as a thin stripe in the reflection, which is completely incorrect.

We can get away with this in *Crackdown* because the environment is filled with thick rows of buildings, and there were very few places where the overstep problem was noticeable. Thin barriers on the sides of bridges and the decorative columns by the tunnel entrance in Los Muertos are examples of the few problem areas. They were fixed by retouching them out of the raw height field image.

However, building corners are everywhere, and if you know what you are looking for, it is fairly straightforward to position the car and camera in such a way that a sharp corner appears in the reflection.

In practice, if you do not know what you are looking for, or the camera and car are not in just the right arrangement, it is not that noticeable. Also, the car is a complex shape, so even the reflection of simple forms will have kinks and curves, so when a building corner (or other such artifact) does appear, it is easy to overlook.

Thin features overhead such as walkways, pipes, and bridges are not a problem at all; they appear as a thin stripe in the reflection, which looks plausible.

The other implication of the fixed step distance is that distant features will never appear in the reflection. As you drive toward a building, it will only appear when you reach the critical radius, and will grow from small to large in the reflection as the car gets closer.

There are several effects that this method does reproduce. When driving down a street, the reflection silhouette of the buildings will have a shape that matches their heights and gaps at cross-streets. The highway has a small gap between the two lanes. This gap shows up as a light stripe in the reflection as you drive under it. Figures 2.6.15 and 2.6.16 show the car reflections in action.

FIGURE 2.6.15 Car reflections. Note the reflection of the overhead bridge in the car roof.

FIGURE 2.6.16 Car reflections. The overhead bridge is reflected in the back window.

As shown in Figure 2.6.17, height is handled too. If the car is on a bridge, the reflection will be clear. Underneath, the dark shape of the same bridge is visible. Unlike a single shared cube map, each car reflects the environment local to it. For instance, cars that enter a tunnel ahead of you will have the dark reflection pass over them at the appropriate time.

FIGURE 2.6.17 Car reflections. The two cars have quite different reflections.

Implementation Notes

For *Crackdown*, we used a 768×768 5-6-5 16-bpp texture with the choice value *C* stored in the red, 5-bit channel; the two height fields were stored in green and blue and so had a 6- and 5-bit channel each. We also experimented with using a smaller texture with a higher bit depth, and a larger texture with lower bit depth (DXT5), but the 16-bpp uncompressed texture was by far the best use of our memory budget. It required a little over 1 MB: 1,179,648 bytes.

The Xenon texture sampling mode could be tweaked to make it automatically scale the *C* channel to the range [−1, +1], which allowed the comparison to be made against zero, thereby saving an instruction.

Errors in the location of the building edge stand out much more than inaccuracies in the height of the top and bottom of the building. This would suggest putting the choice value *C* into the highest-precision channel available, as it controls the transition between height fields. In practice, 5 bits were sufficient. A full explanation for this choice is given in the texture map setup section below.

It is possible to get away with quite a bit of error in the height fields. The height range we settled on has a maximum representable height of roughly 85 m, so the height is stored in steps of 2.6 m or 1.3 m in the 5-bit and 6-bit channels, respectively. The height field that mainly provides the ground level is put into the 6-bit G channel to benefit from the extra precision, since the cars will mostly be on the ground level. Nevertheless, there are some reflection artifacts due to lack of height precision that can be seen while driving on some steep hills in the Volk district.

Before we added this feature to *Crackdown*, the car paint shader calculated the reflection vector and sampled a static cube map. Adding code to step along the reflection ray by the fixed distance, sample the height field, and tint the cube map sample by the light or dark reflection was in theory just a handful of instructions, but it ended up not changing the final shader instruction count, due mainly to good fortune with the shader compiler. Since the shader was already very ALU-heavy, the cost of the additional texture read could be absorbed.

Texture Map Setup

The source, high-resolution map we used was a 6144×6144 image rendered in 3dsMax using an orthographic camera. This resolution is eight times the 768×768 final map's size. It was rendered by loading all the second-highest LOD data into a single scene, which was near the limit of what 3dsMax could handle.

This 6144×6144 texture was touched up by hand to fix the various problem areas that arose, such as the Los Muertos pillars mentioned above. There were a few isolated black or white pixels that were rendering errors; a preprocess step cleaned them up. The preprocessor would also "grow" all features outward by two pixels in each direction to round them off and reduce the risk of undersampling (specifically, for each pixel a 4×4 neighborhood was searched and the maximum height found). Raw and processed images are shown below in Figures 2.6.18 and 2.6.19, respectively.

The texture map setup is built on the observation that if, when bilinearly filtering a texture, the four corner values are a linear function of position, then the interpolated value will also be a linear function of position.

This is useful for setting up the channel that controls the height field choice. If the value is a linear function of position, then the boundary where the value crosses the threshold value will be a straight line. This is ideal for most building edges.

So in principle, the choice value near a straight building edge can be defined to be a linear function of distance from the edge, taking the value 0.5 on the edge.

Specifically,

$0.5 + s \times d \times k$

where d is the distance from the edge (measured in texels), and s is +1 or −1 depending on which height field is the correct one to use on that side of the line. k is a scale applied to bring the values to within the [0, 1] range. The only texels that matter are those that border the

edge; all others can be safely clamped to 0 or 1. This function crosses 0.5 at the edge. It is a linear function of position near straight edges, and provides an acceptable behavior for more complex edges.

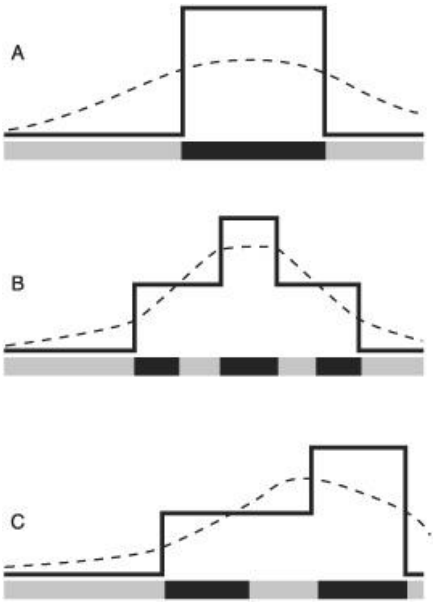
The next question is how to define the regions where each height field is valid, and here things become murkier. The choice function partitions the world into subsets: areas where the *A* height field is valid and complementary areas where the *B* height field is valid. If the city was made out of very simple buildings (such as cuboids), then it would be possible to define the *A* height field to be used for all the building tops and *B* everywhere else. Of course, the city in *Crackdown* is much more complex than this, so a more sophisticated partitioning was desirable.

The approach we took for *Crackdown* worked as follows. The source, highresolution height field was blurred. Points where the original value was greater than the blurred value were specified to use the *B* channel, and all others used the *A* channel. Figure 2.6.18 shows three sample cross-sections; the red and blue sections beneath the cross-section correspond to whether the *A* and *B* height field is valid at that point. Case (a) shows a simple building on a flat plane. The *B* height field is used for the top of the building and the *A* height field for the ground. Case (b) is slightly more complex, and the height field assignment changes in the middle of a flat roof. This is legal and moral; the height fields can be set up in a manner that provides the roof profile exactly, but this behavior is not ideal. The height field switch in the middle of the middle-height roof is not required. However, case (c) requires the height field switch, and establishing whether the height field regions can be divided into concentric rings (thereby allowing the simple partitioning) is far from trivial.

The packed, low-resolution map requires two height-field values and a choice value for each texel. The choice value *C* is set by first sampling the corresponding point in the high-res map to find whether the *A* or *B* map is used. This sets the *s* value in the above equation; *d* is found by searching the neighborhood in the high-res maps for the closest transition between *A* and *B* height fields.

In practice, it was acceptable to set up the choice value by simply searching in the four directions along the axes. Doing a full search of the neighborhood did not improve quality that much. For an eight times supersampled source image, the range of results is {−7.5, −6.5, …, +7.5}, which fits into 5 bits with no loss of precision. It is for this reason that the choice values had no need of the 6-bit channel.

FIGURE 2.6.18 Blurred heights for partitioning.



One option for calculating the *A* and *B* value is to use an expensive method such as hill-climbing, but we found that a simple and cheap weighted average was sufficient. The height was sampled from a 16×16 neighborhood in the high-res height field, with a per-sample weight of 1/squared distance. When setting up the *A* channel, only entries using the *A* height field in the high-res map are considered and vice versa. The 16×16 size of the neighborhood region is chosen because it is exactly the area in the world that would be affected by a change to that low-res texel. In the event that no pixels in the neighborhood (for instance) are *A* when we are setting up the *A* channel, we can be certain that the value used for the low-res *A* texel will have no effect.

Figures 2.6.19 to 2.6.27 show the various images. Figure 2.6.19 is a 512×512 subset of the source 6144×6144 height field rendered out of 3dsMax. Figure 2.6.20 is the cleaned-up version. In Figure 2.6.19, there are some spurious black and white pixels, for example, a white pixel on the edge of the building in the top right at 2 o'clock, and a black region on the right of the

building at 9 o'clock. Figure 2.6.20 shows that these artifacts have been cleaned up by the preprocess step. The Los Muertos pillars mentioned above are the two dots in the upper left-hand corner of the maps. They were painted out of the production maps but are shown here for reference.

FIGURE 2.6.19 Section of the original height field, rendered in 3dsMax. Note the odd black or white pixel. 512×512.

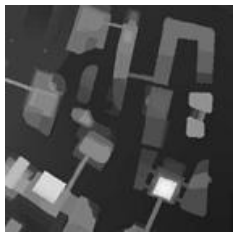


FIGURE 2.6.20 Height field after automatic cleanup. Spurious black or white areas fixed. 512×512.

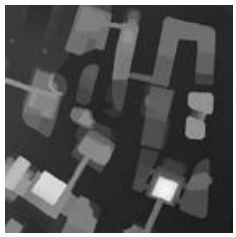


FIGURE 2.6.21 Packed height field *A*. 64×64.

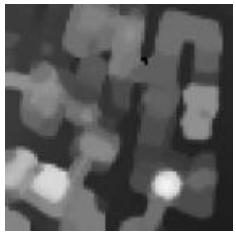


FIGURE 2.6.22 Packed height field *B*. 64×64.

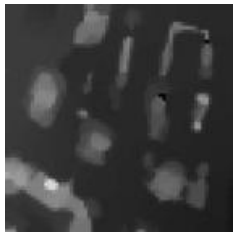


FIGURE 2.6.23 Choice value *C*. 64×64.

Figure 2.6.24 is the height field reconstructed from the packed 64×64 map. Each 8×8 square at 8 bpp is packed into a single 16-bpp texel, so the compression is 32:1. It does look low-quality in comparison to the original, but the only use for this height field is for car reflections, and in practice the approximation is acceptable.

Figure 2.6.25 shows the absolute per-pixel reconstruction error. The sprinkling of bright pixels along an edge shows the errors in the edge's location, which is less than a texel or two in most cases. Complex areas have much lower quality.

FIGURE 2.6.24 Height field reconstructed from packed 64×64 image. 512×512.

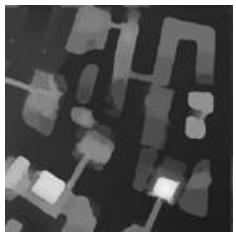


FIGURE 2.6.25 Absolute error between original and reconstruction. 512×512.

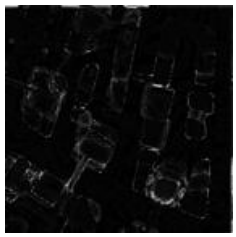


FIGURE 2.6.26 64×64 section of original height field. Black lines indicate the relative size of each texel in the packed height field.

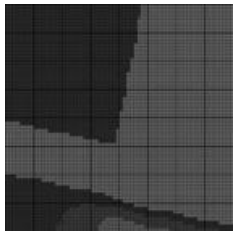
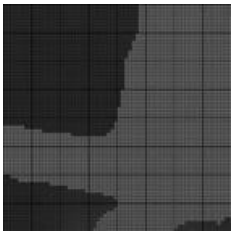


FIGURE 2.6.27 64×64 section of reconstructed height field. Black lines indicate the relative size of each texel in the packed height field.



The overhead bridge mentioned in the description for Figures 2.6.15 and 2.6.16 is included in these height field images. In the 512×512 maps it can be seen on the far left, slightly above the center and running left-right off the left-hand side of the image. The cars are driving toward 6 o'clock.

The fake reflections were a huge asset to the vehicles. Without them, the cars looked dull and very poor. The technique described here has many limitations and quality problems, so the reflections can only stand up to a small amount of inspection before inaccuracies can be seen. But given how cheap they are to render and how little memory they require, when compared to the benefits they provided, they provided exceptional value for us.

This method for packing a height field in a way that preserves straight edges and abrupt transitions may be useful for other data, such as precalculated shadow maps.

Conclusion

This article described a number of rendering techniques that were used in the game *Crackdown*. These are sky, clutter, outlines, deferred lighting, and car reflections.

For the reasons described in the introduction, these features are relatively cheap, and can be implemented cheaply and independently. The aim of this article is to present enough information about these methods' benefits and costs for them to be easily judged, and for the weaknesses and quality problems to be detailed well enough for there to be no surprises if they are employed.

Endnotes

[1] Packing *z*, *dz*/*dx*, and *dz*/*dy* into the components of a vector means adding a second packed function sample, and scaling by a constant can be vectorized. They are the addition of two vectors and the scaling of a vector by a constant, respectively.

Acknowledgments

Providing the dense crowd, shadows, and high-quality vista, and implementing all the other graphics features on discontinued middleware was a struggle. A lot of people made major contributions to *Crackdown's* graphics, and it would be unfair to them for an article on *Crackdown's* graphics to only have my name associated with it. Everyone involved in the graphics provided bug fixes and optimizations too numerous to be mentioned here; here is only the most significant and visible result from each person's work.

A programmer from Xen single-handedly wrote and supported the particle system runtime and editor, the shadow system, the water tech, and the window shader solution, and provided some vital optimizations to the renderer. He managed to add precompiled command buffer support, without which we would have had to cut the LOD1 block count, and *Crackdown's* vista would have been much lower quality.

George Harris worked on graphics (among many other things) from the first prototype all the way to the end. He built or helped build a great deal of the graphics tech, and fixed bugs in all of it. He built and supported most of the prop and car rendering system, including the damage and part separation, and vehicle morphing. He also worked long hours on a long list of unglamorous and unpleasant but necessary tasks, such as the regular breakage of the specular highlight.

Neil Duffield, Dave Lees, and Peter Walsh provided the crowd technology and distant vehicles. Dave Lees also retrofitted the alpha-blended render pass, and added occlusion culling for props and cars. Peter Walsh integrated SpeedTree, and dramatically improved its render time and memory use.

Kutta Srinivasan implemented *Crackdown's* anti-aliasing. Crackdown's depthnormal prepass was not natively supported by the standard tiled rendering technology, and we were searching in vain for alternative AA methods until Kutta found a solution.

References

[Decaudin96], Decaudin, P. “Cartoon Looking Rendering of 3D Scenes,” Research Report INRIA #2919, June 1996.

[Dudash04] Dudash, B. “Mesh instancing,” nVidia Technical Report, May 2004. Can be found online at http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/HLSL_Instancing/docs/HLSL_Instancing.pdf

[Hargreaves04] Hargreaves, S., “Deferred shading,” Game Developers Conference, March 2004. Can be found online at http://myati.com/developer/gdc/D3DTutorial_DeferredShading.pdf.

[Neil05] O’Neil, S. “Accurate Atmospheric Scattering,” *GPU Gems 2,* Chapter 16.

[Rost06] Rost, R. “OpenGL(R) Shading Language (2nd edition),” Chapter 20.6.