Create – Applications From Ideas
Written Response Submission Template

Please see Assessment Overview and Performance Task Directions for Student for the task directions and recommended word counts.

Program Purpose and Development

2a) Narration in video

2b) I began independently developing my program with the intention of making a wave-based 2D space shooter game. After getting a player and a few enemies to display and move on the screen, the first distinct task I encountered was collision control. I figured I could compare the distance between the circle sprites to the addition of their radii. While the detection worked, this was not very efficient since every sprite had to check collisions with every other sprite and enemies would collide with other enemies. I decided to solve this by giving each sprite a team. Sprites on the same team would not check collisions, thereby minimizing distance calculations and negating the possibility of same-team collisions. Another distinct task I encountered was having enemies aim at the player before shooting. I tried using trigonometry to calculate x and y components for bullet movement. Unfortunately, most of what I tried resulting in the bullet firing away from the player. Eventually, I realized I could simply subtract the ship's position by the bullet's position, normalize the vector, then multiply it by the speed of the bullet. After solving these problems I was able to move onto creating enemy waves and attack patterns.

2c) One algorithm in my program in the wave generator. The wave generator is an algorithm that manages the waves with sub algorithms tasked with picking enemies. More difficult enemies are less likely to be picked than easy enemies, and the number of enemies corresponds to the wave number. When the wave controller object is alerted that there are no remaining enemies in the current wave, it calls the "createWave" algorithm. In order to decide which enemy to spawn, the wave creator function calls the "generateRandomNumber" method which picks a number between 0 and the length of the scaled list of enemy types. This list is generated during the creation of the wave generator by the "createEnemyList" method. This method takes in a list of the different enemy types represented by integers and a list of scale values, or percentages, that correspond to the enemy types and returns a scaled list. The random number generated is then used as an index to find the enemy type in the scaled list. The resulting enemy type is then passed to the "chooseEnemy" method that adds the new enemy at position y to the enemies array to become active in the game.

```
// Manages waves
function WaveController() {
```

```
      var self = this;
      // The verticle distance between enemies when spawned
      self.offset = -200;
      // Initial number of enemies
      self.enemyCount = 0;
      // Number of Dead Enemies
      self.deadEnemies = 0;

      // Generates random integer between min and max parameters
      var generateRandomNumber = function(min, max) {
      return Math.floor(Math.random() * (max - min + 1)) + min;
      };

      // Returns a scaled list by applying the scale values to
corresponding list values
      self.createEnemyList = function(list, scale) {
      var scaledList = [];
      //Adds a number of enemy times equal to its corresponding scale value
      for (var i = 0; i < scale.length; i++) {
            for (var j = 0; j < scale[i]; j++) {
                  scaledList.push(list[i]);
            }
      }
      return scaledList;
      };

      // List of Enemy types
      self.enemyList = [0, 1, 2, 3, 4, 5];
      // List of corresponding percentages/scale values
      self.listScale = [20, 20, 20, 20, 15, 5];
      // Generates scaled list
      self.scaledEnemyList = self.createEnemyList(self.enemyList,
self.listScale);

      // Generates new wave of enemies
      self.createWave = function() {
           _waveNumber++;
           for (var i = 0; i < _waveNumber; i++) {
                var randomNumber = generateRandomNumber(0,
self.scaledEnemyList.length);
                self.chooseEnemy(self.scaledEnemyList[randomNumber],
self.offset * i);
```

```
            }
      };

      // Called when an Enemy dies, checks if all the enemies in a wave are
dead
      // If so, it creates a new wave
      self.checkWave = function() {
            if (self.deadEnemies == self.enemyCount) {
                  self.deadEnemies = 0;
                  self.enemyCount = 0;
                  self.createWave();
            }
      };

      // Spawns enemies based on the type of enemy
      self.chooseEnemy = function(type, y) {
            switch(type) {
                  case 0:
                        _enemies.push(new StandardEnemy(y));
                        break;
                  case 1:
                        _enemies.push(new ZigZagEnemy(y));
                        break;
                  case 2:
                        _enemies.push(new StrongEnemy(y));
                        break;
                  case 3:
                        _enemies.push(new HomingEnemy(y));
                        break;
                  case 4:
                        _enemies.push(new PuffEnemy(y));
                        break;
                  case 5:
                        _enemies.push(new BigEnemy(y));
                        break;
                  default:
                        break;
            }
      };

}
```

2d) I managed the complexity of my program by using prototypical inheritance for enemies. There are six different enemy types in my game, each with different characteristics. However, they all share several variables and functions in common, even if their values or uses are different per type. Rather than programming each enemy sequentially and with common code, I abstracted properties of the enemies into a single enemy constructor function from which each enemy type inherits similar variables and functions. The Enemy prototype contains variables for the enemies team, health, max health, radius, speed, color, x position, y position, and whether the health bar is locked in place. The prototype also contains methods for killing an enemy, managing an enemies movement and position, aiming a bullet towards the player, managing the health bar, and updating factors about the enemy each frame. This way, when an enemy is created, is simply calls the enemy constructor, passing in its unique health, radius, speed, and color, to inherit all of the methods and variables. This allows each enemy type to only contain code related to its unique methods and abstracts behaviors such as dying, constraining movement, aiming at player, and updating health bar into simple method calls.

```
// constructor for enemies
function Enemy(health, radius, speed, color) {
      _waveController.enemyCount++;
      this.t = "enemy";
      this.health = health;
      this.maxHealth = health;
      this.r = radius;
      this.speed = speed;
      this.color = color;
      this.x = random(0,width);
      this.y = -this.r/1.5;
      this.staticBar = false;
}

// performs the task of killing off an enemy
Enemy.prototype.die = function() {
      this.health -= 10;
      if (this.health <= 0) {
            _waveController.deadEnemies++;
            _waveController.checkWave();
            _enemies.splice(_enemies.indexOf(this),1);
      }
};

// manages an enemies movement
Enemy.prototype.constrainMovement = function() {
```

```
      // if enemy moves off the screen, it returns to the opposite side
      if (this.x > (width + this.r) || this.x < -this.r) {
            this.x = width - this.x;
      }

      //if an enemy moves off the bottom of the scree, it returns to the
top
      if (this.y > (height + this.r)) {
            this.y = -this.r;
      }
};

// Checks if enemy is on screen, used to determine whether checking
collisions or firing bullets is necessary
Enemy.prototype.checkLocation = function() {
      return (this.y > -this.r && this.y < (height + this.r))
};

// Updates each frame
Enemy.prototype.control = function() {
      this.constrainMovement();
      this.act();
      if (this.checkLocation())
            calcCollisions(this, _bullets);
      noStroke();
      fill(this.color);
      ellipse(this.x, this.y, this.r, this.r);
      this.showHealthBar();
};

// controls the enemies health bar
Enemy.prototype.showHealthBar = function() {
      noStroke();
      fill(color(255,50,50));
      if (!this.staticBar) {
            rect(this.x - ((this.maxHealth + 4)/4), this.y - (this.r/1.2),
(this.health + 4) / 2, this.r/7.5);
      } else {
      if (this.y > -this.r/2)
            rect(width/2 - ((this.maxHealth + 4)/12), 20, (this.health + 4)
/ 6, 10);
      }
```

```
};

// aims bullet towards player
Enemy.prototype.aimBullet = function(speed) {
      var direction = createVector(_ship.x - this.x, _ship.y - this.y);
      direction.normalize();
      direction.mult(speed);
      return direction;
};
```

Example enemy type script:

```
// standard enemy constructor
function StandardEnemy(y) {
      // inherits from Enemy
      Enemy.call(this, 30, 25, 4, color(200, 100, 100));
      this.y = y - this.r/2;
      this.damage = 50;
}

StandardEnemy.prototype = Object.create(Enemy.prototype);
StandardEnemy.prototype.constructor = StandardEnemy;

// unique movement
StandardEnemy.prototype.move = function() {
      this.y += this.speed
};

// frame update
StandardEnemy.prototype.act = function() {
      this.move();
};
```