Laser cooling simulation: Documentation

Matthew Houtput

25 October 2022

1 Introduction

This documentation accompanies an applet which explains the concept of laser cooling. The applet is a clone of a similar applet that was developed in JILA at the University of Boulder, as a part of the Physics-2000 website. This applet was written in Java and is unfortunately no longer available. Because this applet is incredibly instructive in courses on ultracold gases, I remade the applet in today's most popular programming language: Python.

The applet is written using the **Pygame** and **NumPy** packages. Only these two packages and the Python Standard Library are required for running the code. Pygame and NumPy can be installed using pip:

```
> pip install pygame
> pip install numpy
```

or using any package manager you prefer. The applet can be packaged into a Windows executable using PyInstaller on the provided LaserCooling_addfiles.spec file:

```
> pip install PyInstaller
> PyInstaller LaserCooling_addfiles.spec
```

This executable can be run on any Windows machine, without requiring to install Python, Pygame, or NumPy on that machine.

Feel free to modify or distribute this applet at will, as long as you do not remove the credit to me or JILA.

Throughout the documentation I will indicate the different elements of the code using different colors: blue for a function_name, green for a ClassName, and red for a variable_name. I will start the documentation by briefly describing the goal and gameplay of the applet. Then I will describe the physics underlying the collisions. Finally, I will discuss how all of this is implemented in Python, and I will discuss the code in more detail.

A significant portion of the applet is written using the Pygame syntax. For detailed explanations of how this code works, see the Pygame documentation (pygame.org/docs).

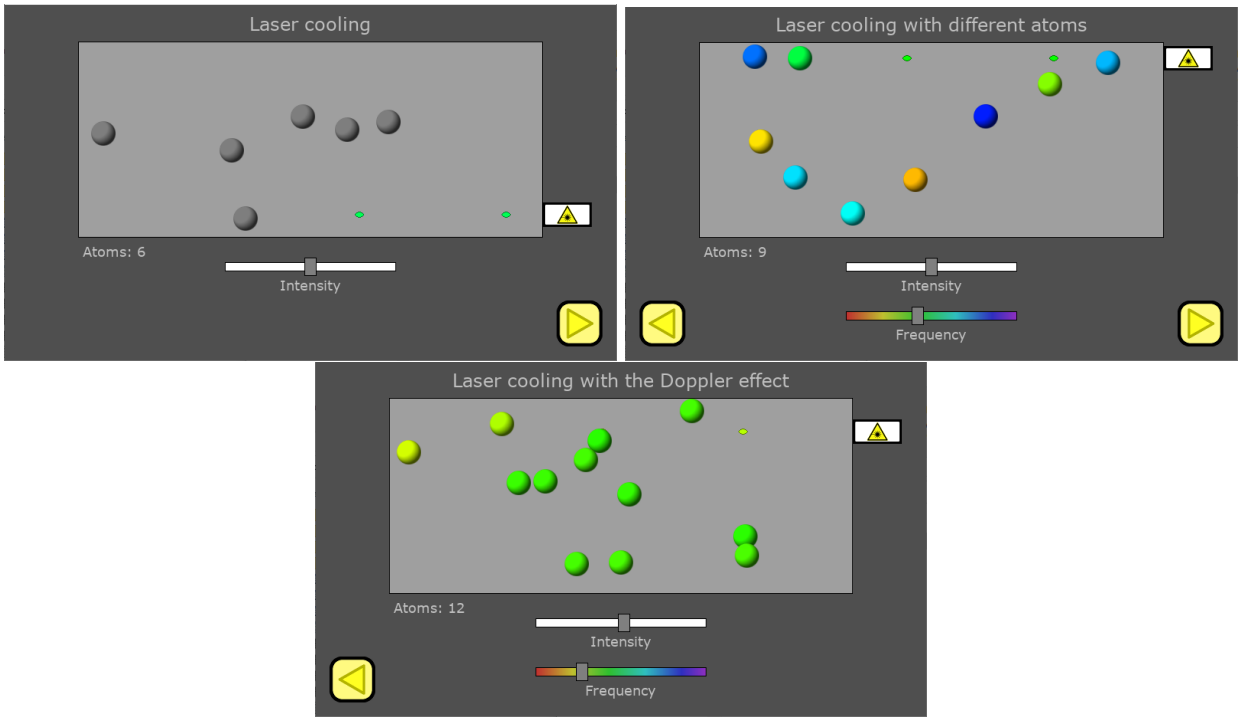

Figure 1: Applet during gameplay of the three different stages. **Top left**: The atoms always react to the laser; **Top right**: The different type of atoms only react to a single laser frequency; **Bottom**: The atoms only react to a single frequency and this frequency is dependent on the velocity of the atom.

2 Applet gameplay

The applet is made to visualize the concept of (one-sided) laser cooling, especially in the context of creating a Bose-Einstein condensate. Atoms appear from the left hand side of the screen. The player can drag the laser on the right side of the screen with the mouse, which shoots photons at these atoms. The "intensity" of the laser (the number of photons per second) can be changed with a slider bar. When a photon hits an atom, the atom absorbs that photon, which transfers the photon's momentum to the atom, slowing it down. The goal of the applet is to slow the atoms down as much as possible, and thereby keeping as many atoms on the screen as possible. Because the atoms are moving more slowly than before, they represent a cooler gas.

2.1 First stage

The applet consists of three stages, which gradually add more physical effects as you go on. The first stage is very basic. All the atoms are the same, and the frequency of the laser is fixed. The atoms will always absorb the photons if they collide. After a while, the player is faced with a problem in this applet: if a new atom appears behind another one that is already slowed down, they can't shoot it because the front atom would absorb all the photons, which would speed it up again. At this point, they are advised to move to the second stage using the button in the bottom right.

2.2 Second stage

In the second stage, the atoms are no longer the same. Each atom is a ficticious type of atom with only one absorption frequency (color). The atom will absorb the photon if the frequency of the photon is close enough to the absorption frequency of the atom[1]. The player can change the frequency of the photons by using the colored slider at the bottom of the applet. The applet gets more difficult since the player now also has to match the colors, but it resolves the problem from the first stage: if an atom spawns behind an atom with a different color, the front atom will not absorb the photons, and the back atom can be successfully slowed down.

2.3 Third stage

In the third stage, the Doppler effect is taken into account. If an atom that has a natural absorption frequency f and a velocity v, the laser frequency f' must actually be equal to the following expression in order to have the photon absorbed by the atom:

$$f' = f\sqrt{\frac{1-\frac{v}{c}}{1+\frac{v}{c}}}, \tag{1}$$

where c is the speed of light, which is arbitrarily chosen in the applet. All atoms have the same natural absorption frequency in the third stage. However, the atoms are colored according to the Doppler-shifted frequency, which is the frequency that the player must choose to slow the atom down. This frequency is velocity-dependent, so the atom changes color as it slows down.

The goal of the third stage is that the player should realize that the Doppler effect can be abused to keep many atoms on the screen with little effort. If the player chooses a laser frequency slightly below the natural absorption frequency of the atoms, the atom will only absorb the photons when it's moving, and will no longer absorb the atoms when its velocity is (almost) zero. The player can then simply keep the laser at that frequency and shoot all the atoms: the atoms will automatically stop absorbing the photons when they are alowed significantly. It is quite easy to keep 50+ atoms on the screen using this method.

[1] In order to be fair to the player, the range of frequencies that the atom reacts to is heavily exaggerated compared to the actual absorption linewidth.

3 Implementation of the atoms and photons

Most of the physics behind the applet is contained within the Atom and Photon classes and their methods. In this section, both of these classes and their functionality is described.

3.1 The Photon class

The Photon class encompasses all the photons that are shot by the laser. It does not have much functionality on its own: the photons are mostly there to interact with other objects. Every photon has the following basic properties:

- A position and velocity, both are 2×1 NumPy arrays
- The width and height of the sprite. The collision hit box of the photon is a circle with a diameter equal to the minimum of the width and height.
- A color: the hue value of this color is associated with the frequency of the photon, and therefore determines whether the photon can collide with atoms in stages 2 and 3 of the applet. In the applet, a linear relation between the frequency and the hue of a color is assumed:

$$\text{frequency} = \text{FREQ\_MIN} + \frac{\text{hue} - \text{HUE\_MIN}}{\text{HUE\_MAX} - \text{HUE\_MIN}}(\text{FREQ\_MAX} - \text{FREQ\_MIN}) \quad (2)$$

  where FREQ_MIN, FREQ_MAX, HUE_MIN, and HUE_MAX are parameters determined in the beginning of the applet, which are chosen in such a way that red is associated with a wavelength of 700nm and purple with a wavelength of 400nm.

Other than this, the Photon class only needs three simple methods to control its behavior during the applet:

- draw(surface), which draws the photon to the given surface. In this applet, there is only one surface (display_surf, defined in the main function), which is drawn to the screen every frame. Currently, the photon is drawn as an ellipse in the correct color, using only the built-in Pygame draw functions.
- move(), a method that is called every frame to move the photon in the direction of its current velocity
- get_bounding_rectangle(), which returns the bounding rectangle of the photon in the standard Pygame format (x, y, width, height).

Throughout the applet, all photons that are currently on screen are kept in a list called photons.

3.2 The Atom class

The Atom class encompasses all the atoms that can enter the screen from the left side. The different types of atoms in the different stages are all objects of this class, and much of the functionality of the applet is implemented in this class. The atoms have similar basic properties to the photons:

- A position and velocity

- A color: this color is stored in both HSV and RGB format, since the hue value of this color determines what color photons the atom can absorb
- The radius of the atom
- image, the image that is used to represent the atoms. The current image is a white ball: the color is added later using the set_color method.

However, on top of these properties, the atoms have some other properties controlling their behavior:

- hue_range, which determines how strictly the color of the photon has to match the color of the atom before it is absorbed: if the hue of the photon differs less than hue_range from the hue of the atom, the photon is absorbed
- doppler, a Boolean that indicates whether the Doppler effect has to be taken into account when calculating the properties of the atom
- hue_bare: The hue of the atom if the Doppler effect is not taken into account, or in other words, when the atom is standing still
- hue: Represents the actual hue of the atom, which determines the color that is drawn to the screen, and also which photons can be absorbed by the atom. If self.doppler is false (in stages 1 and 2), hue is simply equal to hue_bare. However, if self.doppler is true, hue is corrected to account for the Doppler effect.

Just like the photons, the Atom class has basic move and draw methods. In this case, draw simply draws the image that is currently associated with the atom. Atom has several other methods:

- set_color(color): This method changes the color of the image associated with the atom to the given color, by blending the white image with the given color
- set_hue(hue, include_doppler): This method changes the hue of the atom to the given value, and updates all other color related variables. It also calls set_color to recolor the image of the atom. If the flag include_doppler is set to True, the Doppler effect is automatically included. The formula used in the applet results from combining equations (1) and (2).
- collide(photons): This function checks for collisions with any instance of Photon in the given list of photons. The photon is absorbed if and only if:

  1. photon.hue and atom.hue differ by at most atom.hue_range
  2. the circular hit box of the photon overlaps with the circular hit box of the atom

  When the photon is absorbed, it is destroyed, and the velocity and hue (only in stage 3) of the atom are adjusted.

This implementation allows us to create all different atom types for the different stages. In stage 1, we make atoms with a grey color, and hue_range large enough such that all colors are absorbed. In stage 2, we make colored atoms with doppler=False, and in stage 3, we make colored atoms with doppler=True.

4 Implementation of the main game loop

In this chapter I'll go over the applet's main function, and give a more detailed explanation of what each of the functions do. This chapter is meant to be read in parallel with the code.

4.1 Setup

When the applet is started but before the player presses play, we initialize some variables and the playing field. The variables that are supposed to be universally available constants are initialized before the main function is called, which are:

- The dimensions of the window (PLAY_WIDTH, PLAY_HEIGHT, ...)
- The frame rate (FPS)
- Several colors (WHITE, LIGHT_GRAY, ...). Here, the colors are written in RGB, but the colors related to the atoms and the light will be written in HSV, since we can associate the frequency of the light with the hue value of the colors.
- Several variables that control the gameplay:
  - ATOM_RADIUS, the radius of the atoms
  - SPEED_OF_LIGHT, the speed of the photons
  - ATOM_COLLISION_SPEED_GAIN, the speed that an atom gains if it absorbs a photon,
  - HUE_MIN and HUE_MAX, respectively the minimum and maximum values that the hues of the colors of the light can take: this goes from red (HUE_MIN = 0) to purple (HUE_MAX = 282).
  - HUE_ABSORPTION_RANGE: If a photon collides with an atom, it will only be absorbed if their hue values differ by less than this variable
  - TIMES_BETWEEN_ATOMS_INITIAL and TIMES_BETWEEN_ATOMS_FINAL: At the beginning of a stage, the atoms spawn in slowly, with a delay of 300 frames (10 seconds) between atoms. The delay is determined by TIMES_BETWEEN_ATOMS_INITIAL, a 3-tuple that contains the delays for stage 1, stage 2, and stage 3. As the stage continues, the atoms spawn in more quickly to make the applet more challenging: the spawn times are capped by the values in the 3-tuple TIMES_BETWEEN_ATOMS_FINAL.
- The variables associated with the implementation of the Doppler effect:
  - SPEED_OF_LIGHT_DOPPLER, the speed of light that is used in formula (1)
  - FREQ_MIN and FREQ_MAX, the frequencies that correspond to the hue values in HUE_MIN and HUE_MAX. The units of these frequencies are irrelevant: in practice, only the ratio of FREQ_MIN / FREQ_MAX is used.

Then, the main function is called[2], which starts with the initialization of Pygame. Specifically, we:

- create fpsclock, the clock that will take care of the timing of the applet

[2]More precisely, we define all the functions, and the main function is called at the end of the program.

- create display_surf, the surface that the whole applet will be drawn to and which is eventually printed on the screen
- set the caption of the screen to "Laser cooling"

Then, we initialize some fonts for further use, and initialize the variables that we will use to store the state of the mouse. We create the Buttons and Sliders that the player will use to control the applet; more information on the buttons and sliders can be found in section 5. We also create the Laser here, since it functions exactly like one of the sliders.

Then, we initalize some default values for variables that we need later on:

- Create empty lists atoms and photons, that will later contain all the Atom and Photon objects on screen
- Set the atom_timer to 150 frames. atom_timer is a 2-tuple whose first value is increased by 1 every frame. The second value represents the delay between atoms: an atom is created when atom_timer[0] is equal to atom_timer[1].
- Set current_level to 1.
- hsv_color represents the natural absorption color of all the atoms in stage 3. In stages 1 and 2, this value is not needed and is set to None.
- flag_restart indicates whether the applet or a stage needs to be restarted: this is not the case at the start, so it is set to False.

4.2 Game loop

At this point, the applet has all the information it needs to start running, and it enters the main game loop. This loop is a simple "while True" loop, which will be exited when the players presses Escape or closes the applet by pressing the red X button. All of the following subsections occur in the main game loop, and will therefore keep occurring until the applet is terminated. Note that everything in this loop is executed once per frame, or 30 times per second. Indeed, the last line in the main game loop is:

```
fpsclock.tick(FPS),
```

where FPS = 30. Every time the method fpsclock.tick is called, it halts the program until at least 1/30 seconds have passed since the last time it was called. This ensures that the game will run at 30 frames per second, or slower if the calculations are too intensive.

4.2.1 Retrieve user input

First, the user input is retrieved using Pygame's built-in events (pygame.org/docs/ref/event.html). Pygame has a queue of events that track all the user's input, such as button presses and mouse movement, and a list of all of these events can be obtained by calling pygame.event.get().

In the applet, these events are used to do two things. Firstly, the applet checks whether the user has pressed Escape or the window's X button: in either case, the applet is terminated. Secondly, the applet tracks the current position of the mouse and the state of the left mouse button: mouse_xy will

store the current coordinates of the mouse, mouse_is_down stores whether the left mouse button is down or up, and mouse_is_clicked stores whether the left mouse button was clicked on this exact frame. Since the player controls the applet exclusively via the mouse, no other user input is needed.

4.2.2 Creating new atoms

Next, a new atom is created if necessary. This behavior is handled by the functions run_atom_timer and create_random_atom. Every frame, the atom_timer is increased by one. If the timer hits its maximum value, it is reset to zero, and a random atom is created. The process then repeats.

The creation of random atoms is handled by the function create_random_atom. Firstly, it is given a random y-coordinate and velocity. Then, depending on the stage of the applet, it is given different attributes:

- In stage 1, the atom is given a grey color, a hue_range of 360 such that it always absorbs all colors, and the Doppler effect is not included
- In stage 2, the atom is given a random color[3], a hue_range equal to the globally defined value HUE_ABSORPTION_RANGE, and the Doppler effect is not included
- In stage 3, the atom is given a color that is randomly chosen at the beginning of the stage, which is contained in the variable hsv_color. This way, all atoms are the same. The hue_range is again chosen equal to the global value, and the Doppler effect is included.

Once the atom is created, it is appended to the list of atoms.

4.2.3 Move all atoms and photons, and let them collide

Then, we move the atoms and photons for a time equal to 1 frame, using their built-in move method. After their positions are updated, they are also drawn to the surface display_surf using their draw method. Since the atoms and photons are in new positions, we should check for any collisions between the atoms and photons, using the collide method. All the physics is already implemented in this method. Finally, we must check for all atoms if they are still on the screen: if any atom is no longer on the screen, it should be destroyed. This functionality is implemented in the function remove_outside_atoms.

4.2.4 Draw everything to the screen

Every frame, we have to update the graphics on the screen: this includes redrawing the atoms on their correct positions, but also the text on the screen, the background colors, ... The only things that are not drawn in this step, are the buttons and the sliders. Pygame draws images as follows: the graphics are first drawn to a surface, and near the end of the code, the statement pygame.display.update() actually draws this surface to the screen. For this applet, we only have one surface called display_surf, so we draw all graphics to this surface.

Most of the graphics in this applet are fairly simply drawn using Pygame's built-in functions such as pygame.draw.line, pygame.draw.rect, and pygame.draw.ellipse, which respectively draw a line, a rectangle, or an ellipse to a surface. Sprites and text are drawn in a slightly different way, using

[3]Only the hue is randomized. The saturation and value of the color are set to their maximum.

the method display_surf.blit: this method is used e.g. in the Atom.draw method to draw the atoms to a surface.

The grey background, the atoms, and the photons have already been drawn to display_surf at this point. All the other things to draw are grouped together into two draw functions:

- draw_borders draws the dark grey borders around the edge of the scren, on top of which the thermometer, buttons, and slider are located
- Finally, draw_text draws all the remaining text to the screen. This text consists only of the title and the number of atoms, but this is grouped in a separate function since drawing text in Pygame is rather verbose.

4.2.5 Control the sliders and buttons

Next, using the user input in mouse_state = (mouse_xy, mouse_is_clicked, mouse_is_down), we retrieve the information that is stored in the buttons and the sliders. The buttons, sliders, and the laser are implemented manually near the end of the code, in terms of several classes: Button, ImageButton, Slider, and Laser.

In short, the buttons and sliders are implemented such that they are operated by a single control method. control does everything associated with the button or slider. For a button, it checks whether the mouse is inside it, it updates the state of the button depending on whether it is clicked or not and returns a flag that represents its state (True if the button is down, False if the button is up), and it draws the button to the displaysurf surface. For a slider, it does the same, except instead of returning its state as a True or False value, it returns the value of the quantity associated with the slider. For example, when calling the control method of the "Frequency" slider, it returns the hue of the currently selected color.

The applet contains two buttons, one that allows the player to go to the next stage (which is not drawn in stage 3) and one that allows the player to go to the previous stage (which is not drawn in stage 1). If either button is pressed, the applet "restarts" by clearing all the atoms and photons, resetting the atom timer, and choosing a new color for the atoms (only in stage 3). The applet also contains two sliders, one to control the intensity and one to control the frequency of the laser (only in stage 2 and 3). The result of the intensity slider is immediately used to set the fire rate of the laser. Finally, the laser is also a slider, which is controlled by the controlShoot method. This method allows the laser to function as a slider, but it also periodically shoots photons. Its implementation is discussed in 5.2.3.

The Button and Slider classes can be used in other applets as well: in particular, they are also used in the sister applet on evaporation cooling. Therefore, their implementations are separate from the rest of the code. For the interested reader, I have detailed the implementation of these classes in chapter 5.

4.2.6 Update the screen

Finally, after the buttons and sliders has been updated, there are two lines of code left, which we have already discussed. The function pygame.display.update() will update the screen using everything that was drawn on the surface displaysurf, and the method fpsclock.tick will ensure that the game runs at FPS frames per second. After these lines of code have been executed, the main game loop repeats, and we start back at section 4.2.1.

5 Implementation of the buttons and sliders

In this section I will go deeper in the implementation of the Button and Slider classes, as well as the Laser class since this is essentially just a glorified slider. The Button and Slider classes are not specific to this applet, and can be used in other Pygame applications. Both the button and slider are controlled in the main game loop by a single call to the control method.

5.1 Buttons

5.1.1 The Button class

The Button class is a simple button that does not have any visuals. It is meant as a class that other button classes inherit from: it has all the core mechanics that we need from a button, but we will not make any buttons directly from this class.

The button has two possible states: "active" and "idle". Whenever the control method is called, it calls the action method when it is active and the idle method when it is idle. Both of these methods can be quite general, but the basic behavior is that the action method simply returns True and the idle method returns False.

A basic Button has two core variables:

- surface, the surface that the button lives on. Whenever the draw method is called, the button is drawn to this surface.
- bounding_rectangle: A 4-tuple of the form (x, y, width, height) that represents the bounding box of the button. Following the standard Pygame convention for rectangles, the top left corner of this bounding box is (x, y).

Besides this, it has six methods:

- action, the method that is called when the button is active. The default behavior is that it returns True.
- idle, the method that is called when the button is not active. The default behavior is that it returns False.
- is_active: Determines whether the button is active based on the mouse_state. The default behavior is that it returns True when mouse_clicked is true and when mouse_xy is inside the hitbox defined by check_mouse.
- check_mouse: Determines whether the mouse is inside the hitbox of the button. By default, this hitbox is a rectangle equal to the bounding_rectangle.
- draw: Draw the button to self.surface. By default, it simply draws a black rectangle, but usually this function is overwritten to draw an image based on the mouse_state.
- control: This function combines the functionality of the prevous five methods and is the only method used in practice. It draws the button to the screen, checks whether the button is_active, calls action or idle, and returns that method's result.

The typical usage of this button would be to add the following statement to the main game loop:

```
flag_pressed = my_button.control(mouse_state)
```

This statement draws the button, and the variable flag_pressed returns True if the button was clicked and False if it was not. After that, the code can include what needs to happen if flag_pressed is True or False.

5.1.2 The ImageButton class

The ImageButton class inherits from the basic Button and has exactly the same functionality. The difference is only aesthetic: an ImageButton is drawn as an image rather than a black rectangle. In total, an ImageButton can have up to three images associated with it:

- idle_image when the button is idle
- hover_image when the mouse hovers over the button (default: equal to idle_image)
- down_image when the button is pressed (default: equal to hover_image)

These images are loaded in the constructor using the Pygame syntax. All other methods are inherited from Button, except for the draw method. For ImageButton, draw decides which of the images is relevant based on the mouse_state, and draws that image to the surface associated with the button. All of the buttons in this applet belong to the ImageButton class.

5.2 Sliders

5.2.1 The Slider class

A slider plays a similar role to a button in the sense that it allows the player to interact with the applet, and both can be included in the main game loop using a single call to the control method. The main difference is that we use a button to return a Boolean variable, while a slider can return any real number within a given range. Therefore, the implementations of the Slider and Button classes overlap slightly.

An instance of Slider will have the following core attributes:

- surface, the surface that the slider is drawn to
- bounding_rectangle: A 4-tuple of the form (x, y, width, height) that represents the bounding box of the slider
- direction: Either "horizontal" or "vertical", representing whether the slider button should be moved horizontally or vertically
- min_value and max_value, respectively the minimal and maximal values that the slider can return
- activation: The position of the slider will be mapped to an activation $\in [0, 1]$, where activation $= 0$ corresponds to the extreme left of the slider and activation $= 1$ corresponds to the extreme right.
- slider_size: A tuple containing the width and height of the slider knob, which is drawn as a rectangle

- sliding: A boolean that indicates whether the knob is being moved
- text_string, text_font, text_color: Two variables associated with the text that is drawn under the slider

The idea behind the slider is that its x-position can be changed by dragging the slider, which will translate to changing the value of activation. In turn, the control method will return a different value based on the value of activation:

$$\text{return } \text{min\_value} + \text{activation} * (\text{max\_value} - \text{min\_value}) \tag{3}$$

The behavior is implemented in the following methods:

- get_slider_activation, which calculates the activation based on the given position of the slider knob and direction of the slider
- get_slider_xy, which calculates the position of the slider knob based on the current value of the activation; this function is essentially the inverse function of get_slider_activation
- get_slider_value, which returns the output value given by (3)
- set_slider_value, which sets the activation in a way that the output value is equal to a desired value
- is_sliding, a customizable function that decides when the knob is sliding based on the input mouse_state. Currently, it is implemented that clicking the hit box of the slider (determined by check_mouse) and holding the mouse button down allows the knob to slide. Sliding is possible as long as the mouse button is held down, even if the mouse exits the hit box.
- check_mouse, which defines the hit box of the slider. It is currently implemented as a rectangle that is as wide as the slider bounding box and as high as the slider knob if the slider is horizontal, and as high as the slider bounding box and as wide as the slider knob if the slider is vertical.
- draw, which draws the slider to its associated surface. This includes the background rectangle, the knob, and the description text.
- control, which combines the functionality in all of the above methods into a single method. It draws the slider, calculates the slider activation using get_slider_activation if it is_sliding, and finally it returns the slider value using get_slider_value.

The typical usage of a Slider in the main game loop is very similar to the usage of a button:

```
output_value = my_slider.control(mouse_state)
```

with the only difference that it returns a real number rather than a Boolean. In the applet, only one Slider is available, which is the slider that controls the intensity of the laser.

5.2.2 The HueSlider class

The HueSlider class inherits from the Slider class and has exactly the same practical functionality. The only difference is that it is drawn with a rainbow background. It is used for the slider that controls the color of the laser.

In the constructor, a second surface called rainbow_surface is created by drawing vertical lines of all colors between HUE_MIN and HUE_MAX, creating a rainbow pattern. This surface then plays the role of an image. The HueSlider.draw method is the same method as the Slider.draw method, except the first line (which draws a white background) is replaced with the drawing of the rainbow_surface. All other methods are unchanged and are therefore directly inherited from Slider.

5.2.3 The Laser class

Strangely, the laser in the applet has a lot in common with the slider bars. Although it is not used to set a value, controlling the laser with the mouse is exactly the same as controlling a slider with the mouse. Therefore, the Laser class inherits from the Slider class, and further functionality is built on top of this class.

The Laser class inherits all properties from the Slider class, except for the text-related properties. In particular, the surface and bounding_rectangle of the laser should be specified explicitly. Furthermore, the Laser has the following additional properties:

- An image, that can be found in the location specified by image_path. The image will play the role of the slider knob: therefore, it is scaled to match the dimensions of the bounding_rectangle.
- The firing_delay between successive photon shots, measured in frames. An internal timer keeps track of the time until the next photon should be shot.

The Laser class also inherits all the methods from the Slider class, except for the draw method which is simply replaced by drawing the laser image to the given surface. Laser has two additional methods to control its behavior:

- set_fire_rate, which simply sets the value of firing_delay to a different value. It is used in the main loop in conjunction with the intensity slider.
- control_shoot: This method combines the functionality of the Slider.control method and the periodic shooting of photons. Every time this function is called (which is every frame), the internal timer of the laser is increased by one. If it hits its maximum value, it is reset to zero, and the laser creates a Photon. The photon is created at the laser's current location, with a velocity equal to the global value SPEED_OF_LIGHT and a color determined by the current value of the frequency slider. The photon is then appended to the list containing all photons. When the timer and photons are handled, this method calls Slider.control to draw the laser and move it to the current position of the mouse, if necessary.

The laser can be controlled in the main game loop by only calling the control_shoot method once, just like all other buttons and sliders.

6 Closing remarks

I hope you and your students will enjoy this applet as much as I have enjoyed making it! A fun idea is to let multiple students try the applet, and have some competition to see who can leave the most amount of atoms on the screen. For stage 3 of the applet, keeping at least 50 atoms should be possible, although it takes quite some time.