SDK White Paper

Cloth

July 4, 2005

nVSDK

Abstract

The sample demonstrates how to use Shader Model 3.0 to simulate and render cloth on the GPU. The cloth vertex positions are computed through several pixel shaders and saved into a texture. A vertex shader then reads these positions using Vertex Texture Fetch (VTF) to render the cloth.

Cyril Zeller

sdkfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

A piece of cloth is modeled as an array of particles.

Forces

Each particle is subject to a force and its equation of motion is integrated using Verlet integration. In the case of a constant simulation time step Δt, the new position of the particle is thus computed from the old one using the following equation:

$$P(t + \Delta t) = P(t) + k\,(P(t) - P(t - \Delta t)) + \Delta t^2\, F(t) / m \qquad (1)$$

where:

- P(t) is the position of the particle at time t,
- F(t) is the force at time t,
- m is the mass of the particle,
- k is an arbitrary damping coefficient, usually very close to 1.

This equation is derived by approximating $P(t + \Delta t)$ and $P(t - \Delta t)$ by the first terms of their Taylor expansion:

$$P(t + \Delta t) \sim P(t) + \Delta t\, P'(t) + \Delta t^2\, P''(t) / 2$$

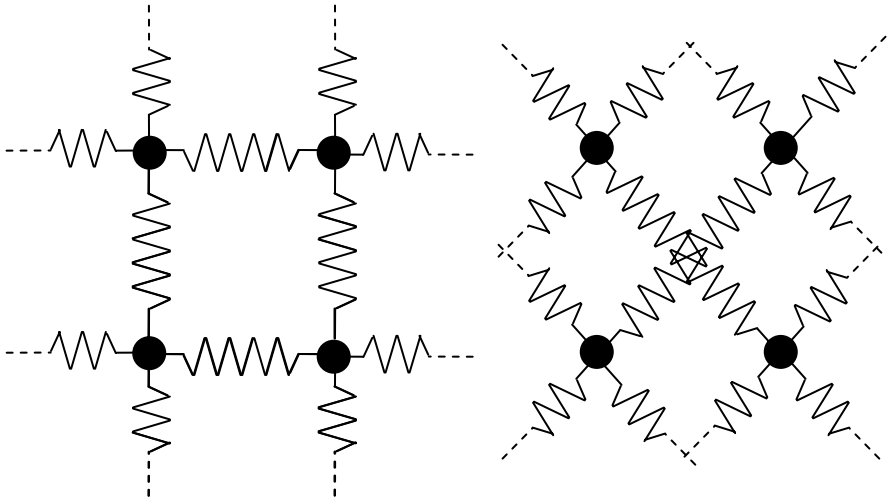
$$P(t - \Delta t) \sim P(t) - \Delta t\, P'(t) + \Delta t^2\, P''(t) / 2$$

adding these two equations, and noting that P"(t) is equal to m F(t).

Note that no force is applied to the particles that have been selected by the user to be either fixed ("nailed") or moved by user interaction.

Constraints

The particles are also linked to each other by structural and shear springs:



Structural springs

Shear springs

The springs are assumed to have infinite stiffness, which is to say that two particles P_1 and P_2 linked by a spring are constrained by:

$$\text{Dist}(P_1, P_2) = d$$

where d is a constant equal to the spring length at rest.

For a flat rectangular piece of cloth of size (e_x, e_y) and modeled by an array of (W x H) particles:

$$d_{\text{structural, x}} = e_x / (W - 1)$$

$$d_{\text{structural, y}} = e_y / (H - 1)$$

$$d_{\text{shear}} = \text{sqrt}(d^2_{\text{structural, x}} + d^2_{\text{structural, y}})$$

For a non-rectangular or non-flat piece of cloth, every spring has a specific length at rest.

Each particle is also constrained to stay out of any collision object – plane, sphere, box, ellipsoid - present in the environment.

A relaxation technique is used to converge towards a state where all these constraints are simultaneously satisfied. This consists in iteratively enforcing one after the other until it is deemed sufficient. See [Jakobsen 2001] for more details.

A distance constraint between P_1 and P_2 is enforced by displacing:

$$P_1 \text{ by } \quad s_1 (1 - d / \text{Dist}(P_1, P_2)) * (P_2 - P_1) \qquad (2a)$$

$$P_2 \text{ by } \quad - s_2 (1 - d / \text{Dist}(P_1, P_2)) * (P_2 - P_1) \qquad (2b)$$

with (s_1, s_2) being the responsivenesses of P_1 and P_2. The responsivenesses are equal to:

(0.5, 0.5)	if both particles are moving freely
(0, 1) or (1, 0)	if P_1 or P_2 is fixed

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 the position at the surface of the object that is the closest to the particle's current position. Note that in the case of the ellipsoid, computing the closest position requires an iterative calculation, so for simplicity and speed, the particle is moved to the intersection of the ellipsoid with the line that goes from the ellipsoid's center to the particle's current position.

A piece of cloth with holes is handled by computing a geometry image of the original cloth mesh [Gu et al. 2002] and using the initial distances between the particles as the springs' lengths at rest. A geometry image is created from a given mesh by cutting it along a network of edge paths such that it becomes topologically equivalent to a disk and then sampling the result over a grid to get the positions of the vertices at each pixel of the image. The particles located along the cut – or seam – are duplicated at different locations on the boundary of the geometry image. The duplicates corresponding to such a particle are simulated independently during the motion integration and distance constraint steps and their positions are reconciled just before the collision step by averaging them.

Algorithm

Here is the algorithm's outline:

- For every simulation time step:
  - **Step 1:** For every particle that isn't fixed or user-moved:
    - Apply force through equation (1)
  - **Step 2:** For every relaxation step:
    - **Step 2a:** For every distance constraint:
      - Apply equations (2a) and (2b)
    - **Step 2b:** For every particle:
      - If the particle is part of the seam:
        - Replace its position with the average of the positions of all the corresponding duplicates
      - For every collision object:
        - If the particle is inside
          - Move the particle out of the object

GPU Implementation

The particles' positions of a piece of cloth made of (W x H) particles are stored in three rotating (W x H) floating-point textures and processed by several pixel shaders successively. One texture holds the old positions; another one holds the current positions and the third one is used to store the results of each pixel shader pass. A pixel shader pass is done by setting this third texture as a render target and by rendering a quad of (W x H) pixels.

The w-component of the 4-component floating-point vector that represents a particle in the particle textures is used to encode:

- Whether the particle is fixed or not,

- Whether the particle is a seam particle,

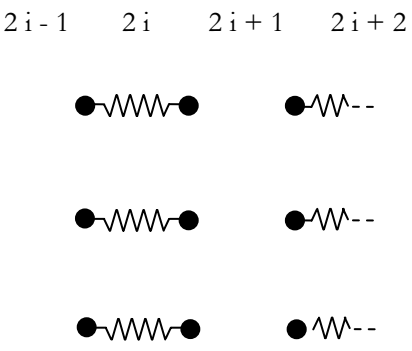
- If the particle is a seam particle, the texture address of the first of its duplicates along the seam.

All the pixel shaders used for the simulation are in `ClothSim.fx`.

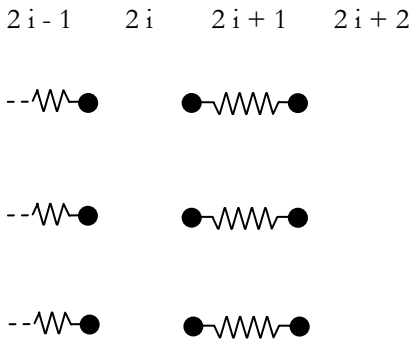
Step 1 is performed by `ApplyForces`. `ApplyForces` moves every particle that is not fixed or user-moved. The position of the fixed or user-moved particles is set by `SetPosition and TransformPosition`.

Step 2a is performed by a series of pixel shaders dealing with separate rows or columns of springs:

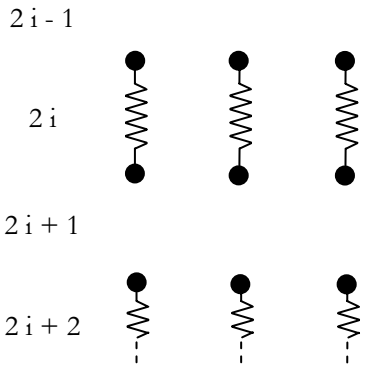
- `SatisfySpringConstraintXSpringEven`:



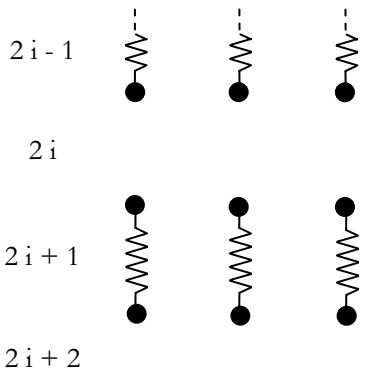
- `SatisfySpringConstraintXSpringOdd`:



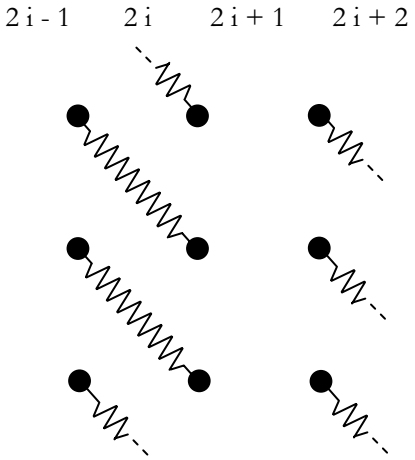
- `SatisfySpringConstraintYSpringEven`:



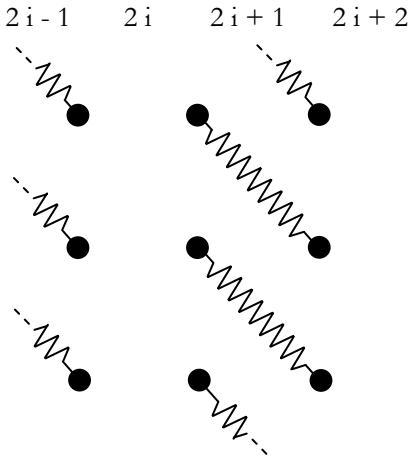
- `SatisfySpringConstraintYSpringOdd`:



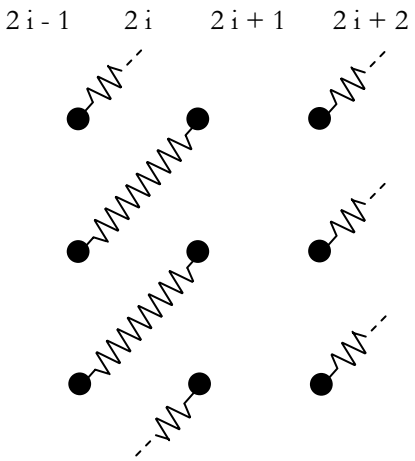
- `SatisfySpringConstraintXYSpringDownEven`:



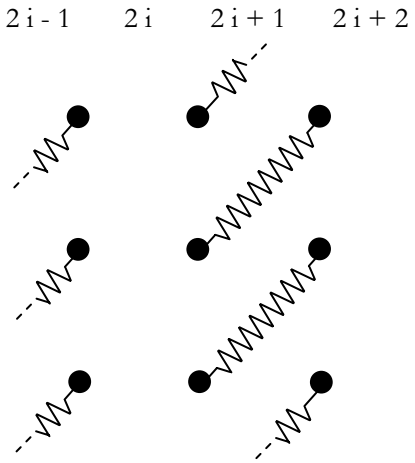
- `SatisfySpringConstraintXYSpringDownOdd`:



- `SatisfySpringConstraintXYSpringUpEven`:



- `SatisfySpringConstraintXYSpringUpOdd`:



The responsivenesses are stored into eight textures that get used once respectively during the execution of the eight pixel shaders above. These textures are updated each time the user selects or unselects a particle, or cuts into the cloth. When a spring between P_1 and P_2 is cut, the responsivenesses are set to zero for both positions.

The springs' lengths at rest are specified, either as constants in the case where they are the same for all the springs updated by one pixel shader call, or, like for the responsivenesses, as eight textures in the case where they differ from one spring to the other.

Note that these eight pixel shader passes can be applied in any order.

Step 2b is performed by `SatisfySeamAndCollisionConstraints`. Collision objects are stored into textures that get updated each time a new object is added or an existing object is removed or moved. There is one texture per type of collision object.

Once the simulation is done, the texture containing the final particle positions is looked up by vertex shader `SimulatedVS` - from `Scene.fx` - to set the vertex positions of a vertex buffer of (W x H) vertices. This vertex buffer is rendered with pixel shader `SimulatedPS` to display the piece of cloth.

Cloth cutting is done by:

- Determining which ones of the cloth triangles the user cuts when dragging the mouse from one window point A to another point B,
- Updating the corresponding responsivenesses,
- Removing these triangles from the cloth index buffer.

The first step above is performed by the `Cut` technique (in `ClothSim.fx`, as well) and consists in intersecting every cloth triangle with the triangle formed by the camera position and the two points of the scene that project to A and B.

Source Code Organization

The source code is divided into four `cpp` files:

- `ClothSim.cpp`: This file contains the implementation of the `ClothSim` class and this is where the simulation takes place. The `ClothSim` class takes as input a gravity force, a wind force, a distance constraint, a list of planes, spheres, boxes and ellipsoids as textures, and a list of anchor points as a vertex buffer – the fixed or user-moved particles -, and computes the new cloth vertex positions and normals every frame.
- `Scene.cpp`: This file contains the definition of the `Scene` name space and depends on the `ClothSim` class. The `Scene` name space gathers all the scene management functions, including editing, simulation and rendering. It exposes an `Object` class and the list of all the objects contained in the scene. There is one `Object` instantiation per collision object and cloth. Each object can be dimensioned, positioned, oriented, rendered, selected, targeted and cut. An object is targeted when the mouse is located on top of it. Cutting only applies to clothes. Internally, the `Object` class is derived into two classes: `CollisionObject`, `Cloth`. The `CollisionObject` class is in turn derived into four classes: `Plane`, `Sphere`, `Box`, and `Ellipsoid`.

  The character isn't implemented as an `Object`, but as a `D3DXFRAME`.
- `GUI.cpp`: This file contains the definition of the `GUI` name space and depends on the `Scene` name space. The `GUI` name space implements the graphical user interface and exposes all the event callbacks setup in `Main.cpp`.
- `Main.cpp`: This file contains the executable's entry point and depends on the `GUI` name space.

Bibliography

[Jakobsen 2001] Jakobsen, T. Advanced character physics. Game Developer's Conference 2001

www.gdconf.com/archives/2001/jakobsent.doc

[Gu et al. 2002] Gu, X., Gortler, S., and Hoppe, H. "Geometry Images." ACM SIGGRAPH 2002, 355-361.

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