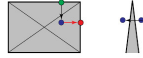


experimental haptics

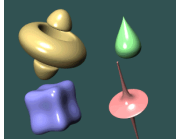
“Haptic Rendering with the God-Object, Proxy, and Friction”

### Why another haptic rendering algorithm?

- Potential fields work with lots of objects, but the “pop through” and force discontinuity effects make it feel unrealistic



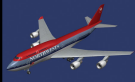

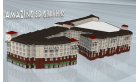



- Implicit methods are pretty robust, but it is difficult to design (and graphically render!) complex shapes using implicit functions



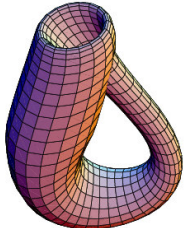
### What do we want in an algorithm?

- Feels right (no popping through or force discontinuities)
- Looks right (don't see avatar going into objects)
- Works with objects of arbitrary shapes
- Fast (for realistic and stable force feedback)
- Haptics is closely coupled with graphics – it would be great to be able to leverage the extensive work from this more established field
  - Graphics cards are designed to optimize rendering of triangular meshes
  - Many modeling programs generate triangular meshes (3D Studio Max, Maya, etc.) and many existing models are in this format (<http://www.3dsafe.com>, etc.)

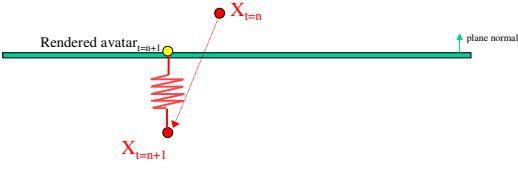
### More about triangular meshes

- Since a triangle has three vertices, and three points define a plane, no concerns about non-planar faces (such as when using four-noded surfaces)
- This also allows you to deform objects on the fly by just moving vertices
- The mesh only defines a surface, which may or may not enclose a volume; no simple “inside/outside” test; if the surface mesh does not surround a closed region, there may not even be a such thing as “inside” or “outside”



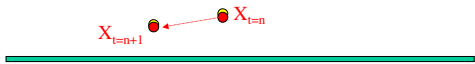
Klein bottle: What's inside and outside?

### Intuition : Virtual Wall



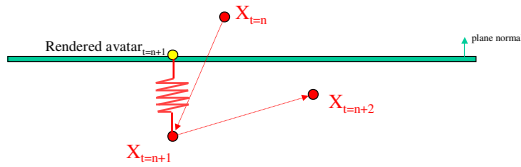
- Device position ( $x$ ) moves from above the plane ( $x_{t=n}$ ) to below it ( $x_{t=n+1}$ )
- Graphically, it would be better to render the avatar where it “should be” – as close as possible to the actual device position, without penetrating the plane
- Haptically, we could render a spring between the actual device position and the graphically rendered avatar, trying to “pull” the device back to where it “should” be
- Force magnitude depends on depth of penetration and stiffness constant of plane (“spring” constant)
- In this case, essentially the same haptic result as with a potential field

### Intuition : Virtual Wall



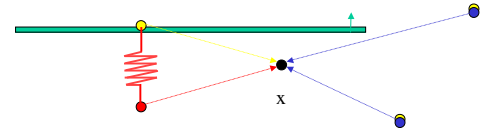
- When device is moving in “free space”, without contacting the plane, the avatar should be collocated with device position

## Intuition : Virtual Wall



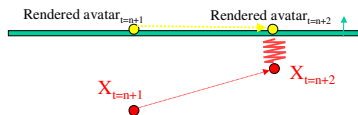
- At next iteration, both current device position ( $x_{t=n+2}$ ) and previous device position ( $x_{t=n+1}$ ) are below the plane
- How do you know if a force should be rendered?
  - In general, no simple inside/outside test
  - Line segment between previous device position and current device position doesn't cross plane
  - Depends on path history not just position or previous position; if you had started at  $x_{t=n+1}$  instead of  $x_t$ , there in fact shouldn't be a force at  $x_{t=n+2}$

## History Matters



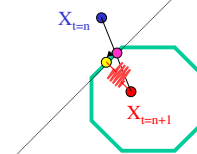
- Whether a force is rendered when the device is at position  $x$  (black dot) depends on how it got there: if from red dot, there should be a force, but if from either blue dot, there should not
- Need to keep track of history, but do you need to remember the whole path?
- Can't just look at previous position of device; in all three cases here, the path from the previous device position to current device position doesn't cross the surface
- Use the "history" stored in the form of the avatar position rendered at the last iteration
  - Note: The haptics loop runs faster than the graphics loop, so this avatar position might never have been actually rendered graphically
- If and only if the surface prevents the avatar at the previous iteration from reaching the current device position should there be a force

## Move the avatar



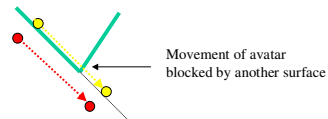
- Update the avatar position to again be as close as possible to the device position without crossing the plane
- Avatar and device will continue to be on opposite sides of the surface, and a force will continue to be generated, until contact is broken
  - Device moves back above surface
  - Device moves to right or left of surface

## Convex Object



- Only one surface stands between previous and current device positions, and no other surfaces impede movement along the plane of that surface
- Basically just need to consider one "virtual wall" at any given time

## Concavities

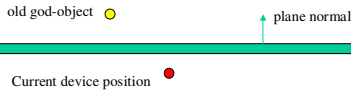


- You find the surface intersected by the line segment between the previous avatar and current device positions and want to minimize the distance of the new avatar position from the current device position without crossing this surface
- However, another surface may impede movement of the avatar along the first surface
- Main idea: want to locally minimize distance between avatar and device position subject to the constraints of the surfaces impeding motion

## God-object algorithm

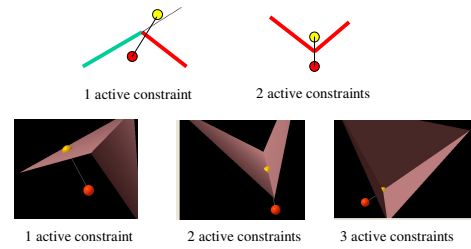
- "A Constraint-based God-object Method For Haptic Display"
- C.B. Zilles and J.K. Salisbury
- *Dynamic Systems and Control*, 1994
- "God-object" is the rendered avatar discussed in previous slides; we have complete freedom to put it wherever we want so as to follow the laws of physics
- Can model more complex effects (such as friction) just by controlling the god-object position

## Active Constraints : Infinite Surfaces



- An infinite surface constraint (a plane) is *active* if a line segment from the old *god-object* (rendered avatar) is located a positive distance from the surface (in the direction of the surface normal) and the device position has a negative distance to the surface
- Signed point-plane distance:  $D = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$
- This one-way constraint is beneficial if you need to “escape” when you start inside an object
- Can make it a two-way constraint if you prefer by just requiring old god-object and device position to be on opposite sides

## Active constraints : Finite Planar Faces



- For non-infinite surfaces (such as triangle facets), we also require that the line from the old god-object to the new device position passes through the facet within all of its edges
  - Need a line segment – triangle mesh collision checker!
- In 3D, up to three constraints can be active at a given time (constrained to a point); “prune” constraints below other active constraints

## Minimization Problem

- If  $(x_p, y_p, z_p)$  is the device position, we want to find a new god-object location  $(x, y, z)$  that minimizes the distance (or energy in the virtual spring) to the device:

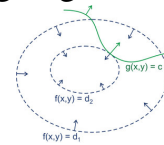
$$Q = \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2$$

subject to

$$\begin{aligned} A_1x + B_1y + C_1z - D_1 &= 0 \\ A_2x + B_2y + C_2z - D_2 &= 0 \\ A_3x + B_3y + C_3z - D_3 &= 0 \end{aligned}$$

- each constraint being the plane equation of one of the active constraints. If fewer constraints are active, the coefficients for the remaining constraints can be set to 0

## Lagrangian Multipliers



- Maximize (or minimize) some function  $f(x, y)$  subject to constraint  $g(x, y) = c$
- Extreme value will be where gradient of  $g$  is parallel to gradient of  $f$ :  $\nabla f(x, y) = \lambda \nabla g(x, y)$
- Generalizing to 3D and multiple constraints:

$$\nabla f(x, y, z) = \sum_{i=1}^n \lambda_i \nabla g_i(x, y, z)$$

Or, since signs of the Lagrange multipliers don't matter

$$\nabla f(x, y, z) + \sum_{i=1}^n \lambda_i \nabla g_i(x, y, z) = 0$$

This gives  $3+n$  ( $x, y, z, \lambda_1, \dots, \lambda_n$ ) unknowns; gradient gives 3 equations, plus  $n$  equations for the  $n$  constraints, so we have a solvable system of equations

## God-object Location Computation

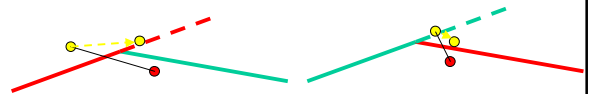
$$\begin{aligned} \nabla f(x, y, z) + \sum_{i=1}^n \lambda_i \nabla g_i(x, y, z) &= 0 \\ f(x, y, z) &= \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2 \\ g_1(x, y, z) &= A_1x + B_1y + C_1z - D_1 = 0 \\ g_2(x, y, z) &= A_2x + B_2y + C_2z - D_2 = 0 \\ g_3(x, y, z) &= A_3x + B_3y + C_3z - D_3 = 0 \end{aligned}$$

$$\begin{bmatrix} x - x_p \\ y - y_p \\ z - z_p \end{bmatrix} + \lambda_1 \begin{bmatrix} A_1 \\ B_1 \\ C_1 \end{bmatrix} + \lambda_2 \begin{bmatrix} A_2 \\ B_2 \\ C_2 \end{bmatrix} + \lambda_3 \begin{bmatrix} A_3 \\ B_3 \\ C_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & B_1 & C_1 & x \\ 0 & 1 & 0 & A_2 & B_2 & C_2 & y \\ 0 & 0 & 1 & A_3 & B_3 & C_3 & z \\ A_1 & B_1 & C_1 & 0 & 0 & 0 & D_1 \\ A_2 & B_2 & C_2 & 0 & 0 & 0 & D_2 \\ A_3 & B_3 & C_3 & 0 & 0 & 0 & D_3 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix}$$

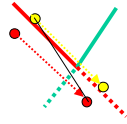
A linear system! Only 65 multiplies/divides to solve for  $x, y, z$  (only 33 when only 2 constraints, and 12 when only 1)

## Small Problem with Convex Objects



- In the first diagram, only the left surface is active
- Algorithm selects new god-object position to minimize distance to device position subject to the plane constraint
- Placement is not within the boundaries of the left surface
- On next iteration, the left surface will no longer be active, and the right surface will become active, so the god-object will “fall” back to the surface
- This distortion is generally imperceptible

## Big Problem with Concave Objects



- Only the red surface is active
- God-object crosses the other surface, and becomes free from it (next iteration won't detect an intersection with it)
- Solution: Iterate the process
  - Generate a *sub-goal* by minimizing the distance between the god-object and the device position subject to the active constraint
  - Check if line segment to this sub-goal intersects any other surfaces; if so, compute location that minimizes distance to this sub-goal subject to the new constraint(s)
  - Repeat until no new constraints are found
  - Have to repeat at most three times

## CHAI's Implementation

- create a segment between **god-object** and **goal position**.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position** and exit.
  - if collision occurs, move **god-object** to **collision point**.
- project **goal position** onto plane constraint (**goal position**).
- create a segment between **god-object** and new **goal position**'.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position**' and exit.
  - if collision occurs, move **god-object** to **collision point**'.
- project **goal position** onto plane constraint (**goal position**)' .
- create a segment between **god-object** and new **goal position**''.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position**'' and exit.
  - if collision occurs, move **god-object** to **collision point**''.
- **god-object** is now constrained at a single point.
- exit.

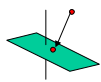
## Projecting a Point onto a Plane

- Given a point  $(p_x, p_y, p_z)$ , we want to find the closest point  $(x, y, z)$  on a plane  $Ax + By + Cz + D = 0$ .
- We could solve this using Lagrange multipliers again
  - Minimize

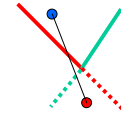
$$f(x, y, z) = (x - p_x)^2 + (y - p_y)^2 + (z - p_z)^2$$

subject to

$$g(x, y, z) = Ax + By + Cz + D = 0$$



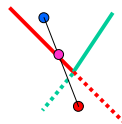
## CHAI's Implementation : Example (1)



2D Version of the algorithm (constraint lines instead of constraint planes; two lines constrain god-object to a point)

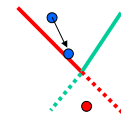
- create a segment between **god-object** and **goal position**.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position** and exit.
  - if collision occurs, move **god-object** to **collision point**.
- project **goal position** onto line constraint (**goal position**).
- create a segment between **god-object** and new **goal position**'.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position**' and exit.
  - if collision occurs, move **god-object** to **collision point**'.
- **god-object** is now constrained at a single point.
- exit.

## CHAI's Implementation : Example (2)



- create a segment between **god-object** and **goal position**.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position** and exit.
  - if collision occurs, move **god-object** to **collision point**.
- project **goal position** onto line constraint (**goal position**).
- create a segment between **god-object** and new **goal position**'.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position**' and exit.
  - if collision occurs, move **god-object** to **collision point**'.
- **god-object** is now constrained at a single point.
- exit.

## CHAI's Implementation : Example (3)

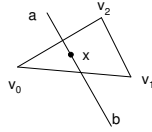


- create a segment between **god-object** and **goal position**.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position** and exit.
  - if collision occurs, move **god-object** to **collision point**.
- project **goal position** onto line constraint (**goal position**).
- create a segment between **god-object** and new **goal position**'.
- search for a collision between segment and environment.
  - if no collision occurs, move **god-object** to **goal position**' and exit.
  - if collision occurs, move **god-object** to **collision point**'.
- **god-object** is now constrained at a single point.
- exit.

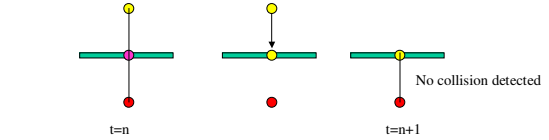


## Problem 1 : Need fast collision detection

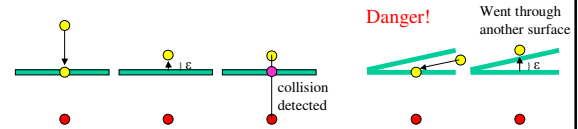
- There is a relatively simple geometric formula for line segment – triangle intersection test
- However, testing for intersection against all triangles at each iteration is much too costly for reasonably large meshes
- Hierarchies of bounding volumes can be used to greatly speed this up – stay tuned for Thursday's lecture



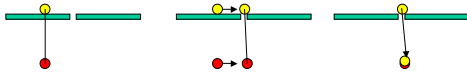
## Problem 2 : Numerical Problems



- If the god-object is placed exactly at the collision point, in the next iteration, due to inexact numerical computation, no collision may be detected
- Solution: Push the god-object out a small distance  $\epsilon$  along the surface's normal
- Danger: Don't push it across another surface! (Perhaps should do another collision test)



## Problem 3 : Small Holes

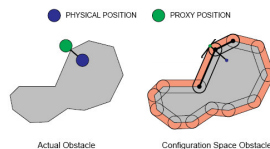


- In many 3D mesh models, vertices of triangles do not exactly match up, resulting in small holes between facets
- The god-object is just a point, so it can fall through an arbitrarily small hole

## Solutions to Small Holes Problems

- Open your model in 3D Studio Max and choose "Cap Holes"
  - Generally works well, but requires extra work, and may cap some holes you don't want capped
- Give the god-object a radius – the proxy sphere
  - Proposed by Diego Ruspini, Kraimir Kolarov, and Oussama Khatib in "The Haptic Display of Complex Graphical Environments" at SIGGRAPH 1997

## The Proxy



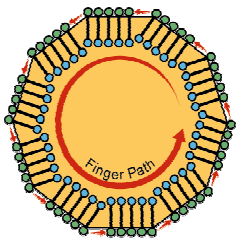
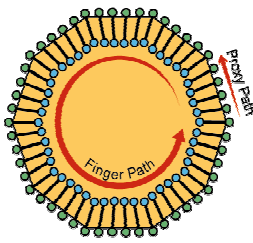
- Instead of testing for intersections between a line segment and the mesh, test for intersections between the cylindrical volume swept out by a spherical proxy moving along the line segment and the mesh
  - Quinlan's algorithm to compute shortest distance between two non-convex bodies
- Analogous to configuration space in robotic motion planning, in which the real obstacles are mapped to C-space obstacles, which include all points within one proxy radius of the obstacles, thus allowing the sphere to be modeled as a point in the configuration space

## Special Effects

- A wide variety of haptic "special effects" can be generated just by cleverly controlling the position of the god-object / proxy
  - Force shading
  - Texturing
  - Friction

## Force Shading

- Interpolate vertex normals across polygon to get continuous, smooth normals (just like Phong shading in graphics)

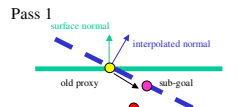



Faceted Cylinder
Shaded Cylinder

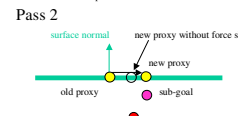
## Force Shading Algorithm

- Two passes
  - Pass 1: Use interpolated planes to determine a proxy position
  - Pass 2: Treat the proxy position found in pass 1 as the goal (projected back onto nearest actual surface if it is above them), and repeat proxy algorithm using the actual (non-interpolated) planes

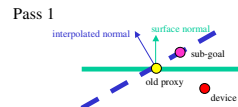
Pass 1



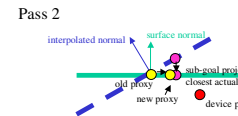
Pass 2



Pass 1




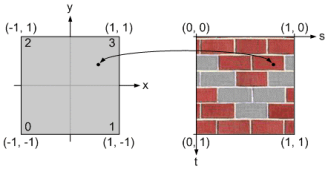
Pass 2



## Haptic Textures

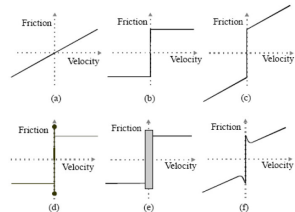
- Textures are frequently used in graphics to specify such values as color (texture mapping) and normals (bump mapping) with high precision over a surface
- Texture coordinates map position in an image to a position on a rendered surface
- Same idea can be applied to haptic properties
  - Have stiffness or friction coefficient vary across a surface in proportion to intensity of a grayscale image
  - Have components of the normals used for force shading vary across a surface in proportion to the RGB components of a color image





## Friction

- Common friction models:
 

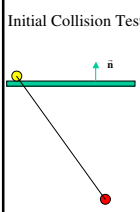

- We will consider stiction (d), with two states
  - Not enough tangential force to overcome static friction force (with coefficient  $\mu_s$ ), so proxy is stuck
  - Sufficient tangential force to overcome static friction force; proxy slips along surface, but is opposed by a (lesser) dynamic friction force (with coefficient  $\mu_d$ )

## Static Friction (1)

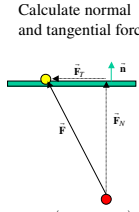
- Stick if there is not sufficient tangential force to “break away”:  $\|\vec{F}_T\| < \mu_s \|\vec{F}_N\|$
- Otherwise, proxy can slip along surface
- With the proxy / god-object, after finding a constraint plane, we calculate the normal (to the constraint plane) and tangential components of the force vector between the collision point and the device position
  - If stick condition holds, we do not move the proxy
  - Otherwise, we can slip along surface towards the goal (projection of haptic device position onto constraint plane)
- Can visualize a “friction cone” defined by rotating a right triangle with one side equal to the normal force component and an angle between this side and the hypotenuse of  $\alpha$ , where  $\alpha = \arctan(\mu_s)$

## Static Friction (2)

Initial Collision Test



Calculate normal and tangential force



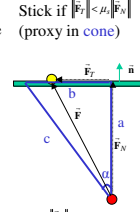
$$\vec{F} = k(\mathbf{x}_{proxy} - \mathbf{x}_{device})$$

$$\vec{F}_N = \left( \frac{\vec{F} \cdot \vec{n}}{\|\vec{n}\|} \right) \vec{n}$$

$$\vec{F} = \vec{F}_N + \vec{F}_T$$

$$\vec{F}_T = \vec{F} - \vec{F}_N$$

Stick if  $\|\vec{F}_T\| < \mu_s \|\vec{F}_N\|$  (proxy in cone)



$$\|\vec{F}_T\| < b$$

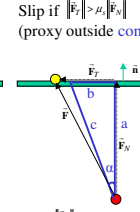
$$\tan(\alpha) = \frac{b}{a}$$

$$\tan(\arctan(\mu_s)) = \frac{b}{\|\vec{F}_N\|}$$

$$b = \mu_s \|\vec{F}_N\|$$

$$\|\vec{F}_T\| < \mu_s \|\vec{F}_N\|$$

Slip if  $\|\vec{F}_T\| > \mu_s \|\vec{F}_N\|$  (proxy outside cone)



$$\|\vec{F}_T\| > b$$

$$\tan(\alpha) = \frac{b}{a}$$

$$\tan(\arctan(\mu_s)) = \frac{b}{\|\vec{F}_N\|}$$

$$b = \mu_s \|\vec{F}_N\|$$

$$\|\vec{F}_T\| > \mu_s \|\vec{F}_N\|$$

