Grids

Introduction

Grids is an easy-to-use easy-to-learn game design programming language based on the use of grids. Through the application of a straightforward and elegant syntax, Grids allows creators to bring their game ideas to life in a quick and simple way. The language accomplishes this by focusing on a grid-based design that makes it easy to keep track of game objects and their locations.

Grids was born out of our love for video games and simplicity. Anyone who's ever been a kid has, at some point or another, loved a game. Whether it's board games, card games, video games or even conversation games, games are a part everyone's life. With Grids, we hope to provide a tool that anyone can use to expand their game library with only one requirement: their imagination!

Language Tutorial

Setting up Grids:

1. Install Python 3
2. Install Pillow (Python Imaging Library)
3. Download Grids.zip file from github page
4. Extract all files

Running Grids:

1. Open a terminal and move to Grids source directory (where grids.py is located)
   Ex: cd C:\Users\<Username>\Desktop\Grids\
2. Execute the following command:
   python grids.py
3. Code away!

Making your first Grids program (Tic-Tac-Toe):

1. First things first, we need to create our play area. The play area is the window where the game will run. To do this we will first create a window of size 500x500px. Afterwards will add a gameplay grid on top. To achieve this, we must run the following commands:

   Create Window(500,500)
   Create Grid(3,3)

2. Now that we have our play area, we'll want to add some sprites to our game to use as our player objects. For that, we'll do:

    Create Sprite(x.png, 150, 150)
    Create Sprite(o.png, 150, 150)

    Note: We can call x.png and o.png this way because they are located in Grids source folder, otherwise we'd have to use their file path.

3. Next we must deal with the game's winning condition. As we all know, tic-tac-toe is won by filling three consecutive slots with your corresponding player (x's or o's). In our game, that means that the player's sprite must occupy three consecutive slots. We'll check for this by adding winning positions in every possible direction as follows:

    Add WinPos((0,0), (0,1), (0,2))
    Add WinPos((1,0), (1,1), (1,2))
    Add WinPos((2,0), (2,1), (2,2))
    Add WinPos((0,0), (1,0), (2,0))
    Add WinPos((0,1), (1,1), (2,1))
    Add WinPos((0,2), (1,2), (2,2))
    Add WinPos((0,0), (1,1), (2,2))
    Add WinPos((0,2), (1,1), (2,0))

    Note: Each one of these lines define a winning line on the grid. The player must get a sprite on each of the parameter position to win by one of these rules.

4. All that is left now is to add a Controller object to receive player input, and a Draw object to handle the drawing of the sprites.

    Create Controller
    Create Draw(x.png, o.png)

5. Finally, our tic-tac-toe is ready to be played!

    Start

Language Reference Manual

Commands:

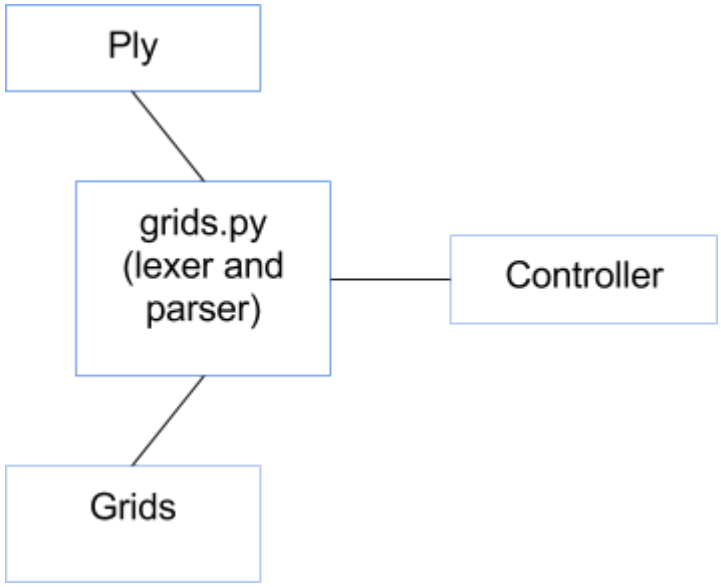
1. Create: Creates an object.
    a. Window(x, y): Object that takes care of window operations. Makes a window of width x and height y.
    b. Grid(x, y): Object that takes care of grid operations. Makes a grid consisting of x * y blocks.
    c. Sprite(“name”, x, y): Object that handles sprite operations. Makes a Sprite of width x and height y.
    d. Controller: Object that handles game logic.
    e. Draw(“sprite1”, “sprite2”): Object that handles the drawing of sprites. Player 1 is sprite1 and player 2 is sprite2.
2. Add WinPos((x,y), (x,y), (x,y) (x,y)): Adds winning position on grid.
    a. Example: “Add WinPos((0,0), (0,1), (0,2))” would make the leftmost column into a winning line for a tic-tac-toe game.
    b. This method can take 2, 3 or 4 coordinates.
3. Start: Starts the game.

Language information:

1. For a program to work it has to be written in a certain order.
    a. Create a window, grid and sprite. The window and grid objects are needed for the controller, and the sprites for the draw object.
    b. Add winning positions, they are also needed for the controller.
    c. Create a Controller object to handle input.
    d. Create a Draw object to handle the player sprites.
    e. Finally run the Start command to begin the program.
2. Images are recommended to be in the same folder as the grids.py file that way the only thing needed when creating a Sprite or Draw object is the image name. If they are not in that folder the image path is required for the program to work.

Language Development

Translator Architecture:



First and foremost, code is written in Grids. This code is then passed to the lexer in the grids.py file, where the code is tokenized. After tokenization is successful the parser comes into play (also located in the grids.py file). The parser makes sure the grammar rules are followed and takes the data from the commands. That data is used to run the intermediate python code. The main component of the intermediate code is the controller.py file which brings everything together (for more details about this read the intermediate code section).

Interfaces between the modules:

The source project contains a file named ‘grids.py’ which includes all of Grids programming language architecture. When code is written, functions inside the Controller class are executed.

The logic behind the intermediate code centers around the Controller class. The Controller class is responsible for bringing together all the different classes, such as Window, Grid, & Draw in order to execute the game according to the way it was programmed.

The Window class creates the game’s window. Its constructor requires window name and size specifications.

The Grid class creates the game’s environmental grid. Each game object (sprite) can be positioned in any grid coordinate. For example, a 3x3 grid only has 9 different coordinates. Therefore, game objects can be created and positioned only in those coordinates. Typically, complex games like Super Mario Bros. would require bigger grid sizes. Each grid cell holds information about every sprite drawn on it.

The Draw class is in charge of drawing all the game objects. To draw something, the programmer must specify the sprite to be drawn, as well as its position and its anchor. The position must be one that exists within the current grid. The anchor refers to the point inside the grid cell in which the sprite will be drawn. The positions are ‘n’, ‘s’, ‘e’, ‘w’, ‘ne’, ‘nw’, ‘se’, ‘sw’, ‘ce’, ‘cw’, & ‘center’ for north, south, east, west, north east, north west, south east, south west, center east, center west and center respectively. Every time a sprite is drawn, its information is stored inside the proper grid cell. When a sprite is erased, its information is removed from the grid cell array of gameobjects.

The Sprite class defines each game object in the game. It consists primarily of the name of the game object, its image path (file location of the image), its size, and the anchor. The anchor refers to the pivot point of the game object. Specifically, the point of origin where the image is drawn. This would be convenient in situations where the sprite’s size is bigger than the grid cell size.

The Action class handles user input. It consists of functions that return a boolean that defines whether a key or mouse event occurred.

Software development environment

Grids was developed using JetBrain’s Pycharm: a Python IDE, using Python 3.6. The parsing tool used was PLY, which is an implementation of lex and yacc. Tkinter library was used to work on Windows and graphics. But because Tkinter only supported gif images, PIL (Python Image Library) was implemented.

Test methodology

Each module was individually tested to ensure that each of them worked as expected on their own. Then modules started to be tested together to see how they would perform when interacting between each other. Once modules behaved successfully with each other, they were implemented and tested with the lexer and parser. Finally once everything was running smoothly simple games like tic-tac-toe were created to ensure the output ran correctly and as expected.

Program used to test translator

```
>>Create Window(500, 500)
>>Create Grid(3,3)
>>Create Sprite(x.png, 150, 150)
>>Create Sprite(o.png, 150, 150)
>>Add WinPos((0,0), (0,1), (0,2))
>>Add WinPos((1,0), (1,1), (1,2))
>>Add WinPos((2,0), (2,1), (2,2))
>>Add WinPos((0,0), (1,0), (2,0))
>>Add WinPos((0,1), (1,1), (2,1))
>>Add WinPos((0,2), (1,2), (2,2))
>>Add WinPos((0,0), (1,1), (2,2))
>>Add WinPos((0,2), (1,1), (2,0))
>>Create Controller
>>Create Draw(x.png, o.png)
>>Start
```

This code will create a tic-tac-toe game in a window.

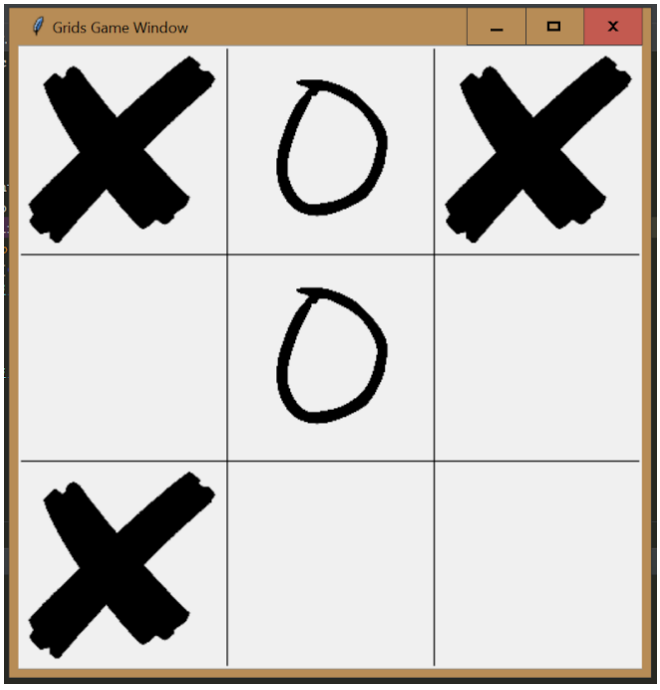


Image: Shows game created by previous code.

Conclusions

When we set out to build Grids, we had a vision in our mind that the language's "heart" would be based around a "Position-Object Type-Rule" relationship. During development, this ended up being changed for a more direct connection between each object. This was due to a lack of time given some circumstances that were out of our control. That being said, we believe that Grids, as it is now, carries within it our initial vision of what it would be. Given more time, the intermediate code could easily be polished (thanks to our development approach) and more functionality could be added. Although limited in functionality at the moment, Grids is still simple and straightforward, with a lot of potential for improvement.

The creation cycle of Grids involved a lot of planning as to how we would approach interactions. We wanted to make the language as modular as possible given the nature of game design, this however proved very challenging and a sizeable feat to face. By sacrificing some of our modular ideas for the design, we managed to cut the time needed to create Grids and meet the deadlines. This, in turn, had some impact on the amount of functionality we were able to achieve, to our dismay.

Even though development wasn't always as straightforward as we would've liked it to be, we managed to bring together something that represents what we all thought Grids should be. All in all, Grids was a fun experience to work on, and having gone through the process of creating an application oriented language from the ground up, we feel we came out the other end prepared for bigger and more demanding projects. Our only regret is that we didn't have more time to turn Grids into what it deserved to be.