Allison Okamura
Autumn 2017

Lab Assignment 5:
Haptic Rendering

In this week's lab assignment, you will modify your program from last week in order to achieve rendering of compelling haptic virtual environments. This laboratory has several parts:

- Step 1: Render a virtual spring
- Step 2: Render a virtual wall
- Step 3: Render a virtual damper
- Step 4: Render a virtual texture using the damping approach
- Step 5: Render something else

Demonstrate each of your virtual environments to the instructor.

Step 1: Render a Virtual Spring

In this part of the lab, you will program a virtual spring. For this example, we will provide the code you need to write. (In the later steps, we will provide only some or none of the code you need to write.)

Here are the substeps to follow for this part of the lab:

1. Begin with your code from the previous lab, but in comment out the `sprintf` statements in **main.cpp** whenever you will have power to the motor. You can use `sprintf` for debugging, but do not do this when you are powering the motor because it will cause poor quality (or even unstable!) haptic rendering. Also in **main.cpp**, comment out your function calls from previous labs (those that blink the LED and output a constant force).
2. You will add all your haptic rendering code to the file **haplink_virtual_environments.cpp**. Note that constants such as `K_SPRING` are defined in the corresponding header file, **haplink_virtual_environments.h**. The numbers given for these constants are ones that work well, but you should start with much *lower* constants until you are sure that your rendering is working.
3. To render a virtual spring, add the following code to the function `hapkitRenderSpring` in **haplink_virtual_environments.cpp**.

```
ForceX = -K_SPRING*(xH/1000); //convert xH to meters
TorqueMotor1 = -((R_MA/R_A) * R_HA) * ForceX;
TorqueMotor1 = TorqueMotor1*0.001; //convert units
outputTorqueMotor1(TorqueMotor1);
```

   Also, start by defining `K_SPRING` as `5.0`, instead of the default `100.0`. Use "`.0`" in all such definitions in order to ensure that the C++ compiler knows that you want to perform floating-point calculations.

   Then, call the `hapkitRenderSpring` in the appropriate place in **main.cpp**.

4. WAIT!! Before you download this new program to your Hapkit, let's think a little about what you should feel from this rendering algorithm:
   - This code should look familiar – it is simply Hooke's Law as described in lecture. The

constant `K_SPRING` is used to define the virtual spring stiffness in units of Newtons per meter. Then the force is computed by multiplying this desired stiffness by the position of the handle calculated in **haplink_position.cpp** (in units of meters).

- Note that 10 N/m is *not* a very high stiffness – if you stretched the spring a whole meter, then you would feel 10 Newtons. Since your Hapkit handle only moves a fraction of a meter, the force you feel (if any) will be very low.
- If you deflect the handle 10 mm (= 1 cm = 0.01 m), you would feel 0.01*10 = 0.1 N. Remembering the range of forces you felt in Lab 4, this is on the edge of what you will be able to feel.

5. Now that you know that to expect, place your Hapkit handle in the vertical position and plug in the power supply to the motor. (Note that if your board has loaded onto it code that outputs a force from the last lab, that code will still be there! So in case you left the board loaded with a high force output program, hold on to the handle.) Download your code while holding onto the handle.
6. You should be able to feel a very weak virtual spring. Try moving the handle to its extreme positions (to the left and right), and see if you can feel a force trying to bring the handle back to center when you get to the extremes. You can also release the handle and see if it returns to center. If you can't feel anything at all, try doubling the spring stiffness (`K_SPRING`) and see if you can feel it then. If you can't feel anything at this point, it's time to debug or ask for help. Note that every time you change the value of `K_SPRING`, you will need to re-upload your sketch in order to feel the new value.
7. Assuming that you have successfully rendered a weak spring, let's kick it up a notch. Let's get the stiffness high enough such that, when you pull the handle to one side and release it, the handle oscillates a few times. This is called an *underdamped* response, and will only occur when the virtual stiffness is high enough relative to the natural friction (damping) in the device. Increase `K_SPRING` until you find a value that allows you to get a similar oscillatory behavior and write it down. Note that there are many values of `K_SPRING` that will give this response – just pick one that you like.
8. Now let's go even further. Increase `K_SPRING` until it is much stiffer. You should be able to get spring values on the order of hundreds of Newtons per meter. At some point, the spring will no longer feel compelling because either (1) the capstan drive slips as soon as you try to move the handle any significant distance, (2) the motor "saturates", that is, there is not sufficient current to allow the motor to output the commanded force, or (3) the system is unstable. If slip occurs, you will need to press the physical reset button on the corner of your board, and center the handle (make it vertical) before trying again. That's because the position sensing system will no longer have an accurate measure of the handle position due to the encoder's relative position sensing. So if slip or saturation seems to be occurring, lower `K_SPRING` until you get the highest `K_SPRING` that still seems to perform well, and write it down. Note that there are many values of `K_SPRING` that will be appropriate – pick one that you like.

WHAT TO RECORD BEFORE DEMONSTRATING TO ALLISON:

- The oscillatory `K_SPRING` in units of N/m you found in Step 1.7.
- The maximum reasonable value of `K_SPRING` in units of N/m you found in Step 1.8.

Step 2: Render a Virtual Wall

The virtual wall is a basic building block for many virtual environments, and we will render a simple version in one dimension.

Here are the substeps to follow for this part of the lab:

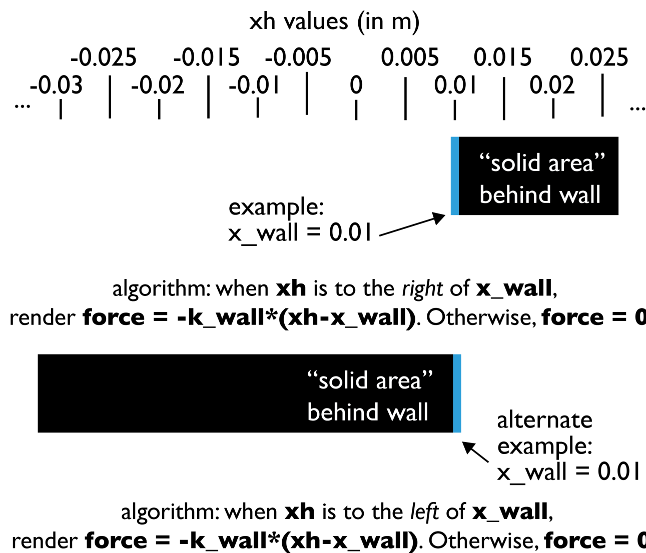
1. Comment out the `hapkitRenderSpring` function call you added in Step 1, and add the function

call hapkitRenderVirtualWall.

2. Decide a location (X_WALL) for your virtual wall, which should be noticeably different from zero (when the handle is vertical). However, let's make sure that it is close enough to the center of the workspace that you don't miss it at the edge of the workspace.

   Before you write any code, think about the logic of the wall rendering algorithm. You will need to check whether xH is beyond the boundary of the wall. If so, compute ForceX based on the stiffness of the wall. If xH is not inside the wall, the ForceX should be zero. There is slight complication in that you must also decide which side of the wall is the solid side, and which is the free space. See the figure below for an example where X_WALL is at the same place, but the "solid area" of the wall is on either side of the boundary.

   The variable K_WALL is a wall stiffness that you define. You can start with the same value you used in Step 1.8 for K_SPRING.



3. Now write code to implement this algorithm in your **haplink_virtual_environments.cpp** file, in the function hapkitRenderVirtualWall. You can copy your virtual spring code as a starting point, and also consider the "if" statement from last week's force output lab.

   Note that the values of the constants K_WALL and X_WALL are defined in **haplink_virtual_environments.h**. Modify the default values provided there.

4. Upload your program to be board, and see if you feel a compelling wall as you move around the Hapkit handle. By "compelling", we mean a wall that, when you make contact with it, feels quite stiff and nearly impenetrable. Increase K_WALL as much as possible to achieve this. Since we don't expect users to push into the virtual wall very far, I am not so concerned about saturation here. However, you don't want the motor to slip upon contact. Also, when you select X_WALL, you pick a value (and which side of the wall is "solid") such that you do not start out inside the wall when the handle is vertical. (Otherwise, you will feel a large force as soon as your program has been uploaded, and this discontinuity is likely to make your capstan drive slip.) Write down your favorite wall position, X_WALL, and stiffness, K_WALL.

WHAT TO RECORD BEFORE DEMONSTRATING TO ALLISON:

- The position of the wall X_WALL in units of mm you decided to use in Step 2.4.
- The value of K_WALL in units of N/m you decided to use in Step 2.4.

Step 3: Render a Virtual Damper

A damper outputs a force proportional to velocity. This is in contrast to the spring, which outputs a force proportional to position.

Here are the substeps to follow for this part of the lab:

1. Comment out the `hapkitRenderVirtualWall` function call you added in Step 1, and add the function call `hapkitRenderLinearDamping`.
2. As discussed in lecture, we have already provided code to calculate the velocity, `dxH`. *Note:* There are many other ways of computing filtered velocity; this is just one that works well for the Hapkit.
3. Now, add the following code to the function `hapkitRenderLinearDamping`:

```
ForceX = -B_LINEAR*dxH/1000;
TorqueMotor1 = -((R_MA/R_A) * R_HA) * ForceX;
TorqueMotor1 = TorqueMotor1*0.001; //convert units
outputTorqueMotor1(TorqueMotor1);
```

4. Start with a low value of `B_LINEAR` and increase the value until you get a significantly damped environment – one that feels very different from the natural Hapkit dynamics. But not so large that it feels gritty (caused by some remaining noise on the velocity signal that is amplified when you multiply it by a large damping coefficient). Choose your favorite value of `B_LINEAR` and write it down.

WHAT TO RECORD BEFORE DEMONSTRATING TO ALLISON:

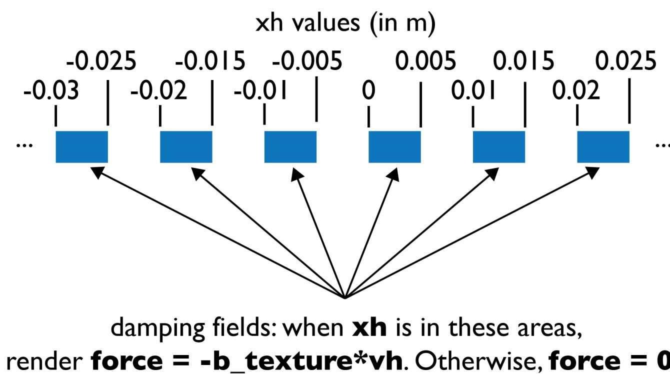
- Your favorite value of **b_damper** in units of Ns/m you found in Step 3.4.

Step 4: Render a Virtual Texture

Now you will render a virtual texture a series damping fields.

Here are the substeps to follow for this part of the lab:

1. Comment out the `hapkitRenderLinearDamping` function call you added in Step 1, and add the function call `hapkitRenderTexture`.
2. From the damper rending you did in the last step, you already have a calculation of velocity. Now you need to add code to render damping only when the handle position is within a damping field. For your first texture, use a series of damping fields each with a width of 0.005 m (0.5 cm), and a separation between the damping fields of the same distance. See the diagram below for the definition of when the handle position (`xH`) is within in a damping field. This diagram only shows the central area of the Hapkit workspace; you should render texture all the way to the edge of your workspace.



damping fields: when **xh** is in these areas,
render **force = -b_texture*vh**. Otherwise, **force = 0**.

The variable `TEXTURE_DAMPING` is a damping value that you define in **haplink_virtual_environments.h**. You can start with the same value you used in Step 3.

Before you write any code, think about the logic of the texture rendering algorithm. You will need to check whether `xH` is inside any of the damping fields. If so, compute `ForceX` based on the damping equation. If `xH` is not within a damping field, the `ForceX` should be zero.

3. Now write code to implement this algorithm in your **haplink_virtual_environments.cpp** file, in the function `hapkitRenderTexture`.

   Hints: If you are an experienced programmer, implementing this algorithm will likely be straightforward. However, if you are unfamiliar with programming, and in particular the Arduino syntax, here are some useful tips: The most straightforward (if not elegant) method is a brute-force check to see if `xH` is within a damping field. For the rightmost damping field in the diagram above, the code could look like this:

```
if ((xH > 0.02) && (xH < 0.025)) {
     ForceX = -TEXTURE_DAMPING*dxH/1000;
} else {
     ForceX = 0;
}
```

   The && is a logical "and", so in this example the position of the handle must be between 0.02 and 0.025 m. To do additional checks for the other damping fields, you could include similar code for each damping field in the same if statement, and use a logical "or" (symbol: ||) to allow the `ForceX` calculation to be active for all of the different damping fields. There are more elegant/efficient ways of doing this -- feel free to come up with a different approach.

4. Download your program to the board, and see if you feel a compelling texture as you move around the Hapkit handle. By "compelling", we mean a texture that could plausibly represent a patterned surface (in this case, a high-spatial-frequency grating) in the real world. Adjust the width of your damping fields and the value of `TEXTURE_DAMPING` until you have virtual texture that you find compelling. For simplicity, please keep the distance between the damping fields the same as the width of the damping fields, and use the same width for all damping fields. Write down your favorite damping field with and value of `TEXTURE_DAMPING`, which you will submit with your data.

WHAT TO RECORD BEFORE DEMONSTRATING TO ALLISON:

- The **width of the damping fields** in units of m you decided to use in Step 3.4.
- The value of `TEXTURE_DAMPING` in units of Ns/m you decided to use in Step 3.4.

Step 5: Render Something Else

If you complete the previous steps and have time left during the lab session, try something new!