White Paper

Cloth Simulation

February 2007
WP-03018-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
_v01	February 15, 2007	CZ, TS	Initial release

Go to sdkfeedback@nvidia.com to provide feedback on Clipmap.

Cloth Simulation

Abstract

The sample demonstrates how to simulate cloth on the GPU using DirectX 10.

The cloth vertex positions are computed in several rendering passes by looping through the vertex and geometry shader stages, using the stream output stage to stream the positions from the geometry stage directly back to video memory.

The few cloth vertices whose positions are interactively controlled by the user ("anchor points") are updated by setting the buffer containing the cloth vertex positions as a render target and rendering point primitives.

The sample also makes extensive use of the bitwise shader operations introduced in DirectX 10 to store and manage the state of each cloth vertex.

Cyril Zeller
NVIDIA Corporation

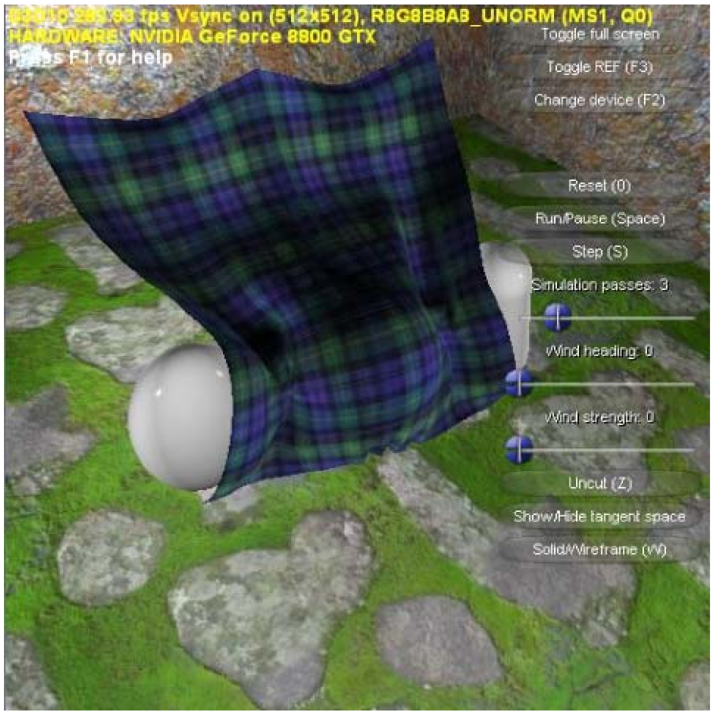
Motivation

Cloth simulation exhibits a lot of data parallelism: a piece of cloth can be modeled as a 2D network of many particles with the same dynamic. This makes it a good candidate for GPU implementation.

The implementation proposed in the sample is geared toward performance and visual realism, rather than physical accuracy. This is more suitable to applications such as 3D games and virtual reality systems. These applications usually do not need the result of the cloth simulation on the CPU as it is only used for rendering. It is therefore all the more natural to entirely off-load it from the CPU to the GPU.

The model and simulation algorithm are the same as the DirectX 9 implementation from the NVIDIA SDK 9.5, described in details in *ShaderX4: Advanced Rendering Techniques,* section 1.2 "Practical Cloth Simulation for Modern GPU."

The DirectX 10 implementation differs from the DirectX 9 implementation in that the particles are stored in a buffer and mostly processed during the vertex and geometry shader stages instead of being stored in a texture and processed during the pixel shader stage. Since the geometry shader stage can output multiple vertices, this allows us to handle more than one particle in one shader call, making the implementation more work-efficient: Fewer rendering passes are required and the distances between each particle are computed only once during each rendering pass. Moreover, it is easier to handle non-rectangular meshes, or meshes with holes since particles are stored in a vertex buffer.



```
D3D10 289.93 fps Vsync on (512x512), R8G8B8A8_UNORM (MS1, Q0)
HARDWARE: NVIDIA GeForce 8800 GTX
Press F1 for help
```

Simulation passes: 3
Wind heading: 0
Wind strength: 0

Figure 1. Cloth Simulation Example

How It Works

Model and Algorithm

A cloth object is modeled as a set of particles. Each particle is subject to:

- External forces, such as gravity, wind and drag;
- Various constraints:
    - To maintain the overall shape of the object (spring constraints),
    - To prevent interpenetration with the environment (collision constraints).

The particle's equation of motion resulting from applying the external forces is integrated using Verlet integration.

The various constraints create a system of equations linking the particles' positions together. This system is solved at each simulation time step by relaxation; that is by enforcing the constraints one after the other for a given number of iterations.

Spring constraints between neighboring particles are of two types as illustrated in Figure 2.

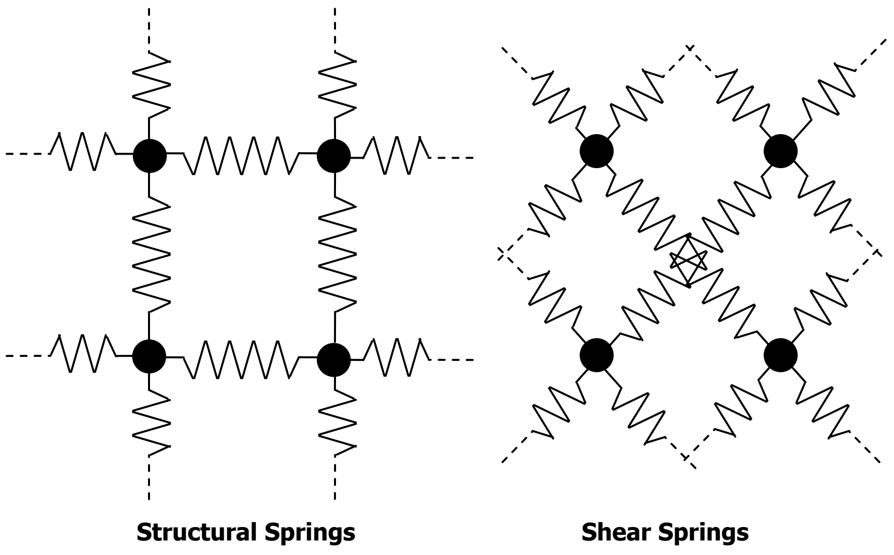


Figure 2. Spring Constraints Between Neighboring Particles

A spring constraint between two particles, **P** and **Q**, is defined as a distance constraint between **P** and **Q**:

Distance(P, Q) = distance at rest

and enforced by moving **P** and **Q** away or towards each other as illustrated in Figure 3.

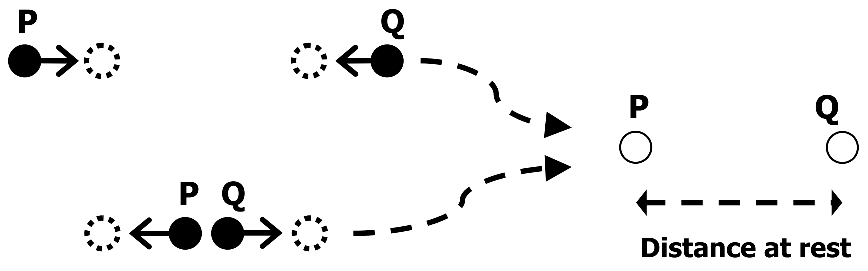


Figure 2. Distance Constraint Between Particles P and Q

The environment is defined as a set of collision objects of various geometric types: planes, spheres, capsules, ellipsoids. A collision constraint between a particle and a collision object is enforced by checking whether the particle is inside the object or not, and if it is inside, by moving the particle to a position at the surface of the object, usually the position at the surface that is the closest to the particle's current position.

Following is an outline of the complete algorithm for every simulation step:

- **Step 1**: For every particle that is not an anchor point:
  Apply force through equation of motion
- **Step 2:** For every particle that is an anchor point:
  Update position
- **Step 3: For every relaxation step:**
  - **Step 3a**: For every spring constraint:
    Enforce spring constraint
  - **Step 3b**: For every particle:
    - For every collision object:
      Enforce collision constraint

GPU Implementation

The particles are stored in a buffer and, at each simulation step, processed through several rendering passes.

Forces can be applied to each particle in parallel. Similarly, collision constraints can be enforced for each particle in parallel. So, **Step 1** and **Step 3b** are implemented as one rendering pass through the vertex shader stage.

Updating the anchor points' positions can also be done in parallel, but given that the number of anchor points represents a small percentage of the total number of particles, **Step 2** is implemented as one rendering pass through the pixel shader stage by rendering one point primitive per anchor point.

A spring constraint involves two vertices; so it is processed during the geometry shader stage as a line primitive. The geometry shader can actually process more than one constraint in one call since it can have up to six vertices as input by declaring its input parameter as a triangle with adjacency. The choice of the optimal number of input vertices is not straightforward: On one hand, if fewer vertices are processed by one geometry shader invocation, then more rendering passes are required to process all constraints; on the other hand, geometry shader performance degrades as more vertices are output. In the sample, we chose to process four vertices per geometry shader (one line with adjacency) as the best compromise. We therefore process one group of spring constraints per geometry shader call.

Two spring constraints can be enforced in parallel if they are independent; meaning that the two pairs of particles involved in each constraint do not share a particle. Similarly, two groups of spring constraints can be enforced in parallel if they are independent (each spring constraint of one of the two groups is independent from each spring constraint of the other group). **Step 3a** therefore consists of several rendering passes, each of them processing one set of independent groups of constraints. These sets must be a partition of the total set of constraints and in order to minimize the number of rendering passes required to process all constraints, they need to be maximal. Figure 4 shows the four sets used in the sample for a rectangular piece of cloth; **Step 3a** is therefore performed in four rendering passes.

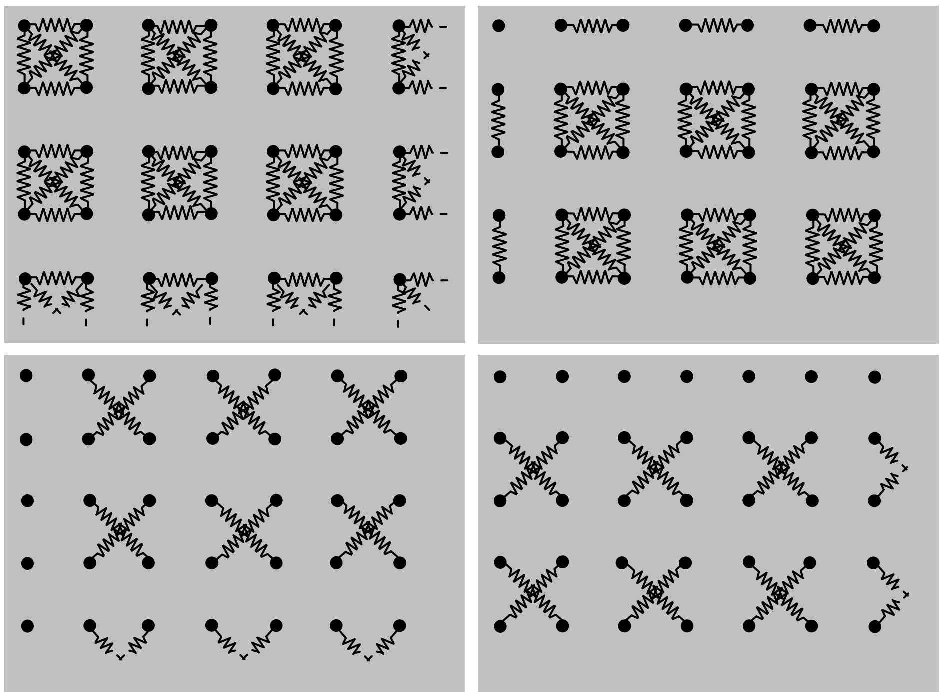


Figure 4. Partitioning of the total set of spring constraints in four sets of independent groups of spring constraints

Implementation Details

The particle attributes (state, position, normal, tangent, color) are stored into attribute buffers.

The state attribute is a 32-bit integer whose bits encode connectivity information and specify whether the particle is an anchor point or not. The state attribute is stored in the position buffer.

There are three copies of the position buffers: One of them is bound as input to the vertex shader stage and another one is bound to the stream output shader stage; the third one is necessary because Verlet integration also requires the particle positions from the previous simulation step; it is bound to the vertex shader stage during **Step 1**. The buffers are rotated after each rendering pass.

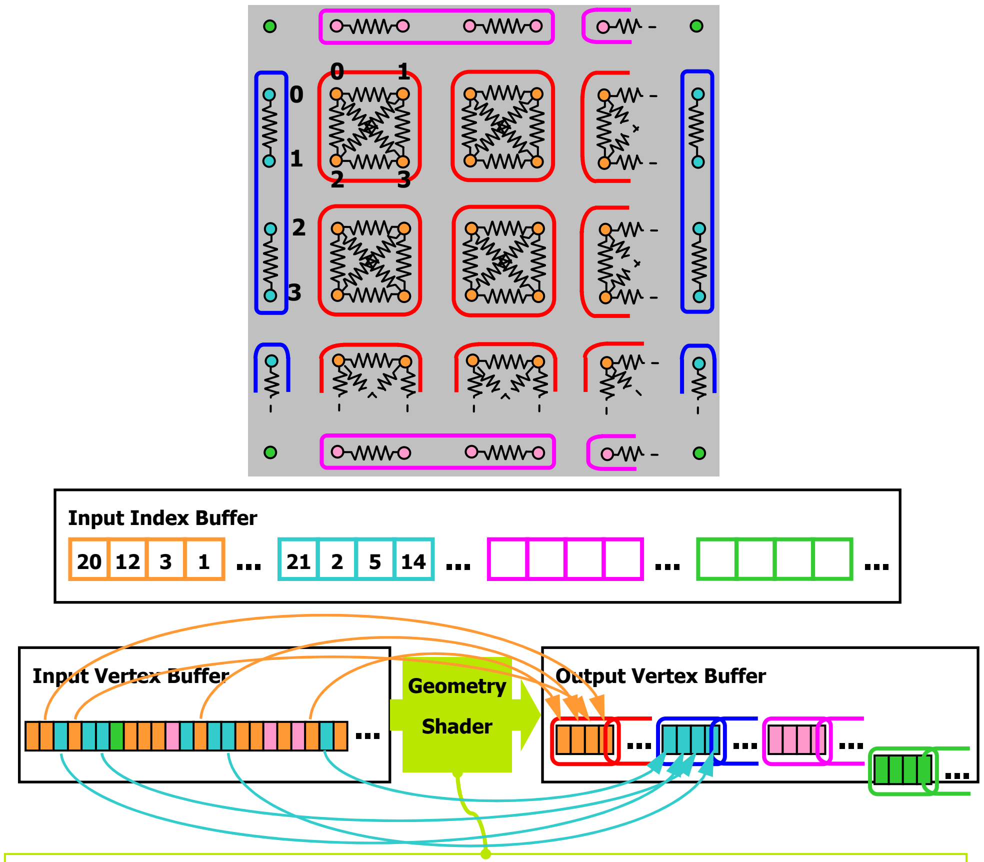
An index buffer is required for each rendering pass of **Step 3a** to feed the geometry shader stage with the right groups of 4-tuples of particles in the most optimal order. Figure 5 illustrates the layout of such an index buffer for the top right set from Figure 4. There are four different connection configurations in this case depending on the location of the particles of the 4-tuples within the piece of cloth (interior, vertical edge, horizontal edge, corners). These connection configurations, encoded in the state attribute of the particles, are handled as a switch statement in the geometry shader. In order to maximize branch coherency, the index buffer is laid out such that the 4-tuples with the same connection configuration are grouped together.

The order in which the particle positions are output to the position buffer through the stream output stage at the end of each rendering pass of **Step 3a** is determined by the corresponding index buffer. Therefore, at the end of **Step 3a**, the particle positions are stored in the position buffer in a different order than the input order at the beginning of **Step 3a**. That is why **Step 3b** also requires an index buffer whose function is to reorder the particle positions into the order assumed at the beginning of **Step 3a** and by the rendering step.

Following is an outline of the complete algorithm for every simulation step:

- **Step 1**:
  - Set a vertex shader that applies force (**`ApplyForces`**)
  - Set position buffer as stream output target
  - Render as a point list
  - Swap position buffers
- **Step 2**:
  - Set a pixel shader that updates position (**`TransformAnchorPointPS`**)
  - Set position buffer as render target
  - Render anchor points only as a point list
  - Swap position buffers

- **Step 3**: For every relaxation step:
  - **Step 3a**: For each sets of independent groups of spring constraints:
    - Set a geometry shader that applies distance constraints (**`SatisfyStructuralAndShearSpringConstraints, SatisfyShearSpringConstraints`**)
    - Set position buffer as stream output target
    - Render as an indexed line list with adjacency
    - Swap position buffers
  - **Step 3b:**
    - Set a geometry shader that applies distance constraints (**`SatisfyCollisionConstraints`**)
    - Set position buffer as stream output target
    - Render as an indexed point list
    - Swap position buffers



```
void SatisfySpringConstraintsGS(lineadj Particle particle[4],
                                inout PointStream<Particle> stream)
{
    switch (connectionConfiguration) {
        case 0:
            SatisfyDistanceConstraint(particle[0], RIGHT, particle[1]);
            SatisfyDistanceConstraint(particle[0], DOWN,  particle[2]);
            SatisfyDistanceConstraint(particle[2], RIGHT, particle[3]);
            // etc.
            break;
        case 1:
            SatisfyDistanceConstraint(particle[0], DOWN, particle[1]);
            SatisfyDistanceConstraint(particle[2], DOWN, particle[3]);
            break;
        case 2:
            ...
        case 4:
            break;
    }
    for (int i = 0; i < 4; ++i) stream.Append(particle[i]);
}
```

Figure 5. Index Buffer Layout for Spring Constraints

The user can also cut cloth by dragging the mouse over it from one window point to another. To compute the cut, these two window points are back-projected to their corresponding world space positions to define a 3D triangle together with the camera origin. This triangle acts as a cutter (see Figure 6); every cloth spring it intersects is suppressed.

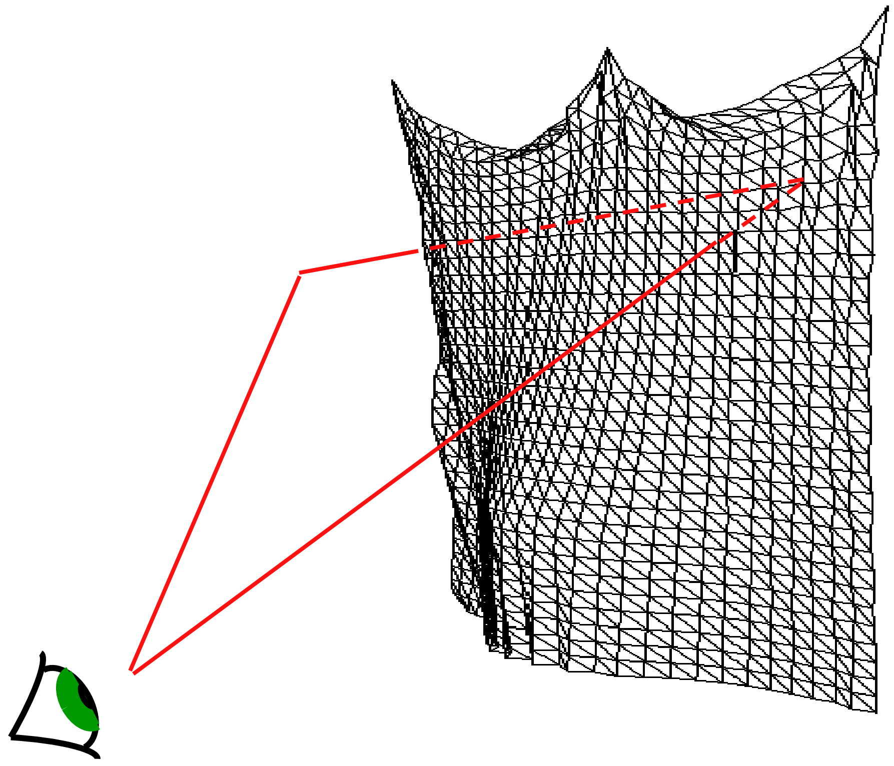


Figure 6. The Triangle Defined by the Mouse Motion Acts as a Cutter

Which springs intersect with the cutter is determined during the first pass through **Step 3a** that immediately follows the user cutting action. A spring is suppressed by modifying the connection configurations of the two corresponding particles.

During the rendering pass, if a triangle has edges that correspond to springs that have been previously suppressed, it is tessellated on-the-fly into up to four new triangles to approximate the cut: The positions of the new vertices are not based on the exact cut location, but simply set as if springs were always cut in their middle. These new vertices are not collided against the environment to save computation time.

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