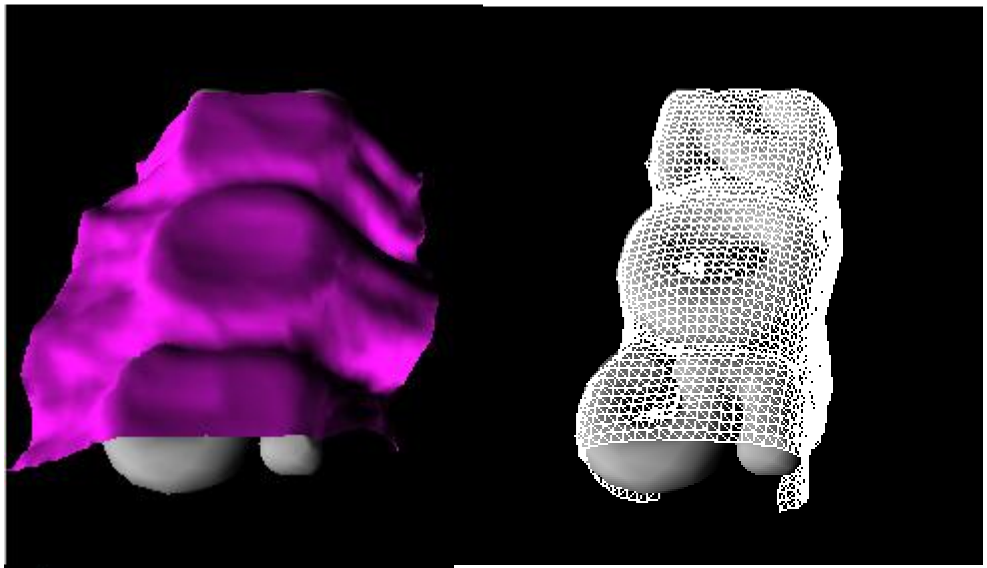
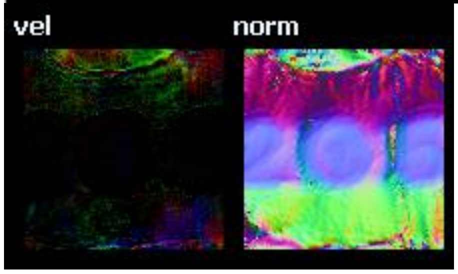




CS205 Final Project:
Real-Time Cloth Simulation on the GPU

Sam Z. Glassenberg
glassenb@stanford.edu

1. Introduction

The goal of this project was to explore how the various methods discussed in CS 205 for solving ODE's and can be implemented on graphics hardware in the context of physical simulation. For this project I implemented a real-time cloth simulation with collision detection.

My goals for the simulation itself, in priority order, were:

1) Realistic visual appearance
2) Fast rendering speed
3) Accuracy

These priorities dictated design choices made throughout the project.

Graphics hardware is heavily parallelized with support for fast floating-point vector math; an optimal combination for real-time graphics and simulation. In addition to its speed, GPU implementations provide the additional advantage of alleviating CPU load. If the computational cost of a cloth simulation in a video game, for example, were passed on to GPU, it would free up vital CPU load which can in turn be used for AI, etc.

I've been interested in GPU programming since I first played with an Xbox Dev kit during my internship at LucasArts in 2001. The Xbox featured a version of NVidia GeForce 3, the first generation of graphics cards to support genuine real-time programmable vertex and fragment (pixel) shading.

Since then, I've done work in haptics and rendering, but I never found a good opportunity to try my hand at real-time GPU programming. The open-ended nature of the CS 205 final project assignment provided me with this opportunity.

The entire simulation runs on the GPU. The only time any simulation data is ever loaded into system memory occurs in one line of code where the scaled position data is transferred immediately without modification from the pixel shader to the vertex shader (see Hardware Limitations).

2. Methods

2.1. Software Platform

I wrote this project in C# using DirectX 9. All pixel and vertex shaders are written in High-Level Shading Language (HLSL) using the DirectX 9 Technique format.
This project could certainly have been implemented in C++, OpenGL, and/or Cg (NVidia's high-level shading language, not to be confused with Conjugate Gradient discussed later). However, I'm already familiar with C++ and OpenGL and wanted to learn new platforms.

2.2. Hardware

I developed the simulation on a 2.4 GHz Pentium 4 with an ATI 9700 Pro All-In-Wonder graphics card. The simulation is heavily optimized for (and dependent on) the ATI 9700 Pro. This was the top-of-the-line video card six months ago, and has a variety of key features including support for second-generation hardware shading languages (ps 2.0 and vs 2.0) and multiple simultaneous render targets. The device itself is still limited to a subset of hardware features support by the latest release of DirectX 9. If the program is run on a card with insufficient support, it will terminate.

2.2.1. Hardware Limitations

Successful implementation entailed working around the many limitations of second-generation graphics hardware. It turns out that many of these are slated to be resolved by future devices.

- No Vertex Shader Output

It would make more intuitive sense to perform the per-vertex physics calculations on a vertex shader as opposed to a pixel shader. Unfortunately, vertex shaders can't output results to any sort of buffer. Their results are interpolated between pixels and passed only to pixel shaders within their adjacent triangles.

- No Indirect Writes

Pixel shaders can only output to their assigned position within the render target texture. In order to write to a location in a texture, a triangle must be drawn on top of it with an associated pixel shader. The render target cannot be simultaneously used as an input.

- No Vertex Shader Texture Access

Vertex shaders can only read from vertex buffers and a short list of constant variables. All physics calculations need to be performed by pixel shaders on textures, but at some point the cloth needs to be rendered as a vertex mesh. This problem could be avoided if a pointer to a float4 texture in video card memory could be passed as a similarly-formatted float4 vertex buffer in the same memory space. This is also not permitted. The only workaround is to copy the texture data to system memory and send the pointer right back to the graphics card. This process is not only slow, it incurs the additional headache of clamping all floating-point values to the range [0.0,1.0].
Vertex shaders reading from textures will be supported in vs 3.0.

- Quantitative limitations

The ATI 9700 pro supports a maximum of four render targets. A vertex shader program can therefore have a maximum of four outputs. Each shader can execute a maximum of 64 instructions per-pixel or vertex. Additional instructions require a separate rendering pass. The simulation's collision-detection shader, for example, has *exactly* 64 instructions. Many other shaders had to be split up and reorganized in order to accommodate this requirement.

The 9700 does not fully support single-precision floating point textures. It provides a maximum of 24 bits of floating point precision, introducing additional rounding error for sensitive simulations.

The card has a limitation of four levels of texture indirection in a shader. This did not affect anything I tried to implement, but will be a factor in several of the proposals outlined in Future Work. Lack of support for certain texture addressing modes (i.e. border) also proved to be a nuisance.

2.3. Implementation

2.3.1. Spring Model

Cloth was simulated as a series of point masses connected by massless structural & bending springs. The total force on each cloth point was computed as:

$$\vec{f}_{total} = \vec{f}_{gravity} + \sum_{\substack{i=0 \\ i \neq j}}^{N} \vec{f}_{spring(i,j)}$$

For N connected springs. Spring(i,j) represents the spring from i to j. Spring forces were calculated using Hooke's spring law:

$$\vec{f}_{spring(i,j)} = -k_s(|\vec{p}_i - \vec{p}_j| - dl_{spring(i,j)}) \bullet (\vec{p}_i - \vec{p}_j) + f_{damping(spring(i,j))}$$

Where p_i is the point position, and *dl* represents the default length of the spring.
Two separate damping methods were implemented:
1) Damping in the spring-direction (as described in Matyka04):

$$f_{damping(spring(i,j))} = -k_d * ((\vec{v}_i - \vec{v}_j) \bullet (\vec{p}_i - \vec{p}_j)) * (\vec{p}_i - \vec{p}_j)$$

2) Artificial viscosity (as described in Desbrun01)

$$f_{damping(spring(i,j))} = -k_d * dt * (\vec{v}_i - \vec{v}_j)$$

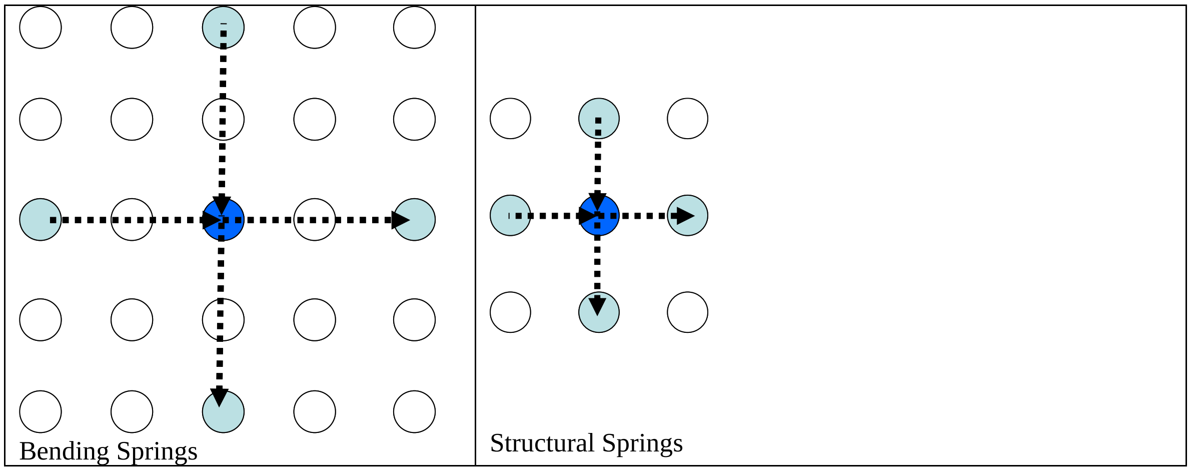
Acceleration was calculated using Newton's second law.

2.3.2. Data Storage

All per-cloth-point and per-spring data is stored in 32-bit floating-point textures of dimensions [w,h], where w and h are the number of cloth points in the x and y directions, respectively. This could easily be generalized to non-rectangular, arbitrary meshes by reducing the dimension of the textures to 1 and using an additional level of indirection (see Future Work).
The simulation is computed by repeatedly rendering a screen-aligned rectangle mesh over the render target texture. The mesh is rendered using a variety of pixel shaders that perform force calculation, explicit integration, collision detection, weighting, etc.
The mesh's texture coordinates are scaled to the range [0-0.5/n,1.0.5/n] where n is the number of pixels in that dimension. This guarantees that input/render operations will be performed at the pixel's center. All image filtering, depth checking, etc. is turned off.

To achieve maximum frame rate, the challenge was to write pixel shaders that were as comprehensive as possible, generating as many outputs as possible (max 4), while remaining under the 64-operation limit. Using input data is costly- reading data from a 4-component float texture uses up four clock cycles, whereas any vector math operation only uses one.

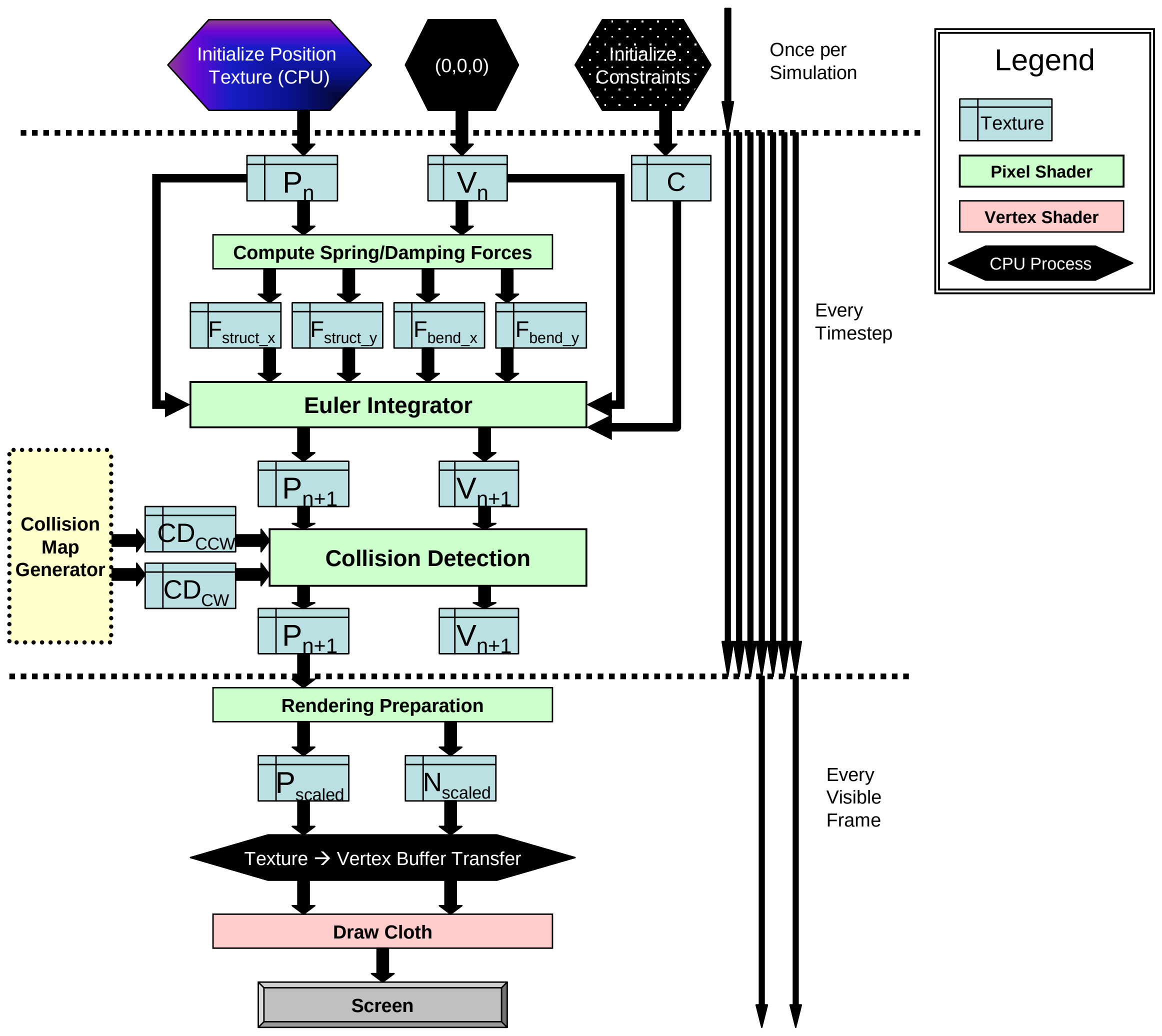
Using the rectangular model, the springs connected to a given cloth point are implicitly defined by the point's interpolated texture coordinate. The orientation (forward or backward) of the spring force relative to the point can also be determined. There are eight springs connected to each internal cloth point. All springs of a certain type in a given direction are stored as a texture: Horizontal structural springs, Vertical structural springs, Horizontal bending springs, Vertical Bending Springs.



Since texture coordinates are automatically clamped or mirrored, additional checks were needed to ensure that a cloth point on an edge didn't include nonexistent springs. These checks would not be necessary if the hardware supported "border" texture addressing, which would allow all spring forces to be automatically set to zero if a texture was addressed with coordinates outside of the range [0,1].
Shearing forces, which would require an additional two passes to satisfy the render target and operation count limit, were not implemented.

2.3.3. Stream Processing

The diagram below illustrates the simplest simulation process, for the Euler solver. The process has additional textures and steps when other solvers are used. Collision detection has a separate pixel/vertex shader set described later. The constraints texture is represented by the letter "C". To simplify the diagram, it is shown at a higher level of abstraction: texture reference swapping is not shown.



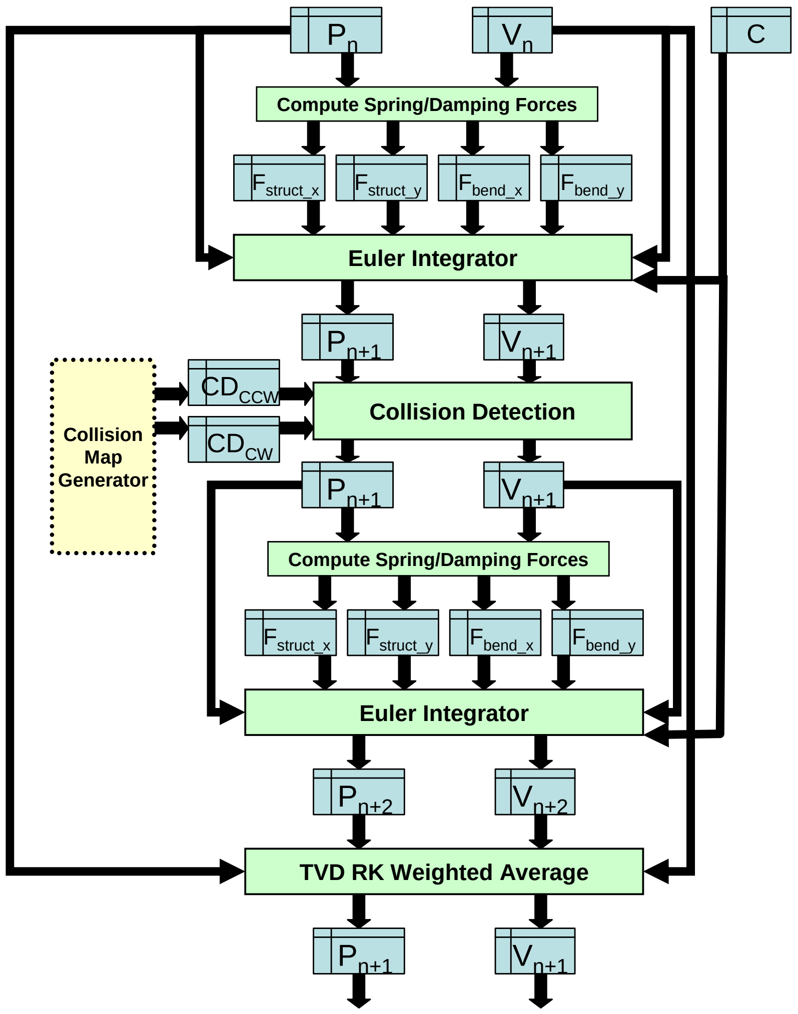
2.3.4. Solvers

I implemented several solvers using this model. All solvers compute $\vec{f}_{total}$, and check the constraints texture to determine if the current point should be moved at all. Acceleration is computed using Newton's second law.

<u>Euler</u> is the most straightforward solver:

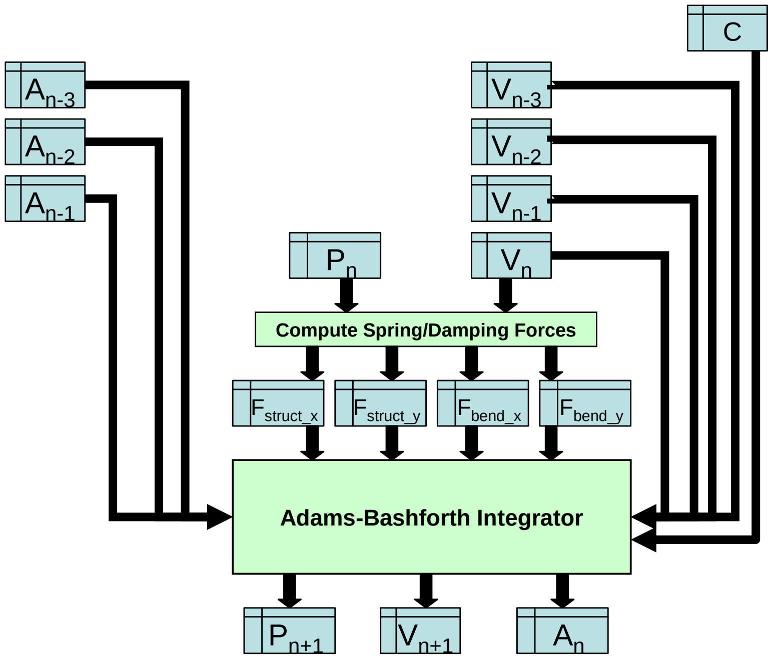
$$\vec{v}_{n+1} = \vec{v}_n + dt * \vec{a}_n$$
$$\vec{p}_{n+1} = \vec{p}_n + dt * \vec{v}_n$$

<u>2nd-Order-Accurate TVD Runge Kutta</u> is computed using the Euler integration pixel shader and a second, averaging pixel shader that combines the results. The following diagram replaces the "Euler Integrator" box in the previous figure:



It's easy to see how this model could be expanded to a higher order RK scheme such as RK4.

Adams Bashforth uses two of its own pixel shaders, as well as seven additional textures to store previous acceleration and velocity textures. For the first three updates, a modified Euler pixel shader is used. This Euler pixel shader renders Acceleration as well, and the thus the rendered Acceleration and Velocity textures are used to fill the initial A_{n-x} and V_{n-x} textures.



I also worked on a Verlet integrator. I realized from reading several papers that "Verlet integrator" includes a variety of different algorithms. I used the version described by (Jacobson03) that appears to be used in most game engines. The previous position (P_{n-1}) is stored explicitly, and is subtracted from the current position instead of storing velocity.

2.3.5. Collision Detection

Like the rest of the simulation, collision detection is performed entirely on the GPU.

A good volume of GPU collision research is being performed at UNC. [Govindajaru03] describes an interesting GPU-based collision-detection algorithm. After reading the paper, it appears that this scheme is *loosely* GPU-based. Triangles are rendered over and over again in forward and reverse order, and the CPU constantly performs occlusion queries on the graphics card to determine if the previously rendered triangle/surface is in a "Potentially Colliding Set". From these queries, the CPU determines which groups of triangles have potentially collided and reduces the set and repeats until it has identified colliding triangles. This algorithm doesn't quite fit with the GPU-only mantra of the project.

I decided to write a simpler algorithm that would work for the majority of arbitrary geometry that one would want to collide with cloth in a video game (projectiles, characters, vehicles, etc). I wanted to simulate friction with moving surfaces, to generate a sphere-spinning-cloth-twisting animation similar to the one created for [Bridson02].

I originally thought of using a single depth map, but it's clear that this method won't even allow a piece of cloth wrap around a sphere.

The "collision scene" is rendered in two passes using a vertex and pixel shader (with depth testing enabled). The vertex shader computes both the current and previous world transform for the current vertex, and determines velocity of the vertex from the difference. This velocity is placed in the first 3 coordinates of the vertex color output. The fourth coordinate holds the Z-value of the vertex in world coordinates. Finally, the vertex shader transforms the vertex's world position into collision-map-render-space. The video card interpolates these values across the triangle and the pixel shader renders it to a collision map. In the first pass, the scene is rendered using back-face culling. This renders the object faces closest to the cloth start point. In the second pass, the scene is rendered using front-face culling, which renders the faces on the other side of the object(s).

Collision detection is performed by projecting the cloth position into collision-map-texture-space (different from render-space due to a DirectX convention). The Z coordinate in texture space is sampled in both the "front" and "back" collision maps. If the current cloth z-position lies between these two, there's a collision. The closer collision map (front/back) is determined, and the point is pushed back out by computing the normal from a gradient of the collision map. The cloth point is assigned the velocity stored in the collision map, thereby moving it with the colliding surface. More interesting collision-dynamics could be simulated (accurate coefficients of friction, etc.), but this would incur disproportionate computational cost of additional passes as the collision detection shader clocks in at exactly 64 instructions.

2.3.6. Rendering/Shading

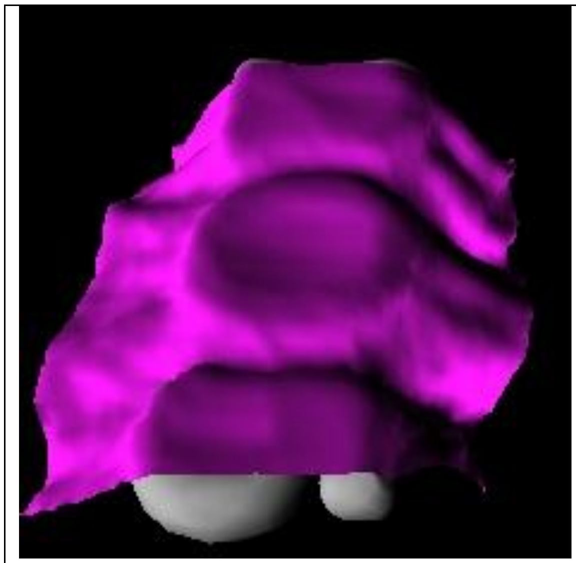
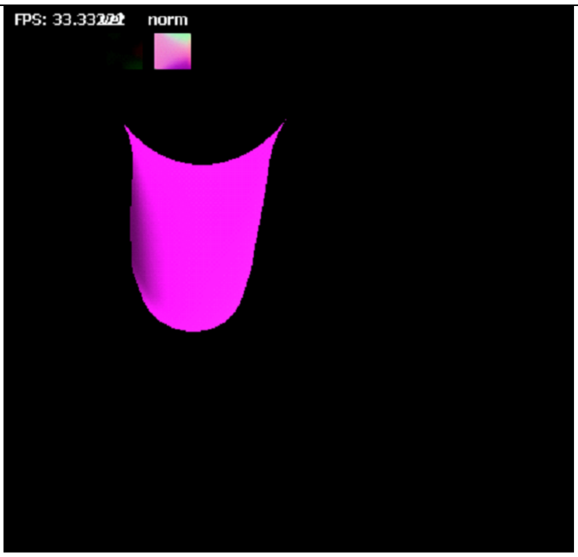
This is the only point in the process where the CPU gets involved. In order to be fed to the vertex buffer, the position data must first be copied to a location in system memory. This same pointer is immediately recast and passed directly to the vertex shader. Unfortunately, this process also clamps all floating-point values to [0.0,1.0], restricting the cloth to a very small box. To overcome this, a "pre-rendering" pixel shader, which runs once per visible frame, scales the position data to this range. This "pre-rendering" shader also computes the cloth surface normals at each cloth point.

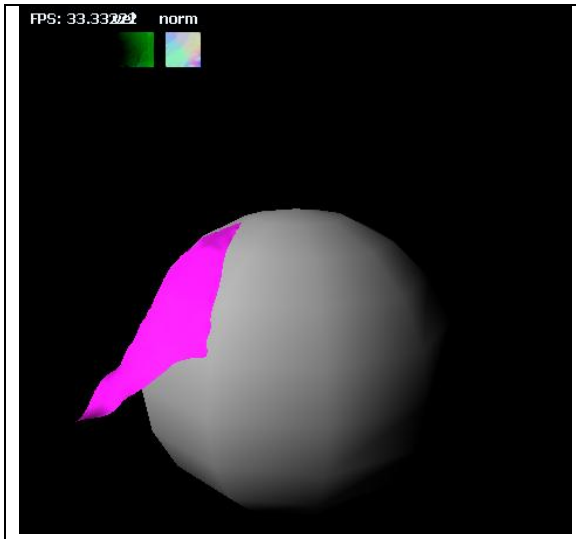
The vertex buffer is un-scaled and transformed to screen coordinates by a vertex shader, which also computes per-pixel diffuse lighting.

The process of copying the texture to system memory and drawing to screen is the slowest part of the process. To alleviate the problem, multiple timesteps are computed between each visible frame. At each frame, the length of time from the start of the last visible frame is calculated, and divided by the maximum timestep to determine the number of iterations to perform before displaying the next frame.

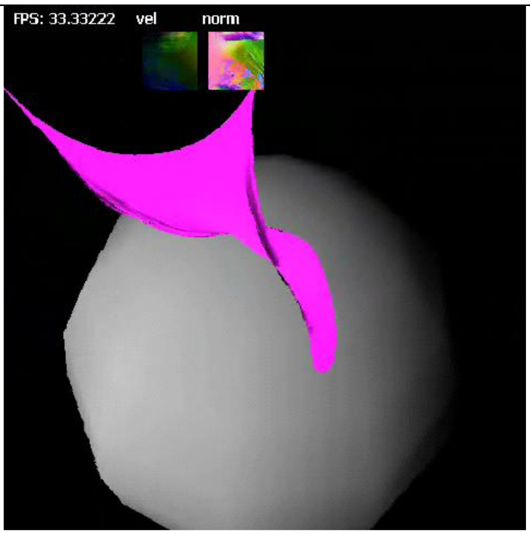
3. Results/Analysis

When I first got the program to produce some cloth-like motion a couple weeks ago, I was astounded at the framerates I was observing. They left CPU-based cloth simulation programs I'd seen in the dust. In the weeks following, I hammered the GPU with more complex cloth, collision detection, and multi-pass integrators and I'm still getting great visual results at high framerates.
The ATI 9700 Pro All-In-Wonder does more than push a ridiculous number of triangles: It also includes a TV-Tuner card. I originally intended to connect the card's video output back into the input to record results in real-time. As could be expected, however, recording to a file while rendering significantly reduced framerate. Eftychis preferred the results not arrive on videotape, so I rendered the results frame-by-frame into a video file. Actual real-time frame rates for these simulations are listed:

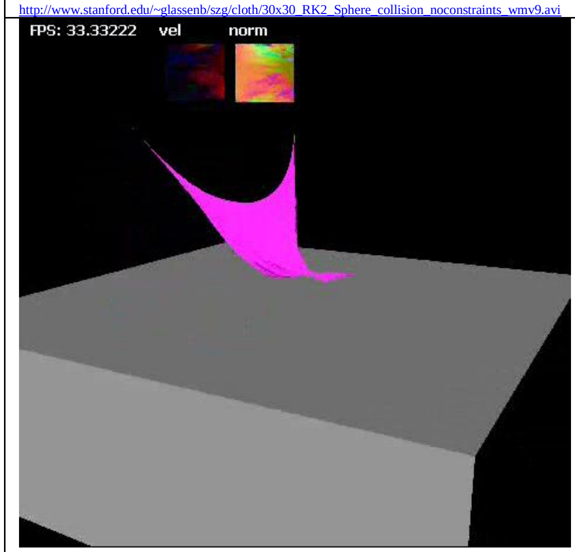
 2500 Cloth Points TVD-RK2 Integration **Collision Map Dimension:** 200 **Timestep:** 0.001 **FPS (real-time):** 41 (**Euler**), 15.5 (**TVD-RK2**) Video Link: http://www.stanford.edu/~glassenb/szg/cloth/50x50_RK2_205_collision_wmv9.avi	 900 Cloth Points Adams-Bashforth Integration **No Collision** **Timestep:** 0.0003 **FPS (real-time):** 28 **Artificial Viscosity** **Video Link:** http://www.stanford.edu/~glassenb/szg/cloth/30x30_AB_ArtificialVisc_0.0003.avi



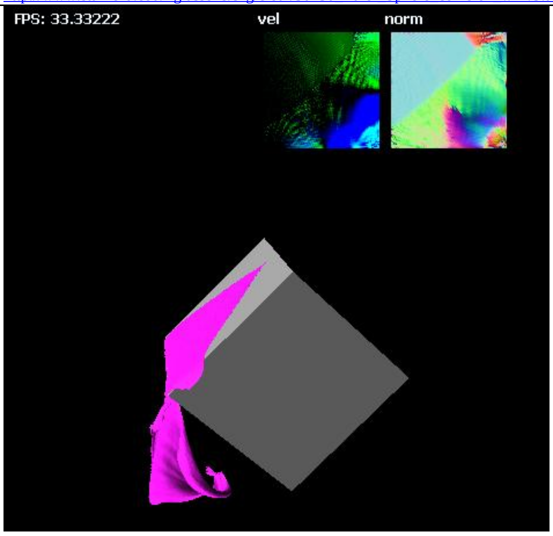
900 Cloth Points
TVD-RK2 Integration
Timestep: 0.001
Collision Map Dimension: 100
FPS (real-time): 32
Video Link:
http://www.stanford.edu/~glassenb/szg/cloth/30x30_RK2_Sphere_collision_noconstraints_wmv9.avi



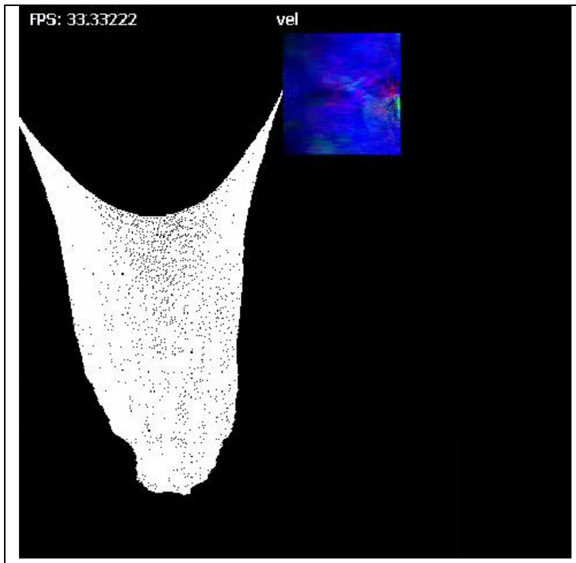
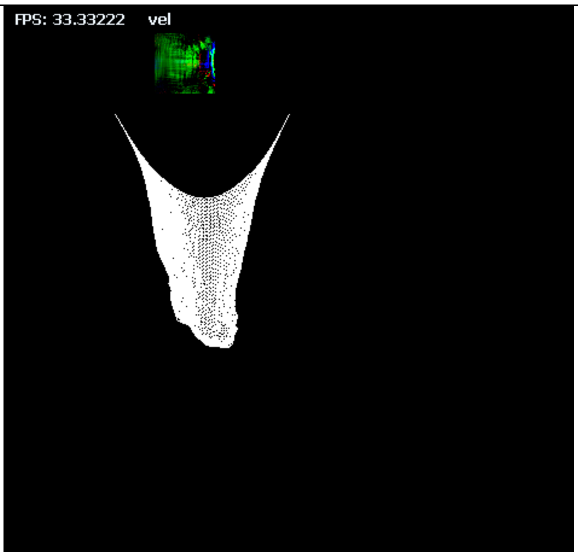
2500 Cloth Points
Euler Integration
Timestep: 0.002
Collision Map Dimension: 100
FPS (real-time): 82
Video Link:
http://www.stanford.edu/~glassenb/szg/cloth/50x50_Euler_Sphere_collision_wmv9.avi



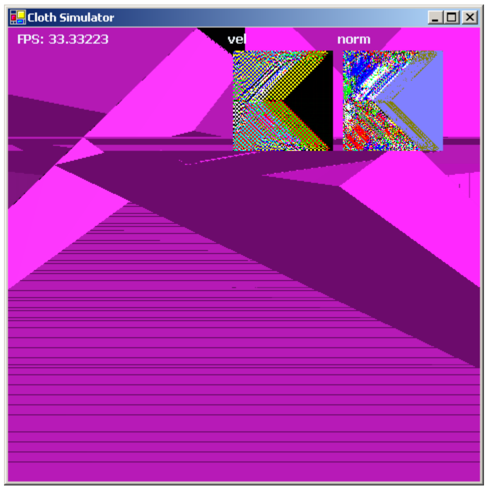
2500 Cloth Points
Euler Integration
Timestep: 0.001
Collision Map Dimension: 100
FPS (real-time): 28
Video Link:
http://www.stanford.edu/~glassenb/szg/cloth/50x50_Euler_box_collision.mpg



10,000 Cloth Points
Euler Integration
Timestep: 0.001
Collision Map Dimension: 100
FPS (real-time): 9
Video Link:
http://www.stanford.edu/~glassenb/szg/cloth/100x100_Euler_SmallBox.mpg

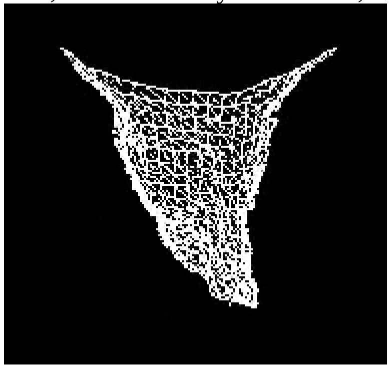
 10,000 Cloth Points Euler Integration **Timestep:** 0.001 **No Collision.** **FPS (real-time):** 42 **k_d = 15** **k_s = 500** **Low Gravity** **Video Link:** http://www.stanford.edu/~glassenb/szg/cloth/100x100_Const_Euler_kd15_ks500_lograv_wmv9.avi	 2,500 Cloth Points Euler Integration **Timestep:** 0.0005 **No Collision.** **FPS (real-time):** 83 **k_d = 15** **k_s = 500** **Video Link:** http://www.stanford.edu/~glassenb/szg/cloth/50x50_NoConst_Euler_kd15_ks500_nocollision_wmv9.avi

With updates on the order of 1000 per second, the stability of a particular simulation can be determined quickly.
The following video illustrates this phenomenon. The timestep in this scenario (0.0008) was too large for **Adams-Bashforth**:



(Video Link: http://www.stanford.edu/~glassenb/szg/cloth/100x100_Const_AB_dt0.0008_wmv9.avi)

It is clear from both damping equations that stability is heavily affected the damping coefficient. One advantage of artificial viscosity over spring damping is that its stability region is unaffected by anything but the timestep and k_d. It is observed that in situations where the simulation is close to the edge of the stability region, an increase in the gravitational force will cause divergence in the spring-damped system. This does not occur when artificial viscosity is used. Nonetheless, artificial viscosity produces significantly less realistic results. With both equations, when the damping coefficient is set close enough to zero, neutral stability is observed, and high-frequency oscillations can be seen:



(Video Link: http://www.stanford.edu/~glassenb/szg/cloth/Low_kd.avi)

In most cases, the stability region for RK2 was only slightly wider than Euler's. In a simulation with 900 cloth points, ks=150, g=-9.8, and a set timestep of 0.001, TVD-RK remained stable for a damping coefficient up to 11. Euler's stability region extended to 9.

Adam Bashforth's stability region was much smaller. For a damping coefficient of 10, Euler remained stable as long as the timestep stayed below 0.00093 and TVD-RK2 below 0.00104. Adams-Bashforth lost stability if the timestep went above 0.00014. The measured stability threshold ratio between Euler and Adams-Bashforth is very close to the calculated one: Measured: 0.00093/0.00015 = 6.2. Calculated: $-2/(-3/10) \approx 6.66$.

Finally, one of the advantages of storing all physics data in textures is the interesting patterns that result. The following is an animated velocity texture for a 10,000-point piece of cloth constrained at two points without collision detection:

(Link: http://www.stanford.edu/~glassenb/szg/cloth/100x100_vel_nocollision_wmv9.avi)

4. Future Work

4.1. Implicit Integration

So far all ODE solvers implemented are purely explicit. Although it would certainly not be trivial to implement an implicit method that runs in the GPU, prior research indicates that it is certainly possible. What's questionable is whether second-generation hardware like the ATI 9700 will be able to perform the necessary computations in real-time.

[Desbrun01] describes a predictor-corrector method that integrates linear forces (predictor) and corrects for angular momentum. This algorithm doesn't look conducive to GPU processing, due to its dependence on reductions.

It would be more interesting to implement something along the lines of the mixed Explicit/Implicit time integration method described in [Bridson03]. This method limits implicit integration to damping forces, but allows for a much larger timestep. The question is whether the overhead of a GPU **conjugate gradient** solver outweighs this gain.

A dense CG solver, although straightforward to implement, would be out of the question due to its computational cost. Results from [Moravanszky03] indicate that the time to complete a dense matrix multiply of a matrix of dimension 1000 is almost one second. According to [Bridson03], "typically less than ten and sometimes as few as one or two [CG] iterations are needed" for each timestep in the model, making real-time calculation with a dense solver out-of-the question.

A sparse solver is more promising. A cloth damping matrix would have only as many nonzero elements in a row as connected springs, whereas the dimension is equal to the total number of cloth points. [Bolz03] describes a sparse GPU-CG solver implemented using a reasonable level of indirection. Their results indicate that the GPU-CG solver *should* be able to perform 110 iterations per second, but driver limitations significantly reduce actual performance. As a result, all performance statistics mentioned in the paper are normalized as if the limitations did not exist; actual performance stats on the hardware are not provided.

I spoke with Ian Buck about this, but he was also unsure about the performance of CG-GPU solvers. He did, however, show me how development of these stream-based algorithms is much more straightforward in Brook, a new high-level shading language being developed at Stanford.

4.2. Inverse Dynamics Constraints/Artificial Damping

This would be straightforward to implement in the existing framework and would provide additional realism and stability. [Matyka03] classifies restricting spring length to a certain percentage past its default length as "Inverse Dynamics Constraints". Goeff Irving strongly disagrees, ascribing the term "artificial damping" to this phenomenon. Either way, calculating position-shifts due to this constraint could be performed when spring forces are calculated and stored in a separate texture. This would require splitting spring force calculation into multiple passes, which would be necessary anyway to simulate shearing forces (another potential stabilizer).

4.3. Extension to Arbitrary Cloth Meshes

The results of this project indicate that simulation performance is more significantly affected by the number of passes (indicated by the timestep and method used), rather than the number of cloth points. By adding an additional level of indirection and reducing the number of texture dimensions to one, this simulator could be extended to work for arbitrary meshes (shirts, dresses, etc.) The cloth point position texture would be a one-dimensional array. Instead of implicit spring/cloth point associations, a single 1D spring texture would contain texture coordinates pointing to two cloth point locations the cloth position texture (as well as storing the default length of the spring). Another texture set (at least two would be necessary, possibly three) would hold pointers to connected springs for each cloth point. The relation of the point to the spring force (forward or backward) could be stored as the sign of this 1D texture coordinate – Mirrored texture address mode could be used to automatically correct for the sign.

4.4. Spline Surface Interpolation

Instead of performing costly physics calculations on a large number of cloth points, a good visual result could be achieved by using a smaller number of cloth points and spline-interpolating between these points using a vertex shader as described in [Dunlop02].

4.5. Wind

By calculating surface normals at every timestep as opposed to every visible frame, these values could be used to calculate wind force against the cloth surface.

5. Acknowledgements

In addition to the CS205 teaching staff, I'd like to thank Mike Mandel, Ian Buck, Josh Bao, Goeffrey Irving, and Neil Molino for taking the time to meet with me to discuss cloth simulation and GPU programming.

6. References

1) Engel, WF et al. “Direct3D ShaderX”. Wordware Publishing. Texas, 2002.
2) “DirectX 9.0 Programmer's Reference”. Microsoft Corporation 2003.
3) Heath, MT. “Scientific Computing: An Introductory Survey”. Second Edition. McGraw-Hill. New York, 2002.
4) Miller, T. “Managed DirectX 9 Graphics and Game Programming”. Sams Publishing. Indianapolis, 2004.
5) Matyka, M. “Inverse Dynamic Displacement Constraints in Real-Time Cloth and Soft-Body Models” in *Graphics Programming Methods*. Charles River Media. Hingham, 2003.
6) Desbrun, M et al. “Interactive Animation of Cloth-like Objects for Virtual Reality”. The Journal of Visualization and Computer Animation, Volume 12, Issue 1, May 2001, pages 1-12.
   Online: http://www-grail.usc.edu/pubs/DMB_chapterBook.pdf
7) Jacobson, T. “Advanced Character Physics”. Gamasutra. 2003.
   Online: http://www.gamasutra.com/resource_guide/20030121/jacobson_04.shtml
8) Govindajaru, NK et al. “CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware”. In *Graphics Hardware*. ACM SIGGRAPH/Eurographics 2003.
   Online: http://www.cs.unc.edu/~geom/CULLIDE/cullide.pdf
9) Bridson, R et al. “Simulation of Clothing with Folds and Wrinkles”. Eurographics/SIGGRAPH Symposium on Computer Animation, 2003.
   Online: http://graphics.stanford.edu/papers/cloth2003/cloth.pdf
10) Moravanszky, A. “Dense Matrix Algebra on the GPU”. In ShaderX2 Programming. 2003.
    Online: http://www.shaderx2.com/shaderx.PDF
11) Bolz, J et al. “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”. SIGGRAPH 2003
    Online: http://www.multires.caltech.edu/pubs/GPUSim.pdf
12) Buck, I. “Data Parallel Computation on Graphics Hardware”. Presentation at Graphics Hardware, 2003.
    Online: http://graphics.stanford.edu/~ianbuck/GH03-Brook.ppt
13) Dunlop, R. “Synthesizing Patches Using Vertex Shaders”. 2002. Online: http://www.mvps.org/directx/articles/shadeland/splinevshade3.htm
14) Bridson, R et al. “Robust Treatment of Collisions, Contact and Friction for Cloth Animation”. SIGGRAPH 2002, ACM TOG 21, 594-603 (2002).
    Online: http://graphics.stanford.edu/~fedkiw/papers/stanford2002-01.pdf
15) Fedkiw, R. “CS205: Class Notes”. Stanford University, 2003.
    Online: http://www.stanford.edu/class/cs205/