com/mrbbot/civilisation/Civilisation.java

```java
package com.mrbbot.civilisation;

import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.MapSize;
import com.mrbbot.civilisation.net.CivilisationServer;
import com.mrbbot.civilisation.net.packet.*;
import com.mrbbot.civilisation.ui.connect.ClientCreator;
import com.mrbbot.civilisation.ui.connect.ScreenConnect;
import com.mrbbot.civilisation.ui.connect.ServerCreator;
import com.mrbbot.civilisation.ui.game.ScreenGame;
import com.mrbbot.generic.net.Client;
import com.mrbbot.generic.net.Server;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Rectangle2D;
import javafx.stage.Screen;
import javafx.stage.Stage;

import java.io.IOException;

/**
 * Class containing the main entry point for the program.
 */
public class Civilisation extends Application
  implements ClientCreator, ServerCreator {
  /**
   * The game client. Exposed so that any component of the game can send
   * packets to the server.
   */
  public static Client<Packet> CLIENT;
  /**
   * The internal game server. There should only be one instance of this per
   * instance of the program.
   */
  private static CivilisationServer SERVER;

  /**
   * Primary stage of the application. This is where the scenes for the
   * different screens go.
   */
  private Stage primaryStage;
  /**
   * Width of the application window.
   */
```

```java
private int width;
/**
 * Height of the application window.
 */
private int height;
/**
 * The game screen for this client. Contains the 3D map render and the UI
 * information overlays.
 */
private ScreenGame screenGame;

@Override
public void start(Stage primaryStage) {
  // Store the primary stage so the screen can be changed later.
  this.primaryStage = primaryStage;

  // Make the game occupy all of the screen
  Rectangle2D screenBounds = Screen.getPrimary().getBounds();
  width = (int) screenBounds.getWidth();
  height = (int) screenBounds.getHeight();
  width = 1000; //1000 //1600
  height = 600; //600 //900

  // Create the initial connection screen, registering this as the client
  // and server creator
  ScreenConnect screenConnect = new ScreenConnect(
    this, this
  );
  // Show the connection screen by default
  primaryStage.setScene(
    screenConnect.makeScene(primaryStage, width, height)
  );

  // Set window details
  primaryStage.setTitle("Civilisation");
  primaryStage.setResizable(false);
  //primaryStage.setFullScreen(true);

  // Terminate the client and server when the user requests the game exit by
  // clicking the window's close button
  primaryStage.setOnCloseRequest((event) -> {
    try {
      if (CLIENT != null) CLIENT.close();
    } catch (IOException ignored) {
    }
```

```
    try {
      if (SERVER != null) SERVER.close();
    } catch (IOException ignored) {
    }
    System.exit(0);
  });

  // Show the game window
  primaryStage.show();
}

/**
 * Function to create a new game client. Creates and shows the game screen
 * when the first packet is received.
 *
 * @param host server host IP/URL
 * @param port server port number
 * @param id   desired id of the player
 * @throws IOException if there was a networking error
 */
public void createClient(
  String host,
  int port,
  String id
) throws IOException {
  // Store the client so that all components of the game can send packets to
  // the server
  CLIENT = new Client<>(
    host,
    port,
    id,
    // Run the packet handler on the UI thread so that UI components can be
    // updated without throwing errors
    ((connection, data) -> Platform.runLater(() -> {
      if (data instanceof PacketGame) {
        // If the packet contains game state information (1st packet), create
        // the game screen with the existing state and show it to the user
        Game game = new Game(((PacketGame) data).map);
        screenGame = new ScreenGame(game, id);
        primaryStage.setScene(
          screenGame.makeScene(primaryStage, width, height)
        );
      } else if (data instanceof PacketChat) {
        // If this was a chat packet, send it to the chat panel
        screenGame.handlePacketChat((PacketChat) data);
```

```
      } else {
        // Otherwise, if it was anything else...

        // If this was a ready packet, make the "Next Turn" button clickable
        // again
        if (data instanceof PacketReady) {
          screenGame.handlePacketReady((PacketReady) data);
        }

        // Get the game to handle it (likely a game state sync [unit moving,
        // city creation, etc])
        screenGame.renderCivilisation.root.handlePacket(data);
      }
    }))
  );
  // Send a request for the current game state
  CLIENT.broadcast(new PacketInit());
}

/**
 * Function to create a new internal game server.
 *
 * @param gameFilePath file path of the game save file (may or may not exist)
 * @param gameName     name of the game (if this is null, we're loading an
 *                     existing game from a file)
 * @param mapSize      desired map size of the new game (ignored if loading
 *                     from a file)
 * @param port         port number to run the server on
 * @throws IOException if there was a networking error
 */
public void createServer(
  String gameFilePath,
  String gameName,
  MapSize mapSize,
  int port
) throws IOException {
  if (gameName == null) {
    // If the game name is null, we're loading an existing game from a file
    // so don't pass the additional parameters
    SERVER = new CivilisationServer(gameFilePath, port);
  } else {
    // Otherwise, create an entirely new game
    SERVER = new CivilisationServer(gameFilePath, gameName, mapSize, port);
  }
}
```

```
  /**
   * Main entry point for the client program. Launches the JavaFX application.
   * @param args command line arguments
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

com/mrbbot/civilisation/geometry/Hexagon.java

```
package com.mrbbot.civilisation.geometry;

import javafx.geometry.Point2D;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Shape3D;
import javafx.scene.transform.Rotate;

/**
 * Class that represents a Hexagon in 2D space
 */
public class Hexagon {
  /**
   * The square root of 3, used in calculations of distances to edges
   */
  public static final double SQRT_3 = Math.sqrt(3);

  /**
   * The center of the hexagon
   */
  private Point2D c;
  /**
   * The radius from the center of the hexagon
   */
  private double r;
  /**
   * The points that join the edges of the hexagon. All of these points are a
   * {@link #r radius} from the {@link #c center}.
   */
  private Point2D[] vertices;

  /**
   * Creates a new hexagon using the specific parameters
   *
```

```
 * @param center center coordinate of the new hexagon
 * @param radius radius from the center of the new hexagon
 */
Hexagon(Point2D center, double radius) {
  this.c = center;
  this.r = radius /*- 0.1*/; // - 0.1 puts a gap in between hexes
  calculateVertices();
}

/**
 * Calculates the vertices of the hexagon. This is only called when any of
 * the data required to calculate them changes.
 */
private void calculateVertices() {
  double cx = c.getX();       // alias for c.getX()
  double cy = c.getY();       // alias for c.getY()
  double hr = r / 2;          // half radius
  double hw = SQRT_3 * hr;    // half width

  // Calculate the vertices and store them in a new array (the old one will
  // be garbage collected)
  this.vertices = new Point2D[]{
    new Point2D(cx, cy - r),
    new Point2D(cx - hw, cy - hr),
    new Point2D(cx - hw, cy + hr),
    new Point2D(cx, cy + r),
    new Point2D(cx + hw, cy + hr),
    new Point2D(cx + hw, cy - hr),
  };
}

/**
 * Gets the vertices of this hexagon
 *
 * @return vertices of this hexagon
 */
public Point2D[] getVertices() {
  return vertices;
}

/**
 * Gets a hexagonal prism created by extruding the cross section
 *
 * @param height the height of the new prism
 * @return a 3D hexagonal prism
```

```
 */
public Shape3D getPrism(double height) {
  Cylinder cylinder = new Cylinder(r, height, 6);
  cylinder.getTransforms().addAll(
    new Rotate(90, Rotate.X_AXIS));
  return cylinder;
}

/**
 * Gets the center of this hexagon
 *
 * @return center of this hexagon
 */
public Point2D getCenter() {
  return c;
}

/**
 * Sets the center of this hexagon and then recalculates the vertices
 *
 * @param center new center of the hexagon
 */
public void setCenter(Point2D center) {
  this.c = center;
  calculateVertices();
}

/**
 * Gets the radius of this hexagon
 *
 * @return radius of this hexagon
 */
public double getRadius() {
  return r;
}

/**
 * Sets the radius of this hexagon and then recalculates the vertices
 *
 * @param radius new radius of the hexagon
 */
public void setRadius(double radius) {
  this.r = radius;
  calculateVertices();
}
```

```
  @Override
  public String toString() {
    return "Hexagon[cx = " + c.getX()
      + ", cy = " + c.getY() + ", r = " + r + "]";
  }
}
```

com/mrbbot/civilisation/geometry/HexagonConsumer.java

```
package com.mrbbot.civilisation.geometry;

/**
 * Represents an operation that accepts part of a hexagon grid and then performs an
action with it.
 * @param <T> type of the hexagon grid
 */
@FunctionalInterface
public interface HexagonConsumer<T> {
   void accept(T t, Hexagon hex, int x, int y);
}
```

com/mrbbot/civilisation/geometry/Path.java

```
package com.mrbbot.civilisation.geometry;

import java.util.List;

/**
 * Class representing a path between two points
 * @param <E> the type of the data within the path
 */
public class Path<E extends Traversable> {
  /**
   * Ordered list of the tiles in the path
   */
  public final List<E> path;
  /**
   * The cost of travelling to the end through this path
   */
  public final int totalCost;

  Path(List<E> path, int totalCost) {
    this.path = path;
    this.totalCost = totalCost;
  }
}
```

com/mrbbot/civilisation/geometry/HexagonGrid.java

```java
package com.mrbbot.civilisation.geometry;

import javafx.geometry.Point2D;

import java.io.Serializable;
import java.util.*;

/**
 * Represents a 2D grid a hexagons and handles the maths for positioning hexagons relative to each other
 *
 * @param <E> the type of the data associated with each hexagon (should implement {@link Traversable} for pathfinding)
 */
public class HexagonGrid<E extends Traversable> implements Serializable {
  /**
   * Grid width of the hexagon grid (how many tiles the grid spans)
   */
  private final int width;
  /**
   * Grid height of the hexagon grid (how many tiles the grid spans)
   */
  private final int height;
  /**
   * Radius of each hexagon within the hexagon grid
   */
  private final double radius;

  /**
   * 2D array for storing the data associated with each tile. Objects within this array should be of type E. Every
   * other tile has +-1 element than the previous row.
   */
  private final Object[][] grid;
  /**
   * 2D array for storing the hexagons which contain the data about tile positioning for this grid. Every other tile
   * has +-1 element than the previous row.
   */
  private final Hexagon[][] hexagonGrid;

  /**
   * The distance from the midpoint of one edge to another midpoint of a hexagon
   */
```

```
  private final double cw;    // cell width
  /**
   * The distance from the center of a hexagon to the midpoint of an edge of the
hexagon
   */
  private final double hcw;   // half cell width
  /**
   * The vertical distance between the center of hexagons on adjacent rows
   */
  private final double ch;    // cell height
  /**
   * The x-coordinate of where the grid starts relative to the origin
   */
  private final double sx;    // start x
  /**
   * The y-coordinate of where the grid starts relative to the origin
   */
  private final double sy;    // start y

  /**
   * Creates a new Hexagon Grid using the default radius of 1
   *
   * @param width  grid width of the grid
   * @param height grid height of the grid
   */
  @SuppressWarnings("unused")
  public HexagonGrid(int width, int height) {
    // Call the other constructor with the default
    this(width, height, 1);
  }

  /**
   * Creates a new Hexagon Grid
   *
   * @param width  grid width of the grid
   * @param height grid height of the grid
   * @param radius radius of each hexagon within the hexagon grid
   */
  public HexagonGrid(int width, int height, double radius) {
    // Initialise class fields
    this.width = width;
    this.height = height;
    this.radius = radius;

    // Construct the grid arrays taking into account the extra element on alternating
```

```
rows
    grid = new Object[height][width + 1];
    hexagonGrid = new Hexagon[height][width + 1];

    // Calculate constants for the grid that are used when laying out the hexagons
    cw = Hexagon.SQRT_3 * radius;    // cell width
    hcw = cw / 2;                    // half cell width
    ch = 3.0 / 2.0 * radius;         // cell height
    double gw = (width - 1.0) * cw;  // grid width
    double gh = (height - 1.0) * ch; // grid height
    sx = -gw / 2;                    // start x
    sy = gh / 2;                     // start y

    // Calculate the positions of the hexagons on the grid
    calculateHexagonGrid();
  }

  /**
   * Iterates through every position for a hexagon and creates a new {@link Hexagon}
for that position.
   */
  private void calculateHexagonGrid() {
    forEach((e, hex, x, y) -> hexagonGrid[y][x] = new Hexagon(
      // Center of the hexagon
      new Point2D(
        // From the start coordinates, add the extra for this position
        sx + (cw * x) - ((y % 2) * hcw),
        sy - (ch * y)
      ),
      // Use the specified radius for hexagons
      radius
    ));
  }

  /**
   * Determines whether a cell actually exists in the hexagon grid, taking into
account alternating numbers of
   * elements on each row
   *
   * @param x x-coordinate of cell to check
   * @param y y-coordinate of cell to check
   * @return whether the cell exists
   */
  private boolean cellExists(int x, int y) {
    return 0 <= x && x < grid[0].length && 0 <= y && y < grid.length && !(y % 2 == 0
```

```
&& x > grid[0].length - 2);
  }

  /**
   * Checks whether the specified neighbouring cell exists, sometimes taking into
account the traversability of the
   * cell.
   *
   * @param x             x-coordinate of neighbouring cell to check
   * @param y             y-coordinate of neighbouring cell to check
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return whether the neighbouring cell exists
   */
  private boolean checkNeighbour(int x, int y, boolean checkTraverse) {
    return cellExists(x, y) && (!checkTraverse || get(x, y).canTraverse());
  }

  /**
   * Gets the data associated with the cell at the coordinate
   *
   * @param x x-coordinate of cell
   * @param y y-coordinate of cell
   * @return data associated with the cell
   */
  @SuppressWarnings("unchecked")
  public E get(int x, int y) {
    // Ensure the cell actually exists in the grid
    assert cellExists(x, y);
    // Return the data, casting it to the data type
    return (E) grid[y][x];
  }

  /**
   * Gets the data associated with a cell called in the context of an adjacency check
   *
   * @param x             x-coordinate of cell
   * @param y             y-coordinate of cell
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @param dx            offset added to the x-coordinate of the cell
   * @return the data associated with the cell or null if the cell doesn't exist
   */
  private E getAdjacent(int x, int y, boolean checkTraverse, int dx) {
    // Check if the cell exists, if it does return it, otherwise return null
```

```
    return checkNeighbour(x + dx, y, checkTraverse) ? get(x + dx, y) : null;
  }

  /**
   * Gets the cell to the top-left of the specified coordinate
   *
   * @param x             x-coordinate to check to the top-left of
   * @param y             y-coordinate to check to the top-left of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the top-left or null if the cell doesn't exist
   */
  public E getTopLeft(int x, int y, boolean checkTraverse) {
    return getAdjacent(x, y - 1, checkTraverse, -(y % 2));
  }

  /**
   * Gets the cell to the top-right of the specified coordinate
   *
   * @param x             x-coordinate to check to the top-right of
   * @param y             y-coordinate to check to the top-right of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the top-right or null if the cell doesn't exist
   */
  public E getTopRight(int x, int y, boolean checkTraverse) {
    return getAdjacent(x + 1, y - 1, checkTraverse, -(y % 2));
  }

  /**
   * Gets the cell to the left of the specified coordinate
   *
   * @param x             x-coordinate to check to the left of
   * @param y             y-coordinate to check to the left of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the left or null if the cell doesn't exist
   */
  public E getLeft(int x, int y, boolean checkTraverse) {
    return getAdjacent(x - 1, y, checkTraverse, 0);
  }

  /**
   * Gets the cell to the right of the specified coordinate
   *
```

```
   * @param x             x-coordinate to check to the right of
   * @param y             y-coordinate to check to the right of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the right or null if the cell doesn't exist
   */
  public E getRight(int x, int y, boolean checkTraverse) {
    return getAdjacent(x + 1, y, checkTraverse, 0);
  }

  /**
   * Gets the cell to the bottom-left of the specified coordinate
   *
   * @param x             x-coordinate to check to the bottom-left of
   * @param y             y-coordinate to check to the bottom-left of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the bottom-left or null if the cell doesn't
exist
   */
  public E getBottomLeft(int x, int y, boolean checkTraverse) {
    return getAdjacent(x, y + 1, checkTraverse, -(y % 2));
  }

  /**
   * Gets the cell to the bottom-right of the specified coordinate
   *
   * @param x             x-coordinate to check to the bottom-right of
   * @param y             y-coordinate to check to the bottom-right of
   * @param checkTraverse whether to check if the cell is traversable or not (used
for pathfinding)
   * @return the data associated with the bottom-right or null if the cell doesn't
exist
   */
  public E getBottomRight(int x, int y, boolean checkTraverse) {
    return getAdjacent(x + 1, y + 1, checkTraverse, -(y % 2));
  }

  /**
   * Gets a list of the neighbouring cells to the specified coordinate
   *
   * @param x             x-coordinate to get neighbours of
   * @param y             y-coordinate to get neighbours of
   * @param checkTraverse whether to check if the cells are traversable or not (used
for pathfinding)
```

```
   * @return ArrayList of the neighbours' data
   */
  public ArrayList<E> getNeighbours(int x, int y, boolean checkTraverse) {
    // Create an empty list to add to
    ArrayList<E> list = new ArrayList<>();

    // Get the adjacent cells
    E topLeft = getTopLeft(x, y, checkTraverse);
    E topRight = getTopRight(x, y, checkTraverse);
    E left = getLeft(x, y, checkTraverse);
    E right = getRight(x, y, checkTraverse);
    E bottomLeft = getBottomLeft(x, y, checkTraverse);
    E bottomRight = getBottomRight(x, y, checkTraverse);

    // Add the cells to the list if they aren't null
    if (topLeft != null) list.add(topLeft);
    if (topRight != null) list.add(topRight);
    if (left != null) list.add(left);
    if (right != null) list.add(right);
    if (bottomLeft != null) list.add(bottomLeft);
    if (bottomRight != null) list.add(bottomRight);

    // Return the list
    return list;
  }

  /**
   * Finds the shortest path between (x1, y1) and (x2, y2) that does not exceed
maxCost using Dijkstra's algorithm. If
   * maxCost is exceeded, the function returns the path up until the cost is
exceeded.
   *
   * @param x1      start x-coordinate
   * @param y1      start y-coordinate
   * @param x2      end x-coordinate
   * @param y2      end y-coordinate
   * @param maxCost max cost of the path
   * @return a path object containing information of the tiles in the path and the
total cost of the path
   */
  public Path<E> findPath(int x1, int y1, int x2, int y2, int maxCost) {
    // Create a map for storing the cost of certain tiles for sorting the queue
    Map<E, Integer> costs = new HashMap<>();
    // Create the queue for the frontier, sorting elements by their cost so the
cheapest elements are at the front
```

```
    PriorityQueue<E> frontier = new PriorityQueue<>
(Comparator.comparingInt(costs::get));

    // Create a map for storing the path back to the beginning
    Map<E, E> cameFrom = new HashMap<>();
    // Create a map for storing the costs of the path to a cell so far
    Map<E, Integer> costSoFar = new HashMap<>();

    // Get the start/end of the path
    E start = get(x1, y1);
    E goal = get(x2, y2);

    // Set the cost of the start to 0 and add it to the queue
    costs.put(start, 0);
    frontier.add(start);
    cameFrom.put(start, null);
    costSoFar.put(start, 0);

    // While there are still tiles to explore...
    while (!frontier.isEmpty()) {
      // Get the cheapest tile
      E current = frontier.remove();
      // Check if this is the goal or doesn't exist
      if (current == null || current == goal) break;
      // For every neighbour of the current tile... (making sure that the neighbour
is traversable)
      for (E next : getNeighbours(current.getX(), current.getY(), true)) {
        // Calculate the cost of getting to this tile
        int newCost = costSoFar.get(current) + next.getCost();
        // If this is a new tile or the cost of using this route is cheaper
        if (!cameFrom.containsKey(next) || newCost < costSoFar.get(next)) {
          // Store the more efficient path
          costSoFar.put(next, newCost);
          costs.put(next, newCost);
          frontier.add(next);
          cameFrom.put(next, current);
        }
      }
    }

    // Build the path list of tiles travelled
    List<E> path = new ArrayList<>();
    E current = goal;
    while (current != null) {
      path.add(current);
```

```
      current = cameFrom.get(current);
    }
    // Reverse the order of this path so the start is at the beginning and the end is
at the end
    Collections.reverse(path);

    // Make sure the path doesn't exceed the max cost
    int travelledCost = 0;
    int lastListIndex = 1;
    while (travelledCost < maxCost && lastListIndex < path.size()) {
      travelledCost += path.get(lastListIndex).getCost();
      lastListIndex++;
    }
    // Get the path up to the point where the max cost is exceeded
    path = path.subList(0, lastListIndex);

    // Return a path object with details on the path
    return new Path<>(path, travelledCost);
  }

  /**
   * Sets the data associated with the cell at the coordinate
   *
   * @param x    x-coordinate of cell
   * @param y    y-coordinate of cell
   * @param cell data associated with the cell
   */
  public void set(int x, int y, E cell) {
    assert cellExists(x, y);
    grid[y][x] = cell;
  }

  /**
   * Gets the hexagon (with position data) associated with the cell at the coordinate
   *
   * @param x x-coordinate of cell
   * @param y y-coordinate of cell
   * @return {@link Hexagon} associated with the cell
   */
  public Hexagon getHexagon(int x, int y) {
    assert cellExists(x, y);
    return hexagonGrid[y][x];
  }

  /**
```

```
   * Iterates through all possible hexes in the hexagon grid and calls the consumer
with data for each cell
   *
   * @param consumer function to be called with details about the cell
   */
  @SuppressWarnings("unchecked")
  public void forEach(HexagonConsumer<E> consumer) {
    // For every row...
    for (int y = 0; y < grid.length; y++) {
      // For every column... (taking into account the alternating numbers of columns
in rows)
      for (int x = 0; x < grid[0].length - ((y + 1) % 2); x++) {
        // Send the details on the cell
        consumer.accept((E) grid[y][x], hexagonGrid[y][x], x, y);
      }
    }
  }

  /**
   * Implementation of {@link Iterator} for iterating over the cells in a hexagon
grid
   */
  private class HexagonGridIterator implements Iterator<E> {
    /**
     * Current x-coordinate state of the iterator
     */
    private int x;
    /**
     * Current y-coordinate state of the iterator
     */
    private int y;

    /**
     * Constructor for iterator. Initialises the state to (0, 0).
     */
    private HexagonGridIterator() {
      this.x = 0;
      this.y = 0;
    }

    /**
     * Checks if the iterator has another cell to give
     *
     * @return whether the current state of the iterator is valid
     */
```

```
  @Override
  public boolean hasNext() {
    return cellExists(x, y);
  }

  /**
   * Gets the next cell and increments the iterator state
   *
   * @return cell currently pointed to by the iterator
   */
  @Override
  public E next() {
    E next = get(x, y);
    if (x < grid[0].length - ((y + 1) % 2) - 1) {
      x++;
    } else {
      x = 0;
      y++;
    }
    return next;
  }
}

/**
 * Creates a new {@link HexagonGridIterator} for this hexagon grid
 *
 * @return the created iterator
 */
public Iterator<E> iterator() {
  return new HexagonGridIterator();
}

@Override
public String toString() {
  // Builder for the output
  StringBuilder builder = new StringBuilder();
  // For every row...
  for (int y = 0; y < grid.length; y++) {
    // Padding the row if needed to make a hexagon grid shape in the string
    if (y % 2 == 0) {
      builder.append(" ");
    }
    // For every column...
    for (int x = 0; x < grid[0].length - ((y + 1) % 2); x++) {
      // Set the output depending on whether the tile exists
```

```
        builder.append(get(x, y) == null ? "-" : "#").append(" ");
      }
      builder.append("\n");
    }
    return builder.toString();
  }

  /**
   * Gets the grid width of this hexagon grid (how many tiles the grid spans)
   *
   * @return grid width of this hexagon grid
   */
  public int getWidth() {
    return width;
  }

  /**
   * Gets the grid height of this hexagon grid (how many tiles the grid spans)
   *
   * @return grid height this hexagon grid
   */
  public int getHeight() {
    return height;
  }
}
```

com/mrbbot/civilisation/geometry/NoiseGenerator.java

```
package com.mrbbot.civilisation.geometry;

import java.util.Random;

/**
 * Class containing static methods for generating random noise to be used by
 * the terrain generator. Code used is from this YouTube video:
 * https://youtu.be/qChQrNWU9Xw
 */
public class NoiseGenerator {
  /**
   * Random number generator used internally
   */
  private static final Random RANDOM = new Random();
  /**
   * Random seed to be used by the generator
   */
  private static final int SEED = RANDOM.nextInt(1000000000);
```

```java
/**
 * Gets some random noise for the specified coordinate. This function will
 * always return the same value for the same coordinate in the same program
 * execution.
 *
 * @param x x-coordinate of noise to get
 * @param y y-coordinate of noise to get
 * @return random noise in the range (-1, 1)
 */
private static double getNoise(int x, int y) {
  // Set the seed of the random generator as a constant plus some multiple of
  // the x and y coordinates. This ensures the same coordinates generate the
  // same noise.
  RANDOM.setSeed((x * 49632) + (y * 325176) + SEED);
  // Return random number in the desired range
  return (RANDOM.nextDouble() * 2) - 1;
}

/**
 * Gets a smoothed version of the random noise for the specified coordinates.
 * Corners, edges, and the center are all taken into account with different
 * proportions.
 *
 * @param x x-coordinate of noise to get
 * @param y y-coordinate of noise to get
 * @return random noise in the range (-1, 1)
 */
private static double getSmoothNoise(int x, int y) {
  double topLeft = getNoise(x - 1, y - 1);
  double topRight = getNoise(x + 1, y - 1);
  double bottomRight = getNoise(x + 1, y + 1);
  double bottomLeft = getNoise(x - 1, y + 1);

  double left = getNoise(x - 1, y);
  double top = getNoise(x, y - 1);
  double right = getNoise(x + 1, y);
  double bottom = getNoise(x, y + 1);

  double center = getNoise(x, y);

  return ((topLeft + topRight + bottomRight + bottomLeft) / 16.0)
    + ((left + top + right + bottom) / 8.0)
    + (center / 4.0);
}
```

```
/**
 * Interpolates between two values using a cos function
 *
 * @param a first value
 * @param b second value
 * @param t amount to interpolate in the interval [0, 1]
 * @return a value between in the range [a, b]
 */
private static double cosInterpolate(double a, double b, double t) {
  t = (1.0 - Math.cos(Math.PI * t)) + 0.5;
  return (a * (1 - t)) + (b * t);
}

/**
 * Gets an interpolated version of the random noise. This function unlike the
 * others in this class takes doubles for the coordinates allowing for
 * decimal positions to be used.
 *
 * @param x x-coordinate of noise to get
 * @param y y-coordinate of noise to get
 * @return random noise in the range (-1, 1)
 */
public static double getInterpolatedNoise(double x, double y) {
  // Gets the fractional components of the x and y coordinates
  double xf = x - Math.floor(x);
  double yf = y - Math.floor(y);

  // Gets the floors and ceilings of the x and y coordinates. These are the
  // vertices of the quadrant to get the noise from.
  int xMin = (int) Math.floor(x);
  int yMin = (int) Math.floor(y);
  int xMax = xMin + 1;
  int yMax = yMin + 1;

  // Get noise for the top of the quadrant
  double top = cosInterpolate(
    getSmoothNoise(xMin, yMin),
    getSmoothNoise(xMax, yMin),
    xf
  );
  // Get noise for the bottom of the quadrant
  double bottom = cosInterpolate(
    getSmoothNoise(xMin, yMax),
    getSmoothNoise(xMax, yMax),
```

```
      xf
    );
    // Get noise for the position in the quadrant proportional to the
    // fractional coordinate components
    return cosInterpolate(top, bottom, yf);
  }
}
```

com/mrbbot/civilisation/geometry/Positionable.java

```
package com.mrbbot.civilisation.geometry;

/**
 * Interface representing a positionable 2D element with a x and y coordinate
 */
public interface Positionable {
  /**
   * Gets the x-coordinate of the element
   *
   * @return x-coordinate of the element
   */
  int getX();

  /**
   * Gets the y-coordinate of the element
   *
   * @return y-coordinate of the element
   */
  int getY();
}
```

com/mrbbot/civilisation/geometry/Traversable.java

```
package com.mrbbot.civilisation.geometry;

/**
 * Interface representing a traversable element
 */
public interface Traversable extends Positionable {
  /**
   * Gets the cost of traversing this element
   *
   * @return cost of traversing this element
   */
  int getCost();

  /**
```

```
   * Checks whether this tile can be traversed
   *
   * @return whether this tile can be traversed
   */
  boolean canTraverse();
}
```

com/mrbbot/civilisation/logic/CityBuildable.java

```
package com.mrbbot.civilisation.logic;

import com.mrbbot.civilisation.logic.techs.Unlockable;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.Building;
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.logic.unit.UnitType;
import com.mrbbot.civilisation.ui.game.BadgeType;

import java.util.ArrayList;
import java.util.Objects;

/**
 * Abstract class representing an item that can be built within a city by the
 * city instead of a worker
 */
public abstract class CityBuildable implements Unlockable {
  /**
   * Gets a city buildable from just its name
   *
   * @param name name of the city buildable to get
   * @return the city buildable with the specified name or null if it doesn't
   * exist
   */
  public static CityBuildable fromName(String name) {
    // Try get it as a unit type
    UnitType unitType = UnitType.fromName(name);
    if (unitType != null) return unitType;
    // Otherwise, try get it as a building (this will return null if it doesn't
    // exist)
    return Building.fromName(name);
  }

  /**
   * Class representing a detail in the city production list. This could be the
   * production cost, the amount of movement points, or something like the
```

```
 * gold per turn increase.
 */
public class Detail {
  /**
   * Type of badge that should be used to represent this detail
   */
  public final BadgeType badge;
  /**
   * Text contents of this detail
   */
  public final String text;

  /**
   * Creates a new detail
   *
   * @param badge badge type of the detail
   * @param text  text to be shown next to the badge
   */
  public Detail(BadgeType badge, String text) {
    this.badge = badge;
    this.text = text;
  }

  /**
   * Creates a new detail with a number that is automatically converted to a
   * string
   *
   * @param badge  badge type of the detail
   * @param number number to be shown next to the badge
   */
  public Detail(BadgeType badge, int number) {
    this(badge, String.valueOf(number));
  }
}

/**
 * Name of this buildable (i.e. what is displayed in the city production
 * list)
 */
protected final String name;
/**
 * Description of this buildable (i.e. what is displayed in the city
 * production list)
 */
protected final String description;
```

```java
/**
 * Amount of production points required to build this thing. Also determines
 * the gold cost (1.5x this value).
 */
@SuppressWarnings("WeakerAccess")
protected final int productionCost;
/**
 * Unlock ID of this buildable. Used to track what things are unlocked by a
 * technology.
 */
protected final int unlockId;

public CityBuildable(
  String name,
  String description,
  int productionCost,
  int unlockId
) {
  this.name = name;
  this.description = description;
  this.productionCost = productionCost;
  this.unlockId = unlockId;
}

@Override
public int hashCode() {
  // Name should be unique
  return name.hashCode();
}

@Override
public boolean equals(Object obj) {
  if (obj instanceof CityBuildable) {
    // Name should be unique
    return Objects.equals(name, ((CityBuildable) obj).name);
  }
  return false;
}

/**
 * Gets the name of the buildable to be displayed in the production list
 *
 * @return name of the buildable
 */
public final String getName() {
```

```
  return name;
}

/**
 * Gets the unlock ID of the buildable
 *
 * @return unlock ID of the buildable
 */
public final int getUnlockId() {
  return unlockId;
}

/**
 * Gets the description of the buildable to be displayed in the production
 * list
 *
 * @return description of the buildable
 */
public final String getDescription() {
  return description;
}

/**
 * Gets the required production total for this buildable
 *
 * @return production cost of this buildable
 */
public final int getProductionCost() {
  return productionCost;
}

/**
 * Calculates the gold cost of this item from the production cost
 *
 * @return productionCost * 1.5
 */
public final int getGoldCost() {
  return (int) Math.round(productionCost * 1.5);
}

/**
 * Checks if this buildable can be built with the player's current production
 * total
 *
 * @param productionTotal player's current production total to check against
```

```
     * @return whether the buildable can be built
     */
    public final boolean canBuildWithProduction(int productionTotal) {
      return productionTotal >= productionCost;
    }

    /**
     * Checks if this buildable can be built with the player's current gold total
     *
     * @param goldTotal player's current gold total to check against
     * @return whether the buildable can be built
     */
    public final boolean canBuildWithGold(int goldTotal) {
      return goldTotal >= getGoldCost();
    }

    /**
     * Gets the details to be displayed in the city production list for this
     * building. This should be overridden in subclasses to add more specific
     * details.
     *
     * @return details to be displayed
     */
    public ArrayList<Detail> getDetails() {
      ArrayList<Detail> details = new ArrayList<>();
      // Add cost details (common for all buildables)
      details.add(new Detail(BadgeType.PRODUCTION, productionCost));
      details.add(new Detail(BadgeType.GOLD, getGoldCost()));
      return details;
    }

    /**
     * Function that builds the buildable in the city. Must be overridden in
     * subclasses for actual implementation.
     *
     * @param city city to build the buildable in
     * @param game game containing the city
     * @return tile updated during the build process
     */
    public abstract Tile build(City city, Game game);

    /**
     * Determine if a buildable can be built in a city given the player's other
     * cities. Designed to be overridden in subclasses.
     *
```

```
   * @param city   target city to build in
   * @param cities player's other cities
   * @return reason why the buildable cannot be built, or an empty string if it
   * can
   */
  public String canBuildGivenCities(City city, ArrayList<City> cities) {
    return "";
  }
}
```

com/mrbbot/civilisation/logic/Living.java

```
package com.mrbbot.civilisation.logic;

import com.mrbbot.civilisation.geometry.Positionable;
import com.mrbbot.civilisation.logic.unit.Unit;
import javafx.geometry.Point2D;

import java.util.HashMap;
import java.util.Map;

/**
 * Abstract base living class. Describes something that has health and does
 * something every turn. Implements positionable, mappable, and turn handler
 * interfaces, but doesn't provide any implementations meaning this must be
 * done by subclasses.
 */
public abstract class Living implements Positionable, Mappable, TurnHandler {
  /**
   * Maximum health of the living object. Should naturally heal towards this
   * value each turn.
   */
  private int baseHealth;
  /**
   * Current health of the living object. If this value reaches 0, the living
   * object should be considered dead.
   */
  private int health;

  /**
   * Constructor for a new living object. Automatically sets the current health
   * to the maximum.
   *
   * @param baseHealth maximum health of this living object
   */
  public Living(int baseHealth) {
```

```
  this(baseHealth, baseHealth);
}

/**
 * Constructor for a new living object that doesn't necessarily have maximum
 * health.
 *
 * @param baseHealth maximum health of this living object
 * @param health     current health of this living object
 */
public Living(int baseHealth, int health) {
  this.baseHealth = baseHealth;
  this.health = health;
}

/**
 * Increases a living's health by the specified amount up to the maximum
 * health
 *
 * @param healing amount to increase the health by
 */
@SuppressWarnings("WeakerAccess")
public final void heal(int healing) {
  setHealth(health + healing);
}

/**
 * Checks if the living's health is below its maximum and heals it up to 5
 * health if it is.
 *
 * @return true if the living needed healing
 */
public final boolean naturalHeal() {
  // Check if healing is needed and heal up to 5 health if it is
  if (health < baseHealth) {
    heal(5);
    // Mark as healed
    return true;
  }
  return false;
}

/**
 * Decreases a living's health by the specified amount
 *
```

```
 * @param damage amount to decrease the health by
 */
public final void damage(int damage) {
  this.health -= damage;
}

/**
 * Checks if the living is dead (i.e. the health is less than or equal to 0)
 *
 * @return whether the living is dead
 */
public final boolean isDead() {
  return this.health <= 0;
}

/**
 * Sets the units maximum health. This will also set the health so that it's
 * the same proportion of health as it was before.
 *
 * @param baseHealth new maximum health
 */
public final void setBaseHealth(int baseHealth) {
  // Get the old proportion
  double percentOfBase = getHealthPercent();
  this.baseHealth = baseHealth;
  // Ensure the proportion remains the same
  this.health = (int) Math.ceil(percentOfBase * (double) this.baseHealth);
}

/**
 * Gets the current health of the living
 *
 * @return current health
 */
public int getHealth() {
  return health;
}

/**
 * Sets the current health of the living, making sure it doesn't exceed the
 * maximum
 *
 * @param health new current health
 */
public void setHealth(int health) {
```

```
    this.health = health;
    // Check if the health exceeds the maximum and set it to the maximum if it
    // does
    if (this.health > this.baseHealth) {
      this.health = this.baseHealth;
    }
  }

  /**
   * Gets the maximum health of the living object
   *
   * @return maximum health of the living object
   */
  public int getBaseHealth() {
    return baseHealth;
  }

  /**
   * Gets the percent health filled of the living
   *
   * @return percent health filled
   */
  public double getHealthPercent() {
    return (double) health / (double) baseHealth;
  }

  /**
   * Stores health information in a map so it can be saved/sent over the
   * network. This should be overridden by subclasses to add their additional
   * information.
   *
   * @return map containing health information
   */
  public Map<String, Object> toMap() {
    Map<String, Object> map = new HashMap<>();

    // Store health information
    map.put("baseHealth", baseHealth);
    map.put("health", health);

    return map;
  }

  /**
   * Handle a unit attacking this living object. Should be overridden in
```

```
     * subclasses to describe how attacking units should be affected.
     *
     * @param attacker the unit attacking this living object
     * @param ranged   whether this was a ranged attack
     */
    public abstract void onAttacked(Unit attacker, boolean ranged);

    /**
     * Gets the living's owner. Should be overridden in subclasses.
     *
     * @return the living's owner
     */
    public abstract Player getOwner();

    /**
     * Gets the living's position in the map. Should be overridden in subclasses.
     *
     * @return the living's position in the map
     */
    public abstract Point2D getPosition();
}
```

com/mrbbot/civilisation/logic/Mappable.java

```
package com.mrbbot.civilisation.logic;

import java.util.Map;

/**
 * Interface describing something that stores the state of itself in a map so
 * that it can be restored later. Used for sending the state over a network or
 * for storing it in a file.
 */
public interface Mappable {
    Map<String, Object> toMap();
}
```

com/mrbbot/civilisation/logic/Player.java

```
package com.mrbbot.civilisation.logic;

import javafx.scene.paint.Color;

import java.io.Serializable;

/**
 * Player object containing the player's ID and a function for calculation
```

```
 * their colour. Also implements serializable so it can be sent over the
 * network.
 */
public class Player implements Serializable {
  /**
   * ID of this player. Chosen by the user when they launch the game.
   */
  public String id;

  public Player(String id) {
    this.id = id;
  }

  /**
   * Gets the colour of this player. This is calculated from the hash code of
   * the player's id, so the same ID will always have the same colour. There's
   * also no need to send the colour over the network as it can be easily
   * recalculated.
   *
   * @return the colour representing this player to be used for unit rendering
   * and UI panels
   */
  public Color getColour() {
    // Calculate the hue from the hash code, hues can be a number from 0 to
    // 360.
    return Color.hsb(id.hashCode() % 360, 1, 1);
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof Player) {
      // The id of the player should be unique
      return id.equals(((Player) obj).id);
    }
    return false;
  }
}
```

com/mrbbot/civilisation/logic/PlayerStats.java

```
package com.mrbbot.civilisation.logic;

/**
 * Data class containing information about a player's statistics (their
 * science and gold) so that they can be displayed in the UI
 */
```

```java
public class PlayerStats {
  /**
   * The player's total science per turn from all their cities
   */
  public final int sciencePerTurn;
  /**
   * The player's total gold amount
   */
  public final int gold;
  /**
   * The player's total gold per turn from all their cities
   */
  public final int goldPerTurn;

  public PlayerStats(int sciencePerTurn, int gold, int goldPerTurn) {
    this.sciencePerTurn = sciencePerTurn;
    this.gold = gold;
    this.goldPerTurn = goldPerTurn;
  }
}
```

com/mrbbot/civilisation/logic/TurnHandler.java

```java
package com.mrbbot.civilisation.logic;

import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.Tile;

/**
 * Interface describing an object that contains logic that should be executed
 * at the beginning of every turn. This might be healing a unit, growing a
 * city, etc
 */
public interface TurnHandler {
  /**
   * Turn handler function definition
   *
   * @param game game the turn is taking place in
   * @return an array of tiles to rerender, if empty, should rerender all
   * tiles, if null, should rerender no tiles
   */
  Tile[] handleTurn(Game game);
}
```

com/mrbbot/civilisation/render/RenderCivilisation.java

```java
package com.mrbbot.civilisation.render;

import com.mrbbot.civilisation.render.map.RenderGame;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.RenderRoot;
import javafx.application.Platform;
import javafx.event.EventHandler;
import javafx.scene.AmbientLight;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseButton;
import javafx.scene.paint.Color;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Translate;

/**
 * Render object containing the game render that handles zooming, panning and
 * lighting. This is the root render object, so extends {@link RenderRoot} in
 * order to access methods that create a sub-scene that can be included in the
 * JavaFX application.
 */
@ClientOnly
public class RenderCivilisation extends RenderRoot<RenderGame> {
  /**
   * Most zoomed in scale.
   */
  private final static double MAX_ZOOM = 5;
  /**
   * Most zoomed out scale.
   */
  private final static double MIN_ZOOM = 80;
  /**
   * Easy way to enable/disable lighting. Used for development as lighting can
   * be quite annoying.
   */
  private final static boolean ENABLE_LIGHTING = false;

  /**
   * Last x-coordinate of drag. Used for calculating how much the screen was
   * dragged.
   */
  private double oldMouseX = -1;
  /**
   * Last y-coordinate of drag. Used for calculating how much the screen was
```

```
     * dragged.
     */
    private double oldMouseY = -1;

    /**
     * Point light source representing the sun. Rotated round the map to simulate
     * day/night.
     */
    private PointLight sun;
    /**
     * Point light source representing the moon. Rotated round the map to
     * simulate day/night.
     */
    private PointLight moon;
    /**
     * Rotate transform applied to both the sun and moon to simulate day/night.
     */
    private Rotate sunMoonRotate;

    /**
     * Creates a new instance of the root render object.
     *
     * @param root   root game render to show
     * @param width  width of the screen
     * @param height height of the screen
     */
    public RenderCivilisation(RenderGame root, int width, int height) {
      // Pass width/height to super constructor so an appropriately sized sub-
      // scene can be created.
      super(root, width, height);

      // Setup lighting if required
      if (ENABLE_LIGHTING) {
        // Create new lights for the sun/moon
        sun = new PointLight(Color.WHITE);
        moon = new PointLight(Color.BLACK);

        // Create and add transforms for the sun and the moon. Notice the moon
        // has a negative x-transform so it will always be on the opposite side
        // to the sun.
        sunMoonRotate = new Rotate(0, Rotate.Y_AXIS);
        sun.getTransforms().addAll(
          sunMoonRotate, new Translate(-20, 0, 0)
        );
        moon.getTransforms().addAll(
```

```
    sunMoonRotate, new Translate(20, 0, 0)
  );

  // Create an ambient light so that you can still see the map at night
  AmbientLight ambientLight = new AmbientLight(
    Color.color(0.1, 0.1, 0.1)
  );
  getChildren().addAll(sun, moon, ambientLight);

  // Set the time to sunset just to update the lights to something sensible
  setTime(90);
  // Start a thread to update the game time on a regular interval
  new Thread(this::runDayNightCycle, "DayNightCycle").start();
}

// Zooming
subScene.setOnScroll((e) -> {
  // Calculate a new zoom value from the current zoom and the scroll amount
  double newValue = camera.translate.getZ() + (e.getDeltaY() / 40);
  // Clamp the value to the min/max values
  if (newValue > -MAX_ZOOM) newValue = -MAX_ZOOM;
  if (newValue < -MIN_ZOOM) newValue = -MIN_ZOOM;
  // Set the new zoom value
  camera.translate.setZ(newValue);
});

// Panning
subScene.setOnMouseDragged((e) -> {
  // Only drag the map if the left mouse button is pressed
  if (e.getButton() == MouseButton.PRIMARY) {
    // Get mouse position
    double x = e.getX(), y = e.getY();
    // Check if this is the first coordinate of the drag
    if (oldMouseX == -1 || oldMouseY == -1) {
      oldMouseX = x;
      oldMouseY = y;
    } else {
      // If it's not, work out how far we've dragged
      double dX = x - oldMouseX;
      double dY = y - oldMouseY;
      // ...and store the now old values
      oldMouseX = x;
      oldMouseY = y;

      // Multiply this movement by a value proportional to the zoom amount
```

```
        double multiplier = (Math.abs(camera.translate.getZ()) / 30) * 2;

        dX *= multiplier;
        dY *= multiplier;

        // Translate the camera by the dragged amount
        camera.translateBy(-dX / 250, -dY / 250, 0);
      }
    }
  });

  subScene.setOnMouseReleased((e) -> {
    switch (e.getButton()) {
      // If the left mouse button was released, reset the old drag
      // coordinates
      case PRIMARY:
        oldMouseX = -1;
        oldMouseY = -1;
        break;
      // If the right mouse button was released, reset pathfinding and move
      // units if they were selected
      case SECONDARY:
        this.root.resetPathfinding();
        break;
    }
  });
}

/**
 * Called by the client to register key handlers. These don't work with the
 * sub-scene.
 *
 * @param scene        scene to add key handlers to
 * @param eventHandler extra event handler for key events
 */
public void setScene(
  Scene scene,
  EventHandler<? super KeyEvent> eventHandler
) {
  // Debug keyboard shortcuts
  scene.setOnKeyPressed((e) -> {
    switch (e.getCode()) {
      // Camera rotation for looking around the map
      case W:
        camera.rotateBy(-1, 0, 0);
```

```
        break;
      case S:
        camera.rotateBy(1, 0, 0);
        break;
      // Shortcut key to force rerender of all tiles
      case R:
        root.updateTileRenders();
        System.out.println("Updated tile renders");
        break;
    }
    // Handle extra shortcuts
    eventHandler.handle(e);
  });
}

/**
 * Function called in a separate thread to run the day/night cycle.
 */
private void runDayNightCycle() {
  try {
    while (true) {
      // Work out the second of the current minute including fractional
      // component
      double second = (System.currentTimeMillis() / 1000.0) % 60.0;

      // Set the time on the UI thread as non-UI threads can't update UI
      // components
      Platform.runLater(() -> setTime(second * 6));

      // Try to update the time 30 times-per-second.
      Thread.sleep(1000 / 30);
    }
  } catch (InterruptedException ignored) {
  }
}

/**
 * Sets the time of day represented by the angle of the sun.
 *
 * @param angle angle of the sun: sunrise(0) - midday (90) - sunset(180)
 *              - midnight (270) - sunrise (360)
 */
private void setTime(double angle) {
  // Update the angle of the sun/moon
  sunMoonRotate.setAngle(angle);
```

```
    // Calculate the intensity of the sun/moon
    double sunValue = Math.max(Math.sin(Math.toRadians(angle)), 0);
    double moonValue = (1 - sunValue) / 4;

    // Calculate/update  the colours of the sun/moon lights
    Color sunColour = Color.color(sunValue, sunValue, sunValue * 0.95);
    Color moonColour = Color.color(moonValue, moonValue, moonValue * 2);

    sun.setColor(sunColour);
    moon.setColor(moonColour);

    // Update the background colour of the game window
    subScene.setFill(sunColour);
  }
}
```

com/mrbbot/civilisation/ui/Screen.java

```
package com.mrbbot.civilisation.ui;

import javafx.scene.Scene;
import javafx.stage.Stage;

/**
 * Abstract class representing a screen that can be displayed. Currently there
 * are only two screens, the connection screen and the game screen. The screen
 * should be able to create a JavaFX scene containing the required UI
 * components.
 */
public abstract class Screen {
  /**
   * Creates a scene representing this screen
   *
   * @param stage  stage the scene would be placed in
   * @param width  width of the screen
   * @param height height of the screen
   * @return scene representing this screen
   */
  public abstract Scene makeScene(Stage stage, int width, int height);
}
```

com/mrbbot/civilisation/ui/UIHelpers.java

```
package com.mrbbot.civilisation.ui;

import javafx.scene.Node;
```

```java
import javafx.scene.control.Alert;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.paint.Color;

/**
 * Utility class containing common functions used by the UI package.
 */
public class UIHelpers {
  /**
   * Creates a solid background of a single colour
   *
   * @param colour colour to make the background
   * @return background object filled with the specified colour
   */
  public static Background colouredBackground(Color colour) {
    return new Background(new BackgroundFill(colour, null, null));
  }

  /**
   * Forces the specified CSS class to either be enabled or disabled
   *
   * @param node      node to toggle the class of
   * @param className CSS class to be toggled
   * @param active    whether to enable the class
   */
  public static void toggleClass(Node node, String className, boolean active) {
    if (active) {
      // Add the class if it's not already there
      if (!node.getStyleClass().contains(className)) {
        node.getStyleClass().add(className);
      }
    } else {
      // Otherwise remove it
      node.getStyleClass().remove(className);
    }
  }

  /**
   * Show a modal dialog to the user
   *
   * @param message message to show
   * @param isError whether to show an error or information icon
   */
  public static void showDialog(String message, boolean isError) {
```

```
    // Create the dialog
    Alert dialog = new Alert(
      isError
        ? Alert.AlertType.ERROR
        : Alert.AlertType.INFORMATION
    );
    dialog.setTitle(isError ? "Error" : "Message");
    dialog.setContentText(message);
    // Show the dialog
    dialog.show();
  }
}
```

com/mrbbot/civilisation/net/CivilisationServer.java

```
package com.mrbbot.civilisation.net;

import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.MapSize;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.net.packet.*;
import com.mrbbot.generic.net.Connection;
import com.mrbbot.generic.net.Handler;
import com.mrbbot.generic.net.Server;
import org.yaml.snakeyaml.Yaml;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Map;

/**
 * Class containing the implementation of the game server.
 */
public class CivilisationServer implements Handler<Packet> {
  /**
   * Instance of the external library used for saving/parsing YAML game saves.
   */
  public static final Yaml YAML = new Yaml();

  /**
   * File for the current game save the server is using
   */
  private final File gameFile;
  /**
```

```java
     * The server instance of the game. Contains all state details but no render
     * references.
     */
    private Game game;
    /**
     * The instance of the generic server for sending/receiving {@link Packet}s.
     */
    private Server<Packet> server;

    /**
     * Creates a completely new game server with the specified details
     *
     * @param gameFileName path for the game save file
     * @param gameName     name of the new game
     * @param mapSize      map size of the new game
     * @param port         port number to run the server on
     * @throws IOException if there are any server networking errors
     */
    public CivilisationServer(
      String gameFileName,
      String gameName,
      MapSize mapSize,
      int port
    ) throws IOException {
      // Create the reference to the game file
      this.gameFile = new File(gameFileName);
      // Create the new game
      game = new Game(gameName, mapSize);
      // Save and then immediately load the game so it's in the same state as if
      // it were just loaded (see the 2nd constructor)
      save();
      load();
      // Start the server using this instance as the packet handler (See
      // accept(Connection<Packet> Packet)).
      server = new Server<>(port, this);
    }

    /**
     * Creates a new game server loaded from an existing game save
     *
     * @param gameFileName path of the game save file
     * @param port         port number to run the server on
     * @throws IOException if there are any server networking errors
     */
    public CivilisationServer(String gameFileName, int port) throws IOException {
```

```
    // Create the reference to the game file
    this.gameFile = new File(gameFileName);
    // Check the save exists
    if (!this.gameFile.exists())
      throw new IllegalArgumentException("game file doesn't exist");
    // Load the game
    load();
    // Start the server using this instance as the packet handler (See
    // accept(Connection<Packet> Packet)).
    server = new Server<>(port, this);
  }

  /**
   * Closes the server's socket disconnecting all clients
   *
   * @throws IOException if there are any networking errors
   */
  public void close() throws IOException {
    server.close();
  }

  /**
   * Saves the game state to the game file
   */
  private void save() {
    // Try and create a file writer, closing it when the game state has been
    // written
    try (FileWriter writer = new FileWriter(gameFile)) {
      // Dump the game state as YAML
      YAML.dump(game.toMap(), writer);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Loads and overwrites the game state from the game file
   */
  private void load() {
    // Try and create a file reader, closing it when the game state has been
    // read
    try (FileReader reader = new FileReader(gameFile)) {
      // Read the game state as YAML
      //noinspection unchecked
      game = new Game(YAML.loadAs(reader, Map.class));
```

```
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Main packet handler for the server
   *
   * @param connection connection object for a client
   * @param data       packet the client has just sent, may be null if the
   *                   client has disconnected
   */
  @Override
  public void accept(Connection<Packet> connection, Packet data) {
    // Alias the connection id
    String id = connection.getId();
    if (data == null) {
      // If the player disconnects, mark them as not ready so the game waits
      // for them to reconnect
      game.readyPlayers.put(id, false);
      return;
    }
    // Otherwise depending on the type of packet...
    if (data instanceof PacketInit) {
      // Check if this is the first time the player has joined this game
      boolean shouldCreateStartingPackets = !game.containsPlayerWithId(id);

      // Broadcast the player change to every other client
      PacketPlayerChange packetPlayerChange = new PacketPlayerChange(id);
      game.handlePacket(packetPlayerChange);
      // Send the current game state to the new player
      connection.broadcastTo(new PacketGame(game.toMap()));
      connection.broadcastExcluding(packetPlayerChange);

      // Create the starting units (initial settler and warrior) if this is the
      // first time the player has joined this game.
      if (shouldCreateStartingPackets) {
        for (PacketUnitCreate packet : game.createStartingUnits(id)) {
          // Broadcast them to every client, not just the new player
          connection.broadcast(packet);
        }
      }
    } else if (data instanceof PacketReady) {
      // Set the players ready state
      game.readyPlayers.put(id, ((PacketReady) data).ready);
```

```
      // Check if all players have marked themselves as ready
      if (game.allPlayersReady()) {
        // Handle the turn and request all clients do the same
        PacketReady packetReady = new PacketReady(false);
        game.handlePacket(packetReady);
        connection.broadcast(packetReady);
      }
    } else if (data instanceof PacketUpdate) {
      // If this was a game state update, update the local state
      Tile[] tilesToUpdate = game.handlePacket(data);
      // Check if any units have died and remove them from the game
      if (tilesToUpdate != null && tilesToUpdate.length != 0) {
        for (Tile tile : tilesToUpdate) {
          if (tile.unit != null && tile.unit.isDead()) {
            game.units.remove(tile.unit);
            tile.unit = null;
          }
        }
      }
      // Send the update to all connected clients but the sender
      connection.broadcastExcluding(data);
    }

    // Save the game state to the file after handling the packet so the game
    // can be easily restored
    save();
  }

  /**
   * Entry point for a dedicated server (one without a UI/client)
   *
   * @param args command line arguments
   * @throws IOException if there are any server networking errors
   */
  public static void main(String[] args) throws IOException {
    new CivilisationServer(
      "saves" + File.separator + "game.yml",
      "Game",
      MapSize.STANDARD,
      1234
    );
  }
}
```

com/mrbbot/generic/net/Broadcaster.java

```java
package com.mrbbot.generic.net;

import java.util.function.Predicate;

/**
 * Interface describing an object that can send packets to another place.
 *
 * @param <T> type of data to be exchanged over the network
 */
public interface Broadcaster<T> {
  /**
   * Broadcasts data to all connections
   *
   * @param data data to be sent
   */
  void broadcast(T data);

  /**
   * Broadcasts data to connections that return true from the predicate
   *
   * @param data data to be sent
   * @param test function to test each connection ID against, if it returns
   *             true, the data is sent to that connection ID
   */
  void broadcastWhere(T data, Predicate<String> test);

  /**
   * Broadcasts data to all but the specified ID
   *
   * @param data data to be sent
   * @param id   connection ID to exclude from sending
   */
  void broadcastExcluding(T data, String id);

  /**
   * Broadcasts data to only the specified ID
   *
   * @param data data to be sent
   * @param id   connection ID to send to
   */
  void broadcastTo(T data, String id);
}
```

com/mrbbot/generic/net/BaseBroadcaster.java

```java
package com.mrbbot.generic.net;

/**
 * Base class for a broadcaster containing implementations of some of the
 * functions that are the same for every broadcaster.
 *
 * @param <T> type of data to be exchanged over the network
 */
abstract class BaseBroadcaster<T> implements Broadcaster<T> {
  /**
   * Broadcasts data to all but the specified ID
   * @param data data to be sent
   * @param id connection ID to exclude from sending
   */
  @Override
  public final void broadcastExcluding(T data, String id) {
    broadcastWhere(data, (testId) -> !testId.equals(id));
  }

  /**
   * Broadcasts data to only the specified ID
   * @param data data to be sent
   * @param id connection ID to send to
   */
  @Override
  public final void broadcastTo(T data, String id) {
    broadcastWhere(data, (testId) -> testId.equals(id));
  }
}
```

com/mrbbot/generic/net/Client.java

```java
package com.mrbbot.generic.net;

import java.io.IOException;
import java.net.Socket;
import java.util.function.Predicate;

/**
 * Generic client class for connecting to a generic server and exchanging data
 *
 * @param <T> type of data to be exchanged over the network
 */
public class Client<T> extends BaseBroadcaster<T> {
  /**
   * Connection object for the client representing the connection to the server
```

```
 */
private Connection<T> connection;
/**
 * TCP socket for transferring data to and from the server
 */
private Socket socket;

/**
 * Creates a new client and connects to the specified server
 *
 * @param host    host name of the server
 * @param port    port number the server is listening on
 * @param id      id for this connection
 * @param handler data handler for when data is received from the server
 * @throws IOException if there was a connection error
 */
public Client(
  String host,
  int port,
  String id,
  Handler<T> handler
) throws IOException {
  // Create the TCP socket
  socket = new Socket(host, port);
  // Create a connection object that waits for data from the socket and
  // facilitates sending data to the server
  connection = new Connection<>(
    socket,
    // Set the ID of the connection when one is received
    Connection::setId,
    (connection, data) -> {
      if (data != null) {
        // Pass data to the handler if it isn't null
        handler.accept(connection, data);
      }
    },
    this
  );
  // Send the desired ID as the first packet
  connection.send(id);
}

/**
 * Broadcasts data to the server
 *
```

```
   * @param data data to be sent
   */
  @Override
  public void broadcast(T data) {
    try {
      connection.send(data);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Broadcasts data to to the server if the connection ID matches the
   * predicate
   *
   * @param data data to be sent
   * @param test function to test each connection ID against, if it returns
   *             true, the data is sent to that connection ID
   */
  @Override
  public void broadcastWhere(T data, Predicate<String> test) {
    // Send the data if the connection ID matches
    if (test.test(connection.getId())) {
      try {
        connection.send(data);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

  /**
   * Closes the client's socket, disconnecting from the server
   *
   * @throws IOException if there was a networking error
   */
  public void close() throws IOException {
    socket.close();
  }
}
```

com/mrbbot/generic/net/ClientOnly.java

```
package com.mrbbot.generic.net;

/**
```

```
 * Annotation for marking something as only usable on the client side. Whilst
 * this doesn't actually enforce anything, it serves as an effective marker
 * during development.
 */
public @interface ClientOnly {
}
```

com/mrbbot/generic/net/Connection.java

```
package com.mrbbot.generic.net;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Date;
import java.util.function.Predicate;

/**
 * Class representing a connection between a client and server or vice-versa
 *
 * @param <T> type of data to be exchanged over the network
 */
public class Connection<T> implements Runnable, Broadcaster<T> {
  /**
   * Stream for sending data to the other side
   */
  private ObjectOutputStream outputStream;
  /**
   * Stream for receiving data from the other side
   */
  private ObjectInputStream inputStream;
  /**
   * Function to be called when an ID is received from the other side
   */
  private final IdHandler<T> idHandler;
  /**
   * Function to be called when data is received from the other side.
   */
  private final Handler<T> inputHandler;
  /**
   * Whether the connection is open and data is being sent.
   */
  private boolean open;
  /**
   * ID of this connection. Used for broadcast targeting.
```

```
 */
private String id;
/**
 * Thread that checks for data being sent and interprets it.
 */
private Thread thread;
/**
 * Broadcaster that actually handles sending data (the client or server)
 */
private Broadcaster<T> broadcaster;

/**
 * Constructor for creating a new connection
 *
 * @param socket       TCP socket for sending/receiving data to/from
 * @param idHandler    function to be called when the connection gets an ID
 * @param inputHandler function to be called when generic data is received
 * @param broadcaster  broadcaster for sending data to other connections
 * @throws IOException if there was an error creating in/output streams
 */
Connection(
  Socket socket,
  IdHandler<T> idHandler,
  Handler<T> inputHandler,
  Broadcaster<T> broadcaster
) throws IOException {
  // Create streams for the socket
  outputStream = new ObjectOutputStream(socket.getOutputStream());
  inputStream = new ObjectInputStream(socket.getInputStream());

  // Store handlers
  this.idHandler = idHandler;
  this.inputHandler = inputHandler;
  this.broadcaster = broadcaster;

  open = true;

  // Create a new thread that constantly attempts to read data
  thread = new Thread(this);
  thread.setName("Connection");
  thread.start();
}

/**
 * Sends the specified data to the receiving end of this connection
```

```
     *
     * @param object data to be sent
     * @throws IOException if the data cannot be sent
     */
    public void send(Object object) throws IOException {
      // Log the send request
      System.out.println(String.format(
        "[%s] %s -> %s",
        new Date().toString(),
        object.getClass().getSimpleName(),
        id
      ));
      // Send the data and flush the stream so that it's sent immediately
      outputStream.writeObject(object);
      outputStream.flush();
    }

    /**
     * Runner for the input checking thread.
     */
    @Override
    public void run() {
      // Keep checking until the socket is closed
      while (open) {
        try {
          // Read the data, this will block the thread until data is received
          Object object = inputStream.readObject();
          // Log the incoming data
          System.out.println(String.format(
            "[%s] %s <- %s",
            new Date().toString(),
            object.getClass().getSimpleName(),
            id
          ));
          // If an ID hasn't been set yet, assume this is an ID
          if (id == null) {
            idHandler.accept(this, (String) object);
          } else {
            // Otherwise, this is generic data, so handle it accordingly
            //noinspection unchecked
            inputHandler.accept(this, (T) object);
          }
        } catch (IOException e) {
          // If there was an error close the connection
          System.out.println(String.format(
```

```
        "[%s] Connection with \"%s\" closed: %s",
        new Date().toString(),
        id,
        e.getMessage()
      ));
      // Send null to the handler to signal the connection closing
      inputHandler.accept(this, null);
      open = false;
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}

/**
 * Gets this connection's ID or null if the ID hasn't been set yet
 *
 * @return this connection's ID
 */
public String getId() {
  return id;
}

/**
 * Sets this connection's ID
 *
 * @param id new ID for this connection
 */
void setId(String id) {
  this.id = id;
  thread.setName("Connection:" + id);
}

/**
 * Broadcasts data to all connections
 *
 * @param data data to be sent
 */
@Override
public void broadcast(T data) {
  broadcaster.broadcast(data);
}

/**
 * Broadcasts data to connections that return true from the predicate
```

```
     *
     * @param data data to be sent
     * @param test function to test each connection ID against, if it returns
     *             true, the data is sent to that connection ID
     */
    @Override
    public void broadcastWhere(T data, Predicate<String> test) {
      broadcaster.broadcastWhere(data, test);
    }

    /**
     * Broadcasts data to all but this connection
     *
     * @param data data to be sent
     */
    public void broadcastExcluding(T data) {
      broadcaster.broadcastExcluding(data, id);
    }

    /**
     * Broadcasts data to all but the specified ID
     *
     * @param data data to be sent
     * @param id   connection ID to exclude from sending
     */
    @Override
    public void broadcastExcluding(T data, String id) {
      broadcaster.broadcastExcluding(data, id);
    }

    /**
     * Broadcasts data to only this connection
     *
     * @param data data to be sent
     */
    public void broadcastTo(T data) {
      broadcaster.broadcastTo(data, id);
    }

    /**
     * Broadcasts data to only the specified ID
     *
     * @param data data to be sent
     * @param id   connection ID to send to
     */
```

```
  @Override
  public void broadcastTo(T data, String id) {
    broadcaster.broadcastTo(data, id);
  }
}
```

com/mrbbot/generic/net/Handler.java

```
package com.mrbbot.generic.net;

/**
 * Function to be called when incoming data is received.
 *
 * @param <T> type of data being exchanged over the network
 */
public interface Handler<T> {
  /**
   * Function called when incoming data is received
   *
   * @param connection the connection this data comes from
   * @param data       the data itself, or null if the connection has closed
   */
  void accept(Connection<T> connection, T data);
}
```

com/mrbbot/generic/net/Server.java

```
package com.mrbbot.generic.net;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Predicate;

/**
 * Generic server class for listening for clients' connections and data
 *
 * @param <T> type of data to be exchanged over the network
 */
public class Server<T> extends BaseBroadcaster<T> implements Runnable {
  /**
   * TCP server socket to listen for incoming connections on
   */
  private final ServerSocket serverSocket;
  /**
```

```
 * Handler function for incoming data coming from clients
 */
private final Handler<T> handler;
/**
 * Map mapping connection IDs to their connection objects
 */
private HashMap<String, Connection> connections;

/**
 * Creates a new server
 *
 * @param port    port number to listen for connections on
 * @param handler function to be called when data is received from a client
 * @throws IOException if the server socket cannot be created (likely
 *                     because the port has already be bound)
 */
public Server(int port, Handler<T> handler) throws IOException {
  // Create the TCP server socket
  this.serverSocket = new ServerSocket(port);
  // Store the handler so it can be called later
  this.handler = handler;
  // Initialise the connections map
  this.connections = new HashMap<>();

  // Create a new thread for handling incoming connections from clients
  Thread thread = new Thread(this, "Server");
  thread.start();
}

/**
 * Broadcasts data to all connections
 *
 * @param data data to be sent
 */
public void broadcast(T data) {
  broadcastWhere(data, (id) -> true);
}

/**
 * Broadcasts data to connections that return true from the predicate
 *
 * @param data data to be sent
 * @param test function to test each connection ID against, if it returns
 *             true, the data is sent to that connection ID
 */
```

```
public void broadcastWhere(T data, Predicate<String> test) {
  for (Map.Entry<String, Connection> connection : connections.entrySet()) {
    if (test.test(connection.getKey())) {
      try {
        connection.getValue().send(data);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}

/**
 * Function that runs the server. Called in a separate thread.
 */
@Override
public void run() {
  boolean open = true;
  // Whilst the socket is open...
  while (open) {
    try {
      // Listen for new connections, this blocks until a new connection
      // request is received or the socket is closed
      Socket socket = serverSocket.accept();
      // Create a connection object that waits for data from the socket and
      // facilitates sending data to the client
      new Connection<>(
        socket,
        (connection, id) -> {
          // Set the connection ID when it is sent
          connection.setId(id);
          // Store the connection in the map
          connections.put(id, connection);
          try {
            // Send the server's ID
            connection.send("Server");
          } catch (IOException e) {
            e.printStackTrace();
          }
        },
        (connection, data) -> {
          // If the connection's been closed (data is null)
          if (data == null) {
            // Remove the connection from the map
            connections.remove(connection.getId());
```

```
            }
            // Forward the incoming data onto the incoming data handler
            handler.accept(connection, data);
          },
          this
        );
      } catch (IOException e) {
        // Ignore the error if it was just the socket closing
        if (!e.getMessage().equals("socket closed")) {
          e.printStackTrace();
        }
        open = false;
      }
    }
  }

  /**
   * Closes the servers's socket, disconnecting all connected clients
   *
   * @throws IOException if there was a networking error
   */
  public void close() throws IOException {
    serverSocket.close();
  }
}
```

com/mrbbot/generic/net/IdHandler.java

```
package com.mrbbot.generic.net;

/**
 * Function to be called when a connection receives an ID
 *
 * @param <T> type of data being exchanged over the network
 */
interface IdHandler<T> {
  /**
   * Function called when a connection receives an ID
   *
   * @param connection the connection this ID is for
   * @param data       the request ID for this connection
   */
  void accept(Connection<T> connection, String data);
}
```

com/mrbbot/generic/net/ServerOnly.java

```java
package com.mrbbot.generic.net;

/**
 * Annotation for marking something as only usable on the server side. Whilst
 * this doesn't actually enforce anything, it serves as an effective marker
 * during development.
 */
public @interface ServerOnly {
}
```

com/mrbbot/generic/render/Render.java

```java
package com.mrbbot.generic.render;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Translate;

/**
 * Base render object class with default transformations that can be adjusted
 * as required. Extends group so more nodes can be added as children.
 */
public class Render
  extends Group {
  /**
   * Scale transform for this render object.
   */
  public Scale scale = new Scale();
  /**
   * Transform for rotations in the X-axis
   */
  public Rotate rotateX = new Rotate(0, Rotate.X_AXIS);
  /**
   * Transform for rotations in the Y-axis
   */
  public Rotate rotateY = new Rotate(0, Rotate.Y_AXIS);
  /**
   * Transform for rotations in the Z-axis
   */
  public Rotate rotateZ = new Rotate(0, Rotate.Z_AXIS);
  /**
   * Transform for translations
   */
  public Translate translate = new Translate();
```

```
public Render() {
  super();
  // Add the transforms so that translations are applied last (means they
  // won't affect scaling and rotation)
  getTransforms().addAll(scale, rotateX, rotateY, rotateZ, translate);
}

/**
 * Adds child nodes to this render object
 *
 * @param elements nodes to add
 */
public final void add(Node... elements) {
  getChildren().addAll(elements);
}

/**
 * Removes child nodes from this render object
 *
 * @param elements nodes to remove
 */
public final void remove(Node... elements) {
  getChildren().removeAll(elements);
}

/**
 * Resets all the transformations of this render object to their default
 * state.
 */
public final void reset() {
  translateTo(0, 0, 0);
  rotateTo(0, 0, 0);
  scaleTo(1);
}

/**
 * Translates this render object to the specified coordinates.
 *
 * @param x new x-coordinate for this render object
 * @param y new y-coordinate for this render object
 * @param z new z-coordinate for this render object
 */
public final void translateTo(double x, double y, double z) {
  translate.setX(x);
```

```
    translate.setY(y);
    translate.setZ(z);
  }

  /**
   * Rotates this render object to the specified angles in each axis
   *
   * @param xDegrees new x rotation in degrees for this render object
   * @param yDegrees new y rotation in degrees for this render object
   * @param zDegrees new z rotation in degrees for this render object
   */
  public final void rotateTo(
    double xDegrees,
    double yDegrees,
    double zDegrees
  ) {
    rotateX.setAngle(xDegrees);
    rotateY.setAngle(yDegrees);
    rotateZ.setAngle(zDegrees);
  }

  /**
   * Scales all axis of this render object to the same factor
   *
   * @param v new scale factor for all axis of this render object
   */
  public final void scaleTo(double v) {
    scaleTo(v, v, v);
  }

  /**
   * Scales the render object to the specified scale factors for each axis
   *
   * @param x new scale factor for the x-axis
   * @param y new scale factor for the y-axis
   * @param z new scale factor for the z-axis
   */
  public final void scaleTo(double x, double y, double z) {
    scale.setX(x);
    scale.setY(y);
    scale.setZ(z);
  }

  /**
   * Translates the render object by the specified amount, adding to the
```

```
   * existing values for translation.
   *
   * @param x increase in the x-axis translation amount
   * @param y increase in the y-axis translation amount
   * @param z increase in the z-axis translation amount
   */
  public final void translateBy(double x, double y, double z) {
    // Add to the existing values
    translate.setX(translate.getX() + x);
    translate.setY(translate.getY() + y);
    translate.setZ(translate.getZ() + z);
  }

  /**
   * Rotates the render object by the specified number of degrees in each axis,
   * adding to the existing values for rotation.
   *
   * @param xDegrees degrees increase in the x-rotation of this object
   * @param yDegrees degrees increase in the y-rotation of this object
   * @param zDegrees degrees increase in the z-rotation of this object
   */
  public final void rotateBy(
    double xDegrees,
    double yDegrees,
    double zDegrees
  ) {
    // Add to the existing values
    rotateX.setAngle(rotateX.getAngle() + xDegrees);
    rotateY.setAngle(rotateY.getAngle() + yDegrees);
    rotateZ.setAngle(rotateZ.getAngle() + zDegrees);
  }

}
```

com/mrbbot/generic/render/RenderCamera.java

```
package com.mrbbot.generic.render;

import javafx.scene.PerspectiveCamera;

/**
 * Render object that just contains a camera that handles displaying the
 * rest of the scene.
 */
public class RenderCamera
```

```
  extends Render {

  /**
   * Camera that displays the scene. Can be positioned as if it were on a
   * tripod so that different perspectives of the same scene can be seen.
   */
  PerspectiveCamera camera;

  RenderCamera() {
    super();

    // Create the camera
    camera = new PerspectiveCamera(true);

    // Give it an initial transformation looking down slightly leant back
    rotateX.setAngle(220);
    translate.setZ(-30);

    add(camera);
  }

}
```

com/mrbbot/generic/render/RenderData.java

```
package com.mrbbot.generic.render;

/**
 * Render object for that stores some additional data required for rendering.
 * Used by the game's render itself and also for individual tile renders.
 *
 * @param <T> type of the data to be stored along with the render
 */
public class RenderData<T>
  extends Render {
  /**
   * Data that is required for this object to be rendered containing
   * information about how the render should look.
   */
  public T data;

  public RenderData(T data) {
    super();
    this.data = data;
  }
```

```
}
```

com/mrbbot/generic/render/RenderRoot.java

```java
package com.mrbbot.generic.render;

import javafx.scene.SceneAntialiasing;
import javafx.scene.SubScene;
import javafx.scene.paint.Color;

/**
 * Root render object that handles creating a sub scene so that other objects
 * added to this can be scene. Also creates the camera for the scene.
 *
 * @param <T> render type that should be the root of the scene where all other
 *            children are added
 */
public class RenderRoot<T extends Render>
  extends Render {
  /**
   * Root render object of the scene where other children that make up the
   * scene should be added.
   */
  public final T root;
  /**
   * The camera used for rendering the scene. Can be transformed to see
   * different perspectives.
   */
  public final RenderCamera camera;
  /**
   * The sub scene that allows the root render object to be visible within a
   * JavaFX applications's scene.
   */
  public final SubScene subScene;

  /**
   * Creates a new root render object with camera and sub scene
   *
   * @param root   root render object to be rendered by the camera
   * @param width  width of the sub scene
   * @param height height of the sub scene
   */
  public RenderRoot(T root, int width, int height) {
    super();
```

```
    this.root = root;

    // Creates and adds the camera
    camera = new RenderCamera();
    add(root, camera);

    // Creates the new sub scene pointing to this as the root
    subScene = new SubScene(
      this,
      width,
      height,
      // Enable the depth buffer for 3D support
      true,
      // Enable antialiasing for smoother edges on 3D objects
      SceneAntialiasing.BALANCED
    );
    // Set the default background of the scene
    subScene.setFill(Color.WHITE);
    // Set the camera responsible for rendering the scene
    subScene.setCamera(camera.camera);
  }

}
```

com/mrbbot/civilisation/logic/map/Game.java

```
package com.mrbbot.civilisation.logic.map;

import com.mrbbot.civilisation.geometry.Hexagon;
import com.mrbbot.civilisation.geometry.HexagonGrid;
import com.mrbbot.civilisation.logic.CityBuildable;
import com.mrbbot.civilisation.logic.Living;
import com.mrbbot.civilisation.logic.Player;
import com.mrbbot.civilisation.logic.PlayerStats;
import com.mrbbot.civilisation.logic.Mappable;
import com.mrbbot.civilisation.logic.TurnHandler;
import com.mrbbot.civilisation.logic.techs.Unlockable;
import com.mrbbot.civilisation.logic.map.tile.Building;
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.map.tile.Terrain;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.logic.techs.PlayerTechDetails;
import com.mrbbot.civilisation.logic.techs.Tech;
import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.logic.unit.UnitAbility;
```

```java
import com.mrbbot.civilisation.logic.unit.UnitType;
import com.mrbbot.civilisation.net.packet.*;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.net.ServerOnly;
import javafx.geometry.Point2D;

import java.util.*;
import java.util.function.BiConsumer;
import java.util.function.Consumer;
import java.util.stream.Collectors;

/**
 * Main game logic class. Connects other parts of game logic together.
 */
public class Game implements Mappable, TurnHandler {
  /**
   * Message shown to user if someone wins by putting their cities on every
   * single tile
   */
  private static final String VICTORY_REASON_DOMINATION
    = "conquering every single tile";
  /**
   * Message shown to user if someone wins by researching, building and
   * launching a rocket unit
   */
  private static final String VICTORY_REASON_SCIENCE
    = "blasting off into space";

  /**
   * Name of the game, chosen by the user on initial create
   */
  private String name;
  /**
   * Hexagon grid of the game, used for positioning tiles
   */
  public HexagonGrid<Tile> hexagonGrid;
  /**
   * List of all the cities in the game (regardless of player)
   */
  public ArrayList<City> cities;
  /**
   * List of all the units in the game (regardless of player)
   */
  public ArrayList<Unit> units;
  /**
```

```
     * List of all the players in the game
     */
    public ArrayList<Player> players;
    /**
     * Total amount of science each player has. Maps players' ids to science
     * counts.
     */
    private Map<String, Integer> playerScienceCounts;
    /**
     * Total amount of gold each player has. Maps players' ids to gold counts.
     */
    private Map<String, Integer> playerGoldCounts;
    /**
     * Techs unlocked by each player. Map players' ids to sets of unlocked techs.
     * Sets are used as each technology can only be unlocked once by each player.
     */
    private Map<String, Set<Tech>> playerUnlockedTechs;
    /**
     * Techs currently being unlocked by each player. Maps players' ids to techs
     * currently being unlocked.
     */
    private Map<String, Tech> playerUnlockingTechs;
    /**
     * Players that have marked themselves as ready. Maps players' ids to their
     * ready state (true = ready).
     */
    @ServerOnly
    public Map<String, Boolean> readyPlayers = new HashMap<>();
    /**
     * The currently selected unit. Units can be selected by clicking on them.
     */
    @ClientOnly
    public Unit selectedUnit = null;
    /**
     * Whether the game is waiting for other players to click the ready button.
     * This is set to true when the user clicks the "Next Turn" button.
     */
    @ClientOnly
    public boolean waitingForPlayers = false;
    /**
     * The ID of the current player.
     */
    @ClientOnly
    private String currentPlayerId;
    /**
```

```
 * A function to be called whenever the current player's stats (gold/science
 * counts) change.
 */
@ClientOnly
private Consumer<PlayerStats> playerStatsListener;
/**
 * A function to be called whenever the current player's tech state changes.
 * This may be progress in researching a
 * technology or the actual unlocking of a technology.
 */
@ClientOnly
private Consumer<PlayerTechDetails> techDetailsListener;
/**
 * A function to be called whenever a message is to be sent to the user. On
 * the client, this is handled by displaying a message box.
 * <p>
 * The first parameter is the message to be displayed, and the second is
 * whether the message is an error or not (true = error).
 */
@ClientOnly
private BiConsumer<String, Boolean> messageListener;

/**
 * Constructor for a new game (not loaded from a file)
 *
 * @param name    name of the game
 * @param mapSize size of the game (contains information on width/height)
 */
public Game(String name, MapSize mapSize) {
  // Store the name
  this.name = name;

  // Create the hexagon grid
  hexagonGrid = new HexagonGrid<>(mapSize.width, mapSize.height, 1);
  // Create a tile object for every tile in the grid
  hexagonGrid.forEach((_tile, hex, x, y) ->
    hexagonGrid.set(x, y, new Tile(hex, x, y))
  );

  // Create empty lists
  cities = new ArrayList<>();
  units = new ArrayList<>();
  players = new ArrayList<>();

  // Create empty maps
```

```
    playerScienceCounts = new HashMap<>();
    playerGoldCounts = new HashMap<>();
    playerUnlockedTechs = new HashMap<>();
    playerUnlockingTechs = new HashMap<>();
  }

  /**
   * Constructor for a game (loaded from a file/over the network)
   *
   * @param map map containing details of game
   */
  public Game(Map<String, Object> map) {
    // Load the name of the game
    this.name = (String) map.get("name");

    // Load the player list
    //noinspection unchecked
    this.players = (ArrayList<Player>) ((List<String>) map.get("players"))
      .stream()
      // Create a new player for each player id
      .map(Player::new)
      .collect(Collectors.toList());

    // Load the terrain
    //noinspection unchecked
    List<List<Double>> terrainList = (List<List<Double>>) map.get("terrain");
    //noinspection unchecked
    List<List<Integer>> treeList = (List<List<Integer>>) map.get("trees");
    int height = terrainList.size();
    for (int y = 0; y < height; y++) {
      List<Double> terrainRow = terrainList.get(y);
      List<Integer> treeRow = treeList.get(y);
      int width = terrainRow.size();

      // If the hexagon grid hasn't been initialised yet, we now have all the
      // information required to make one
      if (hexagonGrid == null) {
        hexagonGrid = new HexagonGrid<>(width, height, 1);
      }

      for (int x = 0; x < width; x++) {
        double terrainHeight = terrainRow.get(x);
        boolean tree = treeRow.get(x) == 1;

        // Put the tile into the grid with the loaded height and tree state
```

```
      hexagonGrid.set(x, y, new Tile(
        hexagonGrid.getHexagon(x, y),
        x, y,
        terrainHeight, tree
      ));
    }
  }

  // Make sure the hexagon grid has been set
  assert hexagonGrid != null;

  // Load the cities
  //noinspection unchecked
  this.cities = (ArrayList<City>)
    ((List<Map<String, Object>>) map.get("cities"))
      .stream()
      .map(m -> new City(hexagonGrid, m))
      .collect(Collectors.toList());

  // Load the units
  //noinspection unchecked
  this.units = (ArrayList<Unit>)
    ((List<Map<String, Object>>) map.get("units"))
      .stream()
      .map(m -> new Unit(hexagonGrid, m))
      .collect(Collectors.toList());

  // Load player resource counts
  //noinspection unchecked
  this.playerScienceCounts = (Map<String, Integer>) map.get("science");
  //noinspection unchecked
  this.playerGoldCounts = (Map<String, Integer>) map.get("gold");

  // Load player unlocked techs. These are stored as strings by default not
  // tech names so have to be converted.
  //noinspection unchecked
  Map<String, ArrayList<String>> unlockedTechs =
    (Map<String, ArrayList<String>>) map.get("unlockedTechs");
  this.playerUnlockedTechs = new HashMap<>();
  for (Map.Entry<String, ArrayList<String>> e : unlockedTechs.entrySet()) {
    // Convert the list of strings to a list of techs
    ArrayList<Tech> list = (ArrayList<Tech>) e.getValue()
      .stream()
      .map(Tech::fromName)
      .collect(Collectors.toList());
```

```
    // Store these techs in a set to ensure uniqueness
    this.playerUnlockedTechs.put(
      e.getKey(),
      new HashSet<>(list)
    );
  }

  // Load player unlocking techs. Again techs are stored as strings so have
  // to be converted.
  //noinspection unchecked
  Map<String, String> unlockingTechs =
    (Map<String, String>) map.get("unlockingTechs");
  this.playerUnlockingTechs = new HashMap<>();
  for (Map.Entry<String, String> e : unlockingTechs.entrySet()) {
    // Store the tech, converting the name to a recognised tech object
    this.playerUnlockingTechs.put(e.getKey(), Tech.fromName(e.getValue()));
  }
}

/**
 * Sets the current player of the game. Also registers the player stats
 * listener so the interface can be updated.
 *
 * @param currentPlayerId     id of the current player
 * @param playerStatsListener function to be called when player stats change
 */
@ClientOnly
public void setCurrentPlayer(
  String currentPlayerId,
  Consumer<PlayerStats> playerStatsListener
) {
  this.currentPlayerId = currentPlayerId;
  this.playerStatsListener = playerStatsListener;
  // Send the initial player stats straight away
  sendPlayerStats();
}

/**
 * Registers the tech details listener.
 *
 * @param techDetailsListener function to be called when tech progress
 *                            changes
 */
@ClientOnly
```

```
public void setTechDetailsListener(
  Consumer<PlayerTechDetails> techDetailsListener
) {
  this.techDetailsListener = techDetailsListener;
  // Send the initial tech details straight away
  sendTechDetails();
}

/**
 * Registers the listener for messages
 *
 * @param messageListener function to be called when a message is sent
 *                        (1st parameter: message, 2nd parameter: isError)
 */
@ClientOnly
public void setMessageListener(BiConsumer<String, Boolean> messageListener) {
  this.messageListener = messageListener;
}

/**
 * Sends a message to the client if there is one
 *
 * @param message message to be sent
 * @param isError whether this message represents an error
 */
private void sendMessage(String message, boolean isError) {
  // Only sends the message if the message listener is defined
  if (this.messageListener != null) {
    this.messageListener.accept(message, isError);
  }
}

/**
 * Sends a message to the client if there is one for a specific user
 *
 * @param forPlayerId id of player to send the message to
 * @param message     message to be sent
 * @param isError     whether this message represents an error
 */
private void sendMessageTo(
  String forPlayerId,
  String message,
  boolean isError
) {
  // Only sends the message if the message listener is defined and if the
```

```
    // current player matches the message recipient
    if (this.messageListener != null
      && Objects.equals(forPlayerId, currentPlayerId)) {
      this.messageListener.accept(message, isError);
    }
  }

  /**
   * Stores the game state in a map so that it can be restored later. Used for
   * sending the game state over a network or for storing it in a file.
   *
   * @return map representing the game state
   */
  public Map<String, Object> toMap() {
    Map<String, Object> map = new HashMap<>();

    // Store the game name
    map.put("name", name);

    // Store the list of player ids
    List<String> playerList = players.stream()
      .map(player -> player.id)
      .collect(Collectors.toList());
    map.put("players", playerList);

    // Store the terrain
    ArrayList<ArrayList<Double>> terrainList = new ArrayList<>();
    ArrayList<ArrayList<Integer>> treeList = new ArrayList<>();
    int gridWidth = hexagonGrid.getWidth() + 1;
    int gridHeight = hexagonGrid.getHeight();
    for (int y = 0; y < gridHeight; y++) {
      ArrayList<Double> terrainRow = new ArrayList<>();
      // Trees are stored as integers to reduce file size
      ArrayList<Integer> treeRow = new ArrayList<>();
      for (int x = 0; x < gridWidth - ((y + 1) % 2); x++) {
        Terrain terrain = hexagonGrid.get(x, y).getTerrain();
        terrainRow.add(terrain.height);
        treeRow.add(terrain.hasTree ? 1 : 0);
      }
      terrainList.add(terrainRow);
      treeList.add(treeRow);
    }
    map.put("terrain", terrainList);
    map.put("trees", treeList);
```

```
// Store the cities
List<Map<String, Object>> cityList = cities.stream()
  // Delegate to the the city's toMap function to store it
  .map(City::toMap)
  .collect(Collectors.toList());
map.put("cities", cityList);

// Store the units
List<Map<String, Object>> unitList = units.stream()
  // Delegate to the the units's toMap function to store it
  .map(Unit::toMap)
  .collect(Collectors.toList());
map.put("units", unitList);

// Store player science/gold counts
map.put("science", playerScienceCounts);
map.put("gold", playerGoldCounts);

// Store player unlocked techs. As raw techs cannot be stored, these must
// be converted to tech names first.
Map<String, ArrayList<String>> unlockedTechsMap = new HashMap<>();
for (Map.Entry<String, Set<Tech>> e : playerUnlockedTechs.entrySet()) {
  unlockedTechsMap.put(
    e.getKey(),
    (ArrayList<String>) e.getValue().stream()
      // Convert the techs to just their names
      .map(Tech::getName)
      .collect(Collectors.toList())
  );
}
map.put("unlockedTechs", unlockedTechsMap);

// Store player unlocking techs. Again, raw techs cannot be stored so must
// be converted.
Map<String, String> unlockingTechsMap = new HashMap<>();
for (Map.Entry<String, Tech> e : playerUnlockingTechs.entrySet()) {
  // Only store the tech if the player is currently unlocking something
  if (e.getValue() != null) {
    unlockingTechsMap.put(
      e.getKey(),
      // Convert the tech to just its name
      e.getValue().getName()
    );
  }
}
```

```
    map.put("unlockingTechs", unlockingTechsMap);

    return map;
  }

  /**
   * Send a message indicating a player has won the game
   *
   * @param playerId id of the winning player
   * @param reason   reason the player has won the game (one of
   *                 {@link #VICTORY_REASON_DOMINATION} or
   *                 {@link #VICTORY_REASON_SCIENCE})
   */
  private void win(String playerId, String reason) {
    // Determine whether it's the current player that has won the game
    boolean victory = currentPlayerId == null
      || playerId.equals(currentPlayerId);
    String messageStart = victory ? "Victory!" : "Defeat!";
    String playerPart = playerId.equals(currentPlayerId)
      ? "You've"
      : String.format("%s has", playerId);
    // Send the formatted message
    sendMessage(
      String.format(
        "%s %s won the game by %s!",
        messageStart, playerPart, reason
      ),
      !victory
    );
  }

  /**
   * Checks if a player has won by covering the map with their cities and sends
   * a victory message if they have.
   */
  private void checkDominationVictory() {
    // Player id of potential winner
    String playerId = null;

    // Create a new iterator to iterate over the hexagon grid
    Iterator<Tile> tileIterator = hexagonGrid.iterator();
    while (tileIterator.hasNext()) {
      // Get the next tile to check
      Tile tile = tileIterator.next();
      // If the tile doesn't have a city, then there are tiles that aren't
```

```
    // covered. In that case, not all tiles are covered by cities.
    if (tile.city == null) return;
    // Set the potential winner to the first tile with a city
    if (playerId == null) playerId = tile.city.player.id;
    // Check that each subsequent tile with a city is owned by the same
    // player
    if (!playerId.equals(tile.city.player.id)) return;
  }

  // If there was a winning player, send the victory message
  if (playerId != null) {
    win(playerId, VICTORY_REASON_DOMINATION);
  }
}

/**
 * Checks if the player already contains a player with the specified ID
 *
 * @param id id to check
 * @return whether a player with that ID already exists
 */
@SuppressWarnings("BooleanMethodIsAlwaysInverted")
public boolean containsPlayerWithId(String id) {
  for (Player player : players) {
    if (player.id.equals(id)) return true;
  }
  return false;
}

/*
 * START PACKET HANDLING
 */

/**
 * Creates the unit described by the packet in the tile closest to the
 * desired location
 *
 * @param packet packet containing unit details
 * @return array containing tile the unit was placed on, or null if a tile
 * couldn't be found
 */
private Tile[] handleUnitCreatePacket(PacketUnitCreate packet) {
  // List of already checked tiles
  ArrayList<Tile> checkedTiles = new ArrayList<>();
  // Queue of tiles to check for placement
```

```
Queue<Tile> placementTilesToCheck = new LinkedList<>();
// Add the desired location as the first tile to check
placementTilesToCheck.add(hexagonGrid.get(packet.x, packet.y));

// While there are still tiles to check...
Tile tileToCheck;
while ((tileToCheck = placementTilesToCheck.poll()) != null) {
  // Determine if the tile is a capital of a city (we don't want to place
  // the unit there if it is)
  boolean tileIsCapital = tileToCheck.city != null
    && tileToCheck.city.getCenter().samePositionAs(tileToCheck);
  // If it's not a capital, there isn't a unit there, and we can traverse
  // it, it's a suitable tile, so break out of the search loop
  if (!tileIsCapital
    && tileToCheck.unit == null
    && tileToCheck.canTraverse()
  ) {
    break;
  }

  // Otherwise mark the tile as checked
  checkedTiles.add(tileToCheck);

  // Add all of the tiles neighbours that haven't already been checked to
  // the queue for checking
  placementTilesToCheck.addAll(
    hexagonGrid.getNeighbours(
      tileToCheck.x,
      tileToCheck.y,
      false
    ).stream()
      .filter(tile -> !checkedTiles.contains(tile))
      .collect(Collectors.toList())
  );
}

// If a suitable tile was found...
if (tileToCheck != null) {
  // Create the unit on that tile
  Player player = new Player(packet.id);
  Unit unit = new Unit(player, tileToCheck, packet.getUnitType());
  units.add(unit);

  // And update that tile's render
  return new Tile[]{tileToCheck};
```

```
    }

    // Otherwise return null
    return null;
  }

  /**
   * Moves a unit described in the packet to a new location, subtracting the
   * amount of movement points required for the operation.
   *
   * @param packet packet containing movement details
   * @return start and end tile of the movement for updating
   */
  private Tile[] handleUnitMovePacket(PacketUnitMove packet) {
    // Get the start/end tiles of the movement
    Tile startTile = hexagonGrid.get(packet.startX, packet.startY);
    Tile endTile = hexagonGrid.get(packet.endX, packet.endY);

    // Subtract the movement points from the unit's remaining this turn
    startTile.unit.remainingMovementPointsThisTurn -= packet.usedMovementPoints;
    // Check the the remaining points aren't negative (should never happen)
    assert startTile.unit.remainingMovementPointsThisTurn >= 0;

    // Move the unit to a new tile, marking the old tile as empty
    startTile.unit.tile = endTile;
    endTile.unit = startTile.unit;
    startTile.unit = null;

    // Update the start/end tiles
    return new Tile[]{startTile, endTile};
  }

  /**
   * Deletes the unit described in the packet, removing it from the game.
   *
   * @param packet packet containing deletion details
   * @return tile the unit was previously occupying
   */
  private Tile[] handleUnitDeletePacket(PacketUnitDelete packet) {
    // Get the tile the unit is currently occupying
    Tile tile = hexagonGrid.get(packet.x, packet.y);

    // Get the unit on the tile and check it exists
    Unit unit = tile.unit;
    if (unit != null) {
```

```
    // If it does, remove it from the tile and the game
    tile.unit = null;
    units.remove(unit);
  }

  return new Tile[]{tile};
}

/**
 * Attacks a living object (city/unit) as described by the packet. This may
 * cause the living object and/or the attacker to loose health. If a city was
 * attacked, and it runs out of health, it is given to the player who last
 * attacked it.
 *
 * @param packet packet containing attack details
 * @return tiles the attacker and target are occupying
 */
private Tile[] handleUnitDamagePacket(PacketDamage packet) {
  // Get the attacker/target tiles
  Tile attackerTile = hexagonGrid.get(packet.attackerX, packet.attackerY);
  Tile targetTile = hexagonGrid.get(packet.targetX, packet.targetY);

  // Determine if the target is a city
  boolean targetIsCity = targetTile.city != null
    && targetTile.city.getCenter().samePositionAs(targetTile);

  // Get the attacker
  Unit attacker = attackerTile.unit;
  // Get the target, prioritising units over cities if a unit is occupying a
  // city (i.e. defending it)
  Living target = targetTile.unit == null
    ? (targetIsCity ? targetTile.city : null)
    : targetTile.unit;

  // Check if the attacker exists and can attack
  if (attacker == null || !attacker.canAttack()) return null;
  // If the target doesn't exist, send a message stating such
  if (target == null) {
    sendMessageTo(
      attacker.player.id,
      "You can't attack nothing!",
      true
    );
    return null;
  }
```

```
// If the target and attacker are owned by the same player, send a message
// stating such
if (attacker.getOwner().equals(target.getOwner())) {
  sendMessageTo(
    attacker.player.id,
    "You can't attack yourself!",
    true
  );
  return null;
}
// If the attacker has already attacked this turn, send a message stating
// such
if (attacker.hasAttackedThisTurn) {
  sendMessageTo(
    attacker.player.id,
    "You've already attacked this turn!",
    true
  );
  return null;
}

// Get the number of tiles between the attacker and target
Point2D targetPos = target.getPosition();
Point2D attackerPos = attacker.getPosition();
double distanceBetween = targetPos.distance(attackerPos);
int tilesBetween = (int) Math.round(distanceBetween / Hexagon.SQRT_3);

// Check if a melee attack could take place, if so, perform it
if (
  attacker.hasAbility(UnitAbility.ABILITY_ATTACK)
    && tilesBetween <= 1
) {
  target.onAttacked(attacker, false);
  attacker.hasAttackedThisTurn = true;
}

// Check if a ranged attack could take place, if so, perform it
if (
  attacker.hasAbility(UnitAbility.ABILITY_RANGED_ATTACK)
    && tilesBetween <= 2
) {
  target.onAttacked(attacker, true);
  attacker.hasAttackedThisTurn = true;
}
```

```
    // Check if the target is a city, if it is and is now dead, give the city
    // to the player that attacked it last
    if (target instanceof City && target.isDead()) {
      City targetCity = (City) target;
      // Increase the cities health
      targetCity.setHealth(10);
      // Set the city's owner
      targetCity.setOwner(targetCity.lastAttacker.player);
      // Check for a domination victory
      checkDominationVictory();
      // Mark every tile for update (re-rendering city walls)
      return new Tile[]{};
    }

    // Mark the attacker and target tile for update
    return new Tile[]{attackerTile, targetTile};
  }

  /**
   * Creates a city at the location described by the packet.
   *
   * @param packet packet containing city creation information
   * @return an empty array of tiles signalling that all tiles should be
   * updated
   */
  private Tile[] handleCityCreatePacket(PacketCityCreate packet) {
    Player player = new Player(packet.id);
    // Create the new city
    cities.add(new City(hexagonGrid, packet.x, packet.y, player));
    // Recalculate player stats with this new city in place
    sendPlayerStats();
    // Signal an update of every tile
    return new Tile[]{};
  }

  /**
   * Grows a city to a specific set of points
   *
   * @param packet packet containing locations of points to grow to
   * @return an empty array of tiles signalling that all tiles should be
   * updated
   */
  private Tile[] handleCityGrowPacket(PacketCityGrow packet) {
    // Get tile with city
    Tile t = hexagonGrid.get(packet.x, packet.y);
```

```
  if (t.city != null) {
    t.city.growTo(packet.getGrownTo());
  }
  // Recalculate player stats with new tiles
  sendPlayerStats();
  // Signal an update of every tile
  return new Tile[]{};
}

/**
 * Renames a city described by the packet
 *
 * @param packet packet containing rename information
 * @return null signalling no tiles need to be updated
 */
private Tile[] handleCityRenamePacket(PacketCityRename packet) {
  // Get tile with city
  Tile t = hexagonGrid.get(packet.x, packet.y);
  if (t.city != null) {
    // Rename the city
    t.city.name = packet.newName;
  }
  // No need to update any tiles
  return null;
}

/**
 * Requests that a city start building something or purchase an item with
 * gold.
 *
 * @param packet packet containing build information
 * @return an array of tiles to be updated, or null if no tiles are to be
 * updated
 */
private Tile[] handleCityBuildRequestPacket(PacketCityBuildRequest packet) {
  // Get tile with city
  Tile t = hexagonGrid.get(packet.x, packet.y);
  if (t.city != null) {
    // Get thing to be built by the city
    CityBuildable buildable = packet.getBuildable();
    // If the city is going to build this with production...
    if (packet.withProduction) {
      // Mark it as the current build
      t.city.currentlyBuilding = buildable;
    } else if (
```

```
      // Otherwise if they can afford to buy it with gold
      buildable.canBuildWithGold(getPlayerGoldTotal(t.city.player.id))
    ) {
      // Subtract that amount from the player's gold balance
      increasePlayerGoldBy(t.city.player.id, -buildable.getGoldCost());
      // Build the thing, getting the tile that was updated
      Tile placed = buildable.build(t.city, this);
      // If there was an update, return it
      if (placed != null) return new Tile[]{placed};
    }
  }
  // No need to update any tiles
  return null;
}

/**
 * Requests that a worker build an improvement on a tile
 *
 * @param packet packet describing the improvement request
 * @return tiles updated by the improvement or null if no tiles are to be
 * updated
 */
private Tile[] handleWorkerImproveRequestPacket(
  PacketWorkerImproveRequest packet
) {
  // Get the tile containing the worker
  Tile t = hexagonGrid.get(packet.x, packet.y);
  if (t.unit != null) {
    // Request that the worker start building the improvement
    t.unit.startWorkerBuilding(packet.getImprovement());
    // Return the updated tile
    return new Tile[]{t};
  }
  // Otherwise, no need to update any tiles
  return null;
}

/**
 * Requests that a player start researching a new technology
 *
 * @param packet packet describing the research request
 * @return null as no tiles need to be updated
 */
private Tile[] handlePlayerResearchRequestPacket(
  PacketPlayerResearchRequest packet
```

```
) {
  // Set the unlocking tech to the one described in the packet
  playerUnlockingTechs.put(packet.playerId, packet.getTech());
  // Update the UI to reflect this change
  sendTechDetails();
  return null;
}

/**
 * Upgrades a unit to a more advanced type with better abilities.
 *
 * @param packet packet describing the upgrade request
 * @return array of tiles to be updated or null if no tiles need to be
 */
private Tile[] handleUnitUpgradePacket(PacketUnitUpgrade packet) {
  // Get te tile containing the unit to be upgraded
  Tile t = hexagonGrid.get(packet.x, packet.y);
  // Check the unit can be upgraded
  if (t.unit != null && t.unit.unitType.getUpgrade() != null) {
    // Upgrade the unit
    t.unit.unitType = t.unit.unitType.getUpgrade();
    // Set the new health proportional to its current health
    t.unit.setBaseHealth(t.unit.unitType.getBaseHealth());
    return new Tile[]{t};
  }
  return null;
}

/**
 * Brings a tile adjacent a city into that city's territory using gold.
 *
 * @param packet packet describing the purchase request
 * @return empty array signalling all tiles should be updated or null if the
 * tile couldn't be bought
 */
private Tile[] handlePurchaseTileRequestPacket(
  PacketPurchaseTileRequest packet
) {
  // Get the capital tile for the target city
  Tile t = hexagonGrid.get(packet.cityX, packet.cityY);
  if (t.city != null) {
    // Get the id of the city owner
    String playerId = t.city.player.id;

    // Get the tile requested to be purchased
```

```
Tile purchaseTile = hexagonGrid.get(packet.purchaseX, packet.purchaseY);

// Check if the tile is already part of a city, if it is, send a message
// stating such
if (purchaseTile.city != null) {
  sendMessageTo(
    playerId,
    "This tile is already part of a city!",
    true
  );
  return null;
}

// Check the tile is adjacent to the cities borders
ArrayList<Tile> neighbours = hexagonGrid.getNeighbours(
  packet.purchaseX,
  packet.purchaseY,
  false
);
boolean isNeighbour = neighbours.stream()
  .anyMatch(tile -> tile.city != null && tile.city.sameCenterAs(t.city));
// If it's not, send a message stating such
if (!isNeighbour) {
  sendMessageTo(
    playerId,
    "This tile isn't adjacent to this city!",
    true
  );
  return null;
}

// Calculate the gold cost of purchasing the tile based on the tile
// distance between it and the city center
double distanceBetween = t.getHexagon().getCenter()
  .distance(purchaseTile.getHexagon().getCenter());
int tilesBetween = (int) Math.round(distanceBetween / Hexagon.SQRT_3);
double cost = 50.0 * tilesBetween;
for (Building building : t.city.buildings) {
  cost *= building.expansionCostMultiplier;
}
int goldTotal = getPlayerGoldTotal(t.city.player.id);
int roundCost = (int) Math.round(cost);
// If the player doesn't have enough money to buy the tile, send them a
// message stating such
if (goldTotal < roundCost) {
```

```
      sendMessageTo(
        playerId,
        String.format(
          "This tile costs %d gold to purchase, you only have %d!",
          roundCost,
          goldTotal
        ),
        true
      );
      return null;
    }

    // Otherwise, decrease the players gold count by that amount
    increasePlayerGoldBy(playerId, -roundCost);
    // ...and grow the city to encompass that tile
    ArrayList<Point2D> grownTo = new ArrayList<>();
    grownTo.add(new Point2D(purchaseTile.x, purchaseTile.y));
    t.city.growTo(grownTo);
    // Update all tiles to regenerate city walls
    return new Tile[]{};
  }
  return null;
}

/**
 * Main turn handler for the game. Updates units (health, attack state,
 * remaining movement), cities (health, build progress) and players
 * (research).
 *
 * @param game game to update (redundant but required by the
 *             {@link TurnHandler} interface)
 * @return an array of tiles to be updated
 */
@Override
public Tile[] handleTurn(Game game) {
  ArrayList<Tile> updatedTiles = new ArrayList<>();

  // Update units
  for (Unit unit : units) {
    Tile[] unitUpdatedTiles = unit.handleTurn(this);
    // Add updated tiles to the list
    if (unitUpdatedTiles != null)
      Collections.addAll(updatedTiles, unitUpdatedTiles);
  }
```

```
    // Update cities
    for (City city : cities) {
      // Add updated tiles to the list
      Collections.addAll(updatedTiles, city.handleTurn(this));
      // If the city was low on health, and was healed, add the center to the
      // update list (for health bar re-render)
      if (city.naturalHeal()) {
        updatedTiles.add(city.getCenter());
      }
    }

    // Reset ready states
    waitingForPlayers = false;
    readyPlayers.clear();

    // Check to see if a player has won from a city growing
    checkDominationVictory();

    // Send player stats and tech details
    sendPlayerStats();
    sendTechDetails();

    // Update required tiles
    return updatedTiles.toArray(new Tile[]{});
  }

  /**
   * Packet handler. Delegates handling to one of the methods in this class
   * depending on the packet type.
   *
   * @param packet packet to process
   * @return an array of tiles to rerender, if empty, should rerender all
   * tiles, if null, should rerender no tiles
   */
  public Tile[] handlePacket(Packet packet) {
    // Checks the type of the packet and handles it accordingly
    if (packet instanceof PacketPlayerChange) {
      String newId = ((PacketPlayerChange) packet).id;
      if (!containsPlayerWithId(newId)) {
        players.add(new Player(newId));
      }
    } else if (packet instanceof PacketCityCreate) {
      return handleCityCreatePacket((PacketCityCreate) packet);
    } else if (packet instanceof PacketCityGrow) {
      return handleCityGrowPacket((PacketCityGrow) packet);
```

```
  } else if (packet instanceof PacketUnitCreate) {
    return handleUnitCreatePacket((PacketUnitCreate) packet);
  } else if (packet instanceof PacketUnitMove) {
    return handleUnitMovePacket((PacketUnitMove) packet);
  } else if (packet instanceof PacketUnitDelete) {
    return handleUnitDeletePacket((PacketUnitDelete) packet);
  } else if (packet instanceof PacketDamage) {
    return handleUnitDamagePacket((PacketDamage) packet);
  } else if (packet instanceof PacketCityRename) {
    return handleCityRenamePacket((PacketCityRename) packet);
  } else if (packet instanceof PacketCityBuildRequest) {
    return handleCityBuildRequestPacket((PacketCityBuildRequest) packet);
  } else if (packet instanceof PacketWorkerImproveRequest) {
    return handleWorkerImproveRequestPacket(
      (PacketWorkerImproveRequest) packet
    );
  } else if (packet instanceof PacketPlayerResearchRequest) {
    return handlePlayerResearchRequestPacket(
      (PacketPlayerResearchRequest) packet
    );
  } else if (packet instanceof PacketUnitUpgrade) {
    return handleUnitUpgradePacket((PacketUnitUpgrade) packet);
  } else if (packet instanceof PacketPurchaseTileRequest) {
    return handlePurchaseTileRequestPacket(
      (PacketPurchaseTileRequest) packet
    );
  } else if (packet instanceof PacketBlastOff) {
    win(((PacketBlastOff) packet).playerId, VICTORY_REASON_SCIENCE);
  } else if (packet instanceof PacketReady) {
    return handleTurn(this);
  }
  return null;
}
/*
 * END PACKET HANDLING
 */

/**
 * Calculates the player's gold/science per turn and sends them along with
 * the gold total.
 */
@ClientOnly
private void sendPlayerStats() {
  if (currentPlayerId != null && playerStatsListener != null) {
    int totalSciencePerTurn = 0;
```

```
    int totalGoldPerTurn = 0;
    for (City city : cities) {
      if (city.player.id.equals(currentPlayerId)) {
        totalSciencePerTurn += city.getSciencePerTurn();
        totalGoldPerTurn += city.getGoldPerTurn();
      }
    }
    playerStatsListener.accept(new PlayerStats(
      totalSciencePerTurn,
      getPlayerGoldTotal(currentPlayerId),
      totalGoldPerTurn
    ));
  }
}

/**
 * Sends the current players unlocked techs, and the currently researching
 * tech with its progress.
 */
@ClientOnly
private void sendTechDetails() {
  if (currentPlayerId != null && techDetailsListener != null) {
    techDetailsListener.accept(new PlayerTechDetails(
      getPlayerUnlockedTechs(currentPlayerId),
      getPlayerUnlockingTech(currentPlayerId),
      getPlayerUnlockingProgress(currentPlayerId)
    ));
  }
}

/**
 * Creates the packets for creating a player's starting units
 *
 * @param playerId player id to create units for
 * @return packets that create the starting units
 */
@ServerOnly
public PacketUnitCreate[] createStartingUnits(String playerId) {
  // Calculate the coordinate of the starting location based on the number
  // of players already in the game
  int numPlayers = players.size();
  int gridWidth = hexagonGrid.getWidth();
  int gridHeight = hexagonGrid.getHeight();
  int x = gridWidth / 2;
  int y = gridHeight / 2;
```

```
  switch (numPlayers) {
    case 1:
      x = 1;
      y = 1;
      break;
    case 2:
      x = gridWidth - 3;
      y = gridHeight - 3;
      break;
    case 3:
      x = gridWidth - 3;
      y = 1;
      break;
    case 4:
      x = 1;
      y = gridHeight - 3;
      break;
  }

  // Create the packets
  PacketUnitCreate[] packetUnitCreates = new PacketUnitCreate[]{
    new PacketUnitCreate(playerId, x, y, UnitType.SETTLER),
    new PacketUnitCreate(playerId, x, y, UnitType.WARRIOR)
  };
  // Handle them (server side only)
  for (PacketUnitCreate packetUnitCreate : packetUnitCreates)
    handlePacket(packetUnitCreate);
  // Return them so they can be sent to the clients
  return packetUnitCreates;
}

/**
 * Checks whether all players have marked themselves as ready
 *
 * @return whether all players have clicked the "Next Turn" button
 */
@ServerOnly
public boolean allPlayersReady() {
  for (Player player : players) {
    if (!readyPlayers.getOrDefault(player.id, false)) {
      return false;
    }
  }
  return true;
}
```

```java
/**
 * Gets the techs researched by the specified player
 *
 * @param playerId id of the player to check the techs of
 * @return researched techs by that player
 */
public Set<Tech> getPlayerUnlockedTechs(String playerId) {
  return playerUnlockedTechs.getOrDefault(playerId, new HashSet<>());
}

/**
 * Gets the tech currently being researched by the specified player
 *
 * @param playerId id of the player to check the tech of
 * @return currently researching tech of that player, or null if they're not
 * researching anything
 */
public Tech getPlayerUnlockingTech(String playerId) {
  return playerUnlockingTechs.get(playerId);
}

/**
 * Gets the progress through the research stage the specified player is
 *
 * @param playerId id of the player to check the progress of
 * @return progress of the currently researching tech, or 0 if not currently
 * researching
 */
public double getPlayerUnlockingProgress(String playerId) {
  // Get the current tech, and check it exists
  Tech unlockingTech = getPlayerUnlockingTech(playerId);
  if (unlockingTech == null) return 0;

  int scienceTotal = getPlayerScienceTotal(playerId);
  int scienceCost = unlockingTech.getScienceCost();
  if (scienceCost == 0) return 0;

  // Calculate the percent unlock, clamping the value to 1
  return Math.min((double) scienceTotal / (double) scienceCost, 1);
}

/**
 * Check whether a player has unlocked the specified item
 *
```

```
 * @param playerId   id of the player to check
 * @param unlockable unlockable item to check if unlocked
 * @return whether the item has been unlocked
 */
public boolean playerHasUnlocked(String playerId, Unlockable unlockable) {
  // Get the unlock id of the item, and check it's not unlocked by default
  int unlockId = unlockable.getUnlockId();
  if (unlockId == 0x00) return true;
  // Get the player's unlocked techs
  Set<Tech> unlockedTechs = getPlayerUnlockedTechs(playerId);
  for (Tech unlockedTech : unlockedTechs) {
    // Check if the tech includes the specified unlockable
    for (Unlockable unlock : unlockedTech.getUnlocks()) {
      if (unlock.getUnlockId() == unlockId) return true;
    }
  }
  return false;
}

/**
 * Gets a player resource count with a default value of 0
 *
 * @param counts   map containing player resource information
 * @param playerId id of the player to get the value of
 * @return player's count of that resource
 */
private int getPlayerResource(Map<String, Integer> counts, String playerId) {
  return counts.getOrDefault(playerId, 0);
}

/**
 * Gets a players gold total with a default value of 0
 *
 * @param playerId id of the player to get the value of
 * @return player's gold total
 */
public int getPlayerGoldTotal(String playerId) {
  return getPlayerResource(playerGoldCounts, playerId);
}

/**
 * Gets a players science total with a default value of 0
 *
 * @param playerId id of the player to get the value of
 * @return player's science total
```

```
 */
@SuppressWarnings("WeakerAccess")
public int getPlayerScienceTotal(String playerId) {
  return getPlayerResource(playerScienceCounts, playerId);
}

/**
 * Increases a player's resource count by the specified amount
 *
 * @param counts   map containing player resource information
 * @param playerId id of the player to increase
 * @param amount   amount to increase by (can be negative)
 */
private void increasePlayerResourceBy(
  Map<String, Integer> counts,
  String playerId,
  int amount
) {
  if (counts.containsKey(playerId)) {
    counts.put(playerId, counts.get(playerId) + amount);
  } else {
    counts.put(playerId, amount);
  }
  sendPlayerStats();
}

/**
 * Increases a player's gold total by the specified amount
 *
 * @param playerId id of the player to increase
 * @param gold     amount to increase by (can be negative)
 */
public void increasePlayerGoldBy(String playerId, int gold) {
  increasePlayerResourceBy(playerGoldCounts, playerId, gold);
}

/**
 * Increases a player's science total by the specified amount
 *
 * @param playerId id of the player to increase
 * @param science  amount to increase by (can be negative)
 */
public void increasePlayerScienceBy(String playerId, int science) {
  increasePlayerResourceBy(playerScienceCounts, playerId, science);
```

```
    // Get player's technology details
    Set<Tech> unlockedTechs = getPlayerUnlockedTechs(playerId);
    Tech unlockingTech = getPlayerUnlockingTech(playerId);
    double progress = getPlayerUnlockingProgress(playerId);

    // Check if a tech has now been unlocked
    if (unlockingTech != null && progress >= 1
      && !unlockedTechs.contains(unlockingTech)) {
      // Unlock the tech if it has
      playerUnlockedTechs.putIfAbsent(playerId, new HashSet<>());
      playerUnlockedTechs.get(playerId).add(unlockingTech);
      // Reset the current research
      playerUnlockingTechs.put(playerId, null);
      // Reduce science by the cost of the tech
      increasePlayerResourceBy(
        playerScienceCounts,
        playerId,
        -unlockingTech.getScienceCost()
      );
      // Send a message to the player
      sendMessageTo(
        playerId,
        String.format("%s has been unlocked!", unlockingTech.getName()),
        false
      );
    }
  }

  /**
   * Gets a player's cities
   *
   * @param id id of the player to get cities of
   * @return list of cities belonging to the player
   */
  public ArrayList<City> getPlayersCitiesById(String id) {
    return (ArrayList<City>) cities.stream()
      .filter(city -> city.player.id.equals(id))
      .collect(Collectors.toList());
  }
}
```

com/mrbbot/civilisation/logic/map/MapSize.java

```
package com.mrbbot.civilisation.logic.map;

/**
```

```
 * Enum for the different sizes a a map can be, used by the host/join screen to
 * list available sizes.
 */
public enum MapSize {
  TINY("Tiny", 5, 4),
  SMALL("Small", 10, 7),
  STANDARD("Standard", 20, 17),
  LARGE("Large", 30, 25);

  /**
   * User facing name of the map size
   */
  public String name;
  /**
   * Width of the hexagon grid for this size
   */
  public int width;
  /**
   * Height of the hexagon grid for this size
   */
  public int height;

  MapSize(String name, int width, int height) {
    this.name = name;
    this.width = width;
    this.height = height;
  }
}
```

com/mrbbot/civilisation/logic/techs/PlayerTechDetails.java

```
package com.mrbbot.civilisation.logic.techs;

import java.util.Set;

/**
 * Class containing details about a player's tech status. Includes their
 * unlocked techs, the tech they're currently unlocking, and the percent
 * unlocked of the current tech.
 */
public class PlayerTechDetails {
  /**
   * Techs that have been unlocked by the player. This is a Set because each
   * player can only unlock each tech once, so there should be no duplicate
   * entries.
   */
```

```
  public Set<Tech> unlockedTechs;
  /**
   * Tech that the player is currently unlocking. May be null if the player is
   * not researching anything at the moment.
   */
  public Tech currentlyUnlocking;
  /**
   * The progress of the current research project. This is a value in the range
   * [0, 1]. If the player isn't researching anything at the
   * moment, this value will be 0.
   */
  public double percentUnlocked;

  public PlayerTechDetails(
    Set<Tech> unlockedTechs,
    Tech currentlyUnlocking,
    double percentUnlocked
  ) {
    this.unlockedTechs = unlockedTechs;
    this.currentlyUnlocking = currentlyUnlocking;
    this.percentUnlocked = percentUnlocked;
  }
}
```

com/mrbbot/civilisation/logic/techs/Tech.java

```
package com.mrbbot.civilisation.logic.techs;

import com.mrbbot.civilisation.logic.map.tile.Building;
import com.mrbbot.civilisation.logic.map.tile.Improvement;
import com.mrbbot.civilisation.logic.unit.UnitType;
import javafx.scene.paint.Color;

import java.util.ArrayList;
import java.util.Set;

/**
 * Class representing a technology that can be unlocked by a player. This class
 * also contains the static tech registry that contains a list of all the
 * available techs. These are all initialised in this class too.
 */
public class Tech {
  /**
   * List of the techs that have been created. The first item added to this
   * list should be the root of the tech tree.
   */
```

```java
private static ArrayList<Tech> REGISTRY = new ArrayList<>();
/**
 * The maximum x-coordinate of a tech on the tech tree. This is used to work
 * out the width of the tech tree, so the scroll pane knows how long to
 * scroll for. See {@link com.mrbbot.civilisation.ui.game.UITechTree}.
 */
public static int MAX_X = 0;

/*
 * START TECH DEFINITIONS
 */
static {
  // Create the initial definitions for each tech, these include the name,
  // position of the tech tree, and their colour.

  // Level 0 (unlocked by default)
  Tech civilisation =
    new Tech("Civilisation", 0, 0, Color.GOLDENROD);

  // Level 1
  Tech agriculture = new Tech("Agriculture", 1, 0, Color.GREEN);

  // Level 2
  Tech forestry = new Tech("Forestry", 2, 0, Color.DARKGREEN);

  // Level 3
  Tech pottery = new Tech("Pottery", 3, -2, Color.FIREBRICK);
  Tech husbandry = new Tech("Husbandry", 3, -1, Color.PINK);
  Tech theWheel = new Tech("The Wheel", 3, 0, Color.GOLDENROD);
  Tech archery = new Tech("Archery", 3, 1, Color.RED);
  Tech mining = new Tech("Mining", 3, 2, Color.GREY);

  // Level 4
  Tech currency = new Tech("Currency", 4, -2, Color.GOLD);
  Tech dramaAndPoetry = new Tech("Drama", 4, -1, Color.PURPLE);
  Tech ironWorking = new Tech("Iron Working", 4, 2, Color.GREY);

  // Level 5
  Tech education = new Tech("Education", 5, 0, Color.LIGHTBLUE);

  // Level 6
  Tech industrialisation =
    new Tech("Industrialisation", 6, -1, Color.BLACK);
  Tech steel = new Tech("Steel", 6, 1, Color.GREY.darker());
```

```
// Level 7
Tech electricity = new Tech("Electricity", 7, -1, Color.GOLD);
Tech plastics = new Tech("Plastics", 7, 1, Color.PINK);

// Level 8
Tech rocketry = new Tech("Rocketry", 8, 0, Color.DARKBLUE);

// Specify the Unlockables that each tech unlocks. These could be
// improvements, buildings, or unit types.

// Level 0 (unlocked by default)
civilisation.unlocks(UnitType.SETTLER);
civilisation.unlocks(UnitType.SCOUT);
civilisation.unlocks(UnitType.WARRIOR);

// Level 1
agriculture.unlocks(Improvement.FARM);
agriculture.unlocks(UnitType.WORKER);

// Level 2
forestry.unlocks(Improvement.CHOP_FOREST);

// Level 3
pottery.unlocks(Building.MONUMENT);
husbandry.unlocks(Improvement.PASTURE);
theWheel.unlocks(Improvement.ROAD);
archery.unlocks(UnitType.ARCHER);
mining.unlocks(Improvement.MINE);
mining.unlocks(Building.WALL);

// Level 4
currency.unlocks(Building.BANK);
dramaAndPoetry.unlocks(Building.AMPHITHEATRE);
ironWorking.unlocks(UnitType.SWORDSMAN);

// Level 5
education.unlocks(Building.SCHOOL);
education.unlocks(Building.UNIVERSITY);

// Level 6
industrialisation.unlocks(Building.FACTORY);
steel.unlocks(UnitType.KNIGHT);

// Level 7
electricity.unlocks(Building.POWER_STATION);
```

```
plastics.unlocks(Building.SUPERMARKET);

// Level 8
rocketry.unlocks(UnitType.ROCKET);

// Specify the requirements for each of the techs. Techs can require
// multiple techs, but these must all be on a previous level.

// Level 0 techs have no requirements as they are unlocked by default.

// Level 1
agriculture.requires(civilisation);

// Level 2
forestry.requires(agriculture);

// Level 3
pottery.requires(agriculture);
husbandry.requires(forestry);
theWheel.requires(forestry);
archery.requires(forestry);
mining.requires(agriculture);

// Level 4
currency.requires(pottery);
currency.requires(husbandry);
dramaAndPoetry.requires(pottery);
dramaAndPoetry.requires(husbandry);
ironWorking.requires(archery);
ironWorking.requires(mining);

// Level 5
education.requires(dramaAndPoetry);
education.requires(theWheel);

// Level 6
industrialisation.requires(currency);
industrialisation.requires(education);
steel.requires(education);
steel.requires(ironWorking);

// Level 7
electricity.requires(industrialisation);
electricity.requires(steel);
plastics.requires(industrialisation);
```

```
    plastics.requires(steel);

    // Level 8
    rocketry.requires(electricity);
    rocketry.requires(plastics);

    // Calculate tech science costs
    for (Tech tech : Tech.REGISTRY) {
      // Even though the costs are only printed, and not used for anything
      // else, this function must be called in level order as the value is
      // cached and previous requirements' tech costs are used in the
      // calculations of later techs.
      int cost = tech.getScienceCost();
      System.out.println(String.format("%s: %d", tech.getName(), cost));
      if (tech.getX() > MAX_X) MAX_X = tech.getX();
    }
  }
  /*
   * END TECH DEFINITIONS
   */

  /**
   * Gets the root of the tech tree. This is the first item that was added to
   * the registry.
   *
   * @return root of the tech tree
   */
  public static Tech getRoot() {
    return REGISTRY.get(0);
  }

  /**
   * Gets a tech from just its name. Used to load techs from a map (network/
   * file)
   *
   * @param name name of the tech to load
   * @return Tech object for the tech or null if the tech doesn't exist
   */
  public static Tech fromName(String name) {
    for (Tech tech : REGISTRY) {
      if (tech.name.equals(name)) return tech;
    }
    return null;
  }
```

```java
/**
 * The user facing name of the tech. Used by
 * {@link com.mrbbot.civilisation.ui.game.UITechTree} when displaying a tech.
 */
private final String name;
/**
 * The x-coordinate of the tech. Used by
 * {@link com.mrbbot.civilisation.ui.game.UITechTree} to position a tech in
 * the tree.
 */
private final int x;
/**
 * The y-coordinate of the tech. Used by
 * {@link com.mrbbot.civilisation.ui.game.UITechTree} to position a tech in
 * the tree.
 */
private final int y;
/**
 * The colour of the tech. Used by
 * {@link com.mrbbot.civilisation.ui.game.UITechTree} when displaying a tech.
 */
private final Color colour;
/**
 * The techs that this tech requires to be unlocked before it itself can be
 * unlocked.
 */
private final ArrayList<Tech> requirements;
/**
 * The techs that require this tech before they can be unlocked. Used by
 * {@link com.mrbbot.civilisation.ui.game.UITechTree} to traverse and render
 * the tech tree.
 */
private final ArrayList<Tech> requiredBy;
/**
 * The unlockable items that this tech unlocks. These could be improvements,
 * buildings, or unit types.
 */
private final ArrayList<Unlockable> unlocks;
/**
 * The total science cost of unlocking this technology. The default value of
 * -1 indicates that the cost has not yet been calculated. Upon calling
 * {@link Tech#getScienceCost()} for the first time it will be calculated
 * and stored.
 */
private int scienceCost = -1;
```

```java
/**
 * Constructor to create a new tech object. Should only be called from within
 * this class.
 *
 * @param name   user facing name of the tech
 * @param x      x-coordinate of the tech in the tree
 * @param y      y-coordinate of the tech in the tree
 * @param colour colour of the tech in the tree
 */
private Tech(String name, int x, int y, Color colour) {
  this.name = name;
  this.x = x;
  this.y = y;
  this.colour = colour;

  // Initialise these as empty lists
  this.requirements = new ArrayList<>();
  this.requiredBy = new ArrayList<>();
  this.unlocks = new ArrayList<>();

  // Add this tech to the registry automatically
  REGISTRY.add(this);
}

/**
 * Marks this tech as requiring another before it can be unlocked. Allows the
 * caller to write tech.requires(other) meaning a tree can be constructed
 * with an English like language.
 *
 * @param tech the required tech
 */
private void requires(Tech tech) {
  requirements.add(tech);
  tech.requiredBy.add(this);
}

/**
 * Marks this tech as unlocking the specified unlockable. Allows the caller
 * to write tech.unlocks(something).
 *
 * @param unlockable an item this tech unlocks
 */
private void unlocks(Unlockable unlockable) {
  unlocks.add(unlockable);
```

```
}

/**
 * Gets the user facing name of this technology
 *
 * @return user facing name to be displayed in the tech tree
 */
public String getName() {
  return name;
}

/**
 * Gets the x-coordinate of this technology
 *
 * @return x-coordinate for positioning this tech in the tech tree
 */
public int getX() {
  return x;
}

/**
 * Gets the y-coordinate of this technology
 *
 * @return y-coordinate for positioning this tech in the tech tree
 */
public int getY() {
  return y;
}

/**
 * Gets the colour of this technology
 *
 * @return colour used for rendering this tech in the tech tree
 */
public Color getColour() {
  return colour;
}

/**
 * Gets a list of the techs required by this tech before it can be unlocked
 *
 * @return list of requirements
 */
public ArrayList<Tech> getRequirements() {
  return requirements;
```

```
}

/**
 * Gets a list of the techs that require this tech before they can be
 * unlocked.
 *
 * @return list of techs that require this tech
 */
public ArrayList<Tech> getRequiredBy() {
  return requiredBy;
}

/**
 * Gets a list of the unlockables that this tech unlocks and allows the
 * player to use. These could be improvements, buildings, or unit types.
 *
 * @return list of things this tech unlocks
 */
public ArrayList<Unlockable> getUnlocks() {
  return unlocks;
}

/**
 * Calculates and stores the science cost of unlocking this tech. This
 * function should be called in order of tech level as the cost depends on
 * the techs previous requirements (and their requirements, and so fourth)
 *
 * @return the total science cost of unlocking this tech
 */
public int getScienceCost() {
  // We only want to calculate this once as it's recursive and could
  // potentially take a long time
  if (scienceCost == -1) {
    if (requirements.size() == 0) {
      // If there aren't any requirements, the cost is 0 (i.e. it's already
      // unlocked)
      scienceCost = 0;
    } else {
      // Otherwise the cost is the sum of the requirements' costs + 25
      scienceCost = requirements.stream()
        .mapToInt(Tech::getScienceCost)
        .sum()
        + 25;
    }
  }
```

```
    return scienceCost;
  }

  /**
   * Checks if a player can unlock this tech given their previously unlocked
   * techs. In other words, this function checks all the tech's requirements
   * are fulfilled.
   *
   * @param unlockedTechs all the techs a player has unlocked
   * @return whether or not the player has met all the requirements
   */
  public boolean canUnlockGivenUnlocked(Set<Tech> unlockedTechs) {
    return unlockedTechs.containsAll(requirements) ||
      (requirements.size() == 1
        && requirements.get(0).requirements.size() == 0);
  }

  @Override
  public int hashCode() {
    // Name of the tech should be unique
    return name.hashCode();
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof Tech) {
      // Name of the tech should be unique
      return name.equals(((Tech) obj).name);
    }
    return false;
  }
}
```

com/mrbbot/civilisation/logic/techs/Unlockable.java

```
package com.mrbbot.civilisation.logic.techs;

/**
 * Interface describing an unlockable item. In the context of this game, this
 * would be an improvement, building, or unit type.
 */
public interface Unlockable {
  /**
   * Gets the user facing name for this unlock. Displayed in the tech tree by
   * {@link com.mrbbot.civilisation.ui.game.UITechTree}.
   *
```

```
     * @return user facing name for this unlock
     */
    String getName();

    /**
     * Gets a ID representing this unlock. This should be 0 (if the item is
     * unlocked by default) or a unique ID for the item
     *
     * @return ID representing this unlock
     */
    int getUnlockId();
}
```

com/mrbbot/civilisation/logic/unit/Unit.java

```
package com.mrbbot.civilisation.logic.unit;

import com.mrbbot.civilisation.geometry.HexagonGrid;
import com.mrbbot.civilisation.logic.Living;
import com.mrbbot.civilisation.logic.Player;
import com.mrbbot.civilisation.geometry.Positionable;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.Improvement;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import javafx.geometry.Point2D;

import java.util.*;

import static com.mrbbot.civilisation.logic.unit.UnitAbility.ABILITY_ATTACK;
import static com.mrbbot.civilisation.logic.unit.UnitAbility.ABILITY_RANGED_ATTACK;

/**
 * Class representing an instance of a unit in the game.
 */
public class Unit extends Living implements Positionable {
    /**
     * Random number generator for generating improvement metadata
     */
    private static final Random RANDOM = new Random();

    /**
     * The owner of this unit. Only the owner can move the unit, or perform an
     * action with it.
     */
    public Player player;
    /**
```

```
     * The tile this unit is currently occupying
     */
    public Tile tile;
    /**
     * The type of this unit. Contains information on the units abilities.
     */
    public UnitType unitType;

    /**
     * The number of movement points this unit has remaining this turn. When a
     * unit moves, the cost of the movement is taken away from this number. The
     * unit can't move if this value is 0.
     */
    public int remainingMovementPointsThisTurn;
    /**
     * Whether or not the unit has attacked a living object this turn. Each
     * attacking unit can only attack one unit per turn, so this ensures that
     * when it's set to true no more attacking can take place.
     */
    public boolean hasAttackedThisTurn;
    /**
     * What the unit is currently trying to build on the tile. Although this is
     * part of the Unit class, this is only relevant for units that have
     * {@link UnitAbility#ABILITY_IMPROVE}.
     */
    public Improvement workerBuilding = Improvement.NONE;
    /**
     * How many turns the current unit has left on its build project. Again, this
     * is only relevant for units that have {@link UnitAbility#ABILITY_IMPROVE}.
     */
    public int workerBuildTurnsRemaining = 0;

    /**
     * Creates a new unit object with the specified data
     *
     * @param player   owner of the unit
     * @param tile     tile the unit is occupying
     * @param unitType type of the unit
     */
    public Unit(Player player, Tile tile, UnitType unitType) {
      // Set required living parameters
      super(unitType.getBaseHealth());
      this.player = player;
      this.tile = tile;
      this.unitType = unitType;
```

```
  // Reset movement and attack state
  this.remainingMovementPointsThisTurn = unitType.getMovementPoints();
  this.hasAttackedThisTurn = false;

  // Check the tile doesn't already have a unit
  if (tile.unit != null) {
    throw new IllegalArgumentException(
      "Unit created on tile with another unit"
    );
  }
  tile.unit = this;
}

/**
 * Loads a unit from a map containing information about it
 *
 * @param grid hexagon grid containing the unit
 * @param map  map containing information on the unit
 */
public Unit(HexagonGrid<Tile> grid, Map<String, Object> map) {
  // Load base health and health for living
  super((int) map.get("baseHealth"), (int) map.get("health"));

  // Load player
  this.player = new Player((String) map.get("owner"));

  // Load tile from the hexagon grid
  this.tile = grid.get((int) map.get("x"), (int) map.get("y"));

  // Load the unit type and check it exists
  this.unitType = UnitType.fromName((String) map.get("type"));
  assert this.unitType != null;

  // Load movement and attack state
  this.remainingMovementPointsThisTurn =
    (int) map.get("remainingMovementPoints");
  this.hasAttackedThisTurn = canAttack()
    && (boolean) map.get("hasAttacked");

  // Load specific worker information
  if (map.containsKey("workerBuilding"))
    workerBuilding =
      Improvement.fromName((String) map.get("workerBuilding"));
  if (map.containsKey("workerBuildTurnsRemaining"))
```

```
    workerBuildTurnsRemaining =
      (int) map.get("workerBuildTurnsRemaining");

  // Check the tile doesn't already have a unit
  if (tile.unit != null) {
    throw new IllegalArgumentException(
      "Unit created on tile with another unit"
    );
  }
  tile.unit = this;
}

/**
 * Gets the x-coordinate of this unit
 *
 * @return x-coordinate of this unit
 */
@Override
public int getX() {
  return tile.x;
}

/**
 * Gets the y-coordinate of this unit
 *
 * @return y-coordinate of this unit
 */
@Override
public int getY() {
  return tile.y;
}

/**
 * Stores all the information required to recreate this unit in map
 *
 * @return map containing unit information
 */
@Override
public Map<String, Object> toMap() {
  // Store health data from living base class
  Map<String, Object> map = super.toMap();

  // Store the owner
  map.put("owner", player.id);
```

```
    // Store the location
    map.put("x", tile.x);
    map.put("y", tile.y);

    // Store the unit type
    map.put("type", unitType.getName());

    // Store unit specific information
    map.put("remainingMovementPoints", remainingMovementPointsThisTurn);
    if (canAttack()) map.put("hasAttacked", hasAttackedThisTurn);
    if (workerBuilding != Improvement.NONE)
      map.put("workerBuilding", workerBuilding.name);
    if (workerBuildTurnsRemaining != 0)
      map.put("workerBuildTurnsRemaining", workerBuildTurnsRemaining);

    return map;
  }

  /**
   * Requests that the unit (if it can) begin working on constructing the
   * specified improvement on it's current tile
   *
   * @param improvement improvement to build
   */
  public void startWorkerBuilding(Improvement improvement) {
    // Check the unit can build and isn't already building something
    if (hasAbility(UnitAbility.ABILITY_IMPROVE)
      && workerBuilding == Improvement.NONE) {
      // Update the building improvement and turns remaining
      workerBuilding = improvement;
      workerBuildTurnsRemaining = improvement.turnCost;
    }
  }

  /**
   * Gets a units abilities. Normally, this should be the same of the unit
   * type's abilities. However, when a worker is building something, it
   * shouldn't be able to move.
   *
   * @return number representing a workers abilities
   */
  private int getAbilities() {
    int abilities = unitType.getAbilities();
    // Check if the worker is building something, and prevent it from moving
    // if it is
```

```
    if (workerBuilding != Improvement.NONE) {
      abilities -= UnitAbility.ABILITY_MOVEMENT;
    }
    return abilities;
  }

  /**
   * Check if a unit has the specified ability. This should be one of the
   * constants in the {@link UnitAbility} class. See {@link UnitAbility} for
   * more details on how this works.
   *
   * @param ability ability to check if the unit has
   * @return whether or not the unit has the ability
   */
  public boolean hasAbility(int ability) {
    return (getAbilities() & ability) > 0;
  }

  /**
   * Check if the unit can attack a living object
   *
   * @return whether the unit can perform a melee or a ranged attack
   */
  public boolean canAttack() {
    return hasAbility(ABILITY_ATTACK) || hasAbility(ABILITY_RANGED_ATTACK);
  }

  /**
   * Handle a turn of the game. This resets a unit's movement counter and
   * attack state, whilst also progressing any improvement that's being built.
   *
   * @param game game to handle the turn of
   * @return an array of tiles to be updated containing the unit's tile, an
   * empty array if all tiles should be updated, or null if no tiles should be
   * updated.
   */
  @Override
  public Tile[] handleTurn(Game game) {
    // Naturally heal the unit and see if the tile now needs to be updated
    boolean tileUpdated = naturalHeal();
    boolean allTilesNeedReRendering = false;

    // Reset movement and attack state
    remainingMovementPointsThisTurn = unitType.getMovementPoints();
    hasAttackedThisTurn = false;
```

```
// Check if the unit is building something
if (workerBuilding != Improvement.NONE) {
  // Increase build progress
  workerBuildTurnsRemaining--;
  // Check if the improvement has now been built
  if (workerBuildTurnsRemaining == 0) {
    // Create a new map for metadata
    HashMap<String, Object> meta = new HashMap<>();

    // Check if the improvement used to be a road. If it did, all tiles
    // will need re-rendering to recalculate road adjacencies.
    if (tile.improvement == Improvement.ROAD)
      allTilesNeedReRendering = true;
    if (workerBuilding == Improvement.CHOP_FOREST) {
      // If the working was chopping a forest, add the production bonus to
      // the cities total.
      if (tile.city != null) tile.city.productionTotal += 30;
      tile.improvement = Improvement.NONE;
    } else {
      // Otherwise just set the tiles improvement to what was being built
      tile.improvement = workerBuilding;

      // Check if there is any metadata that should be added
      if (Improvement.FARM.equals(workerBuilding)) {
        // Generate the number of strips and the angle for the farm
        meta.put("strips", ((RANDOM.nextInt(3) + 1) * 2) + 1);
        meta.put("angle", RANDOM.nextInt(6) * 60);
      } else if (Improvement.MINE.equals(workerBuilding)) {
        // Generate the size/colour for each of the 3 rocks
        List<Double> sizes = new ArrayList<>();
        List<Integer> colours = new ArrayList<>();
        for (int i = 0; i < 3; i++) {
          sizes.add(RANDOM.nextDouble() / 3.0 + 0.5);
          colours.add(RANDOM.nextInt(3));
        }
        meta.put("sizes", sizes);
        meta.put("colours", colours);
      } else if (Improvement.ROAD.equals(workerBuilding)) {
        // Again if we're building a road, all tiles need to be updated to
        // recalculate road adjacencies
        allTilesNeedReRendering = true;
      }
    }
```

```
      tile.improvementMetadata = meta;
      // Reset the worker's building state
      workerBuilding = Improvement.NONE;
    }
    // Make sure the unit's tile is updated
    tileUpdated = true;
  }

  // Return an empty array if all tiles need to be updated
  return allTilesNeedReRendering
    ? new Tile[]{}
    : (tileUpdated
    ? new Tile[]{tile}
    : null);
}

/**
 * Handle another unit attacking this unit
 *
 * @param attacker the other unit attacking this unit
 * @param ranged   whether this was a ranged attack
 */
@Override
public void onAttacked(Unit attacker, boolean ranged) {
  // Damage the unit an amount based on the attacker's strength
  damage(attacker.unitType.getAttackStrength());
  if (!ranged) {
    // Only damage the attack if this wasn't a ranged attack
    attacker.damage(unitType.getBaseHealth() / 5);
  }
}

/**
 * Gets the owner of the unit
 *
 * @return owner of the unit
 */
@Override
public Player getOwner() {
  return player;
}

/**
 * Gets the position of the unit from the tile the unit is occupying
 *
```

```
   * @return position of the unit
   */
  @Override
  public Point2D getPosition() {
    return tile.getHexagon().getCenter();
  }
}
```

com/mrbbot/civilisation/logic/unit/UnitAbility.java

```
package com.mrbbot.civilisation.logic.unit;

/**
 * Class containing constants for unit abilities. Each of these is an integer
 * with one of the bits set to one. This allows for easy checking of a unit's
 * abilities.
 * <p>
 * For example, if we take the worker's ability number (ABILITY_MOVEMENT +
 * ABILITY_IMPROVE) this will result in the binary number: 10001. If we AND
 * this with the ABILITY_IMPROVE constant and then check if the number is
 * greater than 0, we'll be able to tell if the unit has the improve ability.
 * <p>
 * 10001
 * AND 10000
 * = 10000 (16) [16 is greater than 0, so the unit has the ability]
 * <p>
 * If we perform the same check with settling:
 * <p>
 * 10001
 * AND 00010
 * = 00000 (0) [0 is not greater than 0, so the unit doesn't have the ability]
 */
public final class UnitAbility {
  public static final int ABILITY_MOVEMENT = 0b1;
  public static final int ABILITY_SETTLE = 0b10;
  public static final int ABILITY_ATTACK = 0b100;
  public static final int ABILITY_RANGED_ATTACK = 0b1000;
  public static final int ABILITY_IMPROVE = 0b10000;
  public static final int ABILITY_BLAST_OFF = 0b100000;
}
```

com/mrbbot/civilisation/logic/unit/UnitType.java

```
package com.mrbbot.civilisation.logic.unit;

import com.mrbbot.civilisation.logic.CityBuildable;
import com.mrbbot.civilisation.logic.map.Game;
```

```java
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.net.packet.PacketUnitCreate;
import com.mrbbot.civilisation.ui.game.BadgeType;
import javafx.scene.paint.Color;

import java.util.ArrayList;

import static com.mrbbot.civilisation.logic.unit.UnitAbility.*;

/**
 * Class representing a type of unit that can be built within a city. The class
 * contains a variety of constants for the different types of units.
 */
public class UnitType extends CityBuildable {
  /**
   * Base unlock ID for unit types. Used to identify unit types that can be
   * unlocked.
   */
  private static int BASE_UNLOCK_ID = 0x20;

  /*
   * START UNIT TYPE DEFINITIONS
   */
  public static UnitType SETTLER = new UnitType(
    "Settler",
    "Creates cities",
    60,
    0x00,
    Color.GOLD,
    1,
    0,
    5,
    ABILITY_MOVEMENT + ABILITY_SETTLE
  );
  public static UnitType SCOUT = new UnitType(
    "Scout",
    "Moves around",
    40,
    0x00,
    Color.GREEN,
    4,
    5,
    25,
    ABILITY_MOVEMENT + ABILITY_ATTACK
```

```
);
public static UnitType WARRIOR = new UnitType(
  "Warrior",
  "Can attack adjacent units",
  50,
  0x00,
  Color.RED,
  2,
  10,
  50,
  ABILITY_MOVEMENT + ABILITY_ATTACK
);
public static UnitType SWORDSMAN = new UnitType(
  "Swordsman",
  "Can attack adjacent units",
  70,
  BASE_UNLOCK_ID,
  Color.BROWN.darker(),
  2,
  15,
  60,
  ABILITY_MOVEMENT + ABILITY_ATTACK
);
public static UnitType KNIGHT = new UnitType(
  "Knight",
  "Can attack adjacent units",
  90,
  BASE_UNLOCK_ID + 1,
  Color.GREY,
  2,
  20,
  70,
  ABILITY_MOVEMENT + ABILITY_ATTACK
);
public static UnitType ARCHER = new UnitType(
  "Archer",
  "Can attack units up to 2 tiles away",
  60,
  BASE_UNLOCK_ID + 2,
  Color.INDIANRED,
  2,
  5,
  30,
  ABILITY_MOVEMENT + ABILITY_RANGED_ATTACK
);
```

```
public static UnitType WORKER = new UnitType(
  "Worker",
  "Can improve a tile",
  40,
  BASE_UNLOCK_ID + 3,
  Color.DODGERBLUE,
  3,
  0,
  15,
  ABILITY_MOVEMENT + ABILITY_IMPROVE
);
public static UnitType ROCKET = new UnitType(
  "Rocket",
  "Wins the game",
  200,
  BASE_UNLOCK_ID + 4,
  Color.GREY.darker().darker().darker(),
  0,
  0,
  100,
  ABILITY_BLAST_OFF
);
/*
 * END UNIT TYPE DEFINITIONS
 */

//Define unit upgrade paths
static {
  WARRIOR.canUpgradeTo = SWORDSMAN;
  SWORDSMAN.canUpgradeTo = KNIGHT;
}

/**
 * Array containing all defined unit types.
 */
public static UnitType[] VALUES = new UnitType[]{
  SETTLER,
  SCOUT,
  WARRIOR,
  SWORDSMAN,
  KNIGHT,
  ARCHER,
  WORKER,
  ROCKET
};
```

```java
/**
 * Function to get a unit type from just its name
 *
 * @param name name of unit type to get
 * @return the unit type with the specified name or null if the unit type
 * doesn't exist
 */
public static UnitType fromName(String name) {
  // Iterates through all the unit types...
  for (UnitType value : VALUES) {
    // Check if the names match
    if (value.name.equals(name)) return value;
  }
  return null;
}

/**
 * Colour representing this unit (the torso colour, the other belt colour
 * represents the player)
 */
private final Color color;
/**
 * Base number of movement points units of this type should start with
 */
private final int movementPoints;
/**
 * Attack strength of this unit type (i.e. how much damage it will do to
 * other units or cities)
 */
private final int attackStrength;
/**
 * Starting health for the unit
 */
private final int baseHealth;
/**
 * A number representing this unit's abilities (see {@link UnitAbility}) for
 * more information on how this works.
 */
private final int abilities;
/**
 * A unit type that this unit can be upgraded to if it's been unlocked.
 */
private UnitType canUpgradeTo;
```

```
private UnitType(
  String name,
  String description,
  int productionCost,
  int unlockId,
  Color color,
  int movementPoints,
  int attackStrength,
  int baseHealth,
  int abilities
) {
  super(name, description, productionCost, unlockId);
  this.color = color;
  this.movementPoints = movementPoints;
  this.attackStrength = attackStrength;
  this.baseHealth = baseHealth;
  this.abilities = abilities;
}

/**
 * Gets the unit's colour
 *
 * @return unit's colour
 */
public Color getColor() {
  return color;
}

/**
 * Gets the unit's base movement points
 *
 * @return unit's base movement points
 */
@SuppressWarnings("WeakerAccess")
public int getMovementPoints() {
  return movementPoints;
}

/**
 * Gets the unit's attack strength
 *
 * @return unit's attack strength
 */
public int getAttackStrength() {
  return attackStrength;
```

```
}

/**
 * Gets the unit's base health
 *
 * @return unit's base/starting health
 */
public int getBaseHealth() {
  return baseHealth;
}

/**
 * Gets a number representing the units abilities
 *
 * @return unit's abilities
 */
int getAbilities() {
  return abilities;
}

/**
 * Gets the unit type that this unit type can update to
 *
 * @return upgraded unit type, or null if this unit type cannot be upgraded
 */
public UnitType getUpgrade() {
  return canUpgradeTo;
}

/**
 * Gets the details to be displayed in the city production list for this
 * unit type
 *
 * @return details to be displayed
 */
@Override
public ArrayList<Detail> getDetails() {
  ArrayList<Detail> details = super.getDetails();

  details.add(new Detail(BadgeType.HEALTH, baseHealth));
  if (movementPoints != 0)
    details.add(new Detail(BadgeType.MOVEMENT, movementPoints));
  if (attackStrength != 0)
    details.add(new Detail(BadgeType.ATTACK, attackStrength));
```

```
    return details;
  }

  /**
   * Builds an instance of this unit type in the specified city, creating a new
   * unit object
   *
   * @param city city to build in
   * @param game game the city is contained within
   * @return tile to update the render of (i.e. the tile the new unit was
   * created in)
   */
  @Override
  public Tile build(City city, Game game) {
    PacketUnitCreate packetUnitCreate = new PacketUnitCreate(
      city.player.id,
      city.getX(), city.getY(),
      this
    );
    Tile[] placedTiles = game.handlePacket(packetUnitCreate);
    return placedTiles != null
      && placedTiles.length > 0
      ? placedTiles[0]
      : null;
    // No need to broadcast packet as this method is called on the client and
    // the server automatically on receiving a PacketReady
  }
}
```

com/mrbbot/civilisation/render/map/RenderGame.java

```
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.geometry.Path;
import com.mrbbot.civilisation.logic.Player;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.logic.unit.UnitAbility;
import com.mrbbot.civilisation.net.packet.*;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.RenderData;
import javafx.geometry.Point2D;
import javafx.scene.Node;
```

```java
import javafx.scene.input.MouseButton;
import javafx.scene.input.PickResult;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;

import java.util.List;
import java.util.PriorityQueue;
import java.util.function.Consumer;

/**
 * Main render object for representing the game state. Handles rendering tiles
 * (terrain, improvements, units, cities).
 */
@ClientOnly
public class RenderGame extends RenderData<Game> {
  /**
   * Function to be called when the player (de)selects a unit. Null should be
   * passed on deselection.
   */
  private final Consumer<Unit> selectedUnitListener;
  /**
   * Function to be called when the player (de)selects a city. Null should be
   * passed on deselection.
   */
  private final Consumer<City> selectedCityListener;

  /**
   * The current player of the game for this client
   */
  public Player currentPlayer;
  /**
   * The city that has been selected by the player. Null if no city is
   * selected.
   */
  private City selectedCity;

  /**
   * Creates a new game render object
   *
   * @param data                 game to render
   * @param id                   id of the current player
   * @param selectedUnitListener function to be called when the selected unit
   *                             changes
   * @param selectedCityListener function to be called when the selected city
```

```
 *                              changes
 */
public RenderGame(
  Game data,
  String id,
  Consumer<Unit> selectedUnitListener,
  Consumer<City> selectedCityListener
) {
  super(data);
  this.selectedUnitListener = selectedUnitListener;
  this.selectedCityListener = selectedCityListener;

  // Find the current player
  for (Player player : data.players) {
    if (player.id.equals(id)) {
      currentPlayer = player;
      break;
    }
  }
  if (currentPlayer == null) {
    throw new IllegalStateException(
      "current player not found in player list"
    );
  }

  // Priority queue that ensures translucent tiles are added to the render
  // last. This is required for the translucency effect to work.
  final PriorityQueue<RenderTile> tilesToAdd = new PriorityQueue<>(
    // Orders tiles with translucent tiles at the end of the queue
    (a, b) -> {
      if (a.isTranslucent() && !b.isTranslucent()) return 1;
      else if (b.isTranslucent() && !a.isTranslucent()) return -1;
      else return 0;
    }
  );

  // Iterates through every possible hex tile position
  data.hexagonGrid.forEach((tile, hex, x, y) -> {
    // Creates the render object for that tile
    RenderTile renderTile = new RenderTile(
      tile,
      data.hexagonGrid.getNeighbours(x, y, false)
    );
    tile.renderer = renderTile;
```

```
// Registers click listener for this tile
renderTile.setOnMouseClicked((e) -> {
  // Ignore the click if we're waiting for other players
  if (data.waitingForPlayers) return;

  if (e.getButton() == MouseButton.PRIMARY) {
    // If this was a left click, the user is trying to select this tile
    if (tile.city != null
      && tile.city.getCenter().samePositionAs(tile)
      && tile.city.player.equals(currentPlayer)) {
      // Prioritise selecting a capital city is there is one
      setSelectedCity(tile.city);
      setSelectedUnit(null);
      return;
    }
    // Otherwise select the unit on this tile
    setSelectedCity(null);
    // tile.unit will be null if there isn't a unit, deselecting any
    // previously selected units
    setSelectedUnit(tile.unit);
  } else if (e.getButton() == MouseButton.SECONDARY) {
    // If this was a right click, the user is trying to purchase a tile
    // or attack something
    if (selectedCity != null) {
      // If there's a selected city, this must be a purchase request
      PacketPurchaseTileRequest packetPurchaseTileRequest
        = new PacketPurchaseTileRequest(
        selectedCity.getX(),
        selectedCity.getY(),
        tile.x,
        tile.y
      );
      // Try the purchase request, if null was received, then it didn't
      // work so there's no need to broadcast it
      if (handlePacket(packetPurchaseTileRequest) != null) {
        // If it did work, update the game state of other clients
        Civilisation.CLIENT.broadcast(packetPurchaseTileRequest);
      }
    } else if (data.selectedUnit != null) {
      // If there's a selected unit, try and attack with it
      PacketDamage packetDamage = new PacketDamage(
        data.selectedUnit.tile.x,
        data.selectedUnit.tile.y,
        tile.x,
        tile.y
```

```
      );
      // If that returned null, the attack didn't work, so there's no
      // need to broadcast it
      if (handlePacket(packetDamage) != null) {
        // If it did work, update the game state of other clients
        Civilisation.CLIENT.broadcast(packetDamage);
      }
    }
  }
});

// Registers drag listener for this tile
renderTile.setOnMouseDragged((e) -> {
  // Ignore the click if we're waiting for other players
  if (data.waitingForPlayers) return;

  // If this was a right click drag, the user is trying to path-find
  if (e.getButton() == MouseButton.SECONDARY) {
    // If there isn't a start to the path yet, mark this initial tile
    // as it, providing it contains a unit that can be moved
    if (pathStartTile == null
      && renderTile.data.unit != null
      && renderTile.data.unit.player.equals(currentPlayer)
      && renderTile.data.unit.hasAbility(UnitAbility.ABILITY_MOVEMENT)) {
      pathStartTile = renderTile;
      // Show the pathfinding overlay
      pathStartTile.setOverlayVisible(true);
    }

    // If there's a starting tile
    if (pathStartTile != null) {
      // Get the tile the user is dragging over and check it exists
      RenderTile pickedTile = getTileFromPickResult(e.getPickResult());
      if ((pickedTile == null || pickedTile != pathEndTile)
        && pathEndTile != null) {
        // If it's different to the last end, reset the pathfinding end
        resetPathfindingEnd();
      }
      // If there was a tile
      if (pickedTile != null) {
        // Check if a path can be made to the tile
        RenderTile potentialEnd = pickedTile;

        List<Tile> path = data.hexagonGrid.findPath(
          pathStartTile.data.x,
```

```
              pathStartTile.data.y,
              potentialEnd.data.x,
              potentialEnd.data.y,
              pathStartTile.data.unit.remainingMovementPointsThisTurn
            ).path;

            // Mark the path for the user
            path.forEach((p) -> p.renderer.setOverlayVisible(true));

            // Set the end if there's a valid path
            if (path.size() > 1) {
              potentialEnd = path.get(path.size() - 1).renderer;
            }
            pathEndTile = potentialEnd;
            pathEndTile.setOverlayVisible(path.size() > 1);
          }
        }
      }
    });

    // Adds the tile to the render queue
    tilesToAdd.add(renderTile);
  });

  // Add a box underneath the tiles to represent a boardgame board that the
  // game is being played on
  Point2D topLeftCenter =
    data.hexagonGrid.get(0, 0).getHexagon().getCenter();
  Box gameBoard = new Box(
    Math.abs(topLeftCenter.getX() * 2) + 4.5,
    Math.abs(topLeftCenter.getY() * 2) + 3,
    1
  );
  gameBoard.setTranslateZ(-0.5);
  gameBoard.setMaterial(new PhongMaterial(Color.WHITESMOKE));
  add(gameBoard);

  // Add all the renders of the tiles, adding translucent tiles last
  RenderTile t;
  while ((t = tilesToAdd.poll()) != null) add(t);
}

/**
 * Sets the selected unit, notifying the listener of the change
 *
```

```
 * @param unit new selected unit, can be null if a unit has been deselected
 */
public void setSelectedUnit(Unit unit) {
  // Check if the unit exists and belongs to the current player (you can't
  // selected other player's units, that would be unfair)
  if (unit != null && unit.player.equals(currentPlayer)) {
    // Check if there's already a selected unit and deselect it if there is
    if (data.selectedUnit != null) {
      data.selectedUnit.tile.selected = false;
      // Rerender the containing tile
      data.selectedUnit.tile.renderer.updateRender();
    }
    // Mark the new unit as selected and rerender the containing tile
    unit.tile.selected = true;
    unit.tile.renderer.updateRender();
    // Update the game state and notify the unit listener
    data.selectedUnit = unit.tile.unit;
    selectedUnitListener.accept(data.selectedUnit);
  } else {
    // Otherwise, if the unit doesn't exist or doesn't belong to the current
    // player, deselect the current unit
    if (data.selectedUnit != null) {
      data.selectedUnit.tile.selected = false;
      // Rerender the containing tile
      data.selectedUnit.tile.renderer.updateRender();
      // Update the game state and notify the unit listener
      data.selectedUnit = null;
      selectedUnitListener.accept(null);
    }
  }
}

/**
 * Sets the selected city, notifying the listener of the change. There's no
 * need to check the owner here, as this is done in the click handler.
 *
 * @param city new selected city, can be null if a city has been deselected
 */
public void setSelectedCity(City city) {
  // Update the game state and notify the city listener
  selectedCity = city;
  selectedCityListener.accept(city);
}

/**
```

```
 * Deletes a unit from the game, updating the containing tile render
 *
 * @param unit      unit to delete
 * @param broadcast whether to broadcast this change to other clients
 */
public void deleteUnit(Unit unit, boolean broadcast) {
  if (unit != null) {
    // Deselect this unit if it was the selected unit
    if (data.selectedUnit == unit) setSelectedUnit(null);

    // Remove the unit from the tile and rerender the tile
    unit.tile.unit = null;
    unit.tile.renderer.updateRender();

    // Remove the unit and broadcast the change if required
    data.units.remove(unit);
    if (broadcast) {
      Civilisation.CLIENT.broadcast(new PacketUnitDelete(
        unit.tile.x,
        unit.tile.y
      ));
    }
  }
}

/**
 * Requests that all tiles be rerendered following a big game state changed
 * (i.e. city growth)
 */
public void updateTileRenders() {
  data.hexagonGrid.forEach(
    (gridTile, _hex, _x, _y) -> gridTile.renderer.updateRender()
  );
}

/**
 * Start tile of the selected path
 */
private RenderTile pathStartTile;
/**
 * End tile of the selected path
 */
private RenderTile pathEndTile;

/**
```

```
 * Gets a tile render object from a raycast result
 *
 * @param result result of a raycast
 * @return render tile hit by the ray or null if no tile was hit
 */
private RenderTile getTileFromPickResult(PickResult result) {
  // Check if there even was a selected node
  if (result == null) return null;
  Node node = result.getIntersectedNode();
  if (node == null) return null;

  // Traverse up the tree until a tile render object is found, returning it
  // if it was
  do {
    if (node instanceof RenderTile) return (RenderTile) node;
    node = node.getParent();
  } while (node != null);

  // Otherwise return null if no tile render could be found
  return null;
}

/**
 * Reset the path, so that a new path can be drawn in its place
 */
private void resetPathfindingEnd() {
  if (pathEndTile != null) {
    // Resets the path overlays for the path
    pathEndTile.setOverlayVisible(false);
    data.hexagonGrid.forEach(
      (t, _hex, _x, _y) -> t.renderer.setOverlayVisible(false)
    );
    pathEndTile = null;
  }
}

/**
 * Resets the pathfinding state, moving a unit if a path was found. Called
 * when the mouse is released.
 */
public void resetPathfinding() {
  // Check if a path was found
  if (pathStartTile != null && pathEndTile != null) {
    Path<Tile> path = data.hexagonGrid.findPath(
      pathStartTile.data.x,
```

```
      pathStartTile.data.y,
      pathEndTile.data.x,
      pathEndTile.data.y,
      pathStartTile.data.unit.remainingMovementPointsThisTurn
    );
    // Check if the path was valid
    if (path.path.size() > 1) {
      int usedMovementPoints = path.totalCost;

      // Subtract the required movement points, checking the value didn't go
      // below 0
      pathStartTile.data.unit.remainingMovementPointsThisTurn -=
        usedMovementPoints;
      assert pathStartTile.data.unit.remainingMovementPointsThisTurn >= 0;

      // Update the game state of this and other clients
      Civilisation.CLIENT.broadcast(new PacketUnitMove(
        pathStartTile.data.x,
        pathStartTile.data.y,
        pathEndTile.data.x,
        pathEndTile.data.y,
        usedMovementPoints
      ));
      // Move the unit
      pathStartTile.data.unit.tile = pathEndTile.data;
      pathEndTile.data.unit = pathStartTile.data.unit;
      pathStartTile.data.unit = null;

      // Move the unit selection if the unit was selected
      pathEndTile.data.selected = pathStartTile.data.selected;
      pathStartTile.data.selected = false;

      // Update the tile renders to reflect the departure/arrival of units
      pathStartTile.updateRender();
      pathEndTile.updateRender();

      // Notify the selected unit listener of the change too
      selectedUnitListener.accept(data.selectedUnit);
    }
  }

  // Reset the pathfinding overlays
  if (pathStartTile != null) {
    pathStartTile.setOverlayVisible(false);
    pathStartTile = null;
```

```
    }
    if (pathEndTile != null) {
      resetPathfindingEnd();
    }
  }

  /**
   * Handle an incoming packet that affects the game state. Updates the
   * updated tiles' renders too.
   *
   * @param packet packet to handle
   * @return a list of tiles that were updated, an empty array if all tiles
   * were updated, or null if no tiles were updated
   */
  public Tile[] handlePacket(Packet packet) {
    // Handle the packet
    Tile[] tilesToUpdate = data.handlePacket(packet);
    // Rerender the required tiles
    if (tilesToUpdate != null) {
      if (tilesToUpdate.length == 0) {
        updateTileRenders();
      } else for (Tile tile : tilesToUpdate) {
        // Check if any units died, removing them if they did
        if (tile.unit != null && tile.unit.isDead()) {
          deleteUnit(tile.unit, false);
        } else {
          tile.renderer.updateRender();
        }
      }
    }
    return tilesToUpdate;
  }
}
```

com/mrbbot/civilisation/render/map/RenderHealthBar.java

```
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.logic.Living;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.RenderData;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;

/**
```

```java
 * Render object for a health bar. Used by units and cities.
 */
@ClientOnly
public class RenderHealthBar extends RenderData<Living> {
  /**
   * Whether an extended health bar should be used. Primarily for cities.
   */
  private final boolean extended;
  /**
   * Cylinder representing the healthy part of the health bar
   */
  private Cylinder healthPart;
  /**
   * Cylinder representing the damaged part of the health bar
   */
  private Cylinder remainingPart;

  /**
   * Constructor for health bar render object
   *
   * @param data     living object to render health for
   * @param extended whether an extended health bar should be used
   */
  public RenderHealthBar(Living data, boolean extended) {
    super(data);
    this.extended = extended;

    // Translate the health bar up
    translateTo(0, 0, extended ? 1.6 : 0.8);
    rotateTo(0, 0, 90);

    // Create the cylinders
    healthPart = new Cylinder(0.1, extended ? 2 : 1);
    remainingPart = new Cylinder(0.05, 0);

    remainingPart.setMaterial(new PhongMaterial(Color.SLATEGREY));

    add(healthPart);
    add(remainingPart);

    // Update the state of the health bar render
    updateRender(data);
  }

  /**
```

```
 * Calculate the colour for the healthy part of the bar
 *
 * @param healthPercent percentage health of the living object
 * @return colour to be used for rendering the health bar
 */
private Color colorForHealthPercent(double healthPercent) {
  // Use a different colour depending on the interval the health percent is
  // in
  if (healthPercent > 0.6) {
    return Color.LIMEGREEN;
  } else if (healthPercent > 0.4) {
    return Color.YELLOW;
  } else if (healthPercent > 0.2) {
    return Color.ORANGERED;
  } else {
    return Color.RED;
  }
}

/**
 * Updates the render's state based on the health of the living object it
 * represents
 *
 * @param living living object containing health data
 */
void updateRender(Living living) {
  // If the living doesn't exist, or it's at max health, hide the bar
  if (living == null || living.getHealth() == living.getBaseHealth()) {
    setVisible(false);
  } else {
    double length = extended ? 2 : 1;

    double healthPercent = living.getHealthPercent();
    double remainingPercent = 1 - healthPercent;

    // Otherwise set the colour based on the health percent
    healthPart.setMaterial(
      new PhongMaterial(colorForHealthPercent(healthPercent))
    );

    // Set the size and position of the bar based on the health percent
    healthPart.setHeight(healthPercent * length);
    remainingPart.setHeight(remainingPercent * length);

    healthPart.setTranslateY(remainingPercent * length / 2.0);
```

```
      remainingPart.setTranslateY(-healthPercent * length / 2.0);

      // Make the bar visible
      setVisible(true);
    }
  }
}
```

com/mrbbot/civilisation/render/map/RenderTile.java

```
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.render.map.improvement.RenderImprovement;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import com.mrbbot.generic.render.RenderData;
import javafx.geometry.Point2D;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Shape3D;
import javafx.scene.transform.Translate;

import java.util.ArrayList;

/**
 * Render object for a hexagonal tile in the hexagon grid. Handles rendering
 * terrain, improvements, units, selections and city walls.
 */
@ClientOnly
public class RenderTile extends RenderData<Tile> {
  /**
   * Colour to render the hexagonal prism for this tile
   */
  private Color colour;
  /**
   * Render object translated so that z=0 is on the surface of the tile. Holds
   * improvements and units.
   */
  private Render aboveGround;
  /**
   * Render object for the tile's improvement. May not contain any children
   * if the tile doesn't have an improvement.
   */
  private RenderImprovement improvement;
  /**
```

```
 * Render object for the unit selection overlay. Also handles rendering of
 * city boundary walls, and pathfinding routes.
 */
private RenderTileOverlay overlay;
/**
 * Render object for the unit occupying this tile. Hidden if the tile doesn't
 * currently contain a unit.
 */
private RenderUnit unit;
/**
 * Render object for a capital cities health bar. Null by default and only
 * created if there's a capital city on the tile.
 */
private RenderHealthBar cityHealthBar;
/**
 * Height of the hexagonal prism used to represent this tile.
 */
private double height;
/**
 * Tiles that are adjacent to this tile on the map.
 */
private final ArrayList<Tile> adjacentTiles;

/**
 * Constructor for a new render object. This will be used as long as the
 * game is open and isn't destroyed/recreated when a tile update occurs.
 *
 * @param data          tile this render represents
 * @param adjacentTiles tiles adjacent to the tile this render represents
 */
RenderTile(Tile data, ArrayList<Tile> adjacentTiles) {
  super(data);
  this.adjacentTiles = adjacentTiles;

  // Translate this render to the appropriate position on the hex grid
  Point2D center = data.getHexagon().getCenter();
  translateTo(center.getX(), center.getY(), 0);

  // Calculate the colour for the terrain
  colour = data.getTerrain().getColour();

  // Add a hexagonal prism representing the terrain
  height = data.getHeight();
  Shape3D ground = data.getHexagon().getPrism(height);
  ground.getTransforms().add(new Translate(0, height / 2, 0));
```

```
  ground.setMaterial(new PhongMaterial(colour));
  add(ground);

  // Create the render object linked to the top of the terrain
  aboveGround = new Render();
  aboveGround.translateTo(0, 0, height);
  add(aboveGround);

  // Create the improvement render object
  improvement = new RenderImprovement(data, adjacentTiles);
  aboveGround.add(improvement);

  // Create the overlay render object for unit selection/city walls/
  // pathfinding
  overlay = new RenderTileOverlay(Color.WHITE);

  // Create the unit render object
  unit = new RenderUnit(data.unit);
  unit.setVisible(data.unit != null);
  aboveGround.add(overlay, unit);

  // Update the details of all the render objects so they reflect the tile
  // data
  updateRender();
}

void updateRender() {
  // Set the overlay colour depending on whether this tile is selected and
  // traversable by units
  overlay.setColor(
    data.selected
      ? Color.LIGHTBLUE
      : (data.canTraverse() ? Color.WHITE : Color.INDIANRED)
  );

  // Check if there's a city on this tile
  if (data.city != null) {
    // Update the city walls
    overlay.setCityWalls(data.city, data.getCityWalls(), height);

    // If this is a capital city, and the health bar hasn't been added yet,
    // add it
    if (data.city.getCenter().samePositionAs(data)) {
      if (cityHealthBar == null) {
        aboveGround.add(
```

```
          cityHealthBar = new RenderHealthBar(data.city, true)
        );
      }
      // Update the health bar render regardless of whether is was created
      // now
      cityHealthBar.updateRender(data.city);
    }
  }

  // Mark the overlay as selected if it is
  overlay.setSelected(data.selected);

  // Update the unit render object hiding it if there isn't a unit present
  // or changing its colours if there is one
  unit.updateRender(data.unit);
  unit.setVisible(data.unit != null);

  // Update the improvement render object
  improvement.setImprovement(
    data.improvement,
    data.improvementMetadata,
    adjacentTiles
  );
}

/**
 * Marks an overlay as visible. Used when a path dragged out by the user
 * covers this tile.
 *
 * @param visible whether the overlay should always be visible regardless of
 *                its selection state
 */
void setOverlayVisible(boolean visible) {
  overlay.setOverlayVisible(visible);
}

/**
 * Checks if the colour of this tile is translucent. Translucent objects must
 * be added to the render tree last for the translucency effects to work
 *
 * @return whether the tiles terrain is translucent
 */
boolean isTranslucent() {
  return colour.getOpacity() < 1;
```

```
  }
}
```

com/mrbbot/civilisation/render/map/RenderTileOverlay.java

```java
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;

/**
 * Render object representing a tile overlay. Used for highlighting the
 * selected path, and a city's boundaries.
 */
@ClientOnly
class RenderTileOverlay extends Render {
  /**
   * Array of the 6 different parts of the overlay (one for each hexagonal
   * edge)
   */
  private RenderTileOverlayPart[] parts;
  /**
   * The city this overlay covers (may be null)
   */
  private City city;
  /**
   * Array containing information about whether adjacent tiles are part of the
   * same city.
   * <p>
   * [top left, left, bottom left, bottom right, right, top right]
   */
  private boolean[] cityWalls;
  /**
   * Color to render the overlay (depends on unit selection and traversability)
   */
  private Color color;
  /**
   * Whether the overlay should be visible. City walls are always visible.
   */
  private boolean visible;
  /**
   * Whether the tile this overlay represents has been selected
   */
  private boolean selected;
```

```
/**
 * Constructor for a new tile overlay
 *
 * @param color starting colour for the overlay
 */
RenderTileOverlay(Color color) {
  this.color = color;
  // Create an array to hold to parts
  parts = new RenderTileOverlayPart[6];

  // Set default values
  city = null;
  cityWalls = new boolean[]{false, false, false, false, false, false};
  selected = false;

  // Create the part for each hexagonal edge
  for (int i = 0; i < 6; i++) {
    double angle = 30 + (60 * i);

    RenderTileOverlayPart part = new RenderTileOverlayPart(angle, color);
    parts[i] = part;
    add(part);
  }
}

/**
 * Recalculates the visibility of each part from whether or not the overlay's
 * been marked as visible, it's selected, or there are city walls.
 */
private void updateVisibilities() {
  for (int i = 0; i < parts.length; i++) {
    parts[i].setWallVisible(visible || selected || cityWalls[i]);
    // Joins should only be visible for city walls
    parts[i].setJoinVisible(cityWalls[i]);
  }
}

/**
 * Sets the overlay's visibility for pathfinding
 *
 * @param visible whether all overlay parts should be visible
 */
void setOverlayVisible(boolean visible) {
  this.visible = visible;
```

```
  // Recalculate visibilities
  updateVisibilities();
}

/**
 * Sets the overlay's visibility for unit selection
 *
 * @param selected whether all overlay parts should be visible
 */
void setSelected(boolean selected) {
  this.selected = selected;
  // Recalculate visibilities
  updateVisibilities();
}

/**
 * Sets the colour of the overlay if it's been selected or it's part of the
 * path
 *
 * @param color colour of the overlay
 */
void setColor(Color color) {
  this.color = color;
  Color wallColour = city == null ? color : city.wallColour;
  Color joinColour = city == null ? color : city.joinColour;

  // Set the colour for each of the overlay parts
  for (int i = 0; i < parts.length; i++) {
    boolean walled = this.cityWalls[i];
    RenderTileOverlayPart part = parts[i];
    part.setWallColour(walled ? wallColour : color);
    part.setJoinColour(walled ? joinColour : color);
  }
}

/**
 * Sets this overlay's city walls containing information about whether
 * adjacent tiles are part of the same city.
 *
 * @param city       city the overlay is part of
 * @param walls      array containing wall information.
 *                   [top left, left, bottom left,
 *                   bottom right, right, top right]
 * @param tileHeight height of the tile this overlay covers
 */
```

```
  void setCityWalls(City city, boolean[] walls, double tileHeight) {
    this.city = city;
    for (int i = 0; i < parts.length; i++) {
      boolean walled = walls[i];
      double wallHeight = city.greatestTileHeight + 0.2 - tileHeight;
      RenderTileOverlayPart part = parts[i];

      // Update wall state for each part
      part.setWallColour(walled ? city.wallColour : color);
      part.setJoinColour(walled ? city.joinColour : color);

      part.setWallVisible(walled);
      part.setJoinVisible(walled);

      // Set the height of the part to be different depending on whether this
      // is just a selection/path route
      double targetHeight = walled ? wallHeight : 0.1;
      part.setWallHeight(targetHeight, tileHeight, city.greatestTileHeight);
    }
    this.cityWalls = walls;
  }
}
```

com/mrbbot/civilisation/render/map/RenderUnit.java

```
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.logic.unit.UnitType;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import com.mrbbot.generic.render.RenderData;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;
import javafx.scene.transform.Rotate;

/**
 * Render object for a unit. Containing within a {@link RenderTile}. Always
 * added to the tile, but only visible if a unit exists on the tile.
 */
@ClientOnly
class RenderUnit extends RenderData<Unit> {
  /**
   * Constant for the height of a rocket body
```

```
 */
private static final double ROCKET_HEIGHT = 1.2;
/**
 * Constant for the height of a rocket engine
 */
private static final double ROCKET_ENGINE_HEIGHT = 0.2;

/**
 * Array containing the torsos of all the people. Stored so that the colours
 * can be changed when the unit changes. The torso colour is based on the
 * unit type.
 */
private Cylinder[] torsos;
/**
 * Array containing the belts of all the people. Stored so that the colours
 * can be changed when the unit changes. The belt colour is based on the
 * colour of the owning player.
 */
private Cylinder[] belts;
/**
 * Array containing the render objects that wrap all the components of a
 * person. There are 7 people representing each unit. The amount that are
 * shown depends on the health of the unit.
 */
private Render[] people;
/**
 * Render object representing a rocket. Only shown when the unit's type is
 * {@link UnitType#ROCKET}.
 */
private Render rocket;
/**
 * Render object for showing a unit's health.
 */
private RenderHealthBar healthBar;

RenderUnit(Unit data) {
  super(data);

  // Create arrays for render components
  torsos = new Cylinder[7];
  belts = new Cylinder[7];
  people = new Render[7];

  // Create the render objects for the people and rocket
  add(people[0] = buildPerson(0));
```

```
  add(rocket = buildRocket());
  for (int i = 0; i < 6; i++) {
    Render rotor = new Render();
    rotor.add(people[i + 1] = buildPerson(i + 1));
    // Pivot the person around the center
    rotor.translate.setX(0.5);
    rotor.rotateZ.setAngle(60 * i);
    add(rotor);
  }

  // Create and add the health bar
  add(healthBar = new RenderHealthBar(data, false));
}

/**
 * Creates a render object representing a person facing forward
 *
 * @param i index of this person (0 for center, 1 - 6 anticlockwise from
 *          right)
 * @return render object containing components representing a person
 */
@SuppressWarnings("Duplicates")
private Render buildPerson(int i) {
  Render person = new Render();

  // Build legs
  Cylinder leg1 = new Cylinder(0.1, 0.2);
  leg1.setMaterial(new PhongMaterial(Color.LIGHTGOLDENRODYELLOW));
  leg1.setTranslateX(-0.1);
  leg1.setTranslateZ(0.1);
  leg1.setRotationAxis(Rotate.X_AXIS);
  leg1.setRotate(90);
  person.add(leg1);

  Cylinder leg2 = new Cylinder(0.1, 0.2);
  leg2.setMaterial(new PhongMaterial(Color.LIGHTGOLDENRODYELLOW));
  leg2.setTranslateX(0.1);
  leg2.setTranslateZ(0.1);
  leg2.setRotationAxis(Rotate.X_AXIS);
  leg2.setRotate(90);
  person.add(leg2);

  // Build torso
  Cylinder torso = new Cylinder(0.2, 0.4);
  torso.setMaterial(new PhongMaterial(Color.WHITE));
```

```
torso.setTranslateZ(0.2 + 0.2);
torso.setRotationAxis(Rotate.X_AXIS);
torso.setRotate(90);
person.add(torso);
// Store torso so the colour can be changed later
torsos[i] = torso;

// Build belt
Cylinder belt = new Cylinder(0.25, 0.05);
belt.setMaterial(new PhongMaterial(Color.WHITE));
belt.setTranslateZ(0.2 + 0.15);
belt.setRotationAxis(Rotate.X_AXIS);
belt.setRotate(90);
person.add(belt);
// Store belt so the colour can be changed later
belts[i] = belt;

// Build head
Sphere head = new Sphere(0.3);
head.setMaterial(new PhongMaterial(Color.LIGHTGOLDENRODYELLOW));
head.setTranslateZ(0.2 + 0.4 + 0.27);
person.add(head);

// Build eyes
Sphere eye = new Sphere(0.05);
eye.setMaterial(new PhongMaterial(Color.BLACK));
eye.setTranslateZ(0.2 + 0.4 + 0.27);
eye.setTranslateX(0.1);
eye.setTranslateY(0.3);
person.add(eye);

Sphere eye2 = new Sphere(0.05);
eye2.setMaterial(new PhongMaterial(Color.BLACK));
eye2.setTranslateZ(0.2 + 0.4 + 0.27);
eye2.setTranslateX(-0.1);
eye2.setTranslateY(0.3);
person.add(eye2);

// Make the person a bit smaller than it otherwise would be
person.scaleTo(0.5);

// Rotate the person that when it is pivoted, it will still be facing
// forward
double angle = 180;
if (i > 0) angle -= (i - 1) * 60;
```

```
    person.rotateZ.setAngle(angle);

    return person;
  }

  /**
   * Creates a render object representing a rocket
   *
   * @return render object containing components representing a rocket
   */
  private Render buildRocket() {
    Render rocket = new Render();

    // Build rocket engine (bottom bit underneath body)
    Cylinder engine = new Cylinder(0.11, ROCKET_ENGINE_HEIGHT);
    engine.setTranslateZ(ROCKET_ENGINE_HEIGHT / 2);
    engine.setRotationAxis(Rotate.X_AXIS);
    engine.setRotate(90);
    engine.setMaterial(
      new PhongMaterial(UnitType.ROCKET.getColor().brighter())
    );
    rocket.add(engine);

    // Build rocket body
    Cylinder body = new Cylinder(0.22, ROCKET_HEIGHT);
    body.setTranslateZ(ROCKET_HEIGHT / 2 + ROCKET_ENGINE_HEIGHT);
    body.setRotationAxis(Rotate.X_AXIS);
    body.setRotate(90);
    body.setMaterial(new PhongMaterial(UnitType.ROCKET.getColor()));
    rocket.add(body);

    // Build nose cone
    Sphere cone = new Sphere(0.22);
    cone.setTranslateZ(ROCKET_HEIGHT + ROCKET_ENGINE_HEIGHT);
    cone.setMaterial(new PhongMaterial(UnitType.ROCKET.getColor()));
    rocket.add(cone);

    // Hide the rocket by default
    rocket.setVisible(false);

    return rocket;
  }

  /**
   * Update the render for the unit now placed on the tile. Sets the colour and
```

```
   * visibility of various components
   *
   * @param unit unit to take data for the update from
   */
  void updateRender(Unit unit) {
    if (unit != null) {
      // Show the rocket if this is the rocket type
      rocket.setVisible(unit.unitType == UnitType.ROCKET);

      // Build materials for the unit colours
      PhongMaterial torsoMaterial =
        new PhongMaterial(unit.unitType.getColor());
      PhongMaterial beltMaterial =
        new PhongMaterial(unit.player.getColour());

      // Show a proportionate amount of people for the health
      double healthPercent = unit.getHealthPercent();
      double onePersonProportion = 1.0 / (double) people.length;
      for (int i = 0; i < people.length; i++) {
        // Update the colours
        torsos[i].setMaterial(torsoMaterial);
        belts[i].setMaterial(beltMaterial);

        // Update the visibility for the health
        people[i].setVisible(healthPercent >= i * onePersonProportion);
      }
    }

    // Update the health bar render with new information
    healthBar.updateRender(unit);
  }
}
```

com/mrbbot/civilisation/render/map/RenderTileOverlayPart.java

```
package com.mrbbot.civilisation.render.map;

import com.mrbbot.civilisation.geometry.Hexagon;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.PointLight;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.transform.Rotate;
```

```
/**
 * Render object for one part of the tile overlay. One part represents one edge
 * of a hexagon.
 */
@ClientOnly
class RenderTileOverlayPart extends Render {
  /**
   * The wall part. Used for selections/path highlighting and city walls.
   */
  private Box wall;
  /**
   * The join between segments of walls in a city.
   */
  private Cylinder join;

  /**
   * Creates a new tile part
   *
   * @param angle  pivot angle for the part
   * @param colour initial colour of the part
   */
  RenderTileOverlayPart(double angle, Color colour) {
    super();

    // Create and pivot the wall
    Render wallHolder = new Render();
    wallHolder.rotateZ.setAngle(angle);
    wallHolder.translate.setY(Hexagon.SQRT_3 / 2);
    wall = new Box(1, 0.2, 0.1);
    wall.setMaterial(new PhongMaterial(colour));
    wall.setVisible(false);
    wall.setTranslateY(-0.1);
    wall.setTranslateZ(0.05);
    wallHolder.add(wall);

    // Create and pivot the join
    Render joinHolder = new Render();
    joinHolder.rotateZ.setAngle(angle + 30);
    joinHolder.translate.setY(1);
    join = new Cylinder(0.2, 0.1);
    join.setMaterial(new PhongMaterial(colour));
    join.setRotationAxis(Rotate.X_AXIS);
    join.setRotate(90);
    join.setTranslateZ(0.05);
```

```
    join.setVisible(false);

    joinHolder.add(join);

    add(wallHolder);
    add(joinHolder);
  }

  /**
   * Sets the visibility of the wall component of this part
   *
   * @param visible whether the wall should be visible
   */
  void setWallVisible(boolean visible) {
    wall.setVisible(visible);
  }

  /**
   * Sets the visibility of the join component of this part
   *
   * @param visible whether the join should be visible
   */
  void setJoinVisible(boolean visible) {
    join.setVisible(visible);
  }

  /**
   * Sets the colour of the wall component of this part
   *
   * @param colour new colour for the wall component
   */
  void setWallColour(Color colour) {
    wall.setMaterial(new PhongMaterial(colour));
  }

  /**
   * Sets the colour of the join component of this part
   *
   * @param colour new colour for the join component
   */
  void setJoinColour(Color colour) {
    join.setMaterial(new PhongMaterial(colour));
  }

  /**
```

```
   * Sets height of the wall/join components
   *
   * @param height             new height of the wall
   * @param tileHeight         height of the tile this overlay represents
   * @param greatestTileHeight greatest tile height of all tiles in the
   *                           containing city
   */
  void setWallHeight(
    double height,
    double tileHeight,
    double greatestTileHeight
  ) {
    wall.setDepth(height);
    wall.setTranslateZ(height / 2);

    // Update the join height
    updateJoinHeight(tileHeight, greatestTileHeight);
  }

  /**
   * Updates the join height so that it reaches the bottom of the ground
   *
   * @param tileHeight         height of the tile this overlay represents
   * @param greatestTileHeight greatest tile height of all tiles in the
   *                           containing city
   */
  private void updateJoinHeight(double tileHeight, double greatestTileHeight) {
    // Height of this join
    double joinHeight = greatestTileHeight + 0.4;

    join.setHeight(joinHeight);
    // Translate the join so that it has the same height and vertical position
    // for each tile in the containing city
    join.setTranslateZ(-tileHeight + (joinHeight / 2));
  }
}
```

com/mrbbot/civilisation/ui/connect/ClientCreator.java

```
package com.mrbbot.civilisation.ui.connect;

import java.io.IOException;

/**
 * Function called when the user requests a connection be made to the server
 */
```

```
public interface ClientCreator {
  /**
   * Create client callback
   *
   * @param host server host IP/URL
   * @param port server port number
   * @param id   desired id of the player
   * @throws IOException if a connection cannot be established
   */
  void createClient(String host, int port, String id) throws IOException;
}
```

com/mrbbot/civilisation/ui/connect/ScreenConnect.java

```
package com.mrbbot.civilisation.ui.connect;

import com.mrbbot.civilisation.logic.map.MapSize;
import com.mrbbot.civilisation.net.CivilisationServer;
import com.mrbbot.civilisation.ui.Screen;
import com.mrbbot.civilisation.ui.UIHelpers;
import com.mrbbot.generic.net.ClientOnly;
import javafx.application.Platform;
import javafx.beans.value.ChangeListener;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * Screen for letting a user join a game, create a new game, or load an
```

```
 * existing game. The first screen the user lands on when the game starts.
 */
@ClientOnly
public class ScreenConnect extends Screen {
  /**
   * Enum representing the choices the user has on this screen. These are
   * translated to radio buttons to allow the user to select each one. When
   * clicked they en/disable various UI components that are(n't) needed.
   */
  private enum Choice {
    JOIN("Join a Game"),
    HOST("Create and Host a New Game"),
    LOAD("Load and Host a Saved Game");

    /**
     * Description of this choice. To be displayed on the radio button for this
     * choice.
     */
    private String description;

    Choice(String description) {
      this.description = description;
    }
  }

  /**
   * Class representing a game save file to be shown in the save list
   */
  private class GameSave {
    /**
     * Path to the game save file, should end with .yml.
     */
    private String filePath;
    /**
     * Name of the game, loaded from the file
     */
    private String gameName;

    private GameSave(String filePath, String gameName) {
      this.filePath = filePath;
      this.gameName = gameName;
    }
  }

  /**
```

```
     * Function to be called when the user requests a connection to a server
     */
    private final ClientCreator clientCreator;
    /**
     * Function to be called when the user requests a server be created
     */
    private final ServerCreator serverCreator;
    /**
     * Array containing all the game saves in the "saves" directory
     */
    private GameSave[] saves;

    /**
     * User's selected choice. Determines what UI elements to enable and how to
     * handle the Join/Host button click.
     */
    private Choice choice = Choice.JOIN;
    /**
     * User's selected map size. Used when creating a new game.
     */
    private MapSize selectedMapSize = MapSize.STANDARD;
    /**
     * List containing names of game saves. Observable so changes made to it can
     * be reflected in the combo box for names.
     */
    private ObservableList<String> nameList;
    /**
     * Combo box for name that lists all the values in nameList.
     */
    private ComboBox<String> nameBox;
    /**
     * Array containing all radio buttons for controlling map size. Stored so
     * they can be en/disabled when the user's choice changes.
     */
    private RadioButton[] sizeRadioButtons;
    /**
     * Text field for the host name of the server to connect too.
     */
    private TextField hostField;
    /**
     * Text field for the port number of the server to connect too. Should only
     * accept numeric values.
     */
    private TextField portField;
    /**
```

```
 * Text field for the user's player ID when joining a server. Controls unit/
 * cities owners, panel border colours, etc.
 */
private TextField idField;
/**
 * Button that joins/hosts a game depending on the user's choice. Should
 * only be enabled if all the required UI components have sensible data.
 */
private Button joinButton;

/**
 * Pane containing UI elements for choices, host, sizes, id, port, and other
 * connection details. Should be hidden when the user clicks the join button.
 */
private GridPane pane;
/**
 * Loading indicator shown when the user clicks the join button to indicate
 * that something is happening.
 */
private ProgressIndicator progressIndicator;

/**
 * Constructor for a new connection screen
 *
 * @param clientCreator callback function for creating a client
 * @param serverCreator callback function for creating a server
 */
public ScreenConnect(
  ClientCreator clientCreator,
  ServerCreator serverCreator
) {
  this.clientCreator = clientCreator;
  this.serverCreator = serverCreator;

  // Load all available game saves
  try {
    // Get a reference to the saves directory
    // ("current working directory/saves")
    String savesDirectoryPath =
      System.getProperty("user.dir") + File.separator + "saves";
    File savesDirectory = new File(savesDirectoryPath);

    // Check if the folder exists, otherwise make it
    if (!savesDirectory.exists()) {
      boolean made = savesDirectory.mkdir();
```

```
      if (!made) throw new IOException("unable to create saves directory");
    }

    // Load the list of GameSave objects
    saves = Files.list(Paths.get(savesDirectoryPath))
      // Convert path objects to their absolute file path
      .map(Path::toString)
      // We only want files ending with .yml
      .filter(path -> path.endsWith(".yml"))
      .map(path -> {
        // Load the game save as we normally would to extract the name
        String name = "Unknown";
        try (FileReader reader = new FileReader(path)) {
          //noinspection unchecked
          Map<String, Object> map =
            CivilisationServer.YAML.loadAs(reader, Map.class);
          name = (String) map.get("name");
        } catch (IOException e) {
          e.printStackTrace();
        }
        // Return a new game save object with the required data
        return new GameSave(path, name);
      })
      // Convert the stream to an array
      .toArray(GameSave[]::new);
  } catch (IOException e) {
    e.printStackTrace();
  }
}

/**
 * Set the loading state. Shows/hides the details pane/loading indicator.
 *
 * @param loading whether to show the loading indicator
 */
private void setLoading(boolean loading) {
  // Hide the pane if we're loading. Do this with setOpacity not setVisible
  // so the pane is still used in layout calculations (so the wrapping title
  // pane doesn't change size).
  pane.setOpacity(loading ? 0 : 1);
  // We can just use setVisible for the loading indicator. It's smaller than
  // the pane.
  progressIndicator.setVisible(loading);
}
```

```
/**
 * Set all the size radio buttons disabled state
 *
 * @param disable whether all the buttons should be disabled
 */
private void setSizeRadioButtonsDisable(boolean disable) {
  for (RadioButton sizeRadioButton : sizeRadioButtons) {
    sizeRadioButton.setDisable(disable);
  }
}

/**
 * Calculates whether the join button should be enabled from the state of the
 * other UI components. Which UI components to check depends on the user's
 * choice.
 */
private void checkJoinButtonEnabled() {
  boolean enabled = false;
  switch (choice) {
    case JOIN:
      // If we're joining a game, we need a host name, port number, and
      // player ID
      enabled = !hostField.getText().isEmpty()
        && !portField.getText().isEmpty()
        && !idField.getText().isEmpty();
      break;
    case HOST:
      // If we're hosting a game, we need a new game name, port number, and
      // player ID. We also need a map size, but this will always be set.
      enabled = !nameBox.getEditor().getText().isEmpty()
        && !portField.getText().isEmpty()
        && !idField.getText().isEmpty();
      break;
    case LOAD:
      // If we're loading an existing game, we need an existing game name, a
      // port number, and player ID.
      enabled = nameBox.getValue() != null
        && !nameBox.getValue().isEmpty()
        && !portField.getText().isEmpty()
        && !idField.getText().isEmpty();
      break;
  }
  // Set the enabled state
  joinButton.setDisable(!enabled);
}
```

```java
/**
 * En/disables the required UI components for the user's new choice
 * selection.
 *
 * @param choice new selected choice
 */
private void resetForChoice(Choice choice) {
  // Store the choice selection
  this.choice = choice;
  switch (choice) {
    case JOIN:
      // If we're joining a game, we need a host name, port number, and
      // player ID
      nameList.clear();
      nameBox.setDisable(true);
      nameBox.setEditable(false);
      // Disable map size selection buttons
      setSizeRadioButtonsDisable(true);
      hostField.setDisable(false);
      joinButton.setText("Join");
      break;
    case HOST:
      // If we're hosting a game, we need a new game name, map size, port
      // number, and player ID.
      nameList.clear();
      nameBox.setDisable(false);
      nameBox.setEditable(true);
      // Enable map size selection buttons
      setSizeRadioButtonsDisable(false);
      hostField.setDisable(true);
      joinButton.setText("Host and Join");
      break;
    case LOAD:
      // If we're loading an existing game, we need an existing game name, a
      // port number, and player ID.
      nameList.clear();
      // Get existing game names
      nameList.addAll(
        Arrays.stream(saves)
          .map(save -> save.gameName)
          .collect(Collectors.toList())
      );
      nameBox.setDisable(false);
      nameBox.setEditable(false);
```

```
      if (nameList.size() > 0) nameBox.setValue(nameList.get(0));
      // Disable map size selection buttons
      setSizeRadioButtonsDisable(true);
      hostField.setDisable(true);
      joinButton.setText("Host and Join");
      break;
  }
  checkJoinButtonEnabled();
}

/**
 * Called on click of the join button.
 */
private void launch() {
  // Show the loading indicator
  setLoading(true);
  // Start a new thread for launching the client/server. This should be
  // done in a separate thread so the UI thread isn't blocked. This would
  // cause the loading spinner animation not to work.
  Thread bootstrapThread = new Thread(() -> {
    try {
      // Get the port number and player ID as these are required for all
      // choices
      int port = Integer.parseInt(portField.getText());
      String id = idField.getText();

      switch (choice) {
        case JOIN:
          // If we're joining a game, get the host name and connect to it
          String host = hostField.getText();
          clientCreator.createClient(host, port, id);
          break;
        case HOST:
          // If we're hosting a game, get the new game name
          String newGameName = nameBox.getEditor().getText();
          // Make a file name for this game name. (lower case,
          // spaces -> underscores, + .yml)
          String newGameFileName = "saves" + File.separator + newGameName
            .toLowerCase()
            .replaceAll(" ", "_")
            + ".yml";
          // Create the server and then immediately connect to it as if it
          // was over the network. Even though we running these in the same
          // program instance and could exchange data more efficiently, this
          // reduces the need to write duplicate code for exchanging data
```

```
        // between the client and server.
        serverCreator.createServer(
          newGameFileName,
          newGameName,
          selectedMapSize,
          port
        );
        // Use the local loopback address as the host name (this computer)
        clientCreator.createClient("127.0.0.1", port, id);
        break;
      case LOAD:
        // Get the existing game name
        String loadGameName = nameBox.getValue();
        // Try and find the game save with that name, we should be able to
        // because these names come from the list of game saves
        GameSave loadGameSave = null;
        for (GameSave gameSave : saves) {
          if (gameSave.gameName.equals(loadGameName)) {
            loadGameSave = gameSave;
            break;
          }
        }
        // Check we did find a save
        assert loadGameSave != null;
        // Create the server and then immediately connect to it as if it
        // was over the network. See above comment for more details. We
        // pass null as the game name here to signify that we want to load
        // the game save. This makes the passed map size irrelevant.
        serverCreator.createServer(
          loadGameSave.filePath,
          null,
          MapSize.STANDARD,
          port
        );
        // Use the local loopback address as the host name (this computer)
        clientCreator.createClient("127.0.0.1", port, id);
        break;
    }

  } catch (IOException e) {
    // If there was an error, show a dialog on the UI thread stating such
    Platform.runLater(() -> {
      setLoading(false);
      UIHelpers.showDialog(e.getMessage(), true);
    });
```

```
      e.printStackTrace();
    }
  });
  bootstrapThread.setName("Bootstrap");
  bootstrapThread.start();
}

/**
 * Function to check whether text only contains digits
 *
 * @param text text to check against
 * @return whether the text only contains digits
 */
private boolean isDigits(String text) {
  for (char c : text.toCharArray()) {
    int code = (int) c;
    // Check every ASCII code is between 0 and 9.
    if (code < 48 || code > 57) return false;
  }
  return true;
}

/**
 * Wraps a node with a titled border
 *
 * @param title title for the border
 * @param child node to wrap
 * @return titled pane containing the child
 */
private TitledPane makeTitledPane(String title, Node child) {
  TitledPane titledPane = new TitledPane(title, child);
  // Specify a desired size for the child (it would fill the screen
  // otherwise)
  titledPane.setMaxSize(300, 0);
  // Titled panes are collapsible by default which is something we don't
  // really want
  titledPane.setCollapsible(false);
  return titledPane;
}

/**
 * Creates a scene representing this screen
 *
 * @param stage  stage the scene would be placed in
 * @param width  width of the screen
```

```
 * @param height height of the screen
 * @return scene representing this screen
 */
@SuppressWarnings("Duplicates")
@Override
public Scene makeScene(Stage stage, int width, int height) {
  // Create a change listener that will be used in all UI components to
  // check whether the join button should be enabled when data changes.
  ChangeListener<String> changeListener =
    (observable, oldValue, newValue) -> checkJoinButtonEnabled();

  pane = new GridPane();
  pane.setHgap(10);
  pane.setVgap(10);

  // Create choice radio buttons, adding them to a toggle group so only one
  // choice can be selected at once.
  ToggleGroup choiceToggleGroup = new ToggleGroup();
  Choice[] choices = Choice.values();
  for (int i = 0; i < choices.length; i++) {
    final Choice choice = choices[i];
    RadioButton choiceRadioButton = new RadioButton(choice.description);
    choiceRadioButton.setToggleGroup(choiceToggleGroup);
    // Reset other UI components on button selection change
    choiceRadioButton.setOnAction(e -> resetForChoice(choice));
    // Add the button to the pane filling all available width
    pane.add(choiceRadioButton, 0, i, 4, 1);
    // Set a default selection
    if (choice == Choice.JOIN) choiceRadioButton.setSelected(true);
  }

  // Create labels
  Label nameLabel = new Label("Name");
  Label hostLabel = new Label("Host");
  Label portLabel = new Label("Port");
  Label idLabel = new Label("ID");
  nameLabel.setPrefWidth(80);
  hostLabel.setPrefWidth(80);
  portLabel.setPrefWidth(80);
  idLabel.setPrefWidth(80);

  // Create the game name box, this will act like a text field when creating
  // a new game, and a combo box when selecting a game to load
  nameList = FXCollections.observableArrayList();
  nameBox = new ComboBox<>(nameList);
```

```
nameBox.setPrefWidth(300);
nameBox.setEditable(true);
nameBox.valueProperty().addListener(changeListener);
// Watch for changes
nameBox.getEditor().textProperty().addListener(changeListener);

// Create the size radio buttons row
HBox sizeBox = new HBox(10);
ToggleGroup sizeToggleGroup = new ToggleGroup();
MapSize[] mapSizes = MapSize.values();
sizeRadioButtons = new RadioButton[mapSizes.length];
for (int i = 0; i < mapSizes.length; i++) {
  final MapSize mapSize = mapSizes[i];
  RadioButton sizeRadioButton = new RadioButton(mapSize.name);
  sizeRadioButton.setToggleGroup(sizeToggleGroup);
  // Store the new selected state
  sizeRadioButton.setOnAction(e -> selectedMapSize = mapSize);
  // Set a default selection
  if (mapSize == MapSize.STANDARD) sizeRadioButton.setSelected(true);
  sizeRadioButtons[i] = sizeRadioButton;
  sizeBox.getChildren().add(sizeRadioButton);
}

// Create text fields
hostField = new TextField("127.0.0.1");
portField = new TextField("1234");
idField = new TextField();

// Make sure that the port field only allows digits to be inputted
portField.setTextFormatter(
  // Returning null discards the change
  new TextFormatter<>(change -> isDigits(change.getText()) ? change : null)
);

// Register text change listeners
hostField.textProperty().addListener(changeListener);
portField.textProperty().addListener(changeListener);
idField.textProperty().addListener(changeListener);

// Create the join button that launches the game when clicked
joinButton = new Button("Join");
joinButton.setPrefWidth(300);
joinButton.setOnAction(e -> this.launch());

// Add UI components to the pane (argument order: node, column index, row
```

```
// index, [column span, row span]) [spans are optional]
pane.add(
  nameLabel,
  0, choices.length + 1,
  1, 1
);
pane.add(
  nameBox,
  1, choices.length + 1,
  3, 1
);

pane.add(
  sizeBox,
  0, choices.length + 2,
  4, 1
);

pane.add(hostLabel, 0, choices.length + 4);
pane.add(hostField, 1, choices.length + 4);
pane.add(portLabel, 2, choices.length + 4);
pane.add(portField, 3, choices.length + 4);

pane.add(idLabel, 0, choices.length + 5);
pane.add(
  idField,
  1, choices.length + 5,
  3, 1
);

pane.add(
  joinButton,
  0, choices.length + 7,
  4, 1
);

// En/disable UI components for the default choice
resetForChoice(Choice.JOIN);

// Create the size loading indicator and hide it by default
progressIndicator = new ProgressIndicator();
progressIndicator.setMaxSize(100, 100);
setLoading(false);

// Create the layered layout
```

```
    StackPane layers = new StackPane(pane, progressIndicator);
    layers.setAlignment(Pos.CENTER);

    // Title the pane
    StackPane root = new StackPane(makeTitledPane("Game", layers));
    root.setAlignment(Pos.CENTER);

    // Create the scene
    return new Scene(root, width, height);
  }
}
```

com/mrbbot/civilisation/ui/connect/ServerCreator.java

```
package com.mrbbot.civilisation.ui.connect;

import com.mrbbot.civilisation.logic.map.MapSize;

import java.io.IOException;

/**
 * Function called when the user requests that a server be started
 */
public interface ServerCreator {
  /**
   * Create server callback
   *
   * @param gameFilePath file path of the game save file (may or may not exist)
   * @param gameName     name of the game (if this is null, we're loading an
   *                     existing game from a file)
   * @param mapSize      desired map size of the new game (ignored if loading
   *                     from a file)
   * @param port         port number to run the server on
   * @throws IOException if the server cannot be created (e.g. port already
   *                     bound)
   */
  void createServer(
    String gameFilePath,
    String gameName,
    MapSize mapSize,
    int port
  ) throws IOException;
}
```

com/mrbbot/civilisation/ui/game/Badge.java

```java
package com.mrbbot.civilisation.ui.game;

import javafx.geometry.Pos;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.scene.text.Font;

/**
 * Pane showing a "badge", a coloured circle with text on top.
 */
class Badge extends StackPane {
  /**
   * Size of the badge's coloured circle
   */
  private static final int BADGE_SIZE = 16;

  /**
   * Creates a new badge
   *
   * @param badgeType type of the badge containing information on the colour
   *                  and text
   */
  Badge(BadgeType badgeType) {
    super();
    setAlignment(Pos.CENTER);

    // Create the coloured circle
    Canvas canvas = new Canvas(BADGE_SIZE, BADGE_SIZE);
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(badgeType.color);
    g.fillOval(0, 0, BADGE_SIZE, BADGE_SIZE);
    g.fill();

    // Create the text label
    Label label = new Label(badgeType.text);
    label.setTextFill(badgeType.textColor);
    label.setFont(new Font(10));

    // Stack them on top of each other
    getChildren().addAll(canvas, label);
  }
}
```

com/mrbbot/civilisation/ui/game/BadgeType.java

```java
package com.mrbbot.civilisation.ui.game;

import javafx.scene.paint.Color;

/**
 * Enum representing the different types of badges. Badges are just a coloured
 * circle with some text on top.
 */
public enum BadgeType {
  SCIENCE(Color.DEEPSKYBLUE, "S"),
  GOLD(Color.GOLD, "£"),
  PRODUCTION(Color.ORANGE, "P"),
  FOOD(Color.GREEN, "@"),
  HEALTH(Color.PINK, "H"),
  MOVEMENT(Color.LIMEGREEN, "M"),
  ATTACK(Color.CRIMSON, "!");

  /**
   * Colour of the circle
   */
  Color color;
  /**
   * Colour of the text on the circle
   */
  Color textColor;
  /**
   * Text to be displayed on the circle
   */
  String text;

  BadgeType(Color color, String text) {
    this.color = color;
    this.textColor = color.darker();
    this.text = text;
  }
}
```

com/mrbbot/civilisation/ui/game/ScreenGame.java

```java
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.net.packet.PacketChat;
import com.mrbbot.civilisation.net.packet.PacketReady;
import com.mrbbot.civilisation.render.RenderCivilisation;
```

```java
import com.mrbbot.civilisation.render.map.RenderGame;
import com.mrbbot.civilisation.ui.Screen;
import com.mrbbot.civilisation.ui.UIHelpers;
import com.mrbbot.generic.net.ClientOnly;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

/**
 * Main screen for the game. Contains the game render and UI overlays.
 */
@ClientOnly
public class ScreenGame extends Screen {
  /**
   * Game that should be rendered in this screen
   */
  private final Game game;
  /**
   * ID of the current game player
   */
  private final String id;
  /**
   * Root render object
   */
  public RenderCivilisation renderCivilisation;
  /**
   * Panel containing UI overlays for player stats, research progress, unit
   * selection details, city production list, etc
   */
  private UIGame ui;

  /**
   * Creates a new game screen
   *
   * @param game game that should be rendered by the screen
   * @param id   ID of the player who's currently playing the game
   */
  public ScreenGame(Game game, String id) {
    // Store the values so they can be used later
    this.game = game;
    this.id = id;
  }
```

```java
/**
 * Creates a scene representing this screen
 *
 * @param stage  stage the scene would be placed in
 * @param width  width of the screen
 * @param height height of the screen
 * @return scene representing this screen
 */
@Override
public Scene makeScene(Stage stage, int width, int height) {
  StackPane pane = new StackPane();
  pane.setAlignment(Pos.CENTER);

  // Create a render object for the game
  RenderGame renderGame = new RenderGame(
    game,
    id,
    // Register unit and city selection listeners
    (unit) -> ui.onSelectedUnitChanged(game, unit),
    (city) -> ui.onSelectedCityChanged(
      game,
      city,
      game.getPlayersCitiesById(id)
    )
  );
  // Create the root render object allowing for zooming, panning, and
  // lighting
  this.renderCivilisation = new RenderCivilisation(
    renderGame,
    width,
    height
  );

  // Create the game UI
  ui = new UIGame(renderGame, height);
  ui.setPrefSize(width, height);

  // Register game state listeners so changes can be reflected in the UI
  game.setCurrentPlayer(
    id,
    (stats) -> ui.onPlayerStatsChanged(stats)
  );
  game.setTechDetailsListener(
    (details) -> ui.onTechDetailsChanged(game, details)
  );
```

```
    // Show a dialog on new messages (research unlocks, errors, etc)
    game.setMessageListener(UIHelpers::showDialog);

    // Add the 3D render's sub-scene to the pane
    pane.getChildren().addAll(this.renderCivilisation.subScene, ui);

    // Create a new scene
    Scene scene = new Scene(pane, width, height);
    // Register a CSS stylesheet for styling some of the UI panels (mostly
    // the city production list)
    scene.getStylesheets().add("/com/mrbbot/civilisation/ui/game/styles.css");
    // Set the scene of the render, registering keyboard shortcuts
    this.renderCivilisation.setScene(scene, e -> {
      if (e.getCode() == KeyCode.F11) {
        stage.setFullScreen(!stage.isFullScreen());
      }
    });
    return scene;
  }

  /**
   * Forwards a chat packet to the UI, so it can be displayed
   *
   * @param packet packet to forward
   */
  public void handlePacketChat(PacketChat packet) {
    ui.handlePacketChat(packet);
  }

  /**
   * Forwards a ready packet to the UI, so the next turn button can be
   * re-enabled
   *
   * @param packet packet to forward
   */
  public void handlePacketReady(PacketReady packet) {
    ui.handlePacketReady(packet);
  }
}
```

com/mrbbot/civilisation/ui/game/UIGame.java

```
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.logic.Player;
```

```java
import com.mrbbot.civilisation.logic.PlayerStats;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.map.tile.Improvement;
import com.mrbbot.civilisation.logic.techs.PlayerTechDetails;
import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.logic.unit.UnitAbility;
import com.mrbbot.civilisation.net.packet.*;
import com.mrbbot.civilisation.render.map.RenderGame;
import com.mrbbot.generic.net.ClientOnly;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;

import java.util.ArrayList;

/**
 * UI panel containing all other UI panels for the game (stats, techs, etc).
 * Extends anchor pane allowing items to be positioned relative to the edges of
 * the screen.
 */
@ClientOnly
public class UIGame extends AnchorPane {
  /**
   * Default amount of padding for UI panels
   */
  private static final Insets PANEL_PADDING = new Insets(10);

  /**
   * Game render object reference for performing actions on the game state.
   */
  private final RenderGame renderGame;
  /**
   * Colour of the current player. Used for panel borders.
   */
  private final Color playerColor;

  /**
   * Panel for technology stats (current research and progress)
   */
  private final UIPanelTech panelTech;
  /**
   * Panel for chat messages and for sending new ones
```

```
 */
private final UIPanelChat panelChat;
/**
 * Panel for performing actions with units and displaying information on the
 * current selection
 */
private final UIPanelActions panelActions;
/**
 * Panel for current player statistics (gold/science per turn)
 */
private final UIPanelStats panelStats;
/**
 * UI of the technology tree showing techs available for research and
 * previously researched items.
 */
private final UITechTree techTree;
/**
 * UI panel for showing city details and for choosing what to build in a
 * city.
 */
private final UIPanelCityDetails panelCityDetails;
/**
 * Button that when clicked closes the tech tree. The open button is in
 * {@link UIPanelTech}.
 */
private final Button closeTechTreeButton;

/**
 * Creates a new game UI instance
 *
 * @param renderGame reference to the current game render object
 * @param height     height of the screen this is displayed in
 */
public UIGame(RenderGame renderGame, int height) {
  this.renderGame = renderGame;
  // Stop preventing the mouse from reaching the 3D render object that would
  // prevent panning, zooming, and unit selection.
  setPickOnBounds(false);

  Player player = this.renderGame.currentPlayer;
  playerColor = player.getColour();

  // Create the tech panel
  panelTech = new UIPanelTech();
  panelTech.setBorder(makePanelBorder(Pos.BOTTOM_RIGHT));
```

```
panelTech.setBackground(makePanelBackground(Pos.BOTTOM_RIGHT));
panelTech.setPadding(PANEL_PADDING);
// Open the tech tree on clicking the open button
panelTech.setOnOpenTechTree(e -> setTechTreeVisible(true));
// Position it in the top left of the screen
AnchorPane.setTopAnchor(panelTech, 0.0);
AnchorPane.setLeftAnchor(panelTech, 0.0);

// Create the chat panel
panelChat = new UIPanelChat(renderGame.currentPlayer.id);
panelChat.setBorder(makePanelBorder(Pos.BOTTOM_LEFT));
panelChat.setBackground(makePanelBackground(Pos.BOTTOM_LEFT));
panelChat.setPadding(PANEL_PADDING);
// Position it in the top right of the screen
AnchorPane.setTopAnchor(panelChat, 0.0);
AnchorPane.setRightAnchor(panelChat, 0.0);

// Create the unit actions panel
panelActions = new UIPanelActions();
panelActions.setBorder(makePanelBorder(Pos.TOP_LEFT));
panelActions.setBackground(makePanelBackground(Pos.TOP_LEFT));
panelActions.setPadding(PANEL_PADDING);
// Register the listener for when the user requests a unit take an action
// or clicks on the next turn button
panelActions.setUnitActionListener(this::onUnitAction);
// Position it in the bottom right of the screen
AnchorPane.setBottomAnchor(panelActions, 0.0);
AnchorPane.setRightAnchor(panelActions, 0.0);

// Create the player stats panel
panelStats = new UIPanelStats();
panelStats.setBorder(makePanelBorder(Pos.TOP_RIGHT));
panelStats.setBackground(makePanelBackground(Pos.TOP_RIGHT));
panelStats.setPadding(PANEL_PADDING);
// Position it in the bottom left of the screen
AnchorPane.setBottomAnchor(panelStats, 0.0);
AnchorPane.setLeftAnchor(panelStats, 0.0);

// Create the city details panel
panelCityDetails = new UIPanelCityDetails(renderGame);
panelCityDetails.setBorder(new Border(new BorderStroke(
  playerColor,
  BorderStrokeStyle.SOLID,
  CornerRadii.EMPTY,
  // Left border only
```

```
    new BorderWidths(0, 0, 0, 10)
)));
panelCityDetails.setBackground(makePanelBackground(Pos.CENTER));
panelCityDetails.setVisible(false);
// Position it to the right of the screen taking up the full screen height
AnchorPane.setTopAnchor(panelCityDetails, 0.0);
AnchorPane.setRightAnchor(panelCityDetails, 0.0);
AnchorPane.setBottomAnchor(panelCityDetails, 0.0);

// Create initial player technology details for initialising the tech tree
PlayerTechDetails techDetails = new PlayerTechDetails(
    renderGame.data.getPlayerUnlockedTechs(player.id),
    renderGame.data.getPlayerUnlockingTech(player.id),
    renderGame.data.getPlayerUnlockingProgress(player.id)
);
// Create the tech tree UI
techTree = new UITechTree(
    renderGame.data,
    player.id,
    techDetails,
    height
);
techTree.setBorder(makePanelBorder(Pos.CENTER));
// Fill the screen with the tech tree when it's visible
AnchorPane.setTopAnchor(techTree, 0.0);
AnchorPane.setLeftAnchor(techTree, 0.0);
AnchorPane.setBottomAnchor(techTree, 0.0);
AnchorPane.setRightAnchor(techTree, 0.0);
// Set the initial tech details
panelTech.setTechDetails(techDetails);

// Create the close button
closeTechTreeButton = new Button("Close Tech Tree");
closeTechTreeButton.setOnAction(e -> setTechTreeVisible(false));
// Position it in the top left of the screen
AnchorPane.setTopAnchor(closeTechTreeButton, 20.0);
AnchorPane.setLeftAnchor(closeTechTreeButton, 20.0);

// Hide the tech tree initially
setTechTreeVisible(false);

// Add all the panels to the screen
getChildren().addAll(
    panelTech,
    panelChat,
```

```
    panelActions,
    panelStats,
    panelCityDetails,
    techTree,
    closeTechTreeButton
  );
}

/**
 * Create a corner radii object with the radius in the corner specified
 *
 * @param cutout corner for the cutout
 * @param size   size of the cutout
 * @return corner radii object with details on the corner cutout
 */
private CornerRadii makeCornerRadiiForCutout(Pos cutout, int size) {
  return new CornerRadii(
    cutout == Pos.TOP_LEFT ? size : 0,
    cutout == Pos.TOP_RIGHT ? size : 0,
    cutout == Pos.BOTTOM_RIGHT ? size : 0,
    cutout == Pos.BOTTOM_LEFT ? size : 0,
    false
  );
}

/**
 * Makes a border object with a cutout in the specified corner
 *
 * @param cutout corner for the cutout
 * @return border object with a cutout
 */
private Border makePanelBorder(Pos cutout) {
  return new Border(new BorderStroke(
    playerColor,
    BorderStrokeStyle.SOLID,
    makeCornerRadiiForCutout(cutout, 10),
    new BorderWidths(10)
  ));
}

/**
 * Makes a solid white background with a cutout in the specified corner
 *
 * @param cutout corner for the cutout
 * @return background object with a cutout
```

```
 */
private Background makePanelBackground(Pos cutout) {
  return new Background(new BackgroundFill(
    Color.WHITE,
    makeCornerRadiiForCutout(cutout, 20),
    null
  ));
}

/**
 * Callback function called when the selected unit changes in the game
 *
 * @param game game containing the unit
 * @param unit selected unit, may be null if no unit is selected
 */
void onSelectedUnitChanged(Game game, Unit unit) {
  panelActions.setSelectedUnit(game, unit);
}

/**
 * Callback function called when the selected city changes in the game
 *
 * @param game          game containing the city
 * @param city          selected city, may be null if no unit is selected
 * @param playersCities all of the players cities in the game
 */
void onSelectedCityChanged(
  Game game,
  City city,
  ArrayList<City> playersCities
) {
  // Update the city details panel to reflect the change if required
  if (city != null)
    panelCityDetails.setSelectedCity(game, city, playersCities);
  // Show/hide the panel depending on if a city has been selected or not
  panelCityDetails.setVisible(city != null);
}

/**
 * Callback function called when the player requests a unit perform an action
 * or clicks the the next turn button.
 *
 * @param unit          unit the action should be performed on or null if the
 *                      next turn button was pressed
 * @param actionDetails string containing additional details about the action
```

```
 *                      (i.e. what improvement a worker should construct)
 */
private void onUnitAction(Unit unit, String actionDetails) {
  // Check if this was the next turn button
  if (unit == null) {
    System.out.println("Next turn...");
    // Mark the client as waiting for other players, so it can't perform any
    // more actions
    renderGame.data.waitingForPlayers = true;
    // Deselect any units/cities
    renderGame.setSelectedUnit(null);
    renderGame.setSelectedCity(null);
  } else {
    // Otherwise, an action is to be performed
    System.out.printf(
      "%s performed an action (details \"%s\")\n",
      unit.unitType.getName(),
      actionDetails
    );
    if (unit.hasAbility(UnitAbility.ABILITY_SETTLE)) {
      // If this unit was a settler, try and create a city on the unit's tile

      // Broadcast a packet...
      Civilisation.CLIENT.broadcast(new PacketCityCreate(
        renderGame.currentPlayer.id,
        unit.tile.x,
        unit.tile.y
      ));

      // ...and create the city for this client
      renderGame.data.cities.add(new City(
        renderGame.data.hexagonGrid,
        unit.tile.x,
        unit.tile.y,
        renderGame.currentPlayer
      ));
      // Rerender every tile
      renderGame.updateTileRenders();
      renderGame.setSelectedUnit(null);
      // Settlers can only be used once, so delete this unit
      renderGame.deleteUnit(unit, true);
    } else if (unit.hasAbility(UnitAbility.ABILITY_IMPROVE)) {
      // If this unit was a worker, try and improve the unit's tile

      // Get the improvement from the action's details
```

```
      Improvement improvement = Improvement.fromName(actionDetails);
      assert improvement != null;

      // Create a packet detailing the request
      PacketWorkerImproveRequest packetWorkerImproveRequest =
        new PacketWorkerImproveRequest(
          unit.tile.x,
          unit.tile.y,
          improvement
        );
      // Handle it locally and broadcast it so other clients stay in sync
      renderGame.data.handlePacket(packetWorkerImproveRequest);
      Civilisation.CLIENT.broadcast(packetWorkerImproveRequest);
      renderGame.setSelectedUnit(null);
    } else if (unit.unitType.getUpgrade() != null) {
      // If this unit could be upgraded, upgrade the unit

      // Create a packet detailing the request
      PacketUnitUpgrade packetUnitUpgrade = new PacketUnitUpgrade(
        unit.tile.x,
        unit.tile.y
      );
      // Handle it locally and broadcast it so other clients stay in sync
      renderGame.data.handlePacket(packetUnitUpgrade);
      Civilisation.CLIENT.broadcast(packetUnitUpgrade);
      renderGame.setSelectedUnit(null);
    } else if (unit.hasAbility(UnitAbility.ABILITY_BLAST_OFF)) {
      // If this is a rocket, blast off and win the game

      // Create a packet detailing the request
      PacketBlastOff packetBlastOff = new PacketBlastOff(
        renderGame.currentPlayer.id
      );
      // Handle it locally and broadcast it so other clients stay in sync
      renderGame.handlePacket(packetBlastOff);
      Civilisation.CLIENT.broadcast(packetBlastOff);
      renderGame.setSelectedUnit(null);
      // Rockets can only be used once, so delete this unit
      renderGame.deleteUnit(unit, true);
    }
  }
}

/**
 * Sets the tech tree's visibility
```

```
     *
     * @param visible whether the tech tree should be visible
     */
    private void setTechTreeVisible(boolean visible) {
      techTree.setVisible(visible);
      closeTechTreeButton.setVisible(visible);
    }

    /**
     * Callback function for a new chat packet. Adds the new chat message to the
     * chat log.
     *
     * @param packet packet containing the chat message
     */
    void handlePacketChat(PacketChat packet) {
      panelChat.addMessage(packet.message);
    }

    /**
     * Callback function for a ready packet. Resets the action panel's next turn
     * button allowing it to be clicked again.
     *
     * @param data packet containing turn ready information
     */
    void handlePacketReady(PacketReady data) {
      panelActions.setNextTurnWaiting(data.ready);
    }

    /**
     * Callback function called when a player's stats change (usually once per
     * turn)
     *
     * @param stats new stats for the current player
     */
    void onPlayerStatsChanged(PlayerStats stats) {
      panelStats.setPlayerStats(stats);
    }

    /**
     * Callback function called when a player's tech details changed (usually
     * once per turn)
     *
     * @param game    game containing the player
     * @param details new tech details for the current player
     */
```

```
  void onTechDetailsChanged(Game game, PlayerTechDetails details) {
    panelTech.setTechDetails(details);
    techTree.setTechDetails(game, details);
  }
}
```

com/mrbbot/civilisation/ui/game/UIPanelActions.java

```
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.Improvement;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.logic.unit.UnitAbility;
import com.mrbbot.civilisation.logic.unit.UnitType;
import com.mrbbot.civilisation.net.packet.PacketReady;
import com.mrbbot.generic.net.ClientOnly;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;

import java.util.function.BiConsumer;

/**
 * UI panel for controlling the selected unit's actions and for declaring the
 * current player ready for the next turn. Extends {@link VBox} so that items
 * are arranged in a column.
 */
@ClientOnly
public class UIPanelActions extends VBox implements EventHandler<ActionEvent> {
  /**
   * Constant for the preferred width of items in the action list
   */
  private static final int ITEM_WIDTH = 200;
  /**
   * Currently selected unit by the player
   */
```

```
private Unit selectedUnit;
/**
 * Unit for describing the type of the selected unit
 */
private Label selectedUnitLabel;
/**
 * Combo box for the different types of actions that can be performed (only
 * used for worker improvement types at the moment)
 */
private ComboBox<String> actionsComboBox;
/**
 * List of the available actions for the actions box
 */
private ObservableList<String> actionsList;
/**
 * Button to perform the selected action with the selected unit
 */
private Button actionButton;
/**
 * Button to mark the current player as ready and wait for other players to
 * complete their turn
 */
private Button nextTurnButton;
/**
 * Callback function for any actions to be performed. The first parameter is
 * the selected unit or null if the next turn button is pressed. The second
 * parameter contains information of the details of the action (i.e type of
 * improvement)
 */
private BiConsumer<Unit, String> unitActionListener;

UIPanelActions() {
  // Set vertical height
  super(5);
  // Make this panel occupy the minimum height
  setPrefHeight(0);

  // Create the selected unit label and heading
  Label selectedUnitHeading = new Label("Selected unit:");
  selectedUnitLabel = new Label("None");
  selectedUnitLabel.setFont(new Font(24));
  selectedUnitLabel.setPadding(
    new Insets(0, 0, 5, 0)
  );
```

```
    // Create the actions combo box and button
    actionsList = FXCollections.observableArrayList();
    actionsComboBox = new ComboBox<>(actionsList);
    actionsComboBox.setDisable(true);
    actionButton = new Button("");
    // Register this as the click handler
    actionButton.setOnAction(this);
    // Disable it by default
    actionButton.setDisable(true);

    // Create the next turn button
    nextTurnButton = new Button("Next Turn");
    // Register this as the click handler
    nextTurnButton.setOnAction(this);
    // Make the text a bit bigger than usual
    nextTurnButton.setFont(new Font(20));

    actionsComboBox.setPrefWidth(ITEM_WIDTH);
    actionButton.setPrefWidth(ITEM_WIDTH);
    nextTurnButton.setPrefWidth(ITEM_WIDTH);

    // Add all the components to the vertical stack
    getChildren().addAll(
      selectedUnitHeading,
      selectedUnitLabel,
      actionsComboBox,
      actionButton,
      nextTurnButton
    );
  }

  /**
   * Sets the action listener for performing unit actions and requesting the
   * next turn.
   *
   * @param unitActionListener new unit action listener
   */
  public void setUnitActionListener(
    BiConsumer<Unit, String> unitActionListener
  ) {
    this.unitActionListener = unitActionListener;
  }

  /**
   * Sets the current selected unit. Called when the player selects a new unit.
```

```
 *
 * @param game game containing the unit
 * @param unit new selected unit or null if no unit is selected
 */
void setSelectedUnit(Game game, Unit unit) {
  this.selectedUnit = unit;

  // Reset the UI
  actionsList.clear();
  actionsComboBox.setValue("");
  actionsComboBox.setDisable(true);
  actionButton.setText("");
  actionButton.setDisable(true);
  if (unit == null) {
    // If there's no unit selected, use none as the type
    selectedUnitLabel.setText("None");
  } else {
    Tile tile = unit.tile;

    // Set the unit label's text to be the type of the unit
    selectedUnitLabel.setText(unit.unitType.getName());

    // Depending on the units abilities enable different UI components

    // If the unit can settle...
    if (unit.hasAbility(UnitAbility.ABILITY_SETTLE)) {
      // Set the action
      actionButton.setText("Settle");
      // Enable the button if there isn't already a city on the tile
      actionButton.setDisable(tile.city != null);
    }

    // If the unit can improve...
    if (unit.hasAbility(UnitAbility.ABILITY_IMPROVE)) {
      // Add all the improvements that a worker can always do
      for (Improvement improvement : Improvement.VALUES) {
        // Check the player has unlocked the improvement
        if (improvement.workerCanDo
          && game.playerHasUnlocked(unit.player.id, improvement)) {
          actionsList.add(improvement.name);
        }
      }
      // Add chop forest if there's a tree and the player has unlocked it
      if (tile.improvement == Improvement.TREE
        && game.playerHasUnlocked(unit.player.id, Improvement.CHOP_FOREST)) {
```

```
      actionsList.add(Improvement.CHOP_FOREST.name);
    }

    // Set the action
    actionButton.setText("Improve");

    // Set the default improvement
    if (actionsList.size() > 0) {
      actionsComboBox.setValue(actionsList.get(0));
      boolean canImprove = tile.city == null
        || !tile.city.player.equals(unit.player);
      actionsComboBox.setDisable(canImprove);
      actionButton.setDisable(canImprove);
    }

    // If the unit is already building something disable the button and
    // show the progress of the build
    if (unit.workerBuilding != Improvement.NONE) {
      actionButton.setText(
        String.format(
          "Improving... (%d turns remaining)",
          unit.workerBuildTurnsRemaining
        )
      );
      actionsComboBox.setValue(unit.workerBuilding.name);
      actionsComboBox.setDisable(true);
      actionButton.setDisable(true);
    }
  }

  // If the unit can be upgraded and the player has unlocked the upgraded
  // type...
  UnitType upgradedType = unit.unitType.getUpgrade();
  if (upgradedType != null
    && game.playerHasUnlocked(unit.player.id, upgradedType)) {
    // Set the action
    actionButton.setText("Upgrade to " + upgradedType.getName());
    actionButton.setDisable(false);
  }

  // If the unit can blast off, set the action
  if (unit.hasAbility(UnitAbility.ABILITY_BLAST_OFF)) {
    actionButton.setText("Blast off!");
    actionButton.setDisable(false);
  }
```

```
    }
  }

  /**
   * Sets whether or not the game is waiting for players
   *
   * @param waiting whether the game is waiting for other players
   */
  void setNextTurnWaiting(boolean waiting) {
    // Disable the button if we're waiting
    nextTurnButton.setDisable(waiting);
    // Set the button text according to the current state
    nextTurnButton.setText(waiting ? "Waiting..." : "Next Turn");
  }

  /**
   * Handle the next turn/action button events
   *
   * @param event JavaFX event for the button click containing information on
   *              the source of the event
   */
  @Override
  public void handle(ActionEvent event) {
    if (unitActionListener != null) {
      if (event.getSource() == actionButton) {
        // If this was the action button, send the action's event with details
        unitActionListener.accept(selectedUnit, actionsComboBox.getValue());
      } else if (event.getSource() == nextTurnButton) {
        // Otherwise, it was the next turn button, so declare the player ready
        // and broadcast this
        setNextTurnWaiting(true);
        Civilisation.CLIENT.broadcast(new PacketReady(true));
        unitActionListener.accept(null, null);
      }
    }
  }
}
```

com/mrbbot/civilisation/ui/game/UIPanelChat.java

```
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.net.packet.PacketChat;
import com.mrbbot.generic.net.ClientOnly;
import javafx.geometry.Insets;
```

```
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;

/**
 * UI panel for sending/receiving chat messages.
 */
@ClientOnly
class UIPanelChat extends BorderPane {
  /**
   * Log of previous chat messages
   */
  private TextArea log;
  /**
   * Text field containing text to send in next message
   */
  private TextField messageField;
  /**
   * Button that when clicked will send the message
   */
  private Button sendButton;

  /**
   * Creates a new chat panel
   *
   * @param id id of the current player
   */
  UIPanelChat(String id) {
    super();
    setPrefSize(250, 150);

    // Create the chat log
    log = new TextArea("");
    log.setEditable(false);
    // Keep the scroll bar at the bottom of the log when new messages arrive
    log.textProperty().addListener(
      (observable, oldValue, newValue) -> log.setScrollTop(Double.MAX_VALUE)
    );

    BorderPane bottomPane = new BorderPane();
    bottomPane.setPadding(new Insets(5, 0, 0, 0));

    // Create the message field
```

```
    messageField = new TextField();
    bottomPane.setCenter(messageField);
    // Watch the message and enable the send button when any text has been
    // typed in
    messageField.textProperty().addListener(
      (observable, oldValue, newValue) -> sendButton.setDisable(
        newValue.isEmpty()
      )
    );

    // Create the send button
    sendButton = new Button("Send");
    sendButton.setOnAction(e -> {
      // Build the message
      String message = id + "> " + messageField.getText();
      // Add the message locally, and send it to other clients
      addMessage(message);
      Civilisation.CLIENT.broadcast(new PacketChat(message));
      // Clear the message text
      messageField.setText("");
    });
    sendButton.setDisable(true);
    StackPane sendButtonPane = new StackPane(sendButton);
    sendButtonPane.setPadding(new Insets(0, 0, 0, 5));
    bottomPane.setRight(sendButtonPane);

    setCenter(log);
    setBottom(bottomPane);
  }

  /**
   * Adds a chat message to this panel's chat log
   *
   * @param message new message to add with the player id of the sender
   */
  void addMessage(String message) {
    String currentText = log.getText();
    String toAdd = message;
    // Add a new line to the message if there's already text there
    if (!currentText.equals("")) toAdd = "\n" + toAdd;
    // Append the text to the chat log
    log.appendText(toAdd);
  }
}
```

com/mrbbot/civilisation/ui/game/UIPanelStats.java

```java
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.logic.PlayerStats;
import com.mrbbot.generic.net.ClientOnly;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;

/**
 * UI panel for displaying the current player's statistics (gold total, gold
 * per turn, science per turn)
 */
@ClientOnly
public class UIPanelStats extends HBox {
  /**
   * Label for displaying player's science per turn
   */
  private Label scienceLabel;
  /**
   * Label for displaying player's current gold total and their gold per turn
   */
  private Label goldLabel;

  UIPanelStats() {
    // Initialise horizontal box with 10px of horizontal spacing between
    // components
    super(10);

    // Initialise and add the labels and badges to the panel
    scienceLabel = new Label("");
    goldLabel = new Label("");

    getChildren().addAll(
      new Badge(BadgeType.SCIENCE),
      scienceLabel,
      new Badge(BadgeType.GOLD),
      goldLabel
    );
  }

  /**
   * Update the labels' text to reflect the player's new stats
   * @param playerStats object containing the new statistics for the player
   */
  void setPlayerStats(PlayerStats playerStats) {
```

```
    scienceLabel.setText(String.valueOf(playerStats.sciencePerTurn));
    goldLabel.setText(String.format(
      "%d (+%d)",
      playerStats.gold,
      playerStats.goldPerTurn
    ));
  }
}
```

com/mrbbot/civilisation/ui/game/UIPanelCityDetails.java

```
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.logic.CityBuildable;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.map.tile.Building;
import com.mrbbot.civilisation.logic.map.tile.City;
import com.mrbbot.civilisation.logic.unit.UnitType;
import com.mrbbot.civilisation.net.packet.PacketCityBuildRequest;
import com.mrbbot.civilisation.net.packet.PacketCityRename;
import com.mrbbot.civilisation.render.map.RenderGame;
import com.mrbbot.civilisation.ui.UIHelpers;
import com.mrbbot.generic.net.ClientOnly;
import javafx.geometry.Insets;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.text.Font;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * UI panel for showing a cities details and the production list where new
 * units and buildings can be built in the city.
 */
@ClientOnly
public class UIPanelCityDetails extends ScrollPane {
  /**
   * Game the selected city will be contained in
   */
  private final RenderGame renderGame;
  /**
   * Text field for the name of the city
   */
```

```
private TextField cityNameField;
/**
 * Button that when clicked will update the name of the city to reflect the
 * name field.
 */
private Button renameButton;
/**
 * Label for the city's current production
 */
private Label cityProductionLabel;
/**
 * Label for the city's current science
 */
private Label cityScienceLabel;
/**
 * Label for the city's current gold
 */
private Label cityGoldLabel;
/**
 * Label for the city's current food
 */
private Label cityFoodLabel;
/**
 * Panes that contain details on the mapped city buildables in the production
 * list
 */
private HashMap<CityBuildable, Pane> buildablePanes;
/**
 * Tooltips for the mapped city buildables in the production list
 */
private HashMap<CityBuildable, Tooltip> buildableTooltips;
/**
 * Progress indicators for the mapped city buildables in the production list
 */
private HashMap<CityBuildable, ProgressIndicator> buildableProgresses;
/**
 * Array containing the 2 toggle groups for the gold/production buttons.
 * These should be kept in sync with each other.
 */
private ToggleGroup[] productionGoldToggleGroups;
/**
 * Array containing the 2 "build with production" radio buttons. These should
 * have their selected state kept in sync with each other.
 */
private RadioButton[] productionRadioButtons;
```

```
/**
 * Array containing the 2 "build with gold" radio buttons. These should have
 * their selected state kept in sync with each other.
 */
private RadioButton[] goldRadioButtons;
/**
 * Whether or not new buildings should be built with production or gold.
 */
private boolean buildWithProduction;

/**
 * The game containing the last selected city
 */
private Game lastSelectedGame;
/**
 * The city that was last selected by the player
 */
private City lastSelectedCity;
/**
 * All the cities that belonged to the owner of the last selected city
 */
private ArrayList<City> lastSelectedPlayersCities;

UIPanelCityDetails(RenderGame renderGame) {
  super();
  this.renderGame = renderGame;
  setPrefWidth(300);

  // Create empty maps for buildable UI components
  buildablePanes = new HashMap<>();
  buildableTooltips = new HashMap<>();
  buildableProgresses = new HashMap<>();

  VBox list = new VBox();

  // Create the details pane for showing information (production, gold, etc)
  // on a city
  BorderPane detailsTitle = new BorderPane();
  detailsTitle.getStyleClass().add("production-list-title");

  // Create the rename field and button
  detailsTitle.setCenter(cityNameField = new TextField());
  StackPane renamePane =
    new StackPane(renameButton = new Button("Rename"));
  renamePane.setPadding(new Insets(0, 0, 0, 5));
```

```
    detailsTitle.setRight(renamePane);

    // Create the details row
    HBox details = new HBox(7);
    details.setPadding(new Insets(10));
    details.getChildren().addAll(
      new Badge(BadgeType.PRODUCTION),
      cityProductionLabel = new Label("0"),
      new Badge(BadgeType.SCIENCE),
      cityScienceLabel = new Label("0"),
      new Badge(BadgeType.GOLD),
      cityGoldLabel = new Label("0"),
      new Badge(BadgeType.FOOD),
      cityFoodLabel = new Label("0 (0 citizens)")
    );
    list.getChildren().addAll(detailsTitle, details);

    // Create arrays for production/gold buttons
    productionGoldToggleGroups = new ToggleGroup[2];
    productionRadioButtons = new RadioButton[2];
    goldRadioButtons = new RadioButton[2];
    // By default, build buildables with production
    buildWithProduction = true;

    // Add the units header
    list.getChildren().add(buildListTitle("Units", 0));
    for (UnitType unit : UnitType.VALUES) {
      // Add all buildable unit types
      list.getChildren().add(buildListItem(unit));
    }
    // Add the buildings header
    list.getChildren().add(buildListTitle("Buildings", 1));
    for (Building building : Building.VALUES) {
      // Add all buildable buildings
      list.getChildren().add(buildListItem(building));
    }

    // Always show the vertical scrollbar
    setHbarPolicy(ScrollBarPolicy.NEVER);
    setVbarPolicy(ScrollBarPolicy.ALWAYS);

    setContent(list);
  }

  /**
```

```
 * Creates a heading for a buildable list section (units/buildings) with
 * radio buttons for choosing between building with gold and production
 *
 * @param title title for the heading text
 * @param i     index of the section
 * @return pane containing the title
 */
@SuppressWarnings("Duplicates")
private Pane buildListTitle(String title, int i) {
  BorderPane pane = new BorderPane();
  // Add a CSS class for styling
  pane.getStyleClass().add("production-list-title");

  // Add the title
  pane.setLeft(new Label(title));

  //Create a toggle group for this set of buttons
  final ToggleGroup toggleGroup = new ToggleGroup();
  productionGoldToggleGroups[i] = toggleGroup;

  HBox prodGoldBox = new HBox();
  Label buildWithLabel = new Label("Build with");

  // Create build with production and gold buttons
  final RadioButton productionButton = new RadioButton();
  productionRadioButtons[i] = productionButton;
  productionButton.setToggleGroup(toggleGroup);
  productionButton.setSelected(true);
  StackPane productionButtonPane = new StackPane(productionButton);
  productionButtonPane.setPadding(
    new Insets(0, 0, 0, 7)
  );

  final RadioButton goldButton = new RadioButton();
  goldRadioButtons[i] = goldButton;
  goldButton.setToggleGroup(toggleGroup);
  StackPane goldButtonPane = new StackPane(goldButton);
  goldButtonPane.setPadding(new Insets(0, 0, 0, 7));

  // Keep the buttons in this header in sync with the other header by
  // watching for changes
  toggleGroup.selectedToggleProperty().addListener(
    (observable, oldValue, newValue) -> {
      // Whether we're now building with production
      boolean newBuildWithProduction = newValue == productionButton;
```

```
      if (buildWithProduction != newBuildWithProduction) {
        // If they're different, update the other button
        buildWithProduction = newBuildWithProduction;
        int otherIndex = (i + 1) % 2;
        productionGoldToggleGroups[otherIndex].selectToggle(
          buildWithProduction
            ? productionRadioButtons[otherIndex]
            : goldRadioButtons[otherIndex]
        );
        // Recalculate whether or not each item can be built/purchased
        setSelectedCity(
          lastSelectedGame,
          lastSelectedCity,
          lastSelectedPlayersCities
        );
      }
    }
  );

  // Add build with UI components
  prodGoldBox.getChildren().addAll(
    buildWithLabel,
    productionButtonPane,
    new Badge(BadgeType.PRODUCTION),
    goldButtonPane,
    new Badge(BadgeType.GOLD)
  );
  pane.setRight(prodGoldBox);

  return pane;
}

/**
 * Build a list item for a {@link CityBuildable}. Clicking on this will build
 * it in the selected city.
 *
 * @param buildable buildable to constructor a list item for
 * @return list item for that buildable
 */
private Pane buildListItem(CityBuildable buildable) {
  BorderPane listItem = new BorderPane();
  // Add CSS class form styling
  listItem.getStyleClass().add("production-list-item");
  // Make it fill as little height as possible
  listItem.setMaxHeight(0);
```

```
    // Create a tooltip that shows up on hover
    Tooltip tooltip = new Tooltip("Click to build this in the city");
    buildableTooltips.put(buildable, tooltip);
    Tooltip.install(listItem, tooltip);

    VBox list = new VBox();

    // Add the title and description of the buildable
    Label titleLabel = new Label(buildable.getName());
    titleLabel.setFont(new Font(16));
    Label descriptionLabel = new Label(buildable.getDescription());

    // Add the details of the buildable in a row with the relevant badges
    HBox detailsBox = new HBox(7);
    detailsBox.setPadding(new Insets(4, 0, 0, 0));
    for (CityBuildable.Detail detail : buildable.getDetails()) {
      detailsBox.getChildren().add(new Badge(detail.badge));
      if (!detail.text.isEmpty()) {
        detailsBox.getChildren().add(new Label(detail.text));
      }
    }

    list.getChildren().addAll(titleLabel, descriptionLabel, detailsBox);

    listItem.setCenter(list);

    // Add a progress indicator for the build progress
    ProgressIndicator progressIndicator = new ProgressIndicator(0.0);
    progressIndicator.setVisible(false);
    buildableProgresses.put(buildable, progressIndicator);
    listItem.setRight(progressIndicator);

    // Store the pane later so it can be made translucent/invisible
    buildablePanes.put(buildable, listItem);

    return listItem;
  }

  /**
   * Sets the currently selected city and updates the UI to reflect what can be
   * built there.
   *
   * @param game          game the city is contained in
   * @param city          newly selected city
```

```
 * @param playersCities all the cities that belonged to the owner of the
 *                      newly selected city
 */
void setSelectedCity(Game game, City city, ArrayList<City> playersCities) {
  // Store these last selections so this function can be called again later
  lastSelectedGame = game;
  lastSelectedCity = city;
  lastSelectedPlayersCities = playersCities;

  // Update the city details in the labels
  cityProductionLabel.setText(String.valueOf(city.getProductionPerTurn()));
  cityScienceLabel.setText(String.valueOf(city.getSciencePerTurn()));
  cityGoldLabel.setText(String.valueOf(city.getGoldPerTurn()));
  cityFoodLabel.setText(String.format(
    "%d (%d citizen%s)",
    city.getFoodPerTurn(),
    city.citizens,
    city.citizens == 1 ? "" : "s"
  ));
  cityNameField.setText(city.name);

  // Set the rename handler
  renameButton.setOnAction((e) -> {
    String newName = cityNameField.getText();
    // Update the name locally and broadcast a pakcet for the change
    city.name = newName;
    Civilisation.CLIENT.broadcast(new PacketCityRename(
      city.getX(),
      city.getY(),
      newName
    ));
  });

  // Iterate through every buildables' pane
  for (Map.Entry<CityBuildable, Pane> e : buildablePanes.entrySet()) {
    // Get UI components representing the buildable
    CityBuildable buildable = e.getKey();
    Pane buildablePane = e.getValue();
    Tooltip buildableTooltip = buildableTooltips.get(buildable);
    ProgressIndicator progressIndicator = buildableProgresses.get(buildable);

    // Toggle CSS class for opacity
    UIHelpers.toggleClass(
      buildablePane,
      "not-unlocked",
```

```
  !game.playerHasUnlocked(city.player.id, buildable)
);
// Toggle visibility
buildablePane.setVisible(
  game.playerHasUnlocked(city.player.id, buildable)
);
// Prevents the pane from taking up space in the column if it's hidden
buildablePane.setManaged(
  game.playerHasUnlocked(city.player.id, buildable)
);

// Whether this is the currently building item
boolean currentlyBuildingThis = buildable.equals(city.currentlyBuilding);
// Get the reason (if any) why this can't be built in the city
String cantBuildReason = buildable.canBuildGivenCities(
  city,
  playersCities
);

// Calculate the tooltip text
String tooltipText = "Click to build this in the city";
boolean canBuild = true;
if (currentlyBuildingThis) {
  tooltipText = "You are currently building this";
  canBuild = false;
} else if (city.currentlyBuilding != null) {
  tooltipText = "You are currently building something else";
  canBuild = false;
} else if (!buildWithProduction
  && game.getPlayerGoldTotal(city.player.id) < buildable.getGoldCost()) {
  tooltipText = "You don't have enough gold to build this";
  canBuild = false;
} else if (cantBuildReason != null && cantBuildReason.length() > 0) {
  tooltipText = cantBuildReason;
  canBuild = false;
}
// Set the tooltip text to the calculate value
buildableTooltip.setText(tooltipText);

// Update classes for opacity
UIHelpers.toggleClass(
  buildablePane,
  "can-build",
  canBuild
);
```

```
      UIHelpers.toggleClass(
        buildablePane,
        "building",
        currentlyBuildingThis
      );

      // Set the progress of the current build
      if (currentlyBuildingThis) {
        progressIndicator.setProgress(
          Math.min(
            (double) city.productionTotal
              / (double) buildable.getProductionCost(),
            1.0
          )
        );
      }
      progressIndicator.setVisible(currentlyBuildingThis);

      // Add a click listener if this can be built
      if (canBuild) {
        buildablePane.setOnMouseClicked(event -> {
          // Create a packet detailing the build request
          PacketCityBuildRequest packetCityBuildRequest =
            new PacketCityBuildRequest(
              city.getX(),
              city.getY(),
              buildable,
              buildWithProduction
            );
          // Handle it locally and broadcast it to sync the state
          renderGame.handlePacket(packetCityBuildRequest);
          // Update the display of items now that something is being built
          setSelectedCity(game, city, playersCities);
          Civilisation.CLIENT.broadcast(packetCityBuildRequest);
          System.out.println("Clicked on " + buildable.getName());
        });
      } else {
        // Remove it if it can't
        buildablePane.setOnMouseClicked(null);
      }
    }
  }
}
```

com/mrbbot/civilisation/ui/game/styles.css

```
/* Removes default light grey backgrounds on scroll panes */
.scroll-pane > .viewport {
  -fx-background-color: transparent;
}

/* Styles the production list headers (the "Units" and "Buildings") above the
 * list of available buildables.
 */
.production-list-title {
  -fx-pref-width: 275px;
  -fx-background-color: #CCCCCC;
  -fx-padding: 10px;
}

/* Styles a production list item representing a city buildable. */
.production-list-item {
  -fx-pref-width: 275px;
  -fx-background-color: #FFFFFF;
  -fx-padding: 10px;
  /* Make items slightly translucent by default. */
  -fx-opacity: 0.5;
}

/*
 * Removes padding from buildables that haven't been unlocked yet so they don't
 * take up any space in the production list.
 */
.production-list-item.not-unlocked {
  -fx-padding: 0;
}

/* Make items opaque if they can be built or they are currently being built. */
.production-list-item.can-build, .production-list-item.building {
  -fx-opacity: 1;
}

/* Change the background colour of an item when the user is hovering over it */
.production-list-item.can-build:hover {
  -fx-background-color: #EEEEEE;
}

/* Change the background colour of an item when the user is clicking on it */
.production-list-item.can-build:pressed {
  -fx-background-color: #DDDDDD;
}
```

com/mrbbot/civilisation/ui/game/UIPanelTech.java

```java
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.logic.techs.PlayerTechDetails;
import com.mrbbot.generic.net.ClientOnly;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.BorderPane;
import javafx.scene.text.Font;

/**
 * UI panel for basic tech details (what the player is currently researching
 * and their progress towards unlocking it)
 */
@ClientOnly
public class UIPanelTech extends BorderPane {
  /**
   * Label containing the name of the player's current research or "Nothing" if
   * they aren't researching anything
   */
  private Label currentlyResearching;
  /**
   * Progress towards unlocking the current technology
   */
  private ProgressIndicator progress;
  /**
   * Button that when clicked should show the tech tree UI
   */
  private Button openTechTree;

  UIPanelTech() {
    super();

    // Create the current researching label
    Label currentlyResearchingHeading =
      new Label("Currently researching:");
    currentlyResearching = new Label("Nothing");
    currentlyResearching.setFont(new Font(24));
    currentlyResearching.setPadding(
      new Insets(0, 0, 5, 0)
    );
```

```
    // Create the unlock progress indicator
    progress = new ProgressIndicator(0.5);
    progress.setPadding(new Insets(5, 5, 5, 0));

    // Create the open button
    openTechTree = new Button("Open Tech Tree");
    openTechTree.setPrefWidth(230);

    // Position the elements in the border pane
    setTop(currentlyResearchingHeading);
    setLeft(progress);
    setCenter(currentlyResearching);
    setBottom(openTechTree);
  }

  /**
   * Update the UI to reflect new player tech details. Called when a player
   * chooses a new technology to research or the progress of the current
   * project is updated (on new turn)
   *
   * @param details new tech details object for the player
   */
  void setTechDetails(PlayerTechDetails details) {
    // Set the currently researching text to the name of the current tech or
    // "Nothing" if no tech is being researched.
    this.currentlyResearching.setText(details.currentlyUnlocking == null
      ? "Nothing"
      : details.currentlyUnlocking.getName()
    );
    // Set the progress indicator to reflect the current percent unlocked
    this.progress.setProgress(details.percentUnlocked);
  }

  /**
   * Sets the listener to be called when the open tech tree button is pressed.
   * Should show the tech tree UI.
   *
   * @param value listener to be called when the button is pressed
   */
  void setOnOpenTechTree(EventHandler<ActionEvent> value) {
    openTechTree.setOnAction(value);
  }
}
```

com/mrbbot/civilisation/ui/game/UITechTree.java

```java
package com.mrbbot.civilisation.ui.game;

import com.mrbbot.civilisation.Civilisation;
import com.mrbbot.civilisation.logic.techs.Unlockable;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.techs.PlayerTechDetails;
import com.mrbbot.civilisation.logic.techs.Tech;
import com.mrbbot.civilisation.net.packet.PacketPlayerResearchRequest;
import com.mrbbot.generic.net.ClientOnly;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.TextAlignment;

import java.util.*;

import static com.mrbbot.civilisation.ui.UIHelpers.colouredBackground;

/**
 * Tech tree UI. Overlaid on top of the entire game interface when shown
 * so that it takes up the full screen. Extends scroll pane to enable
 * horizontal scrolling.
 */
@ClientOnly
public class UITechTree extends ScrollPane {
  /**
   * Font to be used for rendering the name of technologies in the tree
   */
  private static final Font TECH_TITLE_FONT = Font.font(
    Font.getDefault().getFamily(),
    FontWeight.EXTRA_BOLD,
    15
  );
  /**
   * Default border for technologies in the tree
```

```
 */
private static final Border TECH_BORDER = new Border(
  new BorderStroke(
    Color.BLACK,
    BorderStrokeStyle.SOLID,
    new CornerRadii(5),
    new BorderWidths(5)
  )
);
/**
 * Border for technologies that can be unlocked in the tree. Only used when
 * the player is able to select a new technology (i.e. when one isn't being
 * researched).
 */
private static final Border TECH_CAN_UNLOCK_BORDER = new Border(
  new BorderStroke(
    Color.LIMEGREEN,
    BorderStrokeStyle.SOLID,
    new CornerRadii(5),
    new BorderWidths(5)
  )
);
/**
 * Border for the technology that is currently being unlocked in the tree
 */
private static final Border TECH_UNLOCKING_BORDER = new Border(
  new BorderStroke(
    Color.DEEPSKYBLUE,
    BorderStrokeStyle.SOLID,
    new CornerRadii(5),
    new BorderWidths(5)
  )
);
/**
 * Width of the rounded rectangle that represents a technology in the tree
 */
private static final double TECH_WIDTH = 150;
/**
 * Horizontal spacing between horizontally adjacent techs (spacing between
 * each level of techs). See {@link Tech} for level information.
 */
private static final double TECH_HORIZONTAL_SPACING = 150;
/**
 * Vertical spacing between vertically adjacent techs (spacing between
 * techs on the same level). See {@link Tech} for level information.
```

```
 */
private static final double TECH_VERTICAL_SPACING = 320;

/**
 * ID of the current player
 */
private final String playerId;
/**
 * Graphics context for rendering the connecting curves between technologies
 * in the tree.
 */
private GraphicsContext lineGraphics;
/**
 * Pane containing the rounded rectangles for each of the techs in the tree.
 */
private StackPane techPane;
/**
 * Map mapping techs to their rounded rectangles in the UI.
 */
private Map<Tech, Region> renderedTechs;
/**
 * Middle of the screen. Screen y-coordinate to render techs with a
 * y-coordinate of 0. See {@link Tech#getY()}.
 */
private int lineOffset;

/**
 * Create a new tech tree UI
 *
 * @param game     game containing the current player
 * @param playerId ID of the current player
 * @param details  the current player's tech details
 * @param height   the screen height
 */
UITechTree(
  Game game,
  String playerId,
  PlayerTechDetails details,
  int height
) {
  super();
  this.playerId = playerId;

  // Always show the horizontal scroll bar
  setVbarPolicy(ScrollBarPolicy.NEVER);
```

```
    setHbarPolicy(ScrollBarPolicy.ALWAYS);
    setFitToHeight(true);

    // Position techs with a y-coordinate of 0 in the middle of the screen
    lineOffset = height / 2;
    // Create the canvas and context for rendering connecting curves between
    // the techs
    Canvas lineCanvas = new Canvas(
      // Max it wide enough to contain all the lines
      (Tech.MAX_X * (TECH_WIDTH + TECH_HORIZONTAL_SPACING)) + 20,
      height
    );
    lineGraphics = lineCanvas.getGraphicsContext2D();
    lineGraphics.setLineWidth(5);

    // Create the pane/map that all tech rounded rectangles should be added to
    techPane = new StackPane();
    techPane.setAlignment(Pos.CENTER_LEFT);
    renderedTechs = new HashMap<>();
    // Traverse the tech tree, adding all techs to the UI along the way
    addTechs(Tech.getRoot());

    // Stack the tech rectangles on top of the connecting lines
    StackPane rootPane = new StackPane();
    rootPane.setAlignment(Pos.TOP_LEFT);
    rootPane.getChildren().addAll(lineCanvas, techPane);
    setContent(rootPane);

    // Set the initial player tech details, making certain techs clickable in
    // the tree.
    setTechDetails(game, details);
  }

  /**
   * Adds techs' rounded rectangles to the UI so they can be seen/selected by
   * the user.
   *
   * @param techToRender root of the tech tree containing children that will
   *                     be recursively passed back to this function to render
   *                     their children and so fourth
   */
  private void addTechs(Tech techToRender) {
    // Calculate the position to render this tech in
    int x = techToRender.getX();
    int y = techToRender.getY();
```

```
double renderX = (x * (TECH_WIDTH + TECH_HORIZONTAL_SPACING)) + 10;
double renderY = y * TECH_VERTICAL_SPACING;

// Make sure each tech is only rendered once
if (!renderedTechs.containsKey(techToRender)) {
  // Create the rounded rectangle for this tech with some vertical spacing
  // between its subcomponents
  VBox tech = new VBox(10);
  tech.setMinWidth(TECH_WIDTH);
  tech.setAlignment(Pos.CENTER);
  tech.setPadding(new Insets(
    0,
    0,
    // Give the tech some padding if it unlocks things so the unlock list
    // is centered (if it exists)
    techToRender.getUnlocks().size() > 0 ? 10 : 0,
    0
  ));
  tech.setMaxSize(0, 0);
  tech.setBorder(TECH_BORDER);
  // Store the render so the tech isn't rendered again and so it can be
  // updated later
  renderedTechs.put(techToRender, tech);

  // Create/add a label for the name of the tech
  Label titleLabel = makeCenteredLabel(techToRender.getName());
  titleLabel.setFont(TECH_TITLE_FONT);
  titleLabel.setPadding(new Insets(10));
  titleLabel.setAlignment(Pos.CENTER);
  titleLabel.setPrefWidth(Double.MAX_VALUE);
  titleLabel.setTextFill(Color.WHITE);
  titleLabel.setBackground(colouredBackground(techToRender.getColour()));
  tech.getChildren().add(titleLabel);

  // Create/add labels for each of the techs unlocks
  for (Unlockable unlock : techToRender.getUnlocks()) {
    tech.getChildren().add(makeCenteredLabel(unlock.getName()));
  }

  // Position the tech on the screen
  StackPane.setMargin(tech, new Insets(
    renderY,
    10,
    0,
```

```
    renderX
  ));
  // Add it to the UI
  techPane.getChildren().add(tech);
}

// Update renderY for line coordinates
renderY = (renderY / 2) + lineOffset;

// Render the connections between this tech and any children. Then render
// those children if they haven't already been.
for (Tech child : techToRender.getRequiredBy()) {
  // Calculate the end coordinates of the connecting line
  int endX = child.getX();
  int endY = child.getY();

  double endRenderX =
    (endX * (TECH_WIDTH + TECH_HORIZONTAL_SPACING)) + 10;
  double endRenderY = (endY * TECH_VERTICAL_SPACING / 2) + lineOffset;
  double startX = renderX + TECH_WIDTH;
  double midRenderX = (startX + endRenderX) / 2;

  // Set the stroke colour to a linear gradient of the different techs'
  // colours
  lineGraphics.setStroke(new LinearGradient(
    0, 0,
    1, 0,
    true, null,
    new Stop(0, techToRender.getColour()),
    new Stop(1, child.getColour()))
  );
  // Start drawing the connecting curve
  lineGraphics.beginPath();
  // Start Coordinates
  lineGraphics.moveTo(startX, renderY);
  lineGraphics.bezierCurveTo(
    // Control Point 1
    midRenderX, renderY,
    // Control Point 2
    midRenderX, endRenderY,
    // End Coordinates
    endRenderX, endRenderY
  );

  // Actually draw the line to the canvas
```

```
    lineGraphics.stroke();
    lineGraphics.closePath();

    // Add the child to the UI along with any of its children (recursive call)
    addTechs(child);
  }
}

/**
 * Creates a label with centered text
 *
 * @param text text of the label
 * @return label containing the specified text in the center
 */
private Label makeCenteredLabel(String text) {
  Label label = new Label(text);
  label.setTextAlignment(TextAlignment.CENTER);
  return label;
}

/**
 * Update the rounded rectangles representing the different technologies.
 *
 * @param game    game the current player is contained within
 * @param details current player's technology details
 */
void setTechDetails(Game game, PlayerTechDetails details) {
  // Iterate through all of the tech renders
  for (Map.Entry<Tech, Region> entry : renderedTechs.entrySet()) {
    final Tech tech = entry.getKey();
    final Region render = entry.getValue();

    // Check if this tech is currently being unlocked
    boolean current = tech.equals(details.currentlyUnlocking)
      && details.percentUnlocked < 1;
    // Check if this tech is already unlocked
    boolean unlocked = tech.getScienceCost() == 0
      || details.unlockedTechs.contains(tech)
      || (tech.equals(details.currentlyUnlocking)
      && details.percentUnlocked == 1);
    // Check if the player can unlock this tech given its requirements
    boolean canUnlock = !unlocked &&
      details.currentlyUnlocking == null &&
      tech.canUnlockGivenUnlocked(details.unlockedTechs);
```

```
      // Make the tech opaque only if any of these conditions are true
      render.setOpacity(
        unlocked || current || canUnlock
          ? 1
          : 0.2
      );
      // Set the tech's border depending on these conditions
      render.setBorder(
        canUnlock
          ? TECH_CAN_UNLOCK_BORDER
          : (current ? TECH_UNLOCKING_BORDER : TECH_BORDER)
      );

      // Register a click listener if this tech can be unlocked
      render.setOnMouseClicked(
        canUnlock
          ? (e) -> {
          // Request the player start researching this technology
          PacketPlayerResearchRequest packetPlayerResearchRequest =
            new PacketPlayerResearchRequest(playerId, tech);
          // Handle it locally and broadcast it to keep the game state in sync
          game.handlePacket(packetPlayerResearchRequest);
          Civilisation.CLIENT.broadcast(packetPlayerResearchRequest);
        }
          : null
      );
    }
  }
}
```

com/mrbbot/civilisation/net/packet/Packet.java

```
package com.mrbbot.civilisation.net.packet;

import java.io.Serializable;

/**
 * Base packet class. All packet types extend this. Packets are sent between
 * clients and the server to keep the game state synchronised.
 * <p>
 * Implements serializable so
 * that any subclass can be sent over the network with
 * {@link java.io.ObjectInputStream} and {@link java.io.ObjectOutputStream}
 * both of which allow Java objects to be sent/received. Implementing
 * serializable means that all class fields must themselves be serializable.
 */
```

```
public abstract class Packet implements Serializable {
}
```

com/mrbbot/civilisation/net/packet/PacketBlastOff.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a player activates the blast off action. On receiving
 * this packet, the specified player should win the game.
 */
public class PacketBlastOff extends PacketUpdate {
  /**
   * ID of the player who blasted off and is now the winner of the game
   */
  public String playerId;

  public PacketBlastOff(String playerId) {
    this.playerId = playerId;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketChat.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a player sends a chat message from the UI. On receiving
 * this packet, the UI of this game should update to include the new message.
 */
public class PacketChat extends PacketUpdate {
  /**
   * Chat message sent by the player along with their player ID. An example
   * would be "Player: Hello?".
   */
  public final String message;

  public PacketChat(String message) {
    this.message = message;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketCityBuildRequest.java

```
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.CityBuildable;

/**
```

```java
 * Packet emitted when a player clicks on a buildable in the city production
 * list. On receiving this packet, the game should start the build of this in
 * the selected city, or purchase the item with gold if that was requested.
 */
public class PacketCityBuildRequest extends PacketUpdate {
  /**
   * X-coordinate of the city to build in
   */
  public final int x;
  /**
   * Y-coordinate of the city to build in
   */
  public final int y;
  /**
   * Name of the buildable object to build in the city (buildables aren't
   * serializable so we must store the unique name instead)
   */
  private final String buildable;
  /**
   * Whether to build this with production or to just purchase it outright
   * with gold.
   */
  public final boolean withProduction;

  public PacketCityBuildRequest(
    int x,
    int y,
    CityBuildable buildable,
    boolean withProduction
  ) {
    this.x = x;
    this.y = y;
    this.buildable = buildable.getName();
    this.withProduction = withProduction;
  }

  /**
   * Gets the city buildable this packet contains. Buildables aren't
   * serializable so they must be recreated from their name.
   *
   * @return buildable this packet contains
   */
  public CityBuildable getBuildable() {
    return CityBuildable.fromName(buildable);
```

```
  }
}
```

com/mrbbot/civilisation/net/packet/PacketCityCreate.java

```java
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a player wants to create a city. On receiving this
 * packet, the game should create the city at the specified coordinates for the
 * player.
 */
public class PacketCityCreate extends PacketUpdate {
  /**
   * ID of the player who's creating this city
   */
  public final String id;
  /**
   * X-coordinate of the city
   */
  public final int x;
  /**
   * Y-coordinate of the city
   */
  public final int y;

  public PacketCityCreate(String id, int x, int y) {
    this.id = id;
    this.x = x;
    this.y = y;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketCityGrow.java

```java
package com.mrbbot.civilisation.net.packet;

import javafx.geometry.Point2D;

import java.util.ArrayList;

/**
 * Packet emitted when a city grows to a new set of tiles. On receiving this
 * packet, the game should add the specified tiles to the cities territory.
 */
public class PacketCityGrow extends PacketUpdate {
  /**
```

```
 * ID of the owner of the city to grow
 */
public final String id;
/**
 * X-coordinate of the city to grow
 */
public final int x;
/**
 * Y-coordinate of the city to grow
 */
public final int y;
/**
 * X-coordinates of the new set of tiles to grow to ({@link Point2D} isn't
 * serializable)
 */
private final int[] grownToXs;
/**
 * Y-coordinates of the new set of tiles to grow to ({@link Point2D} isn't
 * serializable)
 */
private final int[] grownToYs;

public PacketCityGrow(String id, int x, int y, ArrayList<Point2D> grownTo) {
  this.id = id;
  this.x = x;
  this.y = y;

  // Split the grown to coordinates into their x and y components so they
  // can be serialized
  int grownToSize = grownTo.size();
  grownToXs = new int[grownToSize];
  grownToYs = new int[grownToSize];
  for (int i = 0; i < grownToSize; i++) {
    grownToXs[i] = (int) grownTo.get(i).getX();
    grownToYs[i] = (int) grownTo.get(i).getY();
  }
}

/**
 * Reconstructs the grown to coordinates from their x and y components
 * @return coordinates of tiles the city should grow to
 */
public ArrayList<Point2D> getGrownTo() {
  int grownToSize = grownToXs.length;
  ArrayList<Point2D> grownTo = new ArrayList<>(grownToSize);
```

```
    for (int i = 0; i < grownToSize; i++) {
      grownTo.add(new Point2D(grownToXs[i], grownToYs[i]));
    }
    return grownTo;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketCityRename.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a user requests the specified city should have a
 * different name. On receiving this packet, the name of the city should be
 * updated.
 */
public class PacketCityRename extends PacketUpdate {
  /**
   * X-coordinate of the city to rename
   */
  public final int x;
  /**
   * Y-coordinate of the city to rename
   */
  public final int y;
  /**
   * New name the user has chosen for the city
   */
  public final String newName;

  public PacketCityRename(int x, int y, String newName) {
    this.x = x;
    this.y = y;
    this.newName = newName;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketDamage.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a unit attacks another living thing (city or other unit).
 * On receiving this packet, the game should check if the attack is valid, and
 * then perform the attack.
 */
public class PacketDamage extends PacketUpdate {
```

```
  /**
   * X-coordinate of the attacking unit
   */
  public final int attackerX;
  /**
   * Y-coordinate of the attacking unit
   */
  public final int attackerY;
  /**
   * X-coordinate of the target living thing (city or another unit)
   */
  public final int targetX;
  /**
   * Y-coordinate of the target living thing (city or another unit)
   */
  public final int targetY;

  public PacketDamage(int attackerX, int attackerY, int targetX, int targetY) {
    this.attackerX = attackerX;
    this.attackerY = attackerY;
    this.targetX = targetX;
    this.targetY = targetY;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketGame.java

```
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.map.Game;

import java.util.Map;

/**
 * Packet emitted a new user joins the game containing the current game state.
 * On receiving this packet, the game should load the state and initialise and
 * display the 3D game render.
 */
public class PacketGame extends Packet {
  /**
   * Map containing the game state. See {@link Game#toMap()}.
   */
  public final Map<String, Object> map;

  public PacketGame(Map<String, Object> map) {
    this.map = map;
```

```
  }
}
```

com/mrbbot/civilisation/net/packet/PacketInit.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted by the client requesting the game state. On receiving this
 * packet, the server should send a {@link PacketGame} containing the game
 * state.
 */
public class PacketInit extends Packet {
}
```

com/mrbbot/civilisation/net/packet/PacketPlayerChange.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted by the server when a new player joins the game. On receiving
 * this packet, the game should add this player to the list of players.
 */
public class PacketPlayerChange extends Packet {
  public final String id;

  public PacketPlayerChange(String id) {
    this.id = id;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketPlayerResearchRequest.java

```
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.techs.Tech;

/**
 * Packet emitted when a player starts to research a new tech. On receiving
 * this packet, the game should change the currently researching tech for the
 * player.
 */
public class PacketPlayerResearchRequest extends PacketUpdate {
  /**
   * ID of the player requesting the research change
   */
  public final String playerId;
  /**
   * Name of the tech to be researched (techs aren't serializable themselves so
```

```
     * they must be remade on receiving the packet)
     */
    private final String techName;

    public PacketPlayerResearchRequest(String playerId, Tech tech) {
        this.playerId = playerId;
        this.techName = tech.getName();
    }

    /**
     * Gets the unserializable tech from the tech name
     *
     * @return tech specified by this packet
     */
    public Tech getTech() {
        return Tech.fromName(techName);
    }
}
```

com/mrbbot/civilisation/net/packet/PacketReady.java

```
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.map.Game;

/**
 * Packet emitted by a client when a player marks themselves as ready or by the
 * server when all players have marked themselves as ready. On receiving this
 * packet on the server, the server should mark the player as ready and check
 * if all other players have done the same thing, moving the game onto the next
 * turn. On receiving this packet on the client, the turn should be handled.
 * See {@link Game#handleTurn(Game)}.
 */
public class PacketReady extends Packet {
    /**
     * Whether the player is ready. Should always be true when sending from the
     * client, and false when sending from the server.
     */
    public final boolean ready;

    public PacketReady(boolean ready) {
        this.ready = ready;
    }
}
```

com/mrbbot/civilisation/net/packet/PacketPurchaseTileRequest.java

```java
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a player attempts to buy a tile. On receiving this
 * packet, the game should check if the player can purchase that tile and if
 * they can, bring the tile into the city's territory.
 */
public class PacketPurchaseTileRequest extends PacketUpdate {
  /**
   * X-coordinate of the city the player is trying to expand
   */
  public int cityX;
  /**
   * Y-coordinate of the city the player is trying to expand
   */
  public int cityY;
  /**
   * X-coordinate of the tile the player is trying to purchase
   */
  public int purchaseX;
  /**
   * Y-coordinate of the tile the player is trying to purchase
   */
  public int purchaseY;

  public PacketPurchaseTileRequest(
    int cityX,
    int cityY,
    int purchaseX,
    int purchaseY
  ) {
    this.cityX = cityX;
    this.cityY = cityY;
    this.purchaseX = purchaseX;
    this.purchaseY = purchaseY;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketUnitCreate.java

```java
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.unit.UnitType;

/**
 * Packet emitted when a player creates a unit (by building one in a city, or
```

```
 * by starting the game with some). On receiving this packet, the game should
 * create a unit belonging to the specified player with the specified type and
 * try to place it as close as possible to the target location.
 */
public class PacketUnitCreate extends PacketUpdate {
  /**
   * ID of the player to create the unit for
   */
  public final String id;
  /**
   * X-coordinate of the tile to place the unit close to
   */
  public final int x;
  /**
   * Y-coordinate of the tile to place the unit close to
   */
  public final int y;
  /**
   * Type of unit to create (unit type's aren't serializable so only the name
   * is stored)
   */
  private final String unitType;

  public PacketUnitCreate(String id, int x, int y, UnitType unitType) {
    this.id = id;
    this.x = x;
    this.y = y;
    this.unitType = unitType.getName();
  }

  /**
   * Finds the unit type specified by the name is this packet
   *
   * @return the unit type specified by this packet
   */
  public UnitType getUnitType() {
    return UnitType.fromName(unitType);
  }
}
```

com/mrbbot/civilisation/net/packet/PacketUnitMove.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a selected unit is moved across the map. On receiving
```

```
 * this packet, the unit should be moved and the used movement points should be
 * deducted from the units remaining count.
 */
public class PacketUnitMove extends PacketUpdate {
  /**
   * X-coordinate of the unit's current tile
   */
  public final int startX;
  /**
   * Y-coordinate of the unit's current tile
   */
  public final int startY;
  /**
   * X-coordinate of the target tile
   */
  public final int endX;
  /**
   * Y-coordinate of the target tile
   */
  public final int endY;
  /**
   * How many movement points should be consumed by moving. Depends on the path
   * taken.
   */
  public final int usedMovementPoints;

  public PacketUnitMove(
    int startX,
    int startY,
    int endX,
    int endY,
    int usedMovementPoints
  ) {
    this.startX = startX;
    this.startY = startY;
    this.endX = endX;
    this.endY = endY;
    this.usedMovementPoints = usedMovementPoints;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketUnitDelete.java

```
package com.mrbbot.civilisation.net.packet;

/**
```

```
 * Packet emitted when a unit should be removed from the game for some reason
 * (settler settling, unit death, etc). On receiving this packet, the unit at
 * the specified coordinate should be removed from the game.
 */
public class PacketUnitDelete extends PacketUpdate {
  /**
   * X-coordinate of the tile containing the unit to be removed
   */
  public final int x;
  /**
   * Y-coordinate of the tile containing the unit to be removed
   */
  public final int y;

  public PacketUnitDelete(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketUnitUpgrade.java

```
package com.mrbbot.civilisation.net.packet;

/**
 * Packet emitted when a user requests that a unit be upgraded to an improved
 * type. On receiving this packet, the game should upgrade the unit type, and
 * proportionally set the health of the unit (see
 * {@link com.mrbbot.civilisation.logic.Living#setBaseHealth(int)}).
 */
public class PacketUnitUpgrade extends PacketUpdate {
  /**
   * X-coordinate of tile containing unit to upgrade
   */
  public final int x;
  /**
   * Y-coordinate of tile containing unit to upgrade
   */
  public final int y;

  public PacketUnitUpgrade(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

com/mrbbot/civilisation/net/packet/PacketUpdate.java

```java
package com.mrbbot.civilisation.net.packet;

/**
 * Abstract class extending Packet that describes a packet containing
 * information relating to game state. On receiving these types of packets, the
 * server should send them to all connected clients but the sender as they will
 * be handled locally there.
 */
public abstract class PacketUpdate extends Packet {
}
```

com/mrbbot/civilisation/net/packet/PacketWorkerImproveRequest.java

```java
package com.mrbbot.civilisation.net.packet;

import com.mrbbot.civilisation.logic.map.tile.Improvement;

/**
 * Packet emitted when a player requests that a worker improve a tile. On
 * receiving this packet, the game should set the workers current build project
 * to the specified one.
 */
public class PacketWorkerImproveRequest extends PacketUpdate {
  /**
   * X-coordinate of the tile containing the worker where the improvement
   * should built
   */
  public final int x;
  /**
   * Y-coordinate of the tile containing the worker where the improvement
   * should built
   */
  public final int y;
  /**
   * Name of the improvement to be built (improvements aren't serializable so
   * only the name is stored)
   */
  private String improvementName;

  public PacketWorkerImproveRequest(int x, int y, Improvement improvement) {
    this.x = x;
    this.y = y;
    this.improvementName = improvement.name;
  }
```

```
  /**
   * Finds the improvement from the name contained in the packet
   *
   * @return improvement specified by this packet
   */
  public Improvement getImprovement() {
    return Improvement.fromName(improvementName);
  }
}
```

com/mrbbot/civilisation/logic/map/tile/Building.java

```
package com.mrbbot.civilisation.logic.map.tile;

import com.mrbbot.civilisation.logic.CityBuildable;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.ui.game.BadgeType;

import java.util.ArrayList;

/**
 * Class representing a building that can be built within a city. Declared
 * abstract as buildings must implement {@link Building#setDetails()} to
 * register the abilities each building has.
 */
@SuppressWarnings("WeakerAccess")
public abstract class Building extends CityBuildable {
  /**
   * Base unlock ID for buildings. Used to identify buildings that can be
   * unlocked.
   */
  private static int BASE_UNLOCK_ID = 0x30;

  /*
   * START BUILDING DEFINITIONS
   */
  public static Building WALL = new Building(
    "Walls",
    "Protect a city",
    100,
    BASE_UNLOCK_ID
  ) {
    @Override
    protected void setDetails() {
      baseHealthIncrease = 100;
    }
```

```
};
public static Building MONUMENT = new Building(
  "Monument",
  "Reduces cost of expansion",
  100,
  BASE_UNLOCK_ID + 1
) {
  @Override
  protected void setDetails() {
    goldPerTurnIncrease = 5;
    expansionCostMultiplier = 0.75;
  }
};
public static Building BANK = new Building(
  "Bank",
  "Doubles a city's gold per turn",
  150,
  BASE_UNLOCK_ID + 2
) {
  @Override
  protected void setDetails() {
    goldPerTurnMultiplier = 2;
  }
};
public static Building AMPHITHEATRE = new Building(
  "Amphitheatre",
  "Gives citizens a place to spend money",
  150,
  BASE_UNLOCK_ID + 3
) {
  @Override
  protected void setDetails() {
    goldPerTurnIncrease = 10;
  }
};
public static Building SCHOOL = new Building(
  "School",
  "Educates citizens",
  100,
  BASE_UNLOCK_ID + 4
) {
  @Override
  protected void setDetails() {
    sciencePerTurnIncrease = 10;
  }
```

```
};
public static Building UNIVERSITY = new Building(
  "University",
  "Must have a school in every city.",
  200,
  BASE_UNLOCK_ID + 5
) {
  @Override
  protected void setDetails() {
    sciencePerTurnMultiplier = 2;
  }

  @Override
  public String canBuildGivenCities(City city, ArrayList<City> cities) {
    // Check if there is a reason why this can't be built already and return
    // it if there is
    String superReason = super.canBuildGivenCities(city, cities);
    if (superReason.length() > 0) return superReason;

    // Otherwise check all other cities contain a school
    for (City otherCity : cities) {
      if (!otherCity.buildings.contains(SCHOOL)) {
        return "You must have a school in all of your cities!";
      }
    }

    // If they do, return an empty string indicating this building can be
    // built
    return "";
  }
};
public static Building FACTORY = new Building(
  "Factory",
  "Increases cities production",
  200,
  BASE_UNLOCK_ID + 6
) {
  @Override
  protected void setDetails() {
    productionPerTurnMultiplier = 2;
  }
};
public static Building POWER_STATION = new Building(
  "Power Station",
  "Increases cities production",
```

```
    400,
    BASE_UNLOCK_ID + 7
  ) {
    @Override
    protected void setDetails() {
      productionPerTurnMultiplier = 2;
    }
  };
  public static Building SUPERMARKET = new Building(
    "Supermarket",
    "Gives citizens a place to get food",
    300,
    BASE_UNLOCK_ID + 8
  ) {
    @Override
    protected void setDetails() {
      goldPerTurnIncrease = 10;
      foodPerTurnMultiplier = 2;
    }
  };
  /*
   * END BUILDING DEFINITIONS
   */

  /**
   * Array containing all defined buildings.
   */
  public static Building[] VALUES = new Building[]{
    WALL,
    MONUMENT,
    BANK,
    AMPHITHEATRE,
    SCHOOL,
    UNIVERSITY,
    FACTORY,
    POWER_STATION,
    SUPERMARKET
  };

  /**
   * Function to get a building from just its name
   *
   * @param name name of building to get
   * @return the building with the specified name or null if the building
   * doesn't exist
```

```
 */
public static Building fromName(String name) {
  // Iterates through all the buildings...
  for (Building value : VALUES) {
    // Checking if the names match
    if (value.name.equals(name)) return value;
  }
  return null;
}

/**
 * Increase in gold per turn for a city containing this building
 */
public int goldPerTurnIncrease = 0;
/**
 * Increase in science per turn for a city containing this building
 */
public int sciencePerTurnIncrease = 0;
/**
 * Increase in base health for a city containing this building
 */
public int baseHealthIncrease = 0;

/**
 * Gold per turn multiplier for a city containing the building
 */
public double goldPerTurnMultiplier = 1;
/**
 * Expansion cost multiplier for a city containing the building
 */
public double expansionCostMultiplier = 1;
/**
 * Science per turn multiplier for a city containing the building
 */
public double sciencePerTurnMultiplier = 1;
/**
 * Production per turn multiplier for a city containing the building
 */
public double productionPerTurnMultiplier = 1;
/**
 * Food per turn multiplier for a city containing the building
 */
public double foodPerTurnMultiplier = 1;

private Building(
```

```
    String name,
    String description,
    int productionCost,
    int unlockId
  ) {
    // Pass required values to CityBuildable constructor
    super(name, description, productionCost, unlockId);
    setDetails();
  }

  /**
   * Called by the constructor to set the increases/multipliers this building
   * provides for the city it's built in
   */
  protected abstract void setDetails();

  /**
   * Get the text to be displayed in the city production list for a resource
   * that may have an increase and/or a multiplier
   *
   * @param increase   increase in resource this building provides
   * @param multiplier multiplier in resource this building provides
   * @return text to be displayed in the city production list, example "7 (x3)"
   */
  private String getDetailTextForIncreaseWithMultiplier(
    int increase,
    double multiplier
  ) {
    StringBuilder text = new StringBuilder();
    // If there is an increase, add it to the text
    if (increase > 0) text.append(increase);
    // If there is a multiplier...
    if (multiplier != 1) {
      // Determine whether there was an increase
      boolean increased = text.length() > 0;
      // If there was, add a space and a bracket
      if (increased) text.append(" (");
      // Even if there wasn't add the multiplier
      text.append("x").append((int) multiplier);
      // Add the closing bracket if required
      if (increased) text.append(")");
    }
    return text.toString();
  }
```

```java
/**
 * Gets the details to be displayed in the city production list for this
 * building
 *
 * @return details to be displayed
 */
@Override
public ArrayList<Detail> getDetails() {
  // Get the details required for all CityBuildables (production/gold cost)
  ArrayList<Detail> details = super.getDetails();

  // Add the gold increase/multiplier (if there is one)
  String goldText = getDetailTextForIncreaseWithMultiplier(
    goldPerTurnIncrease, goldPerTurnMultiplier
  );
  if (goldText.length() > 0)
    details.add(new Detail(BadgeType.GOLD, goldText));

  // Add the science increase/multiplier (if there is one)
  String scienceText = getDetailTextForIncreaseWithMultiplier(
    sciencePerTurnIncrease, sciencePerTurnMultiplier
  );
  if (scienceText.length() > 0)
    details.add(new Detail(BadgeType.SCIENCE, scienceText));

  // Add the production multiplier (if there is one)
  if (productionPerTurnMultiplier != 1) {
    details.add(new Detail(
      BadgeType.PRODUCTION,
      String.format("x%d", (int) productionPerTurnMultiplier))
    );
  }

  // Add the science base health increase (if there is one)
  if (baseHealthIncrease != 0) {
    details.add(new Detail(BadgeType.HEALTH, baseHealthIncrease));
  }

  return details;
}

/**
 * Determine if a building can be built in a city given the player's other
 * cities
 *
```

```
   * @param city   target city to build in
   * @param cities player's other cities
   * @return reason why the building cannot be built, or an empty string if it
   * can
   */
  @Override
  public String canBuildGivenCities(City city, ArrayList<City> cities) {
    return city.buildings.contains(this)
      ? "You can only have one of these buildings per city"
      : "";
  }

  /**
   * Build the building in the specified city
   *
   * @param city city to build in
   * @param game game the city is contained within
   * @return tile to update the render of
   */
  @Override
  public Tile build(City city, Game game) {
    // Add this building to the city
    city.buildings.add(this);
    // Return the city center for re-rendering
    return city.getCenter();
  }
}
```

com/mrbbot/civilisation/logic/map/tile/City.java

```
package com.mrbbot.civilisation.logic.map.tile;

import com.mrbbot.civilisation.geometry.HexagonGrid;
import com.mrbbot.civilisation.logic.CityBuildable;
import com.mrbbot.civilisation.logic.Living;
import com.mrbbot.civilisation.logic.Player;
import com.mrbbot.civilisation.logic.map.Game;
import com.mrbbot.civilisation.logic.unit.Unit;
import javafx.geometry.Point2D;
import javafx.scene.paint.Color;

import java.util.*;
import java.util.stream.Collectors;

public class City extends Living {
  /**
```

```
 * Hexagon grid the city is contained within
 */
private final HexagonGrid<Tile> grid;
/**
 * Player the city is owned by
 */
public Player player;
/**
 * Colour of the walls of the city
 */
public Color wallColour;
/**
 * Colour of the wall joins of the city
 */
public Color joinColour;

/**
 * Tiles that the city owns
 */
public ArrayList<Tile> tiles;
/**
 * Height of the tile with the greatest height
 */
public double greatestTileHeight;

/**
 * Buildings the city has
 */
public ArrayList<Building> buildings;
/**
 * What the city is currently building
 */
public CityBuildable currentlyBuilding;
/**
 * The current production total in the city. When this value reaches the
 * currentlyBuilding's cost it will be built.
 */
public int productionTotal;
/**
 * The number of citizens within the city. Used to calculated gold/science
 * per turn.
 */
public int citizens;
/**
 * The amount of excess food the city has. Controls when the city
```

```
 * grows/starves.
 */
public int excessFoodCounter;

/**
 * The last unit that attacked this city. Used to control who to give the
 * city to if its health reaches 0.
 */
public Unit lastAttacker;

/**
 * The name of the city. Can be edited by the player.
 */
public String name;

/**
 * Constructor for a new city
 *
 * @param grid    hexagon grid for the game
 * @param centerX the center x-coordinate of the city
 * @param centerY the center y-coordinate of the city
 * @param player  the player who owns this city
 */
public City(HexagonGrid<Tile> grid, int centerX, int centerY, Player player) {
  // Pass required parameters to base living class
  super(200);
  // Store passed values
  this.grid = grid;
  setOwner(player);
  this.name = "City";

  // Create empty list for tiles
  tiles = new ArrayList<>();

  // Get center and check it doesn't already have a city
  Tile center = grid.get(centerX, centerY);
  if (center.city != null) {
    throw new IllegalArgumentException(
      "City created on tile with another city"
    );
  }
  tiles.add(center);
  // Make the center of the city a capital
  center.improvement = Improvement.CAPITAL;
```

```
    // Add all of the adjacent tiles that don't already have a city
    ArrayList<Tile> adjacentTiles = grid.getNeighbours(
      centerX, centerY,
      false
    );
    adjacentTiles.removeIf(tile -> tile.city != null);
    tiles.addAll(adjacentTiles);

    // Mark all of the cities tiles as belonging to this city
    tiles.forEach(tile -> tile.city = this);
    // Calculate the greatest height of all the tiles
    updateGreatestHeight();

    // Create empty list for buildings
    buildings = new ArrayList<>();

    // Reset totals
    productionTotal = 0;
    citizens = 1;
    excessFoodCounter = 0;
  }

  /**
   * Constructor for a city loaded from a Map (could be from a file/server)
   *
   * @param grid hexagon grid for the game
   * @param map  map containing city data
   */
  public City(HexagonGrid<Tile> grid, Map<String, Object> map) {
    // Pass required parameters to base living class
    super((int) map.get("baseHealth"), (int) map.get("health"));
    // Store passed values
    this.grid = grid;

    // Load the city owner
    setOwner(new Player((String) map.get("owner")));

    // Load the tiles belonging to the city
    //noinspection unchecked
    tiles = (ArrayList<Tile>) ((List<Map<String, Object>>) map.get("tiles"))
      .stream()
      .map(m -> {
        // Get the tile with the specified coordinates
        int x = (int) m.get("x");
        int y = (int) m.get("y");
```

```
    Tile center = grid.get(x, y);

    // Load the tile's improvement if there is one
    if (m.containsKey("improvement")) {
      //noinspection unchecked
      Map<String, Object> improvement =
        (Map<String, Object>) m.get("improvement");
      center.improvement =
        Improvement.fromName((String) improvement.get("name"));
      //noinspection unchecked
      center.improvementMetadata =
        (Map<String, Object>) improvement.get("meta");
    } else {
      // Otherwise set the improvement to none
      center.improvement = Improvement.NONE;
    }
    return center;
  })
  .collect(Collectors.toList());
// Mark the tiles as belonging to this city
tiles.forEach(tile -> tile.city = this);

// Load the buildings the city has
//noinspection unchecked
buildings = (ArrayList<Building>) ((List<String>) map.get("buildings"))
  .stream()
  .map(Building::fromName)
  .collect(Collectors.toList());

// Load the current build of the city if there is one
if (map.containsKey("currentlyBuilding")) {
  currentlyBuilding =
    CityBuildable.fromName((String) map.get("currentlyBuilding"));
}

// Load totals from the map
productionTotal = (int) map.get("productionTotal");
citizens = (int) map.get("citizens");
excessFoodCounter = (int) map.get("excessFood");

// Load the city name
name = (String) map.get("name");

// Calculate the greatest height of all the tiles belonging to the city
updateGreatestHeight();
```

```
}

/**
 * Grow the city by the specified number of nearby tiles
 *
 * @param newTiles number of tiles to grow
 * @return the points of the tiles the city grew too
 */
public ArrayList<Point2D> grow(int newTiles) {
  final ArrayList<Point2D> grownTo = new ArrayList<>();

  // Get the center coordinate of the city
  final Point2D center = getCenter().getHexagon().getCenter();

  // Create a new queue to pull potential tiles from that sorts tiles by
  // their distance from the center
  PriorityQueue<Tile> potentialTiles = new PriorityQueue<>((a, b) -> {
    double aDist = center.distance(a.getHexagon().getCenter());
    double bDist = center.distance(b.getHexagon().getCenter());
    return Double.compare(aDist, bDist);
  });

  // Add all adjacent tiles that don't have a city to the potential tile list
  tiles.forEach(tile -> potentialTiles.addAll(
    grid.getNeighbours(tile.x, tile.y, false)
      .stream()
      .filter(adjTile -> adjTile.city == null)
      .collect(Collectors.toList()))
  );

  // Keep picking tiles from the queue until the specified number of tiles
  // have been picked
  while (newTiles > 0 && potentialTiles.size() >= newTiles) {
    // Get the next tile
    Tile tile = potentialTiles.remove();
    // Mark it as belonging to this city and add it
    tile.city = this;
    tiles.add(tile);
    // Add the grown coordinate to the list of tiles
    grownTo.add(new Point2D(tile.x, tile.y));
    newTiles--;
  }

  // Calculate the new greatest height
  updateGreatestHeight();
```

```
  // Return the list of grown to coordinates
  return grownTo;
}

/**
 * Grow the city to the tiles pointed to by the points list
 *
 * @param points list of coordinates of tiles to grow to
 */
public void growTo(ArrayList<Point2D> points) {
  for (Point2D point : points) {
    // Get the tile represented by the point
    Tile tile = grid.get((int) point.getX(), (int) point.getY());
    // Mark it as belonging to this city and add it
    tile.city = this;
    tiles.add(tile);
  }
  // Calculate the new greatest height
  updateGreatestHeight();
}

/**
 * Get the directions from the center that should have walls (that is,
 * adjacent directions that don't belong to the
 * city)
 *
 * @param tile tile to get walls from
 * @return boolean array of whether the tile should have walls in that
 * direction
 */
boolean[] getWalls(Tile tile) {
  // If this tile isn't part of the city, return an "empty" array
  if (!tiles.contains(tile))
    return new boolean[]{false, false, false, false, false, false};
  // Alias the x and y coordinates
  int x = tile.x, y = tile.y;
  // Return the array for all of the directions, checking if the tile in each
  // direction belongs to this city.
  return new boolean[]{
    !tiles.contains(grid.getTopLeft(x, y, false)),
    !tiles.contains(grid.getLeft(x, y, false)),
    !tiles.contains(grid.getBottomLeft(x, y, false)),
    !tiles.contains(grid.getBottomRight(x, y, false)),
    !tiles.contains(grid.getRight(x, y, false)),
```

```
    !tiles.contains(grid.getTopRight(x, y, false)),
  };
}

/**
 * Calculates the greatest height of a tile in the city, used for rendering
 * walls
 */
private void updateGreatestHeight() {
  greatestTileHeight = tiles.stream()
    .map(Tile::getHeight)
    .max(Double::compareTo)
    .orElse(0.0);
}

/**
 * Get the center/capital of this city
 *
 * @return Tile representing the center
 */
public Tile getCenter() {
  // The tile is always the first element added to the tile list
  return tiles.get(0);
}

/**
 * Gets the x-coordinate of the center
 *
 * @return x-coordinate of the center
 */
@Override
public int getX() {
  return getCenter().x;
}

/**
 * Gets the y-coordinate of the center
 *
 * @return y-coordinate of the center
 */
@Override
public int getY() {
  return getCenter().y;
}
```

```java
/**
 * Builds a Map containing all required information to rebuild the city
 *
 * @return Map containing city information
 */
@Override
public Map<String, Object> toMap() {
  // Get details on the health of the city (from Living)
  Map<String, Object> map = super.toMap();

  // Store tile information in the map
  ArrayList<Map<String, Object>> tileMaps = new ArrayList<>();
  for (Tile tile : tiles) {
    Map<String, Object> tileMap = new HashMap<>();
    // Store the coordinate
    tileMap.put("x", tile.x);
    tileMap.put("y", tile.y);
    // Store the improvement if there is one
    if (tile.improvement != Improvement.NONE) {
      Map<String, Object> improvementMap = new HashMap<>();
      improvementMap.put("name", tile.improvement.name);
      improvementMap.put("meta", tile.improvementMetadata);
      tileMap.put("improvement", improvementMap);
    }
    tileMaps.add(tileMap);
  }
  map.put("tiles", tileMaps);

  // Store the city owner
  map.put("owner", player.id);

  // Store the city's buildings
  map.put("buildings", buildings.stream()
    .map(CityBuildable::getName)
    .collect(Collectors.toList()));

  // Store the current building project if there is one
  if (currentlyBuilding != null) {
    map.put("currentlyBuilding", currentlyBuilding.getName());
  }

  // Store the totals
  map.put("productionTotal", productionTotal);
  map.put("citizens", citizens);
  map.put("excessFood", excessFoodCounter);
```

```
    // Store the city name
    map.put("name", name);

    return map;
  }

  /**
   * Gets the production per turn produced by this city. Calculated from the
   * number of citizens, buildings with production bonuses, and tile
   * improvements owned by the city.
   *
   * @return city's production per turn
   */
  public int getProductionPerTurn() {
    // Calculate the base production per turn from the citizen count
    int productionPerTurn = 10 + (5 * citizens);
    // Add all tile improvements to the production counter
    for (Tile tile : tiles) {
      if (tile.improvement != null) {
        productionPerTurn += tile.improvement.productionPerTurn;
      }
    }
    // Multiply the counter by all buildings with production multipliers
    for (Building building : buildings) {
      productionPerTurn *= building.productionPerTurnMultiplier;
    }
    return productionPerTurn;
  }

  /**
   * Gets the science per turn produced by this city. Calculated from the
   * number of citizens and buildings with science bonuses.
   *
   * @return city's science per turn
   */
  public int getSciencePerTurn() {
    // Calculate the base science per turn from the citizen count
    int sciencePerTurn = 5 * citizens;
    // Initialise the multiplier
    double multiplier = 1;
    for (Building building : buildings) {
      // Add to the total science per turn
      sciencePerTurn += building.sciencePerTurnIncrease;
      multiplier *= building.sciencePerTurnMultiplier;
```

```
    }
    // Apply the multiplier after all increases have been added
    sciencePerTurn *= multiplier;
    return sciencePerTurn;
  }

  /**
   * Gets the gold per turn provided by this city. Calculated from the number
   * of citizens and buildings with gold bonuses.
   *
   * @return city's gold per turn
   */
  public int getGoldPerTurn() {
    // Calculate the base gold per turn from the citizen count
    int goldPerTurn = 5 * citizens;
    // Initialise the multiplier
    double multiplier = 1;
    for (Building building : buildings) {
      // Add to the total gold per turn
      goldPerTurn += building.goldPerTurnIncrease;
      multiplier *= building.goldPerTurnMultiplier;
    }
    // Apply the multiplier after all increases have been added
    goldPerTurn *= multiplier;
    return goldPerTurn;
  }

  /**
   * Gets the food per turn for this city. Calculated from the number of
   * citizens and buildings and tiles with food bonuses.
   *
   * @return city's food per turn
   */
  public int getFoodPerTurn() {
    // Start with 5 base food per turn and subtract the number of citizens from
    // this
    int foodPerTurn = 5 - citizens;
    // Apply tile bonuses
    for (Tile tile : tiles) {
      if (tile.improvement != null) {
        foodPerTurn += tile.improvement.foodPerTurn;
      }
    }
    // Apply building multipliers
    for (Building building : buildings) {
```

```
    foodPerTurn *= building.foodPerTurnMultiplier;
  }
  return foodPerTurn;
}

/**
 * Handle a city's per turn operations
 *
 * @param game game object containing this city
 * @return tiles that have been updated this turn
 */
@Override
public Tile[] handleTurn(Game game) {
  ArrayList<Tile> updatedTiles = new ArrayList<>();

  // Get totals for resources
  int productionPerTurn = getProductionPerTurn();
  int sciencePerTurn = getSciencePerTurn();
  int goldPerTurn = getGoldPerTurn();
  int foodPerTurn = getFoodPerTurn();

  // Add the production total and check if the currently building thing can
  // now be built
  productionTotal += productionPerTurn;
  if (
    currentlyBuilding != null &&
      currentlyBuilding.canBuildWithProduction(productionTotal)
  ) {
    // If it can build it
    updatedTiles.add(currentlyBuilding.build(this, game));
    // Remove the cost from the total
    productionTotal -= currentlyBuilding.getProductionCost();
    // Reset the currently building item
    currentlyBuilding = null;
  }

  // Add/subtract food to the counter
  excessFoodCounter += foodPerTurn;
  // Calculate growth/starvation values
  double starvationValue = 10 + Math.pow(1.25, citizens - 1);
  double growthValue = 10 + Math.pow(1.25, citizens);
  // Check if the city should grow/starve
  if (citizens > 1 && excessFoodCounter < starvationValue) {
    citizens--;
  } else if (excessFoodCounter > growthValue) {
```

```
      citizens++;
      // If growing, grow the city by an extra tile and mark the grown tiles
      // for updating
      grow(1);
      updatedTiles.addAll(tiles);
    }

    // Increase global player science/gold counts by the counts for this city
    game.increasePlayerScienceBy(player.id, sciencePerTurn);
    game.increasePlayerGoldBy(player.id, goldPerTurn);

    // Return all the tiles updated this turn
    return updatedTiles.toArray(new Tile[]{});
  }

  /**
   * Handle when a unit attacks this city
   *
   * @param attacker unit that is attacking
   * @param ranged   whether the unit performed a ranged attack
   */
  @Override
  public void onAttacked(Unit attacker, boolean ranged) {
    // Mark the attack as the last attacker to this city
    lastAttacker = attacker;
    // Damage the city the correct amount
    damage(attacker.unitType.getAttackStrength());
    // If this wasn't a ranged attack, damage the attacker too
    if (!ranged) {
      attacker.damage(Math.max(attacker.getHealth() / 4, 10));
    }
  }

  /**
   * Gets the owner of this city
   *
   * @return owner of this city
   */
  @Override
  public Player getOwner() {
    return player;
  }

  /**
   * Sets the owner of this city, updating the wall and join colours
```

```
   *
   * @param player new owner of this city
   */
  public void setOwner(Player player) {
    this.player = player;
    this.wallColour = player.getColour();
    this.joinColour = this.wallColour.darker();
  }

  /**
   * Gets the position of the city center relative to the origin
   *
   * @return position of the city center
   */
  @Override
  public Point2D getPosition() {
    return getCenter().getHexagon().getCenter();
  }

  /**
   * Determines if two cities are in the same position
   *
   * @param c other city to check
   * @return if the city centers are equal
   */
  public boolean sameCenterAs(City c) {
    return getCenter().samePositionAs(c.getCenter());
  }
}
```

com/mrbbot/civilisation/logic/map/tile/Level.java

```
package com.mrbbot.civilisation.logic.map.tile;

/**
 * Enum for level types for tiles. Used to determine which colour to use to
 * render a tile.
 */
public enum Level {
  MOUNTAIN(100, 0.8, 1.0, false),
  PLAIN(1, 0.3, 0.8, false),
  BEACH(1, 0.25, 0.3, false),
  OCEAN(100, 0.0, 0.25, true);

  /**
   * Movement cost of traversing this type of tile.
```

```
   */
  public final int cost;
  /**
   * The minimum height for this type of tile
   */
  public final double minHeight;
  /**
   * The maximum height for this type of tile
   */
  public final double maxHeight;
  /**
   * Whether this type of tile should always have the maximum height (only used
   * to maintain constant water height)
   */
  public final boolean fixToMax;

  Level(int cost, double minHeight, double maxHeight, boolean fixToMax) {
    this.cost = cost;
    this.minHeight = minHeight;
    this.maxHeight = maxHeight;
    this.fixToMax = fixToMax;
  }

  /**
   * Gets the height associated with the specified height
   *
   * @param height height to check
   * @return level associated with the specified height
   */
  static Level of(double height) {
    // Get all possible levels
    Level[] levels = values();
    // Check if the height exceeds the minimum and if it does, return that
    // level
    for (Level level : levels) {
      if (height > level.minHeight) {
        return level;
      }
    }
    return levels[levels.length - 1];
  }
}
```

com/mrbbot/civilisation/logic/map/tile/Improvement.java

```java
package com.mrbbot.civilisation.logic.map.tile;

import com.mrbbot.civilisation.logic.techs.Unlockable;

/**
 * Class representing an improvement that can be built within a cities bounds.
 * Most of these are built by workers.
 */
public class Improvement implements Unlockable {
  /**
   * Base unlock ID for improvements. Used to identify improvements that can be
   * unlocked.
   */
  private static int BASE_UNLOCK_ID = 0x10;

  /*
   * START IMPROVEMENT DEFINITIONS
   */
  public static Improvement NONE = new Improvement(
    0x00,
    "None",
    0,
    0,
    0,
    false
  );
  public static Improvement CAPITAL = new Improvement(
    0x00,
    "Capital",
    0,
    0,
    0,
    false
  );
  public static Improvement TREE = new Improvement(
    0x00,
    "Tree",
    0,
    1,
    0,
    false
  );
  public static Improvement FARM = new Improvement(
    BASE_UNLOCK_ID,
    "Farm",
```

```
    2,
    2,
    0,
    true
);
public static Improvement CHOP_FOREST = new Improvement(
    BASE_UNLOCK_ID + 1,
    "Chop Forest",
    3,
    0,
    0,
    false
);
public static Improvement MINE = new Improvement(
    BASE_UNLOCK_ID + 2,
    "Mine",
    2,
    0,
    15,
    true
);
public static Improvement PASTURE = new Improvement(
    BASE_UNLOCK_ID + 3,
    "Pasture",
    4,
    4,
    0,
    true
);
public static Improvement ROAD = new Improvement(
    BASE_UNLOCK_ID + 4,
    "Road",
    2,
    0,
    0,
    true
);
/*
 * END IMPROVEMENT DEFINITIONS
 */

/**
 * Array containing all defined improvements
 */
public static Improvement[] VALUES = new Improvement[]{
```

```
  NONE,
  CAPITAL,
  TREE,
  FARM,
  CHOP_FOREST,
  MINE,
  PASTURE,
  ROAD
};

/**
 * Function to get an improvement from just its name
 *
 * @param name name of improvement to get
 * @return the improvement with the specified name or null if the improvement
 * doesn't exist
 */
public static Improvement fromName(String name) {
  // Iterates through all the improvements...
  for (Improvement value : VALUES) {
    // Check if the names match
    if (value.name.equals(name)) return value;
  }
  return null;
}

/**
 * Unlock ID used to identify an unlockable improvement
 */
public int unlockId;
/**
 * User friendly name of this improvement
 */
public String name;
/**
 * Number of turns a worker unit takes to build this improvement
 */
public int turnCost;
/**
 * Food per turn increase for a city containing the improvement
 */
public int foodPerTurn;
/**
 * Production per turn increase for a city containing the improvement
 */
```

```java
public int productionPerTurn;
/**
 * Whether a worker can build this improvement
 */
public boolean workerCanDo;

private Improvement(
  int unlockId,
  String name,
  int turnCost,
  int foodPerTurn,
  int productionPerTurn,
  boolean workerCanDo
) {
  this.unlockId = unlockId;
  this.name = name;
  this.turnCost = turnCost;
  this.foodPerTurn = foodPerTurn;
  this.productionPerTurn = productionPerTurn;
  this.workerCanDo = workerCanDo;
}

@Override
public boolean equals(Object obj) {
  if (obj instanceof Improvement) {
    // Only check the names are equal, as these should be unique
    return name.equals(((Improvement) obj).name);
  }
  return false;
}

/**
 * Gets the user friendly name of this improvement
 *
 * @return user friendly name of this improvement
 */
@Override
public String getName() {
  return name;
}

/**
 * Gets the unlock ID of this improvement
 *
 * @return unlock ID of this improvement
```

```
   */
  @Override
  public int getUnlockId() {
    return unlockId;
  }
}
```

com/mrbbot/civilisation/logic/map/tile/Tile.java

```
package com.mrbbot.civilisation.logic.map.tile;

import com.mrbbot.civilisation.geometry.Hexagon;
import com.mrbbot.civilisation.geometry.Traversable;
import com.mrbbot.civilisation.logic.unit.Unit;
import com.mrbbot.civilisation.render.map.RenderTile;
import com.mrbbot.generic.net.ClientOnly;

import java.util.HashMap;
import java.util.Map;

/**
 * Class for a tile of the map. Stored in the game's hexagon grid.
 */
public class Tile implements Traversable {
  /**
   * Hexagon this tile represents
   */
  private final Hexagon hexagon;
  /**
   * X-coordinate of this tile within the hexagon grid
   */
  public final int x;
  /**
   * Y-coordinate of this tile within the hexagon grid
   */
  public final int y;
  /**
   * Terrain object for this tile
   */
  private final Terrain terrain;
  /**
   * The city this tile is part of. May be null.
   */
  public City city;
  /**
   * This tiles improvement. Defaults to {@link Improvement#NONE}.
```

```
 */
public Improvement improvement;
/**
 * Metadata associated with the improvement (angle, strip count, width).
 */
public Map<String, Object> improvementMetadata = new HashMap<>();
/**
 * Unit currently on the tile. May be null.
 */
public Unit unit;
/**
 * Whether this tile is the currently selected tile.
 */
@ClientOnly
public boolean selected = false;
/**
 * Renderer for this tile. Only used by the client.
 */
@ClientOnly
public RenderTile renderer;

/**
 * Creates a new tile object for the specified coordinate
 *
 * @param hexagon hexagon the tile is part of
 * @param x       x-coordinate of the tile
 * @param y       y-coordinate of the tile
 */
public Tile(Hexagon hexagon, int x, int y) {
  this.hexagon = hexagon;
  this.x = x;
  this.y = y;
  this.terrain = new Terrain(hexagon.getCenter());
  // Set the improvement to a tree if there is one
  this.improvement = this.terrain.hasTree
    ? Improvement.TREE
    : Improvement.NONE;
}

/**
 * Creates a new tile object for the specified coordinates with a pre-set
 * height and tree state
 *
 * @param hexagon hexagon the tile is part of
 * @param x       x-coordinate of the tile
```

```
 * @param y       y-coordinate of the tile
 * @param height  height of the tile's terrain
 * @param hasTree whether the tile naturally has a tree
 */
public Tile(Hexagon hexagon, int x, int y, double height, boolean hasTree) {
  this.hexagon = hexagon;
  this.x = x;
  this.y = y;
  this.terrain = new Terrain(height, hasTree);
  // Set the improvement to a tree if there is one
  this.improvement = this.terrain.hasTree
    ? Improvement.TREE
    : Improvement.NONE;
}

/**
 * Gets the tiles hexagon
 *
 * @return hexagon attached to the tile
 */
public Hexagon getHexagon() {
  return hexagon;
}

/**
 * Gets the city walls for this tile
 *
 * @return boolean array containing details on whether adjacent edges belong
 * to the same city
 */
public boolean[] getCityWalls() {
  assert city != null;
  return city.getWalls(this);
}

/**
 * Gets the actual height of this tile
 *
 * @return terrain height mapped onto range [1, 3]
 */
public double getHeight() {
  return (terrain.height * 2) + 1; // 1 <= height <= 3
}

/**
```

```java
 * Gets the x-coordinate of this tile
 *
 * @return x-coordinate of this tile
 */
@Override
public int getX() {
  return x;
}

/**
 * Gets the y-coordinate of this tile
 *
 * @return y-coordinate of this tile
 */
@Override
public int getY() {
  return y;
}

/**
 * Gets the terrain object associated with this tile
 *
 * @return terrain object associated with this tile
 */
public Terrain getTerrain() {
  return terrain;
}

/**
 * Gets the cost of travelling over this tile. Returns the tile cost or 0 if
 * the tile has a road.
 *
 * @return cost of travelling over this tile
 */
@Override
public int getCost() {
  return hasRoad() ? 0 : terrain.level.cost;
}

/**
 * Checks whether a unit could actually traverse this tile
 *
 * @return traversability of this tile
 */
@Override
```

```
  public boolean canTraverse() {
    // Traversable if the level isn't an ocean or mountain, and if there isn't
    // already a unit on the tile
    return terrain.level != Level.OCEAN
      && terrain.level != Level.MOUNTAIN
      && unit == null;
  }

  /**
   * Checks whether another tile has the same position as this tile
   *
   * @param t other tile to check
   * @return whether the 2 tiles have the same position
   */
  public boolean samePositionAs(Tile t) {
    return x == t.x && y == t.y;
  }

  /**
   * Checks whether this tile has a road. Capitals have a road by default.
   *
   * @return whether this tile has a road
   */
  public boolean hasRoad() {
    // Check if the improvement is a capital or an actual road
    return improvement == Improvement.CAPITAL
      || improvement == Improvement.ROAD;
  }

  @Override
  public String toString() {
    return String.format("Tile[x=%d, y=%d]", x, y);
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovement.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.civilisation.logic.map.tile.Improvement;
import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.RenderData;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
```

```java
import java.util.ArrayList;
import java.util.Map;

/**
 * Render object for a tile's improvement. All tiles have this object, but it
 * only contains other renders if the improvement isn't equal to
 * {@link Improvement#NONE}.
 */
@ClientOnly
public class RenderImprovement extends RenderData<Improvement> {
  /**
   * Tile the improvement is situated on
   */
  private final Tile tile;

  /**
   * Creates a new render object for the specified tile's improvement
   *
   * @param tile          tile the improvement is situated on
   * @param adjacentTiles adjacent tiles to this tile
   */
  public RenderImprovement(Tile tile, ArrayList<Tile> adjacentTiles) {
    super(tile.improvement);
    this.tile = tile;
    // Set the render's initial state
    setImprovement(data, tile.improvementMetadata, adjacentTiles);
  }

  /**
   * Sets the improvement details for this render, updating what it's showing
   *
   * @param data          new improvement of the tile
   * @param metadata      metadata of the improvement, angle, size, etc
   * @param adjacentTiles adjacent tiles to the improvement's tile
   */
  public void setImprovement(
    Improvement data,
    Map<String, Object> metadata,
    ArrayList<Tile> adjacentTiles
  ) {
    this.data = data;
    // Remove all the previous children renders for the old improvement
    this.getChildren().clear();
    // Reset transformations to their default values
```

```
    this.reset();

    // Add the correct render objects for the new improvement
    if (Improvement.CAPITAL.equals(data)) {
      // Capitals automatically have a road, so add it underneath the capital
      // render
      add(new RenderImprovementRoad(tile, adjacentTiles));
      add(new RenderImprovementHouse(tile.city.wallColour));
    } else if (Improvement.FARM.equals(data)) {
      add(new RenderImprovementFarm(metadata));
    } else if (Improvement.TREE.equals(data)) {
      add(new RenderImprovementTree());
    } else if (Improvement.MINE.equals(data)) {
      add(new RenderImprovementMine(metadata));
    } else if (Improvement.PASTURE.equals(data)) {
      add(new RenderImprovementPasture());
    } else if (Improvement.ROAD.equals(data)) {
      add(new RenderImprovementRoad(tile, adjacentTiles));
    }
  }
}
```

com/mrbbot/civilisation/logic/map/tile/Terrain.java

```
package com.mrbbot.civilisation.logic.map.tile;

import com.mrbbot.civilisation.geometry.NoiseGenerator;
import javafx.geometry.Point2D;
import javafx.scene.paint.Color;

import java.util.Random;

/**
 * Class for the terrain of a tile. Stores information on the height, level,
 * and whether the tile has a tree.
 */
public class Terrain {
  /**
   * Random number generator for trees
   */
  private static final Random RANDOM = new Random();

  /**
   * Height of the tile. This is a number in the range [0, 1];
   */
  public double height;
```

```java
/**
 * Level associated with the height of the tile. Contains information on the
 * colour of the tile.
 */
public final Level level;
/**
 * Whether the tile has a tree in its natural state (regardless of tile
 * improvements that would remove it)
 */
public boolean hasTree;

/**
 * Creates a new terrain object for a specified point
 *
 * @param p point to generate terrain for
 */
Terrain(Point2D p) {
  this(
    // Height of the terrain (rounded to 3 d.p. to reduce file saves)
    Math.round(
      ((NoiseGenerator.getInterpolatedNoise(p.getX(), p.getY()) + 1) / 2)
        * 1000.0) / 1000.0,
    // Whether the tile has a tree (completely random and not dependent on
    // the position, this is ok as this will only be called once per point
    // during the generation stage)
    RANDOM.nextInt(3) == 0
  );
}

/**
 * Cretaes a new terrain object with a specified height and tree
 *
 * @param height  height of the terrain
 * @param hasTree whether the terrain has a tree
 */
Terrain(double height, boolean hasTree) {
  // Store the height
  this.height = height;

  // Get the level for the height
  level = Level.of(this.height);
  // Set the height to the maximum value if required
  if (level.fixToMax) this.height = level.maxHeight;

  // Only keep the tree if this is on the plains level (we don't want trees
```

```
    // in the ocean)
    this.hasTree = hasTree && level == Level.PLAIN;
  }

  /**
   * Calculates the colour that should be used when rendering this terrain
   * @return colour to be used for rendering
   */
  public Color getColour() {
    Color min = Color.BLACK;
    Color max = Color.BLACK;
    switch (level) {
      case MOUNTAIN:
        min = Color.GRAY;
        max = Color.WHITE;
        break;
      case PLAIN:
        min = Color.GREEN;
        max = Color.LIGHTGREEN;
        break;
      case BEACH:
        min = Color.GOLDENROD;
        max = Color.LIGHTGOLDENRODYELLOW;
        break;
      case OCEAN:
        max = new Color(0, 0.66, 1, 0.5);
        break;
    }
    // Calculate the percentage through the terrains level
    double t = (height - level.minHeight) / (level.maxHeight - level.minHeight);
    // Linear interpolate between the min and max colour
    return min.interpolate(max, t);
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementFarm.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;

import java.util.HashMap;
```

```java
import java.util.Map;
import java.util.Random;

/**
 * Render object for a farm improvement. Added to a {@link RenderImprovement}.
 */
@ClientOnly
public class RenderImprovementFarm extends Render {
  /**
   * Width/length of the farm
   */
  private static final double SIZE = 0.7;
  /**
   * Colour of the fence around the farm
   */
  private static final Color FENCE_COLOUR = Color.BROWN.darker().darker();
  /**
   * Colour of the grass strips in the farm
   */
  private static final Color GRASS_COLOUR = Color.GREEN;
  /**
   * Colour of the soil strips in the farm
   */
  private static final Color SOIL_COLOUR = Color.BROWN.darker();

  RenderImprovementFarm(Map<String, Object> metadata) {
    // Get the number of alternating strips this farm has
    double numStrips = (int) metadata.get("strips");

    // Calculate the strips' size and position
    double stripSize = SIZE / numStrips;
    double startTranslate = -(numStrips - 1) / 2.0 * stripSize;

    // Add the strips
    for (int i = 0; i < numStrips; i++) {
      Box strip = new Box(stripSize, SIZE, 0.1);
      strip.setTranslateX(startTranslate + (i * stripSize));
      strip.setMaterial(new PhongMaterial(i % 2 == 0 ? GRASS_COLOUR : SOIL_COLOUR));
      add(strip);
    }

    // Add the fences around the farm
    add(makeWall(0));
    add(makeWall(90));
    add(makeWall(180));
```

```
    add(makeWall(270));

    // Shift the farm up and rotate it by the set angle
    translate.setZ(0.05);
    rotateZ.setAngle((int) metadata.get("angle"));
  }

  /**
   * Makes a segment of the wall
   * @param angle angle to pivot the wall by in degrees
   * @return render object containing the wall
   */
  private Render makeWall(double angle) {
    // Create a render object used to pivot the wall around
    Render wallHolder = new Render();
    wallHolder.rotateZ.setAngle(angle);

    // Add the wall
    Box box = new Box(0.1, SIZE + 0.2, 0.15);
    box.setTranslateX((SIZE / 2) + 0.05);
    box.setTranslateZ(0.05);
    box.setMaterial(new PhongMaterial(FENCE_COLOUR));
    wallHolder.add(box);

    return wallHolder;
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementMine.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;

import java.util.List;
import java.util.Map;

/**
 * Render object for a mine improvement. Added to a {@link RenderImprovement}.
 */
@ClientOnly
public class RenderImprovementMine extends Render {
```

```
/**
 * Various colours the rocks can be
 */
private static final Color[] ROCK_COLOURS = new Color[]{
  Color.WHITESMOKE,
  Color.ORANGERED,
  Color.DIMGREY.darker().darker()
};

@SuppressWarnings("unchecked")
RenderImprovementMine(Map<String, Object> metadata) {
  // Get the rock sizes and colours
  List<Double> sizes = (List<Double>) metadata.get("sizes");
  List<Integer> colours = (List<Integer>) metadata.get("colours");

  // Create the 3 rocks
  for (int i = 0; i < 3; i++) {
    double size = sizes.get(i);
    int colour = colours.get(i);
    int angle = i * 120;
    add(makeRock(size, colour, angle));
  }
}

/**
 * Makes a rock for the mine render
 *
 * @param size  relative size of the rock
 * @param color colour index of the rock
 * @param angle angle the rock should be pivoted by
 * @return render object containing the rock
 */
private Render makeRock(double size, int color, int angle) {
  // Create the rock
  Box rock = new Box(
    size / 2.0,
    size / 2.0,
    size / 2.0
  );
  rock.setMaterial(new PhongMaterial(ROCK_COLOURS[color]));
  rock.setTranslateZ(size / 4.0);

  // Add it to a render object and pivot it the specified number of degrees
  Render rockHolder = new Render();
  rockHolder.rotateZ.setAngle(angle);
```

```
    rockHolder.translate.setX(0.5);
    rockHolder.add(rock);
    return rockHolder;
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementHouse.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;

/**
 * Render object for a capital improvement. Added to a
 * {@link RenderImprovement}.
 */
@ClientOnly
public class RenderImprovementHouse extends Render {
  /**
   * Roof colour of a default house
   */
  private static final Color ROOF_COLOUR = Color.BROWN.darker().darker();
  /**
   * Wall colour of a default house
   */
  private static final Color WALL_COLOUR = Color.GOLDENROD;

  RenderImprovementHouse(Color colour) {
    // Create and add the walls
    Box box = new Box(0.5, 0.5, 0.5);
    box.setTranslateZ(0.25);
    box.setMaterial(new PhongMaterial(colour == null ? WALL_COLOUR : colour));
    add(box);

    // Create and add the roof (triangular prism, cylinder with 3 divisions)
    Cylinder roof = new Cylinder(0.5, 0.7, 3);
    roof.setTranslateZ(0.25 + 0.5);
    roof.setMaterial(new PhongMaterial(
      colour == null
        ? ROOF_COLOUR
        : colour.darker())
```

```
    );
    add(roof);
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementPasture.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.generic.net.ClientOnly;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;

/**
 * Render object for a pasture improvement. Added to a {@link RenderImprovement}.
 */
@ClientOnly
public class RenderImprovementPasture extends Render {
  /**
   * Width/length of the pasture
   */
  private static final double SIZE = 0.4;
  /**
   * Colours of the pasture fences
   */
  private static final Color FENCE_COLOUR = Color.BROWN.darker().darker();

  RenderImprovementPasture() {
    // Add the pasture fences
    add(makeFence(0));
    add(makeFence(90));
    add(makeFence(180));
    add(makeFence(270));
  }

  /**
   * Creates a fence render object and pivots it the specified number of
   * degrees
   *
   * @param angle angle to pivot the fence by in degrees
   * @return render object containing this fence segment and a fence corner
   * post
   */
  private Render makeFence(double angle) {
```

```
    // Create the holder render object and pivot it
    Render fenceHolder = new Render();
    fenceHolder.translate.setX(0.3);
    fenceHolder.rotateZ.setAngle(angle + 45);

    // Create the bottom fence
    Box fence = new Box(0.01, SIZE + 0.2, 0.1);
    fence.setMaterial(new PhongMaterial(FENCE_COLOUR));
    fence.setTranslateZ(0.1);
    // Create the top fence
    Box fence2 = new Box(0.01, SIZE + 0.2, 0.1);
    fence2.setMaterial(new PhongMaterial(FENCE_COLOUR));
    fence2.setTranslateZ(0.25);
    // Create the corner fence post
    Box fencePost = new Box(0.1, 0.1, 0.35);
    fencePost.setMaterial(new PhongMaterial(FENCE_COLOUR));
    fencePost.setTranslateZ(0.35 / 2.0);
    fencePost.setTranslateY((SIZE / 2.0) + 0.1);

    // Add the fence components
    fenceHolder.add(fence);
    fenceHolder.add(fence2);
    fenceHolder.add(fencePost);

    return fenceHolder;
  }
}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementRoad.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.civilisation.logic.map.tile.Tile;
import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.transform.Rotate;

import java.util.ArrayList;

/**
 * Render object for a road improvement. Added to a {@link RenderImprovement}.
 */
public class RenderImprovementRoad extends Render {
```

```
/**
 * Colour of road segments
 */
private static final Color ROAD_COLOUR = Color.DIMGREY.darker();
/**
 * Height of the road off the ground
 */
private static final double ROAD_HEIGHT = 0.075;
/**
 * Width of road segments
 */
private static final double ROAD_WIDTH = 0.3;
/**
 * Length of road segments, from the center of tiles to the edges
 */
private static final double ROAD_LENGTH = Math.sqrt(0.75);

RenderImprovementRoad(Tile thisTile, ArrayList<Tile> adjacentTiles) {
  // Adds the center join which connects road road segments together or just
  // indicates the tile has a road if there are not adjacent connections
  Cylinder join = new Cylinder(
    ROAD_WIDTH / 2,
    ROAD_HEIGHT,
    6
  );
  join.setMaterial(new PhongMaterial(ROAD_COLOUR));
  join.setTranslateZ(ROAD_HEIGHT / 2);
  join.setRotationAxis(Rotate.X_AXIS);
  join.setRotate(90);
  add(join);

  // For every adjacent tile, checks if there is a connecting road
  for (Tile adjacentTile : adjacentTiles) {
    if (adjacentTile.hasRoad()) {
      int dx = adjacentTile.x - thisTile.x;
      int dy = adjacentTile.y - thisTile.y;

      // Calculates the pivot angle of the road segment
      int angle = 0;
      int xOffset = thisTile.y % 2;
      if (dy == 0) {
        angle = dx == 1 ? 0 : 60 * 3;
      } else if (dy == 1) {
        angle = dx == -xOffset ? 60 * 4 : 60 * 5;
      } else if (dy == -1) {
```

```
        angle = dx == -xOffset ? 60 * 2 : 60;
      }

      // Calculates the height difference for the road connector
      double heightDifference =
        adjacentTile.getHeight() - thisTile.getHeight();

      // Adds the road segment for this connection
      add(buildRoadSegment(angle, heightDifference));
    }
  }
}

/**
 * Creates a segment of road to be added to this improvement
 *
 * @param angle            pivot angle of this segment
 * @param heightDifference difference in height between this and the
 *                         connecting road
 * @return render object containing the road segment
 */
@SuppressWarnings("Duplicates")
private Render buildRoadSegment(double angle, double heightDifference) {
  // Build the render object and pivot it
  Render rotor = new Render();
  rotor.rotateZ.setAngle(angle);

  //
  //      #-----
  //      #
  // =====#
  //

  // Build the actual segment of road (the ='s in the above diagram)
  Box road = new Box(ROAD_WIDTH, ROAD_LENGTH, ROAD_HEIGHT);
  road.setMaterial(new PhongMaterial(ROAD_COLOUR));
  road.setTranslateX(ROAD_LENGTH / 2);
  road.setTranslateZ(ROAD_HEIGHT / 2);
  road.setRotationAxis(Rotate.Z_AXIS);
  road.setRotate(90);
  rotor.add(road);

  // Only render joins that go up in height
  double roadJoinHeight = Math.max(heightDifference, 0) + ROAD_HEIGHT;
  // Add the box that joins the road segment on this tile and the road
```

```
    // segment on the adjacent tile (the #'s in the above diagram)
    Box roadJoin = new Box(ROAD_WIDTH, ROAD_HEIGHT, roadJoinHeight);
    roadJoin.setMaterial(new PhongMaterial(ROAD_COLOUR));
    roadJoin.setTranslateX(ROAD_LENGTH - (ROAD_HEIGHT / 2));
    roadJoin.setTranslateZ(roadJoinHeight / 2);
    roadJoin.setRotationAxis(Rotate.Z_AXIS);
    roadJoin.setRotate(90);
    rotor.add(roadJoin);

    return rotor;
  }


}
```

com/mrbbot/civilisation/render/map/improvement/RenderImprovementTree.java

```
package com.mrbbot.civilisation.render.map.improvement;

import com.mrbbot.generic.render.Render;
import javafx.scene.paint.Color;
import javafx.scene.paint.Material;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Translate;

/**
 * Render object for a tree on the map. Added to a {@link RenderImprovement}.
 */
public class RenderImprovementTree extends Render {
  /**
   * Material used for rendering a tree's logs
   */
  private static final Material LOG_MATERIAL =
    new PhongMaterial(Color.BROWN.darker());
  /**
   * Material used for rendering a tree's bushes
   */
  private static final Material BUSH_MATERIAL =
    new PhongMaterial(Color.FORESTGREEN);
  /**
   * Height of a trees trunk
   */
  private static final double TREE_HEIGHT = 0.6;
```

```
  RenderImprovementTree() {
    // Create the tree trunk
    Cylinder log = new Cylinder(0.2, TREE_HEIGHT);
    log.getTransforms().addAll(
      new Rotate(90, Rotate.X_AXIS),
      new Translate(0, TREE_HEIGHT / 2, 0)
    );
    log.setMaterial(LOG_MATERIAL);

    // Create the tree's leaves
    Sphere bush = new Sphere(0.4);
    bush.getTransforms().add(
      new Translate(0, 0, TREE_HEIGHT + 0.2)
    );
    bush.setMaterial(BUSH_MATERIAL);

    getChildren().addAll(log, bush);
  }
}
```