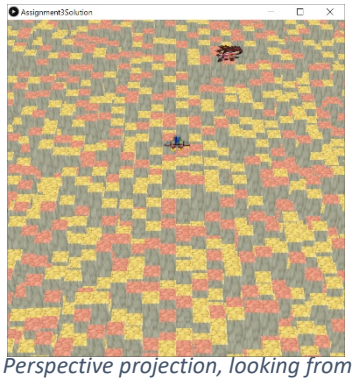
Assignment 3: Interactive World

Orthographic projection, top down

You will implement a full interactive world using the processing graphics pipeline. You will first learn the coordinate system and setup a randomly generated world. You will implement two different camera views that a user can toggle on the fly, and will texture map your world. You will create a character that can move around the world, and enemies, who you shoot at and who shoots at you. You will implement a particle system. You have freedom to implement as long as the requirements are met.

Your world will be square-tile based + height, all axis aligned. Trust me. This is to help you

This assignment has less rudimentary math than the prior assignments, but more code and engine management.



Perspective projection, looking from a little behind. Can be hard to see, check the video.

Marking

Marking for each section will be given below. Each section lists how to earn marks, but that whole section will be scaled by the weight noted in the title. All code must work in the current version of processing. Code that does not compile, does not run, or crashes regularly making it difficult to mark, will receive a grade of 0. This assignment is not marked linearly in proportion to work. The last %, for that A+, is really quite hard! The late policy is that late assignments will receive a grade of 0.

The quality and content of your code will also be marked. There are no hard and fast rules, but basic programming quality such as good variable names and self documenting code, no using magic numbers (literal numbers that are confusing – name them as a constant), reasonably elegant solutions, etc., will impact your grade.

The hotkeys and modes given in the handout must remain unchanged to facilitate marking.

Late assignments will not be accepted or graded.

Processing Pipeline

I have provided some code that you can use and shouldn't change in libJim.

Unlike previous assignments, you will be using the built in processing graphics pipeline instead of making your own. This means you will use the following functions, among others. **Check with me for approval**.

- ortho, frustum/perspective, camera/myCamera, You can't use the pre-loaded no-parameter ortho(), perspective(), or frustum() commands. You need to use the ones where you specify parameters.
- texture mapping functions (textureMode, texture, etc.). loadImage.
- PMatrix3D, PVector, and associated functions
- lerp, constrain, and other such utilities are fine
- pushMatrix, popMatrix, resetMatrix, and all the transforms available
- beginShape, endShape, vertex. NOTE: using TRIANGLES is much faster than just beginShape(), as the latter uses Processing's polygonal engine.

Matrix Management

Processing does not give you access to the viewport Matrix. I have provided functions to enable you to access the projection matrix, but that's only needed for the bonus and for the hack (see below). Instead, set the projection using ortho, frustum, perspective.

Processing combines the view and the model matrices into a single modelview matrix. This is the matrix that you push/pop and load transforms onto by default. When you call camera, it resets this matrix and puts the camera transform on it. **Careful: calling resetMatrix after camera erases the camera transform.**

Coordinate System

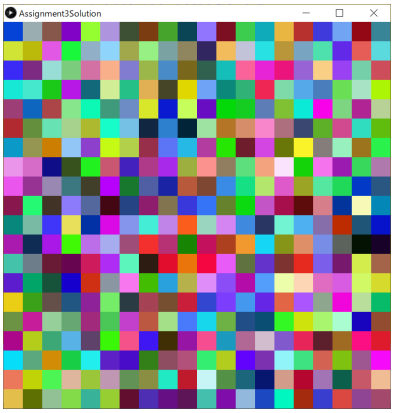
The processing pipeline has some quirks ☹. It all comes down to this upside-down-y that processing uses (stupid decision). It's not as simple as just paying attention, and it turns everything into a LHS when we're used to a RHS. It also makes the Camera "up" point "down". I have a suggested fix:

- for your ortho, select the top/bottom backward to force inversion of the y. This makes it right-way up and RHS.
- For your perspective, I have provided a hack function in libJim that flips Y /after/ the transform, effectively modifying the viewport. Call this function after you call frustum, as in the comments.
- Make sure that both of these are done, and your coordinate system will behave as expected.

Choose, and put it on paper, the orientation and coordinate system you will use for your world. In my case, I made the world X and Y positive correspond with the screen, with positive Z coming out of the screen. I didn't go into negative coordinates at all.

Task 1: Land-ho! (13.6%)

You will create a scrolling backdrop with random tiles, as shown. This constantly scrolls in some direction, always showing new random backdrop. Mine scrolls top down.



Randomly generated tile map, with random colors to facilitate debugging

Hints:

- keep everything matched to some global scale, using a variable
- you can make it scroll by moving the camera OR moving the world. I found it simpler to move the world like a conveyer belt.
- I created a class called World that held all the state info, and a class called WorldSection to represent a patch of land. My game has **two** WorldSection instances. When one scrolls down off the screen, it gets replaced with a new one on top out of sight -> it scrolls forever
- I created a Tile class to store tile info, draw, etc., and be used by the WorldSection
- Start with an orthographic projection to get your Projection and Camera matrices working

(2 marks for an even random space, 2 marks for seamless scrolling forever)

Task 2: 2D/3D switch (22.7%)

Implement a full free-look 3D camera model with two offsets: a straight top-down view (orthographic), and an angle view a little behind (perspective, insets on first page). Let the user toggle between them using the hotkey. Careful to make sure your camera vectors are all perpendicular (use crossproducts). If your world moves, then the camera stays fixed. If you decided to move the camera, implement that here.

Note that perspective view greatly increases the area seen. You'll have to make your land larger to compensate.

Add some features to your land that only show up in 3D, as in the video.

(2 marks for a good ortho view)
(2 marks for a good projection view, no holes or edges)
(2 marks for 3D features only showing in projection)

Task 3: Interactive Character and Enemy (22.7%)

Create a spaceship that can smoothly fly around your world. Create boundaries (left/right, top/bottom) to keep it on screen – these can be smaller / regional depending on the game. Since this is a flying game, 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: one axis for left/right, another for front/back.

When the player shoots, create a bullet that has some animation happening. Implement this using a particle system. The bullet should automatically prune / destroy when sufficiently off screen.

Create an enemy that moves in some predictable fashion, and also stays within some bounds (these bounds can go off-screen for added challenge). The enemy should likewise shoot at some point. Don't implement any collision detection yet.

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

The key to this is managing user input. It's not as simple as using the processing keyPressed() functions because it gets confusing when multiple keys are down, repeating keys, etc. The solution is to keep your own internal record of key state, and poll those.

- In keyPressed() check which key it was, and set a global flag that the key is down
- In keyReleased() check which key it was, and unset that global flag
- Create a function called pollKeys() that checks your flag states each frame and does appropriate character movement.
- The exception are keys that are not stateful, such as shoot: you can just handle that in keyPressed();

(2 marks for smooth and bounded character movement)
(2 marks for automated enemy, 2 marks for smooth movement using lerp)
(2 marks for appropriate bullets)

Task 4: Texture Mapping and Animation (18%)

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. Must be >2 frames. In my case, my ship looks differently when thrusting, and the enemy has 10 frames per move and 10 for shoot.

HINT: if you want transparency, you need to use PNG files with a transparent layer. Also, maintain a good draw order AND z values as this engine doesn't solve everything for you.

(3 marks for properly texture mapped without artifacts)
(2 marks for frame-based animations)

Task 5: Collisions (13.6%)

Implement collisions between all entities, including the player, enemies, and bullets. You will simply implement bounding-circle collision. First, you ignore z and use x,y only, assuming everything is on the same plane. Then, compare the distance between objects with the sum of their radii, which tells you if the circles are overlapping. This doesn't have to be pixel-perfect, as it can get confusing with coordinate system changes, scaling, etc. HINT: you can get away without using a square root for this, making it a lot faster…

So what do you do when you have a collision? You have some freedom here, but you must:

- If the player is hit by enemy bullet, or touches an enemy, game over. Player dies.
- If an enemy is touched by a player bullet, or the player, enemy dies (yes, they both die in that case). Generate a new enemy, and have some spawn mechanism so it gets harder.
- If a player bullet hits an enemy bullet, or vice versa, they both die.
- Player doesn't get hurt by its own bullets. Enemies are not hurt by other enemies, or their bullets.
- HINT: I implemented a ParticleOwner Enum / variable to keep track of who owns each bullet, etc., and used that in my logic.

Yes. This is $O(n^2)$ without any acceleration structure. Fast, it is not.

(4 marks for appropriate collision detection in all cases above)

(2 marks for multiple enemies working properly)

Task 6: Particle Explosions (9.4%)

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

- Have some random element, e.g., the direction.
- Have limited lifespan so they don't go on forever.
- Make it cool.

Key here is management of your data. If you had good code design coming up to here, this should be reasonably easy. Otherwise, this is the point where you may have to refactor.

(4 marks for appropriate particle system explosions)

Task 7: BONUS – lerp the camera (5%)

As shown in the video, lerp the camera and projection between the two views. This is particularly interesting because you're lerping between orthographic and projection!!!!!!

(3 marks, 1 for kind of working, 2 for close but not quite, 3 for perfect!!!)

Implementation Strategy

For many of you, this will be the largest program you have ever written. Good planning, good data types / classes / etc., good named constants., will save you a lot of hassle.

In my solution, I actually created the particle system first. Then, the player, enemy, bullets, and the explosion bits, all fit within the particle system.

I have: (this is not an exhaustive list)

class ParticleList -> uses arrayList to keep track of particles

- add, addLater (hhm..why would I have this??), setup, update, draw, prune. Many of these just go through the array list and call on the children.

abstract class Particle

- setup, update, draw, collide

Then, class Player, class Enemy, class Bullet, and class ExplosionBit, all extend the abstract class Particle. This way, your main engine handles the player, enemy, bullets, and explosions, identically. So in my main draw, all I do is ask the world to update, and the world to draw, in addition to handling the keys. Then the world just asks the particleList to do likewise.

The other advice I'll give is to choose a central class to manage all the global state. I used the World class, with my main having a global instance variable to World world; Thus everything is in one spot and accessible from wherever you need it.

Finally, I use lots of small classes/files /tabs. Here is my tab list: Assignment3, ExplosionBit, Bullet, Camera, Enemy, Modes, Particle, Player, World, libJim

My code is 860 lines excluding libJimda