A reminder about the Honor Code

For previous mini-projects, we have had instances of students submitting solutions that have been copied from the web. Remember, if you can find code on the web for one of the mini-projects, we can also find that code. Submitting copied code violates the Honor Code for this class as well as Coursera's Terms of Service. Please write your own code and refrain from copying. If, during peer evaluation, you suspect a submitted mini-project includes copied code, please evaluate as usual and email the assignment details to *iipphonorcode@online.rice.edu*. We will investigate and handle as appropriate.

Mini-project description - Spaceship

In our last two mini-projects, we will build a 2D space game RiceRocks that is inspired by the classic arcade game Asteroids (1979). Asteroids is a relatively simple game by today's standards, but was still immensely popular during its time. (Joe spent countless quarters playing it.) In the game, the player controls a spaceship via four buttons: two buttons that rotate the spaceship clockwise or counterclockwise (independent of its current velocity), a thrust button that accelerates the ship in its forward direction and a fire button that shoots missiles. Large asteroids spawn randomly on the screen with random velocities. The player's goal is to destroy these asteroids before they strike the player's ship. In the arcade version, a large rock hit by a missile split into several fast moving small asteroids that themselves must be destroyed. Occasionally, a flying saucer also crosses the screen and attempts to destroy the player's spaceship. Searching for "asteroids arcade" yields links to multiple versions of Asteroids that are available on the web (including an updated version by Atari, the original creator of Asteroids).

Mini-project development process

For this mini-project, you will implement a working spaceship plus add a single asteroid and a single missile. We have provided art for your game so its look and feel is that of a more modern game. You should begin by loading the**program template**.The program template includes all necessary image and audio files. Unfortunately, no audio format is supported by all major browsers so we have decided to provided sounds in the **mp3** format which is supported by Chrome (but not by Firefox on some systems). (**ogg** versions are also available.) **We highly recommend using Chrome for the last two weeks of the class.**We have found that Chrome typically has better performance on games with more substantial drawing requirements and standardization on a common browser will make peer assessing projects more reliable.

Phase one - Spaceship

In this phase, you will implement the control scheme for the spaceship.This includes a complete Spaceship class and the appropriate keyboard handlers to control the spaceship. Your spaceship should behave as follows:

- The left and right arrows should control the orientation of your spaceship. While the left arrow is held down, your spaceship should turn counter-clockwise. While the right arrow is down, your spaceship should turn clockwise. When neither key is down, your ship should maintain its orientation. You will need to pick some reasonable angular velocity at which your ship should turn.
- The up arrow should control the thrusters of your spaceship. The thrusters should be on when the up arrow is down and off when it is up. When the thrusters are on, you should draw the ship with thrust flames. When the thrusters are off, you should draw the ship without thrust flames.
- When thrusting, the ship should accelerate in the direction of its forward vector. This vector can be computed from the orientation/angle of the ship using the provided helper function **`angle_to_vector`**. You will need to experiment with scaling each component of this acceleration vector to generate a reasonable acceleration.
- Remember that while the ship accelerates in its forward direction, but the ship always moves in the direction of its velocity vector. Being able to accelerate in a direction different than the direction that you are moving is a hallmark of *Asteroids*.
- Your ship should always experience some amount of friction. (Yeah, we know, "Why is there friction in the vacuum of space?". Just trust us there is in this game.) This choice means that the velocity should always be multiplied by a constant factor less than one to slow the ship down. It will then come to a stop eventually after you stop the thrusters.

Now, implement these behaviors above in order. Each step should require just a few lines of code. Here are some hints:

1. Modify the draw method for the Ship class to draw the ship image (without thrust flames) instead of a circle. This method should incorporate the ship's position and angle. Note that the angle should be in radians, not degrees. Since a call to the ship's draw method already exists in the draw handler, you should now see the ship image. Experiment with different positions and angles for the ship.
2. Implement an initial version of the update method for the ship. This version should update the position of the ship based on its velocity. Since a call to the update method also already exists in the draw handler, the ship should move in response to different initial velocities.
3. Modify the update method for the ship to increment its angle by its angular velocity.
4. Make your ship turn in response to the left/right arrow keys. Add keydown and keyup handlers that check the left and right arrow keys. Add methods to the Ship class to increment and decrement the angular velocity by a fixed amount. (There is some flexibility in how you structure these methods.) Call these methods in the keyboard handlers appropriately and verify that you can turn your ship as you expect.
5. Add a method to the Ship class to turn the thrusters on/off (you can make it take a Boolean argument which is True or False to decide if they should be on or off). Modify the keyboard handler to call this method in response to the thrust key being pressed/released.
6. Modify the ship's draw method to draw the thrust image when it is on. (The ship image is tiled and contains both images of the ship.)
7. Modify the ship's thrust method to play the thrust sound when the thrust is on. Rewind the sound when the thrust turns off.
8. Add code to the ship's update method to use the given helper function **`angle_to_vector`** to compute the forward vector pointing in the direction the ship is facing based on the ship's angle.
9. Next, add code to the ship's update method to accelerate the ship in the direction of this forward vector when the ship is thrusting. You will need to update the velocity vector by a small fraction of the forward acceleration vector so that the ship does not accelerate too fast.

10. Then, modify the ship's update method such that the ship's position wraps around the screen when it goes off the edge (use modular arithmetic!).
11. Up to this point, your ship will never slow down. Finally, add friction to the ship's update method as shown in the "Acceleration and Friction" video by multiplying each component of the velocity by a number slightly less than 1 during each update.

You should now have a ship that flies around the screen,as you would like for *RiceRocks*. Adjust the constants as you would like to get it to fly how you want.

Phase two - Rocks

To implement rocks, we will use the provided Sprite class. Note that the update method for the sprite will be very similar to the update method for the ship. The primary difference is that the ship's velocity and rotation are controlled by keys, whereas sprites have these set randomly when they are created. Rocks should screen wrap in the same manner as the ship.

In the template, the global variable **`a_rock`** is created at the start with zero velocity. Instead, we want to create version of **`a_rock`** once every second in the timer handler. Next week, we will add multiple rocks. This week, the ship will not die if it hits a rock. We'll add that next week. To implement rocks, we suggest the following:

1. Complete the Sprite class (as shown in the "Sprite class" video) by modifying the draw handler to draw the actual image and the update handler to make the sprite move and rotate. Rocks do not accelerate or experience friction, so the sprite update method should be simpler than the ship update method. Test this by giving **`a_rock`** different starting parameters and ensuring it behaves as you expect.
2. Implement the timer handler **`rock_spawner`**. In particular, set **`a_rock`** to be a new rock on every tick. (Don't forget to declare **`a_rock`** as a global in the timer handler.) Choose a velocity, position, and angular velocity randomly for the rock. You will want to tweak the ranges of these random numbers, as that will affect how fun the game is to play. Make sure you generated rocks that spin in both directions and, likewise, move in all directions.

Phase three - Missiles

To implement missiles, we will use the same sprite class as for rocks. Missiles will always have a zero angular velocity. They will also have a lifespan (they should disappear after a certain amount of time or you will eventually have missiles all over the place), but we will ignore that this week. Also, for now, we will only allow a single missile and it will not yet blow up rocks. We'll add more next week.

Your missile should be created when you press the spacebar, not on a timer like rocks. They should screen wrap just as the ship and rocks do. Otherwise, the process is very similar:

1. Add a **`shoot`** method to your ship class. This should spawn a new missile (for now just replace the old missile in **`a_missile`**). The missile's initial position should be the tip of your ship's "cannon". Its velocity should be the sum of the ship's velocity and a multiple of the ship's forward vector.
2. Modify the keydown handler to call this shoot method when the spacebar is pressed.
3. Make sure that the missile sound is passed to the sprite initializer so that the shooting sound is played whenever you shoot a missile.

Phase four - User interface

Our user interface for *RiceRocks* simply shows the number of lives remaining and the score. This week neither of those elements ever change, but they will next week. Add code to the draw event handler to draw these on the canvas. Use the **`lives`** and **`score`** global variables as the current lives remaining and score. For more helpful tips on implementing this mini-project, please visit the Code Clinic tips page for this mini-project.

Grading rubric - 20 pts total (scaled to 100 pts)

When testing the functionality of your peer's projects, remember that some keyboards don't register more three or more simultaneous key presses correctly. So please assess based on single key presses or combinations of two key presses. Also, please assess your peer's mini-projects in Chrome.**If, for some reason, you must use Firefox or another browser (or had issues playing sounds in Chrome), please give your peers full credit on the two sound-related rubric items.**

- 1 pt - The program draws the ship as an image.
- 1 pt - The ship flies in a straight line when not under thrust.
- 1 pt - The ship rotates at a constant angular velocity in a counter clockwise direction when the left arrow key is held down.
- 1 pt - The ship rotates at a constant angular velocity in the clockwise direction when the right arrow key is held down.
- 1 pt - The ship's orientation is independent of its velocity.
- 1 pt - The program draws the ship with thrusters on when the up arrow is held down.
- 1 pt - The program plays the thrust sound only when the up arrow key is held down.
- 1 pt - The ship accelerates in its forward direction when the thrust key is held down.
- 1 pt - The ship's position wraps to the other side of the screen when it crosses the edge of the screen.
- 1 pt - The ship's velocity slows to zero while the thrust is not being applied.
- 1 pt - The program draws a rock as an image.
- 1 pt - The rock travels in a straight line at a constant velocity.
- 1 pt - The rock is respawned once every second by a timer.
- 1 pt - The rock has a random spawn position, spin direction and velocity.
- 1 pt - The program spawns a missile when the space bar is pressed.
- 1 pt - The missile spawns at the tip of the ship's cannon.
- 1 pt - The missile's velocity is the sum of the ship's velocity and a multiple of its forward vector.
- 1 pt - The program plays the missile firing sound when the missile is spawned.
- 1 pt - The program draws appropriate text for lives on the upper left portion of the canvas.
- 1 pt - The program draws appropriate text for score on the upper right portion of the canvas.

Add a method to the Ship class to turn the thrusters on/off (you can make it take a Boolean argument which is True or False to decide if they should be on or off). Modify the keyboard handler to call this method in response to the thrust key being pressed/released

✓ Complete