Water Simulation Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

Document serial number: 000004892117

About this Document

The purpose of this document is to present a tutorial in the use of the water simulation technique with the PlayStation®Vita GPU.

Conventions

The typographical conventions used in this guide are explained in this section.

Hyperlinks

Hyperlinks are used to help you to navigate around the document. To allow you to return to where you clicked a hyperlink, select **View > Toolbars > More Tools…** from the Adobe® Acrobat Reader® main menu, and then enable the **Previous View** and **Next View** buttons.

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a 'hint' surrounded by a box. For example:

> **Hint:** This hint provides a shortcut or tip.

Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

> **Note:** This note provides additional information.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code and command-line text are formatted in a fixed-width font. For example:

```
const float A = delta * delta;
```

Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

1 Introduction

This chapter provides an introduction to the water simulation tutorial.

Overview

The water simulation tutorial is an example of using the PlayStation®Vita GPU to perform a water simulation in the fragment shader. This technique is common in games development, and has been previously demonstrated on a PlayStation®Portable, in which case the effect was generated by the CPU. This tutorial demonstrates that the PlayStation®Vita GPU fragment shader processing is a very efficient way to achieve the effect in real-time.

Purpose

The purpose of the water simulation tutorial is to:

- Show how the PlayStation®Vita GPU can be used to generate physically plausible results in an efficient manner, and how to port existing CPU workloads for such effects on to the GPU.
- Demonstrate common rendering methods such as vertex texturing.
- Provide a useful set of program code that can be used in commercial games development for the platform.

2 Theory

This chapter provides some theoretical background to the water simulation.

Algorithm

The algorithm used in this example is the one demonstrated in *Interactive Simulation of Water Surfaces* (see References). In this method, the 2D wave equation is re-evaluated as a convolution kernel used as a rolling sum of the prior simulation frame.

Importantly, *Interactive Simulation of Water Surfaces* (see References) notes that the motion is influenced only by its nearest neighbors. Because of this, the algorithm is easily tackled by convolution using two buffers, and is relatively inexpensive to compute.

Converting the algorithm to be executed on a CPU is a simple procedure that involves storing and flipping two buffers of height data every frame.

The calculations to determine the current wave height are performed by iterating through the height map element arrays in a loop, as shown in the following pseudo-code:

```
const float A = delta * delta;
const float B = 2.0f - 4 * A;

// Recalculate surface
for (i -> h -1)
{
    for (j -> w -1)
    {
        row = i * w;

            value = (pHeight0[row+j-1]+
                     pHeight0[row+j+1]+
                     pHeight0[row+j-w]+
                     pHeight0[row+j+w]) * A +
                     pHeight0[row+j] -
                     pHeight1[row+j] * B;

                     pHeight1[row+j] = value * dampingCoefficient;
    }
}
```

3 Implementation

This chapter provides information about the practical implementation of the water simulation algorithm using the PlayStation®Vita GPU.

Performing convolution in a GPU fragment shader program is a well-established procedure, and converting standard CPU convolution over to shader code is a relatively simple procedure using the Cg language.

From CPU to GPU

When considering the algorithm in the previous section, the first step is to identify the memory access patterns, and map these over to be used by the GPU.

Data Access and Setup

An analysis of the pseudo-code shows that for each height map calculation there are 6 reads to the buffer data, and one final write to set the new value for each element.

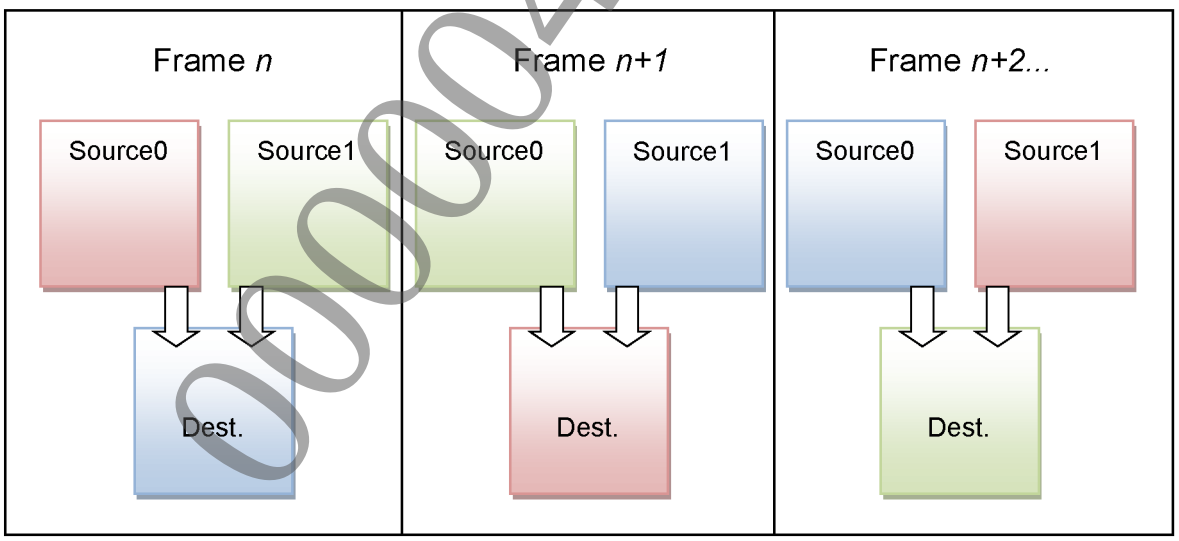
We can equate those reads with texture fetches in the fragment shader, via texture coordinate offsets from the current fragment position.

Instead of using a double buffering scheme which reads and writes to the same buffer each frame, this approach uses the following triple-buffered ring buffer approach:

- The fragment shader samples from two textures in the calculation phase and writes to a third destination render target.
- The render target is cycled round in the following frame and read as one of the source buffers.

Figure 1 shows how destination targets are recycled as textures in subsequent frames.

Figure 1 Ring Buffer Setup



GPU Shader Conversion

After the render targets have been set up to cycle the reads and writes, the data access patterns for the shader are complete. The next step is to convert the CPU C code into Cg shader code for running on the GPU.

The Cg shader language has a very similar syntax to C. Therefore, the Cg shader to compute the water simulation is very similar to the original pseudo code demonstrated previously:

```
sampler2D prevBuffer;
sampler2D currBuffer;
sampler2D obstructionMap;

float texCoordOffset;
float A;
float B;
float dampingFactor;

float4 main(float2 vTexCoord: TEXCOORD0) : COLOR
{
    float2 texCoordPosX = vTexCoord;
    texCoordPosX.x += texCoordOffset;

    float2 texCoordNegX = vTexCoord;
    texCoordNegX.x -= texCoordOffset;

    float2 texCoordPosY = vTexCoord;
    texCoordPosY.y += texCoordOffset;

    float2 texCoordNegY = vTexCoord;
    texCoordNegY.y -= texCoordOffset;

    float obstructionValue = tex2D(obstructionMap, vTexCoord).x;

    float curBufferValPX = tex2D(currBuffer, texCoordPosX);
    float curBufferValNX = tex2D(currBuffer, texCoordNegX);
    float curBufferValPY = tex2D(currBuffer, texCoordPosY);
    float curBufferValNY = tex2D(currBuffer, texCoordNegY);

    float curBufferVal = tex2D(currBuffer, vTexCoord);
    float previousBufferVal = tex2D(prevBuffer, vTexCoord);

    float sum = currBufferValPX + currBufferValNX + currBufferValPY +
                currBufferValNY;
    float result = sum * A + B * currBufferVal - previousBufferVal;

    float final = result * dampingFactor * obstructionValue;

    return final;
}
```

The simulation textures are sampled with point filtering enabled and a texture coordinate offset equal to the reciprocal of the texture size. This ensures that the data accesses follow the patterns outlined in the algorithm.

The Cg code above is very similar to the original version in C, except that the Cg code uses textures as data buffers and offsets the texture coordinates to mimic the access patterns required for convolution.

Obstruction Maps

Another difference in this algorithm is the usage of an obstruction map, which can be used to define the edges of the water simulation as a texture.

A useful property of this type of water simulation is that wherever a value is zero in the simulation buffer, the result of processing the simulation on it is that the water behaves as though it has hit a boundary.

In this manner, programmers can make use of an artist-generated texture to form the shape of the water pool edges.

Figure 2 The Obstruction Map Used in the Tutorial

Generating Normals for Shading

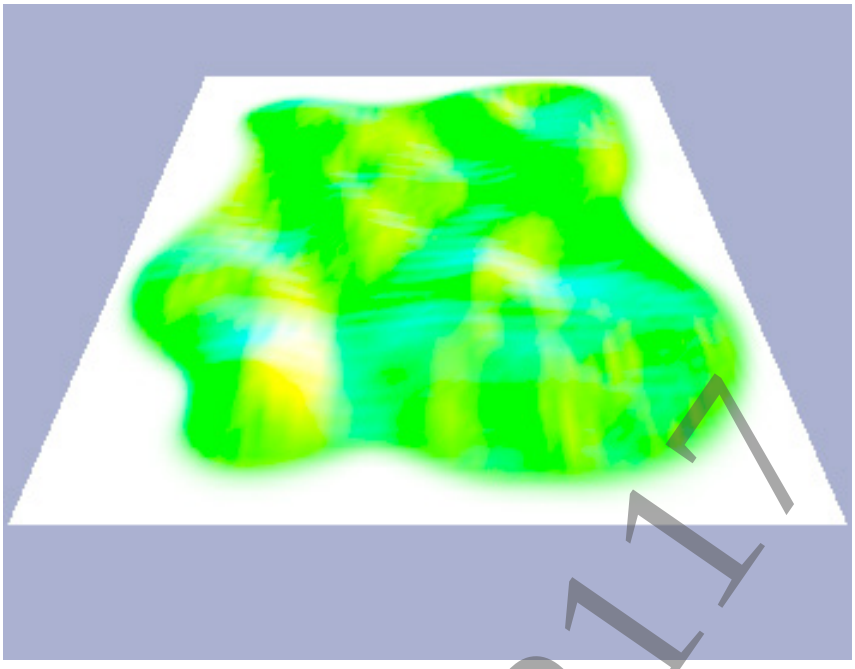
This tutorial makes use of the fragment shader to generate the simulation height data for the water, and the vertex shader texture fetch feature to displace the water surface mesh.

The normal data is generated in the vertex shader when reading in the height map to displace the mesh.

```
float4 height = tex2D(heightTex, aTexCoord);

float2 texCoordPosX = vTexCoord;
texCoordPosX.x += texCoordOffset;

float2 texCoordPosY = vTexCoord;
texCoordPosY.y += texCoordOffset;

float heightPlusX = tex2D(heightTex, texCoordPosX);
float heightPlusY = tex2D(heightTex, texCoordPosY);

// Calculate the normal
float3 dydx = float3(TEX_SIZE, (heightPlusX - height), 0.0);
float3 dydz = float3(0.0, (heightPlusY - height), -TEX_SIZE);
vNormal = cross(dydx, dydz);
```

The normal is generated by creating two vectors from the height map data and texture dimensions, and then performing a cross-product operation on these vectors.

Figure 3 Normal Data Represented as Color, with Obstruction Map Shown



4 Conclusion

This tutorial demonstrates the following features of working with the PlayStation®Vita GPU:

- The PlayStation®Vita GPU can be used to offload CPU workloads such as convolution (in this case used for water simulation) in a relatively simple manner.The render target is cycled round in the following frame and read as one of the source buffers.
- The main difference between the C algorithm and the Cg shader code is the data access patterns – they are mapped to texture reads.
- Vertex texturing is used to fetch the resulting height map, displace the mesh, and generate normal information for lighting and shading.
- An obstruction map can be used to define the border edges of the simulation to make the appearance more user-definable.

References

Gomez, M., *Interactive Simulation of Water Surfaces*, Game Programming Gems, ed. Mark A. DeLoura, Charles River Media Inc., 2000.