**Simple Cloth Simulation with OpenGL**

**Graphics Final Project Report**

2015년 6월 20일

|  |  |  |
| --- | --- | --- |
| 팀 구성 및 개발 환경 | | |
| Team Name | Bokhak Brothers (복학형제) | |
| Members | 김태규  CSE, 20100472  sunburn51037 | |
| 이다열  CSE, 20090402  loving2days | |
| 최준혁  CSE, 20100234 | |
| Environments | Visual Studio | 2013 |
| OpenGL | 4.4 |
| GLSL | 4.4 |
| GLEW | 1.12.0 |

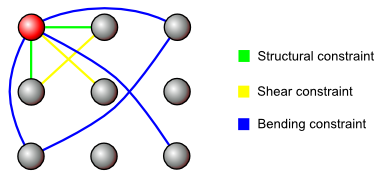
1. 개요

우리 팀은 최종 프로젝트 과제로 Cloth Simulation을 선택하였다. Cloth Simulation은 천의 움직임을 그래픽스 프로그램으로 나타내는 것으로, 그래픽스 분야에서 어려운 문제 중 하나로서 꾸준히 연구되어 왔다. 우리 팀은, Cloth Simulation의 전 과정과 원리에 대하여 이해하고, 여러 자료들을 바탕으로 이를 구현해보는 것을 목표로 최종 프로젝트를 진행하였다. 그 결과, Cloth Simulation의 기본적인 모델인 mass-spring 모델을 완벽히 이해하여, 구현에 성공하였으며, 바람, 중력과 같은 external force와 구, 바닥과의 collision detection, 그리고 마찰력과 간단한 그림자를 구현하였다. 추가적으로, Skybox mapping과 Texture mapping, normal mapping 등을 통하여 오브젝트를 보다 현실적으로 보이게 하고자 노력했다. 또, Xavier Provot [1]에 의해 제안된 Dynamic Inverse Procedure를 통하여 비현실적으로 늘어나는 cloth를 보정하는 과정을 추가하여 cloth simulation을 현실적으로 보이도록 만들었다. 마지막으로 간단한 점광원 projection을 이용하여 천의 그림자까지 구현해보았다. 본 프로젝트를 통하여 Cloth Simulation에 대해 잘 이해할 수 있었으며, GPGPU에서 OpenGL shader program을 만드는 전 과정을 통해 그래픽스 프로그래밍에 대한 포괄적인 이해를 할 수 있었다.

2. 프로그램 설계

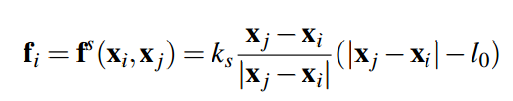
**(1) Internal Forces**

Cloth Simulation에는 Mass-Spring 모델, Energy based 모델, Yarn based 모델 등 많은 모델이 사용되지만, 가장 직관적이고 쉬운 방법은 mass-spring 모델을 사용한 구현이다. 그래서 우리 팀은 mass-spring 모델을 선택하였다. 기본적인 내용은 [2]를 참고하였다.

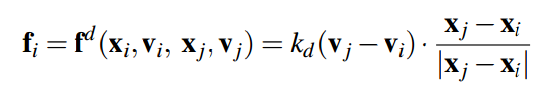


Mass spring 모델은 천을 격자로 배치된 particle, 즉 질량 덩어리(mass)들로 보고, 그들 사이에 spring이 있어서 서로 힘을 주고 받는 것으로 보고 표현한 것이다. 천 내부에서 particle들끼리 작용하는 힘을 spring의 특성에 근사시키는 방법인데, 위와 같이 structural force, shear force, bending force가 있다. 이는 천의 특성으로부터 비롯된 것인데, 천이 늘어나거나 기울어지거나, 접히면 원상복귀 하려는 성질을 모두 mass들 끼리의 spring force로 표현한 것이다.

Spring force는 아래와 같이 나타난다.



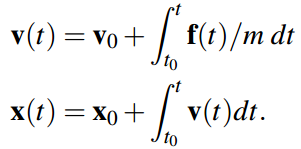
이 때, spring force는 두 mass의 위치에 관한 함수로 나타낼 수 있고, 탄성계수와 길이 변화량의 곱의 크기를 가지며, 길이 변화의 반대방향을 방향벡터로 갖는 힘이 된다.



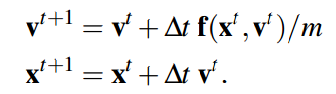
현실 세계에서는 위와 같은 반발력(damping force)가 작용한다. Mass-spring 모델에서의 반발력은 위와 같이 이웃 vertex의 속도와 상대적 위치, 그리고 반발 계수를 이용해서 표현할 수 있다. 최종적인 힘은 아래와 같이 반발력과 탄성력의 합으로 나타난다.



그렇다면 mass들의 위치를 어떻게 힘을 이용하여 구할까? Mass의 위치는 초기위치부터 시간 t에 대한 함수로 나타낼 수 있다. 뉴턴의 공식 에서 나타나듯, 힘은 가속도를 발생시키고, 가속도는 속도를 변화시킨다. 시간에 따른 속도 함수를 적분하여 초기위치에 더해주면 시간 t에서의 위치를 구할 수 있다.



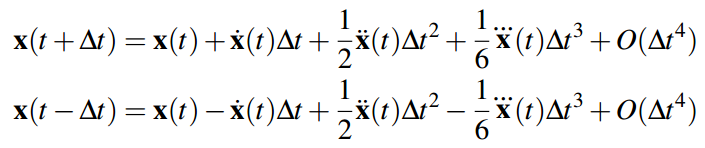
그러나, 구현시에는 적분을 하는 것이 좋은 방법이 아니다. 위와 같은 경우 매번 적분 값을 계산해야 할 뿐만 아니라, v(t)와 f(t) 역시 전혀 예측이 불가능하기 때문에 아래와 같이 discrete 한 time step을 정하고, 해당 time step 뒤의 위치만 반복적으로 계산하여 위치를 구할 수 있다.



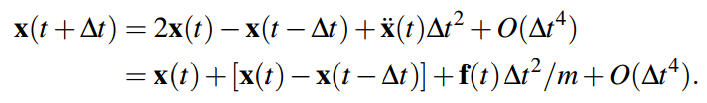
하지만 이 역시 문제가 존재한다. Position과 Speed를 매 time step마다 update해주어야 하는 문제가 있다. 따라서 우리는 Verlet Integration을 사용하였다. Euler Integration, Verlet Integration, Runge-Kutta Method 등 많은 방법이 있지만, Real time application에서는 verlet integration이 가장 많이 쓰인다. 정확도는 Runge-Kutta Method보다 크지 않지만, 매우 간단하고 빠르게 계산할 수 있기 때문에 cloth simulation에 적합하다고 판단하여 사용했다. 원리는 아래와 같다. 위 식으로부터



를 각각 계산하면 아래와 같다.



양 변을 더하면



위와 같은 식이 얻어진다. 이 때 항은 가 작을 경우에 무시할 수 있을 정도로 작은 항들이다. 따라서 이와 같은 항을 제외하고 양 변을 정리하면 다음과 같은 식을 얻을 수 있다.





우리는 position만 매번 update하면 되기 때문에 최소한의 memory write를 수행하게 된다.

**(2) External Forces**

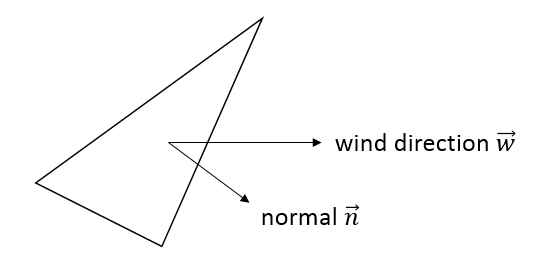
외부에서 작용하는 힘은 크게 중력과 바람의 힘, 그리고 마찰력이 있다.

1) 중력

중력은 비교적 간단하게 계산할 수 있다.

중력은 질량과 중력가속도의 곱으로 나타낼 수 있으며, 중력 역시 반발력(damping force)를 가진다. 이 때 반발력은 공기의 저항과 같은 값으로, 속도가 빨라질수록 큰 값을 가지게 된다.

2) 바람



바람의 경우, face normal과 바람의 방향 간의 각도가 작을수록 바람의 세기가 크다는 아이디어에서 착안했다. 하지만, 이것 만으로는 천 표면에서 생기는 유체의 예측 불가능한 움직임을 표현할 수 없었다. 따라서, 기본적으로는 normal의 방향에 영향을 받지만, 거기에 랜덤한 값을 더해주어 불규칙적인 펄럭임을 표현했다. 바람 역시 반발력이 있는데, 이 경우 반발력이라기 보다는 vertex의 속도가 바람의 속도와 비슷해질수록 당연히 바람으로부터 받는 힘이 줄어든다는 것에서 착안한 상쇄값으로 이해하였다. 최종적인 바람의 힘은 아래와 같이 나타난다.

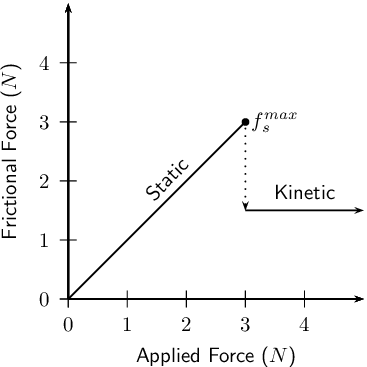


3) 마찰력

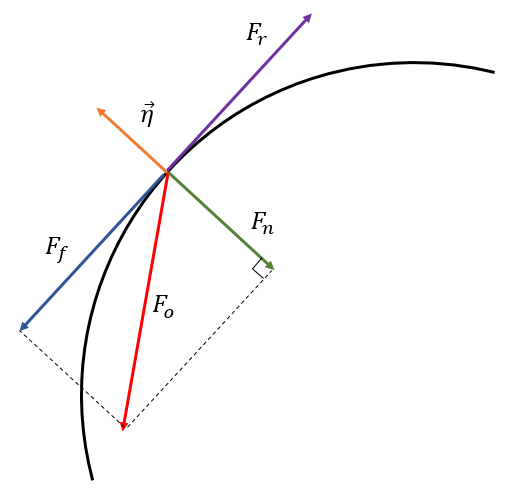
마찰력은 collision이 일어난 vertex에 한하여만 특징적으로 나타나는 힘이며,

위와 같은 경우 물체가 받는 힘을 상쇄시킨다. 는 마찰력이 상쇄하는 대상이 되는 힘이고, 은 수직 항력, 는 마찰 계수이다.

인 경우 최종 작용하는 힘은 로 나타난다. 마찰력의 경우, 아래와 같이 Static과 Kinetic의 경우 서로 다른 힘의 크기를 보이는데, 이러한 현상은 무거운 물체에서 두드러지게 나타나는 현상이기 때문에 우리의 cloth simulation에서는 제외하도록 한다.



우리가 구현한 collision 대상인 구는 normal이 vertex 위치에 따라 다르기 때문에, normal과 tangent 방향을 vertex마다 따로 계산하도록 했다. Normal과 tangent 벡터는 각각 아래와 같이 나타난다.



위와 같은 그림에서, 마찰력, 즉 수직항력과 마찰계수의 곱이 tangent 벡터 방향으로의 힘보다 클 경우 vertex에 작용하는 힘은 0이 된다. 따라서 최종적으로 표면의 vertex에 작용하는 힘 는 아래와 같이 구할 수 있다.

)

**(3) Collision Detection**

Collision은 크게 두 가지, 즉 구(sphere)와 바닥(floor)와의 collision을 구현했다. Collision을 구현하기 본래는 inverse dynamics를 이용해서 반발력을 구하여 force에 더해주어야 한다. 그러나, 본 프로젝트에서 구현한 마찰력과 중력, 그리고 internal force간의 복잡한 계산으로 인하여 collision을 반발력을 통해 구현할 경우 컨트롤하기가 어려워지는 문제가 있었다. 값을 바꾸면서 실험한 결과, collision으로 인한 반발력이 가장 컨트롤을 어렵게 하는 것을 깨달았다. 따라서 우리는 비교적 간단한 방법을 사용하기로 결정했고, 아래와 같은 방법(position correction)으로 collision을 구현했다.

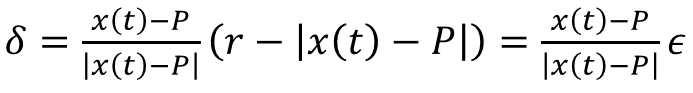
1) Detection

구의 중심부와 vertex의 위치 간의 거리가 구의 반지름보다 작으면 vertex가 polyhedron과 충돌했음을 알 수 있다.



2) Response

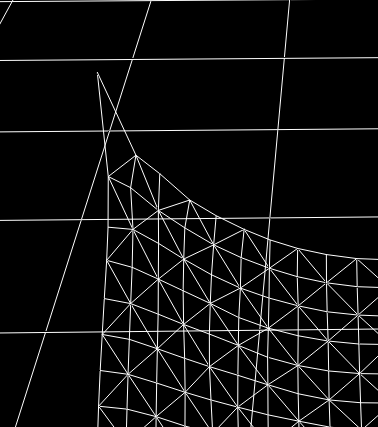
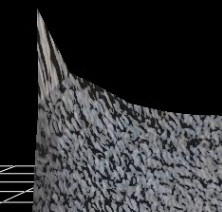
아래와 같이 vertex 위치를 교정하여 해결할 수 있다.



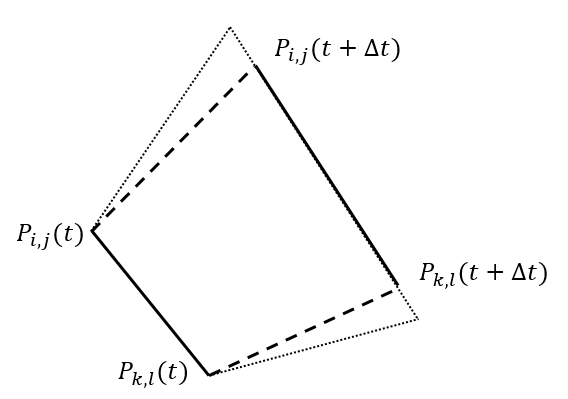
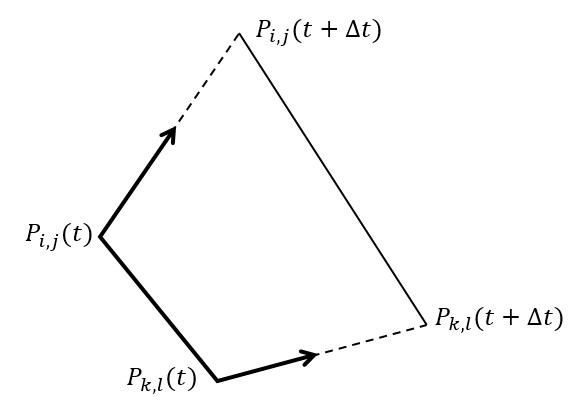


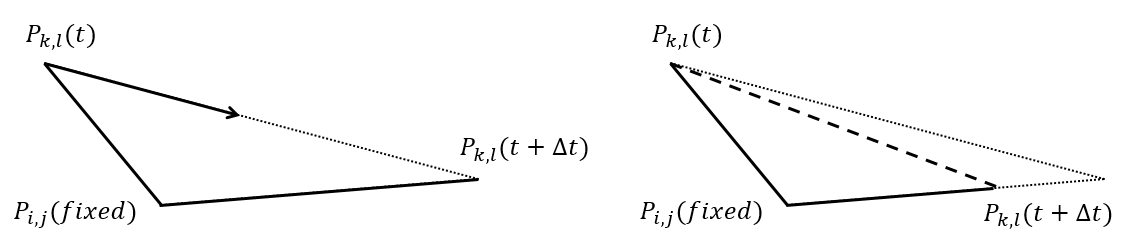
**(4) Deformation Constraint**

Mass-spring 모델에서 큰 문제점 중 하나는 mass-spring이 cloth와 정확하게 일치하는 물리적인 성질을 가지지 않는다는 것이다. 그 중 하나로 super elastic effect가 있다. 이것은 cloth가 지나치게 탄력있는 모습으로 나타나는 문제이며, 특히 가상의 pin을 이용하여 cloth를 나타냈을 경우 아래와 같이 pin 부근에서 천이 과도하게 늘어나는 현상이 나타난다. 하지만, 실제 cloth는 탄성으로 늘어날 수 있는 범위가 10% 미만으로 알려져 있다.



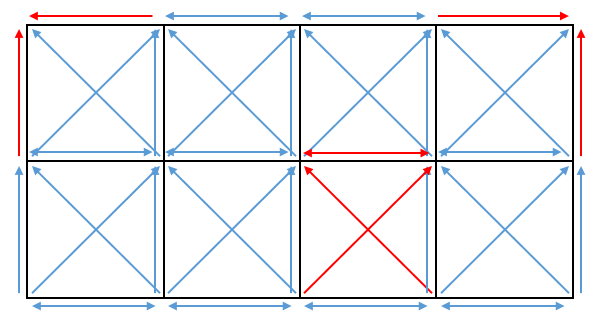
위와 같은 문제를 해결하는 방법의 하나로 Xavier Provot이 제안(1995)한 Dynamic Inverse Procedure를 사용했다. [1] 이 방법은 비교적 간단한 방법으로, mass의 위치가 결정된 후에 과도하게 늘어난(Super-elongated) spring을 원상복귀 하는 procedure를 호출하는 방법이다.





위의 procedure은 중요한 가설을 바탕으로 하는데, 그것은 spring이 과도하게 늘어났을 경우, spring이 늘어난 위치와 orientation은 정확하다는 가설이다. 이와 같은 가설을 바탕으로 free mass는 서로 절반씩 가깝게 조절되어 super elongated spring을 없애고, fixed mass의 경우에는 그 이웃 mass에 대해서만 가깝게 조절되어 원상복귀시키면 된다.

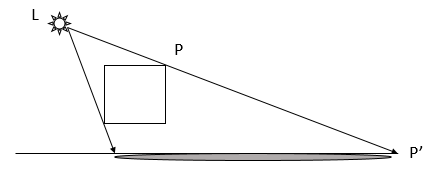
이를 우리의 cloth 모델인 mesh 모델에 적용해야 하는데, 우리는 pin을 양 쪽 끝에 두 개 가지고 있으므로, 아래와 같이 설계했다.



즉, pin의 양 옆 neighbor가 되는 mass는 pin 방향으로만 inverse vector를 더해주고, pin이 위 방향에만 존재하므로, 모든 vertex가 수직으로는 위 방향으로만 inverse vector가 적용된다. 그 외의 다른 것들은 수평 방향으로 절반씩 가깝게 inverse vector가 적용되게 한다.

**(5) Shadow**

Cloth에 대한 그림자는 Global Illumination이 아닌 점광원에서의 Projection에 의한 연산을 이용했다. 아래의 그림처럼 점광원의 위치를 L이라 하고, Object의 한 Vertex의 위치를 P, 지면에 Projection된 그림자의 위치를 P’이라 하면, 아래와 같은 두 점을 이용한 직선의 방정식을 만들 수 있다.

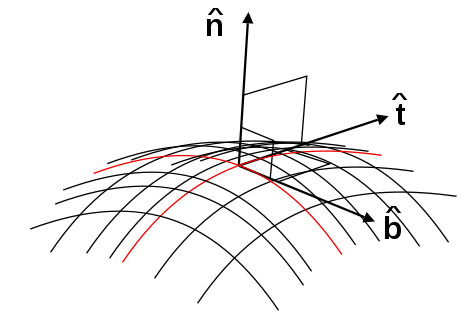


**그림 1::Projection of Point Light Source onto Surface**

P’의 z 값은 평행한 지면으로 가정한다면 이미 상수 값이고, P는 값이 주어져 있으므로, 지면에 Projection된 x, y값을 위 식으로부터 도출할 수 있다.

**(6) Calculate Normal**

각 Particle의 Normal을 구하기 위해서 해당 Particle의 주변 Particle 위치 정보로 Normal을 근사할 수 있다. 아래의 그림은 한 점에서의 Normal, Tangent, Bitangent의 관계를 나타낸 것이다.



**그림2::Relation of Tangent, Bitangent and Normal**

각 Vector들은 서로 수직관계에 있으므로, Tangent와 Binormal을 구하면 두 Vector를 기반으로 Cross Product를 통해 Normal Vector를 구할 수 있다. Tangent와 Binormal은 각각 해당 점에서 주변 점으로의 Vector로 근사하여 구할 수 있다.

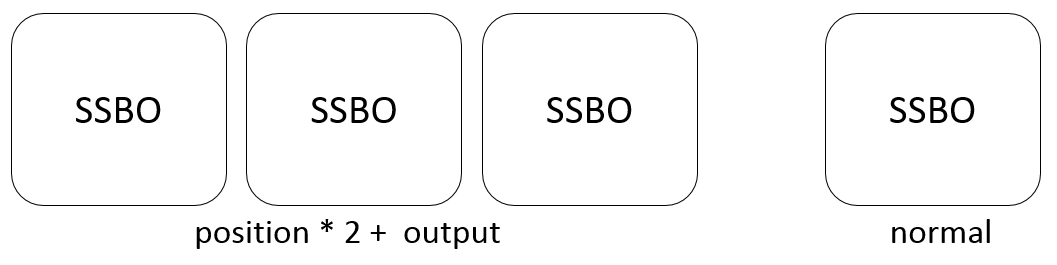
**(7) Improving Reality**

조금 더 사실적인 시뮬레이션을 위해 Color Texture Mapping, Skybox Mapping, Normal Mapping 등의 기법을 사용했다. Color Texture는 천, 공, 지면 등의 Object에 사용했으며, Skybox Mapping은 여섯 개의 배경 이미지를 전개도처럼 나타내어(Top, Bottom, Left, Right, Front, Behind) Cube에 Mapping한다. Normal Mapping은 축구공과 천의 질감을 나타내기 위해 사용했다.

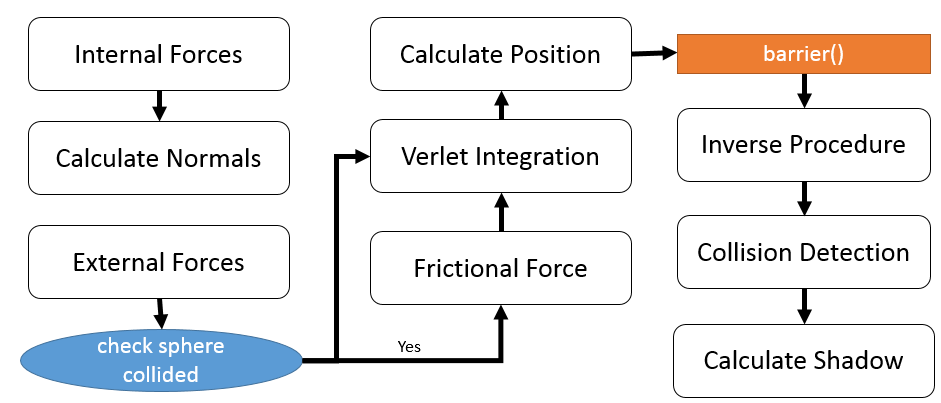
3. 프로그램 구현

**(1) Mass-Spring model with GLSL**

위의 설계를 바탕으로 Mass-spring model을 바탕으로 한 cloth simulation을 구현하였다. 구현을 위해 GLSL 4.3 이상에서 지원하는 Compute shader와 Shader storage buffer object(SSBO)를 이용했다. Compute shader는 GPGPU에서 사용되는 general한 연산을 하기 위한 것으로 rendering과 직접적인 관련이 없는 어떠한 연산이던지 수행할 수 있는 shader type이다. Compute shader를 이용하여 mass의 위치를 계산했고, 그 위치를 저장하기 위하여 SSBO를 사용했다. SSBO는 writable buffer이며, compute shader에서 rendering shader(vertex shader, fragment shader)로 vertex 위치를 전달할 수 있다. Verlet Integration 과정에서 Vertex 현재 위치와 전 time step에서의 vertex 위치가 필요하기 때문에 SSBO 세 개를 사용해서 Role rotation을 하도록 했다. 이 구현의 기본적인 아이디어는 [3]에서 얻을 수 있었다. 또, normal mapping 등을 구현하기 위하여 cloth mass들의 normal 값을 저장하기 위한 SSBO를 추가로 하나 더 사용했다.



Shader Program의 최종적인 구현은 아래와 같은 구조로 구현했다.



우선 Internal force를 계산한 뒤에, 현재 position을 바탕으로 normal을 계산한다. 그 뒤에, normal을 이용해서 wind force를 계산하고 중력을 더해준다. Vertex가 구와 collision이 일어난 경우, 마찰력을 더해주고, Verlet Integration을 통해 포지션을 계산한다. 그 뒤에, 모든 mass가 업데이트 되도록 sync를 맞춘다. 이것은 compute shader의 barrier() 함수 호출로 가능하다. 모든 mass가 업데이트 되면, dynamic inverse procedure를 이용하여 super-elastic effect를 해결하고, collision detection과 response를 한 뒤에, 최종 position을 바탕으로 shadow 프로젝션을 계산한다.

자세한 구현은 아래와 같다.

1) Internal force

2) Calculate normal

아래의 <코드 2>는 compute.glsl에서 파티클의 Normal을 구하는 코드를 나타낸 것이다. 해당 파티클의 위치에서 네 인접한 파티클의 방향으로 Tangent, Bitangent Vector를 만들고 Cross Product를 한 후에 Normal Vector에 값을 더하는 과정으로 Normal을 근사하여 구했다.

|  |
| --- |
| // Calculate Normal  vec3 normal = vec3(0.0, 0.0, 0.0);  vec3 tangent;  vec3 bitangent;  if(col < colmax - 1)  {  tangent = normalize((vertexCurrBuffer[vertexIndex + 1] - vertexCurrBuffer[vertexIndex]).xyz);  if(row < rowmax - 1)  {  bitangent = normalize((vertexCurrBuffer[vertexIndex + rowmax] - vertexCurrBuffer[vertexIndex]).xyz);  normal += normalize(cross(bitangent, tangent));  }  if(row > 0)  {  bitangent = normalize((vertexCurrBuffer[vertexIndex] - vertexCurrBuffer[vertexIndex - rowmax]).xyz);  normal += normalize(cross(bitangent, tangent));  }  }  if(col > 0)  {  tangent = normalize((vertexCurrBuffer[vertexIndex] - vertexCurrBuffer[vertexIndex - 1]).xyz);  if(row < rowmax - 1)  {  bitangent = normalize((vertexCurrBuffer[vertexIndex + rowmax] - vertexCurrBuffer[vertexIndex]).xyz);  normal += normalize(cross(bitangent, tangent));  }  if(row > 0)  {  bitangent = normalize((vertexCurrBuffer[vertexIndex] - vertexCurrBuffer[vertexIndex - rowmax]).xyz);  normal += normalize(cross(bitangent, tangent));  }  }  normalBuffer[vertexIndex] = normalize(vec4(normal,1)); |
| **코드 2: compute.glsl->calculate normal in main function** |

3) External forces

외력은 중력과 바람을 도입하여 compute shader에서 계산하도록 구현했다. 바람의 경우, 현재 파티클의 Normal과 상수로 고정된 바람의 방향을 나타내는 Vector의 Dot Product로 세기를 구하고 바람의 방향을 나타내는 Vector의 크기를 조절하는 양으로 사용했다. 또한 중력은 mass와 중력상수의 곱으로 나타낼 수 있는데, 아래의 <코드3>은 바람과 중력을 구하는 함수를 나타낸 것이다.

|  |
| --- |
| vec4 windForce(vec4 normal)  {  vec4 n = normal;  return dot(n, WIND) \* WIND;  }  void main()  {  ..  ..  float rnd = rand(vec2(0.1, 12)) \*0.25 ;  if (isWind > 0)  {  force += rnd \* windForce(normalize(vec4(normal, 1))) + vel \* DEFAULT\_DAMPING;  }  ..  ..  force += GRAVITY \* mass + vel \* GRAVITY\_DAMPING;  } |
| **코드 3: compute.glsl->calculate wind force and gravity** |

4) Frictional force

5) Verlet integration

6) Calculate position

7) Inverse procedure

8) Collision detection

이번 시뮬레이션은 충돌 처리를 하기에 가장 쉽고 간단한 도형인 구를 이용한 충돌 감지를 간단한 예제를 참고하여 구현했다. [4] 구의 충돌은 물리학적인 법칙을 쓰지 않고 비교적 쉽게 구할 수 있는데, 한 파티클의 위치가 구 내부에 위치하는 경우 구의 중심에서 현재 파티클의 위치를 가리키는 Vector로 구의 표면까지 밀어내는 원리로 구현했다. 아래의 <코드 8>은 충돌처리를 compute.glsl에서 구현한 것을 나타낸 것이다. 바닥의 충돌 처리도 간단한 연산을 통해 계산했는데, 바닥인 z=-1 평면보다 파티클의 z값이 작을 경우에 이전 상태의 위치로 돌려놓는 방식으로 구현했다. 이 방법들은 실제적인 물리학 법칙을 사용하지 않은 간단한 구현으로, 어느 정도 사실적인 결과를 보장하지만, 마찰계수나 다른 인자들의 값에 따라 극단적인 결과를 낳는 경우도 존재했다.

|  |
| --- |
| //collision with sphere  vec3 d0 = next.xyz - vec3(sphereX, 0, 0); // center of sphere vec3(0, 0, 0)  float dist = sqrt(d0.x \* d0.x + d0.y \* d0.y + d0.z \* d0.z);  vec4 X0 = inv\_sphere \* next;  if (dist < radius)  {  d0 = (radius - dist) \* d0 / dist;  // Transform the delta back to originala space  vec3 d = vec3(0.0, 0, 0);  vec3 transformInv = vec3(sphere[0].x, sphere[1].x, sphere[2].x);  transformInv /= dot(transformInv, transformInv);  d.x = dot(d0, transformInv);  transformInv = vec3(sphere[0].y, sphere[1].y, sphere[2].y);  transformInv /= dot(transformInv, transformInv);  d.y = dot(d0, transformInv);  transformInv = vec3(sphere[0].z, sphere[1].z, sphere[2].z);  transformInv /= dot(transformInv, transformInv);  d.z = dot(d0, transformInv);  next += vec4(d, 1);  vertexOutBuffer[vertexIndex] = vec4(next.xyz, 1);  }  else  {  if (movable)  vertexOutBuffer[vertexIndex] = next;  }  if(vertexOutBuffer[vertexIndex].z < -1)  {  vertexOutBuffer[vertexIndex] = vertexCurrBuffer[vertexIndex];  } |
| **코드 8: compute.glsl-> collision detection** |

9) Calculate shadow

그림자는 Global Illumination을 이용하지 않고 점광원에서의 바닥에 대한Projection을 사용하여 구현했다. Particle의 경우 Compute Shader를 거치기 때문에 비교적 쉽게 그림자를 계산할 수 있지만, 구, 막대 등은 다른 Shader를 이용했기 때문에 편의상 천에 대해서만 그림자를 구현했다. 아래의 <코드 9>는 compute.glsl에서 그림자의 위치를 계산하는 함수와 이용 예시를 나타낸 것이다. 점광원의 위치와 현재 Particle의 위치로 직선의 방정식을 세우고 z=-1일 때의 x, y값을 구하여 그림자의 위치를 계산했다.

|  |
| --- |
| shadowBuffer[vertexIndex] = calcShadow(vertexOutBuffer[vertexIndex]);  vec4 calcShadow(vec4 pos)  {  vec4 light = vec4(0, 0, 5, 1);  vec4 diff = light - pos;  float x = light.x;  float y = light.y;  if(diff.z != 0)  {  x = diff.x \* (-1 - light.z) / diff.z + light.x;  y = diff.y \* (-1 - light.z) / diff.z + light.y;  }  return vec4(x, y, -0.98, 1);  } |
| **코드 9: compute.glsl->calculate shadow** |

**(2) Improving Reality**

조금 더 사실적인 묘사를 위해 Texture를 이용한 Mapping으로 Object들을 나타내었다. 여기에 사용된 기법은 Color Texture Mapping, Normal Mapping, Cube Map인데 Color Texture, Normal Mapping은 지난 Assignment4에서 이미 사용한 방법이므로 생략한다. Cube Map의 경우 Skybox를 나타내기 위해 사용한 것으로, World를 커다란 직육면체로 감싸고 내부를 6개의 배경 Texture로 Mapping하는 기법인데, Viewing 방향, 위치에 따라 다르게 보이기 때문에 Viewing Dependent Mapping중 하나에 속한다. 배경을 구현하기 위해 간단한 예제를 이용했다. [5] 아래의 <코드 10>은 Object Class를 상속받아 Skybox Class를 생성하고 Cube Mapping을 구현한 것을 나타낸 것이다.

|  |
| --- |
| void Skybox::draw()  {  mvstack.push(model\_view);  if (controller.isWireframe == false)  {  skyShader.Use();  {  glActiveTexture(GL\_TEXTURE7);  glEnable(GL\_TEXTURE\_CUBE\_MAP);  glUniform1i(skyShader("Skybox"), 7);  glBindTexture(GL\_TEXTURE\_CUBE\_MAP, texture);  glUniformMatrix4fv(skyShader("ModelView"), 1, GL\_TRUE, model\_view \* Translate(0, 0, 0) \* RotateX(90));  glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer);  unsigned int bufferIndex = 0;  glVertexAttribPointer(skyShader["vPosition"], 4, GL\_FLOAT, GL\_FALSE, 0, BUFFER\_OFFSET(bufferIndex));  glDrawArrays(GL\_TRIANGLES, offset, 36);  glActiveTexture(GL\_TEXTURE7);  glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);  glDisable(GL\_TEXTURE\_CUBE\_MAP);  }  skyShader.UnUse();  }  model\_view = mvstack.pop();  } |
| **코드 10: Skybox::draw()** |

Skybox는 skyShader를 이용하는데, Cube Map은 3D Mapping이며, Texture Coordinate가 곧 Position과 동일하기 때문에 다른 Shader와 함께 쓰기에 불편함이 많았다. 따라서 skyVert.glsl, skyFrag.glsl라는 두 Shader를 새로 생성했다. 아래의 <코드 11>은 두 Shader를 나타낸 코드를 나타낸 것이다.

|  |
| --- |
| ---------------------------skyVert.glsl-------------------------------  #version 440  in vec4 vPosition;  out vec3 texCoords;  uniform mat4 Projection;  uniform mat4 LookAt;  uniform mat4 ModelView;  void main()  {  gl\_Position = Projection \* LookAt \* ModelView \* vPosition;  texCoords = vPosition.xyz;  }  ---------------------------skyFrag.glsl-------------------------------  #version 440  in vec3 texCoords;  out vec4 fColor;  uniform samplerCube Skybox;  void main()  {  fColor = texture(Skybox, texCoords);  } |
| **코드 11: Skybox Mapping Shader** |

4. 프로그램 실행 화면

5. 결론

6. 개선 방향

아쉬운 점 중 하나로는, Collision response를 반발력을 이용해 자연스럽게 표현하지 못한 점이다. 사실 Cloth Simulation 자체가 어려웠던 이유 중 하나는 다양한 반발계수와 탄성계수, 마찰계수 등 값을 잘 설정하지 않으면 time step 동안 하나의 mass가 지나치게 강한 힘을 받아서 불안정하게 튀어버리는 현상 이었다. 되도록이면 큰 time step을 사용하여 최대한 자연스럽게 보이게 하고자 했지만, collision의 경우에는 그 부분을 잘 해결하지 못하여 위치 교정만을 이용해서 표현했던 것이다. 이 부분을 개선한다면 좀 더 realistic한 cloth의 움직임을 구현할 수 있지 않을까 생각한다.

또, Large Time step을 사용하는 것과 관련된 논문이 있었으나 자세히 읽어보지는 않았는데, large time step을 이용한 방법도 연구하여 적용하면 좋을 것 같다.

7. 참조문서 및 코드

[1] Provot, X. (1995, May). Deformation constraints in a mass-spring model to describe rigid cloth behaviour. In *Graphics interface* (pp. 147-147). Canadian Information Processing Society.

[2] <http://matthias-mueller-fischer.ch/realtimephysics/coursenotes.pdf>

[3] <https://github.com/McNopper/OpenGL/tree/master/Example40>

[4] Open Cloth: <https://code.google.com/p/opencloth/>

[5] Cube Map: <http://learnopengl.com/#!Advanced-OpenGL/Cubemaps>