12 Simulations

As seen in Chapter 10, "Image Processing," the compute shader can be used to implement new types of algorithms on the GPU in a very intuitive way. However, these new algorithms are certainly not limited to the image processing domain. Many different algorithms lend themselves to being executed on a massively parallel architecture, and this chapter aims to introduce several such applications. In general, as the GPU has continued to become more powerful, there has been a general trend to move more of the workload off of the CPU and onto the GPU. With this trend comes the guideline to minimize the interaction needed between the CPU and GPU during any given frame. These sample programs were designed with this concept in mind—minimal input is required by the CPU to both execute a simulation and to render it.

The first sample algorithm implements a fluid surface simulation technique. In particular, water simulation has become a very important topic in real-time rendering. Providing a fluid simulation that reacts to its environment in a realistic manner is becoming more and more desirable. This sample demonstrates how to use the compute shader to dynamically update the state of a 2D grid of fluid columns. This approximation provides convincing results, while still maintaining a data structure that can take advantage of the new features available in the compute shader.

The second sample algorithm demonstrates one of the oldest effects in computer graphics—a particle system. Even though the concept has been around for a long time, there are some new tricks to be applied to the old problem. This particle system is implemented entirely on the GPU and only interacts with the CPU through parameters provided to the system in constant buffers. This sample simulates how particles would behave around a black hole. While the physics are not calculated to scale, they provide a convincing effect, and demonstrate the general concept of a particle system that can be used as the basis for other types of simulations as well.

These two algorithms demonstrate techniques that use the compute shader to perform calculations that would have been performed on the CPU in older generations of GPU hardware. By leveraging the GPU to perform these simulations, we can directly use the results to render its current state. This reduces the required CPU work, in addition to allowing the massively parallel GPU be used to its fullest potential.

12.1 Water Simulation

Providing believable water rendering in a scene adds a significant sense of realism to its overall appearance. Using a realistic water rendering requires performing a simulation to determine the current state of the system before it can be visualized with its geometry and various optical properties. Water, or any other liquid for that matter, exhibits properties that are very complex, and that are the result of billions of individual molecules interacting with one another and their surrounding environment. Even with the processing power of modern GPUs, performing a true physical simulation of this scale is still out of reach for currently available hardware. However, to add a fluid rendering into a scene does not actually require having access to the state of every molecule in the simulation. Instead, we are only interested in the primary visual properties of the fluid. More specifically, we are interested in the *surface* of the water, since that is what we will end up rendering. Because of this, we can consider the macro-behavior of the fluid, instead of the micro-behavior of the individual elements that make up the simulation body.

This allows us to make a number of simplifications to a potential simulation that can still realistically predict the behavior of the fluid surface, but that can reduce the required computational complexity of the simulation. In the algorithm presented here, we replace the massive number of individual particles with virtual columns of fluid, arranged in a grid-like fashion. Our virtual fluid can move from column to column, but only between physically touching neighbor columns. When considered all together, the height of each column defines the height of the fluid surface at that location in the grid. This property can then be extracted from the simulation and used to generate a number of vertices that will represent the fluid surface. A grid of vertices is precisely the desired result, since it can be used directly in the subsequent rendering of the fluid surface.

By making this model simplification, we can increase or decrease the size of the simulation to allow for either a larger fluid body, or a higher resolution simulation over a smaller physical area. However, the simplification is not without its costs as well. Because of the representation we are choosing to use, it is not possible to include advanced features of a fluid body, such as splashes or breaking waves. However, the resulting behavior of the simulated fluid is still realistic in appearance, and is sufficient for a real-time rendering usage.

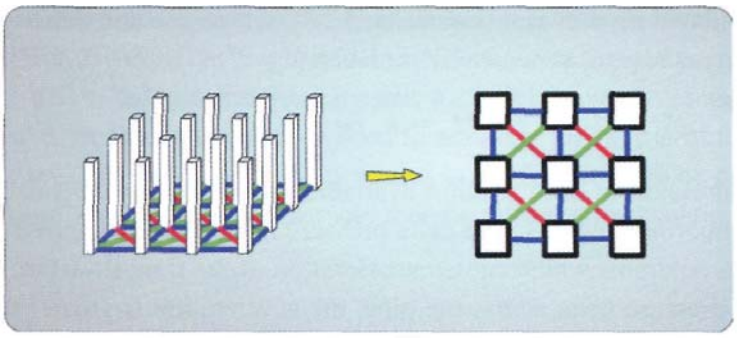

Figure 12.1. The virtual water column and pipe connections between each of them.

12.1.1 Theory

The concept presented in this article is based on a Web chat by Yann Lombard(Lombard), which references a paper entitled "Dynamic Simulation of Splashing Fluids" by James F. O'Brien and Jessica K. Hodgins (O'Brien, 1995). The paper outlines a method of simulating water interactions based on "columns" of water that are connected together by pipes. The columns are organized in a regular grid arrangement, and each column is connected to its immediate neighbors by a virtual pipe. One pipe is used in each of the eight directions outward from each column: up, down, right, left, upper left, upper right, lower right, and lower left. The algorithm is used to approximate the amount of fluid that would be transferred through each of the pipes over a given fixed time step, based on the delta in height values between each column. This water transfer is then used to modify the virtual volumes contained in the columns. This then modifies the height values in each column once again, to be used in the next simulation step. The setup of the columns and pipes is shown in Figure 12.1.

Using this model lets us provide physical parameters for both the fluid being simulated and the physical characteristics of the columns and grids. Each water column is represented as a column with a provided constant cross-sectional area. By using a constant cross-sectional area, we ensure that the volume of fluid contained within the column is directly proportional to the height of the water in the column, as shown in Equation (12.1):

$$h_{ij} = \frac{V_{ij}}{A_{ij}}. \tag{12.1}$$

Once the height of the fluid column is available, we can determine the maximum pressure within the column based on gravity, the density of the fluid, and the atmospheric pressure above the column. The greatest pressure in the column will occur at the bottom of the column, which is where we will assume the pipes are connected at. This maximum

pressure is calculated as shown in Equation (12.2), where p is the density of the fluid, g represents gravity, and p_0 is atmospheric pressure:

$$H_{ij} = h_{ij}\rho g + p_0. \tag{12.2}$$

With the pressure in each column available, we can calculate a delta pressure between two neighboring columns. The delta pressure is then applied across the virtual pipe between the two columns, which causes acceleration of the fluid flow through the pipe. If there is a large pressure delta across the pipe, the acceleration is greater than that from a small delta. This relationship is shown in Equation (12.3), where c is the cross section of the virtual pipe connecting columns and m is the mass of the fluid in the column, which can be calculated from the volume of the fluid and its density:

$$a_{ij\to kl} = \frac{c(H_{ij} - H_{kl})}{m}. \tag{12.3}$$

If we assume that the flow through the pipe is constant over a time interval, the flow through the virtual pipe can be calculated as shown in Equation (12.4), where Q is the flow through the pipe:

$$Q^{t+\Delta t}_{ij\to kl} = Q^{t}_{ij\to kl} + \Delta t(ca_{ij\to kl}). \tag{12.4}$$

This can be used to calculate the total flow through every virtual pipe, which essentially specifies the total change in volume in a fluid column for a given period of time when all pipes for a fluid column are considered together. This is demonstrated in Equation (12.5), which uses an average of the previous and current flow rates over each virtual pipe:

$$\Delta V_{ij} = \Delta t \sum_{kl\in n_{ij}} \left(\frac{Q^{t}_{ij\to kl} + Q^{t+\Delta t}_{ij\to kl}}{2} \right). \tag{12.5}$$

With a change in volume available to us, we can use Equation (12.1) to determine the change in height for a water column. This is ultimately what drives the entire simulation—a difference in fluid heights between neighboring columns causes a fluid flow between them, which ultimately causes a fluid flow between columns, which subsequently alters the volume of the fluid in the columns and their corresponding heights. The process is repeated over and over again to deliver a time varying simulation that provides realistic changes in the surface of the fluid. A sample pair of neighboring columns is provided in Figure 12.2, which visually shows the parameters of our equations.

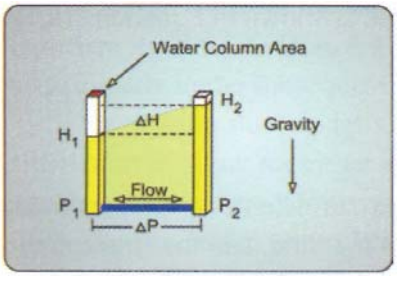

Figure 12.2. A graphical depiction of the important parameters in our simulation.

To accommodate this simulation, we need to maintain two pieces of information for each column of fluid. We need the height of each column, as well as the current flow in each virtual pipe. The height value is the property that we will be using to render the results of our simulation, and the flow values are needed to maintain the inertia of the system from time step to time step. This allows for disturbances in the fluid level to be propagated from one side of the simulation field to another and then be reflected back again.

12.1.2 Implementation Design

With a clear understanding of the theory behind this algorithm, we can explore how to use the tools that are available Direct3D 11 to implement the simulation and render the results. The update phase of the simulation is performed first, and will be executed in the compute shader. This will allow the use of some of the special capabilities of the compute shader, including data sharing and thread synchronization.

Resource Selection

As we have seen in the section on theory, we are interested in maintaining the status of the simulation as a series of height values. Since the simulation is composed of a grid of fluid columns, this height data can naturally be mapped onto a 2D texture resource with a single floating-point component. It is also necessary to maintain a record of the current flow values into and out of each of the water columns. Since there eight different neighbors that can interact with a fluid column, it would seem that we would require two four-component

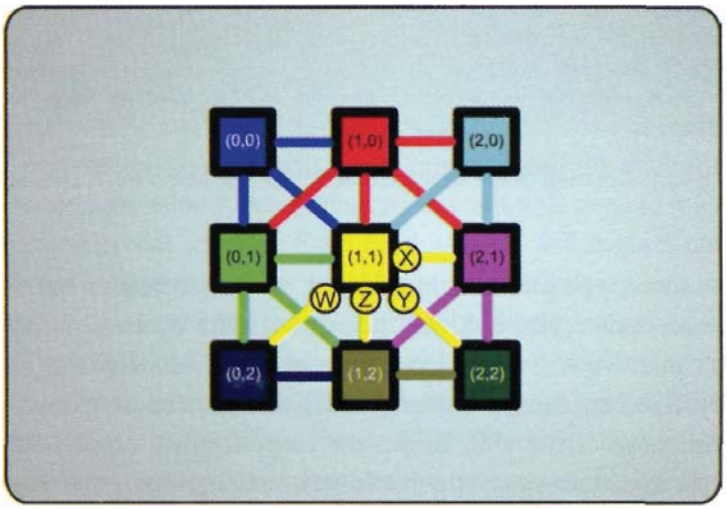

Figure 12.3. The bidirectional flow values managed for each fluid column.

floating-point textures to manage all of the needed data. However, we can take advantage of the fact that the flow into one virtual pipe has an equal and opposite flow, depending on which of the two water columns is using it. This means that we can store only four flow values for every fluid column, and that the remaining four flow values can be taken from the neighboring fluid columns. This concept is depicted in Figure 12.3.

This means that together with height data of the simulation, we need a total of five floating-point variables for each fluid column. Since the maximum number of components per-texel in a texture is four, it would be necessary to use two texture resources together to hold all of the needed data. Instead of doing this, we have chosen to use a structured buffer resource that can provide all five variables together in a single buffer element. This allows us to reference the complete state of a fluid column in a single location, and we can used a simple array syntax to access the data. Unfortunately, structured buffers are always 1D in nature, so we will have to manually convert from our 2D grid coordinates to the appropriate 1D buffer index. In practice this is fairly trivial, since we will know in advance what the total size of the simulation is, so the compromise does not introduce a large amount of additional work. Listing 12.1 demonstrates how the structured buffer is created.

```
// Here size is the number of elements, and structsize is
// the size of the structure to be used.
m_State.ByteWidth = size * structsize;
m_State.BindFlags = D3D11_BIND_SHADER_RES0URCE | D3D11_BIND_UN0RDERED_ACCESS;
m_State.MiscFlags = D3D11_RES0URCE_MISC_BUFFER_STRUCTURED;
m_State.StructureByteStride = structsize;
m_State.Usage = D3D11_USAGE_DEFAULT;
m_State.CPUAccessFlags = 9;

// here pData is assumed to contain the system memory pointer to
// the initial buffer data.
ID3DllBuffer* pBuffer = 8;
HRESULT hr = m_pDevice->CreateBuffer( &m_State, pData, SpBuffer );
```

Listing 12.1. Creation of the structured buffer for use in the water simulation.

With the resource type and format selected, we can consider how the state will be updated during each update phase. Since the current state of the simulation is needed to calculate the new state, we will need to maintain two individual states. These will be kept in two identically sized structured buffers, which are created as shown in Listing 12.1. After each update phase of the simulation, the responsibility of each buffer is reversed from current to new and vice versa, which allows us to ping-pong the states back and forth between the two buffers.

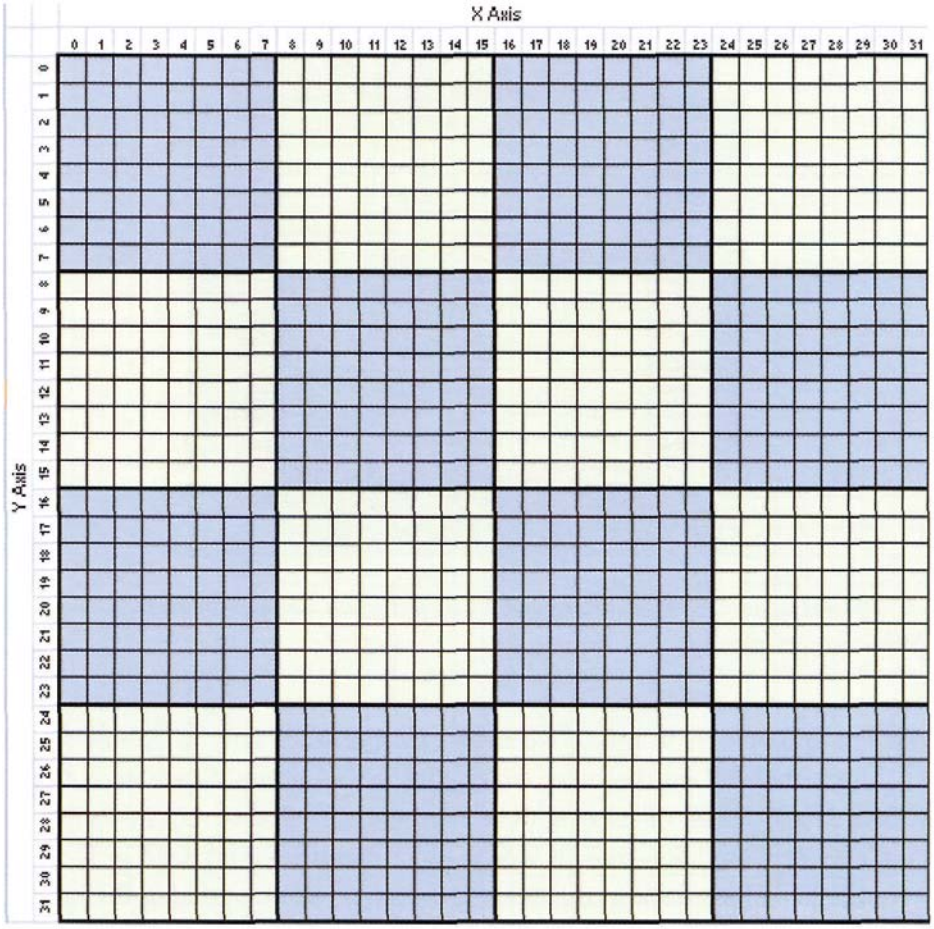

Figure 12.4. The thread group layout of our fluid column grids, depicted as an 8x8 group for visualization.

Threading Scheme

Next, we must determine what type of threading system will be used during the simulation update phase. If we assume that we will use a single thread for each fluid column to be updated, we must consider what types of data accesses it will need to make so that we can choose an appropriate layout scheme. Each fluid column will require data from all of its immediate neighbors to determine how much fluid will be flowing into and out of it. This would indicate that a square-shaped thread group would be appropriate, since it would allow adjacent fluid columns to use the group shared memory for device memory accesses, as well as for intermediate calculations. Thus, the rectangular grid of fluid columns would be split into thread groups as shown in Figure 12.4. We must choose the dispatch size to ensure that enough thread groups are created to update the entire simulation grid. In our sample application, we will be processing a 16^x 16 area of the grid with each thread group, so the total size of the simulation must fit to multiples of this size in each dimension.

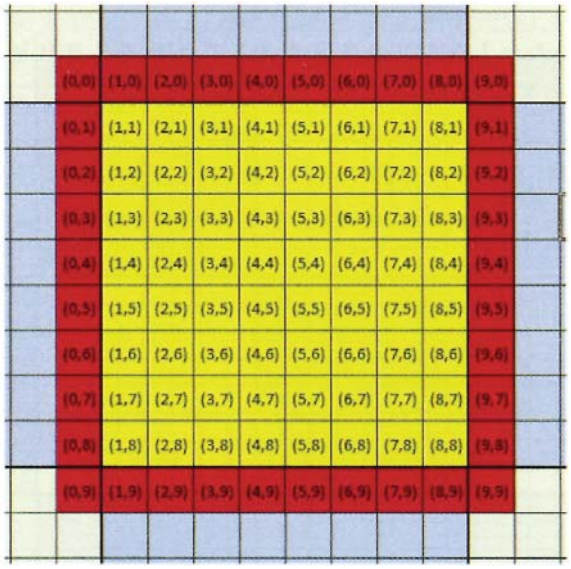

Figure 12.5. The additional threads added around the thread group, depicted with an 8x8 thread group for visualization.

With this orientation in mind, we must also consider how to handle the updating of fluid columns that fall on the outer edges of a thread group. Since each fluid column requires flow data from all of its neighbors, and the data that is required must be calculated in the current simulation pass, it would seem that we would need to write all of the updated flow calculations back to the device resources before updating the height values. This is highly undesirable, since we already have all of the flow and height data loaded to calculate the new flow values; writing them to device memory and then reloading all of that data in a subsequent shader pass essentially doubles the required device memory bandwidth. Fortunately, we can avoid this additional memory accesses by adding some additional threads to the thread group, which can calculate the additional flow values for the fluid columns on the perimeter of the thread group. In this case, we will add an additional row of threads to the complete outer perimeter of the thread group, as shown in Figure 12.5. With the additional threads, each thread group will now use a thread group size of [18,18,1], even though we will only be updating a 16x16 area of the simulation with each thread group.

Each of the threads indicated in Figure 12.5 will behave exactly like the other threads in the group. This means that they will load their height values and calculate the new flow values into and out of themselves. This data can then be stored in the group shared memory, for all threads in the original thread group to use in their final height updates. These perimeter threads will simply not calculate their updated height values, and they will also not update the output resource. The fluid columns represented by these perimeter threads will have their state updated by the adjacent thread group—these flow calculations

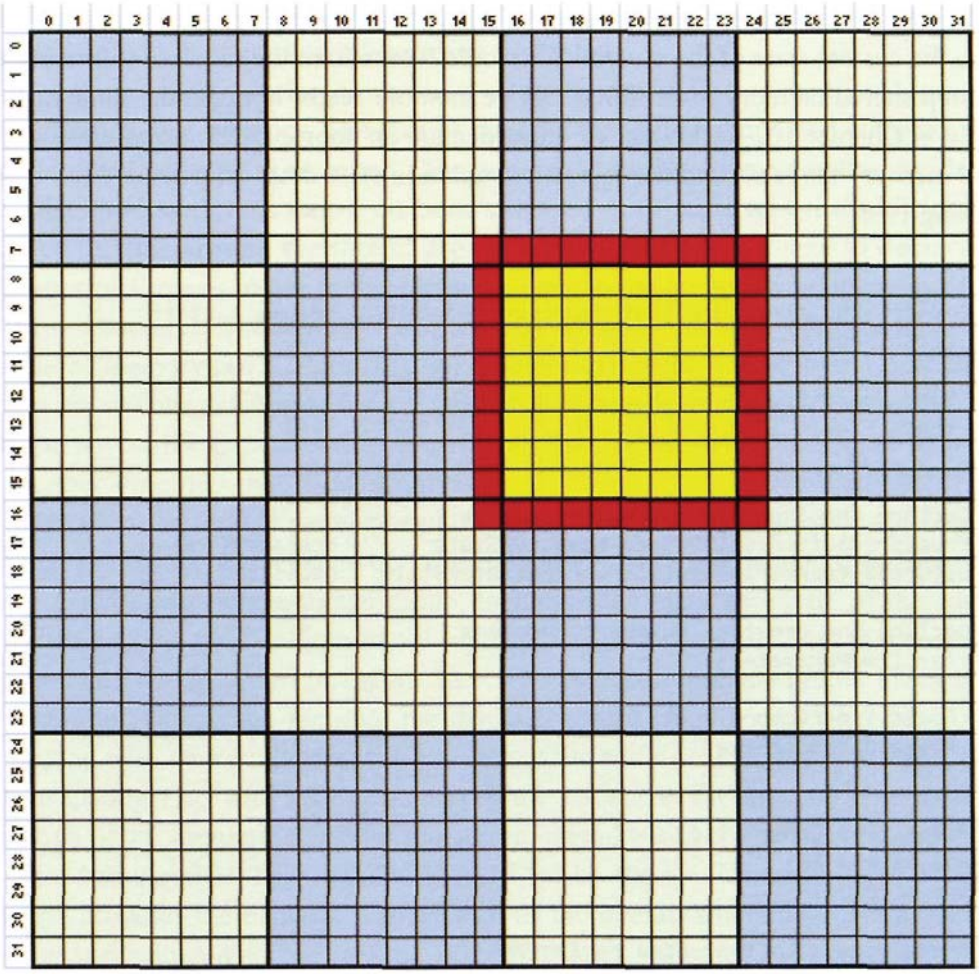

Figure 12.6. Perimeter threads overlapping with adjacent thread groups, depicted with an 8x8 thread group for visualization.

are used strictly to update the current grid of fluid columns, as we identified in Figure 12.4. The concept of using updated thread groups, with the perimeter threads shown in adjacent thread groups, is shown in Figure 12.6.

Fluid Simulation Update Phase

With a general threading model in hand, we can move on to determining the actual work that each individual thread will perform. Each thread will perform four actions. Depending on where a thread is located in the thread group and the overall simulation grid, some of these steps may be reduced or skipped. We will cover these details as they arise in the following discussion.

Accessing the current simulation state. To begin the update process, we must have access to the current state of the system. We would like to load the required information into the group shared memory to minimize device memory reads, in much the same manner as we saw in Chapter 10. To do this, we must declare an appropriately sized array of group shared memory that can hold the relevant simulation state data. This declaration is shown in Listing 12.2.

```
// Declare the structure that represents one fluid column's state
struct GridPoint
{
    float  Height;
    float4 Flow;
};

// Declare the input and output resources
RWStructuredBuffer<GridPoint> NewWaterState     : register( u0 );
StructuredBuffer<GridPoint>   CurrentWaterState : register( t0 );

// Declare the constant buffer parameters
cbuffer TimeParameters
{
    float4 TimeFactors;
    float4 DispatchSize;
};

// Simulation Group Size
#define size_x 16
#define size_y 16

// Group size with the extra perimeter
#define padded_x (1 + size_x + 1)
#define padded_y (1 + size_y + 1)

// Declare enough shared memory for the padded group size
groupshared GridPoint loadedpoints[padded_x * padded_y];

// Declare one thread for each texel of the simulation group. This includes
// one extra texel around the entire perimeter for padding.
[numthreads(padded_x, padded_y, 1)]
```

Listing 12.2. Thread group and group shared memory size declarations.

In this listing, we can see the declaration of the GridPoint structure, whose contents comprise the state required to represent one fluid column. We can also see that the two structured buffer resources are declared with the GridPoint structure as the argument to their template types. This allows us to use the structure members directly on each element of the structured buffer, making the shader code a bit easier to read. Next, we declare the

constant buffer that we will use in this simulation update. These parameters will be discussed individually in the following sections as needed.

With the resources declared, we move on to the declaration of the thread group size and the group shared memory. The size of the area that is being updated with this thread group is defined with the size_x and size_y parameters. The corresponding padded size is then declared with an extra 1 on both sides of it, to represent the additional perimeter of threads around the thread group. We use this padded size to declare both the number of threads to use in this thread group and the group shared memory array of GridPoint structures. The ability to declare a structure as the array element type for the group shared memory is a nice feature, and it demonstrates the flexibility of GSM usage to represent the various data types that are needed. With this layout, we can access the group shared memory with the same array-like syntax that we use with the structured buffers, which further simplifies the shader program.

With all of the needed memory and resources allocated, the algorithm can begin. As mentioned above, the current state of the simulation must be loaded into the group shared memory. Each thread in the thread group, including the extra perimeter of threads, will load the state of the fluid column that it represents. The individual states that correspond to a thread can be visualized in Figure 12.6. The calculation of the proper index to load must take into account the fact that the dispatch thread ID will no longer be a one-to-one mapping with the simulation grid, because of the additional perimeter threads. This is shown in Listing 12.3 with the calculation of the location variable, which provides the 2D location in the simulation grid for the current thread based on the sizes provided in the DispatchSize variable. This variable is provided by the application through a constant buffer and provides the dispatch size as well as the overall simulation grid dimensions. The location is then flattened into the 1D buffer index with the textureindex variable. The data at the textureindex offset is loaded by each thread, then stored directly into the group shared memory for use later in the shader. Once the data is loaded, we perform a memory barrier with a group synchronization, to ensure that the entire thread group has loaded its data into the GSM before proceeding.

```
// Grid size - this matches your dispatch call size!
int gridsize_x = DispatchSize.x;
int gridsize_y = DispatchSize.y;

// The total texture size
int totalsize_x = DispatchSize.z;
int totalsize_y = DispatchSize.w;

// Perform all texture accesses here and load the group shared memory with
// last frame's  height and flow. These parameters are initialized to zero to
// account for  'out of bounds'texels.
```

```
loadedpoints[GroupIndex].Height = 0.0f;
loadedpoints[GroupIndex] .Flow = float4( 0.0f, 8.0f, 0.0f, 0.0f );

// Given your GroupThreadID and the GroupID, calculate the thread's location
// in the buffer.
int3 location = int3( 8, 0, 0 );
location.x = GroupID.x * size_x + ( GroupThreadID.x - 1 );
location.y = GroupID.y * size_y + ( GroupThreadID.y - 1 );
int textureindex = location.x + location.y * totalsize_x;

// Load the data into the GSM
loadedpoints[GroupIndex] = CurrentWaterState[textureindex];

// Synchronize all threads before moving on to ensure everyone has loaded
// their state into the group shared memory.
GroupMemoryBarrierWithGroupSync();
```

Listing 12.3. Loading the current simulation state into the group shared memory.

Calculate the updated flow values. After the current simulation state is available in the GSM, we can start calculating the new state of the simulation. This requires us to determine how much flow is being induced in each of the virtual pipes between the current thread's neighbors. As mentioned earlier, we only need to keep track of four flow values for each thread, and the remaining four flow values can be read from the neighbor fluid column's flow variables. We will choose the convention that each of the four components of the float4 flow variable will indicate the flow to the right, to the lower right, to the bottom, and to the lower left neighbors of the fluid column. This is demonstrated in Figure 12.7, which also shows how the neighboring fluid columns can be used to read the corresponding flow values.

With this orientation in mind, we must calculate the delta in height values between the current fluid column and its neighbors. To do this, we initialize a NewFlow variable to zero flow. Then we calculate the appropriate delta value for each component of the variable, but only if the current thread isn't at the edge of the simulation. It is very important to ensure that we don't calculate a delta value with a non-existent fluid column, because we will either access an incorrect fluid column (due to the 1D buffer storage) or access an out-of-range index. The former will probably introduce erroneous delta values at strange locations in the simulation, and the latter will cause a delta value to be created against a zero

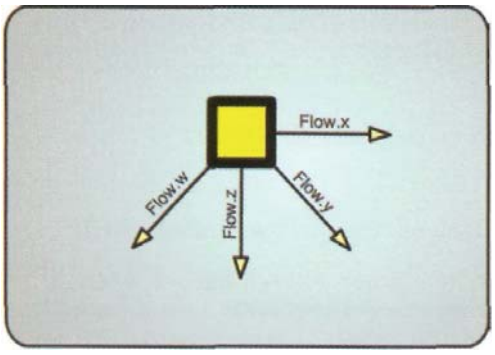

Figure 12.7. The flow values, and which virtual pipes they represent.

value that is returned in out-of-range accesses. In either case, this is undesirable and should be avoided. The process for calculating these delta values is shown in Listing 12.4.

```
// Initialize the flow variable to the last frames flow values
float4 NewFlow = float4( 0.0f, 0.0f, 0.0f, 0.0f );

// Check for 'not' right edge
if ( ( GroupThreadID.x < padded_x - 1 ) && ( location.x < totalsize_x - 1 ) )
{
    NewFlow.x = ( loadedpoints[GroupIndex+1].Height
                - loadedpoints[GroupIndex].Height );

    // Check for 'not' bottom edge
    if ( ( GroupThreadID.y < padded_y - 1 ) && ( location.y < totalsize_y - 1 ) )
    {
        NewFlow.y = ( loadedpoints[(GroupIndex+l) + padded_x].Height
                    - loadedpoints[GnoupIndex].Height );
    }
}

// Check for 'not' bottom edge
if ( ( GroupThreadID.y < padded_y - 1 ) && ( location.y < totalsize_y - 1 ) )
{
    NewFlow.z = loadedpoints[GroupIndex+padded_x].Height
              - loadedpointsfGroupIndex].Height;

    // Check for 'not' left edge
    if ( ( GroupThreadlD.x > 0 ) && ( location.x > 0 ) )
    {
        NewFlow.w = ( loadedpoints[GroupIndex + padded_x - 1].Height
                    - loadedpoints[GroupIndex].Height );
    }
}
```

Listing 12.4. Calculating the height delta between fluid columns.

The delta height values can then be combined, using a number of constants to represent the physical properties of the simulation. These constants are declared as literal constants in the shader, but they could just as easily be loaded into the shader through a constant buffer if they will change over the lifetime of the application. The calculation of the new flow values is shown in Listing 12.5, where the NewFlow variable is updated to hold the new flow values, instead of the delta heights. The new flow value is a combination of the existing flow in the virtual pipe from the previous simulation step and the additional flow induced by the delta in height values, and is also based on the size of the time step that has passed since the last update. The application provides the elapsed frame time in the TimeFactors. x parameter, and a minimum selection is made between the actual time

step and a maximum step size constant. This allows the simulation to be applied over reasonable time steps, even if the reference device is being used to perform the calculations. Finally, a damping factor is used to gradually reduce the amount of flowing inertia in the system. This will eventually allow the system to return to an equilibrium state after an appropriate period of time.

The recalculated flow values are then written back to the group shared memory, so they may be used by all of the threads in the simulation to update their current height values. An important consideration here is that the perimeter threads still calculate their updated flow values, but only write them into the group shared memory. These flow values will eventually be discarded, but they will be needed in the next phase of the implementation to produce the correct height values for the fluid columns that they are in contact with. After the group shared memory is written to, we perform another memory barrier to synchronize the thread group's execution.

```
const float TIME_STEP = 0.05f;
const float PIPE_AREA = 0.0001f;
const float GRAVITATION = 10.0f;
const float PIPE_LENGTH = 0.2f;
const float FLUID_DENSITY = 1.0f;
const float COLUMN_AREA = 0.05f;
const float DAMPING_FACTOR = 0.9995f;

float fAccelFactor = ( min( TimeFactors.x, TIME_STEP ) * PIPE_AREA * GRAVITATION )
                        / ( PIPE_LENGTH * COLUMN_AREA );

// Calculate the new flow, and add in the previous flow value as well. The
// damping factor degrades the amount of inertia in the system.
NewFlow = ( NewFlow * fAccelFactor + loadedpoints[GroupIndex].Flow )
                    * DAMPING_FACTOR;

// Store the updated flow value in the group shared memory for other threads
// to access
loadedpoints[GroupIndex].Flow = NewFlow;

// Synchronize all threads before moving on
GroupMemoryBarrierWithGroupSync();
```

Listing 12.5. Calculating the new flow values based on the delta heights in the current simulation step, as well as the previous step's flow values.

This is an appropriate time to consider the memory accesses that were required to perform the new flow calculations. All of the height values and the existing flow values are read from the GSM, meaning that this phase of the calculation does not touch the device memory at all. In addition, since we synchronized all of the threads in the thread group after filling the GSM, we effectively eliminated the need for further low-level synchronization

during the calculation of the flow values. This makes an efficient use of the GSM and balances out memory barrier synchronization with a fair amount of arithmetic, making it worthwhile to use it.

Calculate the updated height values. With all of the flow values available in the GSM, we can trivially read the current height values and apply the flows out of the current fluid column. We can also read the flow values from the neighboring fluid columns and negate their value to apply the proper magnitude flow into those columns. Since we are only reading from data that was already synchronized in the GSM in the previous step, we can read all of these data values without any further synchronization. This process is shown in Listing 12.6. In contrast with the thread masking that was required earlier, there is no need to perform the conditional instructions in this case. This is because any flow values to out-of-bounds fluid columns will have a zero value. This is in fact why we masked the threads off earlier in the process.

```
// Calculate the new height for each column, then store the height and
modified flow values.
// The updated height values are now stored in the loadedheights shared memory.

// Out of the current column...
loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height + NewFlow.x
                                + NewFlow.y + NewFlow.z + NewFlow.w;

// From left columns
loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height

-loadedpoints[GroupIndex-1].Flow.xj;// From upper left columns

loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height
                                - loadedpoints[GroupIndex-padded_x-1].Flow.y;

// From top columns
loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height
                                - loadedpoints[GroupIndex-padded_x]. Flow.z;

    // From top right columns
loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height
                                - loadedpoints[GroupIndex-padded_x+1].Flow.w;
```

Listing 12.6. Calculating the new height values for this simulation step.

Storing the simulation state. The final process in updating the simulation state is to update the output structured buffer resource with the newly calculated height and flow values, so they can be used in the next simulation step. This is done through an unordered access view, which is the only means of output for the compute shader. The writing of the values

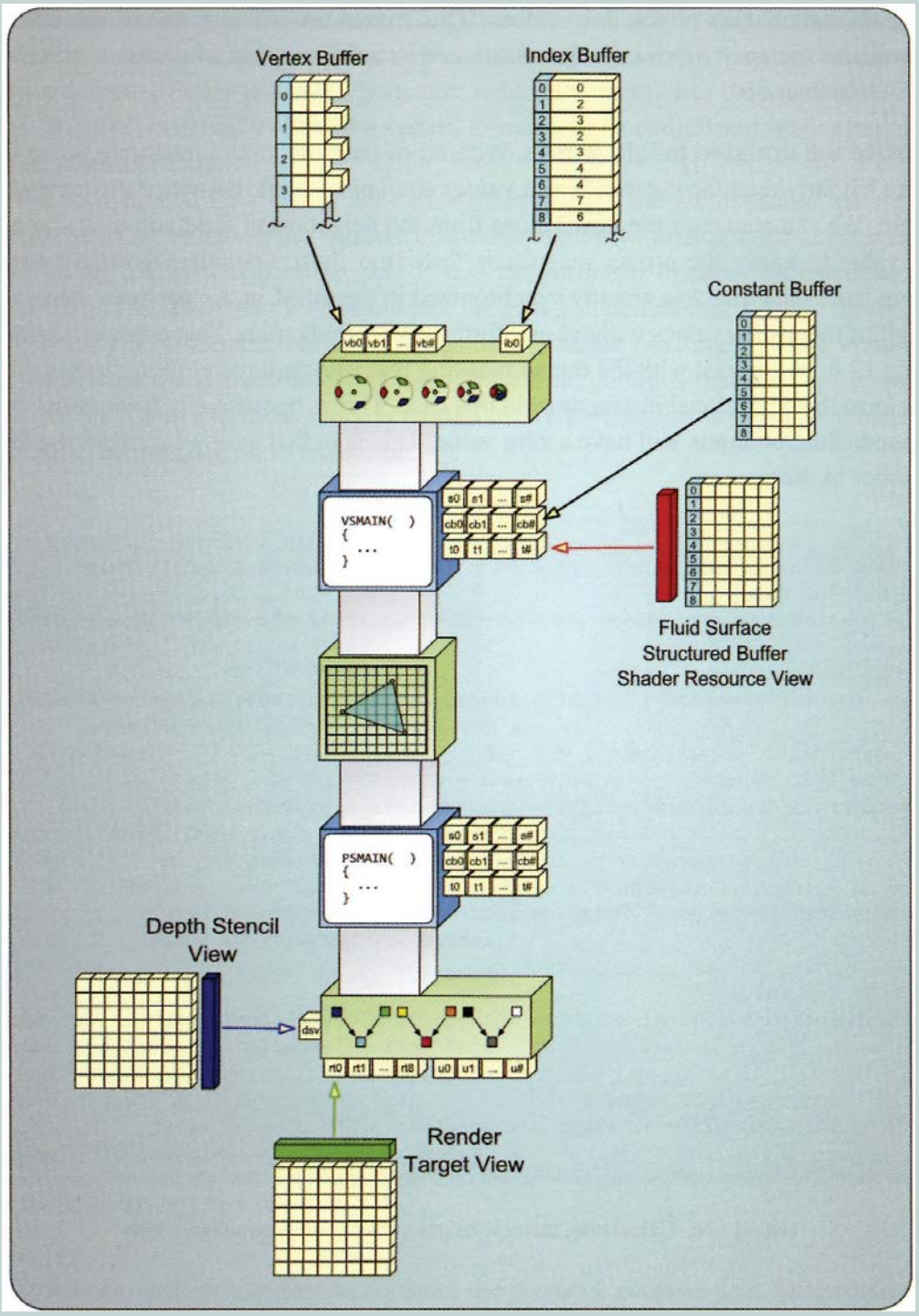

Figure 12.8. The rendering pipeline layout used for visualizing the fluid surface.

to the output resource is shown in Listing 12.7. This final step must mask off the perimeter threads to ensure that they don't modify the contents of the output resource. In theory, all of these values would be overwritten by the other thread groups being executed, but depending on the order that the thread groups are processed in it may lead to invalid data being written structured buffer. The writing of the output data is the first and only access that is performed through the unordered access view. This is a good property to replicate in other algorithms, as it indicates that the device memory has a minimum number of accesses.

```
// Finally store the updated height and flow values only for the threads
// within the actual group (i.e. excluding the perimeter). Otherwise there would
// be a double calculation for each perimeter texel., making the simulation
// inaccurate!
if ( ( GroupThreadID.x > 0 ) S8 ( GroupThreadID.x < padded_x - 1 )
        && ( GroupThreadID.y > 0 ) && ( GroupThreadID.y < padded_y - 1 ) )
{
    NewWaterState[textureindex] = loadedpoints[GroupIndex];
}
```

Listing 12.7. Storing the results of the simulation to the output structured buffer.

Rendering the Fluid Simulation

After the simulation state is updated, the surface of the fluid can be rendered. However, by this point all of the hard work has already been completed and we simply need to read the height values in the structured buffer that contains the current state of the simulation. The pipeline layout is shown in Figure 12.8.

We perform this height lookup in the vertex shader using the same calculation that was used in the compute shader to find the location within the structured buffer to load. The difference in this case is that we use input vertex attributes to determine which point in the 2D simulation grid each vertex should represent. To demonstrate the boundaries of each thread group that is used in the simulation, we color the vertices with one of

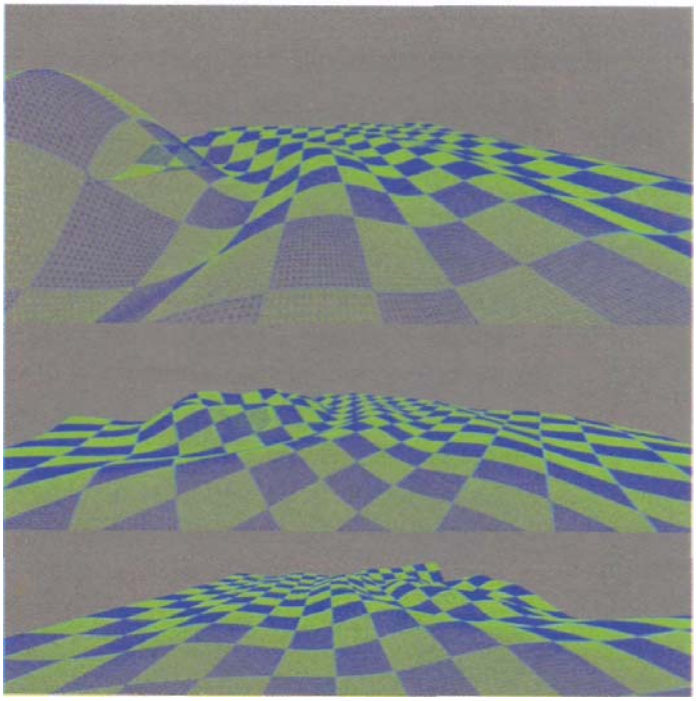

Figure 12.9. The final result of the rendered fluid surface.

two colors, which are alternated for each successive thread group. This produces a checkerboard pattern that can be seen in the final rendered output, as shown in Figure 12.9.

12.1.3 Conclusion

This fluid simulation provides an efficient method to represent realistic, time-varying fluid surfaces. The simulation is run completely on the GPU, freeing up the CPU to perform other tasks. Since the simulation grid size is not fixed, it can be expanded to any desired size, as long as it fits within a structured buffer resource. This allows for potentially huge simulation spaces, or extremely high resolution simulations of smaller spaces. In addition, since it leverages a highly parallel algorithm, the performance of the technique will scale well with future hardware performance improvements.

12.2 Particle Systems

Fluid rendering is a very useful and visually appealing simulation to add to a scene. However, its use is restricted to a fairly specific domain of situations due to the fact that it represents a fluid. There is a whole class of additional natural and man-made phenomena that are desirable to add to a scene, and that are fundamentally different than fluids. These effects include smoke, fire, sparks, or debris from an explosion, just to name a few. As with fluid simulation, the visual appearance of these phenomena is the result of many millions or billions of individual molecules continually interacting with one another in many different ways, over a period of time. This is simply not possible to directly simulate in a real-time rendering context; hence, we must find a more efficient method to produce a rendered image sequence that can approximately produce a similar appearance.

One potential technique for implementing these effects is the use of particle systems. A particle system is a construct composed of many individual elements, referred to as particles. Each particle has a unique set of variables associated with it to define its current status; when considered at the same time, the particles form a particle system. The simulation portion of this concept requires us to define an algorithm for creating new particles, destroying old particles, and incrementally updating the state of the particles between creation and destruction events for a small increment of time. Each particle is created, then is updated once in each simulation step, and finally destroyed after it is no longer needed. The main constraint in a particle system is that all particles must use the same set of variables, which allows their update method to operate in the same way on each particle.

This technique of using many unique elements and updating them with the same set of rules is used to derive complexity out of many simple components. For example,

to simulate the smoke rising from a fire, each particle represents one puff of smoke. Each particle maintains a position, a velocity, and perhaps a rotation and scale, all of which are updated every time step of the simulation. Here, the update method generally makes each particle drift in some direction, with a semi-perturbed pathway. After each simulation step, the particle system is rendered with each particle being represented as a small quad with a smoke-puff texture applied to it. With all of the particles moving independently, but with the same rules applied to them, we can model a more complex system with a simple one. The same concept applies to the other examples mentioned above, except that perhaps the particle properties, the rendering attributes, and the update method would be changed to the appropriate version for that type of particle system.

Particle systems have been used in computer graphics for a very long time. In the past, they were implemented on the CPU and then simply rendered after each update. However, with the parallel nature of the GPU and the very parallel nature of the updating mechanism, the GPU is an ideal processor to perform the simulation. In addition, since the simulation results are already residing in video memory, the rendering process doesn't need to transfer it out of system memory, which results in faster rendering. The sample program discussed in this section implements such a GPU-based particle system, while taking advantage of some of the new features in Direct3D 11. This particle system provides a specific implementation of a particular type of system, but it can easily be adapted to support other particle systems as well.

12.2.1 Theory

The particle system we will build in this section represents a particle emitter and consumer, where the emitter creates particles and the consumer destroys them when they get too close to it. The system will be governed by a simple gravity system based around the consumer. This could be thought of as a simplified black hole, which exhibits a large gravitational pull on each particle and will swallow them up once they pass the event horizon. Before moving to the implementation of the particle system, we will examine the basic laws of gravitation in order to implement our particle update method in a physically plausible manner.

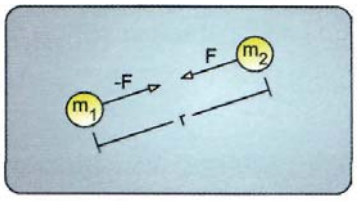

Figure 12.10. Two point masses attracting one another, based on Newton's law of universal gravitation.

Newton's law of universal gravitation states that there is an attractive gravitational force between two point masses, which is proportional to the product of the two masses, and inversely proportional to the square of the distance between them. This relationship is depicted in Figure 12.10, and is defined in Equation (12.6), where F is the gravitational

force, G is the gravitational constant, m_1 and m_2 are the masses of the two objects, and r is the distance between their center of mass:

$$F = G\frac{m_1 m_2}{r^2}. \tag{12.6}$$

According to this relationship, we can see that the closer two objects are to each other, the greater the attractive gravitational force between them becomes. In addition, the more massive the objects are the greater this force is as well. This makes sense when considering a black hole, which represents a singularity of infinitely large mass (this is a simplification, but it will serve as an adequate explanation for our purposes). If an object approaches a black hole, at some point it will be too close to the black hole and will be dragged into it. The acceleration of the particle caused by the gravitational force is calculated with Newton's second law, which is shown in Equation (12.7). The acceleration can be found by dividing the gravitational force by the mass of the object that it is acting on:

$$F = ma. \tag{12.7}$$

In the context of our simulation, the black hole will represent a very large mass instead of an infinite mass, due to the obvious calculation issues with using infinite numbers. Each particle will have a fixed mass and will be subjected to the gravitational pull of the black hole. In addition to the gravitational effects on the particles, they will be created with a randomized initial velocity as they are emitted from the particle emitter. This will let the user see where the particle emissions are coming from, in addition to where they are being attracted to. To calculate the particle velocity after each simulation step, we will use Equation (12.8) where v_0 is the initial velocity at the beginning of the time step, a is the acceleration caused by the gravitational force, and t is the amount of time that has passed in this time step:

$$v = v_0 + at. \tag{12.8}$$

After the new velocity of the particle has been determined, we can determine the modified position of the particle over the current time step. This is performed as shown in Equation (12.9). With these basic physical interactions clarified, we can continue to the implementation design that will use the GPU to efficiently simulate how these bodies will interact:

$$p = p_0 + vt. \tag{12.9}$$

12.2.2 Implementation Design

The concept of a particle system is well known, but implementing one with the help of the new features of Direct3D 11 is not. In this section we will explore one possible

implementation and discuss the features that are used. Our particle system will be simulated in the compute shader, and then rendered using the rendering pipeline in a similar fashion as we have seen in the fluid simulation sample. We will begin with a discussion of what type of resources to use, followed by a description of how we will add particles into the system. We will then clarify how the particle update phase is performed, and finally will discuss the rendering technique used to present the results of the simulation.

Resource Selection

To select an appropriate resource type, we once again must consider the type of data we will be storing in it. We would like to store a list of all of the particles that exist within the particle system. Since the particles can be stored in a list, this implies a ID storage type. In addition, the data that will be stored for each particle will consist of at least a position and a velocity, plus whatever additional specialized data is needed for a particular type of particle system. Since there are more than four individual data items to store (the position and velocity are both vectors), it is not possible to use a single 1D texture to hold the data. Instead, we will use a structured buffer to hold our particles. This will allow the creation of a customized structure for our particle data and will provide a very easy method of accessing the data.

The nature of a particle system allows for the use of one of the special variants of a structured buffer. We have seen in Chapter 2 that a specially made unordered access view allows a structured buffer to be used as an AppendStructuredBuffer or a ConsumeStructuredBuffer from within HLSL. The append and consume buffers allow for a simple storage mechanism, in which the order of the elements is not required to be preserved, which in turn allows for some optimizations to be implemented in the GPU when reading and writing to and from the resource. Our particle system is comprised of a varying number of particles, and there is no reason to be concerned with the order that they appear within the buffer since they are all treated in exactly the same

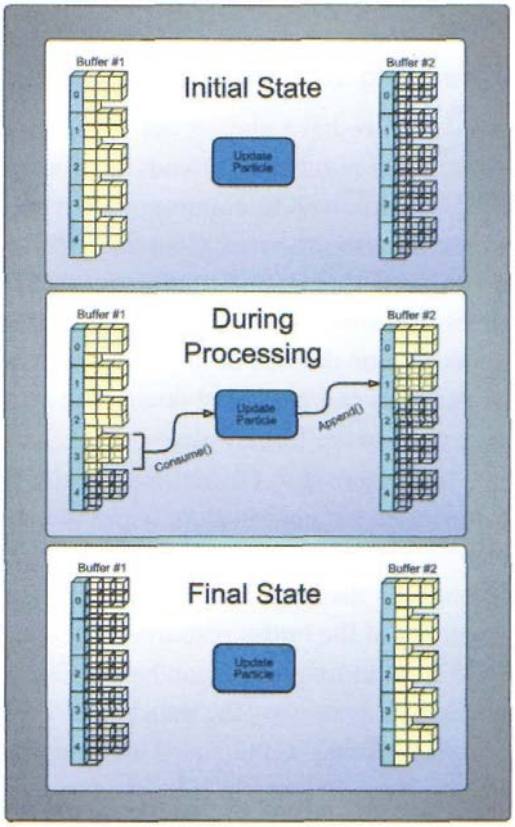

Figure 12.11. Two structured buffer resources used as append and consume buffers to manage particle data.

manner. In other words, they all are updated in the same fashion, so there is no need to know precisely which particle is being processed by a particular thread. This is a fundamental difference between the fluid simulation and the particle system—the fluid simulation columns all rely on information from their neighbors, which requires that each column know its exact location within the simulation. The unordered nature of the particle system makes it particularly well suited to using append and consume buffers to hold them.

With this in mind, we will use two structured buffer resources to contain our particle system. One will hold the current particle data, and the other will receive the updated particle data after an update sequence is performed. Figure 12.11 depicts how these resources are used to provide and then receive the particle data. After an update pass is performed, the buffer that received the updated data will then become the current state of the simulation, and the process can be repeated in the next simulation step to refill the other buffer. Before a simulation update is performed, the current state buffer holds all of the particle data while the other buffer is empty. During processing, the elements are consumed from the current state buffer, and then appended to the new state buffer. After processing, the roles are reversed, and the new state buffer holds all of the particle data.

Threading Scheme

Now that we have chosen our resource model, we must consider how we will invoke an appropriate number of threads to process these data sets. As we described above, the particles do not need to communicate with one another at all, which means there is no need to use the group shared memory. This lets us choose a threading orientation based solely on ensuring that an appropriate number of threads are instantiated to process all of the particles that are currently active. The upper limit to the number of particles in the system is dependent on the size of the buffer resources that are created to hold their data. The number of particles can also vary downward to a smaller number if particles have died off or are destroyed in the black hole.

This seems to present a problem. The number of particles can change from simulation step to simulation step, while the number of active particles is available in the buffer resource. However, the CPU is in control of specifying how many threads to instantiate through its dispatch methods. There is a device context method for reading the element count out of the buffer resource, but this count is copied into another buffer resource. If the CPU tried to map the secondary buffer into system memory, it would require quite a significant delay to copy the data back to the CPU for reading. That would negate the benefits of having the fast GPU-based implementation, since we would be synchronizing data to the CPU in every frame. So instead of trying to read back the number of particles, we can take a more conservative approach with our thread invocation. We can estimate the number of particles in the system, based on the properties of the creation and destruction mechanisms that are used, and then round up to the next higher number of threads specified by our

thread group size. For example, if we choose the maximum size thread group of [1024,1,1], we would round our estimated number of particles up to the next multiple of 1024.

But what happens when the additional threads from rounding up try to read from the consume buffer and no particle data is available? Trying to consume more data than is present produces an undefined behavior, so we must ensure that this does not happen. We can manage this by copying the element count to a constant buffer with the CopyStructureCount method, and then masking off the particle processing code with a conditional statement to ensure that the appropriate number of elements is processed. The threading setup is shown in Listing 12.8, along with the required resource declarations.

```
struct Particle
{
    float3 position;
    float3 velocity;
    float  time;
};

AppendStructuredBuffer<Particle>  NewSimulationState : register( u0 );
ConsumeStructuredBuffer<Particle> CurrentSimulationState  : register( u1 );

cbuffer SimulationParameters
{
    float4 TimeFactors;
    float4 EmitterLocation;
    float4 ConsumerLocation;
};

cbuffer ParticleCount
{
    uint4 NumParticles;
};

[numthreads( 512, 1, 1)]

void CSMAIN( uint3 DispatchThreadID : SV_DispatchThneadID )
{
    // Check for if this thread should run or not.
    uint myID = DispatchThreadID.x
              + DispatchThreadID.y * 512
              + DispatchThreadID.z * 512 * 512;

    if ( myID < NumParticles.x )
    {
        // Perform update process here...
    }
}
```

Listing 12.8. The threading setup used to update all of the particles in the system.

In this listing, we see the particle structure defined as a position, velocity, and a scalar time value. The particle structure is then used to declare the append and consume buffers. Next is the declaration of the constant buffers that are used, which consist of the simulation parameters and the number of particles will be stored in the PanticleCount buffer. After these resources are specified, we declare the thread group size of [512,1,1], which means that our granularity for instantiating threads is in 512-thread increments. In the body of the shader, we determine the global dispatch thread ID of the current thread by flattening the 3D dispatch thread ID. It is then compared to the number of particles in the consume buffer, which was copied into the constant buffer with the CopyStructureCount method.

This arrangement has some additional benefits besides the unordered access optimizations. Since the append and consume buffers manage the counting of the total number of elements that are present in the buffers, we can customize our processing to consider only those buffer elements that are currently active. This decreases the application's sensitivity to knowing how many particles are in existence, as long as the maximum number of elements is not exceeded. With this type of processing, we effectively decouple the CPU from the GPU for this processing action and allow the GPU to perform as quickly as it is capable of, without requiring synchronization with the CPU.

Particle Creation

Now that we know how we will store our particles, and how we will approach them from a threading perspective, we can begin looking into how to add particles into the simulation. There are two different methods used in this sample. The first possibility is that we can initialize the system with a particular number of particles at startup. The second possibility is to add particles into the simulation during runtime. We will look deeper into these two options in the following sections.

System initialization. When we begin the simulation, we have the option to begin with a given number of particles from the very first simulation step. This can be accomplished in two steps. First we must initialize the contents of the structured buffers to hold our desired data. As we have seen in Chapter 2, we can provide an initial set of data for a resource in its creation function. For use in a particle system, this would entail filling in the initial data with an array of particles. If we don't want to initialize the entire contents of the particle system, we must still provide a complete set of initial data to initialize the resource. The second step to providing the initial set of particles is to tell the unordered access view how many particles should be considered to exist within the structured buffer. This is done through setting the initial counts parameter in the ID3D11DeviceContext::CSSetUnordered AccessViews() method.

Specification of how many elements exist in the buffer is only needed during the first use of the buffers. Once the first update phase has been completed, all of the active particles will have been updated and appended to the output buffer through the compute shader

program. Since the writing of the data uses the append method, the count is automatically incremented for each particle added to the output buffer. Likewise, the buffer that provides the initial state of the simulation will have its count automatically decremented for each consume operation performed on it, and should end up with a count of zero after the first update phase is performed. Thus, the buffer that contains the initial particle data should get an initial count that indicates the initial number of particles in the system, and the second structured buffer will get an initial count of zero only for the first binding sequence. For each update phase after the first one, both buffers should be bound with an initial count of -1 to indicate that the internal count value should be used.

Dynamic injection of particles. Since we want to have a dynamic particle system, it is necessary to be able to add new particles into the simulation during runtime. However, after the initial resource creation, the CPU cannot directly modify the contents of the buffer (since it uses a default usage). Even so, the CPU is still able to perform particle injection indirectly, by running a separate compute shader program in which each thread that is invoked will create a new particle and add it to the output buffer resource. In this way, we can allow the CPU to control the emission of new particles, and can still use the parallel nature of the GPU to efficiently implement the desired number of new particles in batches.

For our sample program, we want to produce particles that emanate from an emitter location, and to have them initialized with a randomized velocity direction. To do this, we will create eight particles at a time for each thread group that is dispatched. We will use a static array of direction vectors which point to the eight corners of a unit sized cube, and then reflect the vectors about a random vector provided by the application. Then, each of the eight threads in the thread group will read one of the static vectors, reflect it, and use the resulting vector to specify the initial velocity vector for our system. The intended use of this compute shader program is to dispatch a single thread group at a time, with a single randomized vector supplied in a constant buffer to aid in the initialization of the particles. Tf more thread groups are needed at the same time, a randomized vector could be read out of a resource for each thread group by using the SVJSroupID system value semantic. This compute shader to insert particles is shown in Listing 12.9.

```
struct Particle
{
    float3 position;
    float3 direction;
    float time;
};

AppendStructuredBuffer<Particle> NewSimulationState : register(u0 );

cbuffer ParticleParameters
{
    float4 EmitterLocation;
```

```
    float4  RandomVector;
};

static const float3 direction[8] =
{
    normalize( float3( 1.0f,  1.0f,  1.0f ) ),
    normalized float3( -1.0f,  1.0f,  1.0f ) ),
    normalized float3( -1.0f, -1.0f,  1.0f ) ),
    normalized float3( 1.0f, -1.0f,  1.0f ) ),
    normalize( float3(  1.0f,  1.0f, -1.0f ) ),
    normalize( float3( -1.0f,  1.0f, -1.0f ) ),
    normalize( float3( -1.0f, -1.0f, -1.0f ) ),
    normalize( float3( 1.0f, -1.0f, -1.0f ) )
};


[numthreads( 8, 1, 1)]

void CSMAIN( uint3 GroupThreadID : SV_GroupThreadID )
{
    Particle p;

    // Initialize position to the current emitter location
    p.position = EmitterLocation.xyz;

    // Initialize direction to a randomly reflected vector
    p.direction = reflect( direction[GroupThreadID.x], RandomVector.xyz ) * 5.ef;

    // Initialize the lifetime of the particle in seconds
    p.time = 0.0f;

    // Append the new particle to the output buffer
    NewSimulationState.Append( p );
}
```

Listing 12.9. The compute shader for dynamically adding particles into the system.

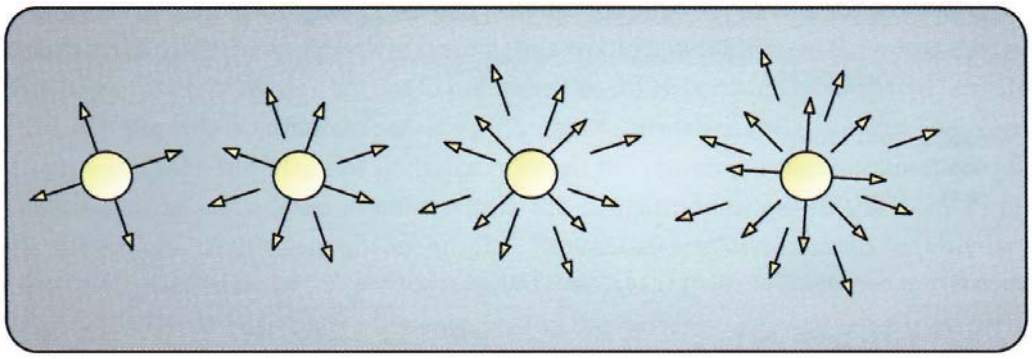

Figure 12.12. Particles emitted in randomized directions from an emission point.

The result of executing this compute shader is to insert eight particles into the system at uniformly distributed, but randomized directions. If this is performed several times in succession, the result is a halo of particles emitted from the emission point in all directions. This is depicted in Figure 12.12, which is a screen shot where the particles are allowed to continue travelling in their initial velocity directions, as determined by the randomized vectors.

Particle State Update Phase

Once all of the particles have been added into the particle system, we can finally execute the actual simulation calculations to update the state of the particle system. This is performed in a compute shader that follows the threading model described in the "Threading Scheme" section. Essentially we dispatch an appropriate number of thread groups to ensure that we have enough threads to process all of the particles in the simulation. In our sample, we can simply retain a count of the number of particles that have been created, and assume that the particles remain in existence for their entire possible lifetime. This estimate is then used to round up to the next higher multiple of 512 to determine how many thread groups to dispatch.

The particle system update shader is set up as shown in Listing 12.8. The update method that is used within the inner loop to actually update the particle is directly based on the physically based gravitation equations described in the "Theory" section. The portion of the shader for executing this update method is shown in Listing 12.10.

```
static const float G = 10.0f;
static const float m1 = 1.0f;
static const float m2 = 1000.0f;
static const float m1m2 = m1 * m2;
static const float eventHorizon = 5.0f;

[numthreads( 512, 1, 1)]

void CSMAIN( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    // Check for if this thread should run or not.
    uint myID = DispatchThreadID.x
              + DispatchThreadID.y * 512
              + DispatchThreadID.z * 512 * 512;

    if ( myID < NumParticles.x )
    {
        // Get the current particle
        Particle p = CurrentSimulationState.Consume();

        // Calculate the current gravitational force applied to it
        float3 d = ConsumerLocation.xyz - p.position;
        float r = length( d );
```

```
            float3 Force = ( G * m1m2 / (r*r) ) * normalize( d );

            // Calculate the new velocity, accounting for the acceleration from
            // the gravitational force over the current time step.
            p.velocity = p.velocity + ( Force / m1 ) * TimeFactors.x;
            // Calculate the new position, accounting for the new velocity value
            // over the current time step.
            p.position += p.velocity * TimeFactors.x;

            // Update the life time left for the particle,
            p.time = p.time + TimeFactors.x;

            // Test to see how close the particle is to the black hole, and
            // don't pass it to the output list if it is too close,
            if ( r > eventHorizon )
            {
                if ( p.time < 10.0f )
                {
                    NewSimulationState.Append( p );
                }
            }
        }
    }
```

Listing 12.10. Updating the state of each particle.

If the current thread corresponds to a live particle, it consumes one particle from the consume buffer. It then calculates the gravitational force from the black hole, the acceleration caused by this force, the new velocity caused by the acceleration, and finally the new position is calculated from the updated velocity. Once the particle state has been updated, we finally test to see if we are within an eventHorizon radius of the black hole. If we are, then the particle is not appended to the output append buffer. This effectively eliminates the particle from the system, since it isn't propagated to the next simulation step. Similarly, if a particle has been alive longer than a threshold amount of time, it is not added to the output buffer.

Rendering the Particle System

After performing a simulation update phase, we have a structured buffer that contains the current state of the particle system. The next step is to render the results of our simulation in an efficient manner. The pipeline configuration is shown in Figure 12.13 for reference during the description. As discussed earlier, if at all possible, we are trying to keep the CPU from having to read back any information about the particle system. For rendering the results of the particle system, we want to do the same thing and avoid having to read the particle data back to system memory. Unfortunately, we cannot directly use the structured buffer as a vertex buffer, since vertex buffers are not allowed to have other bind

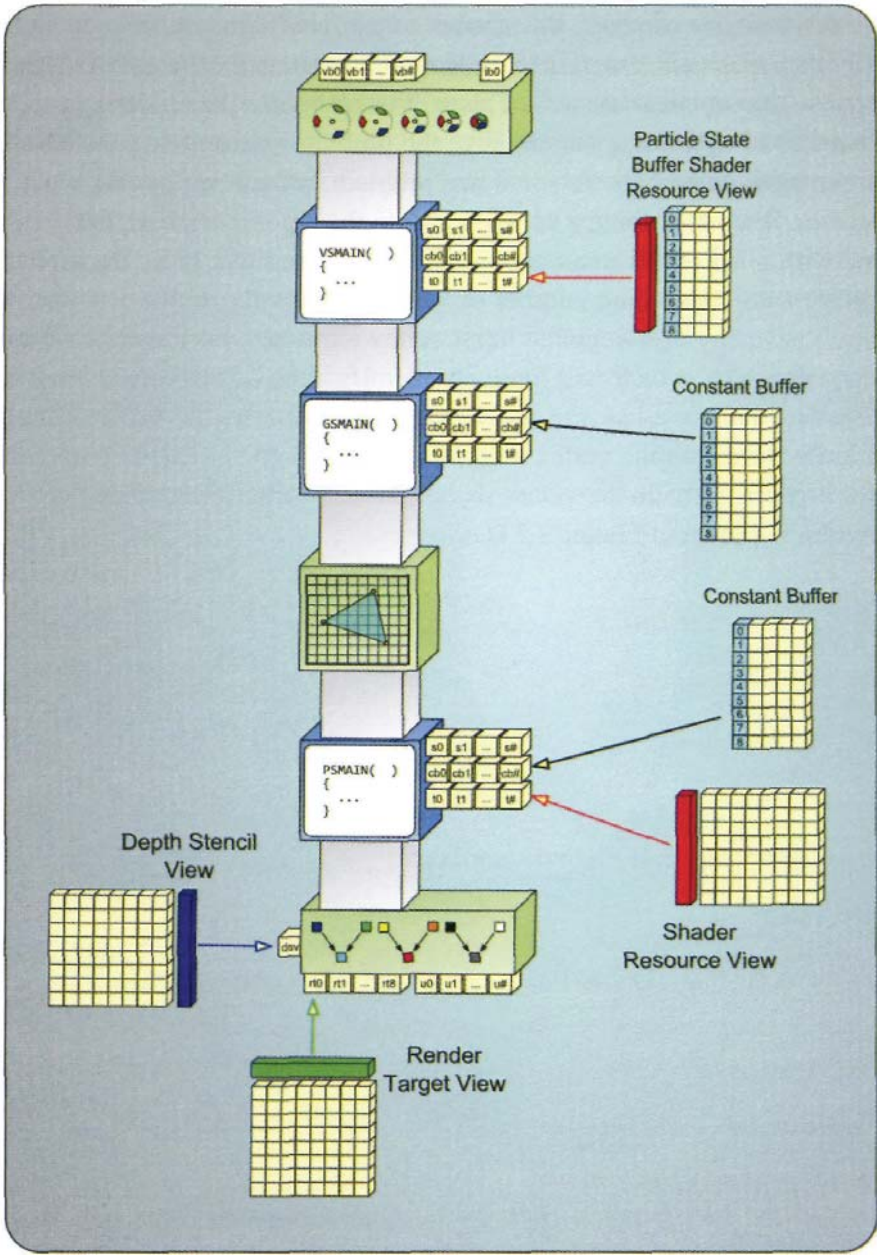

Figure 12.13. The rendering pipeline configuration for rendering the particle system.

flags, which are required for using the append/consume functionality with an unordered access view.

We could copy the contents of the particle system to a third buffer resource that would serve as a vertex buffer. However, depending on how large the particle system is, this could consume a large amount of bandwidth. Instead of doing this, we can use indirect

drawing. In this case, we can copy the number of particles in the buffer to an indirect arguments buffer resource, which can then execute the pipeline with a call to ID3DllDevice Context: :DrawInstancedIndirect. This will let us invoke the rendering pipeline an appropriate number of times, but we still have the problem of supplying the particle data as vertices to the input assembler. To solve this problem, we can bypass the input assembler stage altogether. If we don't bind a vertex buffer to the input assembler, but we still invoke the pipeline with our indirect draw call using a point-primitive type, the input assembler will still create a corresponding number of vertices. The only problem is that these input vertices won't have any user-supplied input vertex attributes. Fortunately, we can still declare the SV_VertexID, which will uniquely identify each of the vertices that are passed into the pipeline. Then we can use the vertex shader to read the vertex data out of the structured buffer based on the vertex identifier. This effectively lets us funnel the particle data into the pipeline through the vertex shader, instead of the input assembler. This vertex shader program is shown in Listing 12.11.

```
struct Particle
{
    float3 position;
    float3 direction;
    float  time;
};

StructuredBuffer<Particle> SimulationState;

struct VS_INPUT
{
    uint vertexid : SV_VertexID;
};

struct GS_INPUT
{
    float3 position : Position;
};

GS_INPUT VSMAIN( in VS_INPUT input )
{
    GS_INPUT output;

    output.position.xyz = SimulationState[input.vertexid].position;

    return output;
}
```

Listing 12.11. Loading the particle data into empty input vertices using the SV_VertexID system value semantic.

With the particle data introduced into the pipeline, we can now render the individual point primitives in the locations of each particle. However, this would be a fairly unsuitable technique for rendering the particle system, since each particle could only occupy a single pixel of the output render target. Instead, we will use the geometry shader to expand our individual point primitives into two triangles that form a quad. The shader program for this operation is shown in Listing 12.12.

```
cbuffer ParticleRenderParameters
{
    float4 EmitterLocationj
    float4 ConsumerLocation;
};

static const float4 g_positions[4] =
{
    float4( -1, 1, 0, 0 ),
    float4( 1, 1, 0, 0 ),
    float4( -1, -1, 0, 0 ),
    float4( 1, -1, 0, 0 ),
};

static const float2 g_texcoords[4] =
{
    float2( 0, 1 ),
    float2( 1, 1 ),
    float2( 0, 0 ),
    float2( 1, 0 ),
};

[maxvertexcount(4)]
void 6SMAIN( point GS_INPUT input[1], inout TriangleStream<PS_INPUT>
SpriteStream )
{
    PS_INPUT output;

    float dist = saturate( length( input[0].position - ConsumerLocation.xyz )
                                                        / 100.0f );
    float4 color = float4( 0.2f, 0.2f, 1.0f, 0.0f ) * ( dist )
                 + float4( 1.0f, 0.2f, 0.2f, 0.0f ) * ( 1.0f - dist );

    // Transform to view space
    float4 viewposition = mul( float4( input[0].position, 1.0f ),
                               WorldViewMatrix );

    // Emit two new triangles
    for ( int i=0; i < 4; i++ )
    {
        // Transform to clip space
        output.position = mul( viewposition + g_positions[i], ProjMatrix );
        output.texcoords = g_texcoords[i];
        output.color = color;
```

```
        SpriteStream.Append(output);
    }

    SpriteStream.RestartStrip();
}
```

Listing 12.12. Expanding points into quads with the geometry shader.

As seen in Listing 12.12, we expand the points into quads in view space which lets us use a static set of offsets for the new vertex locations. These offset positions are then transformed into clip space for use in the rasterizer stage. Similarly, we use a static array of texture coordinates to provide a full texture over the created quad. We also generate a color value that is based on the distance from the black hole, and then pass the color to the output vertices as well. These output vertices are then rasterized and passed to the pixel shader, where it samples a particle texture and writes the output to the output merger. This is shown in Listing 12.13.

```
Texture2D    ParticleTexture : register( t0 );
SamplerState LinearSampler : register( s0 );

float4 PSMAIN( in PS_INPUT input ) : SV_Target
{
    float4 color = ParticleTexture.Sample( LinearSampler, input.texcoords );
    color = color * input.color;

    return( color );
}
```

Listing 12.13. Applying a texture to the output pixels.

After the pixel shader generates the color that is destined for the render target, we must also configure the output merger to accommodate our desired rendering style. Our particle system is going to use what is called *additive blending*, which essentially means that we will modify the blend state so that each pixel produced by the pixel shader will be added to the contents of the render target. This will composite particles onto each other if they overlap, and will create a glowing effect if many particles are occupying the same location simultaneously. In addition, since we are additively blending it is important to disable depth-buffer writing in the depth stencil state, while still using the depth buffer for depth testing. Disabling depth writing requires that any objects in the scene that partially occlude the particle system must be rendered before it, to allow the resulting rendering to be properly sorted according to depth. The resulting rendering can be seen in Figure 12.14.

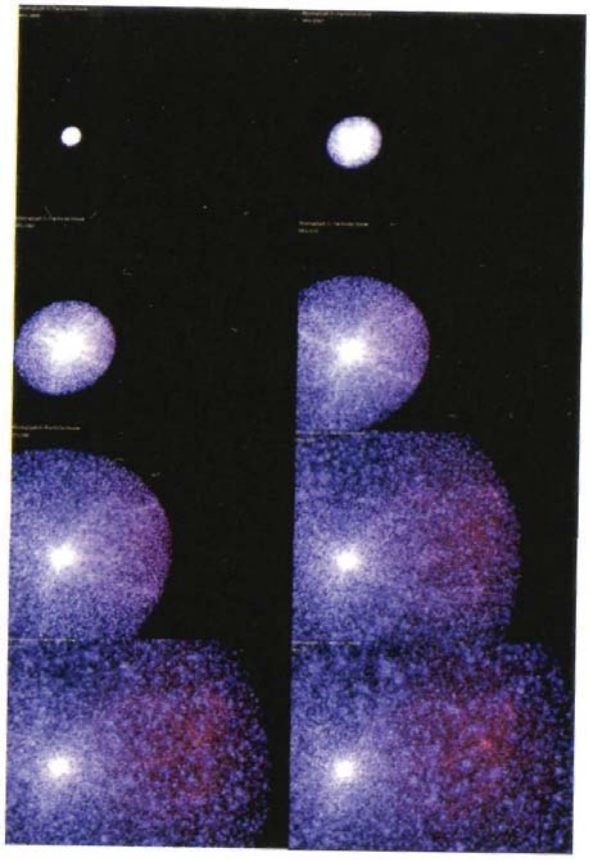

Figure 12.14. The final rendering of the particle system.

12.2.3 Conclusion

The ability of the GPU to efficiently process many parallel data items simultaneously allows the processing of many particles within a particle system. By using some new features of Direct3D 11, we can maintain the particle system completely on the GPU and simply control the update and rendering of the simulation with the CPU. This minimizes any synchronization needed between the GPU and CPU, which allows the GPU to run as fast as possible. In addition, we have seen how to render the resulting particles as quads, using an additive blending technique.