COMP 3490 Assignment 3
Fall 2022

Getting started. Read these instructions all the way through. Then read the supplied template code. 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.

- The background for your world will consist of randomly generated square tiles, all axis aligned, parallel to the xy plane plus some heights in the z direction.
- You will implement two different projection modes, one orthographic and one perspective, that the user can toggle on the fly.
- Your background will be texture mapped.
- You will create a character that the player can move around the world, as well as mobile enemies. The player and the enemies can shoot at each other.
- You will implement a particle system.

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 `ortho()`, `frustum()`, `perspective()` and `camera()` 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 model matrix and 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, matrix management, and a fight with Processing. We have to force Processing to give us direct access to the projection matrix. We also have to undo the Processing convention in which the y axis points downward.

The code in the file `ProjectionFix.pde` is intended to circumvent these problems. You will be implementing two different projection modes: an orthographic projection and a perspective projection.

- To construct the orthographic projection matrix, call `ortho()` in the form

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

  so that the direction of the y axis is flipped.

- To construct the perspective projection matrix, use either `perspective()` or `frustum()`, then call `fixFrustumYAxis()` to correct the y axis.

- Use `getPerspective()` to save a copy of the current projection matrix, and use `setPerspective()` to overwrite the current projection matrix with a new value.

See the comments in the files `ProjectionFix.pde` and `Transforms3D.pde`. The function `setupProjection()` in `Transforms3D.pde` already has the structure you need: you only have to fill in your choices of parameters.

You will also be using a 3D camera, which means setting the camera matrix. Processing combines the view and the model matrices into a single modelview matrix. This is the matrix that you push/pop from the matrix stack, and that is post-multiplied by your transformations. This is also the matrix that is set to the identity by calling `resetMatrix()`.

You will have to call `camera()` with the appropriate parameters. Read the reference entry carefully and make sure you understand what parameters are required. Call `resetMatrix()` before you set the camera, and not afterward.

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 4.0.1. Code that does not compile, that does not run, 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.

Deadline. Monday Dec 12, 11:59pm Winnipeg time. Really, yes really, no extensions, no exceptions. Start planning your world now!

Questions

1. **Draw and Scroll the Land (8 marks)**

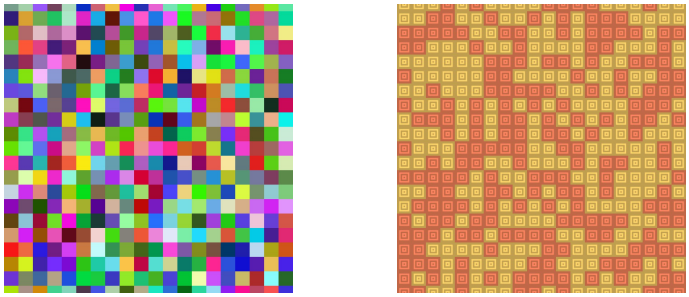
   You will create a backdrop with random tiles. This backdrop constantly scrolls in some fixed direction. New random tiles are generated so that the backdrop does not repeat itself as it scrolls.

   Start with an orthographic projection. That is, your camera can only see the x and y directions, and cannot see any variations in z.

   Give each tile its own randomly generated color. Then add an option to give each tile a texture that is randomly selected from at least two different images.

   Hints:

   - One way to produce the scrolling effect is to move the world like a conveyor belt. Another way is to move the camera.
   - Classes are your friends. Create lots of them! An overall `World` class to manage all of the objects in your world will almost certainly be useful later on. For this question, you might have one class for a single tile, and another class for a grid of tiles.
   - How do you make the tiles scroll infinitely without repeating? One approach is to create two different grids of tiles, one following the next. Once the first grid scrolls off the screen, it can be recreated with newly generated random data and repositioned to follow the second grid.



Randomly generated colors (left), randomly selected textures (right)

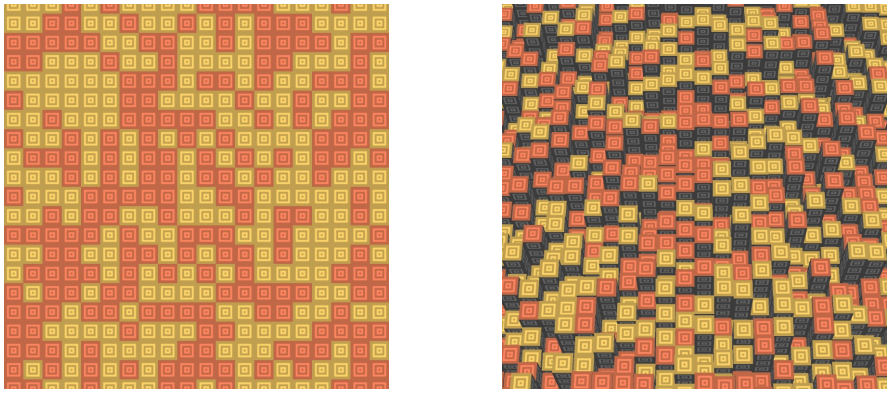
   (4 marks for filling the space evenly and randomly, 4 marks for seamless infinite scrolling)

2. **Switch Between 2D and 3D (14 marks)**

   Implement two different viewing modes: a straight top-down view that uses an orthographic projection, and an angled view from a position further back that uses a perspective projection. Let the user toggle between them using the hotkey. When the viewing mode is changed, both the projection matrix and the camera matrix have to be updated.

   Note that perspective view will significantly increase the area that is visible on the canvas. You will have to make your land larger to compensate.

   Add some features to your land that only show up in 3D.



Orthographic (left) versus perspective (right)

In orthographic mode, the tiles look two-dimensional. In perspective mode, we see that the tiles are actually columns with different heights. The sides of the columns (gray regions) become visible.

(5 marks for orthogonal projection, 5 marks for perspective projection, 4 marks for 3D features that only appear in perspective mode)

3. **Character and Enemy (14 marks)**

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. When the ship moves in any direction, add a 3D rotation so that it appears to bank: one axis for left/right, motion, another for forward/backward motion.

Now give the player the ability to shoot at enemies. When the player shoots, create a bullet (or projectile of your choice) 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 some bounds. 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 given hotkeys 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 smooth and bounded character movement, 4 marks for automated enemy, 4 marks for smooth movement using lerp, 2 marks for projectiles)

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

Texture your scene and player any way you like. You must use at least 3 different textures. 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.

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 3.

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

5. **Collisions (10 marks)**

Implement collisions between all entities, including the player, the enemies, and the bullets. You can ignore the z value 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.

Hint: rearrange the distance comparison so that you do not need to use a square root. This may improve the speed.

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)

6. **Particle Explosions (6 marks)**

When the player or an enemy dies, create a particle-system explosion, with at least 100 particles per explosion. Use 3D so that the explosion looks different in the two camera views.

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

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, 4 marks for integration into world)

7. **Bonus - Lerp the Camera (3 points)**

Instead of switching from orthographic to perspective projection abruptly, lerp them so that you transition smoothly! This will involve calculating intermediate values of the projection matrix.

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.