Metropolis 2127

Gameplay - basic concept

Description : A browser based game, inspired by the 1927 movie by Fritz Lang, and the novel by Thea Von Harbau.

Genre : City builder, Simulation, Adventure, Mystery.

Metropolis 2127 is a 2 Dimensional city builder game, with an extensive story and arc missions revealing the inner workings and secrets of the city as you make progress in the game.

Graphic User Interface / User Interaction :

The player looks at a 2D grid picturing the above city, and a 2D grid below picturing the below city.

On the side there are buttons to select an action, such as build a certain plot, destroy or an inspection tool.

A data screen shows levels of resources, power, food, data on residents, economy, etcetera.

A 3rd 2D grid on the side shows the Moloch / Machine, and has a mini game in it.

You can see the current state of the workers and the machine by the way they look.

If the machine is about to transform, you can either play a mini-game to help the workers, choose a random output, or choose to help them out with your mouse. Keep all the gages and meters on the right levels, and you just might avoid catastrophy!

If the crew is lost, you must click in the workers houses, and send new crew. As this new crew did not see the events accur, the secret is kept so far. Sometimes a worker escapes the Moloch, and you must choose between letting him escape, of killing him by flooding the tunnel he is in. If he escapes to the above, all is fine. Off course, no one above will believe his insane stories anyway! If however, he decides to go back to his family, the rumour will spread and this generation is lost just like the last one. If this happens to often, a riot could break out.

Technical Description and Approach to Developing the Game Mechanics:

HTML Canvas, using Javascript for 100% of Game Engine, Gameplay, Game Data, Game State.

HTML / CSS for G.U.I.

The canvas is divided in Grid Plots. To determine which plot is clicked, the following steps will take place :

1 Calculate X and Y Coordinates on Canvas.

And store these in separate variables XPos and YPos.

2 Use basic math to calculate the specific plot the user is clicking on.

At first, a plot was 10x10px, so a click in y15x27 would translate to :

YPos has 27 stored in it: YPos / 10 which would give the plot number vertically var Yplot would store = 2.

XPos has 15 stored in it: XPos / 10 which would give the plot number horizontally, var Xplot would store = 1.

So now we know it’s the 2nd in the Y-axis (vertical) and the 1st on the X-axis (horizontal)

3 Find this plot in a multidimensional array filled with an array for each plot in the grid.

I use y2x1 as a reference name for the plot as it’s the 2nd on the Y-axis and the 1st on the X-axis

(index starts at 0)

You can see it here, in the same shape as you see these plots on the canvas. Left to right, top to bottom.

aPlotCoordinates [ [x0y0, x1y0, x2y0]

[x0y1, x1y1, x2y1]

[x0y2, x1y2, x2y2] ] etcetera.

To get the array relating to the plot that was clicked, we get our values from the previous step : y2x1.

I can simply refer to aPlotCoordinates indexes and enter these 2 numbers, storing the result in a variable :

Building\_this\_plot = aPlotCoordinates [2][1]; // The y first, x second.

Now Building\_this\_plot points to the array x1y2.

All these arrays hold the specific coordinates on the HTML Canvas, which could later be used to animate on the screen, perform collision detection, etc.

I started this project in Python, so as a legacy from Tkinter, these coordinates are x1y1x2y2.

I can probably do with less data in the end, since JavaScript canvas only uses the starting point of a square.

The canvas coordinates are hardcoded in the beginning of the project :

var x0y0 = [0, 0, 10, 10]; var x1y0 = [10, 0, 20, 10]; var x2y0 = [20, 0, 30, 10];

var x0y1 = [0, 10, 10, 20]; var x1y1 = [10, 10, 20, 20]; var x2y1 = [20, 10, 30, 20];

var x0y2 = [0, 20, 10, 30]; var x1y2 = [10, 20, 20, 30]; var c2y2 = [20, 20, 30, 30];

4 Adding an eventlistener to the window can call a function to animate a square on the canvas.

window.addEventListener('mousedown', buildHouse(), false);

function buildHouse() {

building\_this\_plot = aPlotCoordinates [2][1];

// gets the array var x1y2 = [10, 20, 20, 30]

context.fillStyle = "lightgreen";

context.fillRect(building\_this\_plot[

0],building\_this\_plot[1],plot,plot);

}

Improvement : Later, I changed the occurences of the number 10 (for the size of the plot) into a variable “plot”.

Storing it’s value ofcourse, var plot = 10;

Both in XPos / 10 YPos / 10 to in XPos / plot YPos / plot, and in the hardcoded coordinates.

var x1y2 = [10, 20, 20, 30]; became

var x1y2 = [plot, plot\*2, plot\*2, plot\*3];

This little change made my whole system scalable. Adjusting the var “plot”’s value would change the size of

Data Storage on game state and calculations

Objects vs Arrays

There is a lot of data flying around, such as the attributes and properties of each plot, or that of the item build on it.

We’re talking tax properties (houses/factories bring it in, roads/hospitals bring costs) traffic input, power usage or production, water needs, etcetera. The big question is, do we use arrays or objects?

For ease of producing code, and maintainability, objects are really easy to use.

However, I’m afraid of the impact on performance, as the city will grow in size. Therefor I will mostly rely on arrays.

.