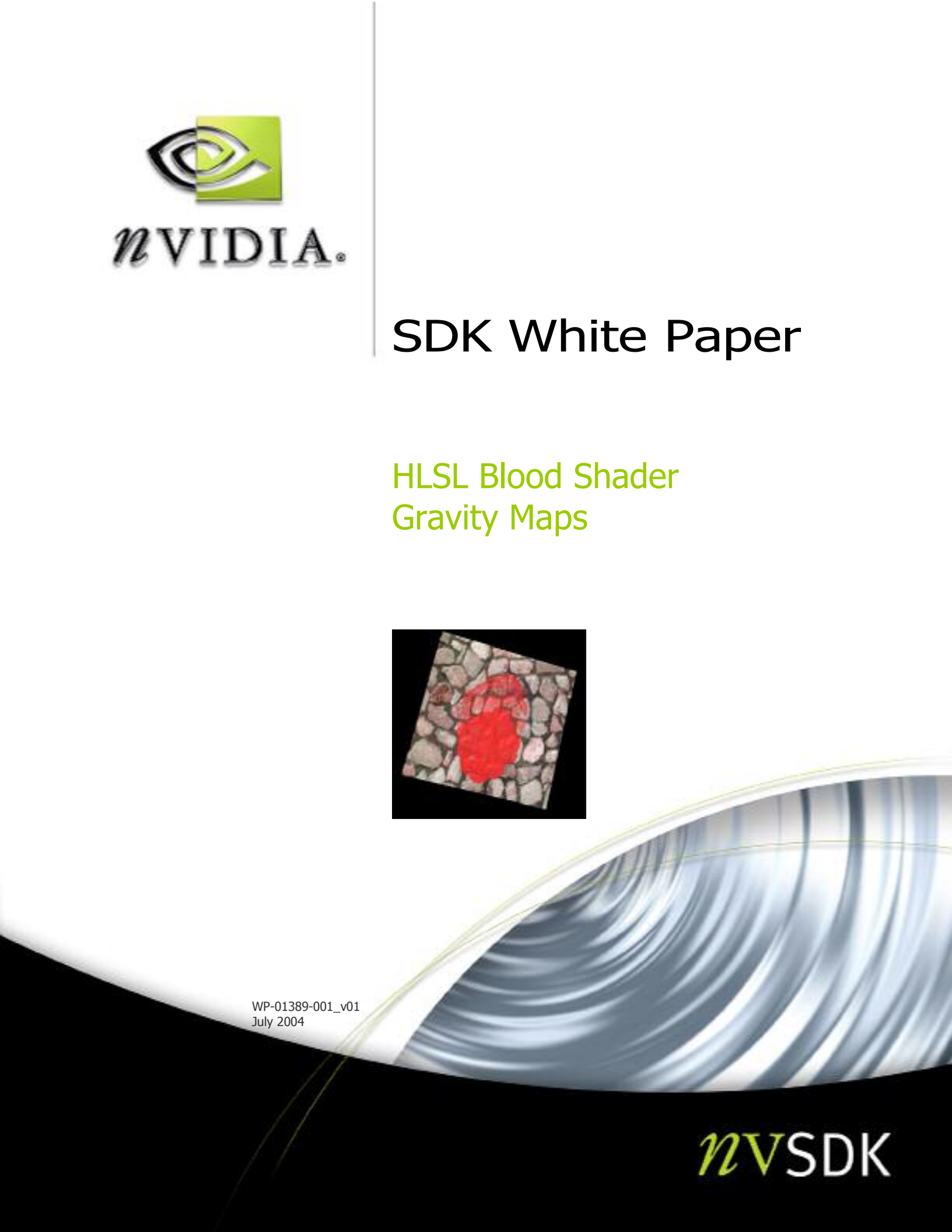

SDK White Paper

HLSL Blood Shader Gravity Maps

WP-01389-001_v01
July 2004

nVSDK

Gravity Maps

With the arrival of programmable pixel shaders, it has become possible to render the world around us in a more convincing and precise manner. A side effect of this increased aesthetic realism is an increase in the correct nature of the physics involved in the rendering process. This increase in accuracy is driven by the demands of techniques such as dot3 bump mapping which require the realistic behavior of surfaces and light. Borrowing from the same concepts that allow the evaluation of lighting equations on a per pixel basis, you can compute physical properties such as gravity at each point. Given this ability to define gravity at each point you can realistically model the dynamics of a fluid across an arbitrary surface.

Kevin Myers
kmyers@nvidia.com
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Introduction

This white paper describes a method for animating fluids across an arbitrary surface. It assumes familiarity with dot3 bump mapping techniques (*Per Pixel Lighting Intro* by Sim Dietrich). The shader utilizes a normal map to compute a per-pixel gravity value which is then used to animate a fluid, stored in a height map. The effect is accomplished using HLSL and PS 2.0.

Creating a Gravity Map

In the preparation of a *Gravity Map* you must first consider the information that a normal map already presents. With a normal map consisting of per pixel normals stored in an RGB texture, you have a description of the direction the surface points in, at each point. If gravity were defined as being a vector in the negative **z** direction, the **x** and **y** components of the normal would define the pull of gravity across texture space. The **z** component can be ignored since the movement is about a plane. Therefore if you were to compute gravity in texture space, a normal map with the **z** component removed would be sufficient. The remaining **x** and **y** components form a 2-D vector that is a ratio of the force of gravity at that point.

The problems with using normal maps to define gravity are the same as those that show up in dot3 bump mapping—the information is defined in tangent space. In reality, gravity direction is a constant vector defined in world space, much like a light vector. Therefore just as the light vector used in dot3 bump mapping, you need to move the gravity vector into tangent space. To accomplish this, apply the **`WorldTranspose`** transformation, followed by an **`ObjectToTangent`** space transformation moving gravity into tangent space. Once the gravity is in tangent space, the **x** and **y** components serve as a 2-D directional vector for gravitational movement.

To determine the final directional vector, take the sum of this 2-D gravity vector and the **x** and **y** components of the pixel's normal. This final 2-D vector then yields the direction that an object will move across the surface in tangent space. As shown in Figure 1, ignore the **z** value and use the **x** and **y** components to dictate direction. Since gravity and normal vectors are normalized the **x** and **y** components can range from -2 to 2. However, any value outside the range -1 to 1 is indicative of a surface that has been tilted upside down. In this case the object falls off the surface, it does not move across the texture.

Apply this process to each point across the surface determining a unique gravity vector for each point. To create a more viscous fluid that responds in a sharper manner to surface details, raise the gravity vector to the third power. This creates a sort of attenuation that ramps the effect of gravity. Finally, write out the **x** and **y** components to **r** and **g** respectively. Since our values can range from -1 to 1, you need to pack the vector applying the equation 1/2 * x + .5.

Animating Blood

At this point, you have defined gravity at every point and saved it out to a texture. You can now use it to animate fluids. Remember that the shader is a self-feeding process with the amount of fluid, in this case blood, stored in a height map. The height map is defined in the unused **b** component of the gravity map. Every frame the shader reads the height value of the blood at each texel, as well as the 2-D gravity vector. It then subtracts from the current texel whatever percentage its gravity vector states it will lose this frame. Finally, it checks its four neighbors for any fluid that may be coming into the current texel.

To compute how much blood will fall into the current texel, the shader must sample the gravity map values of the four neighbors. Each of the four texels left, right, above, and below is checked for gravity vectors pointing in the direction of the current texel. For instance, in checking the neighbor above the current texel the shader looks at the **y** (green) component. Anything in the range of 0 to -1 (unpacked) gives the fraction of blood that this texel will take. This fraction is then multiplied by the height of the blood at the neighbor as given in the equation **`abs(TexelAbove.y) * TexAbove.b`** . This value is computed for all four neighbors and added to the current texels height.

Finally, if the current texels blood height is greater than a predefined minimum, the generation of the gravity map is again performed for the next frame. This cycle continues to repeat until there are no longer any texels with blood high enough (predefined) to move.

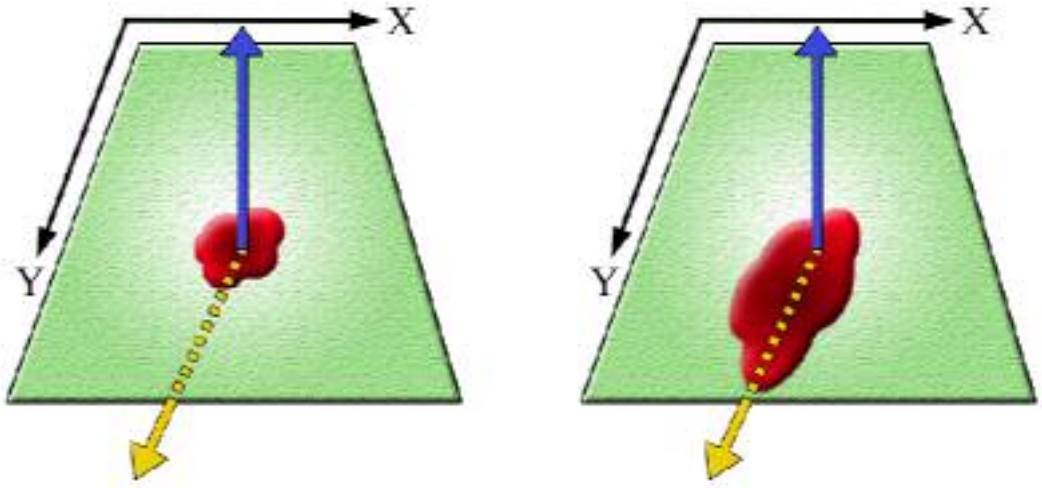

Figure 1. Animating Blood

Displaying the Horror

The final product of this shader is a continuously changing height map with a 2-D gravity induced directional vector defined at each point. Visually, the goal is produce the effect of a thick fluid oozing across a surface. The height map contains information regarding the intensity of the color for each point. Take this value and multiply it by a given blood color. This yields a final blood color contribution which is added to the color of the surface material. In order to increase the intensity of the blood color, decrease the color contribution of the surface by a factor of (**`1 – BloodHeight`**). For added realism, compute deltas for the blood height of each texel based on its neighbors. Derive d**`x = LeftHeight – RightHeight and dy = BelowHeight – AboveHeight`**. These values are then added to the surface normal to give the effect of a thick fluid.

The final product uses a simple moving specular light to show bump mapped blood, dripping down a wall, animated entirely on the GPU.

Bibliography

Per Pixel Lighting Intro

Sim Dietrich

http://developer.nvidia.com/attach/1659

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com