COMP 3490 Assignment 4
Fall 2024

Due date and submission instructions. Submit **one zip file** containing all of the pde files required to run your submission. Name your zip file in the style of `VaughanJenniferA4.zip`, but with your name instead of my name. All submissions should be uploaded to the Assignment 4 folder on UM Learn by 11:59pm on **Monday December 9**.

Getting started. Read these instructions all the way through. Then read the supplied template code. Considerably less coding structure has been supplied this time compared with previous assignments. However, there are several hotkeys that you need to implement.

In this assignment, you will implement a full interactive world using the Processing graphics pipeline. As long as the requirements of each question are met, you are free to design your implementation however you wish. There is less mathematics in this assignment, but a lot more coding and data management.

Processing pipeline. Unlike previous assignments, you will be using the built-in Processing graphics pipeline instead of making your own. The following functions are all allowed.

- The versions of `camera()` and `frustum()` or `perspective()` that receive parameters. Note that you **cannot** use the preset no-parameter versions that apply Processing's default settings.
- Functions for texture mapping: `textureMode()`, `texture()`, etc. You will also need `loadImage()`.
- The `PMatrix3D` and `PVector` classes, and their associated methods.
- Utilities such as `lerp()` and `constrain()`, and anything else from the Math category of the Processing reference page.
- All the functions that Processing uses to manipulate its built-in modelview matrix and the matrix stack: `pushMatrix()`, `popMatrix()`, `resetMatrix()`, and the transformations such as `translate()` etc.
- For drawing, you can use `beginShape()`, `endshape()`, and `vertex()`. Note: using `TRIANGLES` is much faster than just `beginShape()` / `endShape()`, as the latter uses Processing's polygonal engine.

If you want to use something that doesn't fit within these categories, check with me first.

Coordinate sytems. Processing, as we know, wants the y axis to point downward. You can either work with their system or fix it so that the y axis points upward. The easiest way I have found to reverse the y axis is to use the `frustum()` command to set up a perspective projection, and to reverse the order of the top and bottom boundaries.

```
frustum(left, right, top, bottom, near, far);
```

You will also be using a 3D camera, which means setting the camera matrix. You will call `camera()` with the appropriate parameters. Read the reference entry carefully and make sure you understand what parameters are required.

Make a clear choice for the coordinate system that you will use for your world. Sketch it on paper to help keep track of it. For example, you can draw your world entirely in the first quadrant (positive x and positive y), or you can draw it centered on the origin – the details are up to you.

Coding standards. Your code will be read, and you should write it with that in mind. Use reasonable variable and function names, and avoid magic numbers. Poor style may result in lost marks.

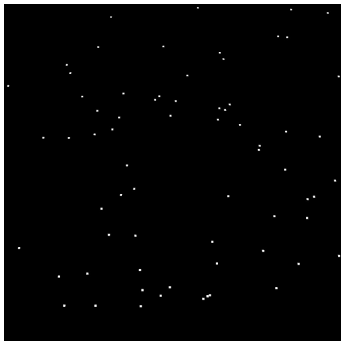
Code will be tested with Processing 4.3. Code that does not compile, that does not run, that runs but produces no image, or that crashes often enough that it is difficult to test will receive a score of 0.

Your code must implement all of the hotkeys set up in the template. You can and should add new functions and global constants and variables as needed. You can and should add new classes and new files as needed.

Questions

1. **Setup, Character, and Enemy (18 marks)**

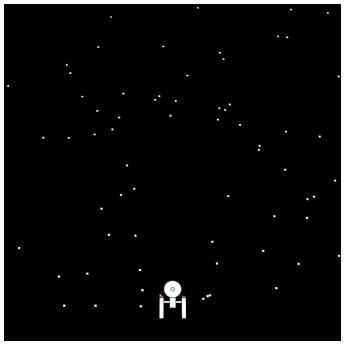
   Here we are in space.

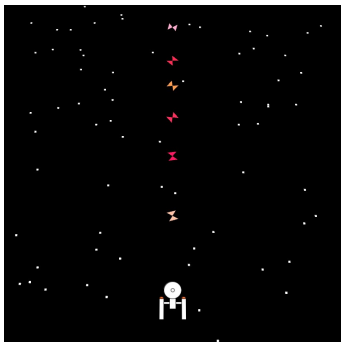
   Set up a 3D perspective projection and a 3D camera model, and draw a simple backdrop for your world. My stars are drawn at "ground-level" ($z = 0$), and my various characters will be drawn at a certain height above that.

   You do not have to set your game in space. Your backdrop can be static, or you can animate it to convey motion. Artistic license is encouraged.

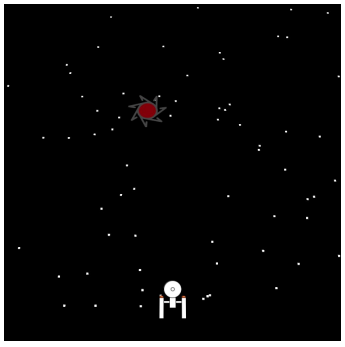
   Create a spaceship or other player character that can smoothly fly around your world. Impose left/right and top/bottom boundaries to keep it on the screen. The ship can begin as just a rectangle floating in space; you will add texture later. Make the ship slowly drift back toward a home location if no keys are pressed.

Now give the player the ability to shoot at enemies. When the player shoots, create a projectile of your choice (for simplicity, I will call it a bullet) that involves some animation as it travels. Implement this using a particle system. The bullet should be pruned/destroyed when it has traveled sufficiently off the screen.

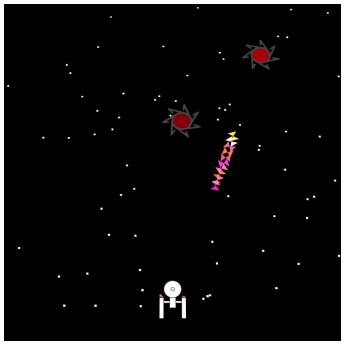


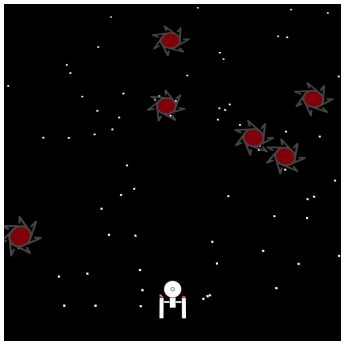
Create an enemy that moves around the world, and that stays within certain boundaries. Again, your enemy can begin as a rectangle to which you will later add texture.



You can invent game mechanics of your choice: for example, allowing the enemy to move on and off the screen for an added challenge to the player. The enemy should fire bullets at the player. Don't implement any collision detection yet.

- Use ease in/out lerping and keyframes to make the enemy movement smooth (see the Animation unit).
- Plan ahead – you'll need multiple enemies on the screen for a later task.





Implement the hotkeys given in the template so that the player can move their character. Make sure your implementation is compatible with multiple keys being held down simultaneously. Use `keyPressed()` and `keyReleased()` to set and unset global boolean flags, and write a separate function that moves the character accordingly: see the Interactivity unit.

(4 marks for perspective projection and 3D camera model, 4 marks for smooth and bounded character movement, 4 marks for automated enemy, 4 marks for smooth movement using lerp, 2 marks for projectiles)

2. **Texture Mapping and Animation (8 marks)**

Texture your player and enemy characters any way you like. Animate at least one object using frame-based animation. Your animation must be at least 3 frames long. In the demo video, the enemy runs through 10 different animation frames each time it moves, and another 10 when it shoots.

You will need to maintain a good draw order and keep track of z values in order for all the components of your world to appear properly. If overlapping enemies result in visual artifacts, consider separating them in the z direction by a small amount, and checking your draw order so that closer ones are drawn on top of further ones.

Important. If you want transparency for your texture files, you need to use PNG files with a transparent layer. You are welcome to find images online (within the bounds of copyright). A search for "free PNG sprites" will turn up a lot of options. Alternatively, you can generate your own images. Sample code that produces a PNG file with a transparent background can be found on UM Learn under Assignment 4.

(4 marks for texture mapping of player and enemies; 4 marks for frame-based animation)

3. **Collisions (10 marks)**

Implement collisions between all entities, including the player, the enemies, and the bullets. You can ignore the z values and assume that everything is on the same plane.

Use bounding-circle collision. That is, assign a radius to each object, and compare the distance between two objects with the sum of their radii. This will tell you if their bounding circles are overlapping. Don't worry about making this pixel-perfect, but try to make the result look reasonable on the screen.

Here are the requirements when a collision occurs.

- If the player is hit by an enemy bullet, or touches an enemy, the player dies. Game over.
- If an enemy touches either a player bullet or the player, the enemy dies. (Contact between the player and an enemy results in mutual destruction.) After one enemy dies, generate a new enemy. Design your spawn mechanism so that the game gets harder.
- If a player bullet hits an enemy bullet, or vice versa, they both die.
- The player doesn't get hurt by its own bullets. Enemies are not hurt by other enemies, or their bullets.

Yes, this is $O(n^2)$. It may be sluggish; that's okay.

Hint: one approach is to make everything – player, enemies, bullets – a particle. Give each particle an owner (e.g., all bullets fired by the player belong to `PLAYER`), and use the owner field in your logic.

(8 marks for collision detection and response in all cases, 2 marks for multiple enemies correctly implemented)

4. **Particle Explosions (4 marks)**

When the player or an enemy dies, create a particle-system explosion. The particles in the explosion should move in all three dimensions.

- Include some random element in the design.
- Give the particles a limited lifespan so the explosion does not go on forever.

You read all the way through the instructions before getting started, right? Keep in mind that this task is coming. If it doesn't fit with your world design to date, you may have to refactor.

(2 marks for explosion design, 2 marks for integration into world)

Implementation Strategy

This is going to be a large and complicated program. Here are some thoughts.

- Plan ahead. Think through all of the different objects and interactions that you will have to implement, and design your overall structure accordingly.
- As noted above, one approach is to make everything a particle. That is, write an abstract Particle class, and have your various components such as Player, Enemy etc. all extend it.
- Your world might have many particles active simultaneously. You can create another class that keeps track of a collection of particles using an ArrayList.
- It can be very useful to have a central World class that manages the overall state of your world.
- Create lots of classes. Create lots of files. Use good class, variable and constant names.