Brian Sandon

IGME 202 Section 2

Asteroids

List of user functionality:

* The main menu and score screen can be navigated through by clicking on the buttons provided.
* Game mode:
  + The mouse can be used to orient the ship.
    - Note: the ship’s orientation can change sharply when it wraps around the screen.
  + The arrow keys can be used to accelerate the ship up, down, left, and right relative to the ship’s current orientation.
    - Note: since the ship always orients towards the mouse pointer movement to the left and right tends to curve.
  + The space bar can be used to shoot forwards, there is a 30 frame delay before another shot can be made.

Reasoning behind design choices:

I designed my game with graphics similar to the original Asteroids game. This choice was made for two reasons: both to pay homage to the original and I feel that the simple vector style would give the game a more polished appearance then if I tried to use more complicated vector art or bitmaps.

The game is mostly done in white on a black background, the notable exceptions where other colors are used all help provide significant information for the player. When the mouse curser is over a button, the button turns green to indicate that the mouse is hovering on that button. When the player collides with an asteroid and takes damage, the player gets 100 frames of invulnerability in which the game ignores any collisions the ship makes with asteroids. During this time the player avatar appears red, and this color fades as the invulnerability wears off. Similarly, bullets (which only exist on the screen for limited time) slowly fade in color until they are nearly invisible against the background to indicate their limited lifespans. Bullets can reach the point where they are near invisible on the screen and still collide with an asteroid, making it appear an asteroid spontaneously split apart without any explanation. Power-ups also break the black and white convention as explained in the above and beyond section.

The game uses position and velocity vectors in the motion of all objects. Asteroids do not head straight after the player (like stated in the rubric), instead they experience acceleration towards the player character. This means that asteroids take some time to ‘adjust’ their trajectory if the player moves out of the way. This gives the asteroid’s movement an aimless drifting quality. The player’s ship is similarly controlled by applying acceleration parallel or perpendicular to the ship’s orientation depending on which key is pressed. This means the ship can drift a little in one direction even as you are controlling it to move in another. In addition, the ship decelerates through an acceleration which is a reversed and scaled-down copy of the velocity vector. I have decided that the bullets should have constant velocity so they do not rely on an acceleration vector.

All objects involved in the game (the player, asteroids, and bullets) share the ICharacter interface so they can be stored in the same ArrayList. This allows the game to easily iterate through all the objects and update them in one for loop. This also allows the game, to check if there is a collision between any two objects in the game using a single formula. When the game detects a collision, it uses the “instanceOf” keyword to identify the types of objects colliding and decide the result. When a bullet and asteroid collide the bullet is removed from the ArrayList and the asteroid splits. The score also increases by 1. When the player and an asteroid collide the player loses a life, the invulnerability timer is set to 100, and the asteroid splits. Please note, your score does not increase of an asteroid is split in an asteroid-player collision, this game does not reward kamikaze maneuvers. There are several inert collision combinations in the game such as asteroid-asteroid collisions, bullet-bullet collisions, and player-bullet collisions. These collisions are detected by the game but do not have any notable meaning in terms of gameplay and are ignored by the program.

ArrayLists are also used to store the buttons in the start and score screens, and condense the code used to update and draw these buttons to the screen.

Brief Description of Above and Beyond Features:

As requested, I developed two above and beyond features for this project.

First, I have implemented a GUI menu system with a button class that adjusts its size based on the text it contains. This GUI utilizes enums to keep track of the game state. The four states the game can be in are START, GAME, INSTRUCTIONS, and SCORE. START and SCORE have options to allow you to access the game completely, START allows you to get to INSTRUCTIONS, INSTRUCTIONS and SCORE allow you to get to START, and START and SCORE allow you to get to GAME. SCORE can only be reached by losing all your lives in GAME.

The second feature I put in is power-up items. Power-ups appear different colors depending on their type. Extra-life power-ups are green, double points power-ups are yellow, speed increase power-ups are blue, and rapid-fire power-ups are magenta. Power-ups appear after every 15 points earned in the game (originally every 5 points during testing). Both of these details are not in the instructions, I am leaving it up to the players to figure out this part on their own. Implementing power-ups used many of the same data types and program elements as other sections of the game. However, I adds new complexity in some areas. Power-up items introduce another object that must be checked for collisions with other objects. I have decided that players can pick-up power-ups by colliding with them. I also made it so getting hit by bullets would destroy power-ups. This decision is meant to encourage players to think more carefully about where they are shooting. For implementing power-ups in the game required introducing a second enum and timer. The enum keeps track of what power-up is actively on the player, and the timer keeps track of how much longer the power-up lasts resetting whenever a new power-up is gained.