Alien Invaders

AUTHOR
Christian Osendorfer

PUBLISHED
December 22, 2023

Introduction

Acknowledgements: This project is taken from the excellent CS1110 course at Cornell

In this project, you will create a classic of the original arcade era: a Space Invaders clone. If you have never played Space Invaders before, there are a few versions available online. The Pacxon version is a very authentic one available online. You should play this to get a good idea of the gameplay. It also demonstrates the importance of making changes that avoid copyright issues.

Copyrighted Material

There is an important issue with this assignment: copyrighted material. Gameplay cannot be copyrighted. You can make a game that plays the same as another. Indeed, it was Space Invaders itself that lost the court case (against Galaxian and Galaga) that established this fact. However, artwork in a game is copyrighted (and in this case, even trademarked). So you should not attempt to use the original Space Invaders characters for your game. You should also not use other copyrighted material like Pokemon.

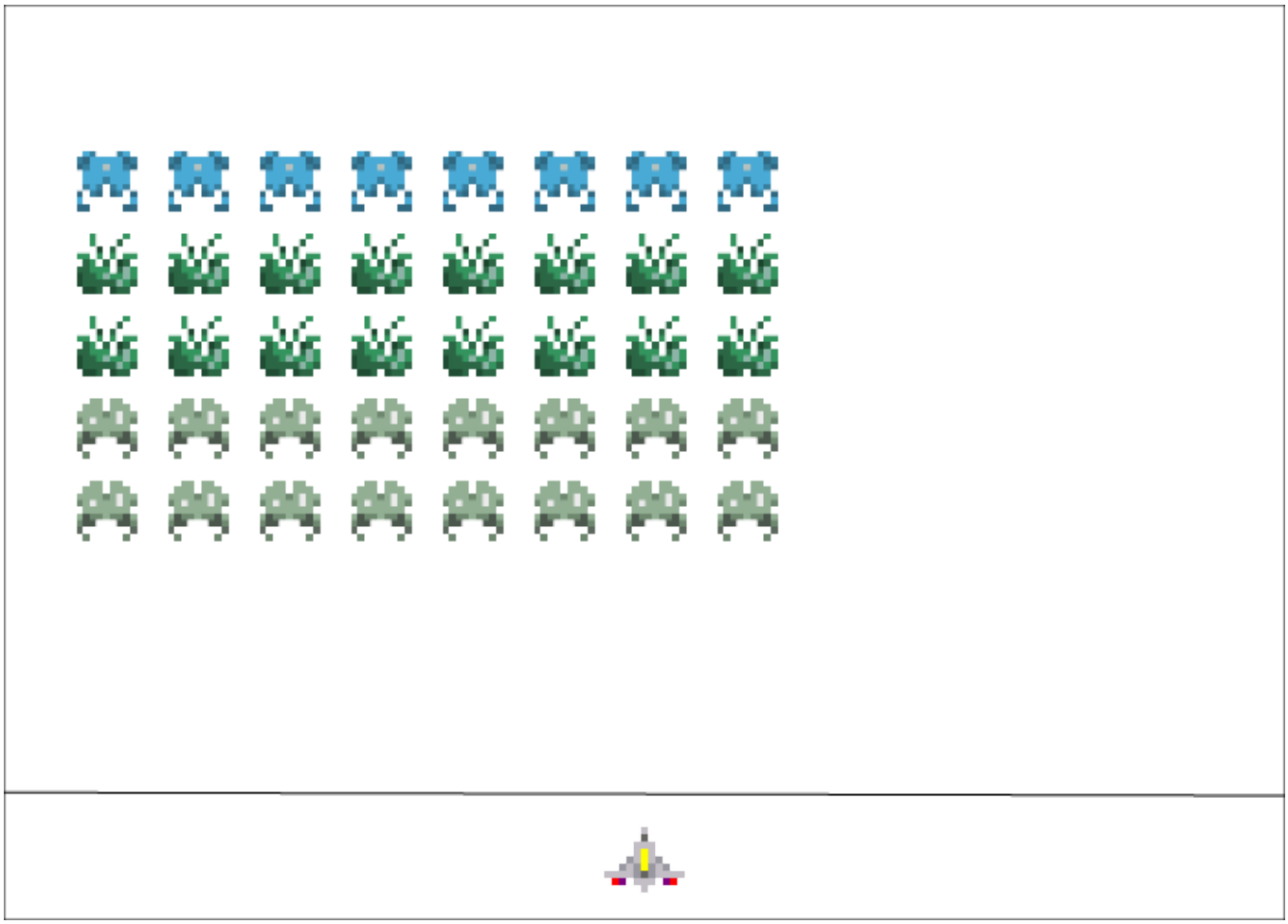
While there is maybe an argument for fair use – this is a class project – your instructor prefers that you avoid the copyright issue entirely. The Space Invaders characters are iconic. While they may not appear in video games these days, they are still sold on T-shirts and are still widely recognized. Furthermore, the current rights holder is a little company called Square Enix, which is not afraid of lawsuits.

In general, you are only allowed to use copyrighted material if you have a license to do so. For example, many of the songs and sound effects in the NewGrounds library are available for you to use under an Attribution License. That means you are free to use it so long as you cite the source in your documentation (e.g. your header comments). This is okay. A license where you have to pay is not okay.

If you are in doubt as to whether you have a license to use something, ask your instructors.

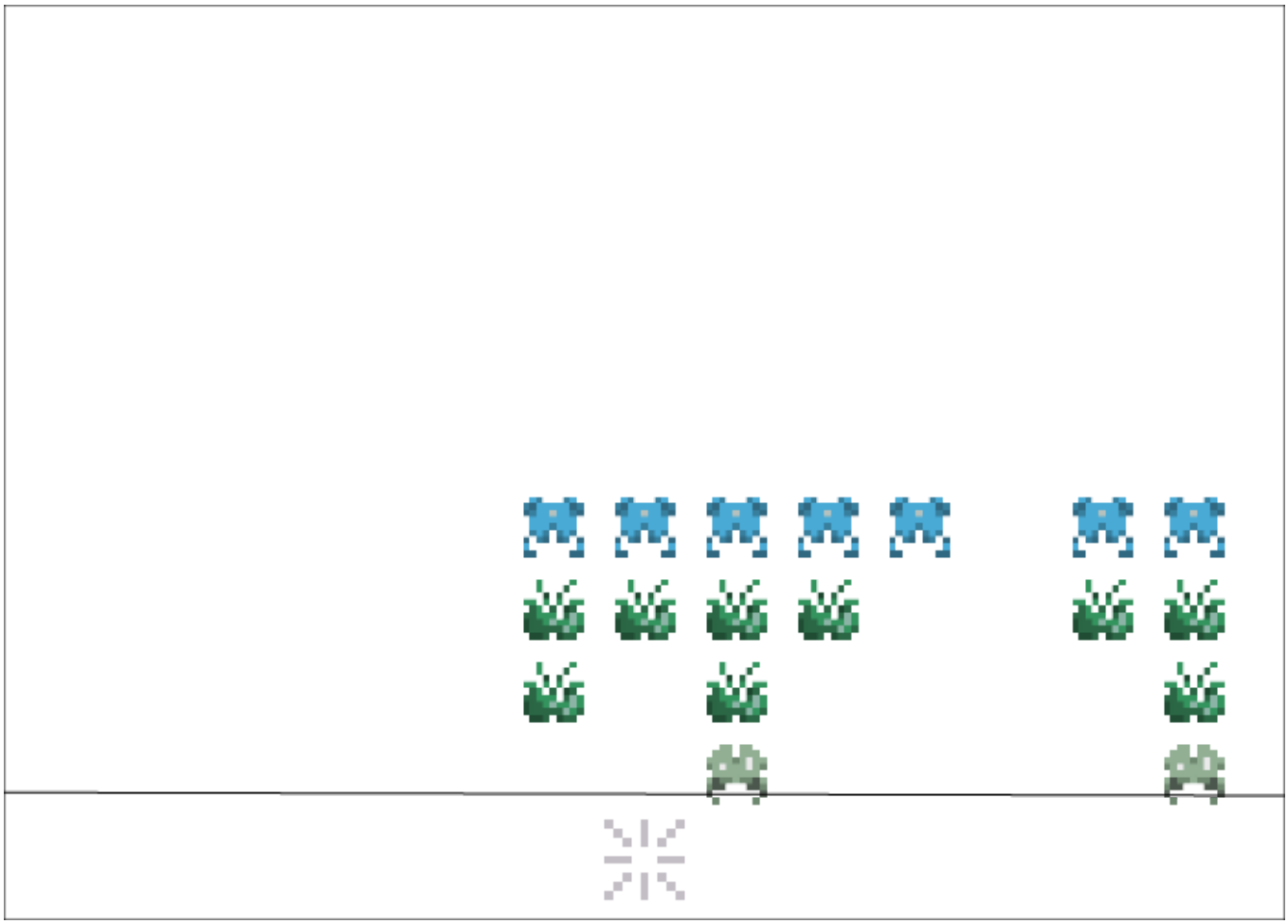
Alien Invaders – The Game

The initial configuration of Alien Invaders is shown in the picture below. There are aliens arrange in rows and columns on the left side of the screen. At the bottom of the screen is the player's ship. There is also a horizontal line at the bottom of the screen. This is the defense line. If the aliens make it past this line, they have successfully invaded and you have lost the game.



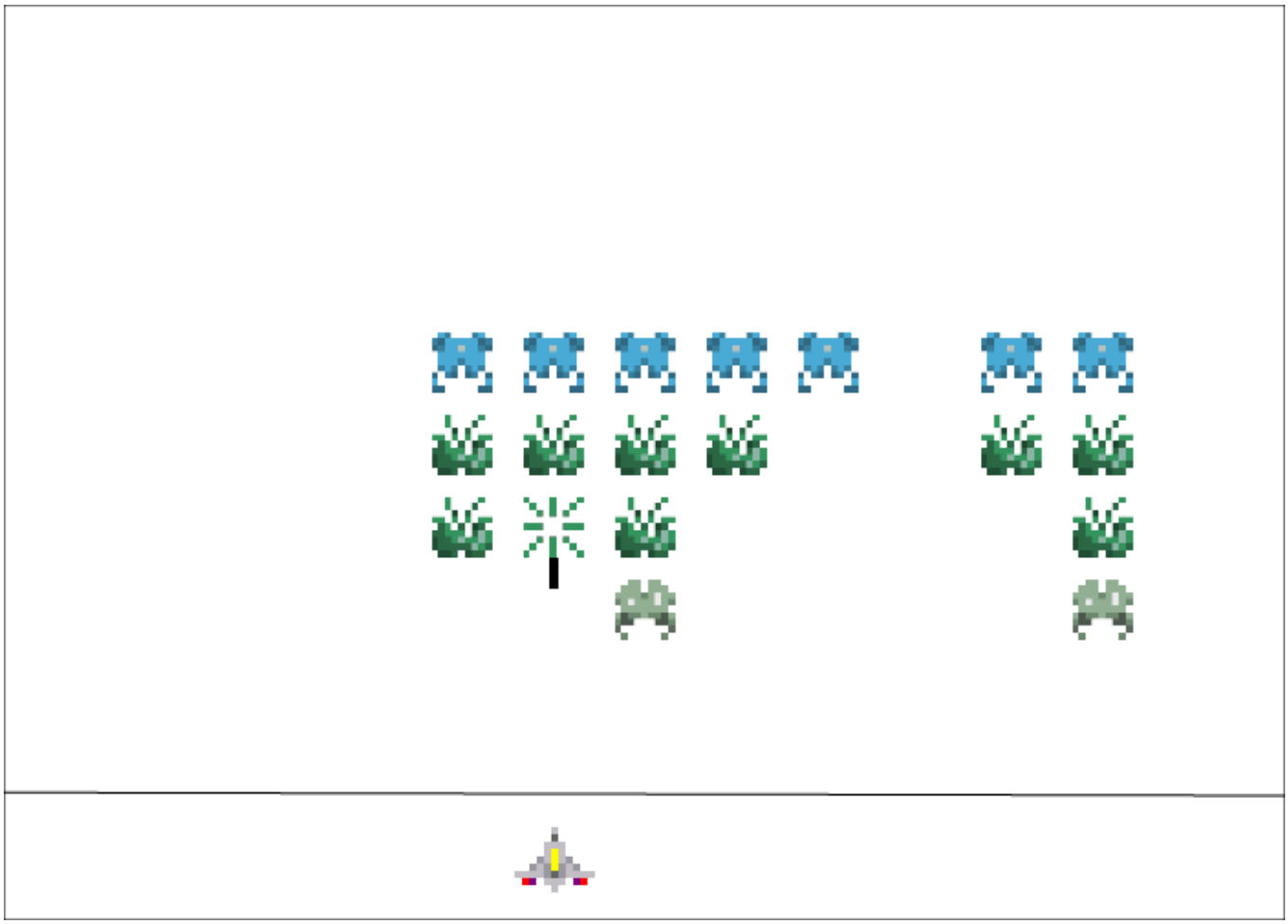
Starting Position

Once the game begins, the aliens march back and forth across the screen. In the beginning they march to the right, moving `ALIEN_H_WALK` (a variable in the file `consts.py`, see below) pixels at a time. When they reach the right hand side of the screen they move down `ALIEN_V_WALK` pixels, and then start marching to the left. They continue this pattern back and forth, dropping down one step whenever they reach the edge of the screen, until they are all destroyed or make it past the defense line. An illustration of the aliens breaking the defense line is shown below.



Breaching the Defense

To protect against the invaders, the player can move the ship left and right, and fire laser bolts. A laser bolt starts from the tip of the ship and moves up in a straight line. If the bolt collides with an alien, that alien is destroyed, as shown below.



Alien Destroyed

The aliens are not defenseless. Every time the aliens take step, there is a random chance that one of the aliens will fire a laser bolt back. That laser bolt will always come from the bottom alien in a column chosen at random. If a laser bolt from an alien collides with the ship, the ship is destroyed. This is shown below.

Ship Destroyed

A player has up to three lives, where one life is lost each time a ship is destroyed. If the player has any lives remaining when the ship is destroyed, the game will briefly pause before starting again. The wave continues until one of three things happens:

- The last alien is destroyed.
- The ship is destroyed and there are no lives remaining.
- Any alien touches the defensive line.

In case 1, the player wins the game. In the other two, the player loses.

Prerequisits

Set up a new conda environment, e.g. named `proj03`, setting python to version `3.9`. The game uses kivy and one needs to install it upfront. Run the following:

```
pip install introcs # this is a util package provided by Cornell CS!
conda install kivy -c conda-forge
```

If you have problems running `kivy` on linux, check out this SO page.

Logistics and Code Overview

The `proj03.zip` archive contains two zip-archives (!):

- `invaders.zip`: The application package, with all the source code. You definitely need to continue to work with this archive.
- `samples.zip`: Several programs that give hints on this assignment.

The first zip-file contains all of the source code necessary to complete the assignment. The second is sample code containing a lot of hints on how to approach some of the harder parts of this assignment, and we reference these samples throughout the instructions.

This assignment is organized as a package with several files. To run the application, change the directory in your command shell to just outside of the folder `invaders` (originating from the first `.zip` file) and type

```
python invaders
```

In this case, Python will run the entire folder. What this really means is that it runs the script in `__main__.py`. This script imports each of the other modules in this folder to create a complex application. To work properly, the invaders folder should contain the following:

- `app.py`: This module contains the controller class `Invaders`. This is the controller that launches the application, and is one of three modules that you must modify for this assignment. While it is the primary controller class, you will note that it has no script code. That is contained in the module `__main__.py` (which you should not modify).

- `wave.py`: This module contains the secondary controller class `Wave`. This class manages a single wave of aliens. It works as a subcontroller, just like the example `subcontroller.py` in the provided sample code. It is another of the three modules that you must modify for this assignment,and the one that will require the most original code.

- `models.py`: This module contains the model classes `Ship`, `Alien` and `Bolt`. These classes are similar to those found in the `pyro.py` demo in the provided sample code. If you want to add other model classes (e.g. power-ups), then you should add those here as well. This is the last of the three files you must modify for this assignment.

- `consts.py`: This module is filled with constants (global variables that should not ever change). It is used by `app.py`, `wave.py`, and `models.py` to ensure that these modules agree on certain important values. It also contains code for adjusting your alien count and speed. You should only modify this file if you are adding additional constants as part of your new feature.

- `game2d.py`: This is a package containing the classes you will use to design you game. These are classes that you will subclass. In particular, the class `Invader` is a subclass of `GameApp` from this package. As part of this assignment, you are expected to read the documentation within the source code. **Under no circumstances should you ever modify this package!**

- `Sounds`: This is a folder of sound effects that you may wish to use as part of your new feature. You are also free to add more if you wish; just put them in this folder. All sounds must be `WAV` files. While we have gotten `MP3` to work on Windows, Python support for MacOS is unreliable.

- `Fonts`: This is a folder of True Type Fonts, should you get tired of the default Kivy font. You can put whatever font you want in this folder, provided it is a `.ttf` file. Other Font formats (such as `.ttc`, `.otf`, or `.dfont`) are not supported. Be very careful with fonts, however, as they are copyrighted in the same way images are. Do not assume that you can include any font that you find on your computer.

- `Images`: This is a folder with image files for the ship and aliens. The `GImage` and `GSprite` classes allow you to include these in your game. You can other images here if you wish. For example, you may wish to draw a background; just remember to draw the background image first.

For the basic game, you only need to worry about the following three files: `app.py`, `wave.py`, and `models.py`. The other files and folders can be left alone. However, when it is time to add a new feature, it helps to understand how all of these fit together.

Assignment Overview

Your game is a subclass of `GameApp`. The parent class does a lot of work for you. You just need to implement three main methods. They are as follows:

- `start()`: Method to initialize the game state attributes.
- `update(dt)`: Method to update the models for the next animation frame.
- `draw()`: Method to draw all models to the screen.

Your goal is to implement all of these methods according to their (provided) specification.

- `start()`: This method should take the place of `__init__`. Because of how Kivy works, initialization code should go here and not in the initializer (which is called before the window is sized properly).

- `update(dt)`: This method should move the position of everything for just one animation step, and resolve any collisions (potentially deleting objects). The speed at which method is called is determined by the (immutable) attribute `fps`, which is set by the constructor. The parameter `dt` (*delta t*) is time in seconds since the last call to update.

- `draw()`: This method is called as soon as update is complete. Implementing this method should be as simple as calling the method `draw`, inherited from `GObject`, on each of the models.

These are the only three methods that you need to implement. But obviously you are not going to put all of your code in those three methods. The result would be an unreadable mess. An important part of this assignment is developing new (helper) methods whenever you need them so that each method is small and manageable. Your grade will depend partly on the design of your program. As one guideline, **your methods should not be longer than 30 lines!**.

You will also need to add methods and attributes to the class `Wave` in `wave.py`, as well as `Ship`, `Alien`, and `Bolt` in `models.py`. These classes are completely empty, though we have given you a lot of hints in the class specification. **You should read all these specifications.**

You may find that you need additional attributes!. All these new instance attributes should be **hidden**. You should list these new attributes and their invariants as single-line comments after the

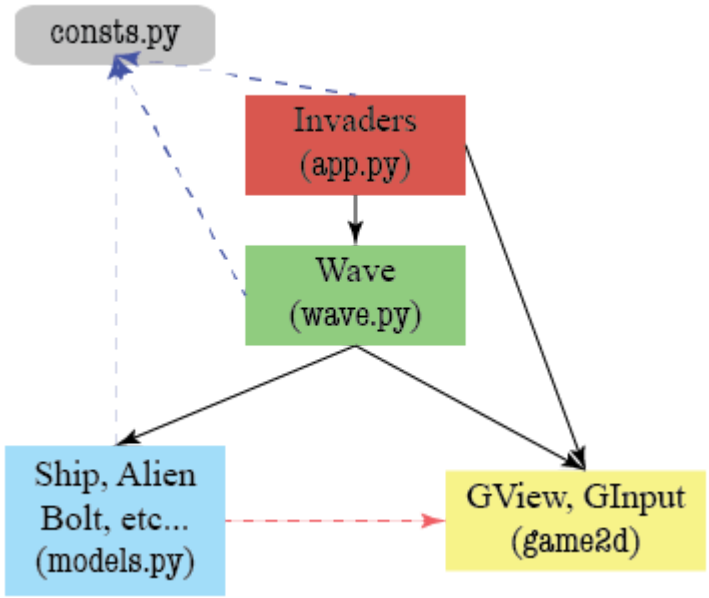
class specification. You do not need to enforce the invariants in the getters and setters, but you must must have them if the attributes are accessed by another class. For example, if the `Wave` class needs to check the velocity of a laser bolt, then you are going to need a getter for the velocity in the `Bolt` class.

Assignment Organization

This assignment follows the so called **model-view-controller pattern** depicted in the figure below. The modules are clearly organized so that each holds models, the view, or a controller. The organization of these files is shown below. The arrows in this diagram mean "accesses". So the `Invaders` controller accesses the view and `Wave` subcontroller. The `Wave` controller accesses the view and the models. This leads to an important separation of files. `Invaders` is never permitted to access anything in `models.py` and `Wave` is never permitted to access anything in `app.py`. This is an important rule that we will check while grading.

You will notice that the models module needs to access the view because it needs the parent `GObject` to perform any drawing. In practice, we often like to separate the model and view to cut down on the number of arrows (less meetings between the various programmers). However, that would make this assignment a lot harder.

Fortunately, the view does not access anything (and should not be modified). This means there are no cycles in this architecture (e.g. `A` accesses `B` accesses `C` accesses `A`). Cyclical accesses are very dangerous and you have to be careful with them in large applications. Avoiding cycles is one of the reasons we draw pictures like this one:



In addition to the four main modules, there is another module with no class or function definitions. It only has constants, which are global variables that do not change. This is imported by the models module and the two controllers. It is a way to help them share information.

When approaching this assignment, you should always be thinking about "what code goes where?". Here are some rough guidelines.

- `Invaders`: This controller does very little. All it does is keep track of the game state (e.g. whether or not the game is paused). Most of the time it just calls the methods in `Wave`, and lets `Wave` do all the work. However, if you need anything between games, like a paused

message or a high score, this goes here. This is similar to the class `MainApp` from the demo `subcontroller.py`.
- `Wave`: This class does all the hard work. In addition to the initializer (which is a proper `__init__`, not `start`), it needs its own update and draw methods. This is a subcontroller, and you should use the demo `subcontroller.py` (in the provided sample code) as a template.

The most complex method will be the update and you will certainly violate the 30-line rule if you do not break it up into helpers. For the basic game, this method will need to do the following:

- Move the ship according to player input.
- March the aliens across the screen.
- Fire a laser bolt from either the ship or an alien.
- Move any laser bolts across the screen.
- Resolve any collisions with a laser bolt.

Each one of these is should be a separate helper function!

The Models

The models just keep track of data. Most of the time, models just have attributes, with getters and setters. Think `Image` from the previous assignment. However, sometimes models have additional methods that perform computation on the data, like `swapPixel`.

The models in this assignment are the game objects on screen: the ship, any aliens, and any laser bolts. Of these three, it is most important to have a class for `Bolt`. `Bolt` needs an additional attribute for its velocity, and you need some extra methods to perform calculations with this velocity.

The classes `Ship` and `Alien` are subclasses of `GImage`. You should not need any new attributes for these two classes. However, you will want to write a `collides` method in each of these classes to detect collisions with a laser bolt.

If your added features include scoring, you will probably need extra attributes in the `Alien` class to track point value. If you add boss aliens or motherships, then you may need additional model classes to display and track them.

Game State

One of the challenges with making an application like this is keeping track of the game state. In the description above, we can identity several distinct phases of the game:

- Before the game starts, and the alien wave has not started.
- When the aliens are set up, but have not started to move.
- While the game is ongoing, and the aliens are on the march.
- While the game is paused (e.g. to show a message).
- While the game is creating a new ship to replace the old one.
- After the game is over.

Keeping these phases straight is an important part of implementing the game. You need this information to implement `update` in `Invaders` correctly. For example, whenever the game is

ongoing, the method `update` should instruct the `Wave` object to move the ship. However, if the game has just started, there is no `Wave` object yet, and the method `update` should create one.

For your convenience, we have provided you with constants for six states:

- `STATE_INACTIVE`, before a a wave has started.
- `STATE_NEWWAVE`, when it is time to create a new wave of aliens.
- `STATE_ACTIVE`, when the game is ongoing and the aliens are marching.
- `STATE_PAUSED`, when the game is paused to display a message.
- `STATE_CONTINUE`, when the player is waiting for a new ship.
- `STATE_COMPLETE`, when the game is over.

All of these constants are available in `consts.py`. The current application state should be stored in a hidden attribute `_state` inside `Invaders`. You are free to add more states when you add more features. However, your basic game should stick to these six states.

The rules for changing between these six states are outlined in the specification of method `update` in `Invaders`. You should read that in its entirety. However, we will cover these rules in the instructions below as well.

The Basic Game

We have divided these instructions into two parts. This first part covers the basic things that you must implement just to get the game running. Afterwards, we talk about additional features you can put into your game. These extra features can earn you extra credit on this assignment.

You should focus most of your effort on the basic game. This is the bulk of your grade. And if you do everything correct on the basic game, you can still earn a 100 on this assignment. The value of extra credit is typically limited to no more than 5 points. Furthermore, unless your features are incredibly innovative, we will not give you more than a 100.

Review the Constants

The very first thing that you should do is read the file `consts.py`. If you ever need a value like the size of the ship, the size of the game window, or so on, this is where you go. When writing code, you should always use the constants, not raw numbers (or "magic numbers," as we call them). Magic numbers make your code hard to debug, and if you make a change (e.g. to make the ship bigger), you have no idea about all of the locations in your code that need to be changed.

With that said, you are welcome to change any of these numbers if you wish. You are also encouraged to add more constants if you think of other numeric values that you need. Anytime that you find yourself putting a number in your code, ask yourself whether or not it would make sense as a constant.

Create a Welcome Screen

We start with a simple warm-up to get you used to defining state and drawing graphics elements. When the player starts the application, they should be greeted by a welcome screen. When you work on your add more features, you can embellish your welcome screen to be as fancy as you wish. But for now, keep it simple. Your initial welcome screen is a simple text message.

Press ‘S’ to Play

Because the welcome message is before any game has started, it belongs in the `Invaders` class, not the `Wave` class. You are already seeing how we separate what goes where.

The text message will look something like the one above. It does not need to say “Press ‘S’ to play”. It could say something else, as long as it is clear that the user should press a key on the keyboard to continue. However, we recommend against allowing the user to press any key, since in later steps that will make it easy for the user to accidentally miss an important message.

To create a text message, you need to create a `GLabel` and store in it an attribute. If you read the class invariant for `Invaders`, you will see an attribute named `_text`. This is for any messages to display to the player. If you wish you may rename this attribute, as long as you make it clear in the class specification.

Since the welcome message should appear as soon as you start the game, it should be created in the method `start`, the first important method of the class `Invaders`. When creating your message, you will want to set things like the font size and position of the text. If you are unsure of how to do this, look at the class `MainApp` from the demo `subcontroller.py` in the provided sample code.

As you can see from the documentation for `GLabel` and `GObject`, graphics objects have a lot of attributes to specify things such as position, size, color, font style, and so on. You should experiment with these attributes to get the welcome screen that you want. The key thing to remember is that – in Kivy – screen coordinates start from the bottom-left corner of the window (and not the center as was the case with the turtle). Note that `x` and `y` are the *center* of the label. If you want to place the `left` edge, use the attribute `left` instead of `x`.

Simply adding this code to `start` is not enough. If you were to run the application right now, all you would see is a blank white window. You have to tell Python what to draw. To do this, simply add the line

```
self._text.draw(self.view)
```

to the method `draw` in `Invaders`. The (non-hidden) attribute `view` is a reference to the window. Hence this method call instructs Python to draw this text label in the window. Now run the application and check if you see your welcome message appears.

Initializing Game State

The other thing that you have to do in the beginning is initialize the game state. The attribute `_state` (included in the class specification) should start out as `STATE_INACTIVE`. That way we know that the game is not ongoing, and the program should (not yet) be attempting to animate anything on the screen. In addition, the other attributes listed (particularly `_wave`) should be `None`, since we have not done anything yet.

The `_state` attribute is an important part of many of the invariants in this game. In particular, we want your new attribute for the welcome message to have the following invariant:

- If the state is `STATE_INACTIVE`, then there is a welcome message
- If the state is not `STATE_INACTIVE`, the welcome message is `None`.

Does your `start()` method satisfy this invariant?

Dismissing the Welcome Screen

The welcome screen should not show up forever. The player should be able to dismiss the welcome screen (and start a new game) when he or she presses a key. To respond to keyboard events, you will need the attribute `input`, which is an instance of `GInput`. This class has several methods for identifying what keys are currently pressed.

The method `update(dt)` is called every 16 milliseconds. Checking if the key is down will return **a lot of matches**. You want to detect a *key press* which is the first time the key is held down. Remember the method (`is_key_pressed`) to do this.

When you do detect a key press, then you should change the state `STATE_INACTIVE` to `STATE_NEWWAVE`. This will start a new game. You are not ready to actually write the code to start the game, but switching states is an important first activity.

Invariants must be satisfied at the end of every method, so you need to assign `None` to the welcome message as well. This will require a simple change to method `draw()` to keep it from crashing (you cannot draw `None`). Once you have done that, run the application. Does the message disappear when you press a key?

Documenting your New Attributes

When working on the steps above, you most likely had to add new attributes beyond the ones that we have provided. Whenever you add a new attribute, you must add it and its corresponding invariant as a comment after the class specification (these comments are the class invariant). Add it just after the comment stating `ADD MORE ATTRIBUTES`, to make it easier for you (and the instructors ...) to find them.

Create a Wave of Aliens

The state `STATE_NEWWAVE` is only supposed *to last one animation frame*. During that frame you should do the following:

- Create a new `Wave` object.
- Assign that Wave object to the attribute `_wave`.
- Switch the state to `STATE_ACTIVE`.

You do not need to worry about the details of `STATE_ACTIVE` for now. However it is very important that you **only create a new `Wave` object if the state is `STATE_NEWWAVE`**. If you continue to create a `Wave` object in `STATE_ACTIVE`, that will cause many problems down the line (which you will not notice until much later).

Right now, the constructor for the subcontroller `Wave` does not do anything. That is because you have not written an initializer yet. Eventually, your initializer is going to create the aliens and the

ship. Right now, we are just going to focus on the aliens.

Creating a Single Alien

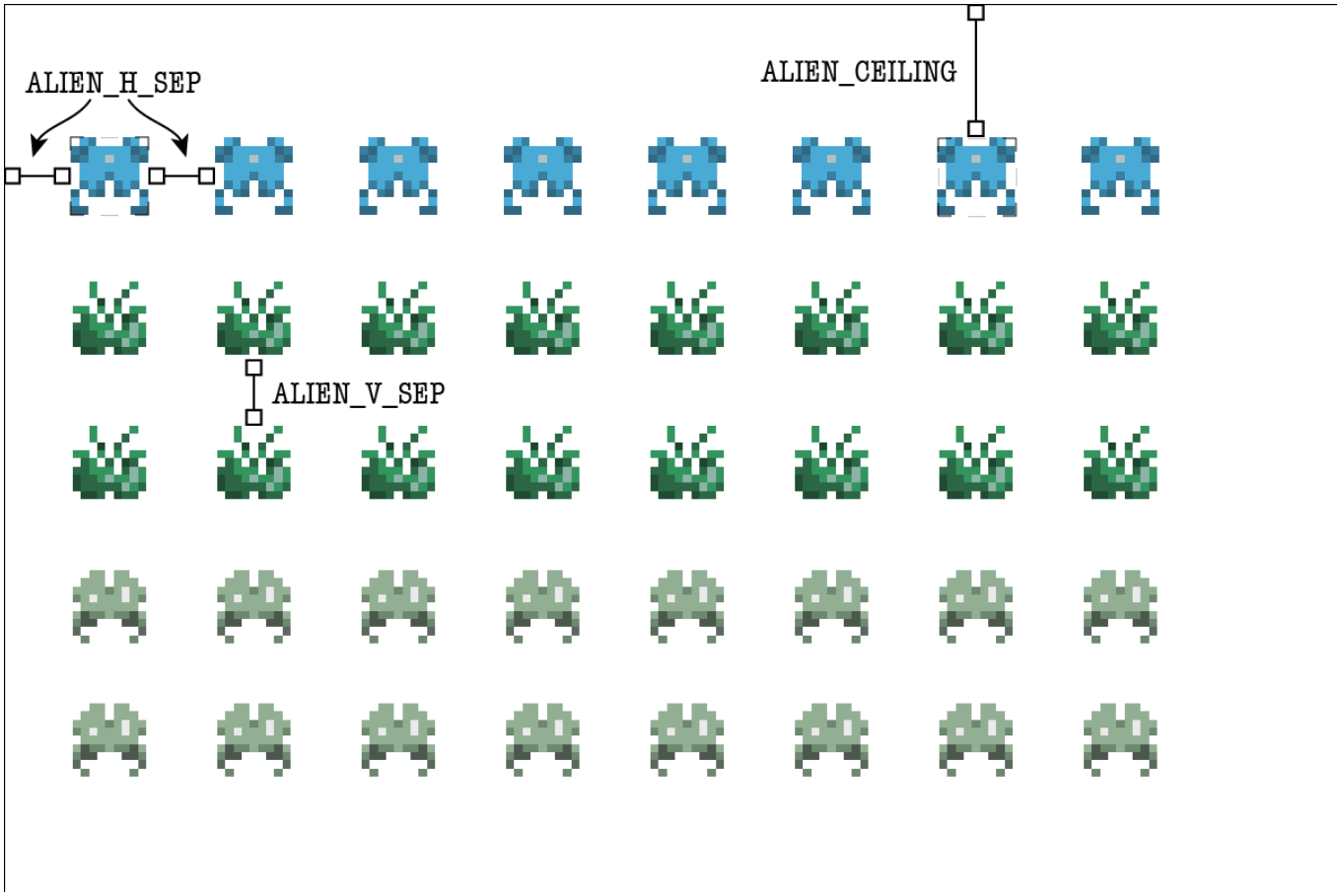
While you have not yet completed the definition of class `Alien` yet, we have gotten you started in the module `models.py`. In particular, the class `Alien` is a subclass of `GImage`. That means it inherits all of its attributes and methods, including the initializer. Hence (unless you override methods to do otherwise), you create and draw aliens the same way you create and draw a `GImage` object.

To define an image, use the attributes `x`, `y`, `width`, `height`, and `source` to specify how it looks on screen. The first four attributes are just like `GLabel`, while source specifies an image file in the `Images` folder. As with the label, you can either assign the attributes after the object is created or assign them in the constructor using keywords. Keyword arguments work like default arguments in that you write `param = value`.

Creating a Wave of Aliens

Read the specification for `Wave`. You will see that it contains a two-dimensional list of `Alien` objects. Your method `__init__` should fill this 2d list with aliens. Because there are other things you will need to do in `__init__`, you should probably make a helper method that does this initialization. Otherwise, `__init__` may go over 30 lines when you add extra feature later.

Look at the constants in `consts.py`. You need to draw `ALIEN_ROWS` rows of aliens with `ALIENS_IN_ROW` many aliens in each row. The module also includes constants for how big to make the aliens and how much space to put between them.



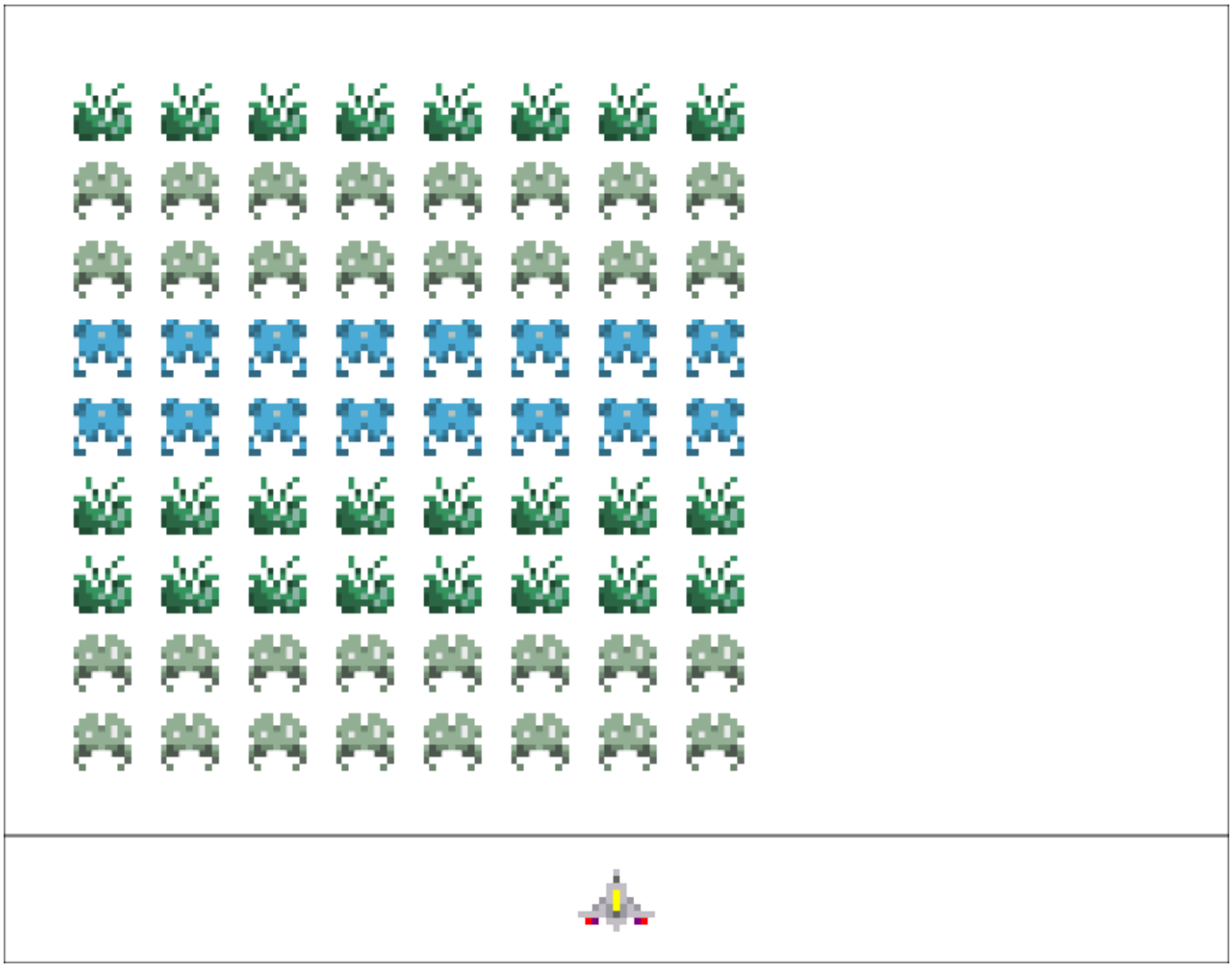
When you are ready, you should set up the aliens as shown in the game overview. Each row of aliens will be a list of `Alien` objects, and each row is an element of the attribute `_aliens`. You can store the rows bottom-up or top-down; it does not matter. The alien positions should line up neatly into rows and columns. The left edge of the wave should be `ALIEN_H_SEP` from the left edge of the window, and the top edge of the wave should be `ALIEN_CEILING` from the top of the window.

You should be able to code this as a straightforward nested loop. There are only three things to keep track of in the loop:

- The `x` position of the next alien.
- The `y` position of the next alien.
- The `image` of the next alien.

All other values are defined by constants. To compute the first two, remember that the aliens should be `ALIEN_H_SEP` and `ALIEN_V_SEP` apart from each other. For the images, however, you should do something different. All aliens in a row should use the same image and the image should be constant for two rows. Images should go from bottom to top. So if there are five rows, that is two rows of `alien1.png`, another two rows of `alien2.png`, and just one row of `alien3.png`.

We have only provided you with three alien images. However, you are free to add more or even change the existing images (within copyright limitations, see above). Regardless, you should be prepared to draw any number of rows of aliens. So when you run out of aliens, you need to loop back to the beginning. For example, suppose you had four alien images but needed nine rows of aliens. You would have two rows of `alien1.png`, `alien2.png`, `alien3.png`, and `alien4.png` in that order. Then you would have one last row of `alien1.png` at the top. See the example below for three images and nine rows. You will find the constant `ALIEN_IMAGES` helpful here.



Drawing Aliens

Once again, creating an `Alien` object is not enough to draw it on the screen. But drawing the aliens is a bit more complicated than drawing the welcome message. The aliens are (hidden) attributes in `Wave`. While the code

```
for row in self._wave._aliens:
  for alien in row:
    alien.draw(self.view)
```

works (and you should try it out), it is not allowed: A class of one module can never accesses the hidden attributes of an object of a class in a different module.

This is the purpose of adding a `draw` method to class `Wave`. The `draw` method in `Invaders` should call the `draw` method in `Wave`, and this in turn should call the the `draw` method for each alien (defined in `GObject`).

However, only `Invaders` has access to the attribute `view`, which is necessary for drawing. The class `Wave` cannot directly access any attributes in `Invaders`. If a method in `Wave` needs an attribute from `Invaders`, then `Invaders` must provide that attribute as an argument in the method call. This means that the `draw` method in `Wave` needs to have `view` as a parameter, and the code in `Invaders` should look like this.

```
self._wave.draw(self.view)
```

Notice this is very similar to how we draw `GObject` objects.

Testing Your Code

When you are testing the later parts of this assignment, you should play with just 3-4 aliens per row and 1-2 rows of aliens. This will save time and let you quickly see whether you can successfully win or lose. If you play with the default number of aliens (5 rows and 12 aliens per row), then each game will take a while to test.

You might assume that testing in this manner requires you to change the values of the global constants that give the number of rows and number of aliens in a row. This is also undesirable, a you might forget to change them back. Instead, some *clever* (?!) code in `consts.py` allows you to change these constants when you start the application.

When you run your application (again, assuming that it is in a folder called `invaders`) try the command

```
python invaders 2 3
```

When you do this, Python changes the value of `ALIEN_ROWS` to `2` and the value of `ALIENS_IN_ROW` to 3`.

You should make sure that your creation of the rows of aliens works with any number of rows and any number of aliens in each row (e.g. 1, 2, ..., 5, and perhaps more). This is one of the things an instructor could be testing when running your program. Technically, the player should lose if the number of aliens causes them to drop below the defense line, but you can ignore that issue in this step.

Create (and Animate) the Ship

Next you need to create the player ship. Again, this is to be stored in an attribute of class `Wave`. That means that you must create it in the `__init__` method of `Wave` and modify your drawing code so that it appears. The ship should be an object of class `Ship` (included in `models.py`), which is once again a subclass of `GImage`.

Creating the Ship

The ship dimensions and position are fully specified by constants in `consts.py`. The ship should be centered horizontally and the bottom should be `SHIP_BOTTOM` from the bottom of the window. You are free to change these constants as you wish, but use them when making the ship.

Remember to draw the ship object in the method `draw`.

Creating the Defense Line

The defense line is the line above the ship that it is defending from the aliens. It is `DEFENSE_LINE` pixels above the bottom of the window. You should create and draw this line. A line is represented by a `GPath` object.

The primary attributes for `GPath` are `points` and `linewidth`. The `linewidth` is the width of the line. Make it larger than 1 (the default) for a thicker line. The attribute `points` is an even-length list of numbers defining the line segment. So if you want the line segment from (0,10) to (20,30), the `points` attribute would be `[0,10,20,30]`.

Remember to draw the defense line in the method draw.

Animating the Ship

To animate the ship, you will need to take into account the player's key presses. The ship only moves when the player presses (or holds down) a key to make it move. By default, we assume that the player will use the left and right arrow keys to move the ship. However, if you prefer `WASD` controls or some other control scheme, that is okay.

To see how to control the ship, you should look at the `arrow.py` demo from the provided sample code. This example shows how to check if the arrow keys are pressed, and how to use that to animate a shape. Note that this is actually easier than dismissing the welcome message. We do not care if a key press is the first one. The ship will continue to move so long as we hold down a key.

The ship movement takes place every animation frame. That is why you want to put it the `update` method of `Wave`. Remember that this method must be called within the update method of `Invaders`, or nothing will happen. Again, see the `subcontroller.py` demo from the sample code to understand what is asked for.

To check the keyboard, the method `update` in `Wave` will need to access the `input` attribute of `Invaders`, which is an instance of `GInput`. Again, since `Wave` is not allowed to access any of the attributes of `Invaders`, that means you need to pass input as an argument in this method call.

In the basic game, the ship should only move left or right, not up and down. Add to the `x` attribute to get it to move right, and subtract from it to move it left. The amount you add or subtract is up to you, though we have provided the constant `SHIP_MOVEMENT` as a suggestion. If the value is too small, then the ship will be slow and sluggish. If the value is too large, then the ship will zip across the screen and be hard to control.

Restricting Movement

You should ensure that the ship stays completely on the board even if the player continues to hold down a key. If you do not do this, the ship is going to be completely lost once it goes off screen. Use the functions `min` and `max` to implement this feature.

Your implementation should only allow the ship to be moved when the state is active. This means that `update` in `Invaders` is starting to get more complicated. At this point you might want to start thinking about helper methods to organize your code better. Look at the `state.py` demo from the provided sample code for ideas on how to organize your code. In particular, you want a method to determine the current state, and then helpers for each of the states.

Walk the Aliens Back-and-Forth

Now that the set-up is complete, it is time to start invading. To invade, the aliens have to march back and forth across the screen. Fortunately, you have already gotten the ship moving, so you

know how to move things. The aliens are similar, with just a few important differences.

Moving the Aliens to the Right

When aliens move, they always move the same amount: `ALIEN_H_WALK`. To move the aliens to the right, simply add this value to the `x` position of each alien. However, aliens have one important feature that makes them different from the ship. They do not move every animation frame.

You will notice that `consts.py` has an attribute called `ALIEN_SPEED`. This is how fast the aliens walk. If the value is `1.0`, they make a step every full second. If it is `0.5`, they move every half of a second. The smaller this number is, the faster they move (increasing the difficulty).

This means that you have to keep track of how long it has been between steps before you move the aliens again. How do you do that? In the list of attributes for `Wave`, you will see an attribute called `_time`. This counts the number of seconds since the last alien step. At the start, and each time the aliens move, it is reset to `0`. Otherwise, you add the number of seconds that have passed to `_time`, and you do not move the aliens. When this value is bigger than `ALIEN_SPEED`, you move the aliens again.

So your next question should be how to count the numbers of seconds that have passed. You will notice that `update` in `Invaders` has an attribute called `dt`. This counts the number of seconds that have passed since the last animation frame (at 60 frames as second, it is somewhere around 0.017 each time). This is the value that you should be summing and adding to the attribute `_time` in `Wave`. You can do this via a setter or by passing `dt` as an argument to the `update` method in `Wave`.

As you work on this code, you will discover that the `update` method in `Wave` is starting to get very long. It is time to start thinking about breaking this method into helpers.

Moving the Aliens Back

Right now, your aliens will march all the way off the screen. You do not want that. When the wave of aliens reaches the right edge of the screen, you want to move the aliens down by `ALIEN_V_WALK`. Then you need to start walking them back to the left.

This means that at each step, you need to find the rightmost alien in the wave. Check if its right edge is closer than `ALIEN_H_SEP` to the right edge of the window. If so, move all the aliens all down by `ALIEN_V_WALK`. You should also move them back to the left so that there is `ALIEN_H_SEP` distance between the closest alien and the right of the window (the aliens should never go offscreen). Do this instead of the normal movement (not in addition to), or else the aliens will move down diagonally.

Repeat this process back and forth. When the aliens get to far to the left, move them down and start moving right again. This suggests that `Wave` needs an extra attribute that keeps track of whether the aliens are moving right or left.

Testing the Speed

The speed is what determines the difficulty of the game, so it is a good idea to test the game at different speeds. Just as you can change the number of rows and aliens per row, you can also change the speed from the command line.

When you run your application (again, assuming that it is in a folder called `invaders`) try the command

```
python invaders 2 3 0.5
```

When you do this, Python changes the value of `ALIEN_ROWS` to `2`, the value of `ALIENS_IN_ROW` to `3`, and the value of `ALIEN_SPEED` to `0.5`. By varying this third number, you can make the aliens march faster or slower. If your aliens do not move at different speeds when you try this, you have a problem that you need to fix before continuing.

Fire Bolts from the Ship

Now that the aliens are marching, it is time to fight back. You are going to give the ship the ability to fire laser bolts.

Creating a Laser Bolt

The game should create a laser bolt when the player presses a fire key. The standard keys for firing the laser are space or up-arrow. However, it can be anything that you want so long as it is clear to the player.

When you detect a fire command, you should create a `Bolt` object. This `Bolt` object should have the same `x` position as the ship, and it should be placed right in front of the ship's nose. Use the constants in `const.py` to calculate this from the ship and bolt dimensions.

Unlike the ship and alien classes, `Bolt` is a subclass of `GRectangle`. It is very similar to `GImage`, except that you specify a `fillcolor` and a `linecolor` instead of a image file. We do not expect your laser bolt to be anything fancy.

`Bolt` objects are stored in the `_bolts` attribute of `Wave`. This is a one-dimensional list that may be empty (because there are no bolts on the screen). You `draw` the bolts in much the same way that you draw the aliens.

Customizing the Bolt Class

Up until now, we have not needed to add additional attributes to any of our model classes. While the aliens march across the screen, they all march in lock-step. However, some laser bolts will be moving upwards, and some will be moving downwards (e.g. the bolts fired by the aliens). Therefore, we need to add an attribute in `Bolt` to keep track of the velocity.

This means that you will need to override the `__init__` method in `GRectangle` and provide your own. We do not care what the parameters for this method are. That is up to you. The velocity should either be `BOLT_SPEED` for bolts that are going up, or `-BOLT_SPEED` for bolts going down.

Because we will later have the aliens fire their own bolts, you want to be able to distinguish bolts shot from the player from those shot from an alien. We suggest that you add a method like `isPlayerBolt` to the class. Remember that all bolts that travel up are fired by the player.

Moving the Laser Bolt

You move a laser bolt in much the same way that you moved the ship and aliens. You just add the velocity to the `y` position. Like the ship, and unlike the aliens, you move the bolt every animation frame. You do not have to stagger the movement.

Removing the Laser Bolt

Right now, your laser bolt will pass through aliens. That is okay. We will worry about the aliens later. However, we do want you to delete the bolt when it goes off screen. To do this, simply check if the bottom of the bolt is above the height of the game window. If so, delete the bolt from the list `_bolts` in `Wave`. The bolt will no longer be drawn and the object will be deleted. See the demo `pyro.py` from the provided sample code for an example of how to do this.

Restricting the Rate of Fire

There is an important restriction on laser bolts that you must implement at this time. The player may only have one laser bolt on the screen at a time. The player cannot fire a new bolt until this bolt goes off screen and is deleted. Otherwise, the player has a machine gun (or even a beam weapon) and the game becomes trivially easy.

Keep in mind that `_bolts` will eventually have multiple laser bolts in the list even though only one of them will belong to the player. This is the reason for the `isPlayerBolt` method we suggested. Look at every bolt in the list. If one of them is a player bolt, the player cannot fire. Otherwise, the player is free to fire.

Fire Bolts from the Aliens

The ship should not be only thing to fire laser bolts. The aliens get to fire bolts as well. The process should be very similar to the laser bolts for the ship. You create a `Bolt` object (though this time its velocity is `-BOLT_SPEED`), add it to the list `_bolts`, draw it, and move it across the screen. An alien bolt should be removed when it goes off the screen, which is when the top of the bolt is less than 0.

However, there is a tricky part about the alien bolts. There a lot of aliens. So you need to control which aliens fire and when. First of all, aliens should only fire when they walk. That means that the faster they walk, the faster they fire. Alien speed is the main difficulty in Space Invaders, so this makes sense.

Picking When to Fire

While we could have the aliens fire every time they step, that would make the game hard. But we also do not want them to fire too slow, as that makes the game to easy. What we want is for the aliens to fire randomly. Sometimes they fire slow and sometimes they fire fast.

In `consts.py` you will see the `BOLT_RATE`. This is the maximum number of steps between alien shots. So if it is `5`, the aliens can take up to five steps before one of them shoots. If it is `10`, they can take up to ten steps between shots.

At the start of the wave, you should use the random module to pick a number between 1 and `BOLT_RATE`. That number should be stored in an attribute in `Wave`; it is the number of steps until the aliens fire. When the aliens are ready to fire, you create a `Bolt`. You will also generate another number between 1 and `BOLT_RATE` for the time of the next laser bolt.

Picking Who to Fire

The laser bolts have to come from an alien. To pick an alien, you should first pick a nonempty column of aliens at random. Note the invariant of `_aliens` says that the table of aliens can have `None` as an entry (which happens when an alien is destroyed). You do not want to pick a column where all of the rows are `None`. But otherwise, you should use the random module to pick the column.

Next, you should identify the bottom-most alien in the column. Again, because aliens may be destroyed, you should not pick a position where the alien is `None`. Search the column and find the alien on the bottom.

Once you have done that, you can create the `Bolt` object. The bolt should have the same `x` position as the alien at the bottom. The top of the bolt should be just below the feet of the alien.

Handle Bolt Collisions

You should have a lot of laser bolts flying back and forth, but they are all pretty useless right now. It is time to give them some bite. You need to detect collisions and remove any aliens (and the player) that are killed by a bolt.

How do you detect collisions? Suppose the bolts were a single point (x,y) rather than a rectangle. Then, for any `GObject gobj`, the method call

```
gobj.contains((x,y))
```

returns `True` if the point is inside of the object and `False` if it is not. Since both `Ship` and `Alien` are sub(sub)classes of `GObject`, they inherit this method. Note that `contains` takes an single argument which is a list or tuple, not two arguments. Alternatively, you could use a `Point2` object instead.

However, the bolt is not a single point. It occupies physical area, so it may collide with something on the screen even though its center does not. The easiest thing to do — which is typical of the simplifying assumptions made in real computer games — is to check the four corners of the bolt and see if any of them are inside of the object. Because the bolt is smaller than either the ship or an alien, we can guarantee that one of these four corners is inside of the object during a collision.

You should add the method `collides` to both the class `Ship` and `Alien`. This method returns `True` if a bolt (fired by the other side) has collided with the instance `self` of that class. For example, here is our specification of `collides` in `Alien`:

```
def collides(self,bolt):
    """
    Returns True if the player bolt collides with this alien

    This method returns False if bolt was not fired by the player.

    Parameter bolt: The laser bolt to check
    Precondition: bolt is of class Bolt
    """
```

These methods will be similar, except for one important difference. The method in `Ship` will only return `True` if the bolt was fired by an alien (e.g. it is going down). In `Alien` it will only return `True` if the bolt was fired by the ship (e.g. it is going up).

If you detect a collision between a bolt and an alien, you should set that position of the table `_aliens` to `None` and remove the bolt from the list. Similarly, if you detect a collision between a bolt and the ship, you should set `_ship` to `None` and remove the bolt from the list.

When you make this change, you are probably going to have to go back and make some changes to other parts of your code. You are going to start running into `NoneType` errors in the code for marching the aliens and drawing the aliens. That is because some of the aliens may now be `None`. You have to prepare your code to deal with this. Every time you access `_ship` or an element of `_aliens`, you must check for `None` before continuing.

Once you have completed this, you should be able to start playing a game.

Finish the Game

You now have a (mostly) working game. However, there are two minor details left for you to take care of before you can say that the game is truly finished.

Managing Player Lives

When the `_ship` attribute becomes `None`, the player cannot really do much any more. What should happen in this case is that the player should lose a life. The player should have three lives before losing the game. You will notice an attribute call `_lives` in `Wave` for managing these lives.

If the player still has lives left after losing a ship, the `update` method in `Invaders` should change the state to `STATE_PAUSED` and display a message (as you did on the welcome screen) that the player press `'S'` to continue. This state does nothing (not even update `Wave`) until the player presses this key. As soon as the player presses this key, switch the state back to `STATE_ACTIVE` and start the game again. The wave should continue where it left off.

Ending the Wave

Eventually the wave of aliens will end. Each time the player loses a life, you need to check if there are any lives left. If not, the game is over. Additionally, the game is over if (1) all the aliens are killed or (2) any alien dips below the defense line. Both of the latter need to be checked in the `update` method of `Wave`. You might want to add an extra attribute to keep track of when either of these happen.

When the wave ends, and the player has either won or lost, you should put up one last message. Use a `GLabel` to put up a congratulating (or admonishing) message. Finally, you should change the state one last time to indicate that the game is over. This is the purpose of the state `STATE_COMPLETE`.

Additional Features

Our suggested timeline gives you just enough time to work on this assignment. But if you are able to finish early, then we will give you the opportunity to extend the game and try to make it more

fun. While this is not required, this is an opportunity for extra credit. We will award up to 5 points for extra credit, according to how interesting your features are.

When you add new features, you might find yourself reorganizing a lot of the code above. You may add new methods or change any of the methods you have written. You may add new classes. For example, you might want to add a class for a boss alien that is different from a regular alien. You can even completely rewrite existing classes like `Alien` to support cool animations.

You are allowed to change anything you want so long as you update the specifications to reflect the changes. There are only four things that you are not allowed to change.

- The win (destroy all aliens) and lose (lose all lives, break the defense line) conditions must remain the same.
- The aliens must be generated two rows at a time, bottom to top (though each row can have some special aliens).
- The aliens must start marching at the speed `ALIEN_SPEED`.
- The ship should never move off screen (aliens can, if creating attack waves).

Everything else is fair game. However, we highly suggest that you save a copy of the basic game in a separate folder before you start to make major changes. That way you have something to revert to if things go seriously awry when implementing your features. Also, we suggest that you make sure to comment your code well in order to keep track of where you are in the coding process. As we have said before, make sure your basic game is working properly before you start on new features.

Possible Features

Here are some possible ways to extend the game, though you should not be constrained by any of them. Make the game you want to make. While this is a fairly simple game, the design space is wide open with possibilities.

Multiple Waves

The easiest feature is to implement multiple waves. If the player completes a wave without losing all the ship lives, it is time for a new wave of aliens. This is really easy, since all you have to do is make a new `Wave` object.

For this to count as a proper feature, we want each wave to increase the alien speed. The alien speed is what makes the game difficult, and the game should get more difficult with each wave. You will have to add some attributes to Wave to keep track of this. You will also need modify the `__init__` method so that `Invaders` can increase the speed each time it makes a new wave.

Sound Effects

Another easy feature is to add appropriate sounds for game events. We have provided several audio files with the source code. You will want to look at them, but you are not restricted to only those sounds. Remember to not use unlicensed copyrighted material.

To load an audio file, you simply create a `Sound` object as follows:

```
pewSound = Sound('pew1.wav')
```

Once it is loaded, you can play it whenever you want (such as when the ship fires the laser) by calling `pewSound.play()`. The sound might get monotonous after awhile, so make the sounds vary, and figure out a way to let the user turn sound off (and on).

Read the specification to see how to use `Sound` objects. You cannot replay a sound until the current version of the sound stops. So if you want to play the same sound multiple times simultaneously (such as if two aliens fire simultaneously), you will need two different Sound objects for the same sound file. Proper game audio can get really complicated. Maybe a GPT can produce cool sound snippets?

Important: Loading sounds can take a while. We recommend that you load all sounds (e.g. create associated the Sound objects) you plan to use at either the start of the game or the start of a wave. Do not try to create a Sound object just before you play it.

Dynamic Aliens

The aliens in Space Invaders do not just speed up when you start a new wave. They also speed up as you kill aliens. This is actually a result of a bug in the original Space Invaders, but it was left in because it made the game fun.

To make the aliens move faster, you need to make the speed value smaller. A speed value of `1.0` means they march every second, while `0.5` means they march every half second. So the easiest way to increase the speed is to multiply by some number less than one each time you kill an alien. Do not make the multiplication factor too small, or else the aliens will jump to lightning speed. You will discover that a factor of `0.97` is reasonably challenging.

Player Score

A large part of the challenge of Space Invaders is getting a high score. Aliens in the front are worth a few points and aliens in the back are worth less. You should display the score at all times using a `GLabel` object. Where you display it is up to you (except do not keep the player from seeing the aliens or ship). Please do not make a new `GLabel` object each time the score changes. This will slow down the program tremendously. Simply change the text attribute in your `GLabel` object.

In classic Space Invaders, this score carries over between waves. If you chose to implement multiple waves, then each wave after the first should start with the score from the previous wave.

Animation

In the video at the start of the animation, you will notice that the aliens are animated as they move and that they have simple explosion animations. These animations are actually made available to you as filmstrips. A filmstrip is single image file that holds multiple animation frames. These frames are arranged in a rectangular grid where each frame has equal size. For example, alien-strip1.png is a 3×2 grid of animation frames. The first row is the alien walking. The other two rows animate the alien explosion.

To use a filmstrip, you need to make a `GSprite` object as follows:

```
alien = GSprite(source='alien-strip1.png',format=(3,2),width= ... ,height= ... )
```

The `GSprite` class has an attribute called frame which tracks which frame in the filmstrip is currently displayed. At the start, the frame is always `0`, showing the alien in the top-left corner.

To walk the alien, you just add the following code to your march update:

```
alien.frame = (alien.frame+1) % 2
```

You are free to use these filmstrips (also called sprite sheets) or make your own. The Stitches website is a great resource for making filmstrips.

If you really want to go beyond, learn how to use coroutines to create animations in Python.

Defense Barriers

The classic Space Invaders has defense barriers that the player can hide behind. These barriers are damaged by laser bolts fired by either the player or the aliens. They provide a bit of respite when the aliens are moving very fast.

The original game destroys the barriers a pixel at a time. This is extremely hard to do with the classes that we have provided you, and impossible without accessing hidden attributes (you have to perform **texture blitting** on the `_texture` attribute in `GImage` or `GSprite`). So we do not recommend this approach at all.

However, a simpler approach is to just give the defense barriers a health meter. As the bolts hit the barrier it is weakened and eventually destroyed. You can even use `GSprite` to change the barrier appearance as it weakens.

Your Imagination

What else have you always wanted a game like this to do? Do you want to have swooping aliens like in Galaxian and Galaga? Should aliens drop power-ups that give the ship rapid laser fire? Do not go too wild with the power-ups, however. We much prefer a few innovations that greatly improvevthe play as opposed to a screen filled with gizmos.

Again, you can make any modifications to the gameplay you want, but the core gameplay of ships and marching aliens should be there.