Foreword

Composition, the organization of elemental operations into a nonobvious whole, is the essence of imperative programming. The instruction set architecture (ISA) of a microprocessor is a versatile composition interface, which programmers of software renderers have used effectively and creatively in their quest for image realism. Early graphics hardware increased rendering performance, but often at a high cost in composability, and thus in programmability and application innovation. Hardware with microprocessor-like programmability did evolve (for example, the Ikonas Graphics System), but the dominant form of graphics hardware acceleration has been organized around a fixed sequence of rendering operations, often referred to as the *graphics pipeline*. Early interfaces to these systems—such as CORE and later, PHIGS—allowed programmers to specify rendering results, but they were not designed for composition.

OpenGL, which I helped to evolve from its Silicon Graphics-defined predecessor IRIS GL in the early 1990s, addressed the need for composability by specifying an architecture (informally called the *OpenGL Machine*) that was accessed through an imperative programmatic interface. Many features—for example, tightly specified semantics; table-driven operations such as stencil and depth-buffer functions; texture mapping exposed as a general 1D, 2D, and 3D lookup function; and required repeatability properties—ensured that programmers could compose OpenGL operations with powerful and reliable results. Some of the useful techniques that OpenGL enabled include texture-based volume rendering, shadow volumes using stencil buffers, and constructive solid geometry algorithms such as capping (the computation of surface planes at the intersections of clipping planes and solid objects defined by polygons). Ultimately, Mark Peercy and the coauthors of the SIGGRAPH 2000 paper "Interactive Multi-Pass Programmable Shading" demonstrated that arbitrary RenderMan shaders could be accelerated through the composition of OpenGL rendering operations.

During this decade, increases in the raw capability of integrated circuit technology allowed the OpenGL architecture (and later, Direct3D) to be extended to expose an

ISA interface. These extensions appeared as programmable vertex and fragment shaders within the graphics pipeline and now, with the introduction of CUDA, as a data-parallel ISA in near parity with that of the microprocessor. Although the cycle toward complete microprocessor-like versatility is not complete, the tremendous power of graphics hardware acceleration is more accessible than ever to programmers.

And what computational power it is! At this writing, the NVIDIA GeForce 8800 Ultra performs over 400 billion floating-point operations per second—more than the most powerful supercomputer available a decade ago, and five times more than today's most powerful microprocessor. The data-parallel programming model the Ultra supports allows its computational power to be harnessed without concern for the number of processors employed. This is critical, because while today's Ultra already includes over 100 processors, tomorrow's will include thousands, and then more. With no end in sight to the annual compounding of integrated circuit density known as Moore's Law, massively parallel systems are clearly the future of computing, with graphics hardware leading the way.

GPU Gems 3 is a collection of state-of-the-art GPU programming examples. It is about putting data-parallel processing to work. The first four sections focus on graphics-specific applications of GPUs in the areas of geometry, lighting and shadows, rendering, and image effects. Topics in the fifth and sixth sections broaden the scope by providing concrete examples of nongraphical applications that can now be addressed with data-parallel GPU technology. These applications are diverse, ranging from rigid-body simulation to fluid flow simulation, from virus signature matching to encryption and decryption, and from random number generation to computation of the Gaussian.

Where is this all leading? The cover art reminds us that the mind remains the most capable parallel computing system of all. A long-term goal of computer science is to achieve and, ultimately, to surpass the capabilities of the human mind. It's exciting to think that the computer graphics community, as we identify, address, and master the challenges of massively parallel computing, is contributing to the realization of this dream.

Kurt Akeley
Microsoft Research

Preface

It has been only three years since the first *GPU Gems* book was introduced, and some areas of real-time graphics have truly become ultrarealistic. Chapter 14, "Advanced Techniques for Realistic Real-Time Skin Rendering," illustrates this evolution beautifully, describing a skin rendering technique that works so well that the data acquisition and animation will become the most challenging problem in rendering human characters for the next couple of years.

All this progress has been fueled by a sustained rhythm of GPU innovation. These processing units continue to become faster and more flexible in their use. Today's GPUs can process enormous amounts of data and are used not only for rendering 3D scenes, but also for processing images or performing massively parallel computing, such as financial statistics or terrain analysis for finding new oil fields.

Whether they are used for computing or graphics, GPUs need a software interface to drive them, and we are in the midst of an important transition. The new generation of APIs brings additional orthogonality and exposes new capabilities such as generating geometry programmatically. On the computing side, the CUDA architecture lets developers use a C-like language to perform computing tasks rather than forcing the programmer to use the graphics pipeline. This architecture will allow developers without a graphics background to tap into the immense potential of the GPU.

More than 200 chapters were submitted by the GPU programming community, covering a large spectrum of GPU usage ranging from pure 3D rendering to nongraphics applications. Each of them went through a rigorous review process conducted both by NVIDIA's engineers and by external reviewers.

We were able to include 41 chapters, each of which went through another review, during which feedback from the editors and peer reviewers often significantly improved the content. Unfortunately, we could not include some excellent chapters, simply due to the space restriction of the book. It was difficult to establish the final table of contents, but we would like to thank everyone who sent a submission.

Intended Audience

For the graphics-related chapters, we expect the reader to be familiar with the fundamentals of computer graphics including graphics APIs such as DirectX and OpenGL, as well as their associated high-level programming languages, namely HLSL, GLSL, or Cg. Anyone working with interactive 3D applications will find in this book a wealth of applicable techniques for today's and tomorrow's GPUs.

Readers interested in computing and CUDA will find it best to know parallel computing concepts. C programming knowledge is also expected.

Trying the Code Samples

GPU Gems 3 comes with a disc that includes samples, movies, and other demonstrations of the techniques described in this book. You can also go to the book's Web page to find the latest updates and supplemental materials: **developer.nvidia.com/gpugems3**.

Acknowledgments

This book represents the dedication of many people—especially the numerous authors who submitted their most recent work to the GPU community by contributing to this book. Without a doubt, these inspirational and powerful chapters will help thousands of developers push the envelope in their applications.

Our section editors—Cyril Zeller, Evan Hart, Ignacio Castaño Aguado, Kevin Bjorke, Kevin Myers, and Nolan Goodnight—took on an invaluable role, providing authors with feedback and guidance to make the chapters as good as they could be. Without their expertise and contributions above and beyond their usual workload, this book could not have been published.

Ensuring the clarity of *GPU Gems 3* required numerous diagrams, illustrations, and screen shots. A lot of diligence went into unifying the graphic style of about 500 figures, and we thank Michael Fornalski and Jim Reed for their wonderful work on these. We are grateful to Huey Nguyen and his team for their support for many of our projects. We also thank Rory Loeb for his contribution to the amazing book cover design and many other graphic elements of the book.

We would also like to thank Catherine Kilkenny and Teresa Saffaie for tremendous help with copyediting as chapters were being worked on.

Randy Fernando, the editor of the previous *GPU Gems* books, shared his wealth of experience acquired in producing those volumes.

We are grateful to Kurt Akeley for writing our insightful and forward-looking foreword.

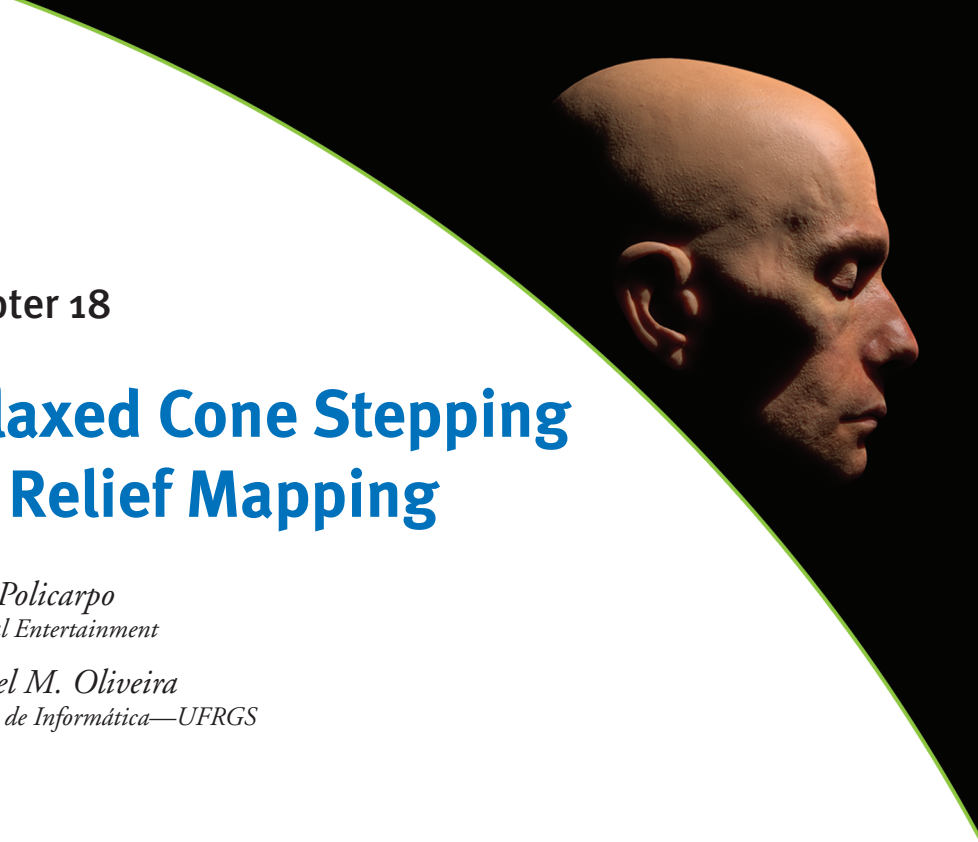
At Addison-Wesley, Peter Gordon, John Fuller, and Kim Boedigheimer managed this project to completion before handing the marketing aspect to Curt Johnson. Christopher Keane did fantastic work on the copyediting and typesetting.

The support from many executive staff members from NVIDIA was critical to this endeavor: Tony Tamasi and Dan Vivoli continually value the creation of educational material and provided the resources necessary to accomplish this project.

We are grateful to Jen-Hsun Huang for his continued support of the *GPU Gems* series and for creating an environment that encourages innovation and teamwork.

We also thank everyone at NVIDIA for their support and for continually building the technology that changes the way people think about computing.

Hubert Nguyen
NVIDIA Corporation



Chapter 18

Relaxed Cone Stepping for Relief Mapping

Fabio Policarpo
Perpetual Entertainment

Manuel M. Oliveira
Instituto de Informática—UFRGS

18.1 Introduction

The presence of geometric details on object surfaces dramatically changes the way light interacts with these surfaces. Although synthesizing realistic pictures requires simulating this interaction as faithfully as possible, explicitly modeling all the small details tends to be impractical. To address these issues, an image-based technique called *relief mapping* has recently been introduced for adding per-fragment details onto arbitrary polygonal models (Policarpo et al. 2005). The technique has been further extended to render correct silhouettes (Oliveira and Policarpo 2005) and to handle non-height-field surface details (Policarpo and Oliveira 2006). In all its variations, the ray-height-field intersection is performed using a binary search, which refines the result produced by some linear search procedure. While the binary search converges very fast, the linear search (required to avoid missing large structures) is prone to aliasing, by possibly missing some thin structures, as is evident in Figure 18-1a. Several space-leaping techniques have since been proposed to accelerate the ray-height-field intersection and to minimize the occurrence of aliasing (Donnelly 2005, Dummer 2006, Baboud and Décoret 2006). *Cone step mapping* (CSM) (Dummer 2006) provides a clever solution to accelerate the intersection calculation for the average case and avoids skipping height-field structures by using some precomputed data (a cone map). However, because CSM uses a conservative approach, the rays tend to stop before the actual surface, which introduces different

kinds of artifacts, highlighted in Figure 18-1b. Using an extension to CSM that consists of employing four different radii for each fragment (in the directions north, south, east, and west), one can just slightly reduce the occurrence of these artifacts. We call this approach *quad-directional cone step mapping* (QDCSM). Its results are shown in Figure 18-1c, which also highlights the technique's artifacts.

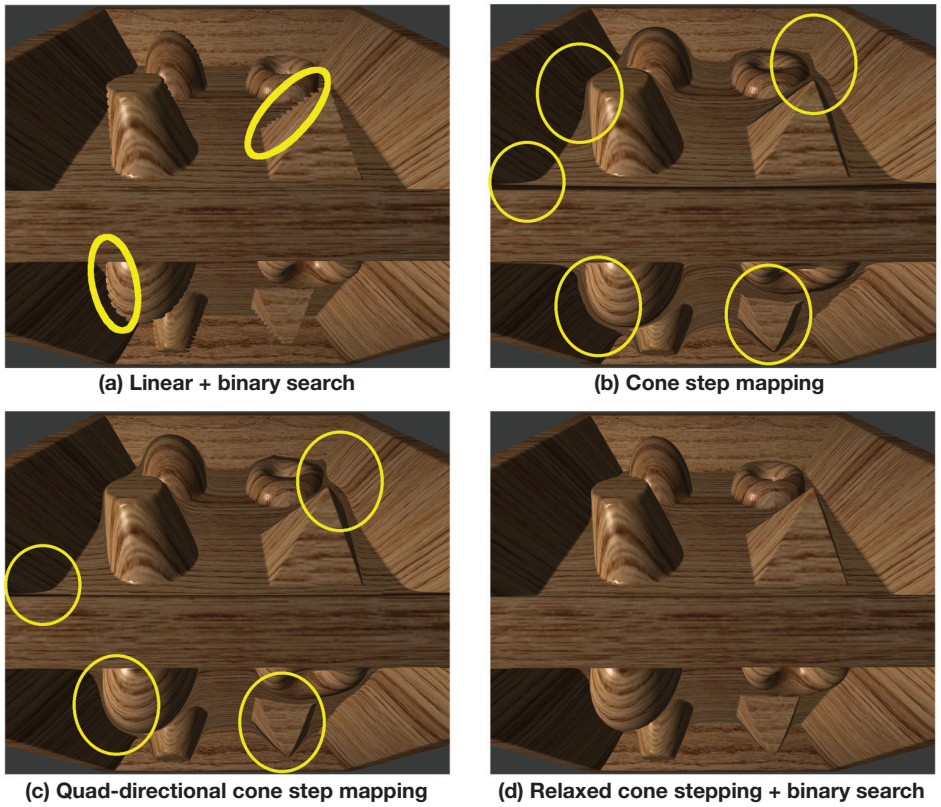


Figure 18-1. Comparison of Four Different Ray-Height-Field Intersection Techniques Used to Render a Relief-Mapped Surface from a 256×256 Relief Texture
(a) Fifteen steps of linear search followed by six steps of binary search. Note the highlighted aliasing artifacts due to the step size used for the linear search. (b) Fifteen steps of the cone step mapping technique. Note the many artifacts caused by the fact that the technique is conservative and many rays will never hit the surface. (c) Fifteen steps of the quad-directional cone step mapping technique. The artifacts in (b) have been reduced but not eliminated. (d) Fifteen steps of the relaxed cone stepping followed by six steps of binary search. Note that the artifacts have been essentially eliminated.

In this chapter, we describe a new ray-height-field intersection strategy for per-fragment displacement mapping that combines the strengths of both cone step mapping and binary search. We call the new space-leaping algorithm *relaxed cone stepping* (RCS), as it relaxes the restriction used to define the radii of the cones in CSM. The idea for the ray-height-field intersection is to replace the linear search with an aggressive space-leaping approach, which is immediately followed by a binary search. While CSM conservatively defines the radii of the cones in such a way that a ray never pierces the surface, RCS allows the rays to pierce the surface at most once. This produces much wider cones, accelerating convergence. Once we know a ray is inside the surface, we can safely apply a binary search to refine the position of the intersection. The combination of RCS and binary search produces renderings of significantly higher quality, as shown in Figure 18-1d. Note that both the aliasing visible in Figure 18-1a and the distortions noticeable in Figures 18-1b and 18-1c have been removed. As a space-leaping technique, RCS can be used with other strategies for refining ray-height-field intersections, such as the one used by *interval mapping* (Risser et al. 2005).

18.2 A Brief Review of Relief Mapping

Relief mapping (Policarpo et al. 2005) simulates the appearance of geometric surface details by shading individual fragments in accordance to some depth and surface normal information that is mapped onto polygonal models. A depth map[1] (scaled to the [0,1] range) represents geometric details assumed to be under the polygonal surface. Depth and normal maps can be stored as a single RGBA texture (32-bit per texel) called a *relief texture* (Oliveira et al. 2000). For better results, we recommend separating the depth and normal components into two different textures. This way texture compression will work better, because a specialized normal compression can be used independent of the depth map compression, resulting in higher compression ratios and fewer artifacts. It also provides better performance because during the relief-mapping iterations, only the depth information is needed and a one-channel texture will be more cache friendly (the normal information will be needed only at the end for lighting). Figure 18-2 shows the normal and depth maps of a relief texture whose cross section is shown in Figure 18-3. The mapping of relief details to a polygonal model is done in the conventional way, by assigning a pair of texture coordinates to each vertex of the model. During rendering, the depth map can be dynamically rescaled to achieve different effects, and correct occlusion is achieved by properly updating the depth buffer.

1. We use the term *depth map* instead of *height map* because the stored values represent depth measured under a reference plane, as opposed to height (measured above it). The reader should not confuse the expression "depth map" used here with shadow buffers.

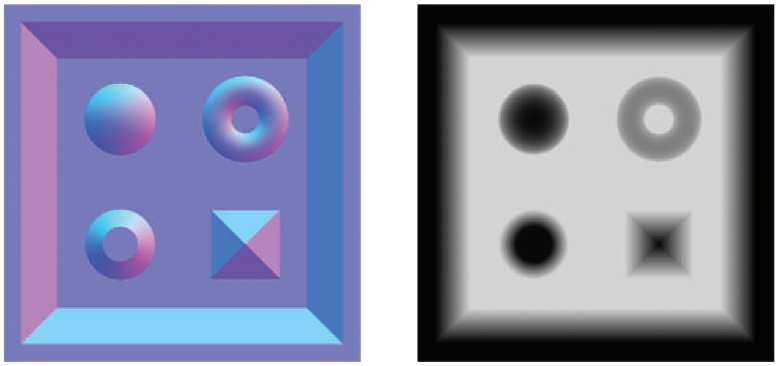


Figure 18-2. Example of a Relief Texture
Left: The normal map is stored in the RGB channels of the texture. Right: The depth map is stored in the alpha channel. Brighter pixels represent deeper geometry.

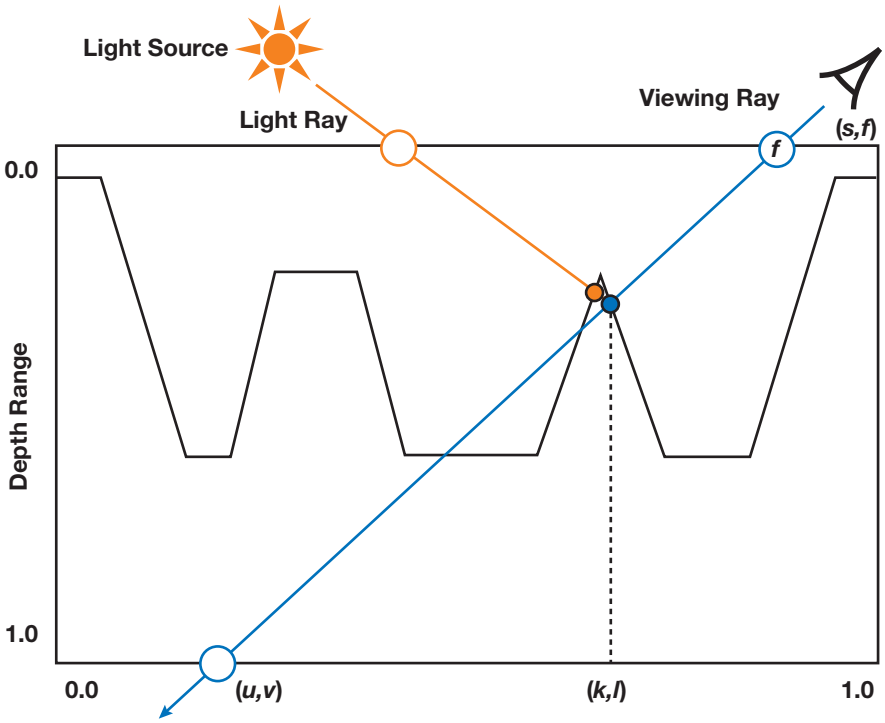


Figure 18-3. Relief Rendering
The viewing ray is transformed to the tangent space of fragment f and then intersected with the relief at point P, with texture coordinates (k, l). Shading is performed using the normal and color stored at the corresponding textures at (k, l). Self-shadowing is computed by checking if the light ray hits P before any other surface point.

Relief rendering is performed entirely on the GPU and can be conceptually divided into three steps. For each fragment f with texture coordinates (s, t), first transform the view direction V to the tangent space of f. Then, find the intersection P of the transformed viewing ray against the depth map. Let (k, l) be the texture coordinates of such intersection point (see Figure 18-3). Finally, use the corresponding position of P, expressed in camera space, and the normal stored at (k, l) to shade f. Self-shadowing can be applied by checking whether the light ray reaches P before reaching any other point on the relief. Figure 18-3 illustrates the entire process. Proper occlusion among relief-mapped and other scene objects is achieved simply by updating the z-buffer with the z coordinate of P (expressed in camera space and after projection and division by w). This updated z-buffer also supports the combined use of shadow mapping (Williams 1978) with relief-mapped surfaces.

In practice, finding the intersection point P can be entirely performed in 2D texture space. Thus, let (u, v) be the 2D texture coordinates corresponding to the point where the viewing ray reaches depth = 1.0 (Figure 18-3). We compute (u, v) based on (s, t), on the transformed viewing direction and on the scaling factor applied to the depth map. We then perform the search for P by sampling the depth map, stepping from (s, t) to (u, v), and checking if the viewing ray has pierced the relief (that is, whether the depth along the viewing ray is bigger than the stored depth) before reaching (u, v). If we have found a place where the viewing ray is under the relief, the intersection P is refined using a binary search.

Although the binary search quickly converges to the intersection point and takes advantage of texture filtering, it could not be used in the beginning of the search process because it may miss some large structures. This situation is depicted in Figure 18-4a, where the depth value stored at the texture coordinates halfway from (s, t) and (u, v) is bigger than the depth value along the viewing ray at point 1, even though the ray has already pierced the surface. In this case, the binary search would incorrectly converge to point Q. To minimize such aliasing artifacts, Policarpo et al. (2005) used a linear search to restrict the binary search space. This is illustrated in Figure 18-4b, where the use of small steps leads to finding point 3 under the surface. Subsequently, points 2 and 3 are used as input to find the desired intersection using a binary search refinement. The linear search itself, however, is also prone to aliasing in the presence of thin structures, as can be seen in Figure 18-1a. This has motivated some researchers to propose the use of additional preprocessed data to avoid missing such thin structures (Donnelly 2005, Dummer 2006, Baboud and Décoret 2006). The technique described in this chapter was inspired by the cone step mapping work of Dummer, which is briefly described next.

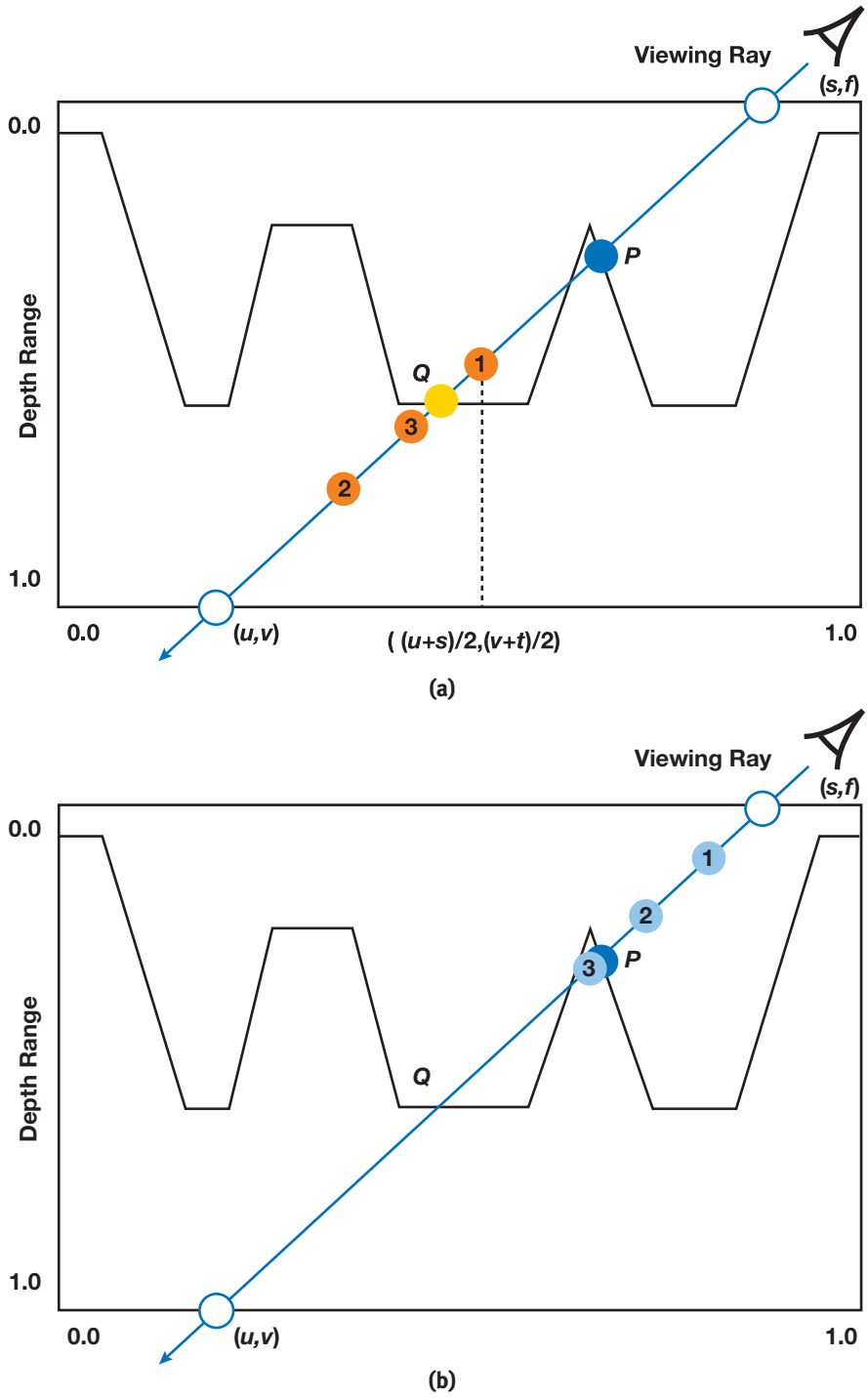


Figure 18-4. Binary Versus Linear Search

(a) A binary search may skip some large structures, missing the first ray-surface intersection (P) and returning a wrong intersection (Q). The numbers inside the circles indicate the order in which the points are visited along the viewing ray. (b) By using smaller steps, the linear search is less prone to aliasing, but not immune to it.

18.3 Cone Step Mapping

Dummer's algorithm for computing the intersection between a ray and a height field avoids missing height-field details by using cone maps (Dummer 2006). A cone map associates a circular cone to each texel of the depth texture. The angle of each cone is the maximum angle that would not cause the cone to intersect the height field. This situation is illustrated in Figure 18-5 for the case of three texels at coordinates (s, t), (a, b), and (c, d), whose cones are shown in yellow, blue, and green, respectively.

Starting at fragment f, along the transformed viewing direction, the search for an intersection proceeds as follows: intersect the ray with the cone stored at (s, t), obtaining point 1 with texture coordinates (a, b). Then advance the ray by intersecting it with the

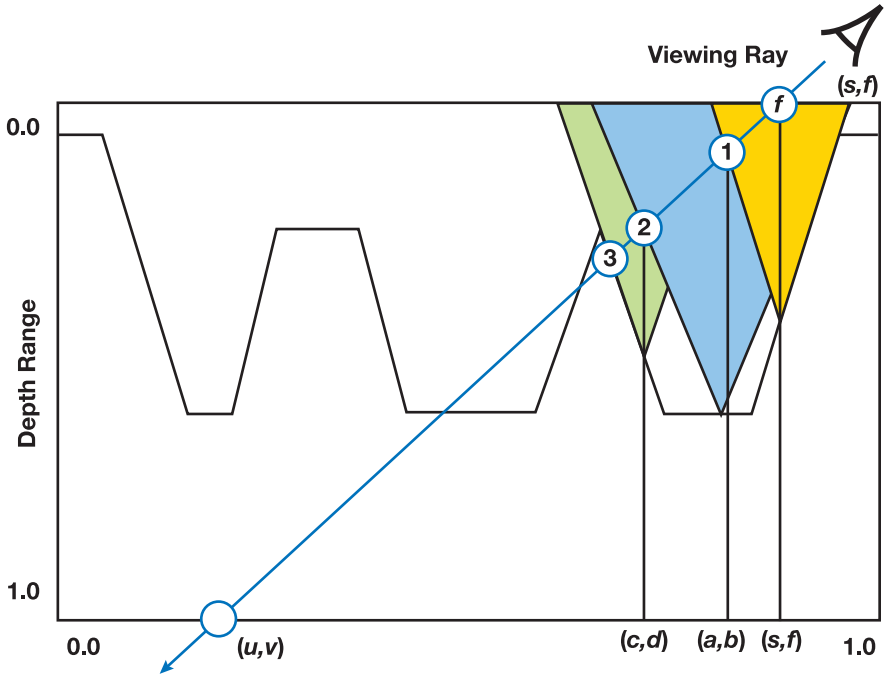


Figure 18-5. Cone Step Mapping
At each pass of the iteration, the ray advances to its intersection with the cone centered at the current texel.

cone stored at (a, b), thus obtaining point 2 at texture coordinates (c, d). Next, intersect the ray with the cone stored at (c, d), obtaining point 3, and so on. In the case of this simple example, point 3 coincides with the desired intersection. Although cone step mapping is guaranteed never to miss the first intersection of a ray with a height field, it may require too many steps to converge to the actual intersection. For performance reasons, however, one is often required to specify a maximum number of iterations. As a result, the ray tends to stop before the actual intersection, implying that the returned texture coordinates used to sample the normal and color maps are, in fact, incorrect. Moreover, the 3D position of the returned intersection, P', in camera space, is also incorrect. These errors present themselves as distortion artifacts in the rendered images, as can be seen in Figures 18-1b and 18-1c.

18.4 Relaxed Cone Stepping

Cone step mapping, as proposed by Dummer, replaces both the linear and binary search steps described in Policarpo et al. 2005 with a single search based on a cone map. A better and more efficient ray-height-field intersection algorithm is achieved by combining the strengths of both approaches: the space-leaping properties of cone step mapping followed by the better accuracy of the binary search. Because the binary search requires one input point to be under and another point to be over the relief surface, we can relax the constraint that the cones in a cone map cannot pierce the surface. In our new algorithm, instead, we force the cones to actually intersect the surface whenever possible. The idea is to make the radius of each cone as large as possible, observing the following constraint: *As a viewing ray travels inside a cone, it cannot pierce the relief more than once*. We call the resulting space-leaping algorithm *relaxed cone stepping*. Figure 18-7a (in the next subsection) compares the radii of the cones used by the conservative cone stepping (blue) and by relaxed cone stepping (green) for a given fragment in a height field. Note that the radius used by RCS is considerably larger, making the technique converge to the intersection using a smaller number of steps. The use of wider relaxed cones eliminates the need for the linear search and, consequently, its associated artifacts. As the ray pierces the surface once, it is safe to proceed with the fast and more accurate binary search.

18.4.1 Computing Relaxed Cone Maps

As in CSM, our approach requires that we assign a cone to each texel of the depth map. Each cone is represented by its *width/height* ratio (ratio w/h, in Figure 18-7c). Because a cone ratio can be stored in a single texture channel, both a depth and a cone map can

be stored using a single luminance-alpha texture. Alternatively, the cone map could be stored in the blue channel of a relief texture (with the first two components of the normal stored in the red and green channels only).

For each reference texel t_i on a relaxed cone map, the angle of cone C_i centered at t_i is set so that no viewing ray can possibly hit the height field more than once while traveling inside C_i. Figure 18-7b illustrates this situation for a set of viewing rays and a given cone shown in green. Note that cone maps can also be used to accelerate the intersection of shadow rays with the height field. Figure 18-6 illustrates the rendering of self-shadowing, comparing the results obtained with three different approaches for rendering per-fragment displacement mapping: (a) relief mapping using linear search, (b) cone step mapping, and (c) relief mapping using relaxed cone stepping. Note the shadow artifacts resulting from the linear search (a) and from the early stop of CSM (b).

Relaxed cones allow rays to enter a relief surface but never leave it. We create relaxed cone maps offline using an $O(n^2)$ algorithm described by the pseudocode shown in Listing 18-1. The idea is, for each source texel `ti`, trace a ray through each destination texel `tj`, such that this ray starts at (`ti.texCoord.s`, `ti.texCoord.t`, `0.0`) and points to (`tj.texCoord.s`, `tj.texCoord.t`, `tj.depth`). For each such ray, compute its next (second) intersection with the height field and use this intersection point to compute the cone ratio `cone_ratio(i,j)`. Figure 18-7c illustrates the situation for a given pair of (`ti, tj`) of source and destination texels. C_i's final ratio is given by the smallest of all cone ratios computed for t_i, which is shown in Figure 18-7b. The relaxed cone map is obtained after all texels have been processed as source texels.

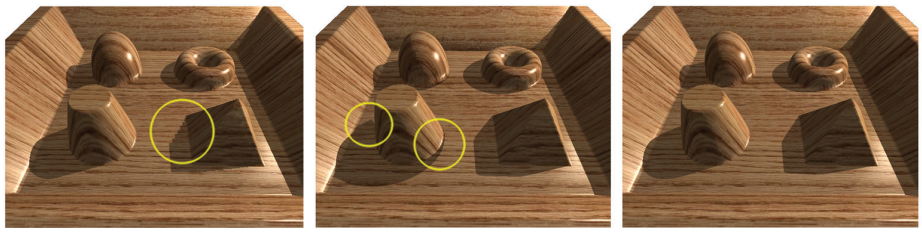


Figure 18-6. Rendering Self-Shadowing Using Different Approaches
(a) Relief mapping with linear search. Note the aliasing on the pyramid shadow. (b) Cone step mapping using cone maps to check the intersection of shadow rays. Note the incorrect shadow cast by the truncated cone on the bottom left. (c) Relief mapping with relaxed cone stepping. Images a, b, and c were generated using the same number of steps shown in Figure 18-1. The intersection with shadow rays used 15 steps/iterations for all images.

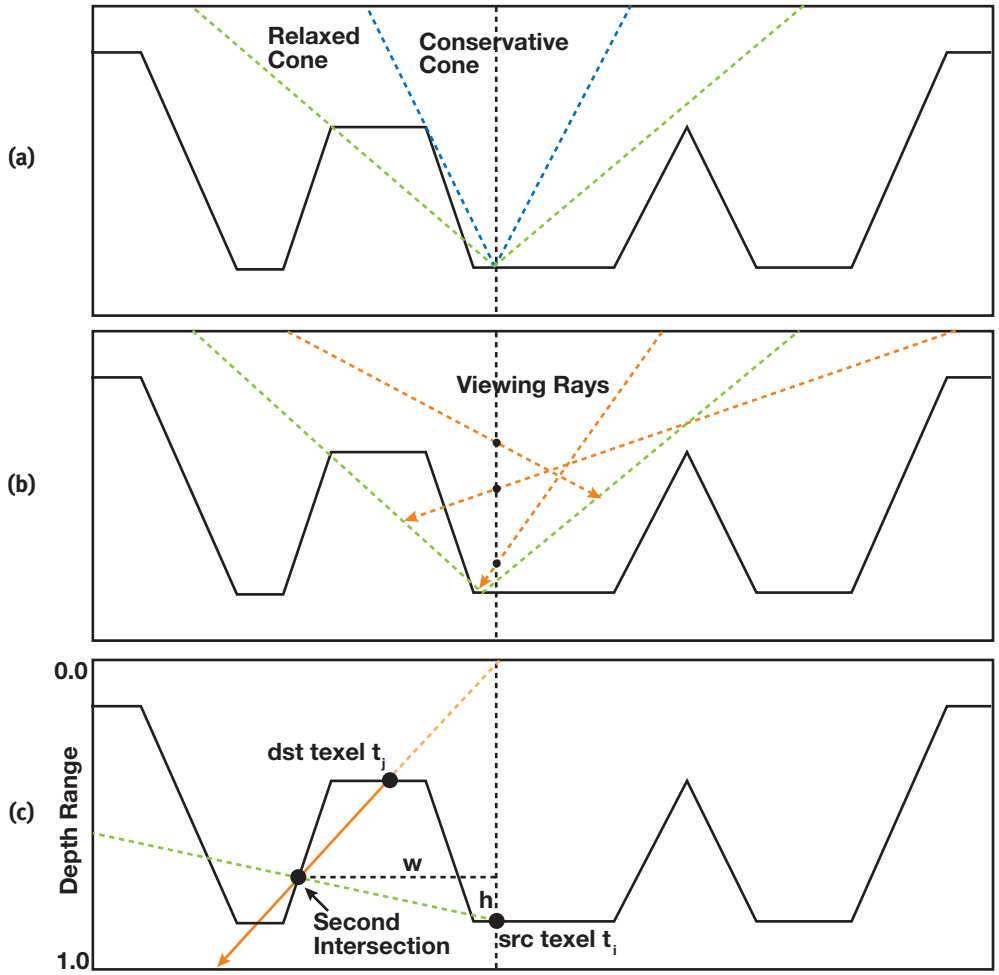


Figure 18-7. Computing Relaxed Cone Maps

(a) Conservative (blue) and relaxed (green) cones for a given texel in the depth map. Notice how the relaxed cone is much wider. (b) No viewing ray can pierce a relief surface more than once while traveling inside a relaxed cone. (c) An intermediate step during computation of the cone ratio for the relaxed cone shown in b.

Listing 18-1. Pseudocode for Computing Relaxed Cone Maps

```
for each reference texel ti do
  radius_cone_C(i) = 1;
  source.xyz = (ti.texCoord.s, ti.texCoord.t, 0.0);
  for each destination texel tj do
    destination.xyz = (tj.texCoord.s, tj.texCoord.t, tj.depth);
    ray.origin = destination;
```

Listing 18-1 (*continued*). Pseudocode for Computing Relaxed Cone Maps

```
ray.direction = destination - source;
(k,w) = text_cords_next_intersection(tj, ray, depth_map);
d = depth_stored_at(k,w);
if ((d  - ti.depth) > 0.0)  // dst has to be above the src
  cone_ratio(i,j) = length(source.xy - destination.xy) /
                    (d - tj.depth);
  if (radius_cone_C(i) > cone_ratio(i,j))
    radius_cone_C(i) = cone_ratio(i,j);
```

Note that in the pseudocode shown in Listing 18-1, as well as in the actual code shown in Listing 18-2, we have clamped the maximum cone ratio values to 1.0. This is done to store the cone maps using integer textures. Although the use of floating-point textures would allow us to represent larger cone ratios with possible gains in space leaping, in practice we have observed that usually only a small subset of the texels in a cone map would be able to take advantage of that. This is illustrated in the relaxed cone map shown in Figure 18-8c. Only the saturated (white) texels would be candidates for having cone ratios bigger than 1.0.

Listing 18-2 presents a shader for generating relaxed cone maps. Figure 18-8 compares three different kinds of cone maps for the depth map associated with the relief texture shown in Figure 18-2. In Figure 18-8a, one sees a conventional cone map (Dummer 2006) stored using a single texture channel. In Figure 18-8b, we have a quad-directional cone map, which stores cone ratios for the four major directions into separate texture channels. Notice how different areas in the texture are assigned wider cones for different directions. Red texels indicate cones that are wider to the right, while green ones are wider to the left. Blue texels identify cones that are wider to the bottom, and black ones are wider to the top. Figure 18-8c shows the corresponding relaxed cone map, also stored using a single texture channel. Note that its texels are much brighter than the corresponding ones in the conventional cone map in Figure 18-8a, revealing its wider cones.

Listing 18-2. A Preprocess Shader for Generating Relaxed Cone Maps

```
float4 depth2relaxedcone(
  in float2 TexCoord : TEXCOORD0,
  in Sampler2D ReliefSampler,
  in float3 Offset ) : COLOR
{
  const int search_steps = 128;
  float3 src = float3(TexCoord,0);  // Source texel
  float3 dst = src + Offset;  // Destination texel
  dst.z = tex2D(ReliefSampler,dst.xy).w;  // Set dest. depth
```

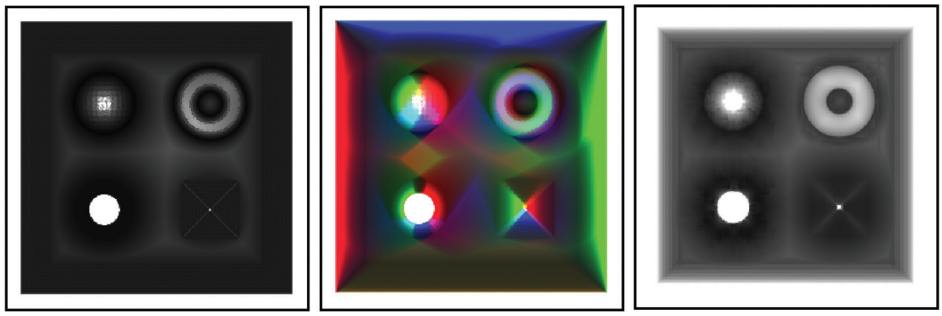


Figure 18-8. A Comparison of Different Kinds of Cone Maps Computed for the Depth Map Shown in Figure 18-2

(a) A conventional cone map (Dummer 2006) (one channel). (b) A quad-directional cone map. The cone ratio values for each of the major four directions are stored in the different channels of the texture. (c) The corresponding relaxed cone map (one channel).

Listing 18-2 (*continued*). A Preprocess Shader for Generating Relaxed Cone Maps

```
float3 vec = dst - src; // Ray direction
vec /= vec.z;  // Scale ray direction so that vec.z = 1.0
vec *= 1.0 - dst.z;  // Scale again
float3 step_fwd = vec/search_steps;  // Length of a forward step

// Search until a new point outside the surface
float3 ray_pos = dst + step_fwd;
for( int i=1; i<search_steps; i++ )
{
  float current_depth = tex2D(ReliefSampler, ray_pos.xy).w;
  if ( current_depth <= ray_pos.z )
    ray_pos += step_fwd;
}

// Original texel depth
float src_texel_depth = tex2D(ReliefSampler,TexCoord).w;
// Compute the cone ratio
float cone_ratio = (ray_pos.z >= src_texel_depth) ? 1.0 :
                   length(ray_pos.xy - TexCoord) /
                   (src_texel_depth - ray_pos.z);
```

Listing 18-2 (*continued*). A Preprocess Shader for Generating Relaxed Cone Maps

```
    // Check for minimum value with previous pass result
    float best_ratio = tex2D(ResultSampler, TexCoord).x;
    if ( cone_ratio > best_ratio )
      cone_ratio = best_ratio;

    return float4(cone_ratio, cone_ratio, cone_ratio, cone_ratio);
}
```

18.4.2 Rendering with Relaxed Cone Maps

To shade a fragment, we step along the viewing ray as it travels through the depth texture, using the relaxed cone map for space leaping. We proceed along the ray until we reach a point inside the relief surface. The process is similar to what we described in Section 18.3 for conventional cone maps. Figure 18-9 illustrates how to find the intersection between a transformed viewing ray and a cone. First, we scale the vector representing the ray direction by dividing it by its z component ($ray.direction.z$), after which, according to Figure 18-9, one can write

$$scaled\_ray.direction.xyz = \frac{ray.direction.xyz}{ray.direction.z} \tag{1}$$

$$m = d \times length(scaled\_ray.direction.xy) = d \times ray\_ratio$$

Likewise:

$$m = g \times cone\_ratio$$

$$m = ((current\_texel.depth - ray\_current\_depth) - d) \times (cone\_ratio) \tag{2}$$

Solving Equations 1 and 2 for d gives the following:

$$d = \frac{(current\_texel.depth - ray\_current\_depth) \times cone\_ratio}{ray\_ratio + cone\_ratio}. \tag{3}$$

From Equation 3, we compute the intersection point I as this:

$$I = ray\_current\_position + d \times scaled\_ray.direction.xyz. \tag{4}$$

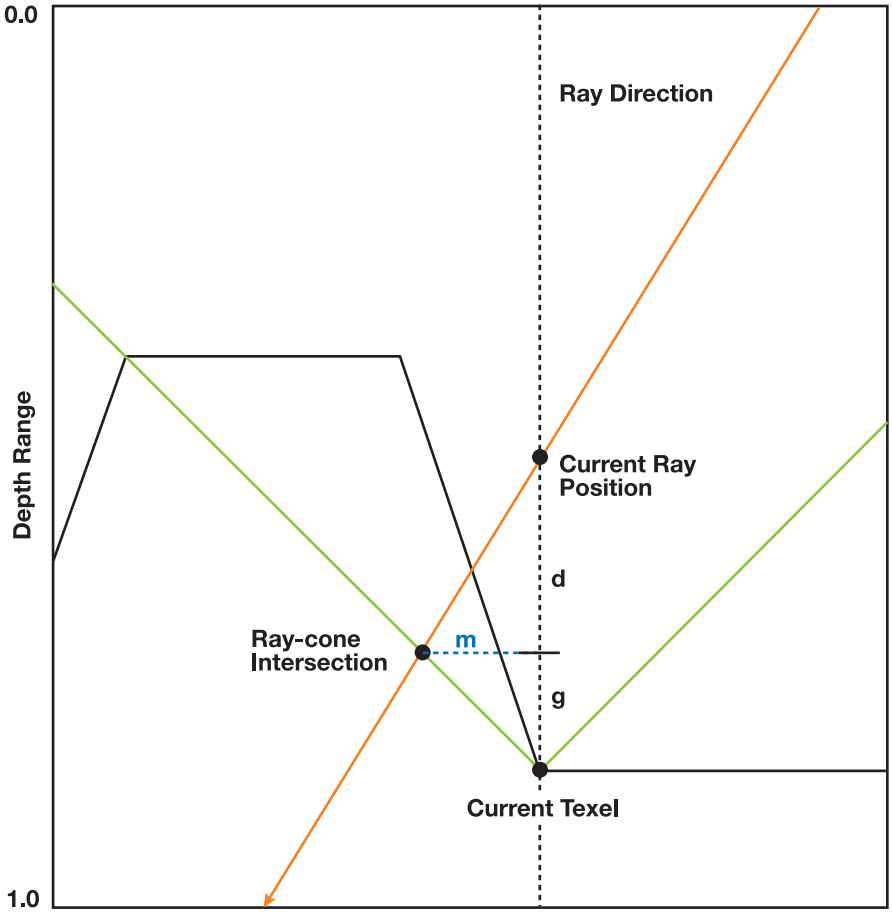


Figure 18-9. Intersecting the Viewing Ray with a Cone
m is the distance, measured in 2D, between the current texel coordinates and the texel coordinates of the intersection point. The difference between the depth at the current ray position and the depth of the current texel is d + g.

The code in Listing 18-3 shows the ray-intersection function for relaxed cone stepping. For performance reasons, the first loop iterates through the relaxed cones for a fixed number of steps. Note the use of the `saturate()` function when calculating the distance to move. This guarantees that we stop on the first visited texel for which the viewing ray is under the relief surface. At the end of this process, we assume the ray has pierced the surface once and then start the binary search for refining the coordinates of the intersection point. Given such coordinates, we then shade the fragment as described in Section 18.2.

Listing 18-3. Ray Intersect with Relaxed Cone

```
// Ray intersect depth map using relaxed cone stepping.
// Depth value stored in alpha channel (black at object surface)
// and relaxed cone ratio stored in blue channel.
void ray_intersect_relaxedcone(
  sampler2D relief_map,  // Relaxed cone map
  inout float3 ray_pos,  // Ray position
  inout float3 ray_dir)  // Ray direction
{
  const int cone_steps = 15;
  const int binary_steps = 6;

  ray_dir /= ray_dir.z;  // Scale ray_dir

  float ray_ratio = length(ray_dir.xy);

  float3 pos = ray_pos;
  for( int i=0; i<cone_steps; i++)
  {
    float4 tex = tex2D(relief_map, pos.xy);
    float cone_ratio = tex.z;
    float height = saturate(tex.w - pos.z);
    float d = cone_ratio*height/(ray_ratio + cone_ratio);
    pos += ray_dir * d;
  }
  // Binary search initial range and initial position
  float3 bs_range = 0.5 * ray_dir * pos.z;
  float3 bs_position = ray_pos + bs_range;

  for( int i=0; i<binary_steps; i++ )
  {
    float4 tex = tex2D(relief_map, bs_position.xy);
    bs_range *= 0.5;
    if (bs_position.z < tex.w)  // If outside
      bs_position += bs_range;  // Move forward
    else
      bs_position -= bs_range;  // Move backward
  }
}
```

Let f be the fragment to be shaded and let K be the point where the viewing ray has stopped (that is, just before performing the binary search), as illustrated in Figure 18-10. If too few steps were used, the ray may have stopped before reaching the surface. Thus, to avoid skipping even thin height-field structures (see the example shown in Figure 18-4a), we use K as the end point for the binary search. In this case, if the ray has not pierced the surface, the search will converge to point K.

Let (m, n) be the texture coordinates associated to K and let d_K be the depth value stored at (m, n) (see Figure 18-10). The binary search will then look for an intersection along the line segment ranging from points H to K, which corresponds to texture coordinates $((s + m)/2, (t + n)/2)$ to (m, n), where (s, t) are the texture coordinates of fragment f (Figure 18-10). Along this segment, the depth of the viewing ray varies linearly from $(d_K/2)$ to d_K. Note that, instead, one could use (m, n) and (q, r) (the texture coordinates of point J, the previously visited point along the ray) as the limits for starting the binary search refinement. However, because we are using a fixed number of iterations for stepping over the relaxed cone map, saving (q, r) would require a conditional statement in the code. According to our experience, this tends to increase the number of registers used in the fragment shader. The graphics hardware has a fixed number of registers and it runs as many threads as it can fit in its register pool. The fewer registers we use, the more threads we will have running at the same time. The latency imposed by the large number of dependent texture reads in relief mapping is hidden when multiple threads are running simultaneously. More-complex code in the loops will increase

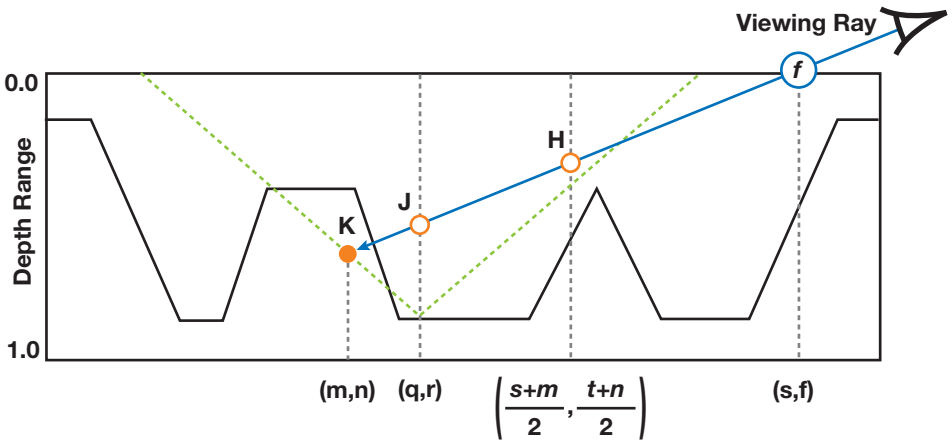


Figure 18-10. The Viewing Ray Through Fragment *f*, with Texture Coordinates (*s*, *t*)
H is the point halfway between f and K, the point where the ray stopped. J is the last visited point along the viewing ray before reaching K.

the number of registers used and thus reduce the number of parallel threads, exposing the latency from the dependent texture reads and reducing the frame rate considerably. So, to keep the shader code shorter, we start the binary search using H and K as limits. Note that after only two iterations of the binary search, one can expect to have reached a search range no bigger than the one defined by the points J and K.

It should be clear that the use of relaxed cone maps could still potentially lead to some distortion artifacts similar to the ones produced by regular (conservative) cone maps (Figure 18-1b). In practice, they tend to be significantly less pronounced for the same number of steps, due to the use of wider cones. According to our experience, the use of 15 relaxed cone steps seems to be sufficient to avoid such artifacts in typical height fields.

18.5 Conclusion

The combined use of relaxed cone stepping and binary search for computing ray-height-field intersection significantly reduces the occurrence of artifacts in images generated with per-fragment displacement mapping. The wider cones lead to more-efficient space leaping, whereas the binary search accounts for more accuracy. If too few cone stepping iterations are used, the final image might present artifacts similar to the ones found in cone step mapping (Dummer 2006). In practice, however, our technique tends to produce significantly better results for the same number of iterations or texture accesses. This is an advantage, especially for the new generations of GPUs, because although both texture sampling and computation performance have been consistently improved, computation performance is scaling faster than bandwidth.

Relaxed cone stepping integrates itself with relief mapping in a very natural way, preserving all of its original features. Figure 18-11 illustrates the use of RCS in renderings involving depth scaling (Figures 18-11b and 18-11d) and changes in tiling factors (Figures 18-11c and 18-11d). Note that these effects are obtained by appropriately adjusting the directions of the viewing rays (Policarpo et al. 2005) and, therefore, not affecting the cone ratios.

Mipmapping can be safely applied to color and normal maps. Unfortunately, conventional mipmapping should not be applied to cone maps, because the filtered values would lead to incorrect intersections. Instead, one should compute the mipmaps manually, by conservatively taking the minimum value for each group of pixels. Alternatively, one can sample the cone maps using a nearest-neighbors strategy. In this case, when an

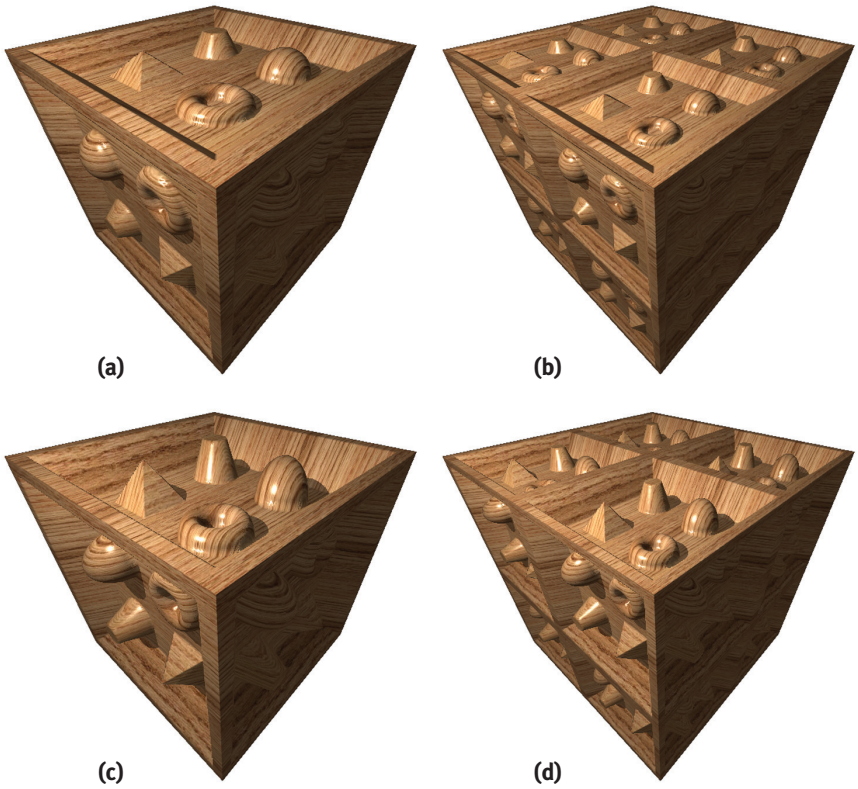


Figure 18-11. Images Showing Changes in Apparent Depth and Tiling Factors
The same relaxed cone map is used for all variations.

object is seen from a distance, the properly sampled color texture tends to hide the aliasing artifacts resulting from the sampling of a high-resolution cone map. Thus, in practice, the only drawback of not applying mipmapping to the cone map is the performance penalty for not taking advantage of sampling smaller textures.

18.5.1 Further Reading

Relief texture mapping was introduced in Oliveira et al. 2000 using a two-pass approach consisting of a prewarp followed by conventional texture mapping. The prewarp, based on the depth map, was implemented on the CPU and the resulting texture sent to the graphics hardware for the final mapping. With the introduction of fragment

processors, Policarpo et al. (2005) generalized the technique for arbitrary polygonal models and showed how to efficiently implement it on a GPU. This was achieved by performing the ray-height-field intersection in 2D texture space. Oliveira and Policarpo (2005) also described how to render curved silhouettes by fitting a quadric surface at each vertex of the model. Later, they showed how to render relief details in preexisting applications using a minimally invasive approach (Policarpo and Oliveira 2006a). They have also generalized the technique to map non-height-field structures onto polygonal models and introduced a new class of impostors (Policarpo and Oliveira 2006b). More recently, Oliveira and Brauwers (2007) have shown how to use a 2D texture approach to intersect rays against depth maps generated under perspective projection and how to use these results to render real-time refractions of distant environments through deforming objects.

18.6 References

Baboud, Lionel, and Xavier Décoret. 2006. "Rendering Geometry with Relief Textures." In *Proceedings of Graphics Interface 2006*.

Donnelly, William. 2005. "Per-Pixel Displacement Mapping with Distance Functions." In *GPU Gems 2*, edited by Matt Pharr, pp. 123–136. Addison-Wesley.

Dummer, Jonathan. 2006. "Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm." Available online at http://www.lonesock.net/files/ConeStepMapping.pdf.

Oliveira, Manuel M., Gary Bishop, and David McAllister. 2000. "Relief Texture Mapping." In *Proceedings of SIGGRAPH 2000*, pp. 359–368.

Oliveira, Manuel M., and Fabio Policarpo. 2005. "An Efficient Representation for Surface Details." UFRGS Technical Report RP-351. Available online at http://www.inf.ufrgs.br/~oliveira/pubs_files/Oliveira_Policarpo_RP-351_Jan_2005.pdf.

Oliveira, Manuel M., and Maicon Brauwers. 2007. "Real-Time Refraction Through Deformable Objects." In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pp. 89–96.

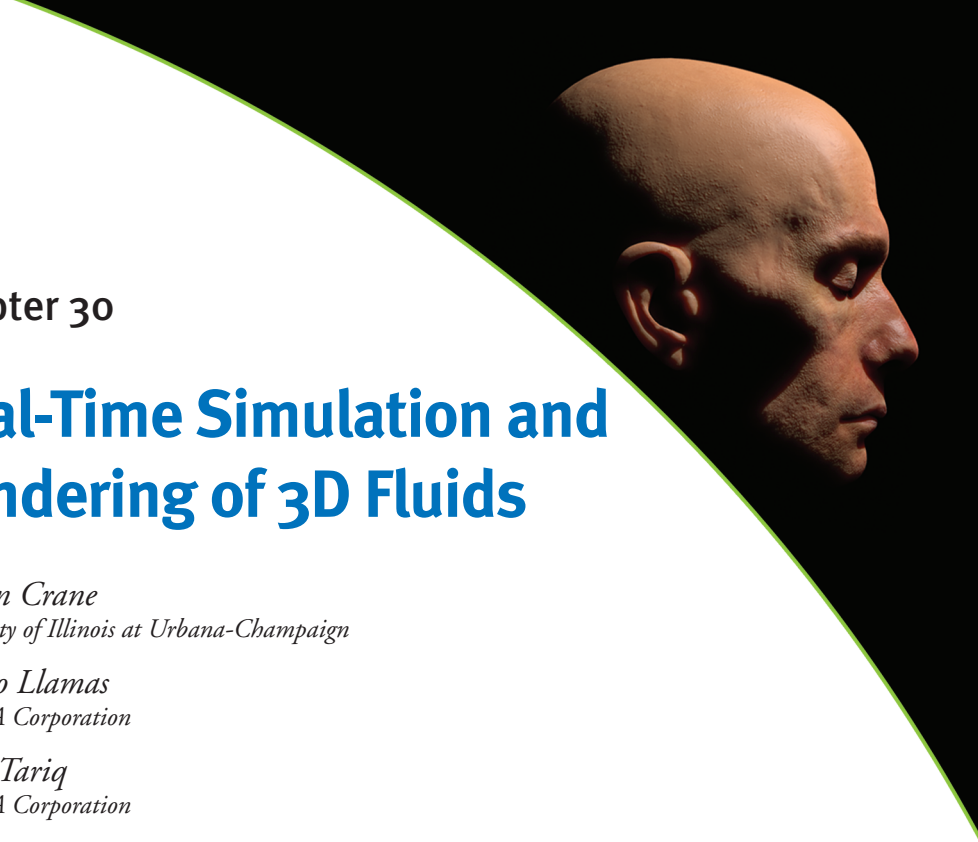
Policarpo, Fabio, Manuel M. Oliveira, and João Comba. 2005. "Real-Time Relief Mapping on Arbitrary Polygonal Surfaces." In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 155–162.

Policarpo, Fabio, and Manuel M. Oliveira. 2006a. "Rendering Surface Details in Games with Relief Mapping Using a Minimally Invasive Approach." In *SHADER X4: Advance Rendering Techniques*, edited by Wolfgang Engel, pp. 109–119. Charles River Media, Inc.

Policarpo, Fabio, and Manuel M. Oliveira. 2006b. "Relief Mapping of Non-Height-Field Surface Details." In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pp. 55–62.

Risser, Eric, Musawir Shah, and Sumanta Pattanaik. 2005. "Interval Mapping." University of Central Florida Technical Report. Available online at http://graphics.cs.ucf.edu/IntervalMapping/images/IntervalMapping.pdf.

Williams, Lance. 1978. "Casting Curved Shadows on Curved Surfaces." In *Computer Graphics (Proceedings of SIGGRAPH 1978)* 12(3), pp. 270–274.



Chapter 30

Real-Time Simulation and Rendering of 3D Fluids

Keenan Crane
University of Illinois at Urbana-Champaign

Ignacio Llamas
NVIDIA Corporation

Sarah Tariq
NVIDIA Corporation

30.1 Introduction

Physically based animation of fluids such as smoke, water, and fire provides some of the most stunning visuals in computer graphics, but it has historically been the domain of high-quality offline rendering due to great computational cost. In this chapter we show not only how these effects can be simulated and rendered in real time, as Figure 30-1 demonstrates, but also how they can be seamlessly integrated into real-time applications. Physically based effects have already changed the way interactive environments are designed. But fluids open the doors to an even larger world of design possibilities.

In the past, artists have relied on particle systems to emulate 3D fluid effects in real-time applications. Although particle systems can produce attractive results, they cannot match the realistic appearance and behavior of fluid simulation. Real time fluids remain a challenge not only because they are more expensive to simulate, but also because the volumetric data produced by simulation does not fit easily into the standard rasterization-based rendering paradigm.

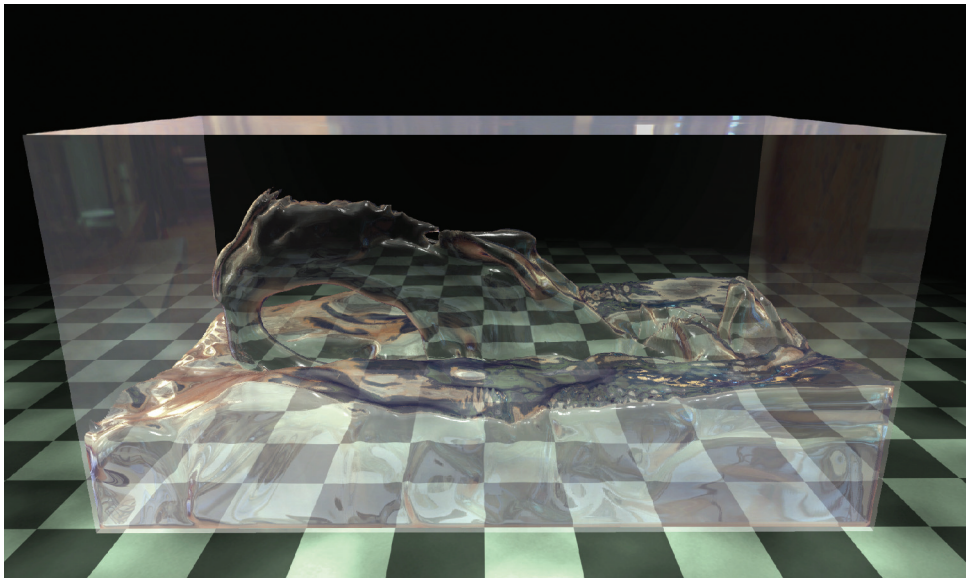


Figure 30-1. Water Simulated and Rendered in Real Time on the GPU

In this chapter we give a detailed description of the technology used for the real-time fluid effects in the NVIDIA GeForce 8 Series launch demo "Smoke in a Box" and discuss its integration into the upcoming game *Hellgate: London*.

The chapter consists of two parts:

- Section 30.2 covers simulation, including smoke, water, fire, and interaction with solid obstacles, as well as performance and memory considerations.
- Section 30.3 discusses how to render fluid phenomena and how to seamlessly integrate fluid rendering into an existing rasterization-based framework.

30.2 Simulation

30.2.1 Background

Throughout this section we assume a working knowledge of general-purpose GPU (GPGPU) methods—that is, applications of the GPU to problems other than conventional raster graphics. In particular, we encourage the reader to look at Harris's chapter on 2D fluid simulation in *GPU Gems* (Harris 2004). As mentioned in that chapter, implementing and debugging a 3D fluid solver is no simple task (even in a traditional programming environment), and a solid understanding of the underlying mathematics

and physics can be of great help. Bridson et al. 2006 provides an excellent resource in this respect.

Fortunately, a deep understanding of partial differential equations (PDEs) is not required to get some basic intuition about the concepts presented in this chapter. All PDEs presented will have the form

$$\frac{\partial}{\partial t} x = f(x, t),$$

which says that the rate at which some quantity x is changing is given by some function f, which may itself depend on x and t. The reader may find it easier to think about this relationship in the discrete setting of *forward Euler integration*:

$$x^{n+1} = x^n + f\left(x^n, t^n\right)\Delta t.$$

In other words, the value of x at the next time step equals the current value of x plus the current rate of change $f(x^n, t^n)$ times the duration of the time step Δt. (Note that superscripts are used to index the time step and do *not* imply exponentiation.) Be warned, however, that the forward Euler scheme is not a good choice numerically—we are suggesting it only as a way to *think* about the equations.

30.2.2 Equations of Fluid Motion

The motion of a fluid is often expressed in terms of its local *velocity* $\mathbf{u}$ as a function of position and time. In computer animation, fluid is commonly modeled as *inviscid* (that is, more like water than oil) and *incompressible* (meaning that volume does not change over time). Given these assumptions, the velocity can be described by the *momentum equation*:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f},$$

subject to the *incompressibility constraint*:

$$\nabla \cdot \mathbf{u} = 0,$$

where p is the pressure, ρ is the mass density, $\mathbf{f}$ represents any external forces (such as gravity), and ∇ is the differential operator:

$$\left[\frac{\partial}{\partial x} \quad \frac{\partial}{\partial y} \quad \frac{\partial}{\partial z}\right]^T.$$

To define the equations of motion in a particular context, it is also necessary to specify *boundary conditions* (that is, how the fluid behaves near solid obstacles or other fluids).

The basic task of a fluid solver is to compute a numerical approximation of **u**. This velocity field can then be used to animate visual phenomena such as smoke particles or a liquid surface.

30.2.3 Solving for Velocity

The popular "stable fluids" method for computing velocity was introduced in Stam 1999, and a GPU implementation of this method for 2D fluids was presented in Harris 2004. In this section we briefly describe how to solve for velocity but refer the reader to the cited works for details.

In order to numerically solve the momentum equation, we must *discretize* our domain (that is, the region of space through which the fluid flows) into computational elements. We choose an *Eulerian* discretization, meaning that computational elements are fixed in space throughout the simulation—only the values stored on these elements change. In particular, we subdivide a rectilinear volume into a regular grid of cubical cells. Each grid cell stores both scalar quantities (such as pressure, temperature, and so on) and vector quantities (such as velocity). This scheme makes implementation on the GPU simple, because there is a straightforward mapping between grid cells and voxels in a 3D texture. *Lagrangian* schemes (that is, schemes where the computational elements are *not* fixed in space) such as smoothed-particle hydrodynamics (Müller et al. 2003) are also popular for fluid animation, but their irregular structure makes them difficult to implement efficiently on the GPU.

Because we discretize space, we must also discretize *derivatives* in our equations: *finite differences* numerically approximate derivatives by taking linear combinations of values defined on the grid. As in Harris 2004, we store all quantities at cell centers for pedagogical simplicity, though a staggered MAC-style grid yields more-robust finite differences and can make it easier to define boundary conditions. (See Harlow and Welch 1965 for details.)

In a GPU implementation, cell attributes (velocity, pressure, and so on) are stored in several 3D textures. At each simulation step, we update these values by running computational *kernels* over the grid. A kernel is implemented as a pixel shader that executes on every cell in the grid and writes the results to an output texture. However, because

GPUs are designed to render into 2D buffers, we must run kernels once for each slice of a 3D volume.

To execute a kernel on a particular grid slice, we rasterize a single quad whose dimensions equal the width and height of the volume. In Direct3D 10 we can directly render into a 3D texture by specifying one of its slices as a render target. Placing the slice index in a variable bound to the `SV_RenderTargetArrayIndex` semantic specifies the slice to which a primitive coming out of the geometry shader is rasterized. (See Blythe 2006 for details.) By iterating over slice indices, we can execute a kernel over the entire grid.

Rather than solve the momentum equation all at once, we split it into a set of simpler operations that can be computed in succession: advection, application of external forces, and pressure projection. Implementation of the corresponding kernels is detailed in Harris 2004, but several examples from our Direct3D 10 framework are given in Listing 30-1. Of particular interest is the routine `PS_ADVECT_VEL`: this kernel implements *semi-Lagrangian* advection, which is used as a building block for more accurate advection in the next section.

Listing 30-1. Simulation Kernels

```
struct GS_OUTPUT_FLUIDSIM
{
  // Index of the current grid cell (i,j,k in [0,gridSize] range)
  float3 cellIndex : TEXCOORD0;

  // Texture coordinates (x,y,z in [0,1] range) for the
  // current grid cell and its immediate neighbors
  float3 CENTERCELL : TEXCOORD1;
  float3 LEFTCELL   : TEXCOORD2;
  float3 RIGHTCELL  : TEXCOORD3;
  float3 BOTTOMCELL : TEXCOORD4;
  float3 TOPCELL    : TEXCOORD5;
  float3 DOWNCELL   : TEXCOORD6;
  float3 UPCELL     : TEXCOORD7;
  float4 pos        : SV_Position; // 2D slice vertex in
                                   // homogeneous clip space
  uint RTIndex     : SV_RenderTargetArrayIndex; // Specifies
                                                // destination slice
};
```

Listing 30-1 (*continued*). Simulation Kernels

```
float3 cellIndex2TexCoord(float3 index)
{
  // Convert a value in the range [0,gridSize] to one in the range [0,1].
  return float3(index.x / textureWidth,
                index.y / textureHeight,
                (index.z+0.5) / textureDepth);
}

float4 PS_ADVECT_VEL(GS_OUTPUT_FLUIDSIM in,
                     Texture3D velocity) : SV_Target
{
  float3 pos = in.cellIndex;
  float3 cellVelocity = velocity.Sample(samPointClamp,
                                        in.CENTERCELL).xyz;

  pos -= timeStep * cellVelocity;
  pos = cellIndex2TexCoord(pos);

  return velocity.Sample(samLinear, pos);
}

float PS_DIVERGENCE(GS_OUTPUT_FLUIDSIM in,
                    Texture3D velocity) : SV_Target
{
  // Get velocity values from neighboring cells.
  float4 fieldL = velocity.Sample(samPointClamp, in.LEFTCELL);
  float4 fieldR = velocity.Sample(samPointClamp, in.RIGHTCELL);
  float4 fieldB = velocity.Sample(samPointClamp, in.BOTTOMCELL);
  float4 fieldT = velocity.Sample(samPointClamp, in.TOPCELL);
  float4 fieldD = velocity.Sample(samPointClamp, in.DOWNCELL);
  float4 fieldU = velocity.Sample(samPointClamp, in.UPCELL);

  // Compute the velocity's divergence using central differences.
  float divergence =  0.5 * ((fieldR.x - fieldL.x)+
                             (fieldT.y - fieldB.y)+
                             (fieldU.z - fieldD.z));

  return divergence;
}
```

Listing 30-1 (*continued*). Simulation Kernels

```
float PS_JACOBI(GS_OUTPUT_FLUIDSIM in,
                Texture3D pressure,
                Texture3D divergence) : SV_Target
{
  // Get the divergence at the current cell.
  float dC = divergence.Sample(samPointClamp, in.CENTERCELL);

  // Get pressure values from neighboring cells.
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);

  // Compute the new pressure value for the center cell.
  return(pL + pR + pB + pT + pU + pD - dC) / 6.0;
}

float4 PS_PROJECT(GS_OUTPUT_FLUIDSIM in,
                  Texture3D pressure,
                  Texture3D velocity): SV_Target
{
  // Compute the gradient of pressure at the current cell by
  // taking central differences of neighboring pressure values.
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);
  float3 gradP = 0.5*float3(pR - pL, pT - pB, pU - pD);

  // Project the velocity onto its divergence-free component by
  // subtracting the gradient of pressure.
  float3 vOld = velocity.Sample(samPointClamp, in.texcoords);
  float3 vNew = vOld - gradP;

  return float4(vNew, 0);
}
```

Improving Detail

The semi-Lagrangian advection scheme used by Stam is useful for animation because it is unconditionally stable, meaning that large time steps will not cause the simulation to "blow up." However, it can introduce unwanted numerical smoothing, making water look viscous or causing smoke to lose detail. To achieve higher-order accuracy, we use a MacCormack scheme that performs two intermediate semi-Lagrangian advection steps. Given a quantity ϕ and an advection scheme A (for example, the one implemented by `PS_ADVECT_VEL`), higher-order accuracy is obtained using the following sequence of operations (from Selle et al. 2007):

$$\hat{\phi}^{n+1} = A\left(\phi^n\right)$$

$$\hat{\phi}^{n} = A^R\left(\hat{\phi}^{n+1}\right)$$

$$\phi^{n+1} = \hat{\phi}^{n+1} + \frac{1}{2}\left(\phi^n - \hat{\phi}^n\right).$$

Here, ϕ^n is the quantity to be advected, $\hat{\phi}^{n+1}$ and $\hat{\phi}^n$ are intermediate quantities, and ϕ^{n+1} is the final advected quantity. The superscript on A^R indicates that advection is reversed (that is, time is run backward) for that step.

Unlike the standard semi-Lagrangian scheme, this MacCormack scheme is not unconditionally stable. Therefore, a limiter is applied to the resulting value ϕ^{n+1}, ensuring that it falls within the range of values contributing to the initial semi-Lagrangian advection. In our GPU solver, this means we must locate the eight nodes closest to the sample point, access the corresponding texels *exactly at their centers* (to avoid getting interpolated values), and clamp the final value to fall within the minimum and maximum values found on these nodes, as shown in Figure 30-2.

Once the intermediate semi-Lagrangian steps have been computed, the pixel shader in Listing 30-2 completes advection using the MacCormack scheme.

Listing 30-2. MacCormack Advection Scheme

```
float4 PS_ADVECT_MACCORMACK(GS_OUTPUT_FLUIDSIM in,
                            float timestep) : SV_Target
{
  // Trace back along the initial characteristic - we'll use
  // values near this semi-Lagrangian "particle" to clamp our
  // final advected value.
  float3 cellVelocity = velocity.Sample(samPointClamp,
                                        in.CENTERCELL).xyz;
```

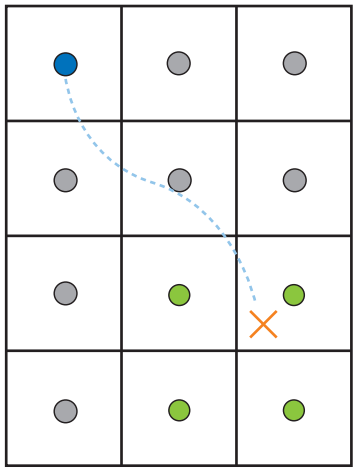


Figure 30-2. Limiter Applied to a MacCormack Advection Scheme in 2D
The result of the advection (blue) is clamped to the range of values from nodes (green) used to get the interpolated value at the advected "particle" (red) in the initial semi-Lagrangian step.

Listing 30-2 (*continued*). MacCormack Advection Scheme

```
float3 npos = in.cellIndex - timestep * cellVelocity;

// Find the cell corner closest to the "particle" and compute the
// texture coordinate corresponding to that location.
npos = floor(npos + float3(0.5f, 0.5f, 0.5f));
npos = cellIndex2TexCoord(npos);

// Get the values of nodes that contribute to the interpolated value.

// Texel centers will be a half-texel away from the cell corner.
float3 ht = float3(0.5f / textureWidth,
                   0.5f / textureHeight,
                   0.5f / textureDepth);

float4 nodeValues[8];
nodeValues[0] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x, -ht.y, -ht.z));
nodeValues[1] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x, -ht.y,  ht.z));
nodeValues[2] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x,  ht.y, -ht.z));
nodeValues[3] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x,  ht.y,  ht.z));
```

Listing 30-2 (*continued*). MacCormack Advection Scheme

```
    nodeValues[4] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x, -ht.y, -ht.z));
    nodeValues[5] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x, -ht.y,  ht.z));
    nodeValues[6] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x,  ht.y, -ht.z));
    nodeValues[7] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x,  ht.y,  ht.z));

    // Determine a valid range for the result.
    float4 phiMin = min(min(min(min(min(min(min(
        nodeValues[0],  nodeValues [1]), nodeValues [2]), nodeValues [3]),
        nodeValues[4]), nodeValues [5]), nodeValues [6]), nodeValues [7]);

    float4 phiMax = max(max(max(max(max(max(max(
        nodeValues[0],  nodeValues [1]), nodeValues [2]), nodeValues [3]),
        nodeValues[4]), nodeValues [5]), nodeValues [6]), nodeValues [7]);

    // Perform final advection, combining values from intermediate
    // advection steps.
    float4 r = phi_n_1_hat.Sample(samLinear, nposTC) +
               0.5 * (phi_n.Sample(samPointClamp, in.CENTERCELL) -
                      phi_n_hat.Sample(samPointClamp, in.CENTERCELL));

    // Clamp result to the desired range.
    r = max(min(r, phiMax), phiMin);

    return r;
}
```

On the GPU, higher-order schemes are often a better way to get improved visual detail than simply increasing the grid resolution, because math is cheap compared to bandwidth. Figure 30-3 compares a higher-order scheme on a low-resolution grid with a lower-order scheme on a high-resolution grid.

30.2.4 Solid-Fluid Interaction

One of the benefits of using real-time simulation (versus precomputed animation) is that fluid can interact with the environment. Figure 30-4 shows an example on one such scene. In this section we discuss two simple ways to allow the environment to act on the fluid.

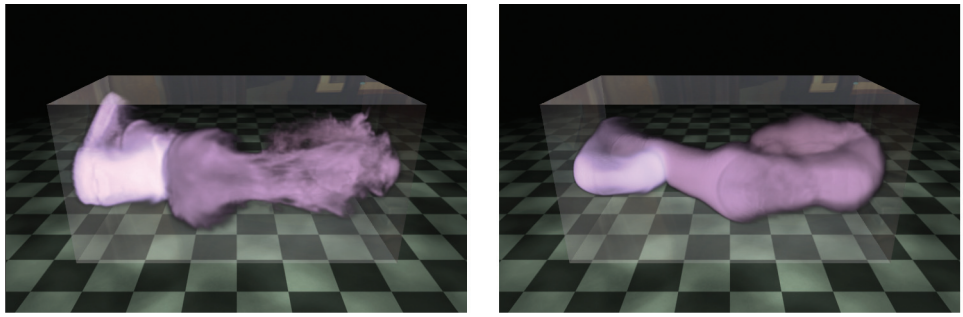


Figure 30-3. Bigger Is Not Always Better!
Left: MacCormack advection scheme (applied to both velocity and smoke density) on a 128×64×64 grid. Right: Semi-Lagrangian advection scheme on a 256×128×128 grid.

A basic way to influence the velocity field is through the application of external forces. To get the gross effect of an obstacle pushing fluid around, we can approximate the obstacle with a basic shape such as a box or a ball and add the obstacle's average velocity to that region of the velocity field. Simple shapes like these can be described with an implicit equation of the form $f(x, y, z) \leq 0$ that can be easily evaluated by a pixel shader at each grid cell.

Although we could explicitly add velocity to approximate simple motion, there are situations in which more detail is required. In *Hellgate: London*, for example, we wanted smoke to seep out through cracks in the ground. Adding a simple upward velocity and smoke density in the shape of a crack resulted in uninteresting motion. Instead, we used the crack shape, shown inset in Figure 30-5, to define *solid obstacles* for smoke to collide and interact with. Similarly, we wanted to achieve more-precise interactions between smoke and an animated gargoyle, as shown in Figure 30-4. To do so, we needed to be able to affect the fluid motion with dynamic obstacles (see the details later in this section), which required a volumetric representation of the obstacle's interior and of the velocity at its boundary (which we also explain later in this section).

Figure 30-4. An Animated Gargoyle Pushes Smoke Around by Flapping Its Wings

Figure 30-5. Smoke Rises from a Crack in the Ground in the Game *Hellgate: London*
Inset: A slice from the obstacle texture that was used to block the smoke; white texels indicate an obstacle, and black texels indicate open space.

Dynamic Obstacles

So far we have assumed that a fluid occupies the entire rectilinear region defined by the simulation grid. However, in most applications, the fluid *domain* (that is, the region of the grid actually occupied by fluid) is much more interesting. Various methods for handling *static* boundaries on the GPU are discussed in Harris et al. 2003, Liu et al. 2004, Wu et al. 2004, and Li et al. 2005.

The fluid domain may change over time to adapt to dynamic obstacles in the environment, and in the case of liquids, such as water, the domain is constantly changing as the liquid sloshes around (more in Section 30.2.7). In this section we describe the scheme used for handling dynamic obstacles in *Hellgate: London*. For further discussion of dynamic obstacles, see Bridson et al. 2006 and Foster and Fedkiw 2001.

To deal with complex domains, we must consider the fluid's behavior at the *domain boundary*. In our discretized fluid, the domain boundary consists of the *faces* between cells that contain fluid and cells that do not—that is, the face *between* a fluid cell and a solid cell is part of the boundary, but the solid cell itself is not. A simple example of a domain boundary is a static barrier placed around the perimeter of the simulation grid to prevent fluid from "escaping" (without it, the fluid appears as though it is simply flowing out into space).

To support domain boundaries that change due to the presence of dynamic obstacles, we need to modify some of our simulation steps. In our implementation, obstacles are represented using an inside-outside voxelization. In addition, we keep a voxelized representation of the obstacle's velocity in solid cells adjacent to the domain boundary. This information is stored in a pair of 3D textures that are updated whenever an obstacle moves or deforms (we cover this later in this section).

At solid-fluid boundaries, we want to impose a *free-slip* boundary condition, which says that the velocities of the fluid and the solid are the same in the direction normal to the boundary:

$$\mathbf{u} \cdot \mathbf{n} = \mathbf{u}_{\text{solid}} \cdot \mathbf{n}.$$

In other words, the fluid cannot flow into or out of a solid, but it is allowed to flow freely along its surface.

The free-slip boundary condition also affects the way we solve for pressure, because the gradient of pressure is used in determining the final velocity. A detailed discussion of pressure projection can be found in Bridson et al. 2006, but ultimately we just need to make sure that the pressure values we compute satisfy the following:

$$\frac{\Delta t}{\rho \Delta x^2}\left(\left|F_{i,j,k}\right| p_{i,j,k} - \sum_{\mathbf{n} \in F_{i,j,k}} p_{\mathbf{n}}\right) = -d_{i,j,k},$$

where Δt is the size of the time step, Δx is the cell spacing, $p_{i,j,k}$ is the pressure value in cell (i, j, k), $d_{i,j,k}$ is the discrete velocity divergence computed for that cell, and $F_{i,j,k}$ is the set of *indices* of cells adjacent to cell (i, j, k) that contain fluid. (This equation is simply a discrete form of the pressure-Poisson system $\nabla^2 p = \nabla \cdot \mathbf{w}$ in Harris 2004 that respects solid boundaries.) It is also important that at solid-fluid boundaries, $d_{i,j,k}$ is computed using obstacle velocities.

In practice there's a very simple trick for making sure all this happens: any time we sample pressure from a neighboring cell (for example, in the pressure solve and pressure projection steps), we check whether the neighbor contains a solid obstacle, as shown in Figure 30-6. If it does, we use the pressure value from the center cell in place of the neighbor's pressure value. In other words, we nullify the solid cell's contribution to the preceding equation.

We can apply a similar trick for velocity values: whenever we sample a neighboring cell (for example, when computing the velocity's divergence), we first check to see if it contains a solid. If so, we look up the obstacle's velocity from our voxelization and use it in place of the value stored in the fluid's velocity field.

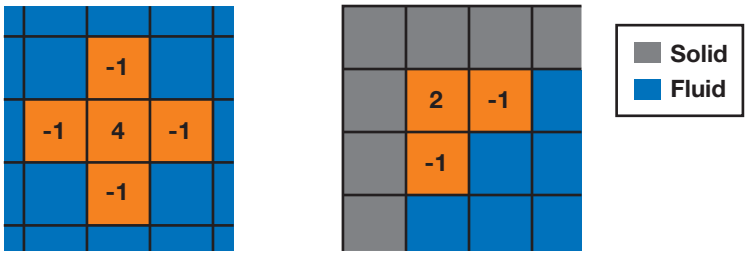


Figure 30-6. Accounting for Obstacles in the Computation of the Discrete Laplacian of Pressure
Left: A stencil used to compute the discrete Laplacian of pressure in 2D. Right: This stencil changes near solid-fluid boundaries. Checking for solid neighbors and replacing their pressure values with the central pressure value results in the same behavior.

Because we cannot always solve the pressure-Poisson system to convergence, we explicitly enforce the free-slip boundary condition immediately following pressure projection. We must also correct the result of the pressure projection step for fluid cells next to the domain boundary. To do so, we compute the obstacle's velocity component in the direction normal to the boundary. This value replaces the corresponding component of our fluid velocity at the center cell, as shown in Figure 30-7. Because solid-fluid boundaries are aligned with voxel faces, computing the projection of the velocity onto the surface normal is simply a matter of selecting the appropriate component.

If two opposing faces of a fluid cell are solid-fluid boundaries, we could average the velocity values from both sides. However, simply selecting one of the two faces generally gives acceptable results.

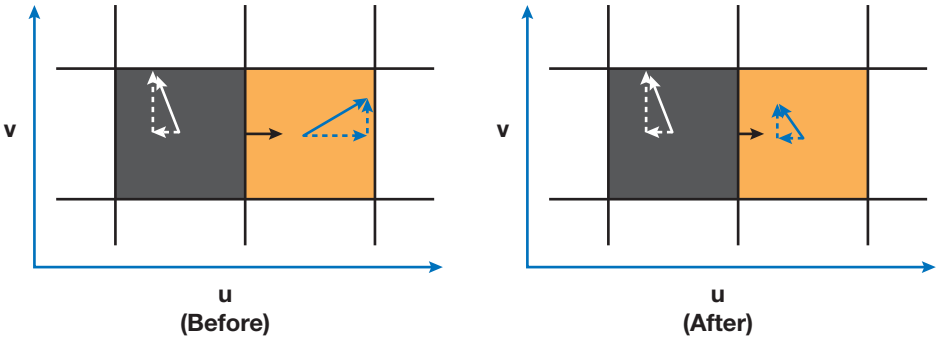


Figure 30-7. Enforcing the Free-Slip Boundary Condition After Pressure Projection
To enforce free-slip behavior at the boundary between a fluid cell (red) and a solid cell (black), we modify the velocity of the fluid cell in the normal (u) direction so that it equals the obstacle's velocity in the normal direction. We retain the fluid velocity in the tangential (v) direction.

Finally, it is important to realize that when very large time steps are used, quantities can "leak" through boundaries during advection. For this reason we add an additional constraint to the advection steps to ensure that we never advect any quantity into the interior of an obstacle, guaranteeing that the value of advected quantities (for example, smoke density) is always zero inside solid obstacles (see the PS_ADVECT_OBSTACLE routine in Listing 30-3). In Listing 30-3, we show the simulation kernels modified to take boundary conditions into account.

Listing 30-3. Modified Simulation Kernels to Account for Boundary Conditions

```
bool IsSolidCell(float3 cellTexCoords)
{
  return obstacles.Sample(samPointClamp, cellTexCoords).r > 0.9;
}

float PS_JACOBI_OBSTACLE(GS_OUTPUT_FLUIDSIM in,
                         Texture3D pressure,
                         Texture3D divergence) : SV_Target
{
  // Get the divergence and pressure at the current cell.
  float dC = divergence.Sample(samPointClamp, in.CENTERCELL);
  float pC = pressure.Sample(samPointClamp, in.CENTERCELL);

  // Get the pressure values from neighboring cells.
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);

  // Make sure that the pressure in solid cells is effectively ignored.
  if(IsSolidCell(in.LEFTCELL)) pL = pC;
  if(IsSolidCell(in.RIGHTCELL)) pR = pC;
  if(IsSolidCell(in.BOTTOMCELL)) pB = pC;
  if(IsSolidCell(in.TOPCELL)) pT = pC;
  if(IsSolidCell(in.DOWNCELL)) pD = pC;
  if(IsSolidCell(in.UPCELL)) pU = pC;

  // Compute the new pressure value.
  return(pL + pR + pB + pT + pU + pD - dC) /6.0;
}
```

Listing 30-3 (*continued*). Modified Simulation Kernels to Account for Boundary Conditions

```
float4 GetObstacleVelocity(float3 cellTexCoords)
{
  return obstaclevelocity.Sample(samPointClamp, cellTexCoords);
}

float PS_DIVERGENCE_OBSTACLE(GS_OUTPUT_FLUIDSIM in,
                             Texture3D velocity) : SV_Target
{
  // Get velocity values from neighboring cells.
  float4 fieldL = velocity.Sample(samPointClamp, in.LEFTCELL);
  float4 fieldR = velocity.Sample(samPointClamp, in.RIGHTCELL);
  float4 fieldB = velocity.Sample(samPointClamp, in.BOTTOMCELL);
  float4 fieldT = velocity.Sample(samPointClamp, in.TOPCELL);
  float4 fieldD = velocity.Sample(samPointClamp, in.DOWNCELL);
  float4 fieldU = velocity.Sample(samPointClamp, in.UPCELL);

  // Use obstacle velocities for any solid cells.
  if(IsBoundaryCell(in.LEFTCELL))
    fieldL = GetObstacleVelocity(in.LEFTCELL);
  if(IsBoundaryCell(in.RIGHTCELL))
    fieldR = GetObstacleVelocity(in.RIGHTCELL);
  if(IsBoundaryCell(in.BOTTOMCELL))
    fieldB = GetObstacleVelocity(in.BOTTOMCELL);
  if(IsBoundaryCell(in.TOPCELL))
    fieldT = GetObstacleVelocity(in.TOPCELL);
  if(IsBoundaryCell(in.DOWNCELL))
    fieldD = GetObstacleVelocity(in.DOWNCELL);
  if(IsBoundaryCell(in.UPCELL))
    fieldU = GetObstacleVelocity(in.UPCELL);

  // Compute the velocity's divergence using central differences.
  float divergence =  0.5 * ((fieldR.x - fieldL.x) +
                             (fieldT.y - fieldB.y) +
                             (fieldU.z - fieldD.z));

  return divergence;
}
```

Listing 30-3 (*continued*). Modified Simulation Kernels to Account for Boundary Conditions

```
float4 PS_PROJECT_OBSTACLE(GS_OUTPUT_FLUIDSIM in,
                           Texture3D pressure,
                           Texture3D velocity): SV_Target
{
  // If the cell is solid, simply use the corresponding
  // obstacle velocity.
  if(IsBoundaryCell(in.CENTERCELL))
  {
    return GetObstacleVelocity(in.CENTERCELL);
  }

  // Get pressure values for the current cell and its neighbors.
  float pC = pressure.Sample(samPointClamp, in.CENTERCELL);
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);

  // Get obstacle velocities in neighboring solid cells.
  // (Note that these values are meaningless if a neighbor
  // is not solid.)
  float3 vL = GetObstacleVelocity(in.LEFTCELL);
  float3 vR = GetObstacleVelocity(in.RIGHTCELL);
  float3 vB = GetObstacleVelocity(in.BOTTOMCELL);
  float3 vT = GetObstacleVelocity(in.TOPCELL);
  float3 vD = GetObstacleVelocity(in.DOWNCELL);
  float3 vU = GetObstacleVelocity(in.UPCELL);

  float3 obstV = float3(0,0,0);
  float3 vMask = float3(1,1,1);

  // If an adjacent cell is solid, ignore its pressure
  // and use its velocity.
  if(IsBoundaryCell(in.LEFTCELL)) {
    pL = pC; obstV.x = vL.x; vMask.x = 0; }
  if(IsBoundaryCell(in.RIGHTCELL)) {
    pR = pC; obstV.x = vR.x; vMask.x = 0; }
```

Listing 30-3 (*continued*). Modified Simulation Kernels to Account for Boundary Conditions

```
  if(IsBoundaryCell(in.BOTTOMCELL)) {
    pB = pC; obstV.y = vB.y; vMask.y = 0; }
  if(IsBoundaryCell(in.TOPCELL)) {
    pT = pC; obstV.y = vT.y; vMask.y = 0; }
  if(IsBoundaryCell(in.DOWNCELL)) {
    pD = pC; obstV.z = vD.z; vMask.z = 0; }
  if(IsBoundaryCell(in.UPCELL)) {
    pU = pC; obstV.z = vU.z; vMask.z = 0; }

  // Compute the gradient of pressure at the current cell by
  // taking central differences of neighboring pressure values.
  float gradP = 0.5*float3(pR - pL, pT - pB, pU - pD);

  // Project the velocity onto its divergence-free component by
  // subtracting the gradient of pressure.
  float3 vOld = velocity.Sample(samPointClamp, in.texcoords);
  float3 vNew = vOld - gradP;

  // Explicitly enforce the free-slip boundary condition by
  // replacing the appropriate components of the new velocity with
  // obstacle velocities.
  vNew = (vMask * vNew) + obstV;

  return vNew;
}

bool IsNonEmptyCell(float3 cellTexCoords)
{
  return obstacles.Sample(samPointClamp, cellTexCoords, 0).r > 0.0);
}

float4 PS_ADVECT_OBSTACLE(GS_OUTPUT_FLUIDSIM in,
                          Texture3D velocity,
                          Texture3D color) : SV_Target
{
  if(IsNonEmptyCell(in.CENTERCELL))
  {
    return 0;
  }
```

Listing 30-3 (*continued*). Modified Simulation Kernels to Account for Boundary Conditions

```
    float3 cellVelocity = velocity.Sample(samPointClamp,
                                          in.CENTERCELL).xyz;
    float3 pos = in.cellIndex - timeStep*cellVelocity;

    float3 npos = float3(pos.x / textureWidth,
                         pos.y / textureHeight,
                         (pos.z+0.5) / textureDepth);

    return color.Sample(samLinear, npos);
}
```

Voxelization

To handle boundary conditions for dynamic solids, we need a quick way of determining whether a given cell contains a solid obstacle. We also need to know the solid's velocity for cells next to obstacle boundaries. To do this, we *voxelize* solid obstacles into an "inside-outside" texture and an "obstacle velocity" texture, as shown in Figure 30-8, using two different voxelization routines.

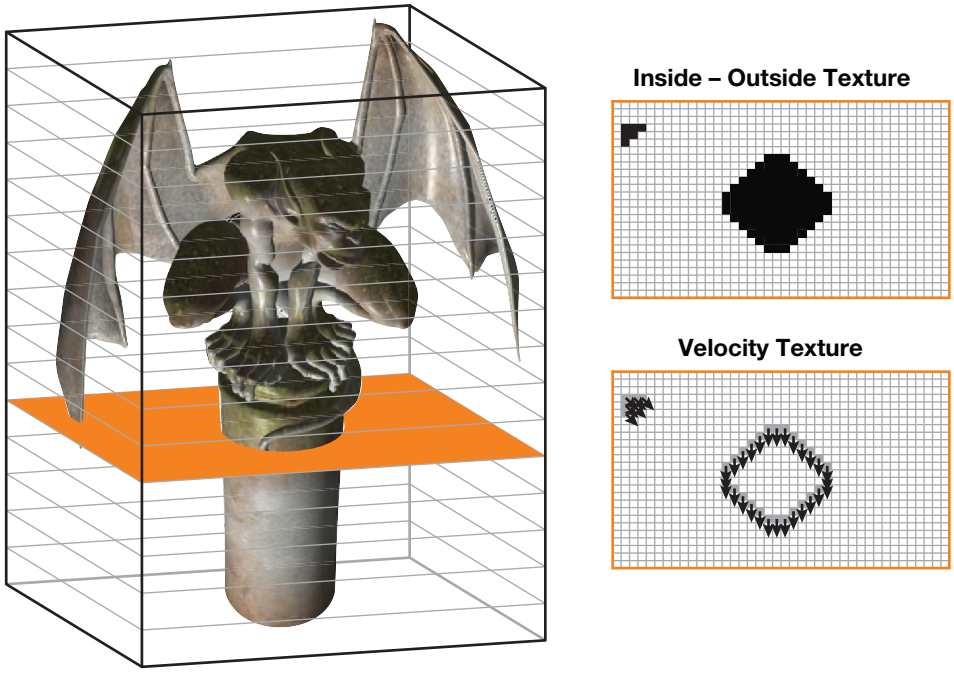


Figure 30-8. Solid Obstacles Are Voxelized into an Inside-Outside Texture and an Obstacle Velocity Texture

Inside-Outside Voxelization

Our approach to obtain an inside-outside voxelization is inspired by the *stencil shadow volumes* algorithm. The idea is simple: We render the input triangle mesh once into each slice of the destination 3D texture using an orthogonal projection. The far clip plane is set at infinity, and the near plane matches the depth of the current slice, as shown in Figure 30-9. When drawing geometry, we use a stencil buffer (of the same dimensions as the slice) that is initialized to zero. We set the stencil operations to *increment* for back faces and *decrement* for front faces (with wrapping in both cases). The result is that any voxel inside the mesh receives a nonzero stencil value. We then do a final pass that copies stencil values into the obstacle texture.[1]

As a result, we are able to distinguish among three types of cells: interior (nonzero stencil value), exterior (zero stencil), and interior but next to the boundary (these cells are tagged by the velocity voxelization algorithm, described next). Note that because this method depends on having one back face for every front face, it is best suited to watertight closed meshes.

Velocity Voxelization

The second voxelization algorithm computes an obstacle's velocity at each grid cell that contains part of the obstacle's boundary. First, however, we need to know the obstacle's velocity at each vertex. A simple way to compute per-vertex velocities is to store vertex positions $\mathbf{p}^{n-1}$ and $\mathbf{p}^{n}$ from the previous and current frames, respectively, in a vertex buffer. The instantaneous velocity $\mathbf{v}_i$ of vertex i can be approximated with the forward difference

$$\mathbf{v}_i = \frac{\mathbf{p}_i^n - \mathbf{p}_i^{n+1}}{\Delta t}$$

in a vertex shader.

Next, we must compute interpolated obstacle velocities for any grid cell containing a piece of a surface mesh. As with the inside-outside voxelization, the mesh is rendered once for each slice of the grid. This time, however, we must determine the intersection of each triangle with the current slice.

The intersection between a slice and a triangle is a segment, a triangle, a point, or empty. If the intersection is a segment, we draw a "thickened" version of the segment into the

1. We can also implement this algorithm to work directly on the final texture instead of using an intermediate stencil buffer. To do so, we can use additive blending. Additionally, if the interior is defined using the *even-odd rule* (instead of the *nonzero rule* we use), one can also use OpenGL's `glLogicOp`.

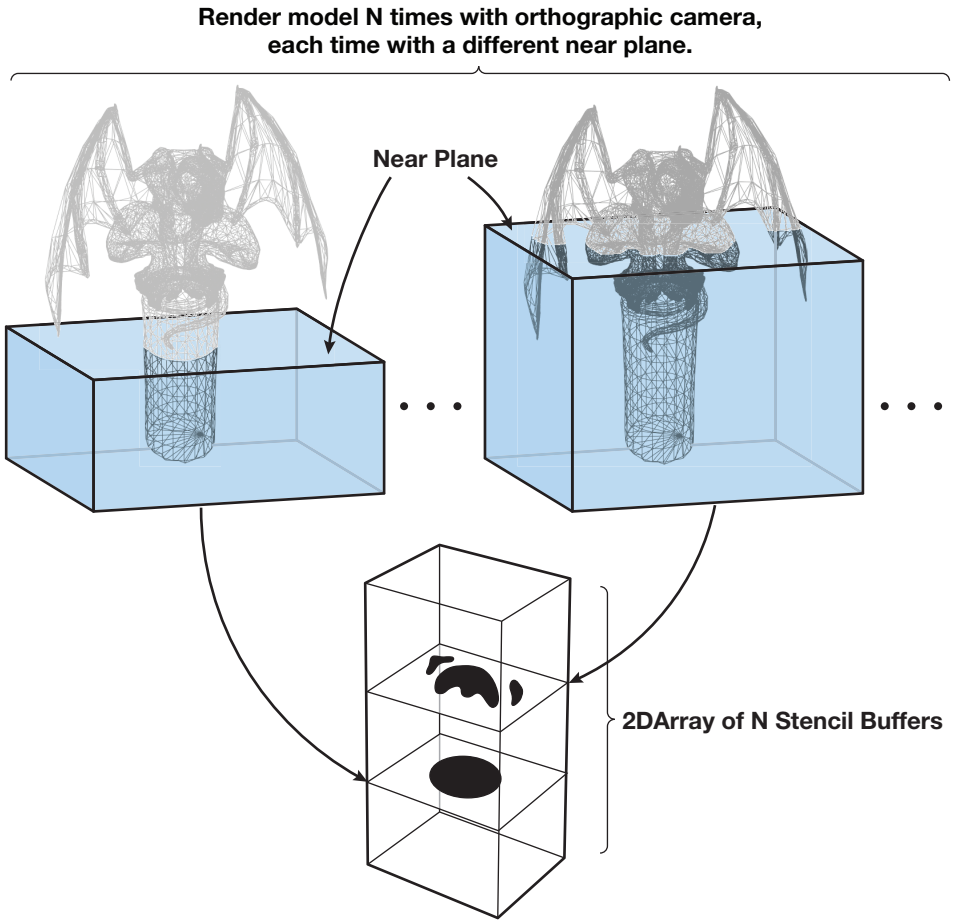


Figure 30-9. Inside-Outside Voxelization of a Mesh

slice using a quad. This quad consists of the two end points of the original segment and two additional points offset from these end points, as shown in Figure 30-10. The offset distance w is equal to the diagonal length of one texel in a slice of the 3D texture, and the offset direction is the projection of the triangle's normal onto the slice. Using linear interpolation, we determine velocity values at each end point and assign them to the corresponding vertices of the quad. When the quad is drawn, these values get interpolated across the grid cells as desired.

These quads can be generated using a geometry shader that operates on mesh triangles, producing four vertices if the intersection is a segment and zero vertices otherwise. Because geometry shaders cannot output quads, we must instead use a two-triangle

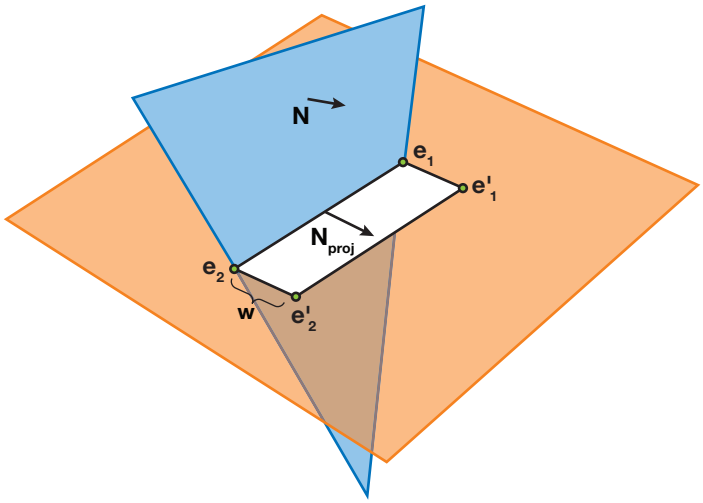


Figure 30-10. A Triangle Intersects a Slice at a Segment with End Points e_1 and e_2.
These end points are offset a distance w in the direction of the projected normal N_{proj} to get the other two vertices of the quad, e_1' and e_2'.

strip. To compute the triangle-slice intersection, we intersect each triangle edge with the slice. If exactly two edge-slice intersections are found, the corresponding intersection points are used as end points for our segment. Velocity values at these points are computed via interpolation along the appropriate triangle edges. The geometry shader `GS_GEN_BOUNDARY_VELOCITY` in Listing 30-4 gives an implementation of this algorithm. Figure 30-12 shows a few slices of a voxel volume resulting from the voxelization of the model in Figure 30-11.

Listing 30-4. Geometry Shader for Velocity Voxelization

```
// GS_GEN_BOUNDARY_VELOCITY:
// Takes as input:
//  - one triangle (3 vertices),
//  - the sliceIdx,
//  - the sliceZ;
// and outputs:
//  - 2 triangles, if intersection of input triangle with slice
//    is a segment
//  - 0 triangles, otherwise
// The 2 triangles form a 1-voxel wide quadrilateral along the
// segment.
```

Figure 30-11. Simplified Geometry Can Be Used to Speed Up Voxelization

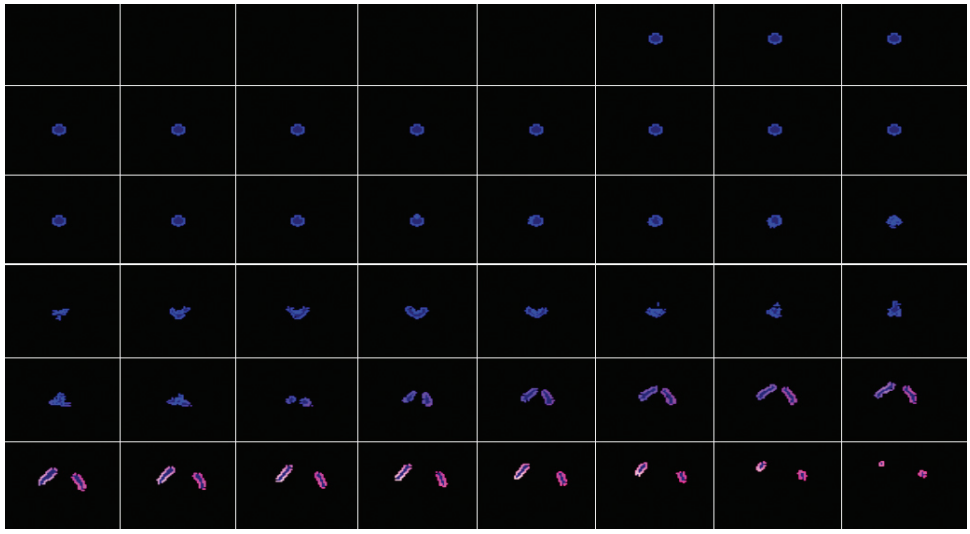


Figure 30-12. Slices of the 3D Textures Resulting from Applying Our Voxelization Algorithms to the Model in Figure 30-11.
The blue channel shows the result of the inside-outside voxelization (bright blue for cells next to the boundary and dark blue for other cells inside). The red and green channels are used to visualize two of the three components of the velocity.

Listing 30-4 (*continued*). Geometry Shader for Velocity Voxelization

```
[maxvertexcount (4)]
void GS_GEN_BOUNDARY_VELOCITY(
  triangle VsGenVelOutput input[3],
  inout TriangleStream<GsGenVelOutput> triStream)
{
  GsGenVelOutput output;
  output.RTIndex = sliceIdx;

  float minZ = min(min(input[0].Pos.z, input[1].Pos.z), input[2].Pos.z);
  float maxZ = max(max(input[0].Pos.z, input[1].Pos.z), input[2].Pos.z);
  if((sliceZ < minZ) || (sliceZ > maxZ))
    // This triangle doesn't intersect the slice.
    return;

  GsGenVelIntVtx intersections[2];
  for(int i=0; i<2; i++)
  {
    intersections[i].Pos = 0;
    intersections[i].Velocity = 0;
  }

  int idx = 0;
  if(idx < 2)
    GetEdgePlaneIntersection(input[0], input[1], sliceZ,
                             intersections, idx);
  if(idx < 2)
    GetEdgePlaneIntersection(input[1], input[2], sliceZ,
                             intersections, idx);
  if(idx < 2)
    GetEdgePlaneIntersection(input[2], input[0], sliceZ,
                             intersections, idx);

  if(idx < 2)
    return;

  float sqrtOf2 = 1.414; // The diagonal of a pixel
  float2 normal = sqrtOf2 * normalize(
    cross((input[1].Pos - input[0].Pos),
          (input[2].Pos - input[0].Pos)).xy);
```

Listing 30-4 (*continued*). Geometry Shader for Velocity Voxelization

```
  for(int i=0; i<2; i++)
  {
    output.Pos = float4(intersections[i].Pos, 0, 1);
    output.Velocity = intersections[i].Velocity;
    triStream.Append(output);

    output.Pos = float4((intersections[i].Pos +
                         (normal*projSpacePixDim)), 0, 1);
    output.Velocity = intersections[i].Velocity;
    triStream.Append(output);
  }
  triStream.RestartStrip();
}

void GetEdgePlaneIntersection(
  VsGenVelOutput vA,
  VsGenVelOutput vB,
  float sliceZ,
  inout GsGenVelIntVtx intersections[2],
  inout int idx)
{
  float t = (sliceZ - vA.Pos.z) / (vB.Pos.z - vA.Pos.z);
  if((t < 0) || (t > 1))
    // Line-plane intersection is not within the edge's end points
    // (A and B)
    return;

    intersections[idx].Pos = lerp(vA.Pos, vB.Pos, t).xy;
    intersections[idx].Velocity = lerp(vA.Velocity, vB.Velocity, t);
    idx++;
}
```

Optimizing Voxelization

Although voxelization requires a large number of draw calls, it can be made more efficient using *stream output* (see Blythe 2006). Stream output allows an entire buffer of transformed vertices to be cached when voxelizing deforming meshes such as skinned characters, rather than recomputing these transformations for each slice.

Additionally, instancing can be used to draw all slices in a single draw call, rather than making a separate call for each slice. In this case, the *instance ID* can be used to specify the target slice.

Due to the relative coarseness of the simulation grid used, it is a good idea to use a low level of detail mesh for each obstacle, as shown in Figure 30-11. Using simplified models allowed us to voxelize obstacles at every frame with little performance cost.

Finally, if an obstacle is transformed by a simple analytic transformation (versus a complex skinning operation, for example), voxelization can be precomputed and the *inverse* of the transformation can be applied whenever accessing the 3D textures. A simple example is a mesh undergoing rigid translation and rotation: texture coordinates used to access the inside-outside and obstacle velocity textures can be multiplied by the inverse of the corresponding transformation matrix to get the appropriate values.

30.2.5 Smoke

Although the velocity field describes the fluid's motion, it does not look much like a fluid when visualized directly. To get interesting visual effects, we must keep track of additional quantities that are pushed around by the fluid. For instance, we can keep track of density and temperature to obtain the appearance of smoke (Fedkiw et al. 2001). For each additional quantity ϕ, we must allocate an additional texture with the same dimensions as our grid. The evolution of values in this texture is governed by the same advection equation used for velocity:

$$\frac{\partial \phi}{\partial t} = -(\mathbf{u} \cdot \nabla)\phi.$$

In other words, we can use the same MacCormack advection routine we used to evolve the velocity.

To achieve the particular effect seen in Figure 30-4, for example, we inject a three-dimensional Gaussian "splat" into a *color* texture each frame to provide a source of "smoke." These color values have no real physical significance, but they create attractive swirling patterns as they are advected throughout the volume by the fluid velocity.

To get a more physically plausible appearance, we must make sure that hot smoke rises and cool smoke falls. To do so, we need to keep track of the fluid temperature T (which again is advected by $\mathbf{u}$). Unlike color, temperature values have an influence on the dynamics of the fluid. This influence is described by the *buoyant force*:

$$\mathbf{f}_{buoyancy} = \frac{Pmg}{R}\left(\frac{1}{T_0} - \frac{1}{T}\right)\mathbf{z},$$

where P is pressure, m is the molar mass of the gas, g is the acceleration due to gravity, and R is the universal gas constant. In practice, all of these physical constants can be treated as a single value and can be tweaked to achieve the desired visual appearance. The value T_0 is the ambient or "room" temperature, and T represents the temperature values being advected through the flow. $\mathbf{z}$ is the normalized upward-direction vector. The buoyant force should be thought of as an "external" force and should be added to the velocity field immediately following velocity advection.

30.2.6 Fire

Fire is not very different from smoke except that we must store an additional quantity, called the *reaction coordinate*, that keeps track of the time elapsed since gas was ignited. A reaction coordinate of one indicates that the gas was just ignited, and a coordinate of less than zero indicates that the fuel has been completely exhausted. The evolution of these values is described by the following equation (from Nguyen et al. 2002):

$$\frac{\partial}{\partial t}Y = -(u \cdot \nabla)Y - k.$$

In other words, the reaction coordinate is advected through the flow and decremented by a constant amount (k) at each time step. In practice, this integration is performed by passing a value for k to the advection routine (`PS_ADVECT_MACCORMACK`), which is added to the result of the advection. (This value should be nonzero only when advecting the reaction coordinate.) Reaction coordinates do not have an effect on the dynamics of the fluid but are later used for rendering (see Section 30.3).

Figure 30-14 (in Section 30.2.10) demonstrates one possible fire effect: a ball of fuel is continuously generated near the bottom of the volume by setting the reaction coordinate in a spherical region to one. For a more advanced treatment of flames, see Nguyen et al. 2002.

30.2.7 Water

Water is modeled differently from the fluid phenomena discussed thus far. With fire or smoke, we are interested in visualizing a density defined throughout the entire volume, but with water the visually interesting part is the *interface* between air and liquid.

Therefore, we need some way of representing this interface and tracking how it deforms as it is pushed around by the fluid velocity.

The *level set method* (Sethian 1999) is a popular representation of a liquid surface and is particularly well suited to a GPU implementation because it requires only a scalar value at each grid cell. In a level set, each cell records the *shortest signed distance* ϕ from the cell center to the water surface. Cells in the grid are classified according to the value of ϕ: if $\phi < 0$, the cell contains water; otherwise, it contains air. Wherever ϕ equals zero is exactly where the water meets the air (the *zero set*). Because advection will not preserve the distance field property of a level set, it is common to periodically *reinitialize* the level set. Reinitialization ensures that each cell does indeed store the shortest distance to the zero set. However, this property isn't needed to simply define the surface, and for real-time animation, it is possible to get decent results without reinitialization. Figure 30-1, at the beginning of this chapter, shows the quality of the results.

Just as with color, temperature, and other attributes, the level set is advected by the fluid, but it also affects the simulation dynamics. In fact, the level set *defines* the fluid domain: in simple models of water and air, we assume that the air has a negligible effect on the liquid and do not perform simulation wherever $\phi \geq 0$. In practice, this means we set the pressure outside of the liquid to zero before solving for pressure and modify the pressure only in liquid cells. It also means that we do not apply external forces such as gravity outside of the liquid. To make sure that only fluid cells are processed, we check the value of the level set texture in the relevant shaders and mask computations at a cell if the value of ϕ is above some threshold. Two alternatives that may be more efficient are to use z-cull to eliminate cells (if the GPU does not support dynamic flow control) (Sander et al. 2004) and to use a sparse data structure (Lefohn et al. 2004).

30.2.8 Performance Considerations

One major factor in writing an efficient solver is bandwidth. For each frame of animation, the solver runs a large number of arithmetically simple kernels, in between which data must be transferred to and from texture memory. Although most of these kernels exhibit good locality, bandwidth is still a major issue: using 32-bit floating-point textures to store quantities yields roughly half the performance of 16-bit textures. Surprisingly, there is little visually discernible degradation that results from using 16-bit storage, as is shown in Figure 30-13. Note that arithmetic operations are still performed

in 32-bit floating point, meaning that results are periodically truncated as they are written to the destination textures.

In some cases it is tempting to store multiple cell attributes in a single texture in order to reduce memory usage or for convenience, but doing so is not always optimal in terms of memory bandwidth. For instance, suppose we packed both inside-outside and velocity information about an obstacle into a single RGBA texture. Iteratively solving the pressure-Poisson equation requires that we load inside-outside values numerous times each frame, but meanwhile the obstacle's velocity would go unused. Because packing these two textures together requires four times as many bytes transferred from memory as well as cache space, it may be wise to keep the obstacle's inside-outside information in its own scalar texture.

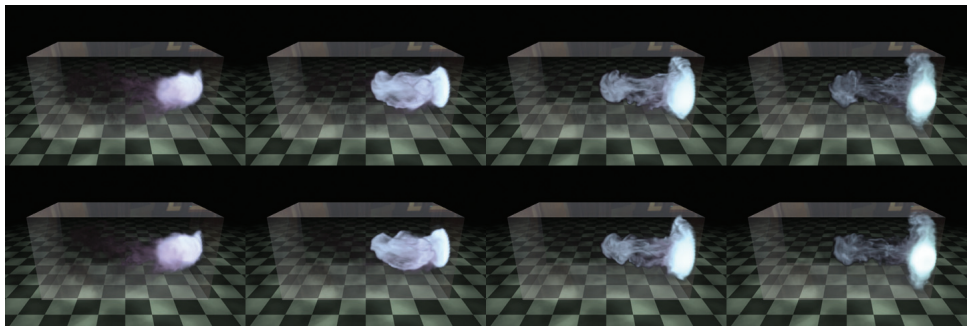


Figure 30-13. Smoke Simulated Using 16-Bit (*Top Row*) and 32-Bit (*Bottom Row*) Floating-Point Textures for Storage
Note that although some fine-scale detail differs between the two sequences, the overall motion is consistent.

30.2.9 Storage

Table 30-1 gives some of the storage requirements needed for simulating and rendering fluids, which amounts to 41 bytes per cell for simulation and 20 bytes per pixel for rendering. However, most of this storage is required only temporarily while simulating or rendering, and hence it can be shared among multiple fluid simulations. In *Hellgate: London*, we stored the exclusive textures (the third column of the table) with each instance of smoke, but we created global versions of the rest of the textures (the last column of the table), which were shared by all the smoke simulations.

Table 30-1. Storage Needed for Simulating and Rendering Fluids

	Total Space	Exclusive Textures	Shared Textures
Fluid Simulation	32 bytes per cell	12 bytes per cell 1×RGBA16 (velocity) 2×R16 (pressure and density)	20 bytes per cell 2×RGBA16 (temporary) 2×R16 (temporary)
Voxelization	9 bytes per cell	—	9 bytes per cell 1×RGBA16 (velocity) 1×R8 (inside-outside)
Rendering	20 bytes per pixel	—	20 bytes per pixel of off-screen render target 1×RGBA32 (ray data) 1×R32 (scene depth)

30.2.10 Numerical Issues

Because real-time applications are so demanding, we have chosen the simplest numerical schemes that still give acceptable visual results. Note, however, that for high-quality animation, more accurate alternatives are preferable.

One of the most expensive parts of the simulation is solving the pressure-Poisson system, $\nabla^2 p = \nabla \cdot \mathbf{u}^*$. We use the Jacobi method to solve this system because it is easy to implement efficiently on the GPU. However, several other suitable solvers have been implemented on the GPU, including the Conjugate Gradient method (Bolz et al. 2003) and the Multigrid method (Goodnight et al. 2003). Cyclic reduction is a particularly interesting option because it is direct and can take advantage of banded systems (Kass et al. 2006). When picking an iterative solver, it may be worth considering not only the overall rate of convergence but also the convergence rate of different *spatial frequencies* in the residual (Briggs et al. 2000). Because there may not be enough time to reach convergence in a real-time application, the distribution of frequencies will have some impact on the appearance of the solution.

Ideally we would like to solve the pressure-Poisson system exactly in order to satisfy the incompressibility constraint and preserve fluid volume. For fluids like smoke and fire, however, a change in volume does not always produce objectionable visual artifacts. Hence we can adjust the number of iterations when solving this system according to available resources. Figure 30-14 compares the appearance of a flame using different numbers of Jacobi iterations. Seen in motion, spinning vortices tend to "flatten out" a bit more when using only a small number of iterations, but the overall appearance is very similar.

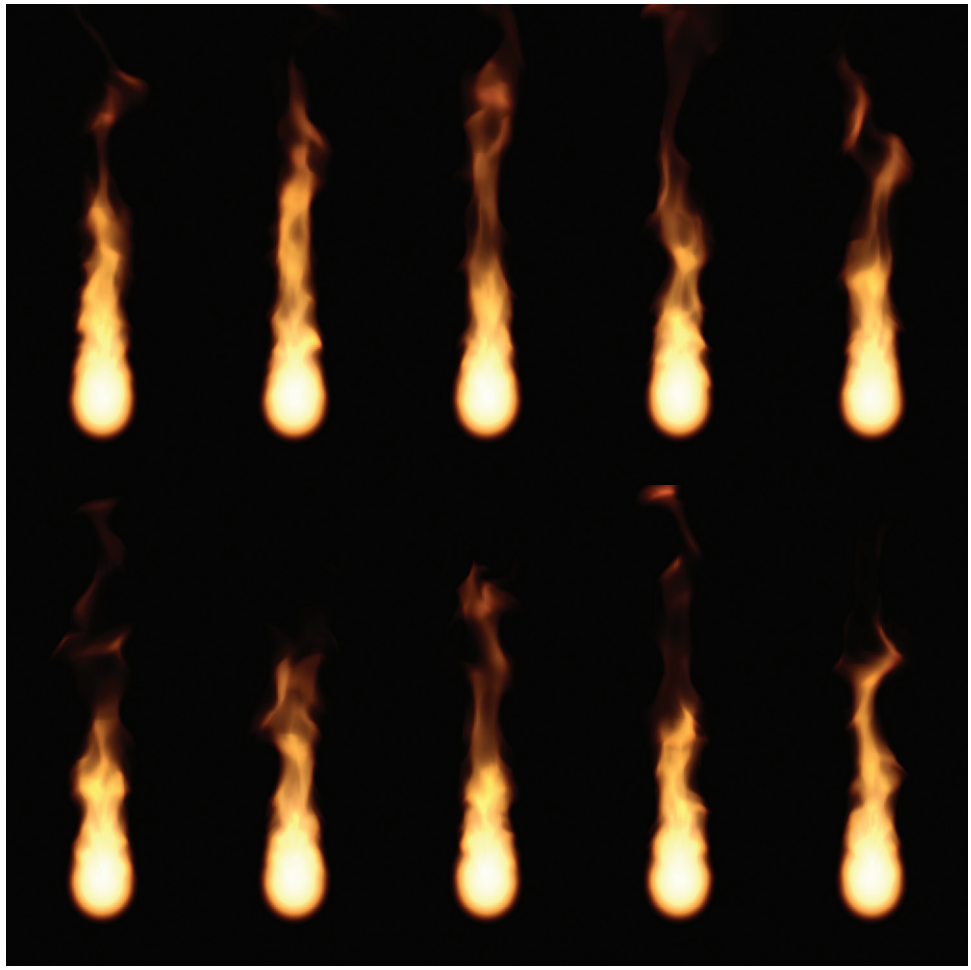


Figure 30-14. Fire Simulation Using 20 Jacobi Iterations (*Top Row*) and 1,000 Jacobi Iterations (*Bottom Row*) for the Pressure Solve

For a more thorough discussion of GPGPU performance issues, see Pharr 2005.

For liquids, on the other hand, a change of volume is immediately apparent: fluid appears to either pour out from nowhere or disappear entirely! Even something as simple as water sitting in a tank can potentially be problematic if too few iterations are used to solve for pressure: because information does not travel from the tank floor to the water surface, pressure from the floor cannot counteract the force of gravity. As a result, the water slowly sinks through the bottom of the tank, as shown in Figure 30-15.

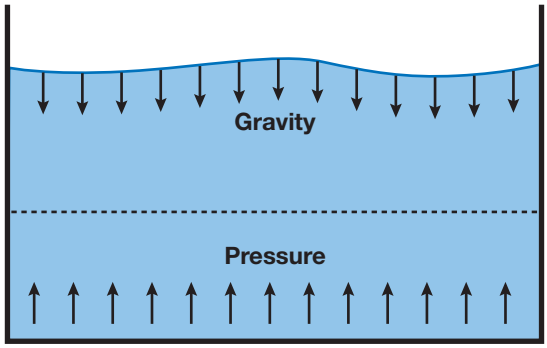


Figure 30-15. Uncorrected Water Simulation
Pressure pushing up from the bottom of the tank may not be able to counteract the force of gravity on the liquid's surface when using a small number of Jacobi iterations to solve for pressure.

Unfortunately, in a real-time application, it is not always possible to solve for p exactly (regardless of the particular solver used) because computation time is constrained by the target frame rate and the resource requirements of other concurrent processes. In simple situations where we know that the liquid should tend toward a static equilibrium, we can force the correct behavior by manipulating the level set in the following way:

$$\phi_{i,j,k}^{n+1} = \begin{cases} A\left(\phi^n\right)_{i,j,k} & \phi_{i,j,k}^{\infty} \geq 0 \\ (1-\beta)A\left(\phi^n\right)_{i,j,k} + \beta\phi_{i,j,k}^{\infty} & \phi_{i,j,k}^{\infty} < 0 \end{cases}$$

Here ϕ^{∞} is a level set whose zero set tells us what the surface *should* look like if we let the liquid settle for a long period of time. For example, the equilibrium level set for a tank of water would be simply $\phi^{\infty}(x, y, z) = y - h$, where y is the vertical distance from the bottom of the tank and h is the desired height of the water. See Figure 30-16.

The function A is the advection operator, and the parameter $\beta \in [0, 1]$ controls the amount of *damping* applied to the solution we get from advection. Larger values of β permit fewer solver iterations but also decrease the liveliness of the fluid. Note that this damping is applied only in regions of the domain where ϕ^{∞} is *negative*—this keeps splashes evolving outside of the domain of the equilibrium solution lively, though it can result in temporary volume *gain*.

Ultimately, however, this kind of nonphysical manipulation of the level set is a hack, and its use should be considered judiciously. Consider an environment in which the player scoops up water with a bowl and then sets the bowl down at an arbitrary location on a table: we do not know beforehand what the equilibrium level set should look like and hence cannot prevent water from sinking through the bottom of the bowl.

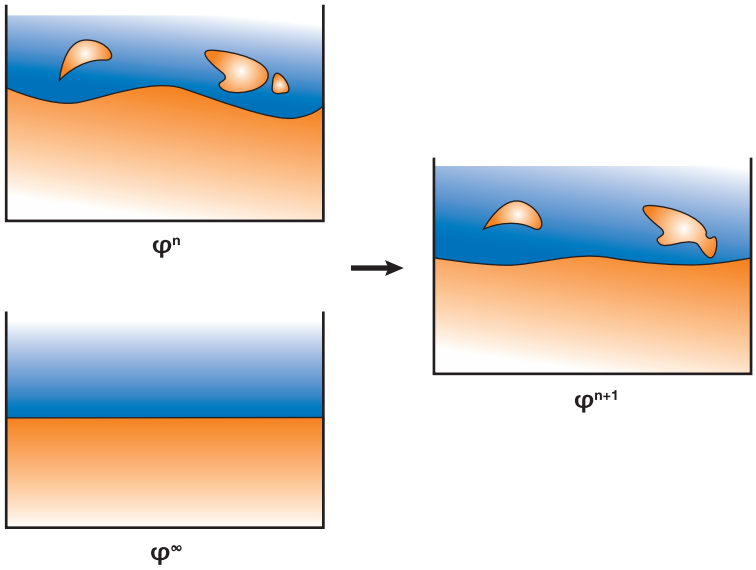


Figure 30-16. Combining Level Sets to Counter a Low Convergence Rate
To preserve fluid volume even under extreme performance constraints, the results of level set advection are combined with a known equilibrium level set ϕ^{∞}.

30.3 Rendering

30.3.1 Volume Rendering

The result of our simulation is a collection of values stored in a 3D texture. However, there is no mechanism in Direct3D or OpenGL for displaying this texture directly. Therefore we render the fluid using a *ray-marching* pixel shader. Our approach is very similar to the one described in Scharsach 2005.

The placement of the fluid in the scene is determined by six quads, which represent the faces of the simulation volume. These quads are drawn into a deferred shading buffer to determine where and how rays should be cast. We then render the fluid by marching rays through the volume and accumulating densities from the 3D texture, as shown in Figure 30-17.

Volume Ray Casting

In order to cast a ray, we need to know where it enters the volume, in which direction it is traveling, and how many samples to take. One way to get these values is to perform several ray-plane intersections in the ray-marching shader. However, precomputing these values and storing them in a texture makes it easier to perform proper compositing and

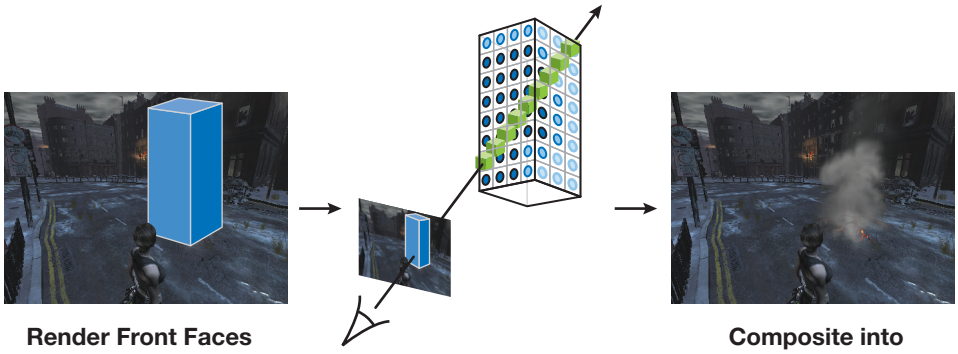


Figure 30-17. A Conceptual Overview of Ray Casting

clipping (more on this later in this section), which is the approach we use here. As a pre-pass, we generate a screen-size texture, called the *RayData* texture, which encodes, for every pixel that is to be rendered, the entry point of the ray in texture space, and the depth through the volume that the ray traverses. To get the depth through the volume, we draw first the back faces of the volume with a shader that outputs the distance from the eye to the fragment's position (in view space) into the alpha channel. We then run a similar shader on the front faces but enable subtractive blending using Equation 1. Furthermore, to get the entry point of the ray, we also output into the RGB channel the texture-space coordinates of each fragment generated for the front faces.

$$\begin{aligned} OutputColor.rgb &= SourceColor.rgb \\ OutputColor.a &= DestinationColor.a - SourceColor.a \end{aligned} \tag{1}$$

To render the volume, we draw a full-screen quad with a ray-marching shader. This shader looks up into the *RayData* texture to find the pixels that we need to ray-cast through, and the ray entry point and marching distance through the volume for those pixels. The number of samples that the ray-marching shader uses is proportional to the marching distance (we use a step size equal to half a voxel). The ray direction is given by the vector from the eye to the entry point (both in texture space). At each step along the ray, we sample values from the texture containing the simulated values and blend them front to back according to Equation 2. By blending from front to back, we can terminate ray casting early if the color saturates (we exit if $FinalColor.a > 0.99$). For a more physically based rendering model, see Fedkiw et al. 2001.

$$FinalColor.rgb \mathrel{+}= SampleColor.rgb \times SampleColor.a \times (1 - FinalColor.a)$$
$$FinalColor.a \mathrel{+}= SampleColor.a \times (1 - FinalColor.a) \quad (2)$$

Compositing

There are two problems with the ray-marching algorithm described so far. First, rays continue to march through the volume even if they encounter other scene geometry. See the right side of Figure 30-18 for an illustration. Second, rays are traced even for parts of the volume that are completely occluded, as the left side of Figure 30-18 shows. However, we can modify our computation of volume depth such that we march through only relevant parts of the grid.

Previously we used the distance to the back faces of the volume to determine where ray marching should terminate. To handle obstacles that intersect the volume, we instead use the minimum of the *back-face distance* and the *scene distance* (that is, the distance between the eye and the closest obstacle in the scene). The scene distance can be calculated by reading the scene depth and reverse projecting it back to view space to find the distance from the eye to the scene. If the depth buffer cannot be read as a texture in a pixel shader, as is the case in Direct3D 10 when using multisample antialiasing, this distance can be computed in the main scene rendering pass using multiple render targets; this is the approach we use.

To deal with cases in which the scene geometry completely occludes part of the volume, we compare the front-facing fragments' distance to the *scene distance*. If the front-face distance is greater than the scene distance (that is, the fragment is occluded), we output

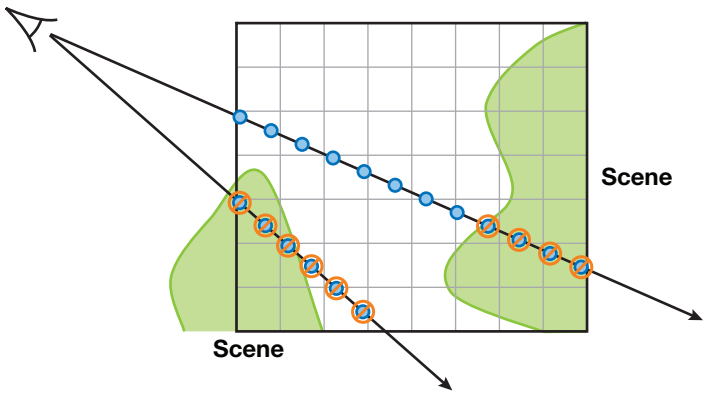


Figure 30-18. Rays Are Clipped According to Scene Depth to Account for Occlusion

a large negative value in the red channel. This way, the final texture-space position computed for the corresponding texel in the *RayData* texture will be outside the volume, and hence no samples will be taken along the corresponding ray.

Clipping

We also need to modify our ray-marching algorithm to handle the cases in which the camera is located *inside* the fluid volume and the camera's near plane clips parts of the front faces of the volume, as shown in Figure 30-19.

In regions where the front faces were clipped, we have no information about where rays enter the volume, and we have incorrect values for the volume depth.

To deal with these regions, we mark the pixels where the back faces of the volume have been rendered but not the front faces. This marking is done by writing a negative color value into the green channel when rendering the back faces of the fluid volume to the *RayData* texture. Note that the *RayData* texture is cleared to zero before either front or back faces are rendered to it. Because we do not use the RGB values of the destination color when rendering the front faces with alpha blending (Equation 1), the pixels for which the green channel contains a negative color after rendering the front faces are those where the back faces of the fluid volume were rendered but not the front (due to

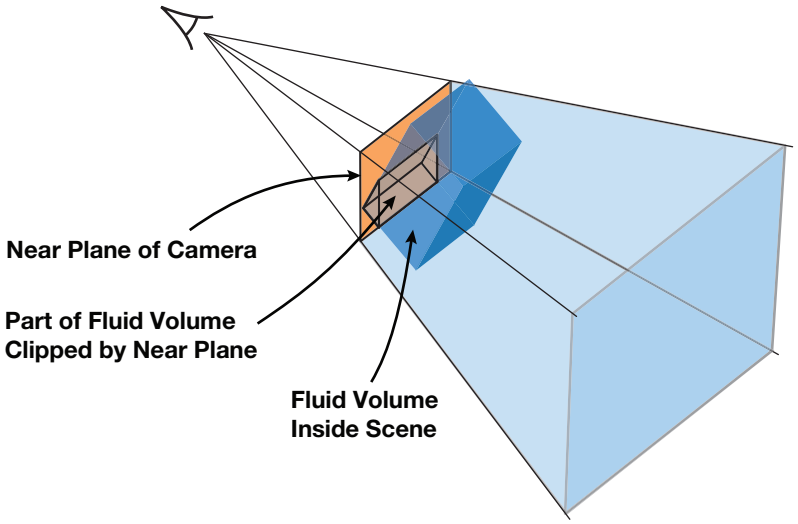


Figure 30-19. Part of the Fluid Volume May Be Clipped by the Near Plane
In areas where the front faces of the fluid volume get clipped by the near plane of the camera, we have incorrect information for ray marching.

clipping). In the ray-casting shader, we explicitly initialize the position of these marked pixels to the appropriate texture-space position of the corresponding point on the near plane. We also correct the depth value read from the *RayData* texture by subtracting from it the distance from the eye to the corresponding point on the near plane.

Filtering

The ray-marching algorithm presented so far has several visible artifacts. One such artifact is *banding*, which results from using an integral number of equally spaced samples. This artifact is especially visible when the scene inside the fluid volume has rapidly changing depth values, as illustrated in Figure 30-20.

To suppress it, we take one more sample than necessary and weigh its contribution to the final color by $d/sampleWidth$, as shown in Figure 30-21. In the figure, d is the difference between the scene distance at the fragment and the total distance traveled by the ray at the last sample, and *sampleWidth* is the typical step size along the ray.

Banding, however, usually remains present to some degree and can become even more obvious with high-frequency variations in either the volume density or the mapping between density and color (known as the *transfer function*). This well-known problem is addressed in Hadwiger 2004 and Sigg and Hadwiger 2005. Common solutions include increasing the sampling frequency, jittering the samples along the ray direction, or using higher-order filters when sampling the volume. It is usually a good idea to combine several of these techniques to find a good performance-to-quality trade-off. In *Hellgate: London*, we used trilinear jittered sampling at a frequency of twice per voxel.

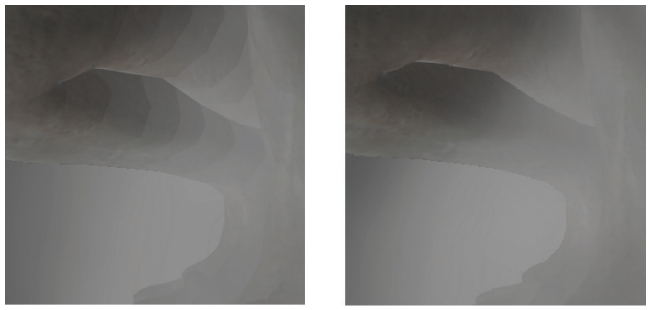


Figure 30-20. Dealing with Banding
Using scene depth can cause banding artifacts (left), which can be solved using weighted sampling (right).

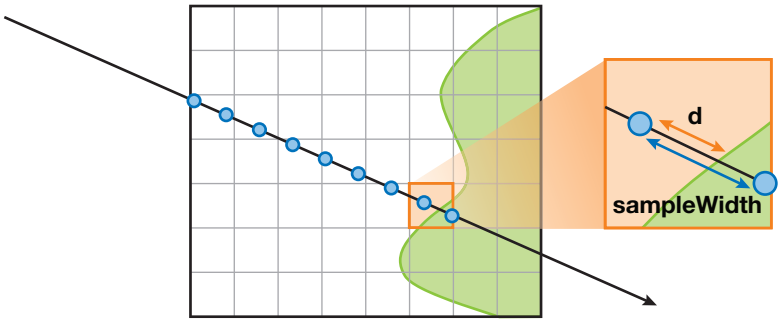


Figure 30-21. Reducing Banding by Taking an Additional Weighted Sample
Taking an additional weighted sample can help reduce banding artifacts such as those seen in Figure 30-20.

Off-Screen Ray Marching

If the resolution of the simulation grid is low compared to screen resolution, there is little visual benefit in ray casting at high resolution. Instead, we draw the fluid into a smaller off-screen render target and then composite the result into the final image. This approach works well except in areas of the image where there are sharp depth discontinuities in scene geometry, as shown in Figure 30-22, and where the camera clips the fluid volume.

This issue is discussed in depth by Iain Cantlay in Chapter 23 of this book, "High-Speed, Off-Screen Particles." In *Hellgate: London*, we use a similar approach to the one presented there: we draw most of the smoke at a low resolution but render pixels in problematic areas at screen resolution. We find these areas by running an edge-

Figure 30-22. Fixing Artifacts Introduced by Low-Resolution Off-Screen Rendering
Left: Ray marching at a low resolution and upsampling can cause artifacts near sharp silhouettes. Center: Detecting these features and rendering the corresponding fragments at higher resolution. Right: The resulting artifact-free image.

detection filter on the *RayData* texture computed earlier in this section. Specifically, we run a Sobel edge-detection filter on the texture's alpha channel (to find edges of obstacles intersecting the volume), red channel (to find edges of obstacles occluding the volume), and green channel (to find the edges where the near plane of the camera clips the volume).

Fire

Rendering fire is similar to rendering smoke except that instead of blending values as we march, we *accumulate* values that are determined by the reaction coordinate Y rather than the smoke density (see Section 30.2.6). In particular, we use an artist-defined 1D texture that maps reaction coordinates to colors in a way that gives the appearance of fire. A more physically based discussion of fire rendering can be found in Nguyen et al. 2002.

The fire volume can also be used as a light source if desired. The simplest approach is to sample the volume at several locations and treat each sample as a point light source. The reaction coordinate and the 1D color texture can be used to determine the intensity and color of the light. However, this approach can lead to severe flickering if not enough point samples are used, and it may not capture the overall behavior of the light. A different approach is to downsample the texture of reaction coordinates to an extremely low resolution and then use *every* voxel as a light source. The latter approach will be less prone to flickering, but it won't capture any high-frequency lighting effects (such as local lighting due to sparks).

30.3.2 Rendering Liquids

To render a liquid surface, we also march through a volume, but this time we look at values from the level set ϕ. Instead of integrating values as we march, we look for the first place along the ray where $\phi = 0$. Once this point is found, we shade it just as we would shade any other surface fragment, using $\nabla\phi$ at that point to approximate the shading normal. For water, it is particularly important that we do not see artifacts of the grid resolution, so we use tricubic interpolation to filter these values. Figure 30-1 at the beginning of the chapter demonstrates the rendered results. See Sigg and Hadwiger 2005 and Hadwiger et al. 2005 for details on how to quickly intersect and filter volume isosurface data such as a level set on the GPU.

Refraction

For fluids like water, there are several ways to make the surface appear as though it refracts the objects behind it. Ray tracing is one possibility, but casting rays is expensive,

and there may be no way to find ray intersections with other scene geometry. Instead, we use an approximation that gives the impression of refraction but is fast and simple to implement.

First, we render the objects behind the fluid volume into a background texture.

Next, we determine the nearest ray intersection with the water surface at every pixel by marching through the volume. This produces a pair of textures containing hit locations and shading normals; the alpha value in the texture containing hit locations is set to zero if there was no ray-surface intersection at a pixel, and set to one otherwise. We then shade the hit points with a refraction shader that uses the background texture. Finally, foreground objects are added to create the final image.

The appearance of refraction is achieved by looking up a pixel in the background image near the point being shaded and taking its value as the refracted color. This refracted color is then used in the shading equation as usual. More precisely, this background pixel is accessed at a texture coordinate $\mathbf{t}$ that is equal to the location $\mathbf{p}$ of the pixel being shaded offset by a vector proportional to the projection of the surface normal $\mathbf{N}$ onto the image plane. In other words, if $\mathbf{P}_h$ and $\mathbf{P}_v$ are an orthonormal basis for the image plane oriented with the viewport, then

$$\mathbf{t} = \mathbf{p} - \beta(\mathbf{N} \cdot \mathbf{P}_h,\ \mathbf{N} \cdot \mathbf{P}_v),$$

where $\beta > 0$ is a scalar parameter that controls the severity of the effect. The vectors $\mathbf{P}_v$ and $\mathbf{P}_h$ are defined by

$$\mathbf{P}_v = \frac{\mathbf{z} - (\mathbf{z} \cdot \mathbf{V})\mathbf{V}}{\|\mathbf{z} - (\mathbf{z} \cdot \mathbf{V})\mathbf{V}\|}$$

$$\mathbf{P}_h = \mathbf{P}_v \times \mathbf{V},$$

where $\mathbf{z}$ is up and $\mathbf{V}$ is the view direction.

The effect of applying this transformation to the texture coordinates is that a convex region of the surface will magnify the image behind it, a concave region will shrink the image, and flat (with respect to the viewer) regions will allow rays to pass straight through.

30.4 Conclusion

In this chapter, we hope to have demonstrated that physically based fluid animation is a valuable tool for creating interactive environments, and to have provided some of the

basic building blocks needed to start developing a practical implementation. However, this is by no means the end of the line: we have omitted discussion of a large number of possible directions for fluid animation, including melting (Carlson et al. 2002), viscoelastic fluids (Goktekin et al. 2004), and multiphase flows (Lossasso et al. 2006). We have also omitted discussion of a number of interesting data structures and algorithms, such as sparse level sets (Lefohn et al. 2004), which may significantly improve simulation performance; or mesh-based surface extraction (Ziegler et al. 2006), which may permit more efficient rendering of liquids.

30.5 References

Blythe, David. 2006. "The Direct3D 10 System." In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)* 25(3), pp. 724–734.

Bolz, J., I. Farmer, E. Grinspun, and P. Schröder. 2003. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid." In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), pp. 917–924.

Bridson R., R. Fedkiw, and M. Muller-Fischer. 2006. "Fluid Simulation." SIGGRAPH 2006 Course Notes. In *ACM SIGGRAPH 2006 Courses*.

Briggs, William L., Van Emden Henson, and Steve F. McCormick. 2000. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics.

Carlson, M., P. Mucha, R. Van Horn, and G. Turk. 2002. "Melting and Flowing." In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Fang, S., and H. Chen. 2000. "Hardware Accelerated Voxelization." *Computers and Graphics* 24(3), pp. 433–442.

Fedkiw, R., J. Stam, and H. W. Jensen. 2001. "Visual Simulation of Smoke." In *Proceedings of SIGGRAPH 2001*, pp. 15–22.

Foster, N., and R. Fedkiw. 2001. "Practical Animation of Liquids." In *Proceedings of SIGGRAPH 2001*.

Goktekin, T. G., A.W. Bargteil, and J. F. O'Brien. 2004. "A Method for Animating Viscoelastic Fluids." In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3).

Goodnight, N., C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. 2003. "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware." In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 102–111.

Hadwiger, M. 2004. "High-Quality Visualization and Filtering of Textures and Segmented Volume Data on Consumer Graphics Hardware." Ph.D. Thesis.

Hadwiger, M., C. Sigg, H. Scharsach, K. Buhler, and M. Gross. 2005. "Real-time Raycasting and Advanced Shading of Discrete Isosurfaces." In *Proceedings of Eurographics 2005*.

Harlow, F., and J. Welch. 1965. "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface." *Physics of Fluids* 8, pp. 2182–2189.

Harris, Mark J. 2004. "Fast Fluid Dynamics Simulation on the GPU." In *GPU Gems*, edited by Randima Fernando, pp. 637–665. Addison-Wesley.

Harris, Mark, William Baxter, Thorsten Scheuermann, and Anselmo Lastra. 2003. "Simulation of Cloud Dynamics on Graphics Hardware." In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 92–101.

Kass, Michael, Aaron Lefohn, and John Owens. 2006. "Interactive Depth of Field Using Simulated Diffusion on a GPU." Technical report. Pixar Animation Studios. Available online at http://graphics.pixar.com/DepthOfField/paper.pdf.

Krüger, Jens, and Rüdiger Westermann. 2003. "Linear Algebra Operators for GPU Implementation of Numerical Algorithms." In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), pp. 908–916.

Lefohn, A. E., J. M. Kniss, C. D. Hansen, and R. T. Whitaker. 2004. "A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets." *IEEE Transactions on Visualization and Computer Graphics* 10(2).

Li, Wei, Zhe Fan, Xiaoming Wei, and Arie Kaufman. 2005. "Flow Simulation with Complex Boundaries." In *GPU Gems 2*, edited by Matt Pharr, pp. 747–764. Addison-Wesley.

Liu, Y., X. Liu, and E. Wu. 2004. "Real-Time 3D Fluid Simulation on GPU with Complex Obstacles." *Computer Graphics and Applications*.

Lossasso, F., T. Shinar, A. Selle, and R. Fedkiw. 2006. "Multiple Interacting Liquids." In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)* 25(3).

Müller, Matthias, David Charypar, and Markus Gross. 2003. "Particle-Based Fluid Simulation for Interactive Applications." In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159.

Nguyen, D., R. Fedkiw, and H. W. Jensen. 2002. "Physically Based Modeling and Animation of Fire." In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)* 21(3).

Pharr, Matt, ed. 2005. "Part IV: General-Purpose Computation on GPUs: A Primer." In *GPU Gems 2.* Addison-Wesley.

Sander, P. V., N. Tatarchuk, and J. Mitchell. 2004. "Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering." ATI Technical Report.

Scharsach, H. 2005. "Advanced GPU Raycasting." In *Proceedings of CESCG 2005.*

Selle, A., R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac. 2007. "An Unconditionally Stable MacCormack Method." *Journal of Scientific Computing* (in review). Available online at http://graphics.stanford.edu/~fedkiw/papers/stanford2006-09.pdf.

Sethian, J. A. 1999. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press.

Sigg, Christian, and Markus Hadwiger. 2005. "Fast Third-Order Texture Filtering." In *GPU Gems 2*, edited by Matt Pharr, pp. 307–329. Addison-Wesley.

Stam, Jos. 1999. "Stable Fluids." In *Proceedings of SIGGRAPH 99*, pp. 121–128.

Wu, E., Y. Liu, and X. Liu. 2004. "An Improved Study of Real-Time Fluid Simulation on GPU." In *Computer Animation and Virtual Worlds* 15(3–4), pp. 139–146.

Ziegler, G., A. Trevs, C. Theobalt, and H.-P. Seidel. 2006. "GPU PointList Generation using HistoPyramids." In *Proceedings of Vision Modeling & Visualization 2006.*

Chapter 38

Imaging Earth's Subsurface Using CUDA

Bernard Deschizeaux
CGGVeritas

Jean-Yves Blanc
CGGVeritas

38.1 Introduction

The main goal of earth exploration is to provide the oil and gas industry with knowledge of the earth's subsurface structure to detect where oil can be found and recovered. To do so, large-scale seismic surveys of the earth are performed, and the data recorded undergoes complex iterative processing to extract a geological model of the earth. The data is then interpreted by experts to help decide where to build oil recovery infrastructure.

The state-of-the-art algorithms used in seismic data processing are evolving rapidly, and the need for computing power increases dramatically every year. For this reason, CGGVeritas has always pioneered new high-performance computing (HPC) technologies, and in this work we explore GPUs and NVIDIA's CUDA programming model to accelerate our industrial applications.

The algorithm we selected to test CUDA technology is one of the most resource-intensive of our seismic processing applications, usually requiring around a week of processing time on a latest-generation CPU cluster with 2,000 nodes. To be economically sound at its full capability for our industry, this algorithm must be an order of magnitude faster. At present, only GPUs can provide such a performance breakthrough.

After much analysis and testing, we were able to develop a fully parallel prototype using GPU hardware to speed up part of our processing pipeline by more than a factor of ten. In this chapter, we present the algorithms and methodology used to implement this seismic imaging application on a GPU using CUDA. It should be noted that this work is not an academic benchmark of the CUDA technology—it is a feasibility study for the industrial use of GPU hardware in clusters.

38.2 Seismic Data

A seismic survey is performed by sending compression waves into the ground and recording the reflected waves to determine the subsurface structure of the earth. In the case of a marine survey, like the one shown in Figure 38-1, a ship tows about ten cables equipped with recording systems called hydrophones that are positioned 25 meters apart. Also attached to the ship is an air gun used as the source of the compression waves.

To acquire seismic data, the ship fires the air gun every 50 meters, and the resulting compression waves propagate through the water to the sea floor and beyond into the subsurface of the earth. When a wave encounters a change of velocity or density in the

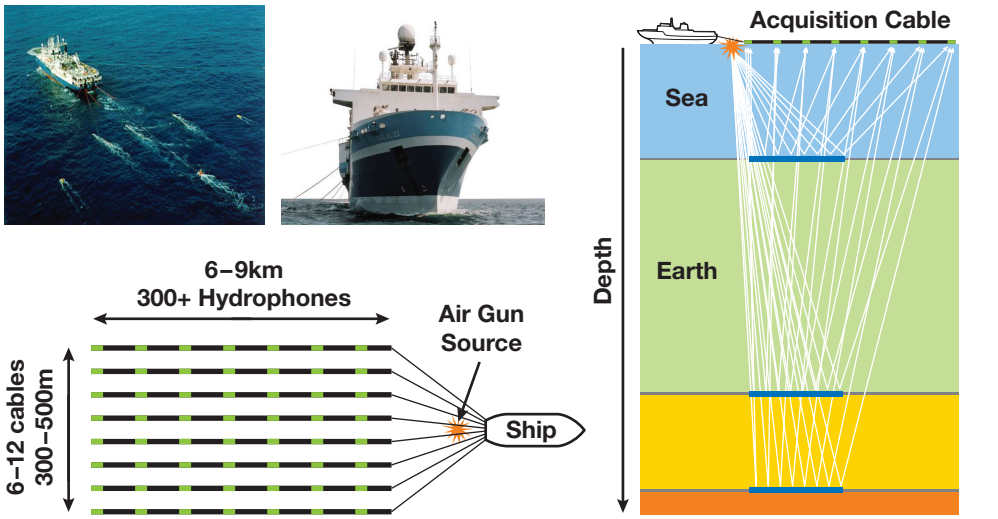


Figure 38-1. Marine Seismic Data Acquisition
A vessel fires an air gun to generate a compression wave that propagates down to the earth and generates reflection waves recorded by hydrophones attached to cables behind the ship.

earth media, it splits in two, one part being reflected back to the surface while the other is refracted, propagating further into the earth (see Figure 38-1). Therefore, each layer of the subsurface produces a reflection of the wave that is recorded by the hydrophones. Because sound waves propagate through water at about 2,500 m/s and through the earth at 3,000 to 5,000 m/s, recording reflection waves for about four seconds after the shot provides information on the earth down to a depth of about 10 to 20 km.

A typical marine survey covers a few hundred square kilometers, which represents a few million shots and several terabytes of recorded data. Processing this amount of data for many studies in parallel is the core business of CGGVeritas processing centers throughout the world. Due to its very low initial signal-to-noise ratio and the large data size, seismic data processing is extremely demanding in terms of processing power. As illustrated by the image in Figure 38-2, CGGVeritas computing facilities consist of PC clusters of several thousand nodes, providing more than 300 teraflops of computing power and petabytes of disk space.

To support increasing survey sizes and processing complexity, our computing power needs to grow by more than a factor of two every year (see the graph in Figure 38-2). Furthermore, heat limitations have forced CPU manufacturers to limit future clock frequencies to around 4 GHz. Increasing the size of clusters in data centers can be realistic for only a short period of time, and this problem enforces the need for new technologies. Therefore,

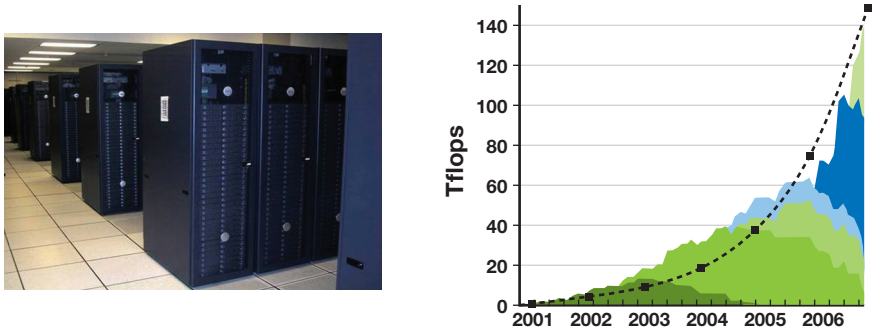


Figure 38-2. Computing Capability Is a Critical Aspect of Our Domain
Our growing trend presented here, color coded according to each different hardware, shows that whenever the technology was available (before 2005), our growth more than doubled every year. The dashed curve gives a reference for exponential growth. As CPU clock frequencies reach a limit, we start to fall below this curve, and only the use of new hardware like GPUs allows us to maintain necessary computing power.

we believe mastering new computing technologies such as general-purpose computing on GPUs is critically important for the future of seismic data processing.

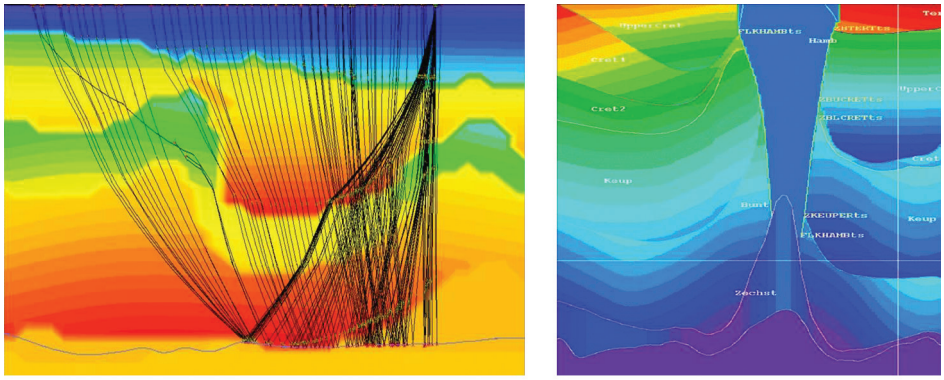
38.3 Seismic Processing

The goal of seismic processing is to convert terabytes of survey data into a 3D volume description of the earth's subsurface structure. A typical data set contains billions of vectors of a few thousand values each, where each vector represents the information recorded by a detector at a specific location and specific wave shot.

The first step in seismic data processing is to correctly position all survey data within a global geographic reference frame. In a marine survey, for instance, we need to take into account the tidal and local streams that shift the acquisition cables from their theoretical straight-line position, and we also need to include any movement of the ship's position. All of the data vectors must be positioned inside a 100 km^2 region at a resolution of 1 meter. Many different positioning systems, both relative and global, are used during data acquisition, and all such position information is included in this processing step.

After correcting the global position for all data elements, the next step is to apply signal processing algorithms to normalize the signal over the entire survey and to increase the signal-to-noise ratio. Here we correct for any variation in hydrophone sensitivity that can lead to nonhomogeneous response between different parts of the acquisition cables. Band-limited deconvolution algorithms are used to verify the known impulse response of the overall acquisition process. Various filtering and artifact removal steps are also performed during this phase. The main goal of this step is to produce data that coherently represents the physics of the wave reflection for a standard, constant source.

The last and the most important and time-consuming step is designed to correct for the effects of changing subsurface media velocity on the wave propagation through the earth. Unlike other echoing systems such as radar, our system has no information about the propagation velocity of the media through which the compression waves travel. Moreover, the media are not homogeneous, causing the waves to travel in curves rather than straight lines, as shown in Figure 38-3a. Therefore, the rather simple task for radar of converting the time of the echo arrival into the distance of the reflection is, in the seismic domain, an extremely complex, inverse problem. To further complicate the



(a) (b)

Figure 38-3. Ray Tracing for a Single Reflector (*Bottom*) Through the Earth, Modeled by a Velocity Field Display in Color
(a) We can clearly see how velocity variations bend the rays even for a rather smooth velocity model. (b) In some cases, the velocity changes are extremely complex and nonhomogeneous, and the wave propagation is extremely difficult to model, especially because we would need to compute billions of rays.

process, more than one reflection occurs after a wave shot, so the recorded signal can in fact be a superposition of many different reflections coming from different places.

Because the velocity field is initially unknown, we generally start by assuming a rather simple velocity model. Then the migration process gives us a better image of the earth's subsurface that allows us to refine the velocity field. This iterative process finally converges toward our best approximation to the exact earth reflectivity model.

At the end of the processing, the 3D volume of data is far cleaner and easier to understand. Some attributes can be extracted to help geologists interpret the results. Typically the impedance of the media is one of those attributes, as well as the wave velocity, the density, and the anisotropy. Figure 38-4 gives an overview of what the data looks like before and after the processing sequence. Also shown is an attribute map representing the wave velocity at a particular depth of the seismic survey. Different rock types have different velocities, so velocity is a good indicator to look for specific rocks such as sand. In the particular case of Figure 38-4c, low velocities (in blue) are characteristic of sand, here from an old riverbed. As a rock, sand is very porous and is typically a good location to prospect for oil.

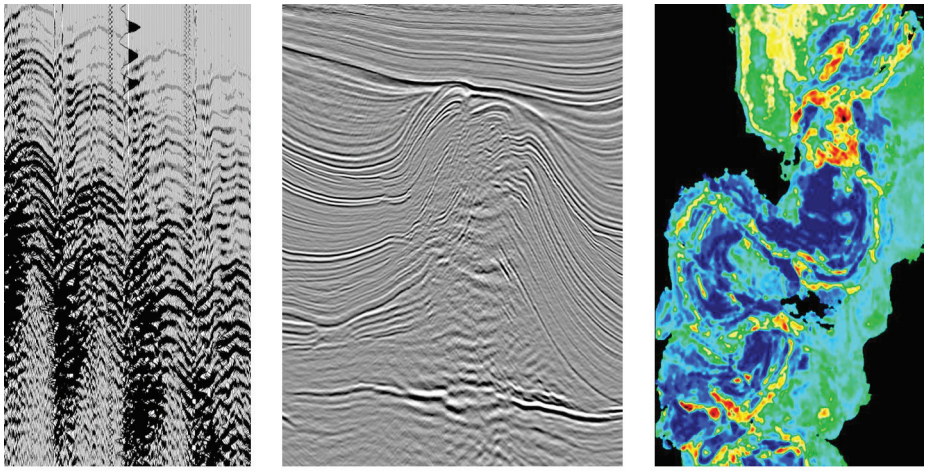


Figure 38-4. A Seismic Processing Example
(a) Raw data recorded during a land survey in Germany showing the poor signal-to-noise ratio and the lack of calibration. (b) A vertical section of about 10 km wide and 5 km deep in the final 3D result shows the layered structure of the earth. (c) This map represents an attribute extracted at a particular depth from a final seismic data set. This attribute is used to distinguish between sand and shale rocks (blue versus green) around a winding shape, which is the remaining channel imprint of a 70-million-year-old river buried under 10 km of earth.

38.3.1 Wave Propagation

For a perfect theoretical seismic data set, the recorded signal $\mathbf{r}_x$ of the wave propagation from a specific source $\mathbf{S}_i$ recorded by a hydrophone $\mathbf{G}_j$ after a reflection of amplitude $\mathbf{R}_x$ at the 3D location $\mathbf{x}(x, y, z)$ can be expressed as follows:

$$\mathbf{r}_x = \mathbf{P}_{xj}^{\mathbf{V}}\left(\mathbf{R}_x \cdot \mathbf{P}_{ix}^{\mathbf{V}}\left(\mathbf{W}_s\right)\right), \tag{1}$$

where $\mathbf{W}_s$ is the source signal, $\mathbf{P}_{ix}$ is the operator that propagates the wave from the source position $\mathbf{i}$ to the reflection position $\mathbf{x}$ through the velocity field $\mathbf{V}$, and $\mathbf{P}_{xj}$ is the operator that propagates the reflected wave from $\mathbf{x}$ to the recorder position $\mathbf{j}$.

To model the complete seismic recording by one receiver, we need to integrate the Equation 1 for all possible reflection positions—that is, integrate on the whole 3D volume of $\mathbf{x}$ values:

$$\mathbf{S}_j = \int_x \left[\mathbf{P}_{xj}^{\mathbf{V}}\left(\mathbf{RC}_{earth}\left(\mathbf{x}\right) \cdot \mathbf{P}_{ix}^{\mathbf{V}}\left(\mathbf{W}_s\right)\right)\right], \tag{2}$$

where $\mathbf{S}_j$ is the seismic recording at position **j** and $\mathbf{RC}_{earth}$ is the reflectivity model of the earth we are looking for. The complexity should be apparent now, because each of the hundreds of millions of data vectors may include information from the whole earth area in a way that depends on the velocity field. Note that in practice the velocity is around a few kilometers per second. Thus if we record wave reflection for a few seconds, only the earth approximately 10 km around the receiver position will contribute to the signal.

It is not realistic to use a brute-force approach to solve this inverse problem, but it can be simplified if we use the property of the propagation operator: $\mathbf{P}_{ij}(\mathbf{P}_{ji}(\mathbf{a})) = \mathbf{I}$. That is, propagation from source to reflection point and back to the source position should give the initial result (that is, there should be no dissipation). From Equation 1 we can see that

$$\mathbf{P}^{\mathbf{V}}_{\mathbf{jx}}(\mathbf{r}_{\mathbf{x}}) = \mathbf{P}^{\mathbf{V}}_{\mathbf{jx}}\left(\mathbf{P}^{\mathbf{V}}_{\mathbf{xj}}\left(\mathbf{R}_{\mathbf{x}}.\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(\mathbf{W}_{\mathbf{s}})\right)\right) = \mathbf{R}_{\mathbf{x}}.\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(\mathbf{W}_{\mathbf{s}}). \tag{3}$$

And if we consider all the possible contributions to a specific record—that is, summing up all contributions for all **x** locations—we can write this:

$$\int_{\mathbf{x}} \mathbf{P}^{\mathbf{V}}_{\mathbf{jx}}(S_j) = \int_{x}\left[RC_{earth}(\mathbf{x}).\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s)\right] = RC_{earth} \otimes \int_{x}\left[\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s)\right]. \tag{4}$$

Hence, the recorded seismic signal $\mathbf{S}_j$, taken as a source and propagated through the earth at all possible **x** locations, is equal to the earth reflectivity model convolved by the initial source shot propagated to any possible reflection position in the earth. It is then clear that if we correlate both sides of this equation by

$$\int_{x}\left[\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s)\right]$$

and sum up information from all receivers for each source, we may extract the earth reflectivity model:

$$RC_{earth} = \sum_{s}\left[\int_{x}\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s) * \sum_{j}\int_{x}\mathbf{P}^{\mathbf{V}}_{\mathbf{jx}}(S_j)\right],$$

where $*$ is the correlation operator, and using

$$\int_{x}\left[\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s)\right] * \int_{x}\left[\mathbf{P}^{\mathbf{V}}_{\mathbf{ix}}(W_s)\right] = 1.$$

Hence, if we propagate the source wave through the earth to all reachable positions **x**, and correlate the result with the recorded data back-propagated to the same **x** location,

we only have to sum up results for all sources and all receivers to obtain the earth reflectivity model. Note that in practice we need to take into account the dispersive effect of the propagation, as well as the fact that the data is band limited. Also, because the velocity field is initially unknown, we need to start with an initial guess (based on expert knowledge of the area) to compute a first reflection model and then refine our velocity field by interpreting the results in terms of the geological structure. (See Yilmaz 2001 and Sherifs 1984 for more information.)

38.3.2 Seismic Migration Using the SRMIP Algorithm

In the case of the CGGVeritas algorithm, called SRMIP, that we want to develop using CUDA, the wave propagation is performed using a finite-difference algorithm applied in the frequency domain.

As presented earlier, the seismic data is composed of a succession of wave shots. Each wave shot is recorded as a 3D volume (x, y, t) where x and y represent the receiver location and t the recording time. This data is transformed into frequency planes by applying a Fast Fourier Transform on the time axis. For each frequency plane, we want to propagate the source wave (called the *downgoing wave*) and the seismic data (called the *upgoing wave*) from the surface (depth = 0) to the maximum depth we want to image. The propagation (also called *downward extrapolation*) is carried out from one depth to the next by applying spatial convolution using finite-length filters.

The SRMIP algorithm relies on a method to take advantage of the circular symmetry of the wave propagator filter: the radial response of the filter is expanded as a polynomial in the Laplacian, which is approximated by the sum of two 1D filters (approximating the second derivative k_x^2 and k_y^2):

$$L = \sum_{n=0}^{n=N_{L_x}} d_x(n)\cos(n\Delta x k_x) + \sum_{n=0}^{n=N_{L_y}} d_y(n)\cos\left(n\Delta y k_y\right)$$

and approximate the exact extrapolation operator:

$$G_0(L) = \exp\left(i\Delta z\left(\frac{\mathbf{w}^2}{\mathbf{v}^2 - L}\right)^{\frac{1}{2}}\right)$$

by a polynomial $G(L)$:

$$G(L) = \sum_{n=0}^{n=N} b_{\mathbf{w}/\mathbf{v}}(n)L^n,$$

where $\mathbf{w}$ is the frequency considered, $\mathbf{v}$ the velocity, and L the Laplacian.

Because we want to extrapolate the wave in an iterative way for all depth values starting from the surface, the choice of the filter parameterization is critical for the stability of the results. To optimize the coefficients of the polynomials, we use the L_∞ norm, because the stability condition is expressed more easily in this norm. In our SRMIP algorithm, we use an expression of the extrapolator using Chebyshev polynomials (see Soubaras 1996 and Hall 1991 for details):

$$G(L) = \sum_{n=0}^{n=N} t_{\mathbf{w}/c}(n) T_n(L),$$

where $T_n(x)$, the Chebyshev polynomial of degree n, is defined by $T_n(x) = \cos(n \arccos x)$ and can be recursively computed using the formula:

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

The degree of the polynomial expansion (that is, the parameter N) is about 15, which means that to propagate the wave from one depth to the next we need to apply a Chebyshev polynomial of Laplacian filter 15 times, recursively.

The pseudocode given in Listing 38-1 shows the implementation of the extrapolator to propagate the wave from one depth to the next. For obvious efficiency reasons, the iterative calculation of the Chebyshev polynomial is computed directly and applied to every point of the input wave grid, saving an operation in the internal loop.

The SRMIP algorithm has a high degree of parallelism. This is because the basic operation is a simple 1D convolution with a constant short filter (approximating the second derivative). The fact that the Chebyshev recursion is not intrinsically parallel is not in this case a problem, because the parallelism is achieved across independent grid elements. Note that for a parallel implementation, some potential improvements that decrease the number of operations at the cost of a more complex data structure—such as making the degree of the polynomials or the length of the second derivative filters vary with the frequency—are not automatically advantageous.

Figure 38-5 gives an example of results obtained by applying the SRMIP algorithm to seismic data. Beyond the general quality improvement, we can see that the results are particularly improved where the earth structure is complex. For instance, the salt body in the top of the earth section has a very high velocity compared to the other surrounding rocks. Therefore, before migration, all data below the salt is not properly focused and appears almost random. After migration, as the propagation within different velocity media has been properly handled, the earth structure below the salt appears.

Listing 38-1. Pseudocode of the Extrapolator

The input `Wave` grid is convolved recursively with two 1D Laplacian filters to produce the propagated `Wave1` grid at the next depth.

```
T(x,y) = Wave(x,y);
TT(x,y)=Laplacian⊗Wave(x,y);
Wave1(x,y) = a_w/v(1,x,y)*T(x,y) + a_w/v(2,x,y)*TT(x,y);
for (n = 2; n < NMAX; n++)
{
  // Compute the Chebyshev polynomial TTT
  // using the two previous stored values TT and T.
  TTT(x,y)=2*Laplacian⊗TT(x,y)-T(x,y);

  // Add the contribution of the iteration to the results.
  Wave1(x,y) += a_w/v(n,x,y)*TTT(x,y);

  // Store Chebyshev results for next iteration.
  T(x,y) = TT(x,y);
  TT(x,y)= TTT(x,y);
};
```

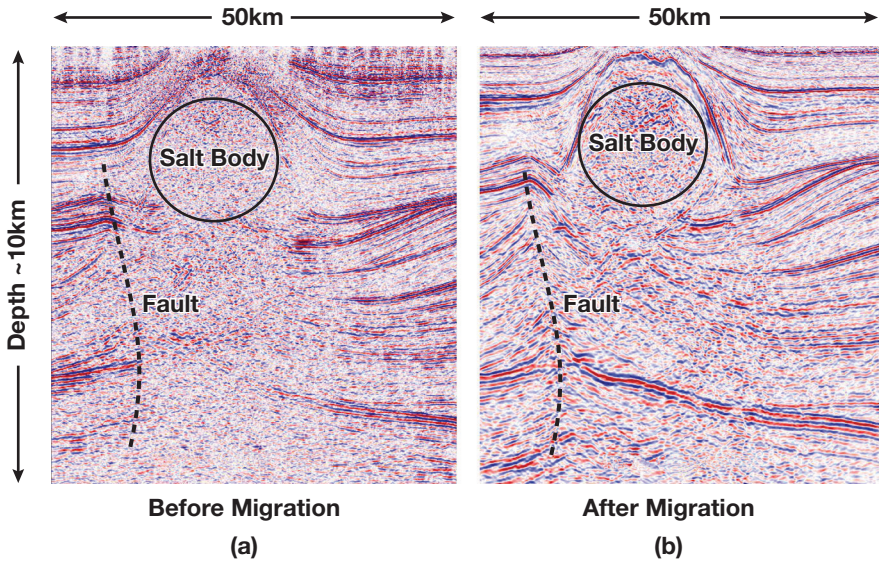


Figure 38-5. The Impact of the Migration Algorithm on a Data Set
(a) The high-velocity salt body blurs the image below. (b) After migration, information below the salt is correctly focused and reveals the earth's structure.

38.4 The GPU Implementation

Selecting algorithms for GPU implementation can be difficult, especially without experience in GPU programming. In our seismic processing sequence, there are several important considerations. For example, the algorithms we port to the GPU are part of an industrial application already running in parallel on a large cluster. Therefore, our goal is an application running on the same kind of cluster but with graphics cards installed in every node. Furthermore, a significant part of the application that deals with all cluster parallelization and efficient data management cannot be changed to accommodate the GPU programming model.

The pseudocode in Listing 38-2 illustrates another consideration. Clearly, the overall benefit of GPU acceleration is limited by the percentage of total execution time attributed to each computational kernel. This code shows the general structure of the SRMIP program that runs independently on every node of the cluster. Each instance of the program (one per processor core) processes a group of seismic shots in sequence and produces a contribution to the final image. Profiling the program with standard parameters shows that 65 percent of the CPU time is consumed in the wave propagation, while all the interpolation routines used 20 percent, and the final correlation and summation use 5 percent. The interpolation step has been added to reduce processing time for the wave propagation. Therefore, it is possible that this step could be removed, depending on how much we accelerate the wave propagation.

Listing 38-2. Pseudocode of the Algorithm Showing the Main Loops and Steps of the Process

```
// uwave = upward wave; dwave = downward wave
// Frequency loop ~ 1000 iterations
for (freq = 0; freq < freq_max; freq++)
{
  Read_frequency_plane(uwave,dwave,nx,ny);

  // Depth loop ~ 1000 iterations
  for (z = 0; z < depth_max; z = z+dz)
  {
    Read_velocity_scalar_field(velocity,nx,ny,z);

    // Propagate uwave and dwave from z to z+dz
    // by applying N time (N~15) Laplacian operator.
    for (i=0; i < N ; i++)
    {
      convolution(uwave,velocity,nx,ny,z,dz);
      convolution(dwave,velocity,nx,ny,z,dz);
    }
```

Listing 38-2 (*continued*). Pseudocode of the Algorithm Showing the Main Loops and Steps of the Process

```
      // Interpolate uwave and dwave between z and z +dz.
      interpolate_wave_over_dz(uwave,velocity,nx,ny,z,z+zd);
      interpolate_wave_over_dz(dwave,velocity,nx,ny,z,z+zd);

      for (zz = z; zz < Z+dz; z++)
      {
        // Interpolate uwave and dwave on output grid.
        Interpolat_xy(uwave,nx,nx,zz,fnx,fny,final_uwave);
        Interpolat_xy(dwave,nx,nx,zz,fnx,fny,final_dwave);

        // Convolve the two waves and sum results.
        sum_udwave(final_uwave,final_dwave,fnx,fny,zz,result);
      }
    }
  }
```

In addition to focusing GPU implementation efforts on the most time-consuming parts of our application, it is equally if not more important to consider the amount of parallelism inherent in our algorithms. Indeed, the CUDA programming model is designed to let users exploit the massive data-parallel processing power of the GPU, so to achieve high performance, we have to choose algorithms with significant data parallelism. In the case of the SRMIP algorithm, the typical grid size we need to process is 400×400 elements, which is determined by the spatial extent of the wave propagation. The data grids correspond to 25 m spacing within a 100 km^2 region, which results in parallelism of roughly 160,000 independent operations. This is more than enough to make efficient use of modern GPUs.

38.4.1 GPU/CPU Communication

A potential problem for GPU-based seismic processing is the cost of GPU/CPU communication. Looking at the general trend of hardware evolution, we predict the GPU will roughly double in performance every year. However, for data transfer between the CPU and GPU (currently using PCIe), the increase in performance is far less impressive. We can expect the PCIe bandwidth to increase by 2× every two or three years at best. Therefore, if we want to design implementations that scale with future GPU performance, we have to avoid potential communication bottlenecks.

By analyzing the data flow of our code and taking into account the large memory available on NVIDIA Quadro FX 5600 hardware (1.5 GB), we were able to develop a communication schema where almost all the relevant data is stored on the GPU. As shown in Listing 38-3, frequency planes are sent one by one to the GPU, which then computes the two waves to be propagated for all depths and interpolates the results in the *x*, *y*, and *z* directions. Only the final result after summing all contribution will have to be sent back to the CPU.

Listing 38-3. Pseudocode Showing the Proposed Communication Scheme

```
// Frequency loop ~ 1000 iterations
for (freq=0; freq < freq_max; freq++)
{
  Read_frequency_plan(uwave,dwave,nx,ny);

  // Send frequency plan (~2 x 1.3 MB).
  Send_freqplan_to_GPU(uwave,dwave,nx,ny);
  // Depth loop ~ 1000 iterations
  for (z=0; z < depth_max; z=z+dz){
    Read_velocity_field(velocity,nx,ny,z);
    // Send velocity field (~0.6 MB).
    Send_Velocity_to_GPU(velocity,nx,ny);
    for (i=0; i < N ; i++)
    {
      convolution(uwave…); //(on the GPU)
      convolution(dwave…); //(on the GPU)
    }
    interpolate_wave_over_dz(uwave…); //(on the GPU)
    interpolate_wave_over_dz(dwave…); //(on the GPU)
    for (zz = z; zz < Z+dz; z++)
    {
      // Interpolate uwave and dwave on output grid.
      Interpolat_xy(uwave…); //(on the GPU)
      Interpolat_xy(dwave…); //(on the GPU)
      // Convolve the two waves and sum results.
      sum_udwave(uwave,dwave…); //(on the GPU)
    }
  }
}
// Get back results (~1.3 GB)
Receive_image_result(result,nx,ny,nz);
```

According to our profiling, the CPU time to compute one depth value is about 30 ms, and the total time of the depth loop is about half a minute. Taking that into account, we can easily compute the throughput needed by our communication scheme and check that we are within PCIe bandwidth limits. Even the velocity transfer (in the inner loop) is around 20 MB/s, which is far below the communication bottleneck even if the GPU implementation is an order of magnitude more efficient than the CPU version.

The 1.5 GB of memory on the NVIDIA Quadro FX 5600 is of great advantage here. Considering that standard cluster nodes have only a few gigabytes of memory to be shared between two to four processor cores, most of the data set handled in memory by one core on the CPU should fit in the GPU memory.

38.4.2 The CUDA Implementation

NVIDIA's CUDA technology provides a flexible programming environment that allows us to address each of the considerations outlined in the last section. After analyzing our core algorithm and the global framework of the GPU, we split our 12 most compute-intensive CPU routines into four separate kernels to be implemented using CUDA. The four kernels more or less correspond to the four routines shown in the pseudocode in Listing 38-3.

All four target algorithms perform local computations on a grid by applying a small operator to every grid element. We divide the computational grid into 2D tiles that map nicely to CUDA's grid of thread blocks. Each kernel loads a tile of grid data from global memory and caches the data in shared memory for further processing. The main advantage of shared memory is its extremely high bandwidth compared to global GPU memory. For three of the kernels, we load the data directly from GPU memory using standard arrays. For the wave propagation algorithm, we use CUDA's texture extensions as a read path to GPU memory. By using texture, we take advantage of hardware caching and automatic boundary handling, which is otherwise difficult and costly to implement in the kernel code. Because the convolution kernel is applied recursively, storing an extra copy of the outputs back into a texture was necessary between iterations.

The GPU code for our algorithms is quite straightforward, because CUDA is a C-based language. However, the G80 architecture has several performance constraints that make optimization somewhat complicated. For example, G80 has 8,000 32-bit registers per multiprocessor, which limits the register count for each kernel. For example, if a kernel executes on 256 threads running in parallel, each thread can use only 32 registers before reaching the limit. In many cases, it is necessary to optimize around this problem in

one of two ways. First, we can simply reduce the kernel complexity (that is, the code size) to decrease register pressure and complete the algorithm using multiple passes. The second, and many times more successful, approach is to adjust the number of threads in a thread block. In this case, the range of useful thread counts is limited not only by the available registers but also by the fact that we need enough threads to hide memory latency (for example, global loads).

Our experience implementing kernels in CUDA is that the most efficient thread configuration partitions threads differently for the load phase and the processing phase. The load phase is usually constrained by the need to access the device memory in a coalesced way, which requires a specific mapping between threads and data elements. During processing, however, we try to organize the workload in such a way that threads do as much processing as possible—at least around 30 operations per byte of data loaded.

38.4.3 The Wave Propagation Kernel

As previously mentioned, our processing time is dominated by the wave propagation operator. Practically, the wave at a given depth is extrapolated to the next depth using the iterative process described in Section 38.3.2 and Listing 38-1. The iteration loop executes on the CPU; the GPU kernel is mainly in charge of the convolution of the wave grid by the Laplacian filter. In addition, at each iteration, the velocity field at each grid position is used to index into a lookup table and scale the input wave by the polynomial coefficients.

Figure 38-6 provides a graphic illustration of how we partition CUDA threads for data loading and convolution with the cross-shaped filter kernel. For loading, warps for a thread block are distributed across a 2D tile region of the computational grid. We use a tile size of 48×32 elements and thread block dimensions of 48×8, so threads with the same y component spread out such that each thread reads four complex frequency coefficients in a vertical column. The data covered by each tile represents a portion of the actual frequency plane as well as a support region (that is, the boundary elements) determined by the cross-filter radius. After storing the tile in shared memory, we synchronize all threads in the block and move to the processing phase. The radius of the convolution filter is four elements, so the output tile is 40×24. Therefore, we redistribute the thread warps so that each thread computes filtered results for three elements. This approach allows us to use all threads in the block for loading and most threads for processing. The less efficient alternative would be to disable more threads before processing, so that each thread outputs four elements.

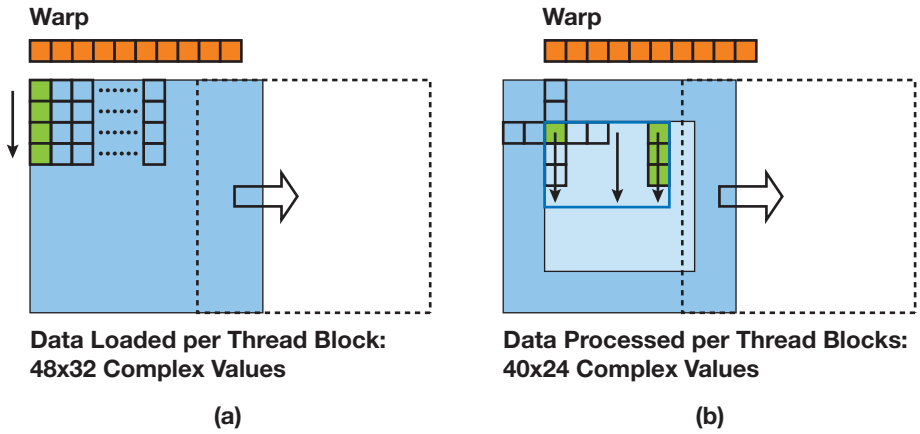


Figure 38-6. Two Thread Organization Strategies for the Convolution Kernel
For data loading, 48×8 thread blocks load 48×32 tiles of complex values. This means each thread loads four values in a column from global memory and writes them to shared memory. For the processing phase, the output tile is 40×24 elements (disregarding the filter support region). In this case, each thread performs a convolution for three output elements in a column.

In addition to giving us an efficient mapping between threads and elements for the load and processing phases, the 48×32 tile size fits nicely within certain resource constraints in the GPU. For example, the G80 architecture has 16 KB of shared memory per multiprocessor. Our tile size (for complex data) takes about 12 KB, so this configuration uses a majority of the shared memory for filtering. A slightly smaller tile size that still uses more than half the available shared memory is less efficient because it prevents multiple thread blocks from running in parallel. Another advantage of this tile size involves coalescing constraints for global memory. In general, it is easier to reason about alignment requirements for fast memory access if the thread block width is a multiple of the SIMD width of the GPU, which for G80 is 16 threads. Finally, it is important to have enough threads in the machine to hide memory latencies, and a 48×8 thread block gives 384 threads, which, in our experience, is plenty of parallelism for G80.

Listing 38-4 shows CUDA C code for the wave propagation kernel used in our SRMIP algorithm. The structure of the code reflects the thread configuration discussed previously. See the comments for a description of the constant terms used in the code. As explained previously, for this kernel we load the input data using CUDA's 2D texture extension. We also read the lookup table through 2D texture, because we need to get efficient, almost random, access to the polynomial coefficients. The cross-shaped filter is stored in CUDA's constant memory.

Listing 38-4. CUDA C Code for Our Convolution-Based Wave Propagation Algorithm

```
__global__ void Convo(float2 *odata1, float2 *odata2,
                      int id, int nx, int ny)
{
  // TW is the logical tile width (40 elements).
  // TH is the logical tile height (24 elements).
  // RW is the tile width including the filter support region.
  // IT is the number of input elements per thread (4).
  // OT is the number of output elements per thread (3).
  // FR is the convolution filter radius (4 elements).

  // Compute local and global thread locations.
  int ltidx = threadIdx.x;
  int ltidy = threadIdx.y * IT;
  int gtidx = blockIdx.x * TW + ltidx - FR;
  int gtidy = blockIdx.y * TH + ltidy - FR;
  int tltid = ltidy * RW + ltidx;

  float2 term;
  int i;
  // Each thread reads 4 input values from global memory.
  // The loop is for clarity and should be unrolled for efficiency.
  for (i = 0; i < 4; i++) {
    term = texfetch(itexref, gtidx, gtidy + i);
    smem[tltid     ] = term.x;
    smem[tltid + IO] = term.y;
    tltid += RW;
  }

  __syncthreads();

  // Each thread compute results for 3 output values.
  if (ltidx < TW) {
    int rtlt = (threadIdx.y * OT + FR) * RW + (ltidx + FR);
    int itlt = rtlt + IO;
    int gthx = blockIdx.x * TW + ltidx;
    int gthy = blockIdx.y * TH + threadIdx.y * OT;
    int rind = gthy * nx + gthx;
    int index;
    float vel, floorvel, residus;
    float2 term0, term1, temp, temp2;
```

Listing 38-4 (*continued*). CUDA C Code for Our Convolution-Based Wave Propagation Algorithm

```
    // Compute one element for 3 consecutive lines.
    if (gthx < nx) {
      // The loop is for clarity and should be unrolled for efficiency.
      for (i = 0; i < 3; i++) {
        if (gthy < ny) {
          temp   = texfetch(otexref, gthx , gthy);
          temp.x = (smem[rtlt-   4] + smem[rtlt+   4])*coeff_X[4] +
                   (smem[rtlt-   3] + smem[rtlt+   3])*coeff_X[3] +
                   (smem[rtlt-   2] + smem[rtlt+   2])*coeff_X[2] +
                   (smem[rtlt-   1] + smem[rtlt+   1])*coeff_X[1] +
                   (smem[rtlt-4*RW] + smem[rtlt+4*RW])*coeff_Y[4] +
                   (smem[rtlt-3*RW] + smem[rtlt+3*RW])*coeff_Y[3] +
                   (smem[rtlt-2*RW] + smem[rtlt+2*RW])*coeff_Y[2] +
                   (smem[rtlt-  RW] + smem[rtlt+  RW])*coeff_Y[1] +
                    smem[rtlt     ]*(coeff_X[0]+coeff_Y[0]) - temp.x;
          temp.y = (smem[itlt-   4] + smem[itlt+   4])*coeff_X[4] +
                   (smem[itlt-   3] + smem[itlt+   3])*coeff_X[3] +
                   (smem[itlt-   2] + smem[itlt+   2])*coeff_X[2] +
                   (smem[itlt-   1] + smem[itlt+   1])*coeff_X[1] +
                   (smem[itlt-4*RW] + smem[itlt+4*RW])*coeff_Y[4] +
                   (smem[itlt-3*RW] + smem[itlt+3*RW])*coeff_Y[3] +
                   (smem[itlt-2*RW] + smem[itlt+2*RW])*coeff_Y[2] +
                   (smem[itlt-  RW] + smem[itlt+  RW])*coeff_Y[1] +
                    smem[itlt     ]*(coeff_X[0]+coeff_Y[0]) - temp.y;
          vel      = texfetch(vtexref, gthx, gthy);
          floorvel = floorf(vel);
          index    = floorvel;
          term0    = texfetch(ltexref, index, id);
          term1    = texfetch(ltexref, index + 1, id);
          residus  = vel - floorvel;
          term0.x  = term0.x + residus*(term1.x - term0.x);
          term0.y  = term0.y + residus*(term1.y - term0.y);
          temp2    = texfetch(olktexref, gthx, gthy);
          temp2.x += term0.x*temp.x - term0.y*temp.y;
          temp2.y += term0.x*temp.y + term0.y*temp.x;
          odata1[rind] = temp2;
          odata2[rind] = temp;
        }
        rtlt += RW; itlt += RW;
        gthy++; rind += nx;
      }
    }
  }
}
```

38.5 Performance

Because of its strategic importance, our wave migration system uses highly optimized CPU code, especially on Intel platforms. Therefore, our GPU-to-CPU performance comparison uses a solid reference on the CPU. However, it should be noted that, because CPU performance for this algorithm does not scale linearly with the number of cores (mainly because of memory access bottlenecks), we compare our GPU kernel to a latest-generation CPU with only one core enabled.

Using a synthetic data set with typical input parameters, our CUDA kernels achieve performance ranging from 8× to 15× over the optimized CPU code. In addition, the kernels perform equally well on real seismic data sets, where the CUDA code is fully integrated into our industrial processing sequence. However, it is important to note that we have not tested the GPU implementation with the full range of input parameters used with the CPU version. The main reason is that the GPU code is designed for a specific problem size and thread configuration, while the CPU can more easily adapt to different kinds of user parameters and data characteristics. Even still, the GPU performance is a significant improvement by any measure.

Including all the kernels in the industrial parallel application is an ongoing process, and many issues still remain to be solved. The algorithm is so time-consuming that even with a speedup of 15×, a few graphics cards will not meet our processing needs. A cluster solution is mandatory, and on the hardware side, the question of how to design a cluster including GPUs is still open. What speedup the overall application will finally achieve and for what hardware price is our main strategic concern for the future.

38.6 Conclusion

With NVIDIA's CUDA technology, we now have access to a powerful data-parallel programming model and language for exploring scientific computing on the GPU. Once mastered, the flexibility of CUDA can be a real advantage when considering the huge variability of algorithm behavior and data size within the scientific domain. Most important, the CUDA implementation of our most expensive seismic algorithm is more than an order of magnitude faster than its CPU version.

In the long term, CPUs are expected to continue to follow Moore's Law due to the rise of multicore architectures, while GPUs should be able to roughly double in floating-point performance twice a year. Another attractive aspect of GPUs is their fast memory, which outperforms the regular DDR or FBDIMM memory typically used by CPUs.

This proved to be very important for all of our algorithms, because they are already memory limited on the normal cluster solution. The main drawback with GPUs is the transfer speed through PCIe, and bus performance is not expected to increase as rapidly as GPU performance.

There are several factors to consider before building a GPU-based seismic processing cluster. First, it is simply not practical to deploy a large-scale cluster built with racked workstations, because it is neither dense enough nor cost-effective. At this point, two paths can be explored: (1) Classical 1U servers with PCIe slots and a companion external package (such as NVIDIA's Quadro Plex) containing the GPUs or (2) a form factor that includes one or more GPUs on the motherboard. Second, because GPUs need CPUs for control, it's important to choose CPUs for each node that are powerful enough to manage the GPU without becoming a bottleneck. Also, there is the issue of whether PCIe bandwidth is enough to drive one or more GPUs per cluster node. Finally, given the scale and processing time of our algorithms, fault-tolerant hardware is critical in order to recover from failures and avoid wasting days of processing time. Future generations of GPUs will need this feature to be viable for inclusion in our processing centers.

Although there are many open questions about how graphics processors can be used in a large-scale cluster, our work in this chapter shows that GPUs definitively have the potential to disrupt the current seismic processing ecosystem.

38.7 References

Hall, D. 1991. "3-D Depth Migration via McClellan Transforms." *Geophysics* 36, pp. 99–114.

Sherifs, R. E., ed. 1984. *Encyclopedic Dictionary of Exploration Geophysics*. Society of Exploration Geophysicists.

Soubaras, R. 1996. "Explicit 3-D Migration Using Equiripple Polynomial Expansion and Laplacian Synthesis." *Geophysics* 61, pp. 1386–1393.

Yilmaz, O. 2001. *Seismic Data Analysis: Processing, Inversion, and Interpretation of Seismic Data (Investigations in Geophysics, No. 10)*. Society of Exploration Geophysicists.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z