Contents

28. General Purpose Computing on Graphics Processing Units

GPGPU stands for **G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits, also known as GPU Computing. Graphics Processing Units (***GPU***) are highly parallel, multithreaded, manycore processors capable of very high computation and data throughput. Once specially designed for computer graphics and programmable only through graphics APIs, today's GPUs can be considered as general-purpose parallel processors with the support for accessible programming interfaces and industry-standard languages such as C.

Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations. This difference comes from the high floating point performance and peak memory bandwidth of GPUs. This is because the GPU is specialized for compute-intensive, highly parallel computation—exactly what graphics rendering is about—and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. From the developer's point of view this means that hardware latencies are not hidden, they must be managed explicitly, and writing an efficient GPU program is not possible without the knowledge of the architecture.

Another reason of discussing GPGPU computing as a specific field of computer science is that although a GPU can be regarded as a parallel system, its architecture is not a clean implementation of parallel computing models (see Chapter 15 of this book titled *Parallel Computations*). Instead, it has the features of many different models, like pipelines, vector or array processors, ***Single-Instruction Multiple-Data*** (***SIMD***) machines, stream-processors, multi-processors connected via shared memory, hard-wired algorithms, etc. So, when we develop solutions for this special architecture, the ideas applicable for many different architectures should be combined in creative ways.

GPUs are available as graphics cards, which must be mounted into computer systems, and a runtime software package must be available to drive the computations. A graphics card has programmable processing units, various types of memory and cache, and fixed-function units for special graphics tasks. The hardware operation must be controlled by a program running on the host computer's CPU through ***Application Programming Interfaces*** (***API***)**.** This includes uploading programs to GPU units and feeding them with data. Programs might be written and compiled

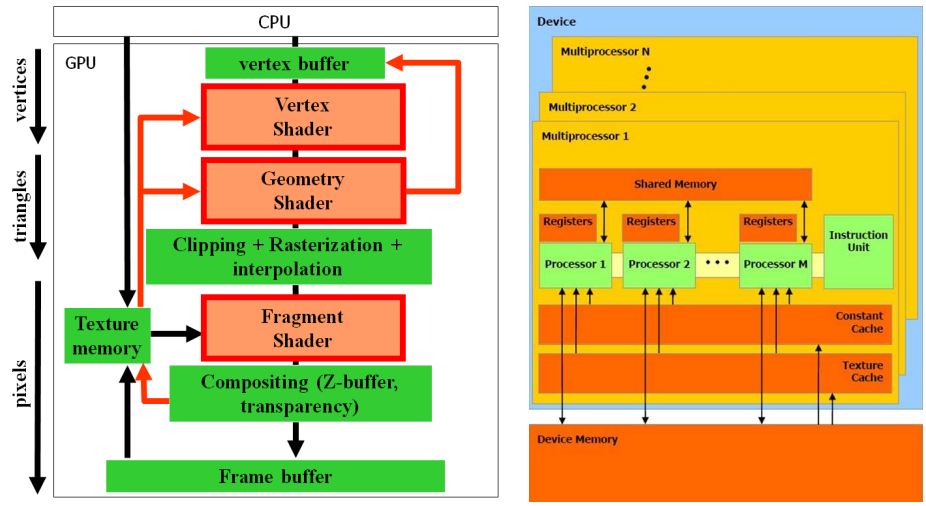

Figure 28.1 GPU programming models for shader APIs and for CUDA. We depict here a Shader Model 4 compatible GPU. The programmable stages of the shader API model are red, the fixed-function stages are green.

from various programming languages, some originally designed for graphics (like ***Cg*** [13] or ***HLSL*** [11]) and some born by the extension of generic programming languages (like CUDA C). The programming environment also defines a ***programming model*** or ***virtual parallel architecture*** that reflects how programmable and fixed-function units are interconnected. Interestingly, different programming models present significantly different virtual parallel architectures (Figure 28.1). Graphics APIs provide us with the view that the GPU is a pipeline or a stream-processor since this is natural for most of the graphics applications. ***CUDA*** [14] or ***OpenCL*** [8], on the other hand, gives the illusion that the GPU is a collection of multiprocessors where every multiprocessor is a wide SIMD processor composed of scalar units, capable of executing the same operation on different data. The number of multiprocessors in a single GPU can range nowadays up to a few hundreds and a single multiprocessor typically contains 8 or 16 scalar units sharing the instruction decoder.

The total number of scalar processors is the product of the number of multiprocessors and the number of SIMD scalar processors per multiprocessor, which can be well over a thousand. This huge number of processors can execute the same program on different data. A single execution of the program is called the ***thread.*** A multiprocessor executes a ***thread block.*** All processors have some fast local memory, which is only accessible to threads executed on the same processor, i.e. to a thread block. There is also global device memory to which data can be uploaded or downloaded from by the host program. This memory can be accessed from mul-

tiprocessors through different caching and synchronization strategies. Compared to the CPU, this means less transistors for caching, less cache performance in general, but more control for the programmer to make use of the memory architecture in an efficient way.

The above architecture favours the parallel execution of short, coherent computations on compact pieces of data. Thus, the main challenge of porting algorithms to the GPU is that of parallelization and decomposition to independent computational steps. GPU programs, which perform such a step when executed by the processing units, are often called ***kernels*** or ***shaders,*** the former alludes to the parallel data processing aspect and the latter is a legacy of the fundamental graphics task: the simulation of light reflection at object surfaces, better known as shading.

GPU programming languages and control APIs have grown pretty similar to each other in both capabilities and syntax, but they can still be divided into graphics and GPGPU solutions. The two approaches can be associated with two different programmer attitudes. While GPGPU frameworks try to add some constructs to programming languages to prepare regular code for parallel execution, graphics APIs extend previously very limited parallel shader programs into flexible computational tools. This second mindset may seem obsolete or only relevant in specific graphics-related scenarios, but in essence it is not about graphics at all: it is about the implicit knowledge of how parallel algorithms work, inherent to the incremental image synthesis pipeline. Therefore, we first discuss this pipeline and how the GPU device is seen by a graphics programmer. This will not only make the purpose and operation of device components clear, but also provides a valid and tried approach to general purpose GPU programming, and what GPU programs should ideally look like. Then we introduce the GPGPU approach, which abandons most of the graphics terminology and neglects task-specific hardware elements in favour of a higher abstraction level.

28.1. The graphics pipeline model

The graphics pipeline model provides an abstraction over the GPU hardware where we view it as a device which performs ***incremental image synthesis*** [18] (see Chapter 22 of this book, titled *Computer Graphics* of this book). Incremental image synthesis aims to render a virtual world defined by a numerical model by transforming it into linear primitives (points, lines, triangles), and rasterizing these primitives to pixels of a discrete image. The process is composed of several algorithmic steps, which are grouped in pipeline stages. Some of these stages are realized by dedicated hardware components while others are implemented through programs run by GPUs. Without going into details, let us recap the image synthesis process (Figure 28.2):

- The virtual world is a collection of model instances. The models are approximated using triangle meshes. This is called .
- In order to perform shading, the objects have to be transformed into the coordinate system where the camera and lights are specified. This is either the ***world space*** or the ***camera space.***

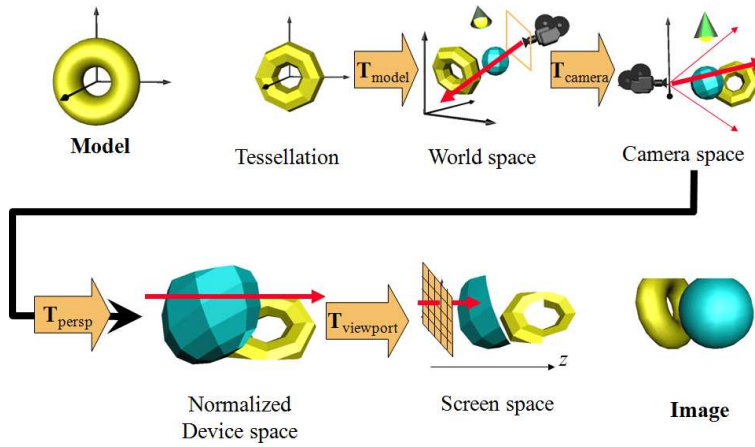

Figure 28.2 Incremental image synthesis process.

- Triangle vertices are projected on-screen according to the camera settings. Where a vertex should appear on the screen is found by applying the ***camera transformation,*** the ***perspective transformation,*** and finally the ***viewport transformation.*** In camera space the camera is in the origin and looks at the $-z$ direction. Rays originating at the camera focus, called the ***eye*** position, and passing through points on the window that represent the pixels of our display form a perspective bundle. The role of perspective transformation is to convert this perspective bundle into a parallel bundle of rays, thus to replace perspective projection by a parallel projection. After perspective transformation, the vertices are in ***normalized device space*** where the visible volume is an axis aligned cube defined by inequalities $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $-1 \leq z \leq 1$. Parts of the geometric primitives that are outside of this volume are removed by ***clipping.*** Normalized device space is further transformed to ***screen space,*** where the target image resolution and position are taken into account. Points of normalized device space coordinates $x = -1, y = -1$ are mapped to the lower left corner of the viewport rectangle on the screen. Points of $x = 1, y = 1$ are projected to the upper right corner. Meanwhile, the z range of $-1 \leq z \leq 1$ is converted to $[0, 1]$.
- In screen space every projected triangle is rasterized to a set of pixels. When an internal pixel is filled, its properties, including the z coordinate, also called the ***depth value,*** and shading data are computed via incremental linear interpolation from the vertex data. For every pixel, a shading color is computed from the interpolated data. The shading color of a pixel inside the projection of the triangle might be the color interpolated from the vertex colors. Alternatively, we can map images called ***textures*** onto the meshes. Texture images are 2D arrays of color records. An element of the texture image is called the ***texel.*** How the texture should be mapped onto triangle surfaces is specified by texture coordinates assigned to every vertex.

- Pixel colors are finally written to the ***frame buffer*** that is displayed on the computer screen. Besides the frame buffer, we maintain a ***depth buffer*** (also called ***z-buffer*** or ***depth stencil texture***), containing screen space depth, which is the z coordinate of the point whose color value is in the frame buffer. Whenever a triangle is rasterized to a pixel, the color and the depth are overwritten only if the new depth value is less than the depth stored in the depth buffer, meaning the new triangle fragment is closer to the viewer. As a result, we get a rendering of triangles correctly occluding each other in 3D. This process is commonly called the ***depth buffer algorithm.*** The depth buffer algorithm is also an example of a more general operation, which computes the pixel data as some function of the new data and the data already stored at the same location. This general operation is called ***merging.***

28.1.1. GPU as the implementation of incremental image synthesis

The GPU architecture as presented by the graphics API is the direct implementation of the image synthesis pipeline (left part of Figure 28.1). This pipeline is configured by the CPU via graphics API calls, and its operation is initiated by the ***draw call.*** A sequence of draw calls during which the configuration of the pipeline does not change (but the inputs do) is called a ***pass.*** A single draw call operates on a sequence of vertices, the attributes of which are stored in a ***vertex buffer.***

Vertices are expected to be specified in modeling space with homogeneous coordinates. A point of ***Cartesian coordinates*** (x, y, z) can be defined by the quadruple of ***homogeneous coordinates*** $[xw, yw, zw, w]$ using an arbitrary, non-zero scalar w (for more details see Chapter 21 *Computer Graphics* of this book). This representation owns its name to the fact that if the elements of the quadruple are multiplied by the same scalar, then the represented point will not change. From homogeneous quadruple $[X, Y, Z, w]$ the Cartesian coordinates of the same point can be obtained by ***homogeneous division,*** that is as $(X/w, Y/w, Z/w)$. Homogeneous coordinates have several advantages over Cartesian coordinates. When homogeneous coordinates are used, even parallel lines have an intersection (an ***ideal point,***) thus the singularity of the Euclidean geometry caused by parallel lines is eliminated. Homogeneous linear transformations include ***perspective projection*** as well, which has an important role in rendering, but cannot be expressed as a linear function of Cartesian coordinates. Most importantly, the widest class of transformations that preserve lines and planes are those which modify homogeneous coordinates linearly.

Having set the vertex buffer, vertices defined by their coordinates and attributes like texture coordinates or color begin their journey down the graphics pipeline, visiting processing stages implemented by programmable shader processors or fixed-function hardware elements. We consider these stages one-by-one.

Tessellation

If the vertices do not directly define the final triangle mesh, but they are control points of a parametric surface or define just a coarse version of the mesh, the first step is the development of the final mesh, which is called ***tessellation.*** As the programmability of this stage is limited and its GPGPU potential is small, we do

not discuss this stage further but assume that the vertex buffer contains the fine mesh that needs no tessellation.

Vertex processing

Objects must be transformed to normalized device space for clipping, which is typically executed by a homogeneous linear transformation. Additionally, GPUs may also take the responsibility of illumination computation at the vertices of the triangle mesh. These operations are executed in the vertex shader. From a more general point of view, the vertex shader gets a single vertex at a time, modifies its attributes, including position, color, and texture coordinates, and outputs the modified vertex. Vertices are processed independently and in parallel.

The geometry shader

The geometry shader stage receives vertex records along with primitive information. It may just pass them on as in the fixed-function pipeline, or spawn new vertices. Optionally, these may be written to an output buffer, which can be used as an input vertex buffer in a consecutive pass. A typical application of the geometry shader is procedural modeling, when a complex model is built from a single point or a triangle [10].

While vertex shaders have evolved from small, specialized units to general stream processors, they have kept the one record of output for every record of input scheme. The geometry shader, on the other hand, works on vertex shader output records (processed vertices), and outputs a varying (but limited) number of similar records.

Clipping

The hardware keeps only those parts of the primitives that are inside an axis aligned cube of corners $(-1,-1,-1)$ and $(1,1,1)$ in normalized device space. In homogeneous coordinates, a point should meet the following requirements to be inside:

$$-w \le x \le w,\ -w \le y \le w,\ -w \le z \le w\,.$$

This formulation complies to the OpenGL [12] convention. It is valid e.g. in the Cg language when compiling for an OpenGL vertex shader profile. The last pair of inequalities can also be defined as $0 \le z \le w$, as Direct3D assumes. This is the case for Cg Direct3D profiles and in the HLSL standard. The difference is hidden by compilers which map vertex shader output to what is expected by the clipping hardware.

Clipping is executed by a fixed-function hardware of the GPU, so its operation can neither be programmed nor modified. However, if we wish our primitives to continue their path in further stages of the pipeline, the conditions of the clipping must be satisfied. In GPGPU, the clipping hardware is considered as a ***stream filter.*** If it turns out that a data element processed by vertex and geometry shader programs needs to be discarded, vertices should be set to move the primitive out of the clipping volume. Then the clipping hardware will delete this element from the pipeline.

After clipping the pipeline executes , that is, it converts homogeneous coordinates to Cartesian ones by dividing the first three homogeneous coordinates by the fourth

(w). The points are then transformed to where the first two Cartesian coordinates select the pixel in which this point is visible.

Rasterization with linear interpolation

The heart of the pipeline is the non-programmable rasterization stage. This is capable of converting linear primitives (triangles, line segments, points) into discrete ***fragments*** corresponding to digital image pixels. More simply put, it draws triangles if the screen coordinates of the vertices are given. Pipeline stages before the rasterizer have to compute these vertex coordinates, stages after it have to process the fragments to find pixel colors.

Even though the base functionality of all stages can be motivated by rasterization, GPGPU applications do not necessarily make use of drawing triangles. Still, the rasterizer can be seen to work as a stream expander, launching an array of fragment computations for all primitive computations, only the triangles have to be set up cleverly.

Rasterization works in ***screen space*** where the x, y coordinates of the vertices are equal to those integer pixel coordinates where the vertices are projected. The vertices may have additional properties, such as a z coordinate in screen space, texture coordinates and color values. When a triangle is rasterized, all those pixels are identified which fall into the interior of the projection of the triangle. The properties of the individual pixels are obtained from the vertex properties using linear interpolation.

Fragment shading

The fragment properties interpolated from vertex properties are used to find the fragment color and possibly a modified depth value. The classical operation for this includes fetching the texture memory addressed by the interpolated texture coordinates and modulating the result with the interpolated color.

Generally, fragment shader programs get the interpolated properties of the fragment and output the color and optionally modify the depth of the fragment. Like the vertex shader, the fragment shader is also one-record-in, one-record-out type processor. The fragment shader is associated with the target pixel, so it cannot write its output anywhere else.

Merging

When final fragment colors are computed, they may not directly be written to the image memory, but the ***output merger*** stage is responsible for the composition. First, the depth test against the depth buffer is performed. Note that if the fragment shader does not modify the z value, depth testing might be moved before the execution of the fragment shader. This ***early z-culling*** might improve performance by not processing irrelevant fragments.

Finally, the output merger blends the new fragment color with the existing pixel color, and outputs the result. This feature could implement ***blending*** needed for ***transparent surface rendering*** (Figure 28.3).

In GPGPU, blending is mainly useful if we need to find the sum, minimum or maximum of results from consecutive computations without a need of reconfiguring the pipeline between them.

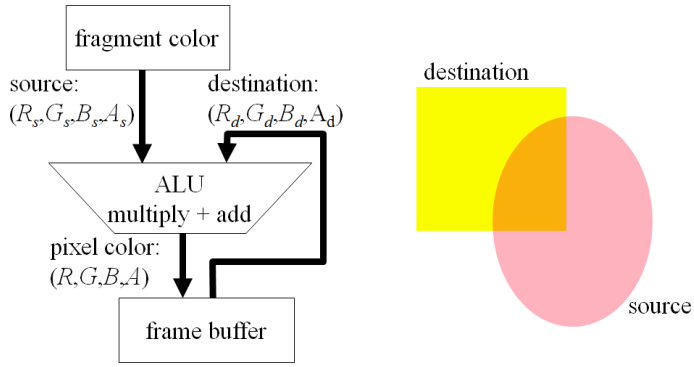

Figure 28.3 Blending unit that computes the new pixel color of the frame buffer as a function of its old color (destination) and the new fragment color (source).

28.2. GPGPU with the graphics pipeline model

In general purpose programming, we are used to concepts like input data, temporary data, output data, and functions that convert input data to temporary and finally to output data according to their parameters. If we wish to use the GPU as presented by a graphics API, our programming concepts should be mapped onto the concepts of incremental image synthesis, including geometric primitives, vertex/primitive/fragment processing, rasterization, texturing, merging, and final image. There are many different possibilities to establish this correspondence, and their comparative advantages also depend on the actual algorithm. Here we consider a few general approaches that have proven to be successful in high performance computing applications. First, we discuss how our general purpose programming concepts can be related to GPU features.

28.2.1. Output

GPUs render images, i.e. two-dimensional arrays of pixels. The ***render target*** can be the frame buffer that is displayed or an output texture (in the latter case, the pixel is often referred to as a texel). In GPGPU the output is usually a texture since texels can be stored in floating point format unlike the final frame buffer values that are unsigned bytes. Furthermore, textures can be used later on as inputs of subsequent computation passes, i.e. the two-dimensional output texture can be interpreted as one or two-dimensional input texture in the next rendering pass, or as a single layer of a three-dimensional texture. In older GPUs, a pixel was capable of storing at most five floating point values since a color is typically identified by red, green, blue, and opacity values, and hidden surface elimination needed a single distance value, which is the z coordinate of the point in screen coordinates. Later, with the emergence of ***multiple render targets,*** a pixel could be associated with several, e.g. four textures, which means that the maximum size of an output record could grow to 17

floats. In current, most advanced Shader Model 5.0 GPUs even this limitation has been lifted, so a single pixel can store a list of varying number of values.

Which pixel is targeted by the rendering process is determined by the geometric elements. Each primitive is transformed to screen space and its projection is rasterized which means that those pixels are targeted that are inside the projection. If more than one element is sent down the pipeline, their projections may overlap, so the pixel value is calculated multiple times. The merging unit combines these partial results, it may keep only one, e.g. the fragment having minimal screen space z coordinate if depth testing is enabled, or it may add up partial results using blending.

An important property of the render target is that it can be read directly by none of the shader processors, and only the fragment shader processor can indirectly write into it via the possible merging operation. Different fragment shaders are assigned to different parts of the render target, so no synchronization problem may occur.

28.2.2. Input

In image synthesis the inputs are the geometry stream and the textures used to color the geometry. As a triangle mesh geometry has usually no direct meaning in a GPGPU application, we use the geometry stream only as a control mechanism to distribute the computational load among the shader processors. The real GPGPU input will be the data stored in textures. The texture is a one-, two- or three-dimensional array of color data elements, which can store one, two, three or four scalars. In the most general case, the color has red, green, blue and opacity channels. These color values can be stored in different formats including, for example, unsigned bytes or 32 bit floats. From the point of view of GPGPU, 32 bit floats are the most appropriate.

A one-dimensional float texture is similar to the linear CPU memory where the usual data structures like arrays, lists, trees etc. can be encoded. However, the equivalence of the CPU memory and the GPU texture fails in two important aspects. In one, the texture is poorer, in the other, it is better than the linear CPU memory.

An apparent limitation is that a texture is parallel read-only for all programmable shaders with the exception of the render target that cannot be read by the shaders and is accessible only for the merger unit. Read-modify-write cycles, which are common in the CPU memory, are not available in shader programs. GPU designers had a good reason not to allow read-modify-write cycles and to classify textures as parallel read-only and exclusive write-only. In this way, the writes do not have to be cached and during reads caches get never invalidated.

On the other hand, the texture memory has much more addressing modes than a linear memory, and more importantly, they are also equipped with built-in ***texture filters.*** With the filters, a texture is not only an array of elements, but also a ***finite element*** representation of a one-, two-, or three-dimensional spatial function (refer to Section 28.7 to learn more of the relation between finite element representations and textures).

For one-dimensional textures, we can use linear filtering, which means that if the texture coordinate u points to a location in between two texels of coordinates U and $U + 1$, then the hardware automatically computes a linear interpolation of the

two texel values. Let these texels be $T(U)$ and $T(U+1)$. The filtered value returned for u is then

$$T(u) = (1-u^*)T(U) + u^*T(U+1), \quad \text{where} \quad u^* = u - U\,.$$

Two-dimensional textures are filtered with ***bi-linear filtering*** taking the four texels closest to the interpolated texture coordinate pair (u, v). Let these be $T(U, V)$, $T(U+1, V)$, $T(U+1, V+1)$, and $T(U, V+1)$. The filtered value returned for (u, v) is then

$$T(U,V)u^*v^* + T(U+1,V)(1-u^*)v^* + T(U+1,V+1)(1-u^*)(1-v^*)$$
$$+T(U,V+1)u^*(1-v^*),$$

where $u^* = u - U$ and $v^* = v - V$.

For three-dimensional textures, ***tri-linear filtering*** is implemented.

28.2.3. Functions and parameters

As the primitives flow through the pipeline, shader processors and fixed-function elements process them, determining the final values in each pixel. The programs of shader processors are not changed in a single rendering pass, so we can say that each pixel is computed by the very same program. The difference of pixel colors is due to data dependencies. So, in conclusion a GPU can be regarded as a hardware that computes an array of records.

In the GPU, primitives are processed by a series of processors that are either programmable or execute fixed algorithms while output pixels are produced. It means that GPUs can also be seen as ***stream processors.*** Vertices defining primitives enter a single virtual stream and are first processed by the vertex shader. With stream processing terminology, the vertex shader is a ***mapping*** since it applies a function to the vertex data and always outputs one modified vertex for each input vertex. So, the data frequency is the same at the output as it was at the input. The geometry shader may change the topology and inputting a single primitive, it may output different primitives having different number of vertices. The data frequency may decrease, when the stream operation is called ***reduction,*** or may increase, when it is called ***expansion.*** The clipping unit may keep or remove primitives, or may even change them if they are partially inside of the clipping volume. If we ignore partially kept primitives, the clipping can be considered as a ***.*** By setting the coordinates of the vertices in the vertex shader to be outside of the clipping volume, we can filter this primitive out of the further processing steps. Rasterization converts a primitive to possibly many fragments, so it is an expansion. The fragment shader is also a mapping similarly to the vertex shader. Finally, merging may act as a selection, for example, based on the z coordinate or even as an ***accumulation*** if blending is turned on.

Shader processors get their stream data via dedicated registers, which are filled by the shader of the preceding step. These are called ***varying input.*** On the other hand, parameters can also be passed from the CPU. These parameters are called ***uniform input*** since they are identical for all elements of the stream and cannot be changed in a pass.

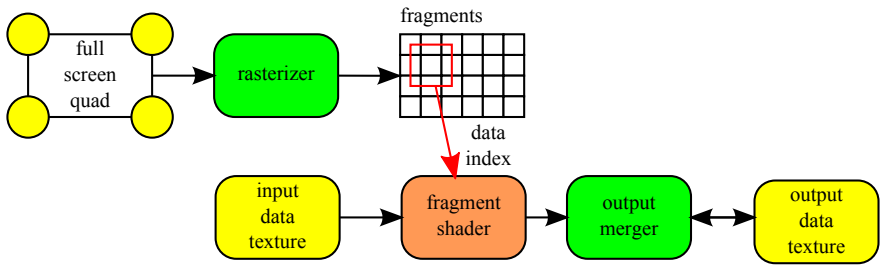

Figure 28.4 GPU as a vector processor.

28.3. GPU as a vector processor

If the computation of the elements is done independently and without sharing temporary results, the parallel machines are called ***vector processors*** or ***array processors.*** As in the GPU hardware the fragment shader is associated with the elements of the output data, we use the fragment shader to evaluate output elements. Of course, the evaluation in a given processor must also be aware which element is being computed, which is the fundamental source of data dependency (it would not make sense to compute the very same data many times on a parallel machine). In the fragment shader, the index of the data element is in fact the pair of the pixel coordinates. This is available in screen space as a pair of two integers specifying the row and the column where the pixel is located.

In the simplest, but practically the most important case, we wish to have a result in all pixels in a single rendering pass. So we have to select a geometric primitive that is mapped to all pixels in screen space and a single pixel is mapped only once. Such a geometric primitive is the virtual display itself, thus we should render a rectangle or a quadrilateral that represents the window of our virtual camera. In screen space, this is the viewport rectangle, in clipping space, this is a square on the x, y plane and having corners in homogeneous coordinates $(-1,-1,0,1)$, $(1,-1,0,1)$, $(1,1,0,1)$, $(-1,1,0,1)$. This rectangle is also called the ***full screen quad*** and is processed by the hardware as two triangles (Figure 28.4).

Suppose that we want to compute an output array $\mathbf{y}$ of dimension N from an input array $\mathbf{x}$ of possibly different dimension M and a global parameter p with function F:

$$\mathbf{y}_i = F(i, \mathbf{x}, p), \qquad i = 1, \ldots, N\,.$$

To set up the GPU for this computation, we assign output array $\mathbf{y}$ to the output texture that is the current render target. Texture size is chosen according to the output size, and the viewport is set to cover the entire render target. A two-dimensional array of H horizontal resolution and V vertical resolution is capable of storing $H \times V$ elements. If $H \times V \geq N$, then it is up to us how horizontal and vertical resolutions are found. However, GPUs may impose restrictions, e.g. they cannot be larger than 2^{12} or, if we wish to use them as input textures in the next pass or compute binary reductions, the resolutions are preferred to be powers of two. If power of two dimensions are advantageous but the dimension of the array is different, we can extend

the array by additional void elements.

According to vector processing principles, different output values are computed independently without sharing temporary results. As in the GPU hardware the fragment shader is associated with the elements of the output data and can run independently of other elements, we use the fragment shader to evaluate function F. To find its parameters, we need to know i, i.e. which element is currently computed, and should have an access to input array $\mathbf{x}$. The simplest way is to store the input array as an input texture (or multiple input textures if that is more convenient) since the fragment shader can access textures.

The only responsibility of the CPU is to set the uniform parameters, specify the viewport and send a full screen quad down the pipeline. Uniform parameters select the input texture and define global parameter p. Assuming the OpenGL API, the corresponding CPU program in C would look like the following:

```
StartVectorOperation( ) {
    Set uniform parameters p and arrayX identifying the input texture

    glViewport(0, 0, H, V);     // Set horizontal and vertical resolutions, H and V
    glBegin(GL_QUADS);          // The next four vertices define a quad
        glVertex4f(-1,-1, 0, 1);   // Vertices assuming normalized device space
        glVertex4f(-1, 1, 0, 1);
        glVertex4f( 1, 1, 0, 1);
        glVertex4f( 1,-1, 0, 1);
    glEnd( );
}
```

Note that this program defines the rectangle directly in normalized device space using homogeneous coordinates passed as input parameters of the *glVertex4f* functions. So in the pipeline we should make sure that the vertices are not transformed.

For the shader program, the varying inputs are available in dedicated registers and outputs must also be written to dedicated registers. All of these registers are of type *float4,* that is, they can hold 4 float values. The role of the register is explained by its name. For example, the current value of the vertex position can be fetched from the ***POSITION*** register. Similar registers can store the texture coordinates or the color associated with this vertex.

The vertex shader gets the position of the vertex and is responsible for transforming it to the normalized device space. As we directly defined the vertices in normalized device space, the vertex shader simply copies the content of its input *POSITION* register to its output *POSITION* register (the input and output classification is given by the ***in*** and ***out*** keywords in front of the variable names assigned to registers):

```
void VertexShader( in float4  inputPos   : POSITION,
                   out float4 outputPos  : POSITION )
{
    outputPos = inputPos;
}
```

The geometry shader should keep the rectangle as it is without changing the vertex coordinates. As this is the default operation for the geometry shader, we do not specify any program for it. The rectangle leaving the geometry shader goes to the clipping stage, which keeps it since we defined our rectangle to be inside the clipping region. Then, Cartesian coordinates are obtained from the homogeneous ones by dividing the first three coordinates by the fourth one. As we set all fourth

homogeneous coordinates to 1, the first three coordinates are not altered. After homogeneous division, the fixed-function stage transforms the vertices of the rectangle to the vertices of a screen space rectangle having the x, y coordinates equal to the corners of the viewport and the $z = 0$ coordinate to 0.5. Finally, this rectangle is rasterized in screen space, so all pixels of the viewport are identified as a target, and the fragment shader is invoked for each of them.

The fragment shader is our real computing unit. It gets the input array and global parameter p as uniform parameters and can also find out which pixel is being computed by reading the ***WPOS*** register:

```
float FragmentShaderF(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT arrayX, // input array
        uniform float p             // global parameter p
        ) : COLOR   // output is interpreted as a pixel color
{
    float yi = F(index, arrayX, p);  // F is the function to be evaluated
    return yi;
}
```

In this program two input parameters were declared as uniform inputs by the ***uniform*** keyword, a float parameter p and the texture identification *arrayX*. The type of the texture is ***samplerRECT*** that determines the addressing modes how a texel can be selected. In this addressing mode, texel centers are on a two-dimensional integer grid. Note that here we used a different syntax to express what the output of the shader is. Instead of declaring a register as *out,* the output is given as a return value and the function itself, and is assigned to the output *COLOR* register.

28.3.1. Implementing the SAXPY BLAS function

To show concrete examples, we first implement the level 1 functionality of the ***Basic Linear Algebra Subprograms*** (***BLAS***) library (http://www.netlib.org/blas/) that evaluates vector functions of the following general form:

$$\mathbf{y} = p\mathbf{x} + \mathbf{y},$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors and p is a scalar parameter. This operation is called ***SAXPY*** in the BLAS library. Now our fragment shader inputs two textures, vector $\mathbf{x}$ and the original version of vector $\mathbf{y}$. One fragment shader processor computes a single element of the output vector:

```
float FragmentShaderSAXPY(
        in float2 index : WPOS,       // target pixel coordinates
        uniform samplerRECT arrayX,   // input array x
        uniform samplerRECT arrayY,   // original version of y
        uniform float p               // global parameter p
        ) : COLOR                     // output is interpreted as a pixel color
{
    float yoldi = texRECT(arrayY, index);    // yoldi = arrayY[index]
    float xi    = texRECT(arrayX, index);    // xi = arrayX[index]
    float yi = p * xi + yoldi;
    return yi;
}
```

Note that instead of indexing an array of CPU style programming, here we fetch the element from a texture that represents the array by the ***texRECT*** Cg function.

The first parameter of the *texRECT* function is the identification number of a two-dimensional texture, which is passed from the CPU as a uniform parameter, and the second is the texture address pointing to the texel to be selected.

Here we can observe how we can handle the limitation that a shader can only read textures but is not allowed to write into it. In the operation, vector $\mathbf{y}$ is an input and simultaneously also the output of the operation. To resolve this, we assign two textures to vector $\mathbf{y}$. One is the original vector in texture *arrayY,* and the other one is the render target. While we read the original value, the new version is produced without reading back from the render target, which would not be possible.

28.3.2. Image filtering

Another important example is the discrete convolution of two textures, an image and a filter kernel, which is the basic operation in many image processing algorithms:

$$\tilde{L}(X,Y) \approx \sum_{i=-M}^{M} \sum_{j=-M}^{M} L(X-i, Y-j) w(i,j)\,, \tag{28.1}$$

where $\tilde{L}(X,Y)$ is the filtered value at pixel X,Y, $L(X,Y)$ is the original image, and $w(x,y)$ is the ***filter kernel,*** which spans over $(2M+1) \times (2M+1)$ pixels.

Now the fragment shader is responsible for the evaluation of a single output pixel according to the input image given as texture *Image* and the filter kernel stored in a smaller texture *Weight.* The half size of the filter kernel M is passed as a uniform variable:

```
float3 FragmentShaderConvolution(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT Image,  // input image
        uniform samplerRECT Weight, // filter kernel
        uniform float M             // size of the filter kernel
        ) : COLOR  // a pixel of the filtered image
{
    float3 filtered = float3(0, 0, 0);

    for(int i = -M; i <= M; i++)
        for(int j = -M; j <= M; j++) {
            float2 kernelIndex = float2(i, j);
            float2 sourceIndex = index + kernelIndex;
            filtered += texRECT(Image, sourceIndex) * texRECT(Weight, kernelIndex);
        }
    }
    return filtered;
}
```

Note that this example was a linear, i.e. convolution filter, but non-linear filters (e.g. median filtering) could be implemented similarly. In this program we applied arithmetic operators (*, +=, =) for *float2* and *float3* type variables storing two and three floats, respectively. The Cg compiler and the GPU will execute these instructions independently on the float elements.

Note also that we did not care what happens at the edges of the image, the texture is always fetched with the sum of the target address and the shift of the filter kernel. A CPU implementation ignoring image boundaries would obviously be wrong, since we would over-index the source array. However, the texture fetching

hardware implementing, for example, the *texRECT* function automatically solves this problem. When the texture is initialized, we can specify what should happen if the texture coordinate is out of its domain. Depending on the selected option, we get the closest texel back, or a default value, or the address is interpreted in a periodic way.

Exercises

28.3-1 Following the vector processing concept, write a pixel shader which, when a full screen quad is rendered, quantizes the colors of an input texture to a few levels in all three channels, achieving a *cell shading* effect.

28.3-2 Following the gathering data processing scheme, write a pixel shader which, when a full screen quad is rendered, performs *median filtering* on an input grayscale image, achieving dot noise reduction. The shader should fetch nine texel values from a neighborhood of 3×3, outputting the fifth largest.

28.3-3 Implement an *anisotropic, edge preserving low-pass image filter* with the gathering data processing scheme. In order to preserve edges, compute the Euclidean distance of the original pixel color and the color of a neighboring pixel, and include the neighbor in the averaging operation only when the distance is below a threshold.

28.3-4 Write a parallel *Mandelbrot set rendering* program by assuming that pixel x, y corresponds to complex number $c = x + \mathbf{i}y$ and deciding whether or not the $z_n = z_{n-1}^2 + c$ iteration diverges when started from $z_0 = c$. The divergence may be checked by iterating $n = 10^6$ times and examining that $|z_n|$ is large enough. Divergent points are depicted with white, non-divergent points with black.

28.4. Beyond vector processing

Imagining the GPU as a vector processor is a simple but efficient application of the GPU hardware in general parallel processing. If the algorithm is suitable for vector processing, then this approach is straightforward. However, some algorithms are not good candidates for vector processing, but can still be efficiently executed by the GPU. In this section, we review the basic approaches that can extend the vector processing framework to make the GPU applicable for a wider range of algorithms.

28.4.1. SIMD or MIMD

Vector processors are usually SIMD machines, which means that they execute not only the same program for different vector elements but always the very same machine instruction at a time. It means that vector operations cannot contain data dependent conditionals or loop lengths depending on the actual data. There is only one control sequence in a SIMD parallel program.

Of course, writing programs without if conditionals and using only constants as loop cycle numbers are severe limitations, which significantly affects the program structure and the ease of development. Early GPU shaders were also SIMD type processors and placed the burden of eliminating all conditionals from the program

on the shoulder of the programmer. Current GPUs and their compilers solve this problem automatically, thus, on the programming level, we can use conditionals and variable length loops as if the shaders were MIMD computers. On execution level, additional control logic makes it possible that execution paths of scalar units diverge: in this case it is still a single instruction which is executed at a time, possibly with some scalar units being idle. Operations of different control paths are serialized so that all of them are completed. The overhead of serialization makes performance strongly dependent on the coherence of execution paths, but many transistors of control logic can be spared for more processing units.

The trick of executing all branches of conditionals with possibly disabled writes is called ***predication.*** Suppose that our program has an if statement like

```
if (condition(i)) {
    F( );
} else {
    G( );
}
```

Depending on the data, on some processors the *condition*(i) may be true, while it is false on other processors, thus our vector machine would like to execute function F of the first branch in some processors while it should evaluate function G of the second branch in other processors. As in SIMD there can be only one control path, the parallel system should execute both paths and disable writes when the processor is not in a valid path. This method converts the original program to the following conditional free algorithm:

```
enableWrite = condition(i);
F( );
enableWrite = !enableWrite;
G( );
```

This version does not have conditional instructions so it can be executed by the SIMD machine. However, the computation time will be the the sum of computation times of the two functions.

This performance bottleneck can be attacked by decomposing the computation into multiple passes and by the exploitation of the feature. The early z-cull compares the z value of the fragment with the content of the depth buffer, and if it is smaller than the stored value, the fragment shader is not called for this fragment but the fragment processor is assigned to another data element. The early z-cull is enabled automatically if we execute fragment programs that do not modify the fragment's z coordinate (this is the case in all examples discussed so far).

To exploit this feature, we decompose the computation into three passes. In the first pass, only the *condition* is evaluated and the depth buffer is initialized with the values. Recall that if the z value is not modified, our full screen quad is on the xy plane in normalized device space, so it will be on the $z = 0.5$ plane in screen space. Thus, to allow a discrimination according to the condition, we can set values in the range $(0.5, 1)$ if the condition is true and in $(0, 0.5)$ if it is false.

The fragment shader of the first pass computes just the condition values and stores them in the depth buffer:

```
float FragmentShaderCondition(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT Input,  // input vector
```

```
        ) : DEPTH  // the output goes to the depth buffer
{
    bool condition = ComputeCondition( texRECT(Input, index) );
    return (condition) ? 0.8 : 0.2; // 0.8 is greater than 0.5; 0.2 is smaller than 0.5
}
```

Then we execute two passes for the evaluation of functions F and G, respectively. In the first pass, the fragment shader computes F and the depth comparison is set to pass those fragments where their $z = 0.5$ coordinate is less than the depth value stored in the depth buffer. In this pass, only those fragments are evaluated where the depth buffer has 0.8 value, i.e. where the previous condition was true. Then, in the second pass, the fragment shader is set to compute G while the depth buffer is turned to keep those fragments where the fragment's depth is greater than the stored value.

In Subsection 28.7.1 we exploit early z-culling to implement a variable length loop in fragment processors.

28.4.2. Reduction

The vector processing principle assumes that the output is an array where elements are obtained independently. The array should be large enough to keep every shader processor busy. Clearly, if the array has just one or a few elements, then only one or a few shader processors may work at a time, so we loose the advantages of parallel processing.

In many algorithms, the final result is not a large array, but is a single value computed from the array. Thus, the algorithm should reduce the dimension of the output. Doing the in a single step by producing a single texel would not benefit from the parallel architecture. Thus, reduction should also be executed in parallel, in multiple steps. This is possible if the operation needed to compute the result from the array is associative, which is the case for the most common operations, like sum, average, maximum, or minimum.

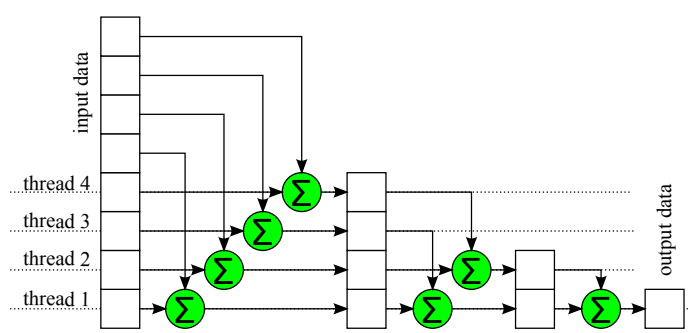

Figure 28.5 An example for parallel reduction that sums the elements of the input vector.

Suppose that the array is encoded by a two-dimensional texture. At a single phase, we downsample the texture by halving its linear resolution, i.e. replacing four neighboring texels by a single texel. The fragment shaders will compute the operation on four texels. If the original array has $2^n \times 2^n$ resolution, then n reduction steps are

needed to obtain a single 1×1 output value. In the following example, we compute the sum of the elements of the input array (Figure 28.5). The CPU program renders a full screen quad in each iteration having divided the render target resolution by two:

```
Reduction( ) {
    Set uniform parameter arrayX to identify the input texture

    for(N /= 2 ; N >= 1; N /= 2) {  // log_2 N iterations
        glViewport(0, 0, N, N);     // Set render target dimensions to hold NxN elements
        glBegin(GL_QUADS);          // Render a full screen quad
             glVertex4f(-1,-1, 0, 1);
             glVertex4f(-1, 1, 0, 1);
             glVertex4f( 1, 1, 0, 1);
             glVertex4f( 1,-1, 0, 1);
        glEnd( );

        Copy render target to input texture arrayX
    }
}
```

The fragment shader computes a single reduced texel from four texels as a summation in each iteration step:

```
float FragmentShaderSum( ) (
        in float2 index : WPOS,      // target pixel coordinates
        uniform samplerRECT arrayX,  // input array x
        ) : COLOR                    // output is interpreted as a pixel color
{
    float sum = texRECT(arrayX, 2 * index);
    sum += texRECT(arrayX, 2 * index + float2(1, 0));
    sum += texRECT(arrayX, 2 * index + float2(1, 1));
    sum += texRECT(arrayX, 2 * index + float2(0, 1));
    return sum;
}
```

Note that if we exploited the bi-linear filtering feature of the texture memory, then we could save three texture fetch operations and obtain the average in a single step.

28.4.3. Implementing scatter

In vector processing a processor is assigned to each output value, i.e. every processor should be aware which output element it is computing and it is not allowed to deroute its result to somewhere else. Such a static assignment is appropriate for ***gathering*** type computations. The general structure of gathering is that we may rely on a dynamically selected set of input elements but the variable where the output is stored is known a-priory:

```
index = ComputeIndex( );     // index of the input data
y = F(x[index]);
```

Opposed to gathering, algorithms may have ***scattering*** characteristics, i.e. a given input value may end up in a variable that is selected dynamically. A simple scatter operation is:

```
index = ComputeIndex( );     // index of the output data
y[index] = F(x);
```

Vector processing frameworks and our fragment shader implementation are unable to implement scatter since the fragment shader can only write to the pixel it

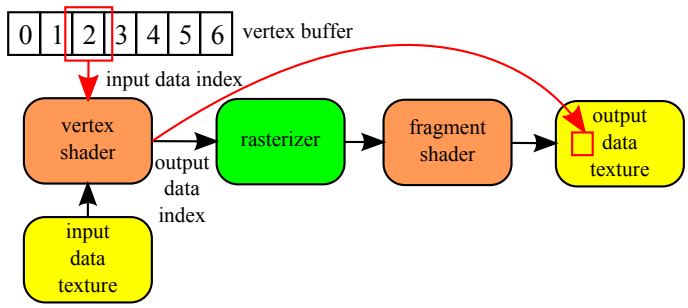

Figure 28.6 Implementation of scatter.

has been assigned to.

If we wish to solve a problem having scattering type algorithm on the GPU, we have two options. First, we can restructure the algorithm to be of gathering type. Converting scattering type parallel algorithms to gathering type ones requires a change of our viewpoint how we look at the problem and its solution. For example, when integral equations or transport problems are considered, this corresponds to the solution of the adjoint problem [19]. Secondly, we can move the index calculation up to the pipeline and use the rasterizer to establish the dynamic correspondence between the index and the render target (Figure 28.6).

Let us consider a famous scattering type algorithm, ***histogram generation.*** Suppose we scan an input array $\mathbf{x}$ of dimension M, evaluate function F for the elements, and calculate output array $\mathbf{y}$ of dimension N that stores the number of function values that are in bins equally subdividing range $(Fmin, Fmax)$.

A scalar implementation of histogram generation would be:

```
Histogram( x ) {
    for(int i = 0; i < M; i++) {
        index = (int)((F(x[i]) - Fmin)/(Fmax - Fmin) * N); // bin
        index = max(index, 0);
        index = min(index, N-1);
        y[index] = y[index] + 1;
    }
}
```

We can see that the above function writes to the output array at random locations, meaning it cannot be implemented in a fragment shader which is only allowed to write the render target at its dedicated index. The problem of scattering will be solved by computing the index in the vertex shader but delegating the responsibility of incrementing to the rest of the pipeline. The indices are mapped to output pixels by the rasterization hardware. The problem of read-modify-write cycles might be solved by starting a new pass after each increment operation and copying the current render target as an input texture of the next rendering pass. However, this solution would have very poor performance and would not utilize the parallel hardware at all. A much better solution uses the arithmetic capabilities of the merging unit. The fragment shader generates just the increment (i.e. value 1) where the histogram needs to be updated and gives this value to the merging unit. The merging unit, in turn, adds the increment to the content of the render target.

The CPU program generates a point primitive for each input data element. Additionally, it sets the render target to match the output array and also enables the merging unit to execute add operations:

```
ScanInputVector( ) {
    Set uniform parameters Fmin, Fmax, N

    glDisable(GL_DEPTH_TEST);     // Turn depth buffering off
    glBlendFunc(GL_ONE, GL_ONE); // Blending operation: dest = source * 1 + dest * 1;
    glEnable(GL_BLEND);           // Enable blending

    glViewport(0, 0, N, 1);       // Set render target dimensions to hold N elements
    glBegin(GL_POINTS);           // Assign a point primitive to each input elements
    for(int i = 0; i < M; i++) {
        glVertex1f( x[i] );       // an input element as a point primitive
    }
    glEnd( );
}
```

The vertex positions in this level are not important since it turns out later where this point will be mapped. So we use the first coordinate of the vertex to pass the current input element $x[i]$.

The vertex shader gets the position of the vertex currently storing the input element, and finds the location of this point in normalized device space. First, function F is evaluated and the bin index is obtained, then we convert this index to the $[-1, 1]$ range since in normalized device space these will correspond to the extremes of the viewport:

```
void VertexShaderHistogram(
    in float   inputPos   : POSITION,
    out float4 outputPos  : POSITION,
    uniform float Fmin,
    uniform float Fmax,
    uniform float N )
{
    float xi = inputPos;
    int index = (int)((F(xi) - Fmin)/(Fmax - Fmin) * N); // bin
    index = max(index, 0);
    index = min(index, N-1);
    float nindex = 2.0 * index / N - 1.0;   // normalized device space
    outputPos = float4(nindex, 0, 0, 1);    // set output coordinates
}
```

The above example is not optimized. Note that the index calculation and the normalization could be merged together and we do not even need the size of the output array N to execute this operation.

The fragment shader will be invoked for the pixel on which the point primitive is mapped. It simply outputs an increment value of 1:

```
float FragmentShaderIncr( ) : COLOR   // output is interpreted as a pixel color
{
    return 1;  // increment that is added to the render target by merging
}
```

28.4.4. Parallelism versus reuse

Parallel processors running independently offer a linear speed up over equivalent scalar processor implementations. However, scalar processors may benefit from recognizing similar parts in the computation of different output values, so they can

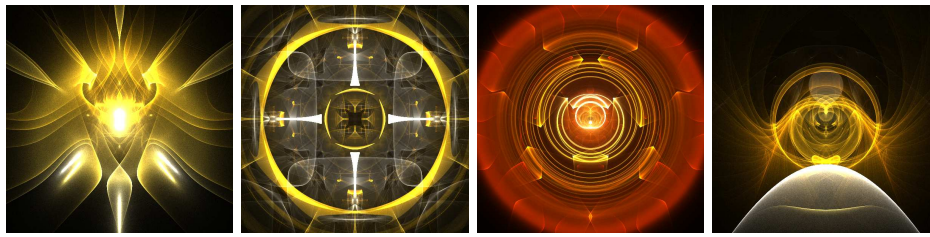

Figure 28.7 Caustics rendering is a practical use of histogram generation. The illumination intensity of the target will be proportional to the number of photons it receives (images courtesy of Dávid Balambér).

increase their performance utilizing ***reuse.*** As parallel processors may not reuse data generated by other processors, their comparative advantages become less attractive.

GPUs are parallel systems of significant streaming capabilities, so if data that can be reused are generated early, we can get the advantages of both independent parallel processing and the reuse features of scalar computing.

Our main stream expander is the rasterization. Thus anything happens before rasterization can be considered as a global computation for all those pixels that are filled with the rasterized version of the primitive. Alternatively, the result of a pass can be considered as an input texture in the next pass, so results obtained in the previous pass can be reused by all threads in the next pass.

Exercises

28.4-1 Implement a *parallel regula falsi equation solver* for $(2-a-b)x^3 + ax^2 + bx - 1 = 0$ that searches for roots in $[0, 1]$ for many different a and b parameters. The a and b parameters are stored in a texture and the pixel shader is responsible for iteratively solving the equation for a particular parameter pair. Terminate the iteration when the error is below a given threshold. Take advantage of the early z-culling hardware to prevent further refinement of the terminated iterations. Analyze the performance gain.

28.4-2 Based on the reduction scheme, write a program which applies simple linear *tone mapping* to a high dynamic range image stored in a floating-point texture. The scaling factor should be chosen to map the maximum texel value to the value of one. Find this maximum using iterative reduction of the texture.

28.4-3 Based on the concept of scatter, implement a *caustics renderer* program (Figure 28.7). The scene includes a point light source, a glass sphere, and a diffuse square that is visualized on the screen. Photons with random directions are generated by the CPU and passed to the GPU as point primitives. The vertex shader traces the photon through possible reflections or refractions and decides where the photon will eventually hit the diffuse square. The point primitive is directed to that pixel and the photon powers are added by additive alpha blending.

28.4-4 Based on the concept of scatter, given an array of GSM transmitter tower coordinates, compute cell phone signal strength on a 2D grid. Assume signal strength

diminishes linearly with the distance to the nearest transmitter. Use the rasterizer to render circular features onto a 2D render target, and set up blending to pick the maximum.

28.5. GPGPU programming model: CUDA and OpenCL

The ***Compute Unified Device Architecture*** (***CUDA***) and the interfaces provide the programmer with a programming model that is significantly different from the graphics pipeline model (right of Figure 28.1). It presents the GPU as a collection of multiprocessors where each multiprocessor contains several SIMD scalar processors. Scalar processors have their own registers and can communicate inside a multiprocessor via a fast ***shared memory.*** Scalar processors can read cached textures having built-in filtering and can read or write the slow global memory. If we wish, even read-modify-write operations can also be used. Parts of the global memory can be declared as a texture, but from that point it becomes read-only.

Unlike in the graphics API model, the write to the global memory is not exclusive and ***atomic add*** operations are available to support semaphores and data consistency. The fixed-function elements like clipping, rasterization, and merging are not visible in this programming model.

Comparing the GPGPU programming model to the graphics API model, we notice that it is cleaner and simpler. In the GPGPU programming model, parallel processors are on the same level and can access the global memory in an unrestricted way, while in the graphics API model, processors and fixed-function hardware form streams and write is possible only at the end of the stream. When we program through the GPGPU model, we face less restrictions than in the graphics pipeline model. However, care should be practiced since the graphics pipeline model forbids exactly those features that are not recommended to use in high performance applications.

The art of programming the GPGPU model is an efficient decomposition of the original algorithm to parallel threads that can run with minimum amount of data communication and synchronization, but always keep most of the processors busy. In the following sections we analyze a fundamental operation, the matrix-vector multiplication, and discuss how these requirements can be met.

28.6. Matrix-vector multiplication

Computational problems are based on mathematical models and their numerical solution. The numerical solution methods practically always rely on some kind of linearization, resulting in algorithms that require us to solve linear systems of equations and perform matrix-vector multiplication as a core of the iterative solution. Thus, matrix-vector multiplication is a basic operation that can be, if implemented efficiently on the parallel architecture, the most general building block in any nu-

merical algorithm. We define the basic problem to be the computation of the result vector $\mathbf{y}$ from input matrix $\mathbf{A}$, vectors $\mathbf{x}$ and $\mathbf{b}$, as

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}\,.$$

We call this the MV problem. Let $N \times M$ be the dimensions of matrix $\mathbf{A}$. As every input vector element may contribute to each of the output vector elements, a scalar CPU implementation would contain a double loop, one loop scans the input elements while the other the output elements. If we parallelize the algorithm by assigning output elements to parallel threads, then we obtain a gathering type algorithm where a thread gathers the contributions of all input elements and aggregates them to the thread's single output value. On the other hand, if we assigned parallel threads to input elements, then a thread would compute the contribution of this input element to all output elements, which would be a scatter operation. In case of gathering, threads share only input data but their output is exclusive so no synchronization is needed. In case of scattering, multiple threads may add their contribution to the same output element, so atomic adds are needed, which may result in performance degradation.

An implementation of the matrix-vector multiplication on a scalar processor looks like the following:

```
void ScalarMV(int N, int M, float* y, const float* A, const float* x, const float* b)
{
    for(int i=0; i<N; i++) {
        float yi = b[i];
        for(int j=0; j<M; j++) yi += A[i * M + j] * x[j];
        y[i] = yi;
    }
}
```

The first step of porting this algorithm to a parallel machine is to determine what a single thread would do from this program. From the options of gathering and scattering, we should prefer gathering since that automatically eliminates the problems of non-exclusive write operations. In a gathering type solution, a thread computes a single element of vector $\mathbf{y}$ and thus we need to start N threads. A GPU can launch a practically unlimited number of threads that are grouped in thread blocks. Threads of a block are assigned to the same multiprocessor. So the next design decision is how the N threads are distributed in blocks. A multiprocessor typically executes 32 threads in parallel, so the number of threads in a block should be some multiple of 32. When the threads are halted because of a slow memory access, a hardware scheduler tries to continue the processing of other threads, so it is wise to assign more than 32 threads to a multiprocessor to always have threads that are ready to run. However, increasing the number of threads in a single block may also mean that at the end we have just a few blocks, i.e. our program will run just on a few multiprocessors. Considering these, we assign 256 threads to a single block and hope that $N/256$ exceeds the number of multiprocessors and thus we fully utilize the parallel hardware.

There is a slight problem if N is not a multiple of 256. We should assign the last elements of the vector to some processors as well, so the thread block number should be the ceiling of $N/256$. As a result of this, we shall have threads that are not associated with vector elements. It is not a problem if the extra threads can detect

it and cause no harm, e.g. they do not over-index the output array.

Similarly to the discussed vector processing model, a thread should be aware which output element it is computing. The CUDA library provides implicit input parameters that encode this information: *blockIdx* is the index of the thread block, *blockDim* is the number of threads in a block, and *threadIdx* is the index of the thread inside the block.

The program of the CUDA kernel computing a single element of the output vector is now a part of a conventional CPU program:

```
__global__ void cudaSimpleMV(int N, int M, float* y, float* A, float* x, float* b)
{
    // Determine element to process from thread and block indices
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) {  // if the index is out of the range of the output array, skip.
        float yi = b[i];
        for(int j=0; j<M; j++) yi += A[i * M + j] * x[j];
        y[i] = yi;
    }
}
```

The ***global*** keyword tells the compiler that this function will run not on the CPU but on the GPU and it may be invoked from the CPU as well. The parameters are passed according to the normal C syntax. The only special feature is the use of the implicit parameters to compute the identification number of this thread, which is the index of the output array.

The kernels are started from a CPU program that sets the parameters and also defines the number of thread blocks and the number of threads inside a block.

```
__host__ void run_cudaSimpleMV()
{
    int threadsPerBlock = 256; // number of threads per block
    int blockNum = (N +  threadsPerBlock - 1)/threadsPerBlock; // number of blocks
    cudaSimpleMV<<<blockNum, threadsPerBlock>>>(N, M, y, A, x, b);
}
```

The compiler will realize that this function runs on the CPU by reading the ***host*** keyword. The parallel threads are started like a normal C function call with the exception of the *<blockNum, threadsPerBlock>* tag, which defines how many threads should be started and how they are distributed among the multiprocessors.

28.6.1. Making matrix-vector multiplication more parallel

So far, we assigned matrix rows to parallel threads and computed scalar product $\mathbf{A}_i\mathbf{x}$ serially inside threads. If the number of matrix rows is less than the number of parallel scalar processors, this amount of parallelization is not enough to supply all processing units with work to do, and the execution of individual threads will be lengthy. Reformulating the scalar product computation is a well known, but tougher parallelization problem, as the additions cannot be executed independently, and we require a single scalar to be written for every row of the matrix. However, parts of the summation can be executed independently, and then the results added. This is a classic example of . It is required that the threads whose results are to be added both finish execution and write their results to where they are accessible for the thread that needs to add them. Thus, we use ***thread synchronization*** and available only for the threads of the same block.

Let us assume first—unrealistically—that we can have M threads processing a row and the shared memory can hold M floating point values. Let $\mathbf{Q}$ be the vector of length M residing in shared memory. Then, every thread can compute one element Q_j as $A_{ij}x_j$.. Finally, elements of $\mathbf{Q}$ must be reduced by summation. Let us further assume that $M = 2^k$.. The reduction can be carried out in k steps, terminating half of the threads, while each surviving thread adds the value in $\mathbf{Q}$ computed by a terminated one to its own. The final remaining thread outputs the value to the global memory.

```
#define M THE_NUMBER_OF_MATRIX_COLUMNS
__global__ void cudaReduceMV(int N, float* y, float* A, float* x, float* b)
{
    int i = blockIdx.x;
    int j = threadIdx.x;

    __shared__ float Q[M];  // in the shader memory inside a multiprocessor

    Q[j] = A[i * M + j] * x[j];  // a parallel part of matrix-vector multiplication

    for(int stride = M / 2; stride > 0; stride >>= 1) // reduction
    {
        __syncthreads();    // wait until all other threads of the block arrive this point
        if(j + stride < M)
            Q[j] += Q[j + stride];
    }

    if(j == 0)              // reduced to a single element
        y[i] = Q[0] + b[i];
}

__host__ void run_cudaReduceMV()
{
    cudaReduceMV<<< N, M >>>(N, y, A, x, b);
}
```

For practical matrix dimensions ($M > 10^4$), neither the number of possible threads of a single multiprocessor nor the size of the shared memory is enough to process all elements in parallel. In our next example, we use a single block of threads with limited size to process a large matrix. First, we break the output vector into segments of size T. Elements within such a segment are evaluated in parallel, then the threads proceed to the next segment. Second, for every scalar product computation, we break the vectors $\mathbf{A}_i$ and $\mathbf{x}$ into segments of length Z. We maintain a shared vector $\mathbf{Q}_t$ of length Z for every row being processed in parallel. We can compute the elementwise product of the $\mathbf{A}_i$ and $\mathbf{x}$ segments in parallel, and add it to $\mathbf{Q}_t$. As T rows are being processed by Z threads each, the block will consist of $T \times Z$ threads. From one thread's perspective this means it has to loop over $\mathbf{y}$ with a stride of T, and for every such element in $\mathbf{y}$, loop over $\mathbf{A}_i$ and $\mathbf{x}$ with a stride of Z. Also for every element in $\mathbf{y}$, the contents of $\mathbf{Q}_t$ must be summed by reduction as before. The complete kernel which works with large matrices would then be:

```
__global__ void cudaLargeMV(int N, int M, float* y, float* A, float* x, float* b)
{
    __shared__ float Q[T * Z]; // stored in the shared memory inside a multiprocessor

    int t = threadIdx.x / Z;
    int z = threadIdx.x % Z;

    for(int i = t; i < N; i += T)
    {
```

```
        Q[t * Z + z] = 0;
        for(int j = z; j < M; j += Z)
            Q[t * Z + z] += A[i * M + j] * x[j];

        for(int stride = Z / 2; stride > 0; stride >>= 1)
        {
            __syncthreads();
            if(z + stride < Z)
                Q[t * Z + z] += Q[t * Z + z + stride];
        }

        if(z == 0)
            y[i] = Q[t * Z + 0] + b[i];
    }
}

__host__ void run_cudaLargeMV()
{
    cudaReduceMV<<< 1, T*Z >>>(N, M, y, A, x, b);
}
```

This can easily be extended to make use of multiple thread blocks by restricting the outer loop to only a fraction of the matrix rows based on the *blockIdx* parameter.

The above algorithm uses shared memory straightforwardly and allows us to align memory access of threads through a proper choice of block sizes. However, every element of vector $\mathbf{x}$ must be read once for the computation of every row. We can improve on this if we read values of $\mathbf{x}$ into the shared memory and have threads in one block operate on multiple rows of the matrix. This, however, means we can use less shared memory per line to parallelize summation. The analysis of this trade-off is beyond the scope of this chapter, but a block size of 64×8 has been proposed in [5]. With such a strategy it is also beneficial to access matrix $\mathbf{A}$ as a texture, as data access will exhibit 2D locality, supported by texture caching hardware.

Even though matrix-vector multiplication is a general mathematical formulation for a wide range of computational problems, the arising matrices are often large, but sparse. In case of sparse matrices, the previously introduced matrix-vector multiplication algorithms will not be efficient as they explicitly compute multiplication with zero elements. Sparse matrix representations and MV algorithms are discussed in [1].

Exercises

28.6-1 Implement matrix-vector multiplication for large matrices in CUDA. Compare results to a CPU implementation.

28.6-2 Implement an inverse iteration type Julia set renderer. The Julia set is the attractor of the $z_n = z_{n-1}^2 + c$ iteration where z_n and c are complex numbers. Inverse iteration starts from a fixed point of the iteration formula, and iterates the inverse mapping, $z_n = \pm\sqrt{z_n - c}$ by randomly selecting either $\sqrt{z_n - c}$ or $-\sqrt{z_n - c}$ from the two possibilities. Threads must use pseudo-random generators that are initialized with different seeds. Note that CUDA has no built-in random number generator, so implement one in the program.

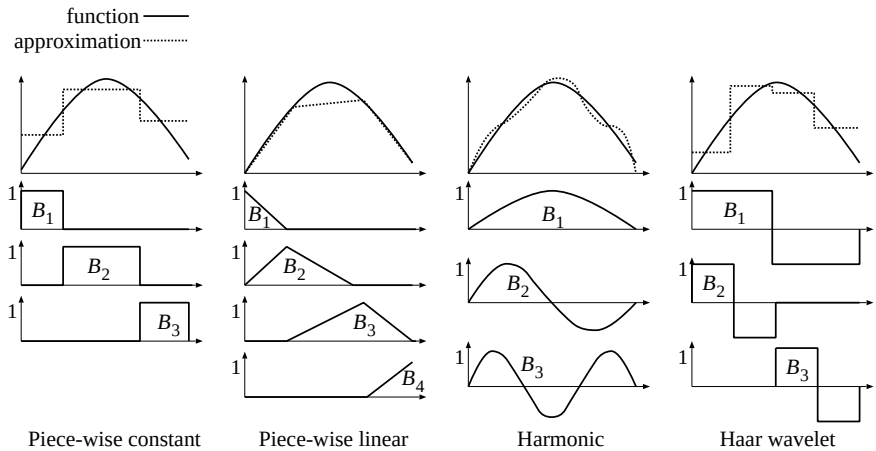

Figure 28.8 Finite element representations of functions. The texture filtering of the GPU directly supports finite element representations using regularly placed samples in one-, two-, and three-dimensions and interpolating with piece-wise constant and piece-wise linear basis functions.

28.7. Case study: computational fluid dynamics

Problems emerging in physics or engineering are usually described mathematically as a set of partial differential or integral equations. As physical systems expand in space and time, derivatives or integrals should be evaluated both in temporal and spatial domains.

When we have to represent a value over space and time, we should use functions having the spatial position and the time as their variables. The representation of general functions would require infinite amount of data, so in numerical methods we only approximate them with finite number of values. Intuitively, these values can be imagined as the function values at discrete points and time instances. The theory behind this is the ***finite element method.*** If we need to represent function $f(\vec{r})$ with finite data, we approximate the function in the following finite series form (Figure 28.8):

$$f(\vec{r}) \approx \tilde{f}(\vec{r}) = \sum_{i=1}^{N} f_i B_i(\vec{r}),$$

where $B_1(\vec{r}), \ldots, B_N(\vec{r})$ are pre-defined ***basis functions*** and $f_1, \ldots, f_N$ are the coefficients that describe $\tilde{f}$.

A particularly simple finite element representation is the piece-wise linear scheme that finds possibly regularly placed sample points $\vec{r}_1, \ldots, \vec{r}_N$ in the domain, evaluates the function at these points to obtain the coefficients $f_i = f(\vec{r}_i)$ and linearly interpolates between $\vec{r}_i$ and $\vec{r}_{i+1}$.

When the system is dynamic, solution f will be time dependent, so a new finite element representation is needed for every time instance. We have basically two

options for this. We can set sample points $\vec{r}_1, \ldots, \vec{r}_N$ in a static way and allow only coefficients f_i to change in time. This approach is called ***Eulerian.*** On the other hand, we can also allow the sample points to move with the evaluation of the system, making also sample points $\vec{r}_i$ time dependent. This is the ***Lagrangian*** approach, where sample locations are also called ***particles.***

Intuitive examples of Eulerian and Lagrangian discretization schemes are how temperature and other attributes are measured in meteorology. In ground stations, these data are measured at fixed locations. However, meteorological balloons can also provide the same data, but from varying positions that follow the flow of the air.

In this section we discuss a case study for GPU-based scientific computation. The selected problem is ***computational fluid dynamics.*** Many phenomena that can be seen in nature like smoke, cloud formation, fire, and explosion show fluid-like behavior. Understandably, there is a need for good and fast fluid solvers both in engineering and in computer animation.

The mathematical model of the fluid motion is given by the Navier-Stokes equation. First we introduce this partial differential equation, then discuss how GPU-based Eulerian and Langrangian solvers can be developed for it.

A fluid with constant density and temperature can be described by its velocity $\vec{v} = (v_x, v_y, v_z)$ and pressure p fields. The velocity and the pressure vary both in space and time:

$$\vec{v} = \vec{v}(\vec{r}, t), \qquad p = p(\vec{r}, t)\,.$$

Let us focus on a fluid element of unit volume that is at point $\vec{r}$ at time t. At an earlier time instance $t - \mathrm{d}t$, this fluid element was in $\vec{r} - \vec{v}\mathrm{d}t$ and, according to the fundamental law of dynamics, its velocity changed according to an acceleration that is equal to total force $\vec{F}$ divided by mass ρ of this unit volume fluid element:

$$\vec{v}(\vec{r}, t) = \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t) + \frac{\vec{F}}{\rho}\mathrm{d}t\,.$$

Mass ρ of a unit volume fluid element is called the ***fluid density.*** Moving the velocity terms to the left side and dividing the equation by $\mathrm{d}t$, we can express the ***substantial derivative*** of the velocity:

$$\frac{\vec{v}(\vec{r}, t) - \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t)}{\mathrm{d}t} = \frac{\vec{F}}{\rho}\,.$$

The total force can stem from different sources. It may be due to the pressure differences:

$$\vec{F}_{\text{pressure}} = -\vec{\nabla}p = -\left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}\right),$$

where $\vec{\nabla}p$ is the ***gradient*** of the pressure field. The minus sign indicates that the pressure accelerates the fluid element towards the low pressure regions. Here we used the ***nabla operator,*** which has the following form in a Cartesian coordinate system:

$$\vec{\nabla} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right).$$

Due to friction, the fluid motion is damped. This damping depends on the ***viscosity*** ν of the fluid. Highly viscous fluids like syrup stick together, while low-viscosity fluids flow freely. The total damping force is expressed as a diffusion term since the viscosity force is proportional to the ***Laplacian*** of the velocity field:

$$\vec{F}_{\text{viscosity}} = \nu\vec{\nabla}^2\vec{v} = \nu\left(\frac{\partial^2\vec{v}}{\partial x^2} + \frac{\partial^2\vec{v}}{\partial y^2} + \frac{\partial^2\vec{v}}{\partial z^2}\right).$$

Finally, an external force field $\vec{F}_{\text{external}}$ may also act on our fluid element causing acceleration. In the gravity field of the Earth, assuming that the vertical direction is axis z, this external acceleration is $(0, 0, -g)$ where $g = 9.8\ [m/s^2]$.

Adding the forces together, we can obtain the ***Navier-Stokes equation*** for the velocity of our fluid element:

$$\rho\frac{\vec{v}(\vec{r},t) - \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t)}{\mathrm{d}t} = -\vec{\nabla}p + \nu\vec{\nabla}^2\vec{v} + \vec{F}_{\text{external}}\,.$$

In fact, this equation is the adaptation of the fundamental law of dynamics for fluids. If there is no external force field, the momentum of the dynamic system must be preserved. This is why this equation is also called ***momentum conservation equation.***

Closed physics systems preserve not only the momentum but also the mass, so this aspect should also be built into our fluid model. Simply put, the mass conservation means that what flows into a volume must also flow out, so the divergence of the mass flow is zero. If the fluid is incompressible, then the fluid density is constant, thus the mass flow is proportional to the velocity field. For incompressible fluids, the mass conservation means that the velocity field is ***divergence*** free:

$$\vec{\nabla}\cdot\vec{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} = 0\,. \tag{28.2}$$

28.7.1. Eulerian solver for fluid dynamics

The Eulerian approach tracks the evolution of the velocity and pressure fields on fixed, uniform grid points. The grid allows a simple approximation of spatial derivatives by finite differences. If the grid points are in distances Δx, Δy, and Δz along the three coordinate axes and the values of scalar field p and vector field $\vec{v}$ at grid point (i, j, k) are $p^{i,j,k}$ and $\vec{v}^{i,j,k}$, respectively, then the gradient, the divergence and the Laplacian operators can be approximated as:

$$\vec{\nabla}p \approx \left(\frac{p^{i+1,j,k} - p^{i-1,j,k}}{2\Delta x}, \frac{p^{i,j+1,k} - p^{i,j-1,k}}{2\Delta y}, \frac{p^{i,j,k+1} - p^{i,j,k-1}}{2\Delta x}\right), \tag{28.3}$$

$$\vec{\nabla}\cdot\vec{v} \approx \frac{v_x^{i+1,j,k} - v_x^{i-1,j,k}}{2\Delta x} + \frac{v_y^{i,j+1,k} - v_y^{i,j-1,k}}{2\Delta y} + \frac{v_z^{i,j,k+1} - v_z^{i,j,k-1}}{2\Delta z}, \tag{28.4}$$

$$\vec{\nabla}^2 p \approx \frac{p^{i+1,j,k} - 2p^{i,j,k} + p^{i-1,j,k}}{(\Delta x)^2} + \frac{p^{i,j+1,k} - 2p^{i,j,k} + p^{i,j-1,k}}{(\Delta x)^2}$$

$$+\frac{p^{i,j,k+1}-2p^{i,j,k}+p^{i,j,k-1}}{(\Delta x)^2}\,. \tag{28.5}$$

The Navier-Stokes equation and the requirement that the velocity is divergence free define four scalar equations (the conservation of momentum is a vector equation) with four scalar unknowns (v_x, v_y, v_z, p). The numerical solver computes the current fields advancing the time in discrete steps of length Δt:

$$\vec{v}(\vec{r},t)=\vec{v}(\vec{r}-\vec{v}\Delta t,t-\Delta t)+\frac{\nu\Delta t}{\rho}\vec{\nabla}^2\vec{v}+\frac{\Delta t}{\rho}\vec{F}_{\text{external}}-\frac{\Delta t}{\rho}\vec{\nabla}p\,.$$

The velocity field is updated in several steps, each considering a single term on the right side of this equation. Let us consider these steps one-by-one.

Advection

To initialize the new velocity field at point $\vec{r}$, we fetch the previous field at position $\vec{r}-\vec{v}\Delta t$ since the fluid element arriving at point $\vec{r}$ was there in the previous time step [17]. This step computes advection, i.e. the phenomenon that the fluid carries its own velocity field:

$$\vec{w}_1(\vec{r})=\vec{v}(\vec{r}-\vec{v}\Delta t,t-\Delta t)\,.$$

Diffusion

To damp the velocity field, we could update it proportionally to a diffusion term:

$$\vec{w}_2=\vec{w}_1+\frac{\nu\Delta t}{\rho}\vec{\nabla}^2\vec{w}_1\,.$$

However, this type of ***forward Euler integrator*** is numerically unstable. The reason of instability is that forward methods predict the future based on the present values, and as time passes, each simulation step adds some error, which may accumulate and exceed any limit.

Unlike forward integrators, a backward method can guarantee stability. A backward looking approach is stable since while predicting the future, it simultaneously corrects the past. Thus, the total error converges to a finite value and remains bounded. Here a backward method means that the Laplacian is obtained from the future, yet unknown velocity field, and not from the current velocity field:

$$\vec{w}_2=\vec{w}_1+\frac{\nu\Delta t}{\rho}\vec{\nabla}^2\vec{w}_2\,. \tag{28.6}$$

At this step of the computation, the advected field $\vec{w}_1$ is available at the grid points, the unknowns are the diffused velocity $\vec{w}_2^{i,j,k}$ for each of the grid points. Using (28.5) to compute the Laplacian of the x,y,z coordinates of unknown vector field $\vec{w}_2$ at grid point (i,j,k), we observe that it will be a linear function of the $\vec{w}_2$ velocities in the (i,j,k) grid point and its neighbors. Thus, (28.6) is a sparse linear system of equations:

$$\mathbf{w}_2=\mathbf{w}_1+\mathbf{A}\cdot\mathbf{w}_2 \tag{28.7}$$

where vector $\mathbf{w}_1$ is the vector of the known velocities obtained by advection, $\mathbf{w}_2$ is

the vector of unknown velocities of the grid points, and matrix-vector multiplication $\mathbf{A} \cdot \mathbf{w}_2$ represents the discrete form of $(\nu \Delta t/\rho)\nabla^2 \vec{w}_2(\vec{r})$.

Such systems are primary candidates for ***Jacobi iteration*** (see Chapter 12 of this book, titled *Scientific Computation*). Initially we fill vector $\mathbf{w}_2$ with zero and evaluate the right side of (28.7) iteratively, moving the result of the previous step to vector $\mathbf{w}_2$ of the right side. Thus, we traced back the problem to a sequence of sparse vector-matrix multiplications. Note that matrix $\mathbf{A}$ needs not be stored. When velocity field $\vec{w}_2$ is needed at a grid point, the neighbors are looked up and the simple formula of (28.5) gives us the result.

Updating a value in a grid point according to its previous value and the values of its neighbors are called ***image filtering.*** Thus, a single step of the Jacobi iteration is equivalent to an image filtering operation, which is discussed in Section 28.3.2.

External force field

The external force accelerates the velocity field at each grid point:

$$\vec{w}_3 = \vec{w}_2 + \frac{\Delta t}{\rho} \vec{F}_{\text{external}} \,.$$

Projection

So far, we calculated an updated velocity field $\vec{w}_3$ without considering the unknown pressure field. In the projection step, we compute the unknown pressure field p and update the velocity field with it:

$$\vec{v}(t) = \vec{w}_3 - \frac{\Delta t}{\rho} \vec{\nabla} p \,.$$

The pressure field is obtained from the requirement that the final velocity field must be divergence free. Let us apply the divergence operator to both sides of this equation. After this, the left side becomes zero since we aim at a divergence free vector field for which $\vec{\nabla} \cdot \vec{v} = 0$:

$$0 = \vec{\nabla} \cdot \left(\vec{w}_3 - \frac{\Delta t}{\rho} \vec{\nabla} p \right) = \vec{\nabla} \cdot \vec{w}_3 - \frac{\Delta t}{\rho} \vec{\nabla}^2 p \,.$$

Assuming a regular grid where vector field $\vec{w}_3$ is available, searching the unknown pressure at grid positions, and evaluating the divergence and the Laplacian with finite differences of equations (28.4) and (28.5), respectively, we again end up with a sparse linear system for the discrete pressure values and consequently for the difference between the final velocity field $\vec{v}$ and $\vec{w}_3$. This system is also solved with Jacobi iteration. Similarly to the diffusion step, the Jacobi iteration of the projection is also a simple image filtering operation.

Eulerian simulation on the GPU

The discretized velocity and pressure fields can be conveniently stored in three-dimensional textures, where discrete variables are defined at the centers of elemental cubes, called ***voxels*** of a grid [6]. At each time step, the content of these data sets should be refreshed (Figure 28.9).

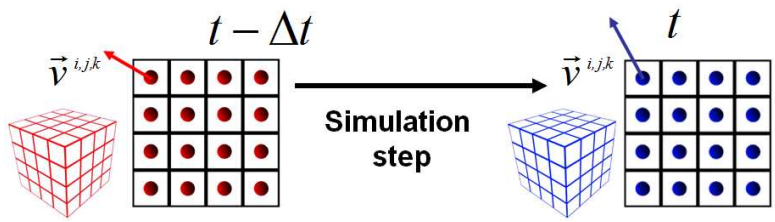

Figure 28.9 A time step of the Eulerian solver updates textures encoding the velocity field.

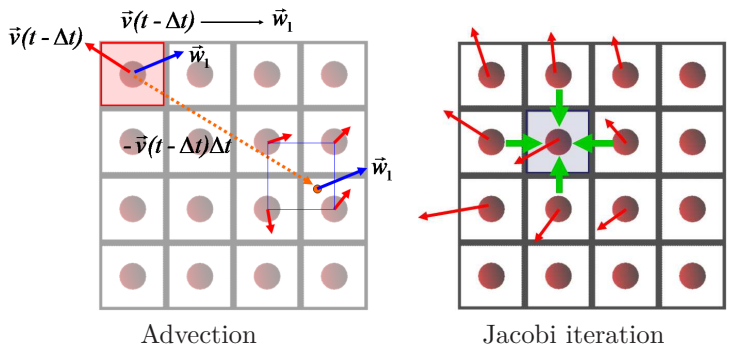

Figure 28.10 Computation of the simulation steps by updating three-dimensional textures. Advection utilizes the texture filtering hardware. The linear equations of the viscosity damping and projection are solved by Jacobi iteration, where a texel (i.e. voxel) is updated with the weighted sum of its neighbors, making a single Jacobi iteration step equivalent to an image filtering operation.

The representation of the fields in textures has an important advantage when the advection is computed. The advected field at voxel center $\vec{r}_i$ is obtained by copying the field value at position $\vec{r}_i - \vec{v}_i \Delta t$. Note that the computed position is not necessarily a voxel center, but it can be between the grid points. According to the finite element concept, this value can be generated from the finite element representation of the data. If we assume piece-wise linear basis functions, then the texture filtering hardware automatically solves this problem for us at no additional computation cost.

The disadvantage of storing vector and scalar fields in three-dimensional textures is that the GPU can only read these textures no matter whether we take the graphics API or the GPGPU approach. The updated field must be written to the render target in case of the graphics API approach, and to the global memory if we use a GPGPU interface. Then, for the next simulation step, the last render target or global memory should be declared as an input texture.

In order to avoid write collisions, we follow a gathering approach and assign threads to each of the grid points storing output values. If GPUs fetch global data via textures, then the new value written by a thread becomes visible when the pass or the thread run is over, and the output is declared as an input texture for the next run.

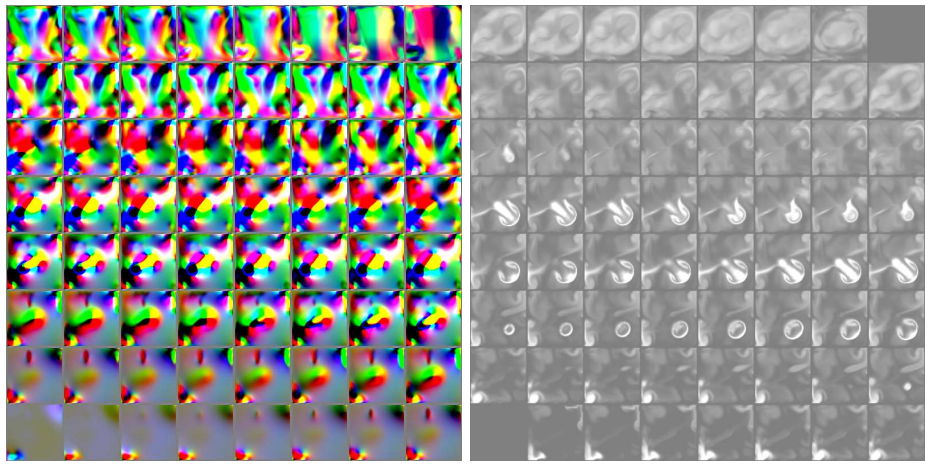

Figure 28.11 Flattened 3D velocity (left) and display variable (right) textures of a simulation.

Thus, the computation of the time step should be decomposed to elemental update steps when the new output value of another grid point is needed. It means that we have and advection pass, a sequence of Jacobi iteration passes of the diffusion step, an external force calculation pass, and another sequence of Jacobi iteration passes of the projection step. With a GPGPU framework, a thread may directly read the data produced by another thread, but then synchronization is needed to make sure that the read value is already valid, so not the old but the new value is fetched. In such cases, synchronization points have the same role and passes or decomposed kernels.

In case of graphics APIs, there is one additional limitation. The render target can only be two-dimensional, thus either we flatten the layers of the three-dimensional voxel array into a large two-dimensional texture, or update just a single layer at a time. Flattened three-dimensional textures are shown by Figure 28.11. Once the textures are set up, one simulation step of the volume can be done by the rendering of a quad covering the flattened grid.

The graphics API approach has not only drawbacks but also an advantage over the GPGPU method, when the linear systems are solved with Jacobi iteration. The graphics API method runs the fragment shader for each grid point to update the solution in the texel associated with the grid point. However, if the neighbor elements of a particular grid point are negligible, we need less iteration steps than in a grid point where the neighbor elements are significant. In a quasi-SIMD machine like the GPU, iterating less in some of the processors is usually a bad idea. However, the exploitation of the ***early z-culling*** hardware helps to sidestep this problem and boosts the performance [20]. The z coordinate in the depth value is set proportionally to the maximum element in the neighborhood and to the iteration count. This way, as the iteration proceeds, the GPU processes less and less number of fragments, and can concentrate on important regions. According to our measurements, this optimization reduces the total simulation time by about 40 %.

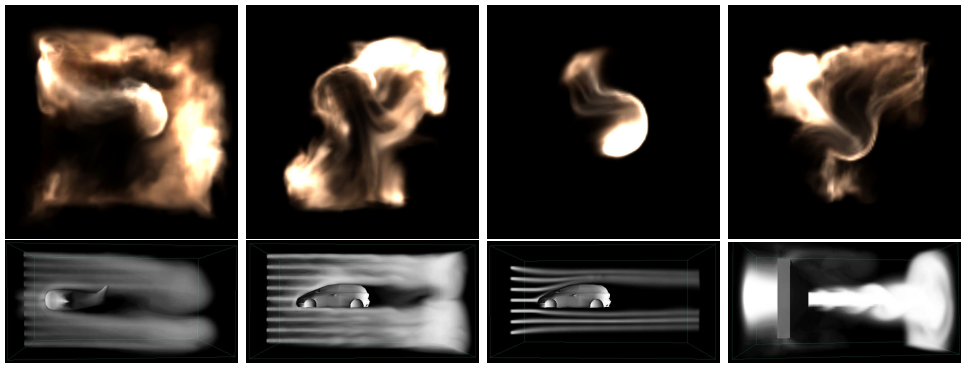

Figure 28.12 Snapshots from an animation rendered with Eulerian fluid dynamics.

When we wish to visualize the flow, we can also assume that the flow carries a scalar *display variable* with itself. The display variable is analogous with some paint or confetti poured into the flow. The display variable is stored in a float voxel array.

Using the advection formula for display variable D, its field can also be updated in parallel with the simulation of time step Δt:

$$D(\vec{r}, t) = D(\vec{r} - \vec{v}\Delta t, t - \Delta t)\,.$$

At a time, the color and opacity of a point can be obtained from the display variable using a user controlled transfer function.

We can use a 3D texture slicing rendering method to display the resulting display variable field, which means that we place semi-transparent polygons perpendicular to the view plane and blend them together in back to front order (Figure 28.12). The color and the opacity of the 3D texture is the function of the 3D display variable field.

28.7.2. Lagrangian solver for differential equations

In the Lagrangian approach, the space is discretized by identifying , i.e. following just finite number of fluid elements. Let us denote the position and the velocity of the ith discrete fluid element by $\vec{r}_i$ and $\vec{v}_i$, respectively. We assume that all particles represent fluid elements of the same mass m, but as the density varies in space and will be the attribute of the particle, every particle is associated with a different volume $\Delta V_i = m/\rho_i$ of the fluid. The momentum conservation equation has the following form in this case:

$$\frac{\mathrm{d}\vec{r}_i}{\mathrm{d}t} = \vec{v}_i\,,$$

$$m\frac{\mathrm{d}\vec{v}_i}{\mathrm{d}t} = \left(-\vec{\nabla}p(\vec{r}_i) + \nu\vec{\nabla}^2\vec{v}(\vec{r}_i) + \vec{F}_{\mathrm{external}}(\vec{r}_i)\right)\Delta V_i\,. \tag{28.8}$$

If particles do not get lost during the simulation, the mass is automatically conserved. However, temporarily this mass may concentrate in smaller parts of the volume, so the simulated fluid is not incompressible. In Lagrangian simulation, we usually

assume compressible gas.

From the knowledge of the system at discrete points, attributes are obtained at an arbitrary point via interpolation. Suppose we know an attribute A at the particle locations, i.e. we have $A_1, \ldots, A_N$. Attribute A is interpolated at location $\vec{r}$ by a weighted sum of contributions from the particles:

$$A(\vec{r}) = \sum_{i=1}^{N} A_i \Delta V_i W(|\vec{r} - \vec{r}_i|)\,,$$

where ΔV_i is the volume represented by the particle in point $\vec{r}_i$, and $W(d)$ is a ***smoothing kernel,*** also called ***radial basis function,*** that depends on distance d between the particle location and the point of interest. From a different point of view, the smoothing kernel expresses how quickly the impact of a particle diminishes farther away. The smoothing kernel is normalized if smoothing preserves the total amount of the attribute value, which is the case if the kernel has unit integral over the whole volumetric domain. An example for the possible kernels is the *spiky kernel* of maximum radius h:

$$W(d) = \frac{15}{\pi h^6}(h-d)^3, \text{ if } 0 \le d \le h \text{ and zero otherwise}\,.$$

For normalized kernels, the particle density at point $\vec{r}_j$ is approximated as:

$$\rho_j = \rho(\vec{r}_j) = \sum_{i=1}^{N} m W(|\vec{r}_j - \vec{r}_i|)\,.$$

As each particle has the same mass m, the volume represented by particle j is

$$\Delta V_j = \frac{m}{\rho_j} = \frac{1}{\sum_{i=1}^{N} W(|\vec{r}_j - \vec{r}_i|)}\,.$$

According to the ***ideal gas law,*** the pressure is inversely proportional to the volume on constant temperature, thus at particle j the pressure is

$$p_j = \frac{k}{\Delta V_j}\,,$$

where constant k depends on the temperature.

The pressure at an arbitrary point $\vec{r}$ is

$$p(\vec{r}) = \sum_{i=1}^{N} p_i \Delta V_i W(|\vec{r} - \vec{r}_i|)\,.$$

The acceleration due to pressure differences requires the computation of the gradient of the pressure field. As spatial variable $\vec{r}$ shows up only in the smoothing kernel, the gradient can be computed by using the gradient of the smoothing kernel:

$$\vec{\nabla} p(\vec{r}) = \sum_{i=1}^{N} p_i \Delta V_i \vec{\nabla} W(|\vec{r} - \vec{r}_i|)\,.$$

Thus, our first guess for the pressure force at particle j is:

$$\vec{F}_{\text{pressure},j} = -\vec{\nabla}p(\vec{r}_j) = -\sum_{i=1}^{N} p_i \Delta V_i \vec{\nabla} W(|\vec{r}_j - \vec{r}_i|)\,.$$

However, there is a problem here. Our approximation scheme could not guarantee to satisfy the physical rules including symmetry of forces and consequently the conservation of momentum. We should make sure that the force on particle i due to particle j is always equal to the force on particle j due to particle i. The symmetric relation can be ensured by modifying the pressure force in the following way:

$$\vec{F}_{\text{pressure},j} = -\sum_{i=1}^{N} \frac{p_i + p_j}{2} \Delta V_i \vec{\nabla} W(|\vec{r}_j - \vec{r}_i|)\,.$$

The viscosity term contains the Laplacian of the vector field, which can be computed by using the Laplacian of the smoothing kernel:

$$\vec{F}_{\text{viscosity},j} = \nu \vec{\nabla}^2 \vec{v} = \nu \sum_{i=1}^{N} \vec{v}_i \Delta V_i \vec{\nabla}^2 W(|\vec{r}_j - \vec{r}_i|)\,.$$

Similarly to the pressure force, a symmetrized version is used instead that makes the forces symmetric:

$$\vec{F}_{\text{viscosity},j} = \nu \sum_{i=1}^{N} (\vec{v}_i - \vec{v}_j) \Delta V_i \vec{\nabla}^2 W(|\vec{r}_j - \vec{r}_i|)\,.$$

External forces can be directly applied to particles. Particle-object collisions are solved by reflecting the velocity component that is perpendicular to the surface.

Having computed all forces, and approximating the time derivatives of (28.8) by finite differences, we may obtain the positions and velocities of each of the particles in the following way:

$$\begin{aligned} \vec{r}_i(t + \Delta t) &= \vec{r}_i(t) + \vec{v}_i(t)\Delta t\,, \\ \vec{v}_i(t + \Delta t) &= \vec{v}_i(t) + (\vec{F}_{\text{pressure},i} + \vec{F}_{\text{viscosity},i} + \vec{F}_{\text{external},i})\Delta V_i \Delta t/m\,. \end{aligned}$$

Note that this is also a forward Euler integration scheme, which has stability problems. Instead of this, we should use a stable version, for example, the ***Verlet integration*** [2].

The Lagrangian approach tracks a finite number of particles where the forces acting on them depend on the locations and actual properties of other particles. Thus, to update a system of N particles, $O(N^2)$ interactions should be examined. Such tasks are generally referred to as the ***N-body problem.***

Lagrangian solver on the GPU

In a GPGPU framework, the particle attributes can be stored in the global memory as a one-dimensional array or can be fetched via one-dimensional textures. In

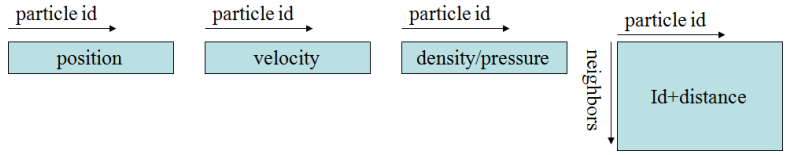

Figure 28.13 Data structures stored in arrays or textures. One-dimensional float3 arrays store the particles' position and velocity. A one-dimensional float2 texture stores the computed density and pressure. Finally, a two-dimensional texture identifies nearby particles for each particle.

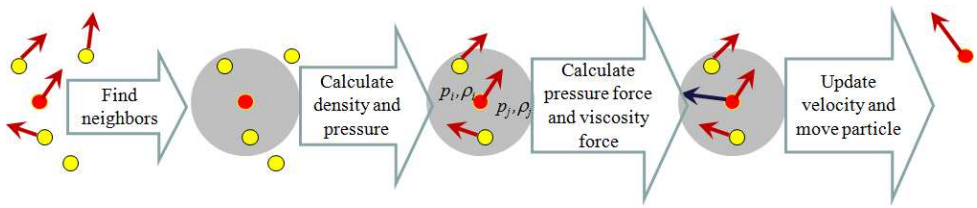

Figure 28.14 A time step of the Lagrangian solver. The considered particle is the red one, and its neighbors are yellow.

graphics API frameworks, particle attributes can only be represented by textures. The advantage of reading the data via textures is only the better caching since now we cannot utilize the texture filtering hardware. A gathering type method would assign a thread to each of the controlled particles, and a thread would compute the effect of other particles on its own particle. As the smoothing kernel has finite support, only those particles can interact with the considered one, which are not farther than the maximum radius of the smoothing filter. It is worth identifying these particles only once, storing them in a two-dimensional texture of in the global memory, and using this information in all subsequent kernels.

A GPGPU approach would need three one-dimensional arrays representing the particle position, velocity, density and pressure, and a two-dimensional array for the neighboring particles (Figure 28.13). In a graphics API approach, these are one- or two-dimensional textures. We can run a kernel or a fragment shader for each of the particles. In a GPGPU solution it poses no problem for the kernel to output a complete column of the neighborhood array, but in the fragment shaders of older GPUs the maximum size of a single fragment is limited. To solve this, we may limit the number of considered neighbor particles to the number that can be outputted with the available multiple render target option.

The processing of a single particle should be decomposed to passes or kernel runs when we would like to use the already updated properties of other particles (Figure 28.14). The first pass is the identification of the neighbors for each particles, i.e. those other particles that are closer than the support of the smoothing kernel. The output of this step is a two-dimensional array where columns are selected by the index of the considered particle and the elements in this column store the index and the distance of those particles that are close by.

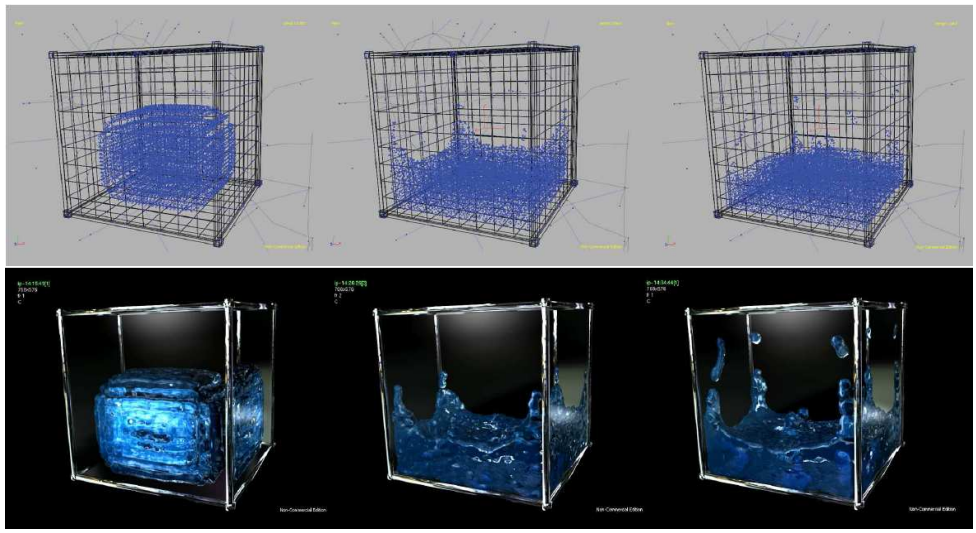

Figure 28.15 Animations obtained with a Lagrangian solver rendering particles with spheres (upper image) and generating the isosurface (lower image) [7].

The second pass calculates the density and the pressure from the number and the distance of the nearby particles. Having finished this pass, the pressure of every particle will be available for all threads. The third pass computes the forces from the pressure and the velocity of nearby particles. Finally, each particle gets its updated velocity and is moved to its new position.

Having obtained the particle positions, the system can be visualized by different methods. For example, we can render a point or a small sphere for each particle (upper image of Figure 28.15). Alternatively, we can splat particles onto the screen, resulting in a rendering style similar to that of the Eulerian solver (Figure 28.12). Finally, we can also find the surface of the fluid and compute reflections and refractions here using the laws of geometric optics (lower image of Figure 28.15). The surface of fluid is the isosurface of the density field, which is the solution of the following implicit equation:

$$\rho(\vec{r}) = \rho_{\text{iso}} \,.$$

This equation can be solved for points visible in the virtual camera by ***ray marching.*** We trace a ray from the eye position through the pixel and make small steps on it. At every sample position $\vec{r}_s$ we check whether the interpolated density $\rho(\vec{r}_s)$ has exceeded the specified isovalue ρ_{iso}. The first step when this happens is the intersection of the ray and the isosurface. The rays are continued from here into the reflection and refraction directions. The computation of these directions also requires the normal vector of the isosurface, which can be calculated as the gradient of the density field.

Exercises

28.7-1 Implement a game-of-life in CUDA. On a two-dimensional grid of cells, every cell is either *populated* of *unpopulated*. In every step, all cell states are re-evaluated. For populated cells:

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For unpopulated cells:

- Each cell with three neighbors becomes populated.

Store cell states in arrays accessible as textures. Always compute the next iteration state into a different output array. Start with a random grid and display results using the graphics API.

28.7-2 Implement a *wave equation solver*. The wave equation is a partial differential equation:

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

where $z(x, y, t)$ is the wave height above point x, y in time t, and c is the speed of the wave.

Chapter Notes

The fixed transformation and multi-texturing hardware of GPUs became programmable vertex and fragment shaders about a decade ago. The high floating point processing performance of GPUs has quickly created the need to use them not only for incremental rendering but for other algorithms as well. The first GPGPU algorithms were also graphics related, e.g. ***ray tracing*** or the simulation of natural phenomena. An excellent review about the early years of GPGPU computing can be found in [15]. Computer graphics researchers have been very enthusiastic to work with the new hardware since its general purpose features allowed them to implement algorithms that are conceptually different from the incremental rendering, including the physically plausible light transport, called global illumination [18], physics simulation of rigid body motion with accurate collision detection, fluid dynamics etc., which made realistic simulation and rendering possible in real-time systems and games. The GPU Gems book series [4, 9, 16] and the ShaderX (currently GPU Pro [3]) series provide a huge collection of such methods.

Since the emergence of GPGPU platforms like CUDA and OpenCL, GPU solutions have showed up in all fields of high performance computing. Online warehouses of papers and programs are the gpgpu.org homepage and the NVIDIA homepage [13, 14], which demonstrate the wide acceptance of this approach in many fields. Without aiming at completeness, successful GPU applications have targeted high performance computing tasks including simulation of all kinds of physics phenomena, differential equations, tomographic reconstruction, computer vision, database searches and compression, linear algebra, signal processing, molecular dynamics and docking, financial informatics, virus detection, finite element methods, Monte Carlo

methods, simulation of computing machines (CNN, neural networks, quantum computers), pattern matching, DNA sequence alignment, cryptography, digital holography, quantum chemistry, etc.

To get a scalable system that is not limited by the memory of a single GPU card, we can build GPU clusters. A single PC can be equipped with four GPUs and the number of interconnected PCs is unlimited [21]. However, in such systems the communication will be the bottleneck since current communication channels cannot compete with the computing power of GPUs.

Bibliography

[1] N. Bell, M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 1–11 pages. ACM, 2009. 1476

[2] D. Eberly. *Game Physics*. Morgan Kaufmann Publishers, 2004. 1486

[3] W. Engel (Ed.). *GPU Pro*. A K Peters, 2010. 1489

[4] F. Fernando (Ed.). *GPU Gems*. Addison-Wesley, 2004. 1489

[5] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *Parallel and Distributed Processing,* IPDPS 2008, 1–8 pages. IEEE, 2008. 1476

[6] M. J. Harris, W. V. Baxter, T. Scheuerman, A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS'03, 92–101 pages, 2003. Eurographics Association. 1481

[7] P. Horváth, D. Illés. Sph-based fluid simulation in distributed environment. In *MIPRO 2009: 32nd International Convention on Information and Communication Technology, Electronics and Microelectronics*, 249–257 pages, 2009. 1488

[8] Khronos. OpenCL overview. 2010. http://www.khronos.org/opencl/. 1452

[9] H. (Ed.). *GPU Gems*. Addison-Wesley, 2008. 1489

[10] M. Magdics, G. Klár. Rule-based geometry synthesis in real-time. In W. Engel (Ed.), *GPU Pro: Advanced Rendering Techniques*, 41–66 pages. A K Peters, 2010. 1456

[11] Microsoft. HLSL. 2010. http://msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx. 1452

[12] J. Neider, T. Davis, W. Mason. *The Official Guide to Learning OpenGL*. Addison-Wesley, 1994. http://fly.cc.fer.hr/~unreal/theredbook/appendixg.html. 1456

[13] NVIDIA. Cg homepage. 2010. http://developer.nvidia.com/page/cg_main.html. 1452, 1489

[14] NVIDIA. CUDA zone. 2010. http://www.nvidia.com/object/cuda_home_new.html. 1452, 1489

[15] J. D. Owens, D. Luebke, N. Govindaraju, M. J. Harris, J. Krüger, A. Lefohn, T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. 1489

[16] M. Pharr (Ed.). *GPU Gems 2*. Addison-Wesley, 2005. 1489

[17] J. Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 121–128 pages, 1999. 1480

[18] L. Szirmay-Kalos, L. Szécsi, M., Sbert. *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, 2008. 1453, 1489

[19] L. Szirmay-Kalos B. Tóth, M. Magdics, D. Légrády, A. Penzov. Gamma photon transport on the GPU for PET. *Lecture Notes on Computer Science*, 5910:433–440, 2010. 1469

[20] N. Tatarchuk, P. Sander, J. L. Mitchell. Early-z culling for efficient GPU-based fluid simulation. In W. Engel (Ed.), *ShaderX 5: Advanced Rendering Techniques*, 553–564 pages. Charles River Media, 2006. 1483

[21] F. Zhe, F. Qiu, A. Kaufman, S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC'04, 47–59 pages, 2004. IEEE Computer Society. 1490

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Subject Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *exe* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Z

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.