Human Survival Simulation

Young Chen, Leo Foo, Lucy Zhao

Class Overview

A list of notable classes to look out for in the simulation

WorldManagement

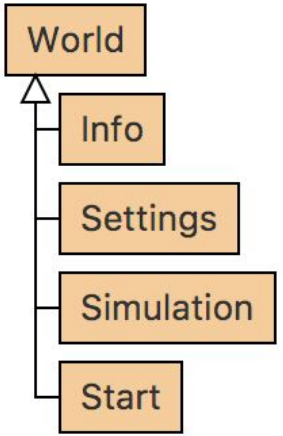
The class that manages the entire simulation. It updates sprites, data, camera movement and other important aspects of the simulation.

Utils

A set of useful and common math helper methods (distance and angle) for the simulation.

World Superclass

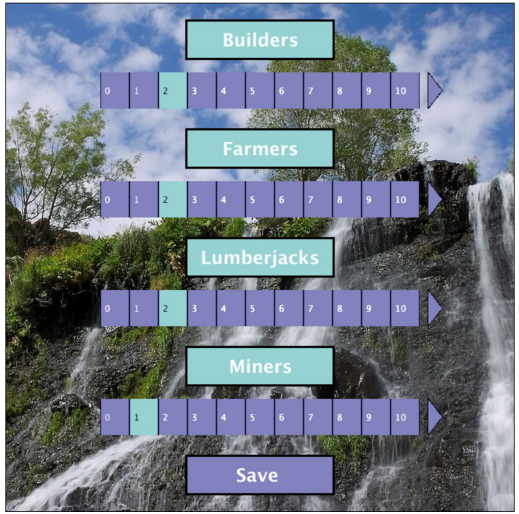
Notable subclasses: Start, Settings, Info and Simulation



Start

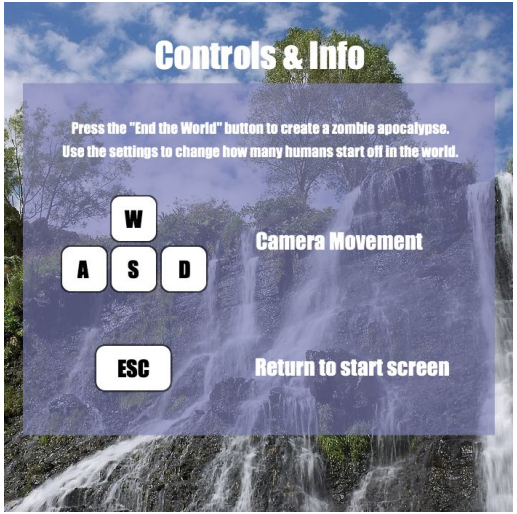
Start screen of the simulation

Settings



Allows for customization of starting humans

Info



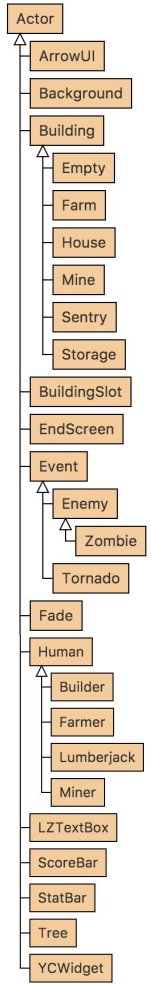
Controls and important information

Simulation

The main world where the simulation takes place. In the world, humans survive by collecting resources and building structures. Events can occur, such as a tornado or zombies, which hinder the human's ability to survive. When all humans die, the world ends. This process can be sped up by clicking the "End the World" button.

Actor Superclass

Notable subclasses: Human, Building, Event, Tree, BuildingSlot, ScoreBar, StatBar and EndScreen



Abstract

Human

Farmer

Works at farms and collects food

Miner

Works at mines and collects iron

Builder

Builds buildings for humans

Lumberjack

Chops down trees and collects wood

Abstract

Building

Farm

Produces food for humans

Mine

Produces iron for sentries

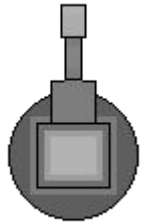
House

Raises the population and population capacity

Storage

Raises the resource capacity

Sentry



Protects humans by killing zombies

Abstract

Event

Enemy

Zombie

Targets humans and kills them. Can turn humans into zombies.

Tornado

Moves randomly across the world destroying trees, buildings and humans.

Tree

An entity that spawns randomly across the world. Drops wood when destroyed.

BuildingSlot

A valid position where buildings can be built on.It appears transparent and are randomly generated each simulation when the world is created.

ScoreBar

Keeps score of the stats of the world.

StatBar

Human HP Bar

Human Work Bar

Represents the human’s HP and work time. Credit to this class goes to Mr. Cohen.

EndScreen

When all humans die and there is not enough food to spawn another, the end screen will fade in after a few seconds.

After awhile, the world will reset, bringing the user back to start screen to run the simulation again.

Final Thoughts

Object-oriented Programming

Object oriented programming allowed us to reduce the amount of redundant code present in the project through inheritance, which in turn sped up the development of some components in our simulation.

It also brought structure into the project with the use of abstract classes, so we know what each subclass does and can be sure that the subclass will do those actions when called upon.

The polymorphic aspect of object-oriented programming also allowed us to reduce the number of data structures required to keep track of all the actors, since only one ArrayList was required for each major superclass.

Documentation, API and Class Diagrams

Documentation and writing API helps a lot in organizing and identifying what each method does.

Using the JavaDocs API format for complicated private methods was also helpful (even if it doesn’t appear in the final API Doc). We all worked on different classes, so doing that allowed us to understand the changes and/or features we added relatively easily.

As for class diagrams, we kind of left it for the end. Starting with rough class diagrams could have helped sped up the early stages of planning.

Greenfoot

One of the biggest issues we faced during the development of our simulation was with greenfoot itself, particularly with its built-in IDE and the behaviours related to the “reset” button.

The first one is that the reset button does not reset the static variables contained in each class. This led to some unexpected behaviour, like unsynced sprites and buildings being destroyed by seemingly nothing. This resulted in the initial versions of the simulation using almost none of the greenfoot api other than the world act method to start the update loop and sprite actor classes to draw the sprites on screen. This may have also been a double-edged sword, as it allowed us to easily implement a moveable camera since the project no longer relied on the actor class’ location to keep track of the objects.

The second one has to do with an IOException when playing long sound files. When a long sound file was played, it would sometimes throw an IOException, which itself didn’t affect the simulation, but also doesn’t look that good either. After some digging, one of our group members (Leo) found that it was caused by the input stream not being closed when “reset” is pressed, which meant that the file was still being read from when “run” was pressed given that the game had been reset. We were only able to solve this issue by stopping every sound that was being played when the game was stopped. (Continued in the next slide)

Leo’s Greenfoot Sound Solution (skip if too long)

When I first encountered the sound bug I thought it would be a simple google search and the forums would guide me through the problem. What ended up happening was realizing the problem had existed for many years without anyone fixing it. Another user encountering the same problem had davmac reply “there's nothing you can do about it” (link) but what surprised me was the solution to our problem was so simple that a few lines of code solved it.

Many Greenfoot threads suggested users to re-encoding sounds, removing extra file tags, add a few seconds of silence at the of the sound using audacity. None of these worked. Luckily, one of the threads provided me with another lead on what was the problem. Davmac said the Javazoom player, Jplayer, is causing the exception. At that point Greenfoot was dry of solutions so I resorted to looking through stackoverflow with my new lead. This thread and the comment by honestyrocksu specifically gave me insight on what was causing the error and how to fix it. His solution was to close the inputstream before opening it everytime. I didn’t know what bitstream, inputstream and other terms meant but I decided to take a peek at Greenfoot’s source code to confirm if they were right. The Image class did depend on bitstream for input. I played around with the “pause” and “reset” buttons and recognized a pattern where I could control the error to happen consistently. After starting the game and pressing reset the very next game would start with distorted audio and generate errors, and after pressing reset the next game would play normally and so on. I also noticed the songs continuing to play even after pressing pause. My theory was that the sound player still played sound and read input, whereas the inputstream was closed. This meant the input would be interrupted and when the next game started, it would throw an error. My solution was first to pause the song if the scenario is paused. Next after the game resets, set the song to be played equal to null so the inputstream is not interrupted but later change the inputstream back to desired song.

Greenfoot

A third issue we had with greenfoot was the lack of file management support. Although the greenfoot GUI displayed each file contained in the project folder in their corresponding structures, greenfoot does not have good support for files in the subdirectories of the project folder. For example, if we created a “simulation” folder, inside the project folder, java would still recognise the files inside it as valid and read from them, but the BlueJ IDE that greenfoot uses would not so if there were any errors in the files contained in the “simulation” folder, no debug information would be provided. This lack of file structure support overall meant that it took longer to find the desired files when sharing the project between group members, since every file was in one directory.

A fourth issue is with the BlueJ IDE, which we were forced to use because greenfoot would only compile and register the edits made in the BlueJ IDE, meaning that if we were to use a third party IDE to develop our simulation, we would need to either restart greenfoot or open every file that was edited in BlueJ each time we wanted to run our program since the changes made would not be registered and compiled. Another issue with BlueJ is that it compiles the file after every edit, which means that writing one line of code would take longer than needed due to pauses from the file compilation.

Lessons Learned

Using a collaborative platform is a good way to organize the master version of the game. For this project, we used GitHub and Git, which definitely took time to figure out, but made organising our work easier in the long run. We would recommend using GitHub, but only if all group members at least know how to use GitHub and/or it’s desktop app.

Although we initially assigned roles, such as Project Manager, we didn’t really follow through with it. We’re not sure if it would have made a huge difference in this project, but it is definitely something that could be improved in the next project.

Credits

Programmers: Young Chen, Lucy Zhao, Leo Foo

StatBar Class: Mr. Cohen

Graphics/Sprites: Lucy Zhao, Young Chen

Background: Daniel Stephens (opengameart.org)

Music/Audio: (freesound.org)

Background Music: Main Menu, Simulation and EndScreen

Sound Effects: Button, Building Destruction, Zombie One, Zombie Two, Human Hurt, Building, Mining, Chopping, Sentry Fire

That’s it!

Hope you enjoyed our simulation!