Chapter 3

Artist-Directable Real-Time Rain Rendering in City Environments

Natalya Tatarchuk[5]
ATI Research

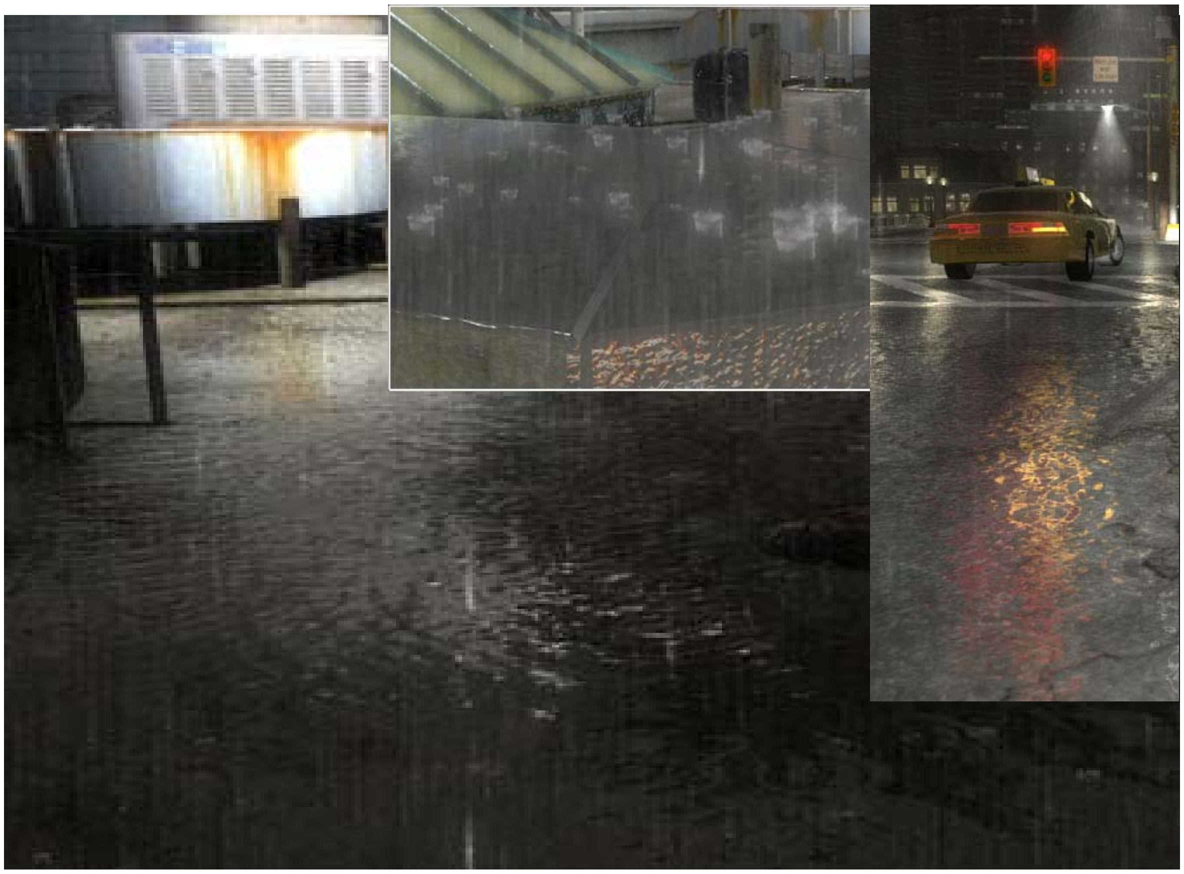


Figure 1. *Various rain effects seen in the state-of-the-art interactive demo "ToyShop": dynamic water puddle rendering with raindrops, raindrop splashes, and wet view-dependent reflections of bright light sources.*

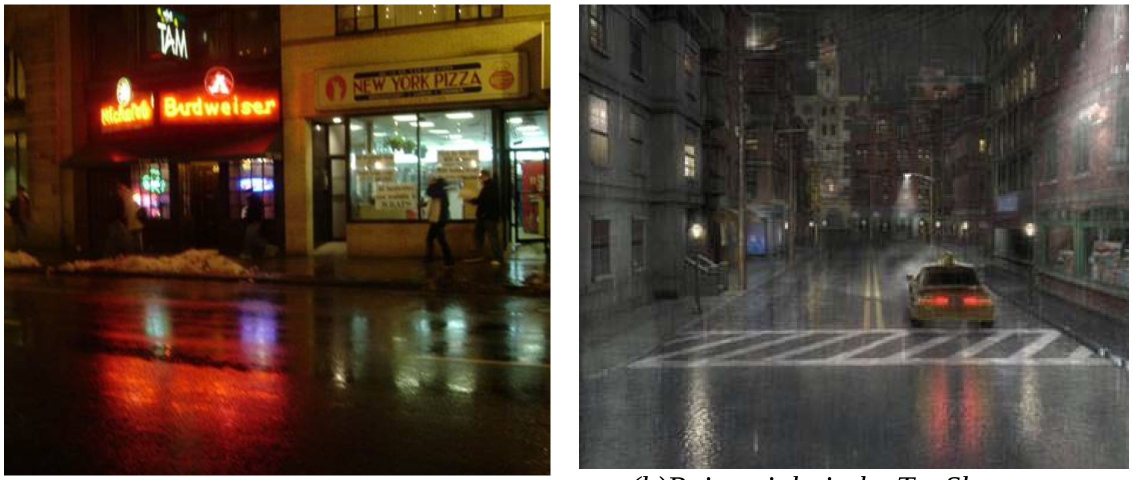
[5] natasha@ati.com

3.1 Abstract

In this chapter we will cover approaches for creating visually complex, rich interactive environments as a case study of developing the world of ATI "ToyShop" demo. We will discuss the constraints for developing large immersive worlds in real-time, and go over the considerations for developing lighting environments for such scene rendering. Rain-specific effects in city environments will be presented. We will overview the lightning system used to create illumination from the lightning flashes, the high dynamic range rendering techniques used, various approaches for rendering rain effects and dynamic water simulation on the GPU. Methods for rendering reflections in real-time will be illustrated. Additionally, a number of specific material shaders for enhancing the feel of the rainy urban environment will be examined.

3.2 Introduction

Our goal was to create a moment in a dark city, downtown, during a rainy night. Fortunately, we had many opportunities for research, having started on the concept for the demo in the middle of October in Boston. As a comparison, in Figure 2 we see a snapshot of the theater district in Boston downtown compared with the final rendering in the ToyShop demo.



(a) Rainy night in downtown Boston *(b)Rainy night it the ToyShop town*

Figure 2. *Comparison of a photograph from a real city during a rainy night versus a synthetic rendering of the interactive environment of ToyShop.*

Some games which incorporate rain rendering in their worlds use a very straight-forward approach: rendering stretched, alpha-blended particles to simulate falling raindrops. This approach fails to create a truly convincing and interesting rain impression. Frequently, the games only include one or two effects such as the stretched rain particles and perhaps a simple CPU-based water puddle animation to simulate the impression of rainy environment. This results in an unrealistic rendering with the rain not reacting accurately to scene illumination, such as lightning or spotlights. Some notable exceptions to this are the recently released *Spinter Cell* from Ubisoft and *Need for Speed: Most Wanted* (Xbox 360) with much improved rain rendering.

We present a number of novel techniques for rendering both particle-based rain and rain based on a post-processing image-space technique, as well as many additional secondary rain effects, without which one cannot generate an immersive rain environment. Rain is a very complex atmospheric physical phenomenon and consists of numerous visual cues:

- Strong rainfall
- Falling raindrops dripping off objects' surfaces
- Raindrop splashes and splatters
- Various reflections in surface materials and puddles
- Misty halos around bright lights and objects due to light scattering and rain precipitation
- Water, streaming off objects and on the streets
- Atmospheric light attenuation
- Water ripples and puddles on the streets

We have developed methods to render all of the above in a real-time environment. Our techniques provide a variety of artist-directable controls and respect the rules of physics for simulating rainfall. They utilize light reflection models to allow the rain to respond dynamically and correctly to the lighting changes in the complex environment of the ATI "ToyShop" demo due to illumination from atmospheric effects (such as lightning).

3.3 Rendering system requirements and constraints

We faced a number of strict constraints while developing this interactive environment. First and foremost, the memory consumption needed to be in check – the goal was to fit the entire world of this environment into less than 256 MB of video memory. This was required for optimal performance on any graphics hardware, although this demo was specifically targeted to push the limitations of ATI Radeon X1800 XT graphics card. This memory requirement was a severe constraint for our development due to the large scope of the environment we wanted to create. At the end of the production we managed to fit the entire assets into 240 MB including 54 MB memory used for back buffer and offscreen storage, 156 MB for texture memory (including many high resolution textures) and 28 MB for vertex and index buffers.

We must note that the high performance interactivity of this environment would not be achievable without using 3Dc texture compression. In order to create a realistic environment, we used a great deal of high resolution textures to capture the extreme detail of the represented world (for example, light maps and lightning maps, high resolution normal maps and height maps, high resolution color maps and specular maps). Using 3Dc technology allowed us to compress nearly half of a gigabyte of texture assets (478 MB, to be exact) to 156 MB. Specifically, we used texture compression formats such as ATI2N and ATI1N, DXT1 and DXT5 to compress majority of our textures. Additionally, we recommend using vertex data format DEC3N, which allows 10-bit per-channel data storage, to reduce memory footprint for geometry. This format gives

3:1 memory savings while maintaining reasonable precision. We used it for encoding vertex normal, tangent and binormal data, as well as some miscellaneous vertex data. Note that this format is available on a wide range of consumer hardware and will be included as a first-class citizen in the upcoming DirectX10 API.

We would like to provide a visual comparison of what the usage of 3Dc texture compression technology and the vertex compression format DEC3N allowed us to do in our interactive environment: in Figure 3a, the scene is rendered using compressed formats and in Figure 3b, only a portion of the original scene is rendered. This is due to the fact that the entire scene would simply not fit into the graphics card memory, and thus would not be rendered in real-time. Although this example is a bit contrived (since in real-life productions, one would first reduce the resolution of texture maps and decimate the geometric meshes), it serves the purpose of stressing the importance of high quality compression technology for interactive environments.

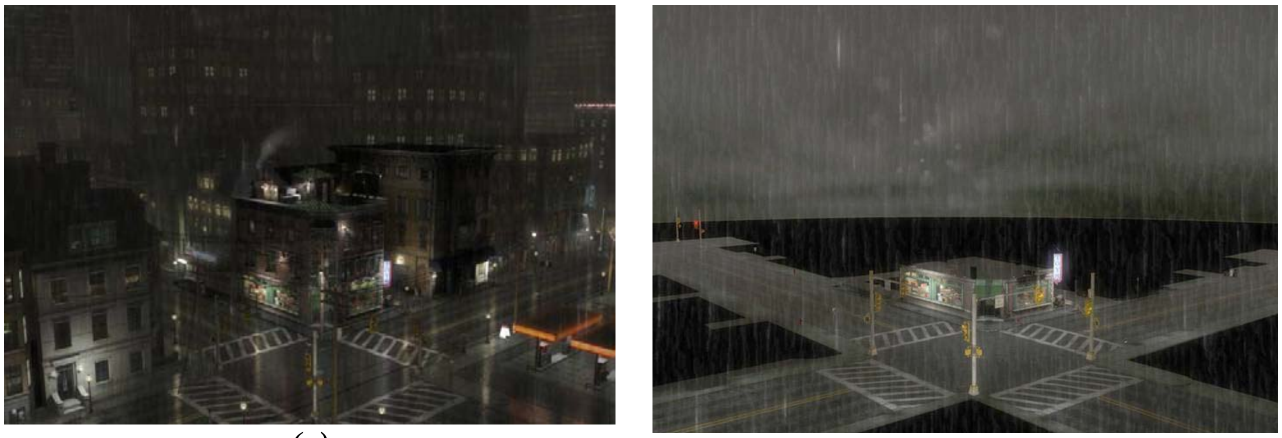

(a) *(b)*

Figure 3. *In (a) the entire scene is rendered with the use of 3Dc and DEC3N technology. Without using this technology, we are only able to fit a small portion of the original environment into the graphics card memory (b).*

With the advances in programmability of the latest graphics cards and recent innovative games such as FarCry® by Crytek and Half-Life 2 by Valve Software, everyone realizes the need and the desire to create immersive interactive environments. In such worlds a player has many options to explore the surroundings; and the complexity of the environments helps make the experience truly multifaceted. However, that said, rich detailed worlds require complex shaders, and require a variety of those.

In order to create the imagery of the ToyShop environment, we have developed a number of custom unique material and effects shaders (more than 500 unique shaders). Of course, approximately half were dedicated to rendering rain-related effects, including dynamic water simulations and wet surface material shaders. Just about one third of the entire shader database was used for rendering depth information or rendering proxy geometry objects into reflection buffers (see section 3.5). Finally, a number of shaders (roughly one sixth) were dedicated to rendering post-processing effects such as glow and blurring, along with the many custom particle-based effects. We highly recommend developing an extensive include framework for your projects. Separating shader sections responsible for rendering specific materials or effects allowed us faster iteration. For example, we used more than 20 include files containing functions to compute reflections, shadows using shadow mapping, high dynamic range rendering including HDR lightmap decoding and tone-mapping, mathematical helper functions and lighting helper functions. In essence the include file system allows a game developer to emulate the richness of shader programming languages such as RenderMan®.

3.3 Lighting system and high dynamic range rendering

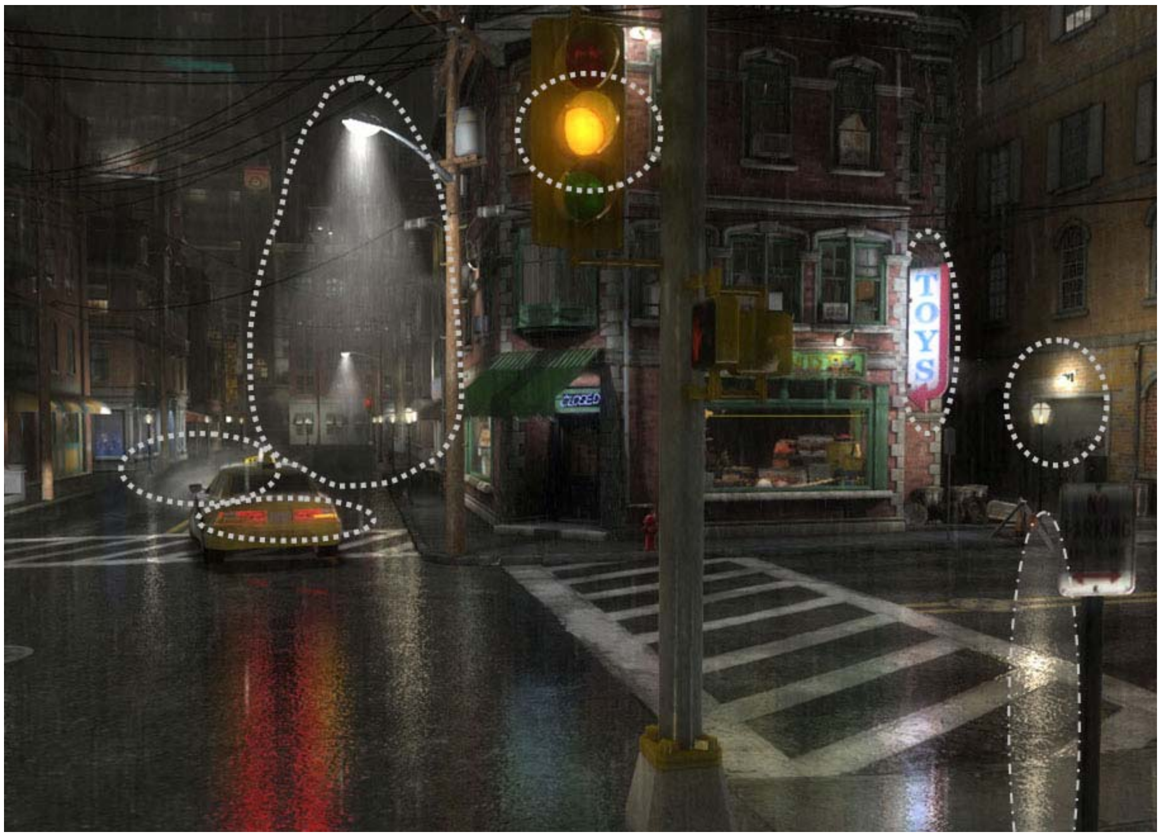


Figure 4. *In the dark stormy night in the ToyShop town, we see a large number of apparent light sources*

We set out to create a moment in a dark city in a stormy, rainy night. But even in the midst of a night there is light in our somber downtown corner. Notice the large number of perceived light sources visible while flying around the environment (see Figure 4), such as the bright street lights, the blinking neon sign on the corner of the toy shop, various street lamps and car head and tail lights, and, last but not least, the lightning flashes. Every bright light source displays reflections in various wet materials found in this city. The sky also appears to illuminate various objects, including the raindrops and their splashes in the street puddles. While there are a number of different approaches that were used for implementing specific effects, an overall lighting system was developed in order to create a consistent look.

3.3.1 HDR rendering on a budget

Any current state-of-the-art rendering must be done using high dynamic range for lighting and rendering[6]. Without using this range of colors, the resulting lighting in the scene will feel dull. (See chapter 7, section 7.5 for an excellent review of the HDR system in Valve's Source engine). However, considering the memory constraints placed on this production, strict care had to be taken with regards to the specific format and precision selection for the rendering buffers. Additionally, since every rendered pixel on the screen goes through the tone-mapping process, the choice of the tone-mapping function directly affects the performance. Depending on the specific selection, it can constitute a significant performance hit. Because of the multitude of effects desired for the immersive environment rendering, both memory requirements and performance considerations were very specific and stringent.

The goal lied in balancing performance and memory usage with an expanded dynamic range and good precision results. Recent graphics hardware such as ATI Radeon X800 and above provides access to renderable surfaces which have 10 bits per channel. This surface format provides excellent precision results at half the memory usage of 16 bit floating point formats. All of the back buffers and auxiliary buffers were created with this surface format, as well as the HDR lightmaps used to light the environment.

One challenge with using HDR for rendering in production environments currently lies in the scarcity of publicly available tools to work with the HDR art assets. Aside from previewing the lightmaps in-engine in real-time, there was a considerable difficulty for visualizing the lightmaps outside of the engine. We recommend using [HDRShop] for visualization of the light maps early on, as they take a considerable amount of time to render and ideally should not be rendered multiple times.

3.3.2 Tone mapping and authoring HDR lightmaps

For our HDR lightmap decoding, we used the RGBS (fixed-point RGB with shader Scale) approach for HDR lightmap decoding in shaders to maximize the available dynamic range of the 10 bit surface format. Simply using the straight 10 bit format for encoding and decoding light information in an expanded range results in the range expanded from [0, 1] to [0, 4]. However, if we are able to use the extra two alpha bits as a shared exponent, the range can be stretched to [0, 16]. See [Persson06] for a very thorough overview of available texture formats in the context of HDR rendering.

The tone mapping curve was expressed as a four-point artist-editable spline for more control of the lighting. Some suggestions for integrating HDR into a game engine or a production pipeline:

- Start by applying a linear tone mapping curve

[6] For an excellent introduction and overview of image-based lighting and high dynamic range rendering, see [Reinhard05].

- This allows you to make sure that the values that you are getting from lightmaps in engine are correlated to the actual stored values in lightmaps
- Remember – lightmaps are extremely time-consuming to render (especially if using any precomputed global illumination effects). Ideally the pipeline should be thoroughly tested prior to starting the process of lightmap rendering. That means testing the tone mapping curves first for accuracy

Let's look at some examples of HDR lightmaps in the context of HDR rendering and tone mapping. Figure 5 shows an example of the tone mapping curve incorrectly darkening and significantly reducing the available range for lighting. Therefore all details in the shadows are completely lost.

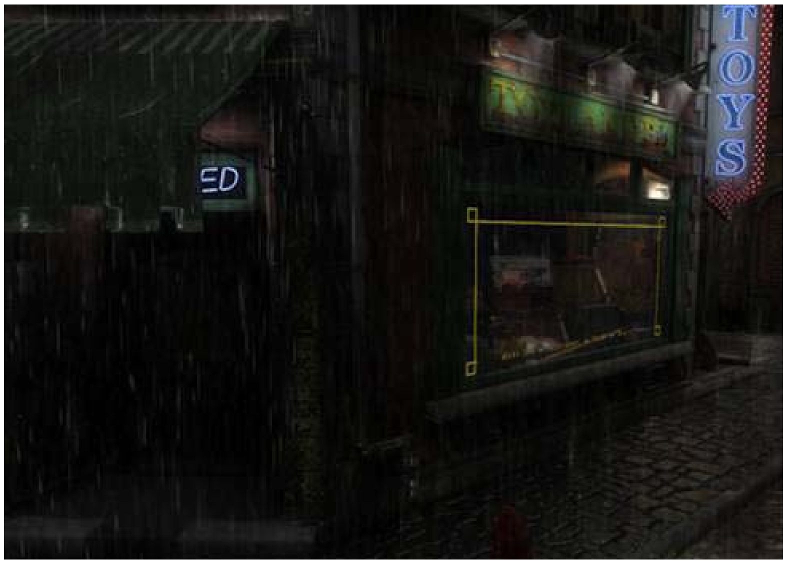


***Figure 5**. Example of a tone mapping curve set to excessively darken the scene*

In Figure 6 we are seeing the opposite effect, where the tone mapping curve is over - saturating the scene and all of the details in the bright regions are blown out and lost.

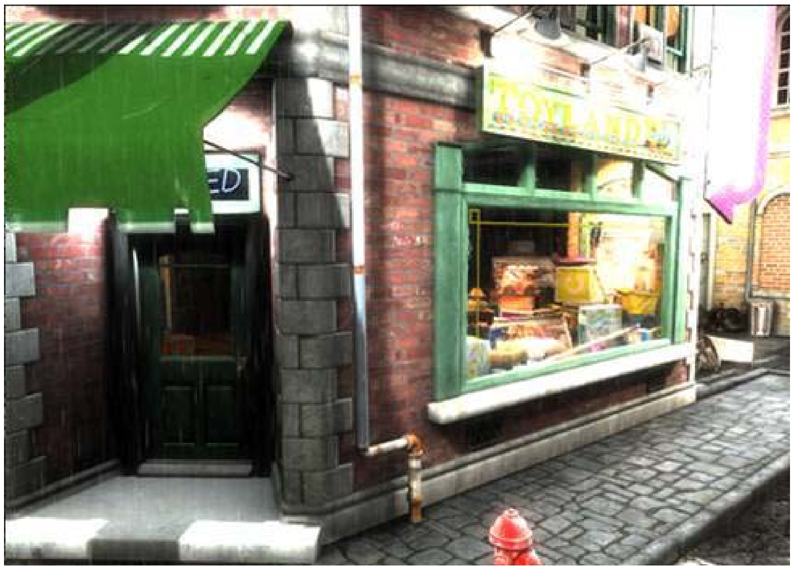


***Figure 6**. Example of a tone mapping curve set to excessively oversaturate the scene*

We can see the tone mapping used in our interactive environment in Figure 7 where the contrast is set to the artistic choice. In essence, the tone mapping process comes down to being an aesthetic preference, and all of the three figures will look appealing in some circumstances and undesirable in many others. Additionally, the tone mapping curves can be dynamically animated depending on the scene intensity, location and timing. For an example of dynamic tone mapping, see chapter 7 of this course for the description of the HDR system in Valve's Source engine.

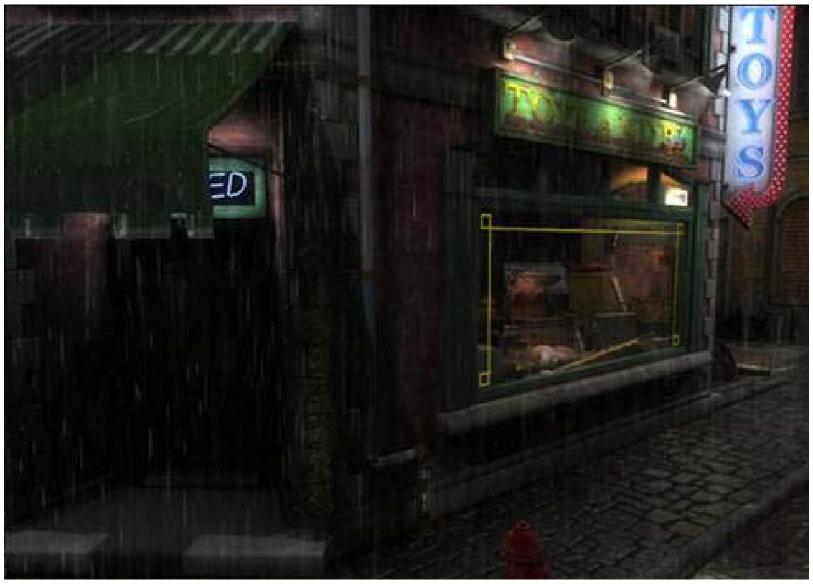


***Figure 7**. The tone mapping settings used in our environment allowed us to preserve desired amount of details in shadowed areas while maintaining overall contrast in the scene. Note the details under the "Closed" sign of the store.*

With respect to authoring lightmaps for any interactive rendering, the key notion to remember is that the process of rendering lightmaps traditionally is rather time consuming (in terms of the rendering time), since it is an offline process. Thus the goal is to postpone this to the later parts of the development after iterating on some test lightmaps as a proof of concept. And, naturally, rendering lightmaps can only happen when the scene geometry and texturing, along with the lighting setup, has been completed and approved for the final environment.

We chose to approximate global illumination effects with a less computationally-intensive approach. Due to time constraints and hardware limitations, we could not render both the final gather and global illumination in Maya® 6.0 Mental Ray for the lightmaps. However, the visual results of global illumination are crucial for the impression of light bouncing off nearby objects, and we needed a method to approximate it in the lightmaps. As an alternative, we chose to render lightmaps into a 32 bit texture format with final gather only using HDR light sources in Maya® (however, the light intensities for these light sources were set such that the final lighting values never exceed 16). See section 3.3.1 for the description of our HDR lighting and rendering setup. Additionally, the light intensities varied depending on the falloff type and distance from surfaces. An example of such a resulting lightmap render is in Figure 8a.

To help approximate the soft lighting of global illumination we rendered the ambient occlusion map (in Figure 8b) into a 32 bit grayscale texture. This texture can be thought of as a global illumination noise map, to provide us with the darkening of the lighting in objects' creases and in the areas where objects meet. To combine the ambient occlusion map and the prerendered lightmap, we used the Digital Fusion software package, first applying tone mapping to the ambient occlusion map to move the values into the appropriate range for the rendered lightmap and then adding it to the lightmap texture. The resulting final lightmap in Figure 8c displays an example of the actual art asset used in the interactive environment. Note that we are able to preserve both the hard contact shadows and the soft bounced lighting in the resulting lightmap.

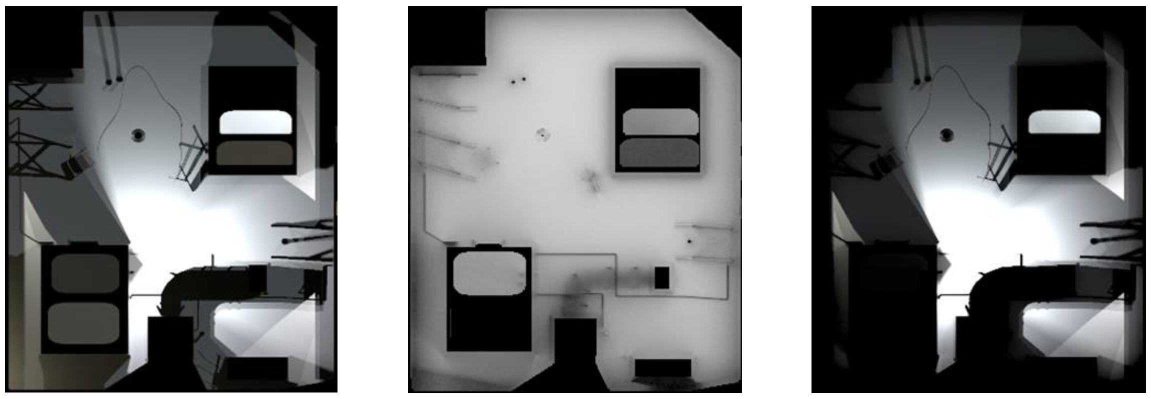

(a) *(b)* *(c)*

Figure 8. *An example of the lightmap asset creation process. We start out with the lightmap rendered with the final gather process in Maya® 6.0 Mental Ray (a), and then combine it with the ambient occlusion map from the same environment (b) to create the final art asset (c) used in the real-time environment.*

3.3.3 Shadowing system

In the heart of the night, shadows rule the darkness. Enhancing the dynamism of the immersive environment of our interactive city meant having support for varied light conditions including dynamic lights. The shadow mapping technology allowed us a good balance between the final performance and the resulting quality of dynamic soft shadows. Shadow mapping refers to the process of generating shadows by rendering from the point of view of the light source and then using the resulting texture (called the *shadow map*) during the shading phase to darken material properties depending on whether they are occluded with the respect to a specific light source. For an excellent overview of the shadow mapping and other shadowing techniques see chapter 6 in [Akenine-Möller02].

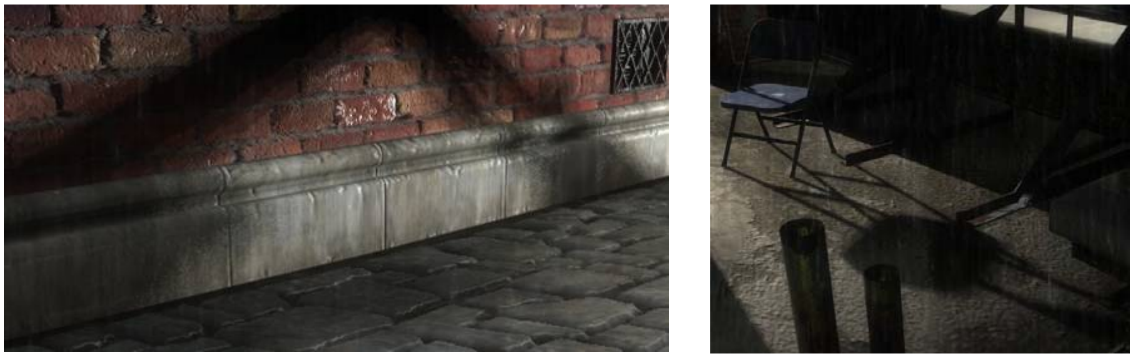

(a) *(b)*

Figure 9. *Examples of shadow mapping technology used to render dynamic shadows on varied surfaces in the ToyShop demo*

We used a single animated shadowing light for our environment, which resulted in a single shadow map. To conserve memory, we took advantage of the novel hardware feature in the ATI Radeon series called 'depth textures'. These are efficient and flexible surfaces for shadow mapping and other algorithms (see [Isidoro06] for a meticulous overview of these hardware features, and the efficient methods to implement shadow mapping and soft shadow filtering). Since these surface formats do not mandate binding of an identical sized color buffer, they provide excellent savings in memory for depth-only rendering (we were able to bind a 1x1 color buffer to a 1024x1024 shadow map). A 16-bit DF16 texture format provided good precision for our depth complexity needs. In Figures 9a and 9b we see some examples of shadow mapping applied in our environment.

Shadow mapping has several important considerations that any developer needs to address in order to obtain visually pleasing results. The actual resolution of the shadow map will be directly proportional to the appearance of the shadows (as well as affect the performance). If the shadow map resolution is not high enough for the desired scene and viewer closeness to the shaded objects, potential aliasing artifacts may appear. We used a relatively large shadow map of 1024x1024 to improve the results. Additionally, a common bane of shadow maps is the need to filter the penumbral regions of the shadows in order to create soft shadows without aliasing artifacts. A typical approach includes using percentage closer filtering (PCF) ([Reeves87]) to soften the shadow edges. However, simple PCF does not typically yield visually satisfactory results without aliasing artifacts or visible filtering patterns. We improve on the approach by using a randomized PCF offset sampling with custom filtering kernels ([Isidoro06]). For bright materials (or regions, such as in Figure 9a) we used an 8-tap Poisson disk kernel with random rotation offsets encoded into a lookup texture. Random rotation of the sampling kernel is needed for bright surfaces with smooth albedo, since the shadowing artifacts are most visible for these materials. For dark and noisy textured surfaces, we can relax the sampling requirement and can use a fixed Poisson disk kernel for sampling the shadow maps (as in Figure 9b).

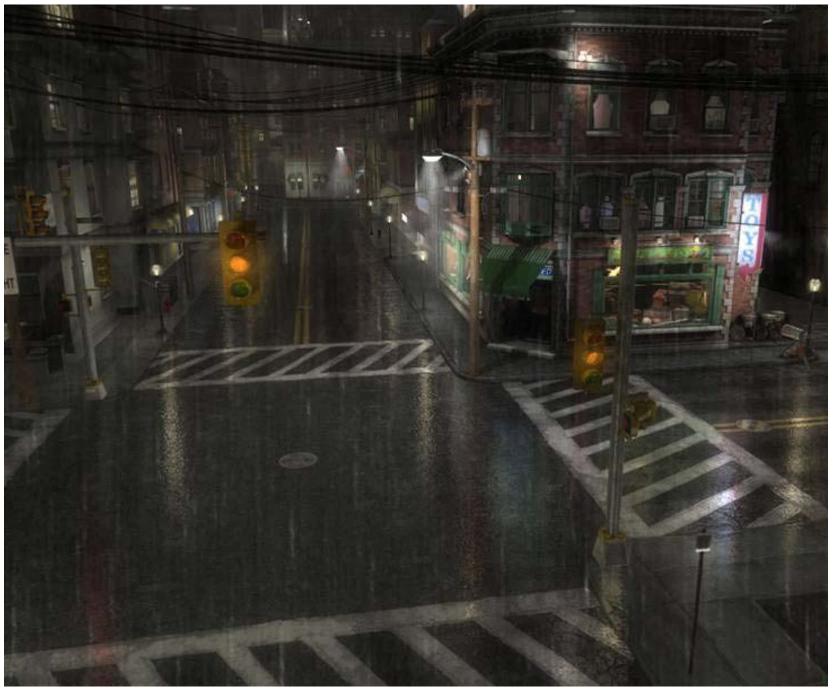
3.3.4 Lightning System and Integration

Once we have the lighting and shadowing systems in place, we can start putting together the other components in order to create the illusion of a dark stormy night. Lightning and thunder increase the feel of a rainy, turbulent environment. Illumination from the lightning flashes needs to affect every object in the scene. As such, uniformly aligned shadows are crucial – we need to create a feeling that somewhere in the sky, a lighting bolt just struck in a particular location. At the same time, we have to keep in mind that we are using a single shadowing light and a single shadow map for our shadowing solution. Computing lightning shadows for each additional lightning 'light' can significantly impact performance and memory footprint (with additional shadow maps).

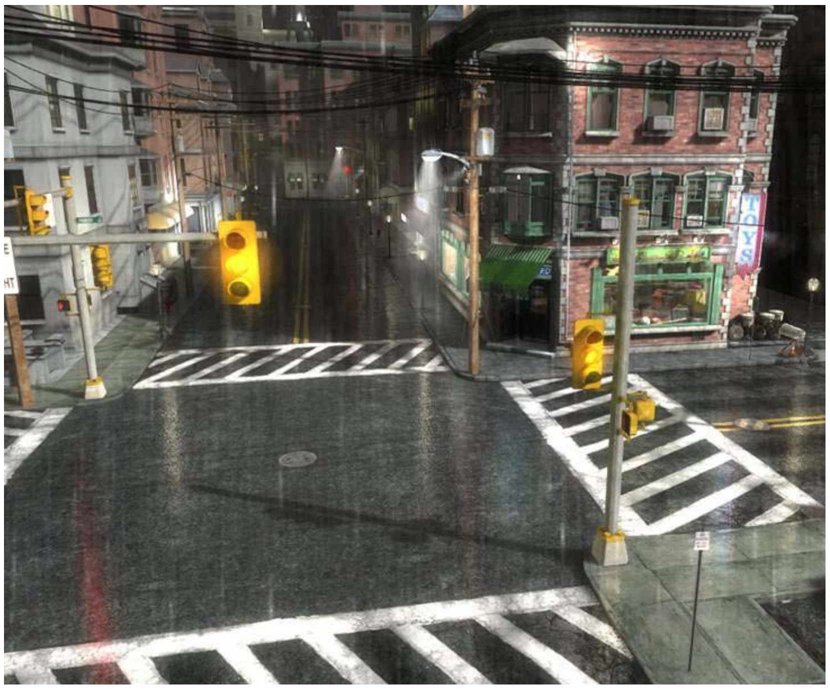
In Figure 10a the city corner is rendered with just the regular lighting and shadowing system. In Figure 10b we captured the effect of a lightning flash on our environment – note all of the objects are throwing uniformly aligned shadows onto the street, creating a strong impression of actual lightning flash (combined with the sound effects of thunder).

Lightning is a strong directional light that affects every object in the scene. Creating a convincing and realistic lightning effect is challenging for a variety of reasons. In our interactive environment, the viewer can get very close to the objects when lightning strikes. That means that the resolution of the shadows generated by the lightning flash must hold up regardless of viewer's distance from an object. Finally, the illumination from the lightning must seamlessly integrate into the main lighting solution for our environment, as we are using a consistent illumination model for all objects in our scene (including shadow mapping).

For artistic reasons, we felt it was important to include lightning illumination in our environment, despite the challenges – it heightens the mood. A dark night with rough weather would not affect the viewer in the same manner without the sudden surprise of a lightning flash followed by the inevitable thunder. Additionally, the extra illumination from the lightning helped us show off the details of various effects in the scene, which may have gone unnoticed in the pure darkness of the night.



(a) Scene rendered in the regular manner without a lightning flash



(b) Scene rendered during a lightning flash

Figure 10. *Comparison of the regular environment rendering in (a) versus the same environment lit by a lightning strike from the right (b). Notice the correctly aligned object shadows in (b).*

Our solution lies in special *lightning lightmaps* for the illumination due to lightning flashes. We can prerender the result of illuminating the environment from several directions, mimicking the light from a lightning flash into a lightning lightmap texture. Unlike a regular lightmap, this texture does not need to store full lighting color information – we are only planning to use it to modulate the regular illumination computed for each pixel (as an intensity multiplier of the underlying HDR lightmap). Therefore we simply encode the value into a single channel 8 bit texture. In our case, we found that computing the illumination for two unique lightning light locations was sufficient and provided good results for the additional increase in memory consumption (as a two-channel 8-bit-per-channel lightning lightmap, example in Figure 11). The scene information is encoded in a manner similar to regular lightmaps. We provide the artists an editable intensity parameter for custom mixing of the two lightmaps – which can be animated and controlled dynamically on a per object basis by a rendering script in our engine (we use the Lua programming language). The first lightmap contained the illumination from a lightning flash at an angle from far away, and the second lightning lightmap contained the illumination from a lightning flash directly above the center of the scene. Mixing these two maps in different object- and time- specific ways creates an illusion that we have a wider variety of lightning flash directions that we actually did.

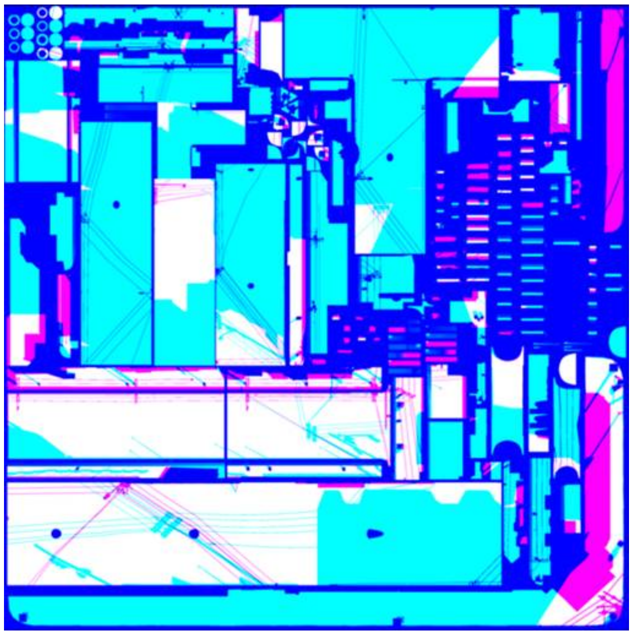


Figure 11. *An example of a lightning lightmap where an individual lightning intensity value is stored for two lightning light locations in red and blue channels of the texture.*

Every shaded pixel in our environment uses lightning illumination information. The rendering script propagates the animation parameter for each of the two lightning flashes to all of the shaders in the form of uniform parameters (floating point value of lightning brightness and location). In a specific material shader we can either read the lightning lightmap for the intensity value for the specific lightning selection or simply use the lightning brightness parameter (controlled by the artists from outside the script). (Or both types of parameters can be used simultaneously). The lightning lightmap sample is added to the regular lightmap sample before tone mapping. The performance cost for integrating this type of lightning illumination computation is very low – a single texture fetch plus several ALU operations in the shader to compute lightning flashes from varied locations. All objects in our real-time environment use this scheme and thus appear to respond accurately to lightning illumination in the scene.

Additionally, note that in realistic scenes, translucency of water is affected by the lightning flash illumination. We mimic this effect in our rendering. This can be accomplished by using the lightning brightness value to adjust the pixel's opacity (alpha value) when the lightning flash occurs. We use this approach extensively in the rain effects, improving the visual quality of those effects and making the lightning flash appear more natural.

3.4 Post-processing effects for rain rendering

In recent years post-processing has become a popular approach for adding visual variety to games, as well as approximate many camera or environment properties of the world around us. For example, the post-processing pipeline is used to add the depth-of-field effects (as described in [Riguer03] and [Scheuermann04]), enable high dynamic range rendering by providing a tone mapping step in the end of the scene processing, various image processing for artistic effects (some examples of post processing in a game environment are covered in chapter 7 of this course). In the Toyshop demo we used the flexible post-processing pipeline available in our engine to approximate atmospheric effects such as misty glow due to light scattering, to perform tone mapping for HDR rendering and for a variety of specific blurring effects for creation of rain effects.

3.3.1 Creating appearance of misty glow due to inclement weather

Water particles in the atmosphere during the rain increase the amount of light scattering around objects. Multiple scattering effects are responsible for the appearance of glow around light sources in stormy weather ([Van de Hulst81]). In order to approximate the effect of halos around bright light sources, we make use of the post-processing pipeline available in our engine and controllable through the rendering script. See Figure 12 below for an example of misty glow in our environment.

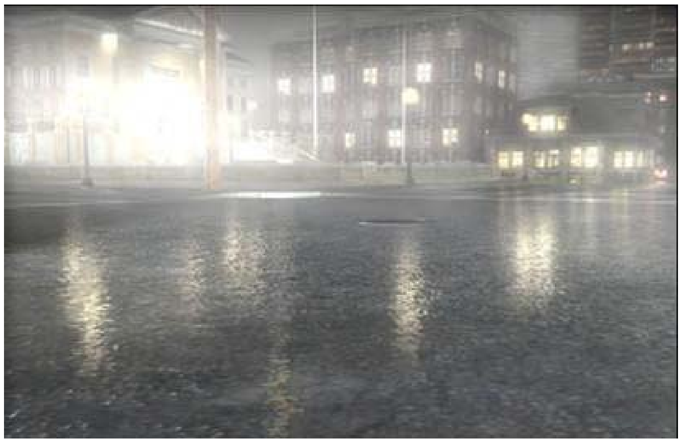


Figure 12. *Misty glow in the ToyShop environment*

To approximate the atmospheric point spread function which can be used to model multiple scattering around the light sources in the stormy weather (as in [Narasimhan03]), we use the Kawase post-processing approach for rendering glows in our scene ([Kawase03]). The main concept lies in blurring the original image to create the glow halos around the objects and bright light sources. Blur is a ‘magic’ tool: it adds softness to the scene, and successfully hides some artifacts (similar to the depth of effects).

First we render our environment into an offscreen buffer, where the alpha channel is used to specify the amount of glow for each pixel. Since we use 10-10-10-2 buffers for rendering, we only use 2 bits of alpha for glow amount. This is not ideal for many scenes; however, with sufficient attention to details for material rendering we are able to achieve very good quality of the resulting effects even with just mere two bits of information and clever usage of blending states. Once the scene is rendered into an offscreen buffer (using 10-10-10-2 format), we downsample the rendering by a quarter in each dimension (giving a total of ¼ x ¼ = $^1/_{16}$ reduction). We apply small blur filters (shown in Figure 13 below) repeatedly to the downsampled image, performing four feedback ping-pong passes for computing blurring.

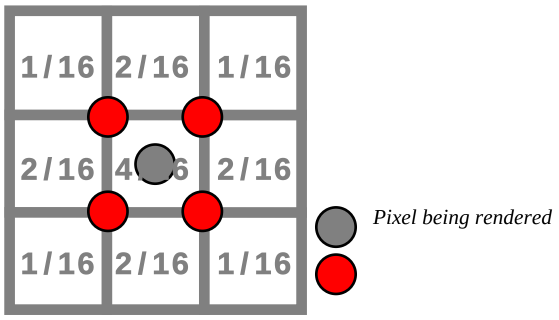


***Figure 13**. Kawase bloom filter. The weights for each sample are provided. (from [Kawase03]*

Each iteration of blurring ‘ping-pongs’ between two renderable textures used for storing the intermediate results. Each successive application of the bloom filter to the downsampled image takes the previous results as input and applies a new, larger kernel (as illustrated in Figure 14) to increase blurriness. The final blurring result is combined as described in [Kawase03]. More iterations will allow higher levels of blurriness; but we determined empirically that four passes give good visual results.

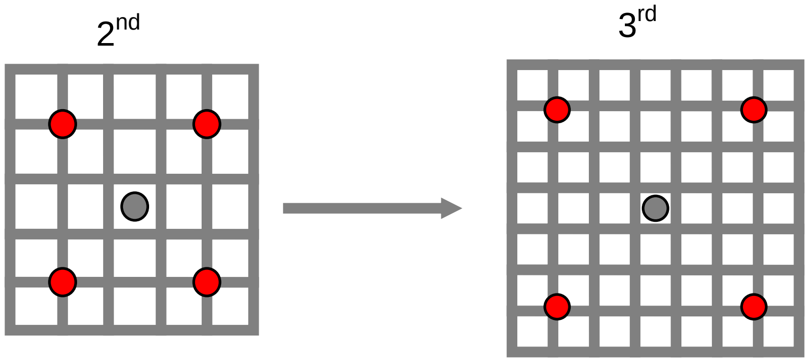


***Figure 14**. Two successive applications of the bloom filter on a texture grid. (from [Kawase03]*

To model fog attenuation due to water scattered in the atmosphere we implemented light attenuation based on distance in shaders. We attenuate the light information based on distance in shaders. In the vertex shader (Listing 1) we compute the distance of the object to the observer and then compute the linear fog value which is then sent to the interpolator for rasterization.

```
float4x4 mViewFog;
float2   vFogParams;

float ComputeFogFactor(float4 vWorldPos)
{
   // Compute distance to eye
   float4 vViewPos = mul (vWorldPos, mViewFog);
   float fDepth = sqrt(dot(vViewPos.xyz, vViewPos.xyz));

   // Compute linear fog = (d - end) / (end - start)
   float fFog = (fDepth - vFogParams.x) /
                 (vFogParams.y - vFogParams.x);
   fFog = saturate(fFog);

   return fFog;
}
```

***Listing 1.** Vertex shader fog segment*

In the pixel shader (Listing 2), we use the computed and interpolated fog value to attenuate pixel color value before tone mapping.

```
float3 cFogColor;
float4 vFogParams;

float4 ComputeFoggedColor(float3 cFinalColor, // Pixel color
                          float glow, // Glow amount
                          float fFog) // Vertex shader computed fog
{
   float4 cOut;

   // Foggy factor
   float fogFactor = fFog * (1-(SiGetLuminance(cFinalColor)/10));
   fogFactor = min (fogFactor, vFogParams.z);

   // First figure out color
   cOut.rgb = lerp(cFinalColor, cFogColor, fogFactor);

   // Then alpha (which is the glow)
   cOut.a = lerp(glow, fogFactor*vFogParams.w + glow, fogFactor);

   return cOut;
}
```

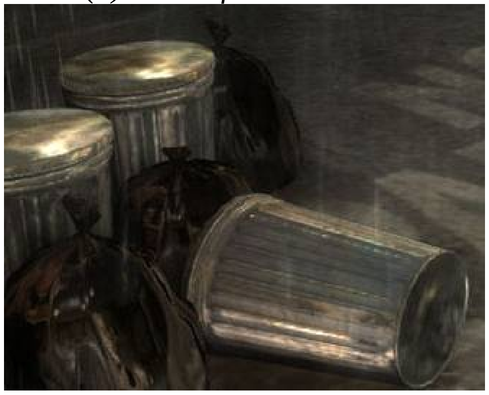
***Listing 2.** Pixel shader fog segment*

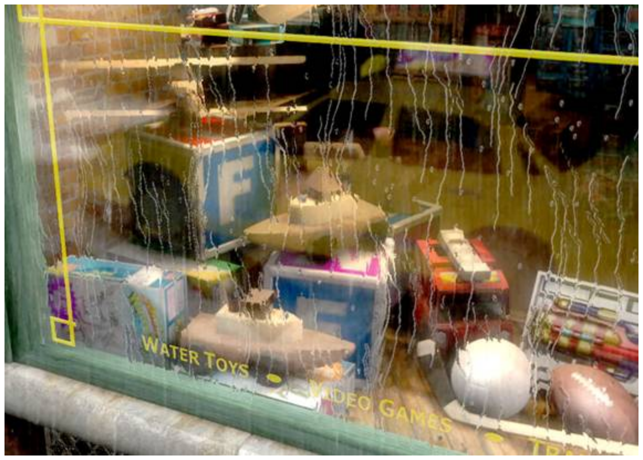
3.5 Wet reflective world

Realistic streaky reflections increase the feel of rain on wet streets and various object surfaces. These reflections are very prominent in any rainy scene and appear to stretch toward the viewer. Wet environments display a great deal of reflectivity – without realistic reflections the illusion is broken. Therefore, adding convincing reflections is a must for any rainy environment. To simulate the appearance of a wet city street in the rainy night, we render a number of various reflection effects in our scene:

(a)Wet surface materials

(b) Glass reflections of the store from the inside


(c)Wet metallic objects



(d) Glass reflections from the outside as well as raindrop reflection

Figure 15. *A selection of reflection effects in the ToyShop environment.*

- Stretchy warped water reflections in the street, puddles and other wet surfaces (Figures 16b, 16c)
- Various wet surface materials (wet granite, pavement, cobblestones, plastic, metal grates, etc) (Figures 15a and 15c above)
- All of the rain effects used reflection and refraction effects (see section 3.6) (Figure 15d)
- The inside of the toy shore and the outside scene reflected in the glass panes of the store windows (Figures 15b and 15d)
- The drenched taxi cab turning around the corner displayed dynamic reflections of the scene around it (Figure 16a)

Depending on the polygonal properties of a particular object, highly specular surfaces can display a great deal of aliasing if one is not careful. We dedicated a significant amount of effort to ensuring that these artifacts are reduced, if not completely removed, from our interactive environment. The solution was to attenuate both reflection illumination and specular highlights at the objects’ edges using a Fresnel term of varied powers.

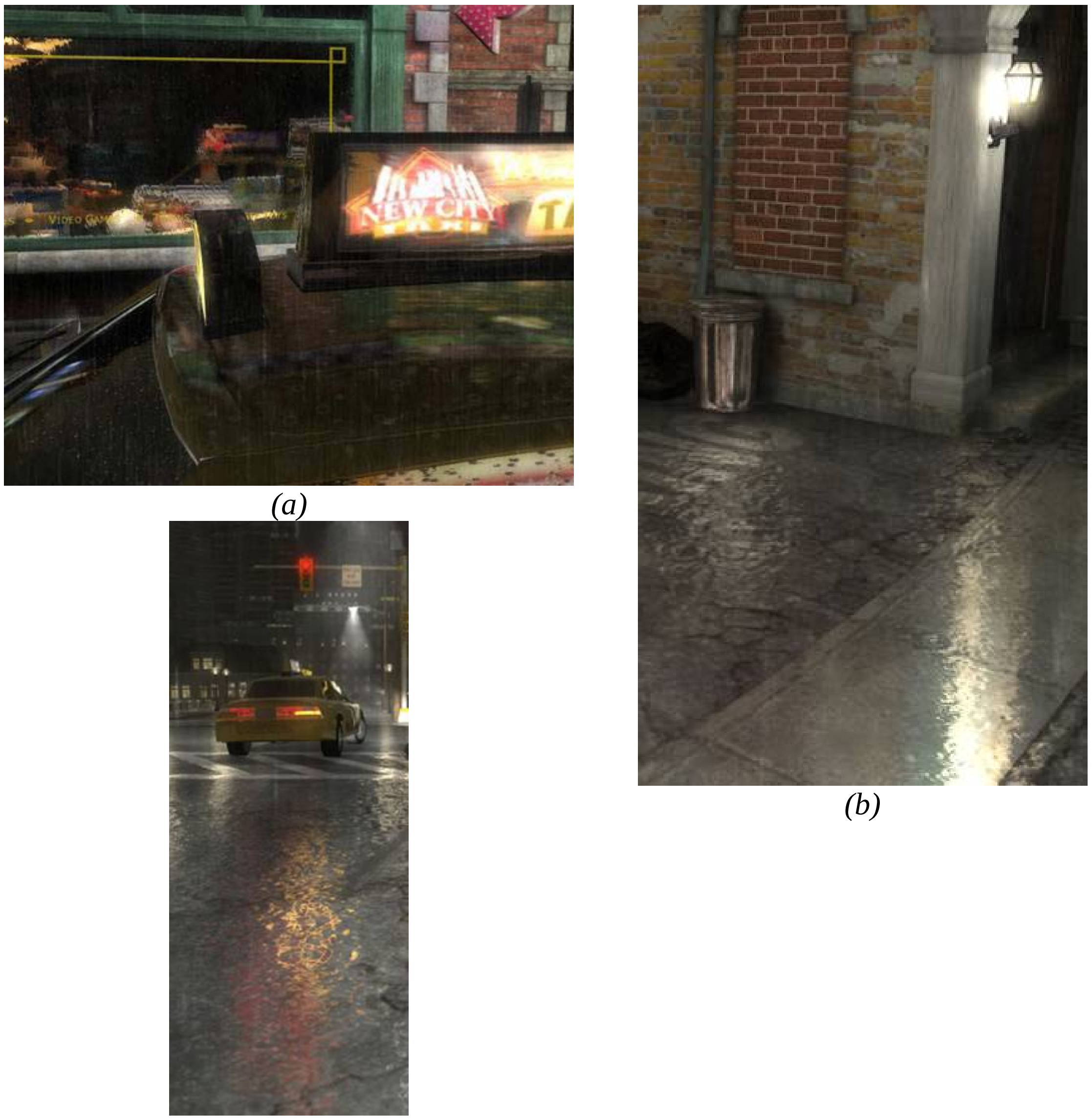


Figure 16.*Dynamic reflection effects for rendering a wet taxi cab (a) and streets (b) in our interactive environment.*

3.5.1 View-Dependent Streaky Reflections

When moving around any city streets late night during a rain, one of rain's strongest visual cues are the stretched reflections of bright light sources (such as street lamps, cars lamps, and store signs in the streets and sidewalks). These reflections are very prominent in any rainy scene. They appear to stretch very strongly toward the viewer, distorting the original reflecting object vertically proportional to the distance from the viewer. Water in the puddles and on the streets further warp the reflections, increasing the feeling of wetness in the environment (especially during the actual rain, the falling raindrops hitting the puddles create dynamic warping of the reflections). It is also easy to notice that these types of reflections are strongly saturated for bright light sources.

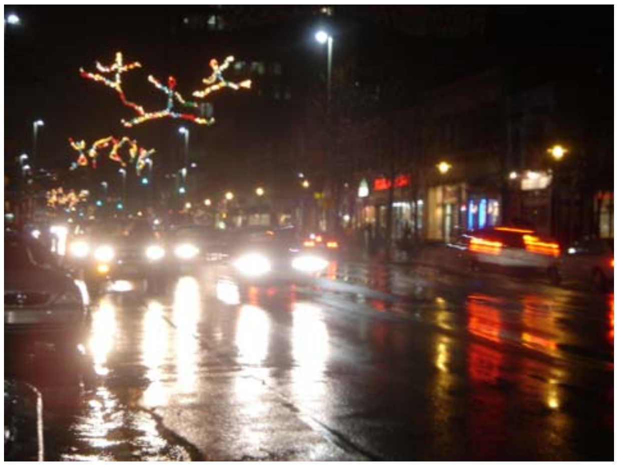


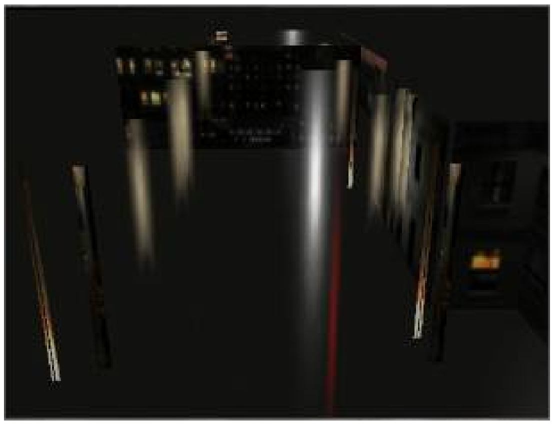
Figure 16. *Real-life photograph of a rainy night in Central Square, Cambridge, MA.*

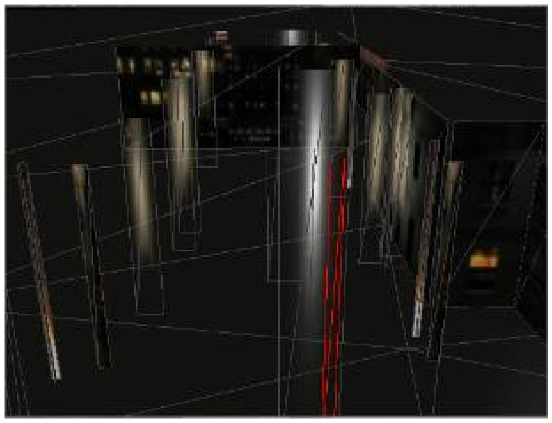
A good example of real-life scene during the rain in Central Square in Cambridge is in Figure 16 on the right. There we see a number of store signs, car head and tail lights and street lights reflected in the street. Notice that the original shape of each reflector is only distinguishable by the blurred dominant colors (such as the reddish-orange glow of the taxi tail lights or the nearly white blobs of cars headlight reflections. Similarly, we want to preserve the brightest principal colors and create a blurry glowing reflection image for each light source or bright reflecting object in our scene.

Realistic streaky reflections increase the feel of rain on wet streets and surfaces. In our environment we create reflections for all bright objects onto the paved streets and large flat surfaces, such as the rooftop ledge (see figure 18 for examples of reflections in our interactive scene). All objects that can be viewed as reflectors are identified as such by the artists a priori. Examples of the bright reflector objects in our environment are the neon lights (such as the toy shop sign, street and building lamps (such as the lamp on the rooftop), the car head and tail lights, and bright building windows). Note that we render both bright *light* objects (such as street lamps), as well as the dark objects (such as the telephone poles and wires) (their colors are deepened).

Rather than simply rendering these objects directly into the reflection buffer as they are in the final rendering pass, we improve performance by rendering proxy geometry into the reflection buffer instead. For each reflector object the artists generate a quad with a texture representing the object, which is slightly blurred out (since these reflections tend to be blurry in the end) (see Figure 17a). Note that we aren't simply rendering the unlit proxy object texture into the reflection buffer. At run-time this proxy object is lit in a similar fashion to the original object (to make sure that the reflections appear to respond correctly to the environment lighting) during rendering into the reflection buffer. The reflection shader uses a simplified lighting model to preserve dominant colors, but does not waste performance on subtle effects of a particular material. This dynamic lighting allows us to represent reflected animated light sources (such as the flickering neon sign of the shop or the blinking traffic lights on the streets) correctly in the street reflections (which dim and light in sync with their corresponding reflector objects).

The proxy reflection objects are dynamically stretched view-dependently toward the viewer in the vertex shader (you can see the reflection quad objects with wireframe displayed in Figures 17a and 17b). The amount of stretching varies depending on the distance of the object to the viewer.


(a) Bright reflector objects rendered into the reflection buffer


(b) Overlaid wireframe proxy reflector objects' quads

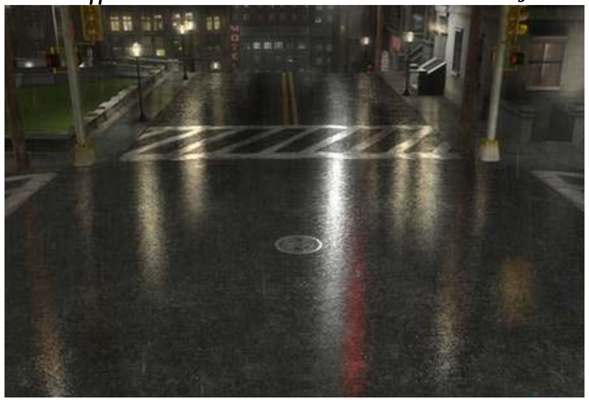

(c) Resulting scene using the above reflection buffer after processing

Figure 17. *View-dependent streaky reflection rendering*

One aspect that we want to mention for using the proxy objects is the issue of culling the objects if the original reflector objects are no longer in the view. Since the proxy objects are only rendered into the offscreen reflection buffer, they do not go through the visibility culling process in our rendering engine. Therefore, we ran into a situation where the taxi cab, turning around a corner, would disappear from the view, but even a few seconds later we could still notice the stretchy red tail light reflections. To work around this problem, we place separate reflector blocker objects, which act to hide the proxy reflector objects from rendering into the reflection objects if the original reflector object is no longer in view.

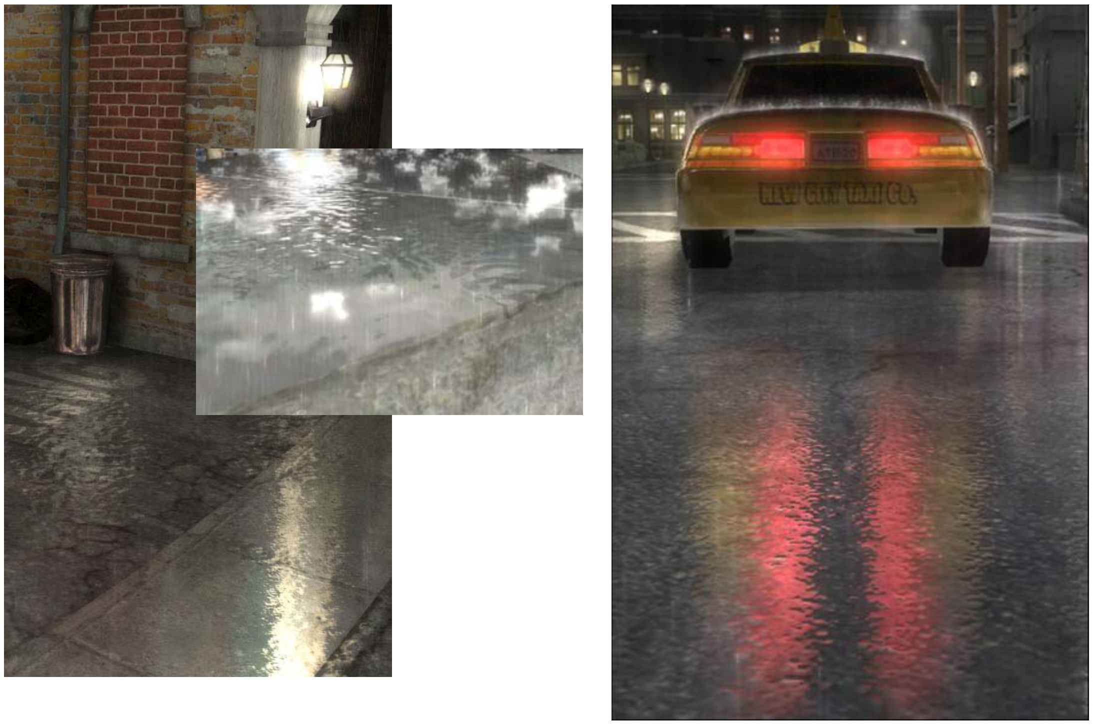


Figure 18. *View-dependent streaky reflections in the ToyShop demo*

For performance reasons the reflection buffer is scaled to be half size of the original back buffer (and a separate quarter sized reflection buffer for the rooftop reflections). We utilize an expanded dynamic range for representing the rendered colors so that we can preserve the brightest and darkest colors for reflections (such as street lamps or taxi headlights or telephone poles) by using the 10-10-10-2 HDR format for the reflection buffer.

Next we need to address the issue of making these view-dependent reflections appear blurry, glowing and streaky. For that, we turn to the post-processing system already in place (as described in section 3.3.1). We use a post-processing technique to dynamically streak the reflection buffer in the *vertical* direction only to simulate warping due to raindrops striking in the puddles. Note that this is done in separate passes from the regular scene post-processing.

During rendering of the final scene prior to post-processing, we sample from the reflection buffer using screen space projection of the input vertex coordinate for each reflective material (such as the street pavement or the roof ledge granite, see Figure 18). Reflections are also distorted based on the normals of the surface they pass through. We use object’s per-pixel normal in tangent space to account for stretching of the reflection in view space and warp the reflection based on the surface normal. The post-process-based blurring and this warping aid in removing specular aliasing and excessive flickering from reflections which would otherwise be highly distracting.

Since the number of draw calls in an interactive rendering application is an example of a typical rendering bottleneck of many games, we paid particular care to their optimization. Given the sheer number of various effects we designed to implement in our environment, we had very strict requirements for performance tuning and tried to save every percent of the frame rendering time. Since we rendered a large number of reflector objects, the goal was to render them in a single draw call. This was accomplished by specifying their world position in the skinning matrix using only a single bone. Therefore all objects with similar materials (such as the telephone poles or the street lamps) were rendered as one single big object using skinning to position them around the scene.

3.5.2 Dynamic reflections for a reflective taxi

While the taxi cab is moving through the streets of our interactive city, the environment is reflected in its metallic and glass surfaces (Figure 16a). We implement these reflections through the environment map reflection method (see [Akenine-Möller02], pages 153-166 for more details). In order to generate dynamic reflections, we render our environment into a cubemap with the camera placed at the center of the taxi cab as its moving through the scene. This dynamic cubemap is used for reflection color lookup for the cab surfaces (Figure 19 below shows an example of the contents of this cubemap).

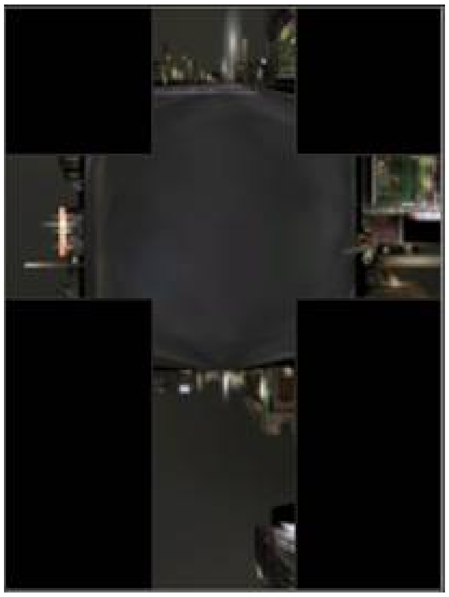


Figure 19. *An example of the dynamically rendered environment map for taxi cab reflections.*

Using the rendering script allows us to only render the environment cubemap for frames when the taxi was actually moving. At the same time, note that the environment gets rendered 6 times (for every face of the cubemap), so rendering the full scene is suboptimal. To improve that, we build a low resolution 'billboard' version of the city environment. We place the billboard quads along the taxi cab path. The quads contain textures of the buildings and environment as viewed from the point of view of the cab. These textures are created by taking in-engine snapshots by placing the camera on the taxi cab path while rendering the full scene (figure 20 contains an example of this billboard texture atlas). Similar to the approach for rendering reflector objects in section 3.5.1, we light these quads dynamically to get more accurate reflections in the final rendering. However, rendering just the billboard quads into the faces of the cubemap (rather than the full geometry) saves a great deal on performance. Instead of using the billboard versions of the environment, another suggestion for in-game rendering would be to use one of the lower levels of details of the scene, if the game contains support for level-of-detail rendering.

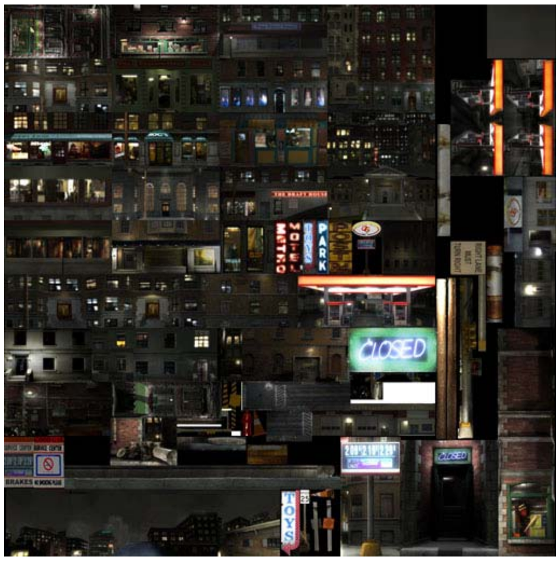


Figure 20. *Billboard dynamic reflector texture atlas*

3.6 Rendering rain

Rain is a complex visual phenomenon. It is composed of many visual components. Rainfall consists of specially distributed water drops falling at high velocity. Each individual drop refracts and reflects the environment. As the raindrops fall through the environment, they create the perception of motion blur and generate ripples and splashes in the puddles. Rain effects have been extensively examined in the context of atmospheric sciences ([Wang75] and [Mason75]), as well as in the field of computer vision ([Garg04]). We developed a number of effects for rendering rain in our interactive environment in real time, consisting of a compositing effect to add rainfall into the final rendering, a number of particle-based effects and dynamic water effects, simulated directly on the GPU.

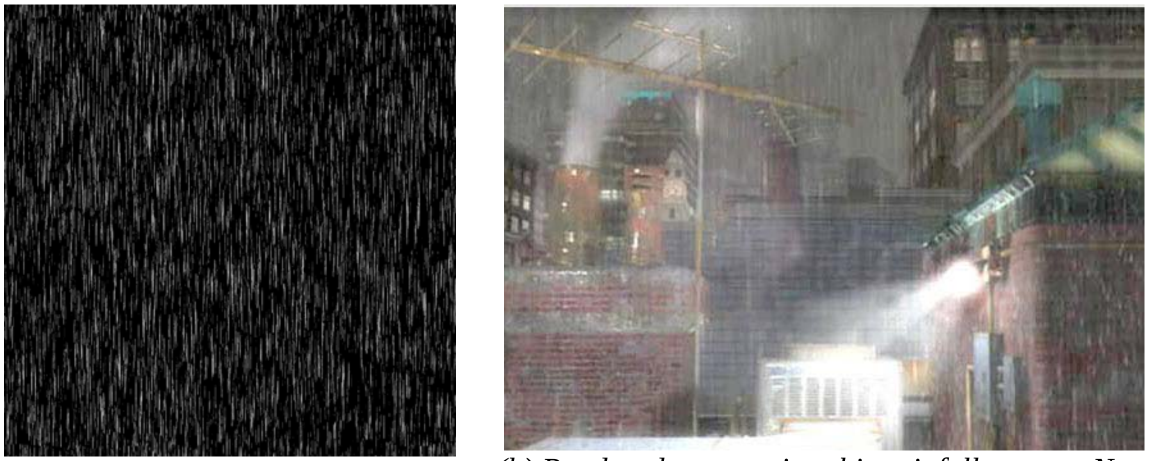
3.6.1 Rendering multiple layers of rain with a post-processing composite effect

We developed a novel post-processing rain effect simulating multiple layers of falling raindrops in a single compositing pass over the rendered scene. We create motion parallax for raindrops utilizing projective texture reads. The illumination for rain is computed using water-air refraction for individual raindrops as well as reflection due to surrounding light sources and the Fresnel effect. We provide a set of artist knobs for controlling rain direction and velocity, and the rainfall strength. The raindrop rendering receives dynamically-updated parameters such as lightning brightness and direction from the lightning system to allow correct illumination resulting from lightning strikes.

Creating rainfall We render a composite layer of falling rain as a full-screen pass before the final post-processing of the scene. Rainfall is simulated with an 8 bit texture (see

Figure 20 for an example texture and the resulting rain scene). To simulate the strong mistiness of the raindrops, we blur the rain by using the post-processing system (as described in section 3.3.1). The artists can specify the rain direction and speed in world-space to simulate varied rainfall strength.

Although so far this approach sounds rather straight-forward, there are several challenges with rendering rain through a composite layer. The first difficulty lies in minimizing repeating patters that are inevitable when using a single static texture to model dynamic textured patterns. The second concern lies with the consideration that the rain pass is a full-screen pass, and therefore every pixel on the screen will go through this shader. This has direct effect on performance, and we must design the composite rain rendering such that it gives pleasing visual results without an expensive shader.


(a) Rainfall texture

(b) Rendered scene using this rainfall texture. Note that the image intensities have been brightened for better contrast since this is a static capture of rain.

Figure 20. *Rainfall texture applied for a composite rain effect in the interactive scene*

Computer vision analysis of rain models ([Garg04]) and video rain synthesis ([Starik03]) helps us to observe that one cannot easily recognize rainfall from a single static frame; however, rain is easily noticeable in a dynamic simulation or a video. Perceptual analysis of rain video shows that the individual raindrop motion cannot be tracked by human perception accurately due to swift movement and density of raindrops, which allows us to assume temporal independence of rain frames. However, our empiric experiments showed that purely random movement of raindrops does not yield satisfactory results (generating excessive visual noise). Therefore to simulate strong rainfall, we simultaneously use the concepts of individual rain drop rendering and the principles stochastic distribution for simulation of dynamic textures (as in [Bar-Joseph01] and [Doretto03]).

The first part of our algorithm simulates individual rainfall movement. The artist-specified rain direction vector is moved into clip space. We use this vector to determine a raindrop position in screen space by using the current position in clip space, specified rainfall velocity and current time. Given these parameters and computed the raindrop position, we can scroll the rainfall texture using the specified velocity vector. However, although texture scrolling is a very straight-forward approach, even with several texture fetches in varied directions with slight randomization, repeating rain patterns become rather obvious in a full-screen pass.

Multiple layers of rain Our goal is to simulate several layers of raindrops moving with different speeds at varied depths in a single rendering layer. This better approximates real-life rain movement and allows us to create a feeling of raindrop motion parallax (a strong visual cue in any dynamic environment). The artists can specify a rain parallax parameter which provides control for specifying the depth range for the rain layers in our scene. Using the concepts of stochastic distribution for simulation of dynamic textures, we compute a randomized value for an individual raindrop representation to use in the rain shader. Using the rain parallax value, the screen-space individual raindrop parameter and the distribution parameter, we can model the multiple layers of rain in a single pass with a single texture fetch. This allows us to simulate raindrops falling with different speed at different layers. The rain parallax value for the rain drop, multiplied by a distribution value, can be used as the *w* parameter for a projective texture fetch to sample from the rainfall texture. Note that we use a single directional vector for all of our raindrops which is crucial for creating a consistent rainfall effect. This creates excellent visual effects of random streaking for the raindrops.

Rain appearance Given a moving raindrop, we need to shade it. Raindrops behave like lenses, refracting and reflecting scene radiances towards the camera. They refract light from a large solid angle of the environment (including the sky) towards the camera. Specular and internal reflections further add to the brightness of the drop. Thus, a drop tends to be much brighter than its background (the portion of the scene it occludes). The solid angle of the background occluded by a drop is far less than the total field of view of the drop itself. In spite of being transparent, the average brightness within a stationary drop (without motion-blur) does not depend strongly on its background.

Falling raindrops produce motion-blurred intensities due to the finite integration time of a camera. Unlike a stationary drop, the intensities of a rain streak depend on the brightness of the (stationary) drop as well as the background scene radiances and integration time of the camera. We simulate the motion blur for the raindrops by applying blurring via post-processing after the rain pass has been blended onto the scene rendering. This simulates both raindrop motion-blur and multiple-scattering glow for individual raindrops. To shade an individual raindrop, we use a tangent-space normal map corresponding to the rainfall texture. Note that since this is a full-space pass, the tangent space is simply specified by the view matrix. For each pixel in the rain pass, we compute reflection based on the individual raindrop normal and air-to-water refraction. Both are attenuated toward the edges of the raindrop by using the Fresnel effect.

Raindrop transparency An interesting observation is that as the lightning strikes, the raindrops should appear more transparent. In other words, the opacity of each individual raindrop must be a function of the lightning brightness; otherwise water surfaces appear too solid. As mentioned in section 3.3, our rendering script propagates the lightning system parameters to all of our rain shaders, as well as the material shaders. For the raindrop rendering, we use a combined lightning brightness parameter (mixing both lightning 'light sources' as they flash in the environment) to compute the bias value to adjust the amount of reflection and refraction.

Realistic rain is very faint in bright regions but tends to appear stronger when the light falls in a dark area. If this is modeled exactly, the rain appears too dim and unnoticeable in many regions of the scene. While this may be physically accurate, it doesn't create a perception of strong rainfall. Instead of rendering a precise representation, we simulate a Hollywood film trick for cinematic rain sequences. The film crew adds milk to water or simply films milk to make the rain appear stronger and brighter on film. We can bias the computed rain drop color and opacity toward the white spectrum. Although this may seem exaggerated, it creates a perception of stronger rainfall

Compositing rain via blending We would like to make a few notes on specifying the blending for the rain pass. The rain layer is rendered both as a transparent object and a glowing object (for further post-processing). However, since we wish to render the rain layer in a single pass, we are constrained to using a single alpha value. Controlling both opacity and glow with a single alpha blending setting can be rather difficult. Despite that, we want to render transparent objects that glow, controlling each state separately for better visual results. We found that we can use two sets blending parameters to control blending for glow and for transparency for all rain effects all rain effects (composite rain, raindrops, splashes). In the latest DirectX9.0c there is a rendering state for separate alpha blending called `D3DRS_SEPARATEALPHABLENDENABLE`. Using this state along with the regular alpha blending function (via `D3DRS_ALPHATESTENABLE`) allows us to specify two separate blending functions for the regular opacity blending and for the alpha used for glow for post-processing blurring pass.

Finally, we would like to mention a few other considerations for including this composite post-processing rain layer effect in other interactive scenarios such as games. In many extensive environments which may include changing weather conditions as well as the changes between outdoor and indoor locations, the issue of controlling composite rain rendering can appear challenging. In reality it is not so – there is a number of ways to efficiently accomplish that goal. In our interactive scene, we use the rendering script to determine whether the camera is located inside the toy store or whether it is outside. This information is used to dynamically turn off composite rain rendering. A similar concept (an engine state specifying what environment the camera is located, for example) can be used in many game setups. Likewise, an engine state that specifies the current weather condition can be used to control rain rendering by turning on and off rendering of the rain quad. If there is no notion of the appropriate engine state, another approach may involve using a sky visibility overhead lightmap (see section 3.6.3 for more on overhead lightmap). One can encode a Boolean sky visibility value (precomputed for the entire environment at preprocessing time for every point in the scene, similar to the overhead lighting lightmap used in section 3.6.3 for raindrop splashes lighting). This value can be used directly in the rain quad pixel shader to turn off rendering pixels based on the current camera location. However, we would like to note that this approach is far less efficient than the rendering script-based control of rain rendering.

3.6.2 Raindrop Particles Rain

To simulate raindrops falling off various objects in our scene, we use screen-aligned billboard particle systems with normal-mapped rain droplets (Figure 22a and 22b). In our scenes we found that using on the order of 10-15,000 particles gives excellent results.

We created a base template particle system that uses the physical forces of gravity, wind and several artist-animated parameters. The artists placed a number of separate particle systems throughout the environment, to generate raindrops falling off various surfaces, such as rooftop ledge, building lamps, and so on, onto the streets.

To render an individual raindrop particle, we stretch the particle billboard based on the particle velocity (with slight randomization offsets to vary velocity per individual particle within a particle system). The illumination model used for these particles is similar to that of the composite rain layer. We use a normal map for a water droplet for each individual raindrop. Instead of using an accurate droplet-shape representation, we pre-blurred and stretched the drop normal map to improve the perception of motion blur as the raindrops move through the environment (Figure 21). Note that the tangent space for a billboard particle is defined by the view matrix.

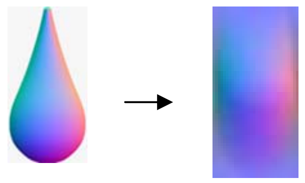


***Figure 21**. Pre-blurred droplet normal map (on the right)*

To shade the raindrop particle, we only compute specular reflection and air-to-water refraction effects, using the pre-blurred normal map. Since droplet should appear more reflective and refractive when a lightning flashes, biased lightning brightness value adjusts the refraction and reflection color contributions.

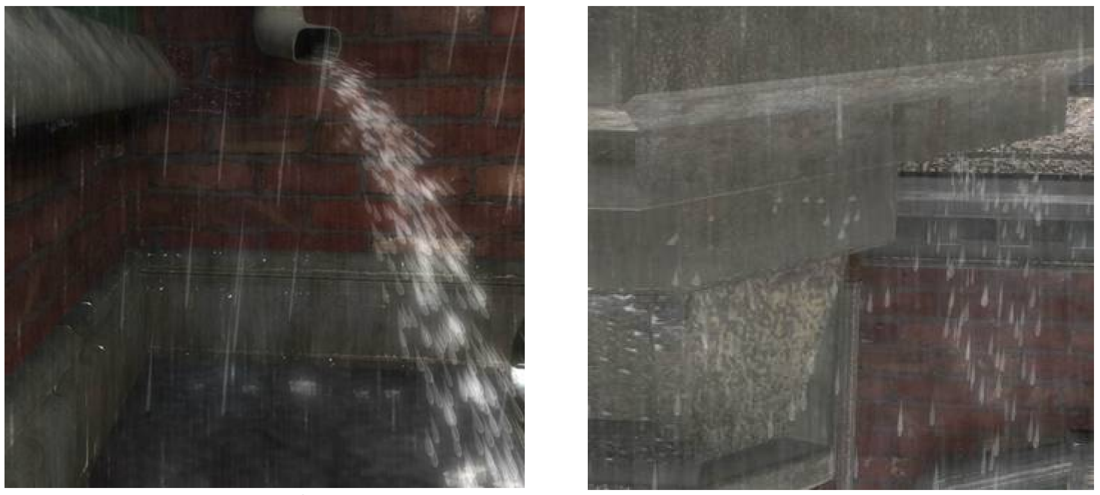

(a)Raindrops pouring from a gutter pipe *(b) Raindrops falling off the rooftop ledge*

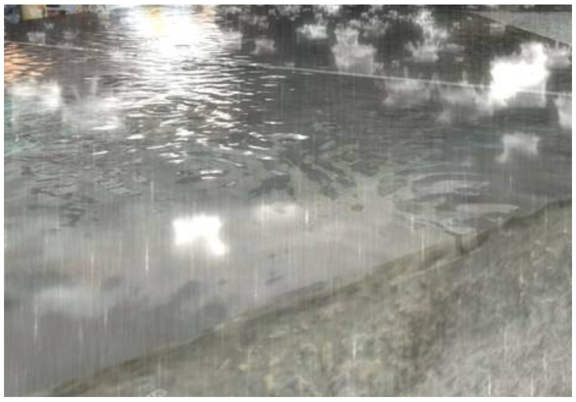
***Figure 22.** Raindrops falling off objects in our environment.*

To control raindrop transparency, we attenuate raindrop opacity by its distance in the scene. We wish to make the individual raindrop particles appear less solid and billboard-like as they move through the environment. This can be accomplished by attenuating the particle opacity value by Fresnel value, scaled and biased by two artist-specified parameters for droplet edge strength and bias (which could be specified per particle system). We used the observation that the raindrops should appear more transparent and water-like when the lightning strikes, and increased the raindrop transparency as a function of the lightning brightness to maintain physical illusion of water. This can be easily done by biasing droplet transparency by 1 – ½ * *lightning brightness*. The particles still maintain their artist-specified transparency in the regular lighting without any lightning flashes. We used this approach for both regular raindrop rendering and for raindrop splash rendering.

3.6.3 Rendering raindrop splashes

We simulate raindrops splashing when hitting solid objects by colliding individual particles with objects in the scene (Figure 23c). In our system we use special collider proxy objects. In a different engine environment this may be done by colliding particles directly with game objects. We used on the order of 5-8,000 particles to render raindrop splashes each frame. Figures 23a and 23b show an example of raindrop splashes rendered with regular illumination (a) and lit by a lightning flash on the rooftop ledge (a).

(a) Raindrop splashes on the rooftop ledge


(b) Raindrop splashes on the rooftop ledge lit by a lightning flash

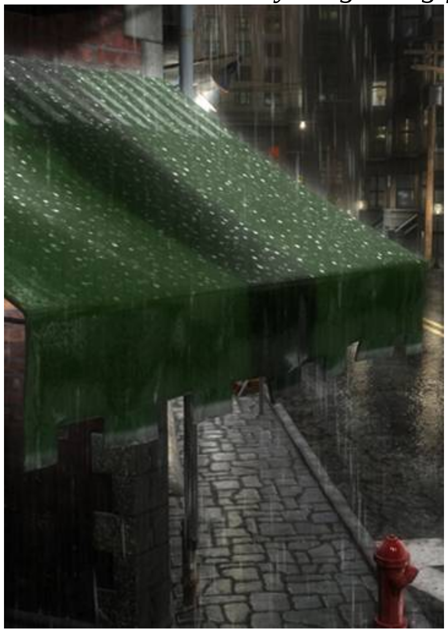

(c) Raindrop splashing hitting the store awning

Figure 23. *Raindrop splash rendering*

To shade these splashes, we use a pre-rendered high-quality splash sequence for a milk drop (Figure 24). A single filmed high-quality splash sequence for a milk drop was used to drive the raindrop splash event for *all* of the thousands of splashing raindrops. The challenge lied in reducing the noticeable repetition of the splash animation (especially considering that viewer could get rather close to the splashes). To address this concern we incorporated a high degree of randomization for particle parameters (particle size and transparency), and dynamically flipped horizontal texture sampling for the filmed sequence based on a randomly assigned particle vertex color.

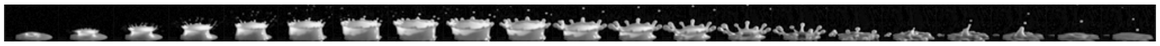


Figure 24. *Milk drop sequence for raindrop splash animation*

Splashes should appear correctly lit by the environment lights. We added backlighting to the splashes so that they accurately respond to the environment lights (and thus display the subtle effects of raindrops splashing under a street light). If light sources are behind the rain splashes, we render the splash particles as brightened backlit objects; otherwise we only use ambient and specular lighting for simplicity. We compute specular lighting for all available dynamic lights in the vertex shader for performance reasons.

Aside from the dynamic lights, we wanted to simulate the splashes lit by all of the bright objects in the environment (such as street lamps, for example), even though those objects are not actual light sources in our system. Using a special 'overhead' lightmap let us accomplish that goal (see Figure 25 for an example). We can encode the light from these pseudo light sources into a lightmap texture to simulate sky and street lamp lighting. We can then use the splash world-space position as coordinates to look up into this lightmap (with some scale and bias). The overhead lightmap value modulates otherwise computed splash illumination.

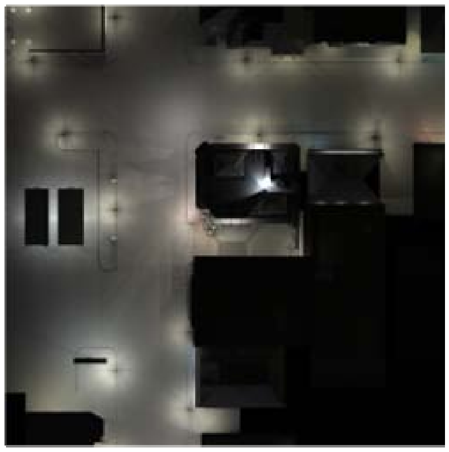


Figure 24. *Overhead lightmap example*

3.7 GPU-Based water simulation for dynamic puddle rendering

The raindrop particle collisions generate ripples in rain puddles in our scene. The goal was to generate dynamic realistic wave motion of interacting ripples over the water surface using the GPU for fast simulation. We use an implicit integration scheme to simulate fluid dynamics for rendering dynamically lit puddle ripples. Similar to real-life raindrops, in our system we generate multiple ripples from a single raindrop source which interact with other ripples on the water surface. The physics simulation for water movement is done entirely on the GPU. We treat the water surface as a thin elastic membrane, computing forces due to surface tension and displacing water sections based on the pressure exerted from the neighboring sections. Our system provides simple controls to the artists to specify water puddle placement and depth. Figure 25 shows water puddle on the rooftop and in the streets using our system.

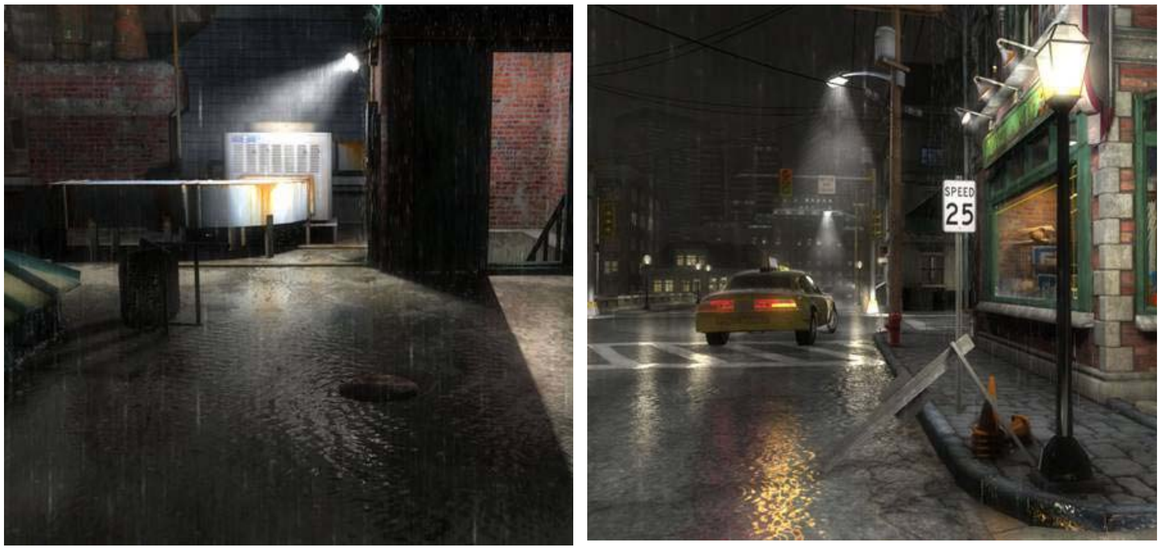


Figure 25. *Dynamic puddles with ripples from rain drops*

Water ripples are generated as a result of raindrops falling onto the geometry in the scene. This can be a direct response from the actual raindrop particle system colliding with the scene objects. In our implementation we approximated this effect by stochastically rendering raindrops into a 'wave seeding' texture. In the case of direct particle response, the approach is similar; however, the initial wave texture must be rather large to accommodate raindrops falling throughout the entire environment. In order to conserve memory, we decided against that approach, and limited our simulation to 256 x 256 lattice. Raindrop seeds are rendered as points into the water simulation texture, where the color of the raindrop is proportional to its mass. The method can be extended to generate dynamic water surface response for arbitrary objects. This can be achieved by rendering an orthographic projection of the objects into the seeding texture, encoding object's mass as the color of the object's outline. This would generate a wake effect in the water surface.

We render the raindrop seeds into the first water simulation buffer in the first pass. These rendered seeds act as the initial ripple positions. They 'excite' ripple propagation in the subsequent passes. In the next two passes we perform texture feedback approach for computing water surface displacements. In our case two passes are sufficient for the time step selected. If a larger time step is desired, more passes or a more robust integration scheme may be selected (we use Euler integration). In the fourth pass we use the Sobel filter ([Jain95]) on the final water displacement heights texture to generate water puddle normals.

Real-life raindrops generate multiple ripples that interact with other ripples on the water surface. We implement the same model. We render a raindrop into a wave seed texture using a dampened sine wave as the function for raindrop mass. This approximates the concentric circular ripples generated by a typical raindrop in the water puddle.

We approximate the water surface as a lattice of points. Each lattice point contains the information about the water surface in that location. In particular we store the current position as a height value and the previous time step position (see Figure 26). Since we perform all of the water simulation computations directly on the GPU, the lattice information is stored in a 32 bit floating point texture (with 16 bits per each position channel). A related approach was described in [Gomez00] where the lattice of water displacements was simulated with water mesh vertices displaced on the CPU.

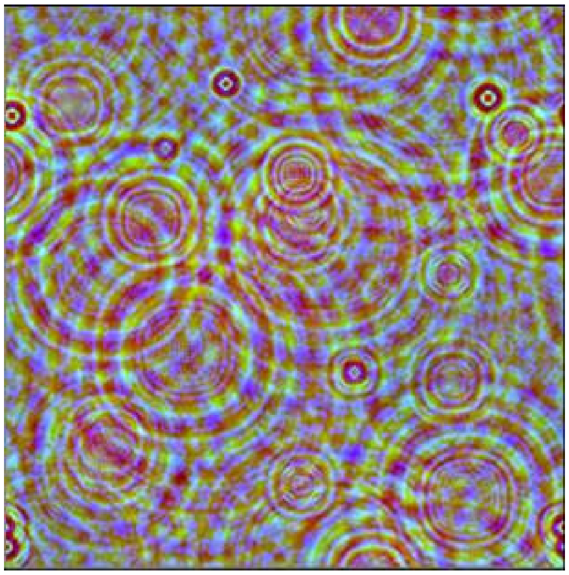


Figure 26. *Water displacements encoded into the feedback texture. The red channel contains water heights at current time step, and the green channel contains previous time step displacements.*

To compute water surface response we treat the water surface as a thin elastic membrane. This allows us to ignore gravity and other forces, and just account for the force due surface tension. At every time step, infinitesimal sections of the water surface are displaced due to tension exerted from their direct neighbors acting as spring forces to minimize space between them (Figure 27).

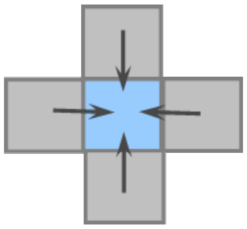


Figure 27. *Water neighbor cell acting on the current cell*

Vertical height of each water surface point can be computed with partial differential equation:

$$\frac{\partial^2 z}{\partial t^2} = v^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right)$$

where
z is the water displacement height,
v is the velocity of the water cell
x and y are the lattice coordinates of the water cell

This PDE is solved with Euler integration in DirectX9.0 pixel shaders in real-time by using the texture feedback approach to determine water wave heights for each point on the lattice.

Water puddles integration. We render a single 256 x 256 water simulation for the entire environment. Therefore we have to use a bit of cleverness when sampling from this simulation texture - since many different objects all use the same wave ripples simulation at the same time. We sample from the water membrane simulation using the object’s current position in world space, specifically the *xz* coordinates as a lookup texture coordinates into the wave normal map (Figure 28).

The artists control the sampling space per object with a scaling parameter that allows them to scale the size of water ripples at the same time (by essentially scaling the lookup coordinates). To reduce visual repetition for puddles, we rotate the water normals lookup coordinates by an angle specified per-object. Since we sample from the water normals texture when rendering an object with puddle, we do not require additional puddle geometry. It is even possible to dynamically turn water puddle rendering on and off by simply using a shader parameter and dynamic flow control. To render an object with water puddles, we perturb the original object's normal from a bump map with the normal from the water membrane simulation.

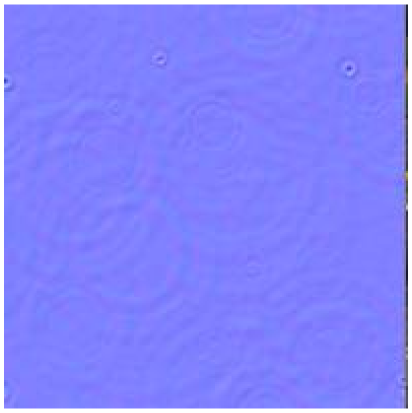


Figure 28. *Dynamic ripples normals*

The artists can also specify a puddle ripple influence parameter per object. This parameter controls how much the water ripple normal perturbs the original bump map normal. This allows us create different water motion for various objects.

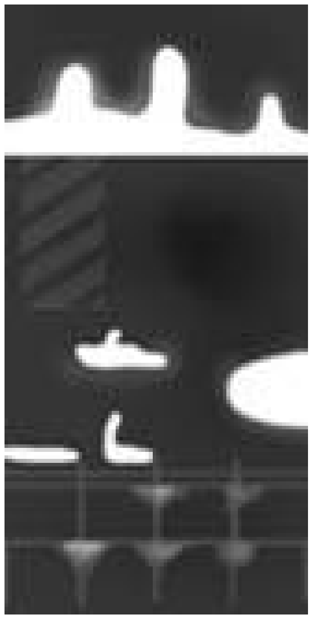
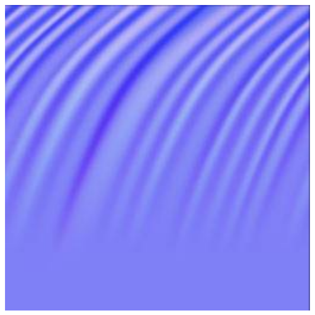


Figure 29. *Puddle depth and placement map.*

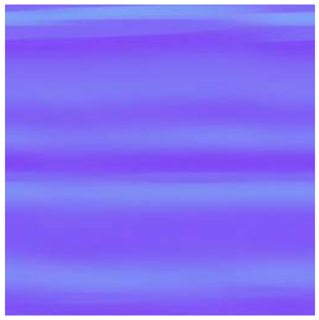
Puddle Placement and Depth To render deep puddles, we use just the water puddle normal sampled as just described, along with the color and albedo attributes of the object. We wanted to mimic varied puddle depths of the real-world and allow artists creative control over the puddle placement. A puddle depth mask was our answer (Figure 29 on the left). Adding puddles with ripples to objects is straight-forward:

- Define the ripple scale parameter and sample ripple normals using the world-space position
- Sample puddle depth map
- Interpolate between the object normal map and the water ripple surface normal based on the puddle depth value and artist-specified puddle influence parameter

Creating Swirling Water Puddle For the rooftop puddle, we want to create an impression of water, swiftly swirling toward the drain, with ripples from raindrops warping the surface (using the above approach). We used several wake normal maps to create the whirlpool motion. The first normal map (Figure 30a) was used to swirl water radially around the drain. Combined with it, we used the wake normal map from Figure 30b to create concentric circles toward the drain.



(a) Radial movement wake map



(b) Concentric circles wake map

Figure 30. *Wake normal maps*

3.8 Raindrop movement and rendering on glass surfaces in real time

We adopted the offline raindrop simulation system [Kaneda99] to the GPU to convincingly simulate and render water droplets trickling down on glass planes in real-time. This system allows us to simulate the quasi-random meandering of raindrops due to surface tension and the wetting of the glass surfaces due to water trails left by droplets passing on the window. Our system produces a correctly lit appearance including refraction and reflection effects.

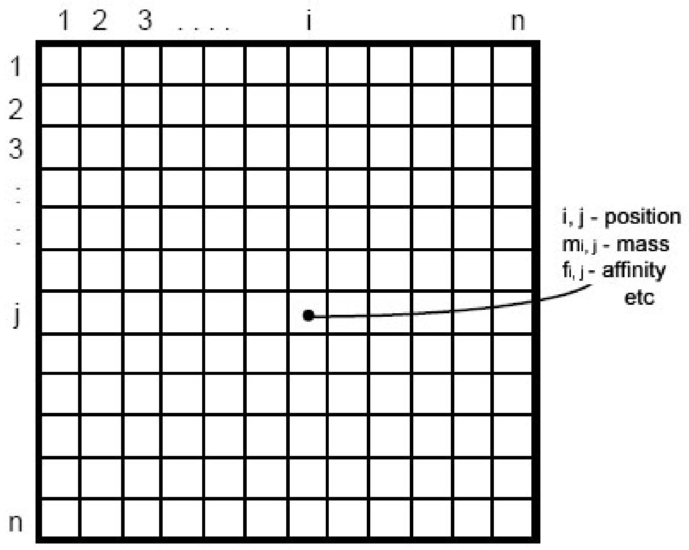


Figure 31*. Discrete lattice model for storing water droplet information at run-time*

Droplet movement. The glass surface is represented by a lattice of cells (Figure 31) where each cell stores the mass of water in that location, water x and y velocity, and the amount of droplet traversal within the cell. This information can be conveniently packed into a 16 bit per channel RGBα texture. Additionally we store droplet mass and affinity information for each cell as well.

The force of gravity depending on the mass of the droplet is used to compute the downward movement force on the droplet. Static friction for stationary droplets and dynamic friction for moving droplets is used to compute the competing upward force. The static and dynamic friction varies over the surface of the glass. This resultant force is applied to the initial velocity to determine the new velocity value for the droplet.

At any given time, droplets can flow into the three cells below its current cell. New cell for the flow is randomly chosen, biased by the droplet directional velocity components, friction based affinity of current cell and the ‘wetness’ of the target cell. The glass friction coefficients are specified with a special texture map. Droplets have a greater affinity for wet regions of the surface. We update the droplet velocity based on the selected cell.

Droplet rendering. First we render the background scene. Then we render the water droplet simulation on the window. This allows us to reflect and refract the scene through the individual water droplets. In order to do that, we use the water density for a given rendered pixels. If there is no water in a given pixel, we simply render the scene with regular properties. However, if the water is present, then we can use the water mass as an offset to refract through that water droplet. At the end of the droplet movement simulation, each cell contains a new mass value (Figure 32). Based on the mass values, we can dynamically derive a normal map for the water droplets. These normals are used to perturb the rendered scene to simulate reflection and refraction through water droplets on the glass surface (figure 33a).

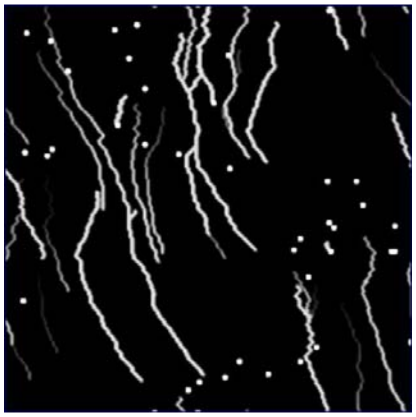
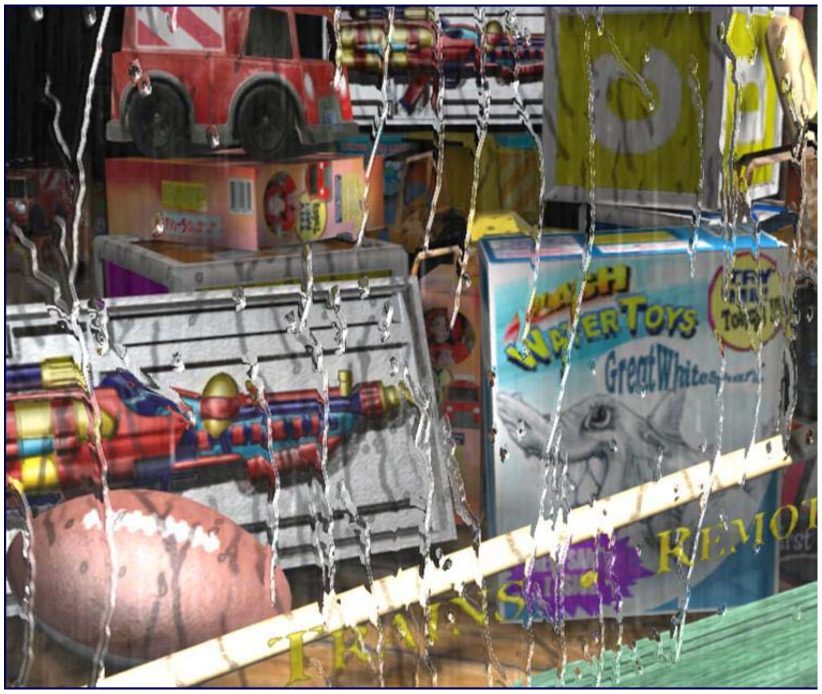
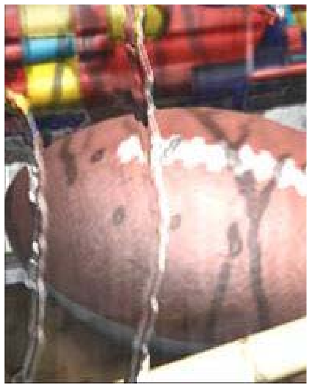


Figure 32. *Water droplet mass*

The droplet mass is also used to render dynamic shadows of the simulation onto the objects in the toy store (using the mass texture as a projective shadow for the other objects). If the droplet mass is large enough, we render a pseudo-caustic highlight in the middle of the shadow for that droplet (Figure 33b)



(a) Water droplet rendering on the glass window



(b) Bright pseudo-caustic highlight for heavy droplets seen in the close-up

Figure 33. *Water droplets refracting the scene of the toy store interior through and reflecting external lights. Note the shadows from the water droplets on the toys inside*

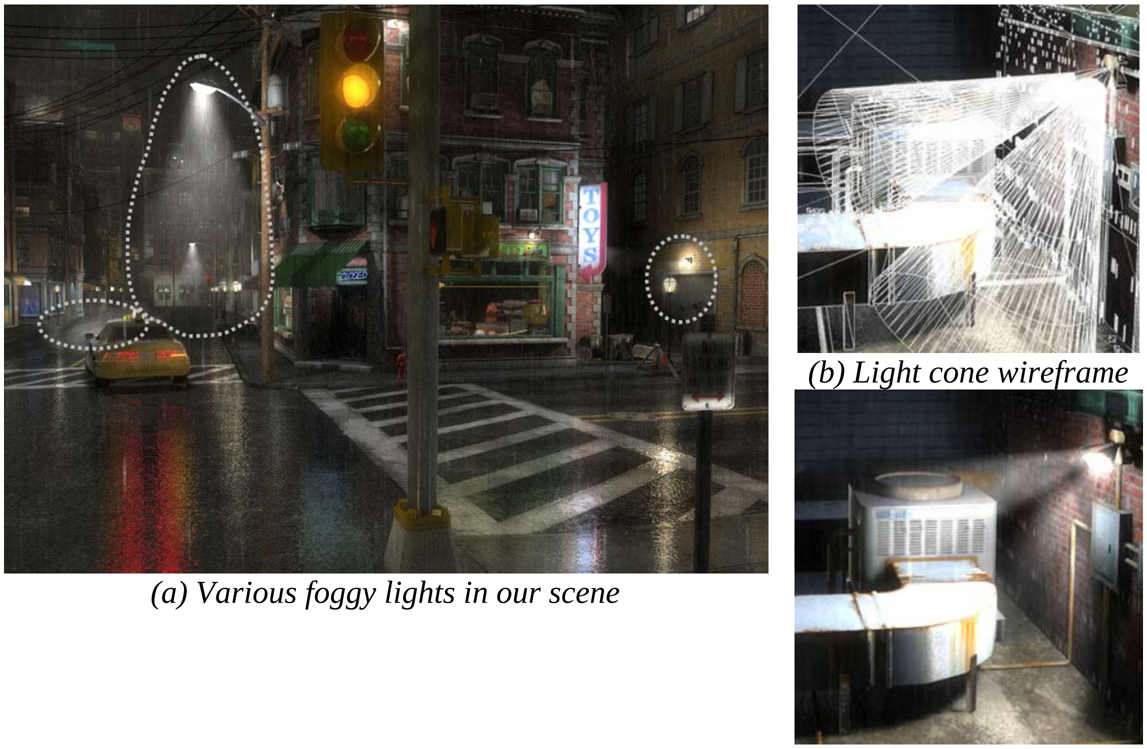
3.8 Effects medley

Aside from the main rain-related effects, we developed a number of secondary objects' effects that we would like to briefly mention here since they help increase the realism of our final environment.

3.8.1 Foggy lights in the street

Our scene has many foggy lights with rain (Figure 34a). We used an approximation shader instead of an expensive volumetric technique to render the volume of light under each lamp. We can simulate these lights as pseudo-volumetric light cones by rendering noisy cloud-like fog on the light cone surface (light cone objects' wireframes are shown in Figure 34b). This efficient approach uses a tangent space technique to control lit fog fading towards the edges of the light frustum (Figure 34c). In order to get smooth falloff on the silhouette edge of the cone, the view vector is transformed into tangent space and its angle is then used to attenuate the falloff. In the pixel shader we simulate distance attenuation of a light by a square of the *v* texture coordinate. Then we perform two fetches from a noise map scrolled in different directions to create perception of participating media in this light's frustum.

To render rain, falling under the bright light, we render an inserted quad plane in the middle of the light frustum with rain texture scrolling vertically matching the composite rain effect. The rain must fade out toward the edges of the frustum in the same fashion as the volumetric light. We use a map to match the cone light attenuation. For angled lamps, we orient the inserted rain quad around the up-axis in world-space in the vertex shader to ensure that the rain continues to respect the laws of gravity and falls downwards.



(a) Various foggy lights in our scene

(b) Light cone wireframe

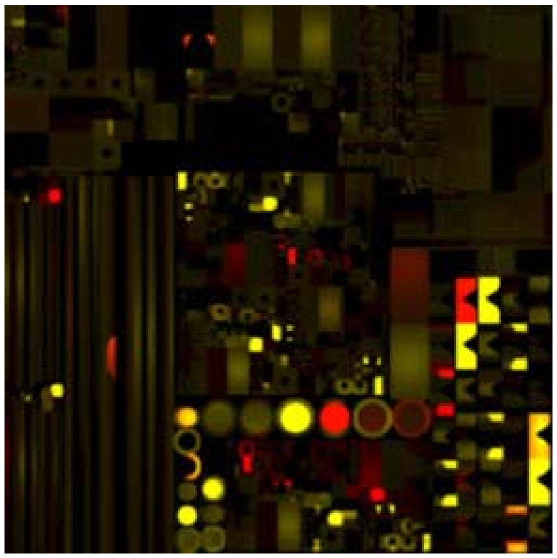
(c) Rooftop light cones

Figure 34. *Foggy volumetric light rendering*

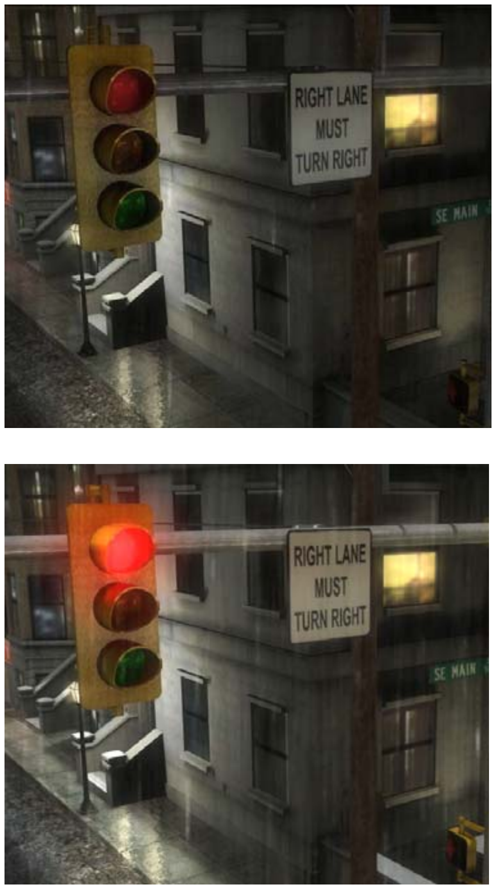
3.8.3 Traffic light illumination

The traffic lamps should dynamically illuminate the traffic light signal object. Computing the full global illumination effects to simulate color bleeding and inter-reflection is an expensive operation (see chapter 9 for a longer discussion of global illumination effects). As a different approach, we use the concept of lightmap to help us simulate these subtle effects in a more efficient manner at run-time. We precompute the global illumination lightmap for the traffic light object, animated in accordance with the blinking traffic light signal (Figure 35a). This lightmap stores color as a result of color bleeding and inter-reflection effects computed with Maya®'s Mental Ray. Note that for a compact object such as the traffic light, this lightmap is very small. At rendering time, we sample the color of the lightmap using the traffic lamp animation parameter as a parameter for computing sampling texture coordinates.

We simulate the internal reflection through the colored reflective glass of the traffic lamp by fetching a normal from the glass normal map and using that normal vector to look-up into an environment map, coloring the resulting reflection color by the lamp color. These internal reflections will only appear when the lamp's glow is faint. We reflect the other parts of the lamp on the outside surface of the glass by another environment map fetch to render external reflections.



(a) Traffic light global-illumination lightmap atlas



(b) Traffic light in off (above) and on (below) state

Figure 35. *Approximating color bleeding for traffic light illumination*

3.8.4 Rendering Misty Rain Halos on Objects

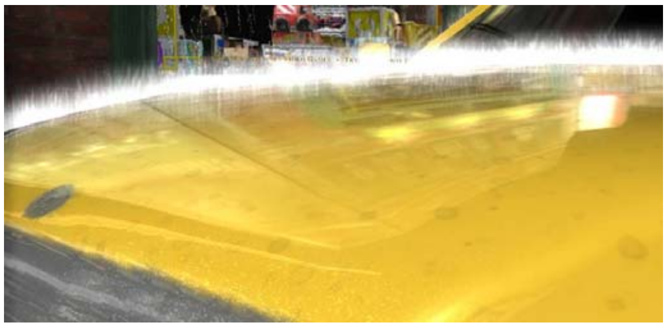
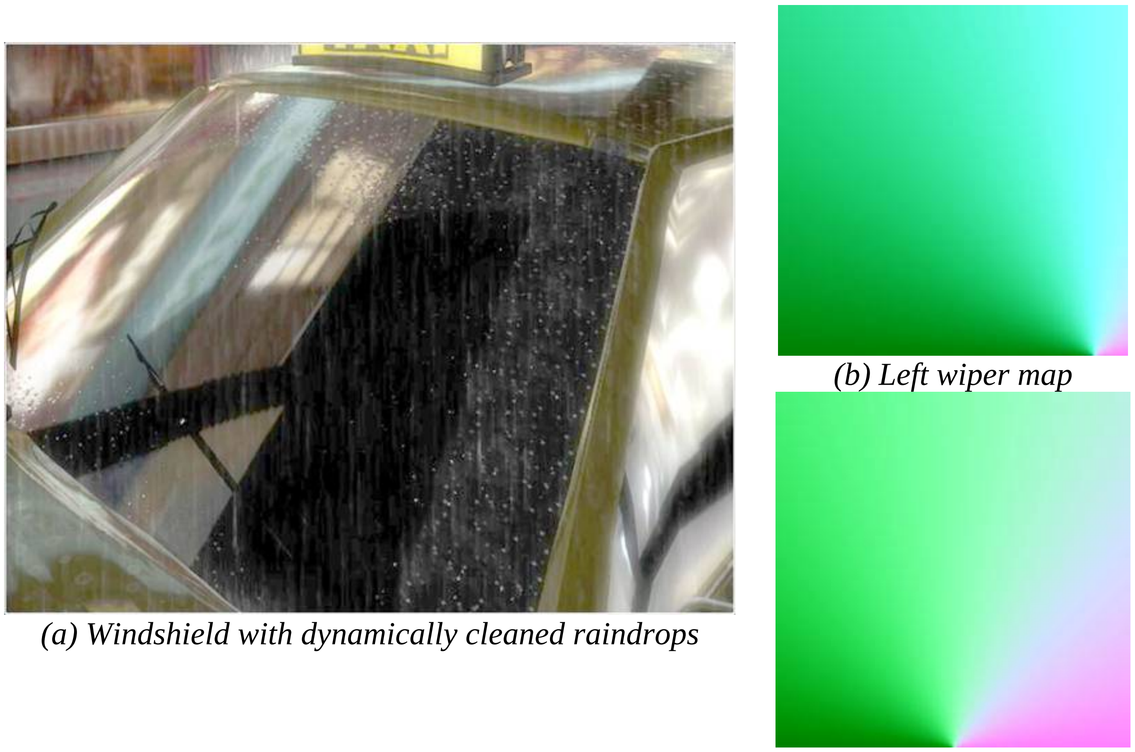


Figure 36. *Misty halos on the taxi with a fins effect and rain splatter via a shells effect*

In a strong rainfall, as the raindrops strike solid objects, they generate not only the splashes, but also a misty halo outlines along the edges of objects. We created a similar effect using the fins and shells technique (similar to real-time fur rendering from [Isidoro02]) (Figure 36). The rain halos are rendered with fin quads with scrolling rain (similar to the composite rain effect). Note that this effect requires additional fin geometry. Using the shells approach, we render rain splatters on the surface of objects in the form of concentric circles. In each successive shell we expand the splash circle footprint with a series of animated texture fetches and blend onto the previous shells.

3.8.5 Taxi windshield wipers effect for wiping off the droplets

The taxi cab windshield wipers can dynamically wipe away the static raindrops on the windshield (Figure 37a). Computing collision with the wipers and affecting droplet movement as a function of that calculation was not practical in our scenario due to many other effects already in place. Since the windshield wasn't prominent in the main fly path through our environment, we wanted an inexpensive approach to render this effect. As a solution, we use two wiper maps (Figures 37b and 37c) to determine which regions on the windshield were recently swiped clean by the wipers. The animation parameters from the wipers are used in the shaders in conjunction with the wiper maps to control the rendering of raindrops depending on which regions were wiped. We use two separate maps so that the wiped regions can overlap, similar to the real-life windshield wipers.



(a) Windshield with dynamically cleaned raindrops

(b) Left wiper map

(c) Right wiper map

Figure 37. *Taxi windshield rendering with animated windshield wipers*

3.9 Conclusions

Rain is a very complex phenomenon and in this chapter we presented a number of effects that help to generate an extensive, detail-rich urban environment in stormy weather. Each technology applied to the ToyShop demo adds detail to the scene. Each additional detail changes the way we experience the environment. It is this attention to detail that seduces the viewer and delivers a lasting impression of the limitless expression of real time graphics. All of these combined effects allow us to create a very believable, realistic impression of a rainy night in a cityscape at highly interactive rates. Rich, complex environments demand convincing details. We hope that the new technology developed for this interactive environment can be successfully used in the next generation of games and real-time rendering.

3.10 Acknowledgements

We would like to thank the ATI ToyShop team whose hard work and dedication resulted in the striking images of this interactive environment. The artists: Dan Roeger, Daniel

Szecket, Abe Wiley and Eli Turner, the programmers: John Isidoro (who has been a crucial part of the development of many effects described in this chapter and to whom we are deeply thankful for his insightful ideas), Daniel Ginsburg, Thorsten Scheuermann, Chris Oat, David Gosselin, and the producers (Lisa Close and Callan McInally). Additionally, we want to thank Jason L. Mitchell from Valve Software and Chris Oat for their help reviewing this chapter and overall encouragement and good humor.

3.11 Bibliography

AKENINE-MÖLLER, T., HEINES, E. 2002. *Real-Time Rendering*, 2nd Edition, A.K. Peters

BAR-JOSEPH, Z., EL-YANIV, R., LISCHINSKI, D., AND M. WERMAN. 2001. Texture mixing and texture movie synthesis using statistical learning. IEEE Transactions on Visualization and Computer Graphics, 7(2):120-135.

DORETTO, G., CHIUSO, A., WU, Y. N., SOATTO, S. 2003. Dynamic textures. International Journal of Computer Vision, 51(2):91-109.

HDRSHOP: HIGH DYNAMIC RANGE IMAGE PROCESSING AND MANIPULATION, version 2.0. Available from http://www.hdrshop.com/

GARG, K., NAYAR, S. K., 2004. Detection and Removal of Rain from Videos. IEEE Conference on Computer Vision and Pattern Recognition, pp. 528-535

GOMEZ, M. 2000. Interactive Simulation of Water Surfaces, Game Programming Gems. De Loura, Marc (Ed.), Charles River Media.

ISIDORO, J. 2006. Shadow Mapping Tricks and Techniques. In the proceedings of Game Developer Conference, San Jose, CA https://www.cmpevents.com/sessions/GD/S1616i1.ppt

JAIN, R., KASTURI, R., SCHUNK, B. G. 1995. Machine Vision. McGraw-Hill.

KANEDA K., IKEDA S., YAMASHITA H. 1999 Animation of Water Droplets Moving Down a Surface, Journal of Visualization and Computer Animation, pp. 15-26

KAWASE, M. 2003. Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless), GDC 2003 lecture. San Jose, CA.

MASON, B. J. 1975. Clouds, Rain and Rainmaking. Cambridge Press.

ISIDORO, J. 2002. User Customizable Real-Time Fur. SIGGRAPH 2002 Technical sketch.

NARASIMHAN, S.G. AND NAYAR, S.K., Shedding Light on the Weather. IEEE CVPR, 2003.

PERSSON E. 2006. HDR Texturing. ATI Technologies Technical Report, ATI SDK, March 2006. http://www2.ati.com/misc/sdk/ati_sdk(mar2006).exe

REINHARD E., WARD G., PATTANAIK S., DEBEVEC P., 2005, High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting. Morgan Kaufmann

REEVES, W. T., SALESIN, D. H., COOK, R. L. 1987. Rendering Antialiased Shadows with Depth Maps. Computer Graphics (SIGGRAPH '87 Proceedings), pp. 238-291.

RIGUER, G., TATARCHUK, N., ISIDORO, J. 2003. Real-Time Depth of Field Simulation. ShaderX^2: Shader Programming Tips and Tricks With DirectX 9.0. W., Ed. Wordware

SCHEUERMANN, T., TATARCHUK, N. 2004. Improved Depth of Field Rendering. ShaderX^3: Advanced Rendering with DirectX and OpenGL. Engel, W., Ed. Charles River Media

STARIK, S., AND WERMAN, M., 2003. Simulation of Rain in Videos. Texture 2003 (The 3rd international workshop on texture analysis and synthesis)

TOYSHOP DEMO, 2005. ATI Research, Inc. Can be downloaded from http://www.ati.com/developer/demos/rx1800.html

VAN DE HULST, H. C. 1981. Light Scattering by Small Particles. Dover Publications.

WANG, T. AND CLIFFORD, R. S. 1975. Use of Rainfall-Induced Optical Scintillations to Measure Path-Averaged Rain Parameters. JOSA, pp. 8-927-237.