

CS 330 Homework

WASD

1 Overview

Your responsibility in this homework is to learn about lambdas, thread pools, and parallel algorithms. You will do this in the context of writing a first-person raycaster, a technique pioneered by John Carmack, formerly of id Software, and now a lead developer at Oculus VR. One of the first blockbuster games to use raycasting was Wolfenstein 3D, a World War II-themed DOS game released in 1992. Its first episode was distributed freely as shareware; you could mail in \$50 (plus \$4 shipping and handling) to get the remaining five. Our version will have no Nazis, no theme, no goal, and no cost—unless you add them in. Figure 1 shows a screenshot of the reference implementation.

Before diving into the particulars of the homework, let's discuss the algorithms and architecture at a higher level.

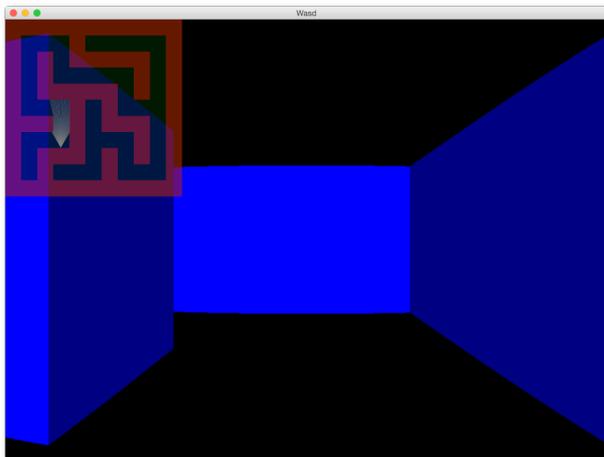


Figure 1: A screenshot of the Wasd raycaster. The minimap in the top-left corner is an instructor-provided debugging overlay showing the maze the player is navigating.

1.1 Raycasting

The raycasting algorithm can be summarized compactly in pseudocode:

```
to draw frame
  for each column of pixels in player's field of view
    shoot a ray from the player through the column
    walk along ray until we hit a wall
    draw vertical line for column inversely scaled by distance to wall
```

The fewer steps it takes to reach a wall, the taller the wall line in that column of pixels. The more, the shorter. This effect simulates perspective vision.

To compute the distance from the player to the wall, we use a *digital differential analyzer* algorithm. DDA algorithms walk a continuous line that has been overlaid across a discrete 2D grid. One may use such an algorithm to efficiently draw pixelated lines on an image or to efficiently walk a gridded world looking for collisions with rays.

1.2 Thread Pool

Compared to many other rendering algorithms, raycasting is considerably fast. It's a *per-column* algorithm instead of *per-pixel*. Since each column is computed independently, why not compute these columns in parallel? One could take this approach to parallelize the rendering of a window that's 1000 pixels wide across 10 threads, for example:

```
nthreads = 10
ncolumns = 1000
ncolumns per thread = ncolumns / nthreads

to draw frame
  c = 0
  for i to nthreads
    spawn new thread to raycast columns c and onward
    c += ncolumns per thread
```

The problem with this approach is that a lot of time is spent spawning threads. New threads are born on each frame, only to die at its end. We could have reused those threads in the next frame! Instead, let's create one fixed pool of threads that live as long as the program. To do this, we must decouple the thread from its job. We'll create a generic job queue, and each thread will simply pull a task off the queue when it is ready to do some work.

Our thread pool then must manage two things: a list of threads and a list of jobs that the threads will pick off and execute. The pool might look something like this:

```
class Pool
  to initialize n
    create empty queue of jobs
    create and start n threads

  to schedule job
    enqueue job on jobs queue
```

And each thread might look something like this:

```
class Thread
  to run
    until thread is stopped
      dequeue job from jobs queue
      execute job
```

This pool reduces the resource footprint of our raycasting algorithm considerably:

```
to initialize
  create pool of n threads
```

```
to draw frame
  for each column of pixels in player's field of view
    enqueue in pool the raycast for this column
```

2 Requirements

To receive credit for this homework, you must satisfy these requirements:

1. Mimic an Eclipse project structure. Inside `YOUR-REPOSITORY/wasd`, place two directories: `src` and `bin`. Inside `src`, place a package directory named `wasd`. All Java source files should be placed in `YOUR-REPOSITORY/wasd/src/wasd`. All compiled classes should be placed in `YOUR-REPOSITORY/wasd/bin`. By structuring your project in this way, you can open and edit your repository directly from Eclipse.
2. All code must run on a standard Linux machine. If you use another operating system, test your code on a Linux machine before submitting.
3. Complete the classes listed below in individual source files.

2.1 Tuple2

Write class `Tuple2` to represent a pair of values, possible of differing types. We often use tuples in programming languages to return multiple values from a function. `Tuple2` has the following:

- Two generic parameters, which we'll call `T` and `U`.
- A constructor which accepts a first value of type `T` and a second value of type `U`.
- Method `getFirst`, which gives back the pair's first value.
- Method `getSecond`, which gives back the pair's second value.

2.2 Vector2

Write class `Vector2` to represent a coordinate in the 2D plane. Consider this class immutable—once constructed, its state does not change. `Vector2` has the following:

- A constructor that accepts two parameters in the following order:
 - A `double` x-coordinate
 - A `double` y-coordinate
- Method `getX` that returns the x-coordinate as a `double`.
- Method `getY` that returns the y-coordinate as a `double`.
- Method `length` that returns as a `double` the vector's *magnitude*, which is the coordinates' distance from the origin, computed as the hypotenuse of the right triangle they form: $\sqrt{x^2 + y^2}$.

- Method `add` that accepts a `Vector2` as a parameter. It returns a brand new `Vector2` that is the sum of the vectors. For example, `new Vector2(1, 3).add(new Vector2(-1, 1)) → new Vector2(0, 4)`.
- Method `subtract` that accepts a `Vector2` as a parameter. It returns a brand new `Vector2` that is the difference of the vectors. For example, `new Vector2(1, 3).add(new Vector2(-1, 1)) → new Vector2(2, 2)`.
- Method `multiply` that accepts a `double` scale factor. It returns a brand new `Vector2` that is the *scalar multiplication* of the vector. For example, `new Vector2(1, 3).multiply(2) → new Vector2(2, 6)`.
- Method `divide` that accepts a `double`. It returns a brand new `Vector2` that is the *scalar division* of the vector. For example, `new Vector2(1, 3).divide(2) → new Vector2(0.5, 1.5)`.
- Method `normalize` that returns a brand new `Vector2` that points in the same direction as this one, but whose length is 1. A vector can be normalize by scaling it by the reciprocal of its length. For example, `new Vector2(10, 0).normalize() → new Vector2(1, 0)`.
- Method `rotate` that accepts a `double` number of degrees as a parameter. It returns a brand new `Vector2` that is like this vector, but rotated by the specified number of degrees. Use the following equations to calculate the new vector's x- and y-coordinates:

$$x' = x * \cos \theta - y * \sin \theta$$

$$y' = x * \sin \theta + y * \cos \theta$$

For example, `new Vector2(10, 0).rotate(90) → new Vector2(0, 10)`.

- Method `toString` that returns a `String` representation of this `Vector2` in the form "X,Y", where X and Y are the x- and y-coordinates listed to the default precision of `String.format`.

2.3 World

Write class `World` to model a navigable "level". `World` has the following:

- A constructor that accepts a `File` as a parameter. Inside the file is a plain text representation of the world in this format:

```
width height
playerX playerZ
lookAtX lookAtZ
tile00 tile01 tile02 ...
tile10 tile11 tile12 ...
...
```

The width and height are integers specifying the number of tiles in the world. Next is the player's starting position and view direction in the world, all of which are `doubles`. The remainder of the file specifies the tiles of the world. For example, a 5-by-3 world with walls around the edge and with the player in the center looking right is represented as:

```
5 3
2 1
1 0
1 1 1 1 1
1 0 0 0 1
1 1 1 1 1
```

A tile value of 1 represents a wall. A tile value of 0 represents a passage. Note that the world is flat, and that we are viewing it from above in the file. From this vantage point, we label the left-to-right direction as the x-axis, and the top-to-bottom direction as the z-axis. At runtime, the player's vertical or y-axis position will not change.

- Method `getStart` that returns the player's starting position (as specified in the file) as a `Vector2`.
- Method `getLookAt` that returns the player's initial viewing direction (as specified in the file) as a `Vector2`.
- Method `getWidth` that returns the world's width as an `int`.
- Method `getHeight` that returns the world's height as an `int`.
- Method `get` that accepts as parameters an x-coordinate and a z-coordinate of a tile, both as `ints`. It returns 0 if the specified tile is a passage and 1 if the tile is a wall. Tiles outside the bounds of the world are considered walls.
- Method `isPassage` that accepts as parameters an x-coordinate and a z-coordinate of a tile, both as `ints`. It returns `true` if the specified tile is a passage and `false` otherwise.
- Method `isPassage` that accepts as a parameter a `Vector2` location. It returns `true` if the tile specified by the vector's truncated coordinates is a passage and `false` otherwise.
- Method `getDistanceToWall` that accepts as parameters a player's location and a normalized view direction, both as `Vector2s`. It marches along the player's lookat vector until it hits a wall, reporting the distance. This value will be used to scale the height of the wall; longer distances make for shorter walls to give a feeling of perspective. Implement the DDA algorithm from Lode Vandevenne's raycaster tutorial¹ using our types. Begin at the sentence "In the next code piece, more variables are declared and calculated, these have relevance to the DDA algorithm...", and end at the calculation of the perpendicular wall distance.

Return a `Tuple2` whose first component is the perpendicular wall distance (as a `Double`) and whose second component is a `Boolean` indicating the direction of the wall. The x direction corresponds to `false` and the z direction corresponds to `true`.

2.4 Player

Write class `Player` to model the player that moves through a world. `Player` has the following:

¹<http://lodev.org/cgtutor/raycasting.html>

- A constructor that accepts two parameters in this order:
 - A `World` that the player is to traverse
 - A field of view as a `double`

It initializes the player to be situated at the world's starting position and looking along the world's starting lookat direction.

- Method `getFieldOfView` that returns the player's field of view as a `double`.
- Method `getPosition` that returns the player's current position as a `Vector2`.
- Method `getLookAt` that returns the player's current viewing direction as a `Vector2`.
- Method `getRight` that returns the player's current right vector as a `Vector2`. As the lookat vector points ahead of the player, the right vectors points to player's right. It can be computed by rotating the lookat vector -90 degrees.
- Method `rotate` that accepts a number of degrees as a `double`. It turns the player by the specified number of degrees—by steering its lookat vector. Negative values turn the player to the right, positive to the left.
- Method `advance` that accepts a distance as a `double`. It moves the player by the specified distance along the lookat vector. Negative distances move the player backward, while positive distances move the player forward. The lookat vector is not modified.
- Method `strafe` that accepts a distance as a `double`. It moves the player by the specified distance along the right vector. Negative distances move the player left, while positive distances move the player right. The lookat vector is not modified.

2.5 BooleanTask

Write interface `BooleanTask` to describe the operations of job that can be scheduled on a separate thread. It is nearly identical to `Runnable`, except its `run` method returns a `boolean`. It has the following public interface:

- Method `run`, which returns a `boolean`. Implementers will use this method to perform a task and then return some sort of status value. A more specific use case will be provided in the `ThreadPool` specification.

2.6 ThreadPool

Write class `ThreadPool` to represent a fixed-size set of generic worker threads. It has the following public interface:

- A constructor that accepts a number of threads as an `int`. It also creates a tasks queue that the threads will pull from to claim their tasks. Since many threads may try to access this queue concurrently, access must be synchronized. Use Java's `LinkedBlockingQueue` class, which is thread-safe.

It also spawns off the specified number of threads. Each thread follows the general algorithm described earlier, where each task is a `BooleanTask`:

```
until told to stop
  dequeue task
  execute task
end
```

How do we tell the thread to stop? We'll use a *poison pill*. Recall that unlike `Runnable`, `BooleanTasks` return a `boolean`. If the task yields `true`, the thread should stop. If the job yields `false`, the thread will continue on to the next job. Let's refine our pseudocode description:

```
while not poisoned
  dequeue task
  is poisoned = execute task
end
```

Ignore any `InterruptedExceptions`, which may happen while waiting for a task from an empty queue. When we want the threads to stop, we'll feed a dummy task into the queue that yields `true`.

- Method `schedule`, which accepts a `Runnable` parameter for a task to enqueue. We can't directly add this to the tasks queue, because it doesn't return a `boolean`. We could have asked the caller to just provide a `BooleanTask` instead of a `Runnable`, but the `boolean` only exists as a poison pill, which is an implementation detail that we should not expect the caller to manage.

Instead, wrap this task up in a lambda that runs the `Runnable` and sneaks in a `false` return value to keep the thread alive. This lambda will match up with *Single Abstract Method Interface (SAMI)* of `BooleanTask`. Put the lambda on the queue. Do not create an instance of `BooleanTask` in any way besides a lambda.

- Method `stop`. It stops all threads gracefully by scheduling tasks that do nothing but yield a `true` poison pill. Putting a task on the queue may cause an `InterruptedException`, which will result in an unscheduled task. To make sure you do indeed get a poison pill task scheduled, follow this pattern:

```
is scheduled = false
while is not scheduled
  try
    put poison pill on tasks queue
    is scheduled = true
  catch
    don't do anything, but let the loop try again
```

How many of these tasks are needed to ensure all threads exit? As each thread consumes a poison pill, it exits and won't process any more tasks. So, we need as many poison pills as we have threads.

After inserting the pills, we must `join` each thread to clean up its resources. While trying to join a thread, you may receive an `InterruptedException`. As you did when scheduling the poison pill, spin around until the join succeeds.

2.7 Raycaster

Write class `Raycaster` to manage first-person drawing of a world from a player's perspective. It has the following:

- A constructor that accepts three parameters in this order:
 - A `World`
 - A `Player`
 - A number of threads, as an `int`

It creates a thread pool with the specified number of threads. It also creates a `ReentrantLock`, which will be used later.

- Method `drawColumn` that accepts four parameters in this order:
 - A width of type `int`
 - A height of type `int`
 - A drawing surface of type `Graphics`
 - A column index of type `int`

This method casts a ray from the player through the given column. It walks along that ray until it hits a wall. A line is drawn in the column, but its height is scaled down according to the ray's distance. A longer ray means a wall that's farther away and is therefore shorter.

Follow this algorithm:

1. Compute the horizontal proportion of this column within the image by dividing the column index by the maximum possible column. Proportions are in $[0, 1]$. Note that the maximum column is one less than the image's width.
2. Compute the field-of-view proportion by range-mapping the proportion to $[-1, 1]$. Multiply the proportion by 2 and subtract 1. A negative number means the ray is pointing left of the player's lookat vector. A positive number means the ray is pointing right of the player's lookat vector. 0 means the ray is pointing along the lookat vector.
3. Compute the angle of the ray away from the lookat vector by applying it to half the player's field of view. For example, if the field of view is 60 degrees, a proportion of -1 yields -30, -0.5 yields -15, 0 yields 0, 0.5 yields 15, and 1 yields 30.
4. Fix the sign of the angle. Mathematics says left (counterclockwise) turns have positive angles, right (counterclockwise) turns have negative angles. This is opposite of the angle computed in the previous step So, negate it.
5. Compute a `Vector2` representing the ray by rotating the player's lookat vector by the angle.
6. Get the distance to the nearest wall.
7. Compute the height of the column in pixels. The farther the distance, the shorter the wall. Because the distance and height are inversely proportional, a straightforward solution is to simply divide the drawing surface's height by the distance.

8. Draw a non-black line at the given column using the height calculated in the previous step. To add depth cues, draw walls along the x-axis differently than z-axis. (The example image above uses two shades of blue.) Center the line vertically. Its top y-coordinate is the surface's middle y-coordinate minus half the column's height. Its bottom y-coordinate is the surface's middle y-coordinate plus half the column's height.

- Method `render`, which accepts two parameters in this order:
 - A width of type `int`
 - A height of type `int`

It returns a `BufferedImage` rendering of the world from the player's perspective.

Complete this method in a serial first—it'll be much simpler. Follow this algorithm:

1. Create a new `BufferedImage` with an RGB pixel format.
2. Get a `Graphics` object from `BufferedImage` to facilitate drawing.
3. Set the color to black and `fillRect` a black background.
4. Iterate through the columns and draw each one using `drawColumn`.

At this point, you can test your work using `Main`, which is described in the next section. Once the serial version is working, return here to add parallel rendering.

The task we want to parallelize is the drawing of all the columns. Each call to `drawColumn` is independent of the others. We call an operation where each task is independent of the others *embarrassingly parallel*. Note the following before we discuss the parallel algorithm:

1. Though our drawing will run asynchronously, we don't want this method to return until all columns have been drawn. So, we will need to keep a counter that gets incremented after each `drawColumn` finishes.
2. This counter must be shared across all tasks. Java requires that shared local variables be `final`. But when we qualify an `int` as `final`, it becomes a constant. Constants don't make very good counters. We can't use `Integer` either, because this class is immutable. Therefore, we will use `AtomicInteger`. This class is threadsafe on its own, but we will be using our own lock to provide thread safety. (A better name for our purposes would be `MutableInteger`.)
3. The thread that called this method must wait until the counter reaches the surface's width before return the image. Instead of polling (looping around continuously) until the counter reaches the width, we will suspend the thread until the last draw task signals it.

Now we can adjust the code to draw in parallel. Here's the algorithm we want, in entirety:

```
create image and drawing surface
fill background

final Condition allColumnsDone = use lock to create a new condition
final AtomicInteger counter = create new AtomicInteger of 0
lock the lock
```

```
for each column
  schedule on the pool this task
  draw column
  lock the lock
  if increment and get the counter == width
    signal the condition
  unlock the lock

await the condition uninterruptibly
unlock the lock

return image
```

Does the parallel algorithm run any faster? It may not. Why might it not be faster than the serial version?

- Method `stop`, which stops the thread pool.

2.8 Main

The class `Main` is provided to you in the template repository. Copy it from `specs/grade_wasd/Main.java` to your `wasd` directory. It creates a JavaFX window, adds a mouse listener that allows the mouse to turn the player, and adds a key listener that responds to the following keys:

- W to advance forward
- A to strafe left
- S to advance backward
- D to strafe right
- Q to turn left
- E to turn right
- O to draw a topdown overlay of the player and world
- F to toggle fullscreen
- Escape to quit.

Invoke `Main` with a file containing a world and a number of threads:

```
java -cp bin wasd.Main world.txt 2
```

Tweak this class as you see fit, but please note any changes you make in file `README`.

3 Later Week

To qualify for later-week submission, you must successfully complete the classes `Tuple2`, `Vector2`, `World`, `Player`, and `BooleanTask`.

4 SpecChecker Notes

Classes `ThreadPool` and `Raycaster` are not automatically graded by unit tests, unlike the other classes. However, the `SpecChecker` will still compile and run them for manual testing.

5 Extra Credit

For an extra credit participation point, design a world and share it on Piazza under folder `wasd_share`.

The `SpecChecker` must be run from your `YOUR-REPOSITORY/wasd` directory. If you do most of your editing from `wasd/src/wasd`, you can test your code without changing directories like so:

```
(cd ../../ && ../specs/grade)
```

The parentheses force the command to execute in a subshell. The `cd` command alters the state of only the subshell, leaving your “supershell” in its current directory.

6 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others’ code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don’t want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.