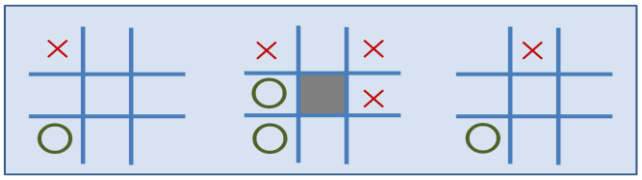
THE UNIVERSITY of NEW MEXICO

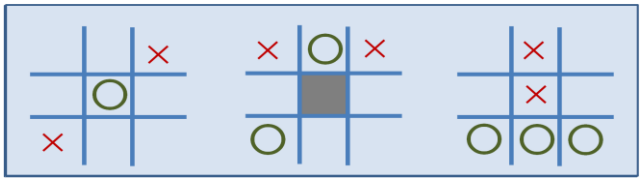
Lab 7:
3D Tic-Tac-Toe

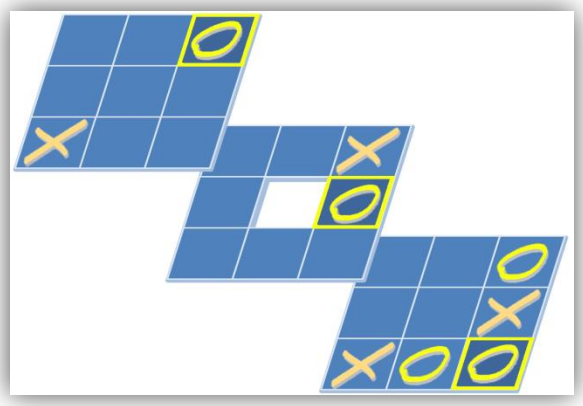
Overview:

Khan Academy has a great video that shows how to create a memory game. This is followed by getting you started in creating a tic-tac-toe game. Both games use a 2D grid or array to represent the game state.

In this lab, you will be creating a 3D game of Tic-Tac-Toe. However, you will NOT be doing 3D graphics. Think of 3D Tic-Tac-Toe as just 3 different 2D tic-Tac-Toe games. You will draw three 2D boards on the canvas and the player will make a move each turn in one of the three boards. A player wins when getting three-in-a-row in the usual way on any one of the three levels. A player can also win by getting three-in-a-row vertically, either in a single column or on a diagonal. Additionally, the middle space on the middle level is banned. For example, each board below shows a win case for player 0:







You are free to choose the style in which you draw the three layers of the 3×3×3 3D tac-tac-toe board.

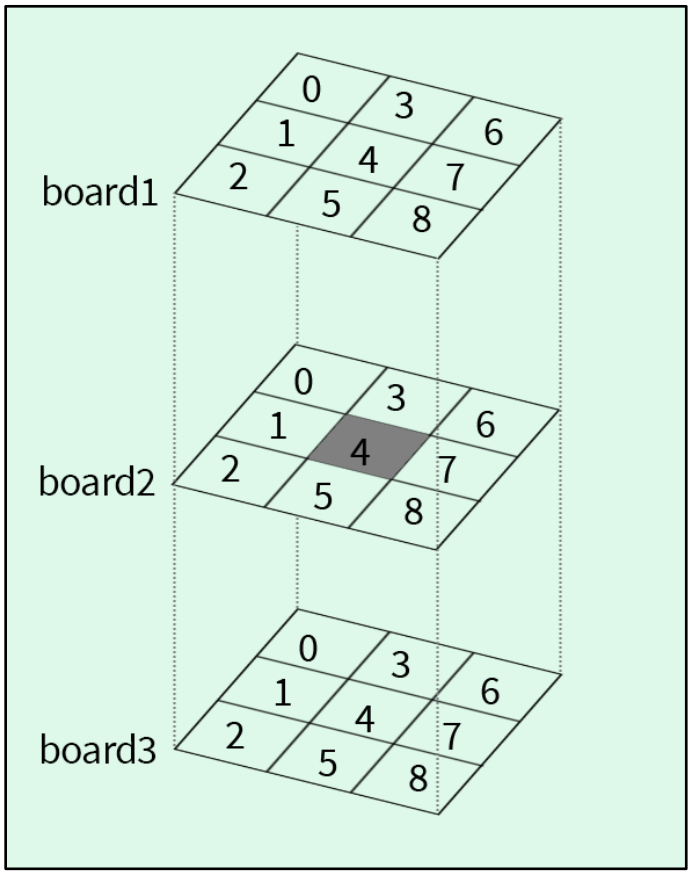
The two on the left are examples I coded in JavaScript. The first is very basic, but meets all the requirements. The second has more style, and is drawn using just the 2D JavaScript/Processing primitives we have been working with this semester. The X and O sprites are images I created in Photoshop. I have posted the image files on the website. If you want to use images, you may use those or use some third party asset or draw your own.

The second example highlights the winning cells with a darker background and brighter, thicker boarder. When the player or AI wins, your program is required to recognize and report the win, but it is not required that there is a graphic element to the reporting.

Visualizing the board:

This is a photo of a 3D tic-tac-toe game made from 4 long bolts, 4 rounded top thumb bolts, three acrylic trays, 14 amber colored marbles and 14 green colored marbles.

The whole thing assembles and disassembles in a few minutes and fits in a box that is the length of one bolt, the width of one tray and the height of a tray plus a marble.



To start off, imagine the board as a cube of layered 2d tic-tac-toe boards.

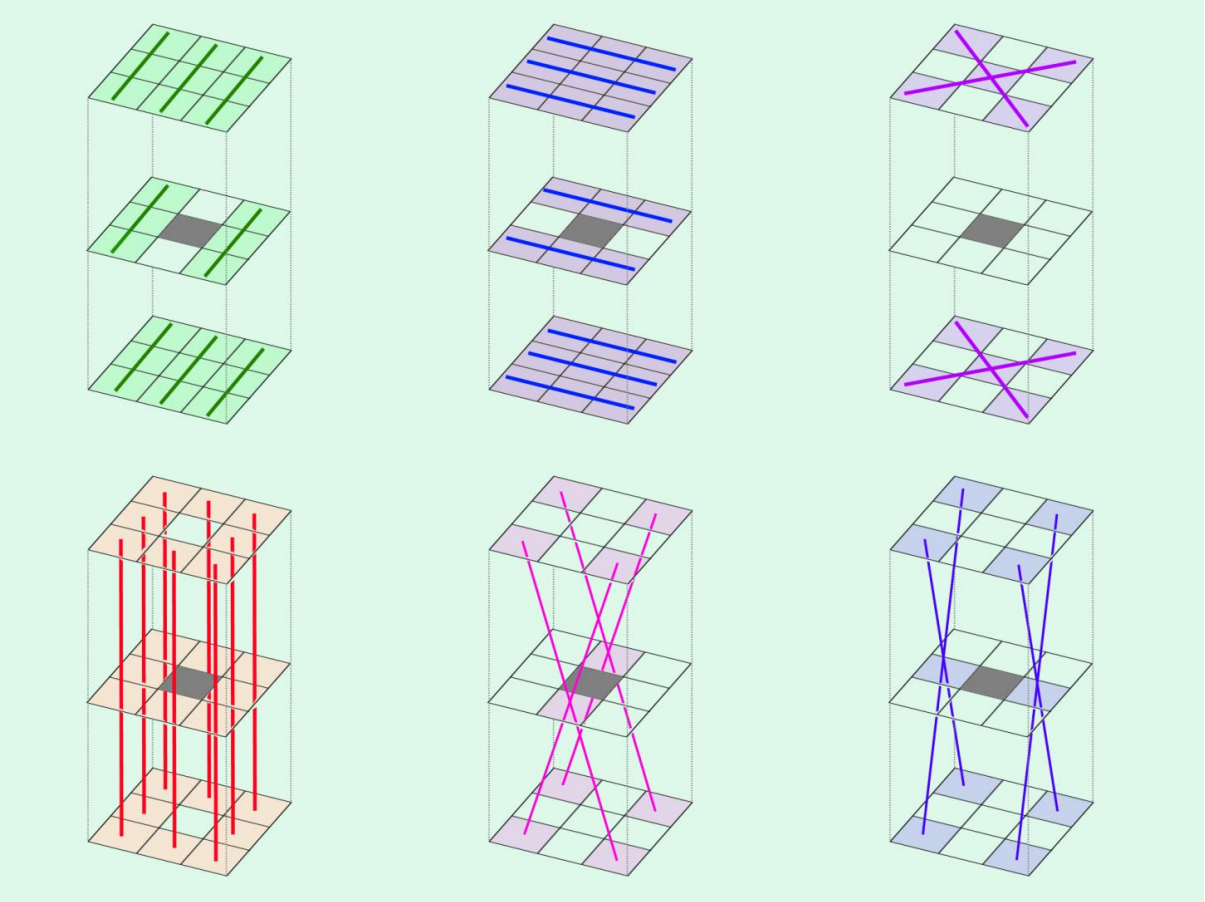
To write JavaScript code for this, it is very useful to invent a naming system that allows each space to be unambiguously identified.

There is more than one possible naming system that could work, but each programmer should pick one and use it throughout his or her program. The naming system used on the right is the system used in the example code on the class website.

This system has some nice properties that make it easy code in JavaScript. For example, on each of the three levels, cells 0, 1 and 2 are a win, as are cells 6, 7 and 8. Indeed, even 3, 4, and 5, if they all match, are a win on every level.

Of course, cell 4 on board2 is an illegal move, but the simplest way to implement that is by filling board2 cell 4 at the start of the game with something that is NOT blank, NOT X and NOT O.

The 36 Possible Ways to Win (by UNM student/lab instructor Ben Matthews):



Grading Rubric [20 points total]:

[Web Site: 3 point]: In Blackboard Learn, submit a link to a web page you create that runs your working game.

[Drawing the Board: 2 points]: Draw a set of 3 tac-tac-toe boards with the middle board missing the middle spot.

[Getting Input: 2 points]: When it is the user's turn, and the user clicks on an empty spot, your board draws an X in that spot. Be sure to use the Khan Academy video on the memory game as it will help greatly with this.

[No Cheating: 3 points]: Each AI turn, your AI must make a legal move and the game must always prevent the player from making an illegal move (For example, an illegal move is to move in a location already taken or in the center location of the middle board).

[Recognizing a Win: 4 points]: Your game must recognize when either the player or the AI has attained three-in-a-row, must report the winner and must query for a new game.

[Taking a Win: 3 points]: Your AI does not need to be super smart; however, if the AI can win on its current turn, then the AI must make a winning move.

[Blocking a Win: 3 points]: If your AI cannot win in a single move, and the player can win on his or her next move, then the AI must block at least one possible three-in-a-row that the player could make.

Spoilers:

On the class website, there are 4 sample programs we developed in class:

`TicTacToe1_KahnAcademy.html`
`TicTacToe2_CheckSomeWins.html`,
`TicTacToe3_ColorSomeWins.html`,
`TicTacToe4_AI.html`,
`TicTacToe5_MultiBoard.html`

`TicTacToe1_KahnAcademy.html` is the code you get when you work through the Khan Academy video.

The data structure used in **`TicTacToe1_KahnAcademy.html`** to represent the board is an array of **`Tile`** objects, called **`tiles`**. It is a global field populated in the **`setup`** function:

```
for (var i = 0; i < NUM_COLS; i++)
{
  for (var j = 0; j < NUM_ROWS; j++)
  {
    var x = i * (canvasWidth/NUM_COLS-1);
    var y = j * (canvasHeight/NUM_ROWS-1);
    tiles.push(new Tile(x, y));
  }
}
```

This nested loop pushes each new Tile into the tiles array such that the first element of the array, **`tiles[0]`**, is the tile in the upper right of the board. The second element of the array, **`tiles[1]`**, is in the second row of the first column, etc.:

0	3	6
1	4	7
2	5	8

Check Win Spoiler: The second, checks for **some** ways of winning and all it does is print "Winner" in the debug console. To fully make this work you can do these things:

1) Check for all the other ways to win.

2) When a win is found, have the code do two things: Display some "game over" or "winner" message on the canvas. This message could be drawn superimposed the board or above, to the side or below. Also, set a global field, say for example, **`gameOver`** to **`true`**.

3) In the given example, the **`draw()`** function always fills the canvas with the background color (erasing everything) and then redraws the board. You will want to change the **`draw()`** function to check the new **`gameOver`** field and if **`gameOver===true`**, then return without erasing or drawing anything. Also, now that draw() no longer calls drawTiles() when the game is over, you need to call drawTiles() from the place where your code finds the win. The tiles need to be drawn AFTER the winning move is made and BEFORE the "winner" message is displayed.

4) In the **`mouseReleased()`** function check the new **`gameOver`** field and if **`gameOver===true`**, then restart the game. One way to do this is to reload the page. Another way to restart the game is to loop through all the tiles, set every tile's label to the empty string ("") and set **`gameOver=false`**. This will restart the game because 1/60 of a second later, when **`draw()`** is called, the board will be erased and all the cells will have been cleared.

`TicTacToe3_ColorSomeWins.html:` This example shows how to do something that is not a requirement, but nice: when the game finds a win, it draws in bright green whichever symbols that are three-in-a-row. This is done by adding a new field to the Tile object: this.partOfAWin. When each tile object is created, this new field is set to **`false`**. When the method that checks for a win finds a win, it sets this field to **`true`** for each tile that is part of the win.

AI Kickstart Spoiler: The third sample code has some basic AI working. In this example, inside the **`mouseReleased()`** function, if and only if the player successfully makes a move (clicks in a cell that is not already occupied) , then the player has finished a move so a new function, **`makeAIMove()`** is called. The new function, **`makeAIMove()`** has a loop. Each iteration of the loop picks a random tile and tries to move there. If the tile is empty, the move is made and the loop exits.

AI GameOver Spoiler: There are three ways for the game to end: Player X gets 3 in a row, Player O gets 3 in a row, and when all cells are full. Your AI must recognize all three of these cases. For example, the given AI will get stuck in an infinite loop if all the spaces are full. It will forever keep picking random spaces and keep finding the space is not empty – over and over again (until the computer burns out or some human closes the webpage). One way recognize when the board is full is to add a global field set initially to the number of cells on the board. Then, whenever the player or the AI makes a move, decrement that field. After each move, check the value of the field. If it is zero, then there are no empty cells left, the game is over. Thus, display a "Game Over" message on the canvas and set the **`gameOver`** field to true.

AI Takes a Win Spoiler: The given kickstart AI makes a random move (on the first board only since this version only has one board). This code for making a random move should be at the end of your improved **`makeAIMove()`** function. The first part of your function should

check if a one-move win is possible and, if so, make (or block) that move. I suggest writing a new method, perhaps called :

```
checkForOneMoveWin(tileIdx, playerSymbol)
```

I suggest passing the player symbol as an argument so that the same function can be used by the AI to check if it can win in a single move and if not, it can also be used to check if the human player can win in a single move.

Later, when you add three boards, you might change this function to:

```
checkForOneMoveWin(board, tileIdx, playerSymbol)
```

This function should return true if the tile at **`tileIdx`** is empty and the given player symbol, if placed there, will make three-in-a-row. Otherwise, it should return false.

Once you have this function, update **`makeAIMove()`** to loop through each of the tiles and if the check for win function, called with the symbol 'O' ever returns true, then make the move.

AI Block Win Spoiler: This is easy, once you have the above step working. If the AI could not find a place to move to win, then repeat the same loop only call the check of win function with the player symbol, 'X'.

Multi-board Spoiler: The multi-board sample code we developed in class does some, but not all of the stuff you need:

- It draws three boards with a nice space between each.
- It creates a list of the cells in each board so that it can keep track of moves on all three boards.
- It recognizes when the player clicks in any cell of any of the three boards and draws the player symbol in the correct space of the correct board.
- It sets the center cell of the center board to a label that is not 'X', not 'O' and not empty. It then, in **`Tile.prototype.draw()`** if the cell has that invalid symbol, it draws that cell in the background color and returns.

That is a start, but there is more to do. In particular, the AI needs to move on all three boards, three-in-a-row needs to be checked on all three boards AND, having three boards makes totally new ways of making three-in-a-row (such as cell 0 of all three boards or cell 2 of all three boards, ...).

Note: the example code puts each board in a different variable: **`board1`**, **`board2`** and **`board3`**. Then, all the places where it uses **`board1`**, it also repeats with **`board2`** and

board3.That works, but it makes for lots of repeated code. If you can handle the abstraction, you can make your code much shorter by making a variable:

```
boardList = [board1, board2, board3];
```

With that, rather than repeating all the code with each board, you can just loop through the **boardList[]** array.

Extra Credit:

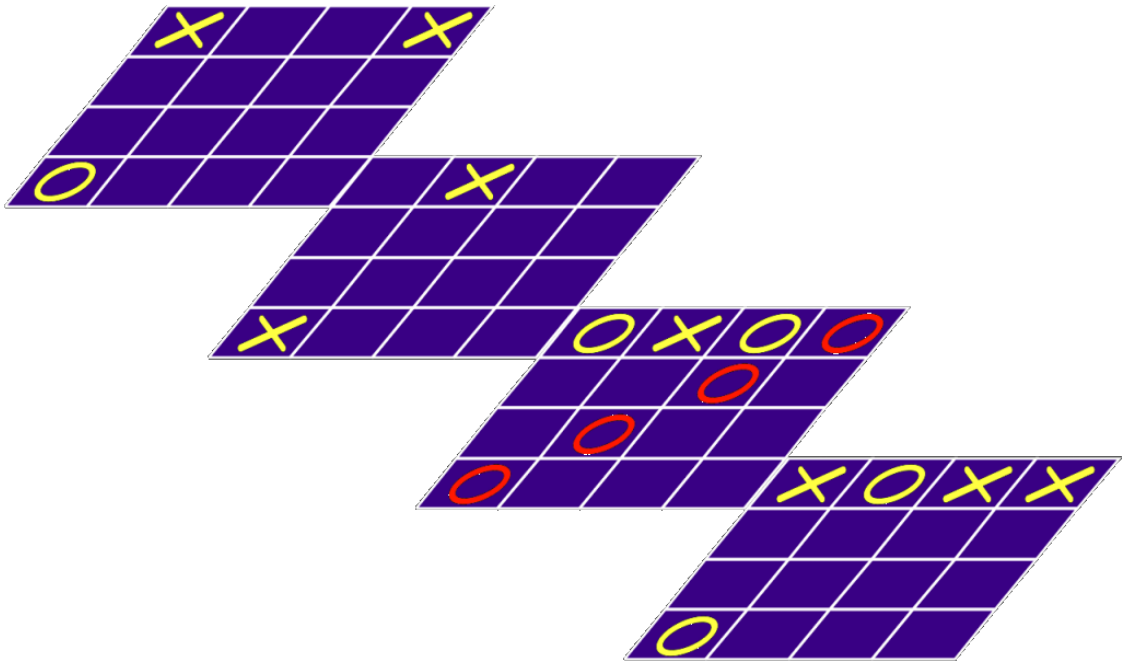
[Getting Smarter: +5]: Make your AI significantly smarter than the base requirement. Here are some suggestions for how to make your AI smarter:

1) **Play the corners first:** Just like in regular 2d tic-tac-toe, the corners are the most valuable space, because they have the most win lines that emanate out from them.
2) **Play next to a move you played before:** You can make the AI stronger by having it pick a particular direction it wants to move in and continue to play along that line until it gets blocked.
3) **Block before the last minute:** The basic AI will wait until the last move before a win to block a win line. You can make this better by searching for lines that are only missing 2 spaces as well as just one.

[Wizard: +5]: After earning the +5 for *Getting Smarter*, make your AI even smarter by adding yet another key strategy.

[Polish: up to +10]: A normal credit program needs to draw a board with lines, and shapes that get drawn in response to user actions. A well-polished program has the perfect lines and the perfect shapes that are summoned into existence not just when required, but with style to really make the user experience pop.

[4×4×4: +20]: Replace the 3×3×3 game requirements with a program that plays 4×4×4 Tac-Tac-Toe AND teach it to play at the Wizard level as defined above for the 3×3×3 game. Note, 4×4×4 Tac-Tac-Toe does not have any banded spots on any levels.



The 4×4×4 layout shown above is not required for this extra credit option. You are free to come up with your own layout. This example layout is nice in that it gives the 3D appearance without using any 3D graphics: The vertical are drawn at an angle on top of a solid blue rhombus with simple, slightly slanted images for the Xs and Os.