Level 1

Introduction to Game Structure

Riccardo Pucella

September 9, 2014

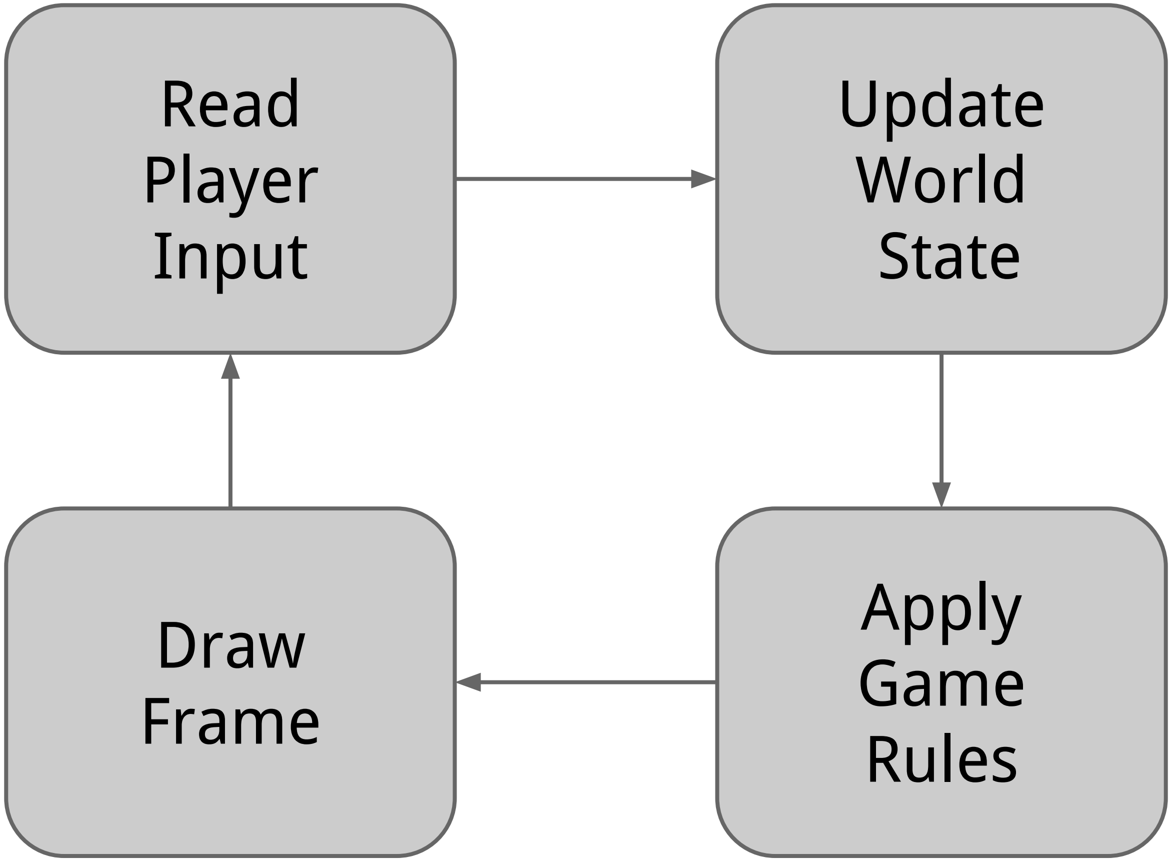
“Game programming is easy”

Game programming **is** easy

- As long as you focus on code architecture
  - That is, the “shape” of the code for the game

- First part of the course for us

Architecture: cheat sheet



Read Player Input

Update World State

Apply Game Rules

Draw Frame

Architecture: pseudocode

```
initialize

while not done:
   read_player_input
   update_world_state
   apply_game_rules
   draw_frame

if won:
   say_something_nice
```

Architecture: pseudocode

```
initialize

while not done:
    read_player_inp
    update_world_state
    apply_game_rules
    draw_frame

if won:
    say_something_nice
```

Set up world state

Architecture: pseudocode

```
initialize

while not done:
   read_player_input
   update_world_state
   apply_game_rules
   draw_frame

if won:
   say_something_nice
```

Termination condition on world state

Architecture: pseudocode

```
initialize

while not done:
    read_player_input
    update_world_state
    apply_game_rules
    draw_frame

if won:
    say_something_nice
```

May well change world state too

(often merged with previous stage)

Software engineering on a slide

Two general approaches to coding:

(at least at the unit level)

Bottom-Up

- Start coding individual functions
- Hook them together to construct program

Software engineering on a slide

Two general approaches to coding:

(at least at the unit level)

Top-Down

- Start with high-level structure and stubs
- Refine stubs into actual functionality

Top-Down: pseudocode → code

2048 clone

```
def main ():
    initialize



    while not done:
        read_player_input
        update_world_state
        apply_game_rules
        draw_frame


    if won:
        say_something_nice
```

Top-Down: pseudocode → code

2048 clone

```
def main ():
    brd = initial_board()



    while not done:
        read_player_input
        update_world_state
        apply_game_rules
        draw_frame


    if won:
        say_something_nice
```

Top-Down: pseudocode → code

2048 clone

```
def main ():
    brd = initial_board()



    while not done:
        read_player_input
        update_world_state
        apply_game_rules
        draw_frame


    if won:
        say_something_nice
```

```
def initial_board ():
    return None
```

Details to be figured out later

stub functions for now

Top-Down: pseudocode → code

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done:
        read_player_input
        update_world_state
        apply_game_rules
        draw_frame


    if won:
        say_something_nice
```

2048 clone

```
def print_board (brd):
    print “<a board>”
```

Top-Down: pseudocode → code

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        read_player_input
        update_world_state
        apply_game_rules
        draw_frame



    if won:
        say_something_nice
```

2048 clone

```
def done (brd):
    return False
```

Top-Down: pseudocode → code

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        move = read_player_input(brd)
        update_world_state
        apply_game_rules
        draw_frame



    if won:
        say_something_nice
```

2048 clone

```
def read_player_input (brd):
    return None
```

Top-Down: pseudocode → code

2048 clone

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        move = read_player_input(brd)
        brd = update_board(brd,move)

        draw_frame



    if won:
        say_something_nice
```

```
def update_board (brd,move):
    return brd
```

Top-Down: pseudocode → code

2048 clone

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        move = read_player_input(brd)
        brd = update_board(brd,move)

        print_board(brd)



    if won:
        say_something_nice
```

Top-Down: pseudocode → code

2048 clone

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        move = read_player_input(brd)
        brd = update_board(brd,move)

        print_board(brd)


    if winning_board(brd):
        print 'Congratulations!\n'
    else:
        print 'Ouch. Sorry about that...\n'
```

```
def winning_board (brd):
    return False
```

Top-Down: pseudocode → code

```
def main ():
    brd = initial_board()

    print_board(brd)

    while not done(brd):
        move = read_player_input(brd)
        brd = update_board(brd,move)

        print_board(brd)



    if winning_board(brd):
        print 'Congratulations!\n'
    else:
        print 'Ouch. Sorry about that...\n'
```

2048 clone

This + stub functions:

We have a hobbled program with (almost) all the pieces in.

If language was ~~sane~~ statically typed, program would type check.

What next?

Fleshing out stub functions?

What’s missing before we can provide more details to those functions?

What’s the one decision we have to make?

Representing the world state

Everything hinges on the world state (board)

Need to choose a representation that will facilitate implementing our functions

Often the hardest choice to make

- Depends on expected operations
- Expected operations depend on implementation of stub functions
- → *iterative process*

Repre

Everything h

Need to cho
facilitate im

Ideally, an abstract data type:

- a set of values
- an associated interface

(Actual implementation independent of the interface)

Often the harde t choice to make

- Depends on expected operations
- Expected operations depend on implementation of stub functions
- → *iterative process*

A sample world representation

For our 2048 clone, a world state is a board

A board is a 4 x 4 grid
Some cells with contained valued tiles

Expected operations:
- is there a tile in a cell of the grid?
- get the value of a tile
- add a tile (with a certain value) to the grid

A sample world representation

For our

A boar

Some

Expect

- is the
- get th
- add a

Straightforward representation:

A board is a 4x4 array

Each entry in the array is either:

0 → represents *no tile*

v → represents a tile with value *v* (v>0)

Choices:

- two-dimensional array
- one-dimensional array with calculations
- others?

Initialization

2048 clone

```
def initialize_board ():
    board = create empty board

    add INITIAL_NUMBER_OF_TILES tiles to board at random


    return board
```

Initialization

2048 clone

```
def initialize_board ():
    board = [0]*(GRID_SIZE * GRID_SIZE)

    add INITIAL_NUMBER_OF_TILES tiles to board at random


    return board
```

Initialization

2048 clone

```
def add_random_free_cell (board):
    return board
```

```
def initialize_board ():
    board = [0]*(GRID_SIZE * GRID_SIZE)

    for i in range(INITIAL_NUMBER_OF_TILES):
        board = add_random_free_cell(board)

    return board
```

Initialization

2048 clone

```
def add_random_free_cell (board):
    if 0 in board:
        free_positions = [pos for (pos,val) in list(enumerate(board))
                                  if val == 0]
        new_cell_pos = random.choice(free_positions)
        new_cell_val = (2 if random.random() < PROB_OF_2_VS_4 else 4)
        board[new_cell_pos] = new_cell_val
    return board


def initialize_board ():
    board = [0]*(GRID_SIZE * GRID_SIZE)

    for i in range(INITIAL_NUMBER_OF_TILES):
        board = add_random_free_cell(board)

    return board
```

Initialization

2048 clone

```
def add_random_free_cell (board):
    if 0 in board:
        free_positions = [pos for (pos,val) in list(enumerate(board))

        new_cell_pos     ran
        new_cell_val
        board[new_cell_
    return board


def initialize_board ():
    board = [0]*(GRID_SIZ

    for i in range(INITIA
        board = add_rando

    return board
```

End up with lots of little functions

Why is this a good thing?

Initialization

2048 clone

```
def add_random_free_cell (board):
    if 0 in board:
        free_positions = [pos for (pos,val) in list(enumerate(board))

        new_cell_pos   ran
        new_cell_val
        board[new_cell_
    return board



def initialize_board ():
    board = [0]*(GRID_SIZ

    for i in range(INITIA
        board = add_rando

    return board
```

End up with lots of little functions

Why is this a good thing?

Code reuse: we'll need to add cells to the board later

Testing: we can *unit-test* functions

Input and output

2048 clone

```
def read_player_input (board):
```

```
def print_board (board):
```

Input and output

2048 clone

```
def read_player_input (board):
    while True:
        move = raw_input('Input a move (u,d,l,r,q): ')
        if len(move) == 1 and move in 'udlrq':
            if move == 'q':
                exit(0)
            return ???



def print_board (board):
```

Input and output

2048 clone

```
def read_player_input (board):
    while True:
        move = raw_input('Input a move (u,d,l,r,q): ')
        if len(move) == 1 and move in 'udlrq':
            if move == 'q':
                exit(0)
            return move   # for now



def print_board (board):
```

Input and output

2048 clone

```
def read_player_input (board):
    while True:
        move = raw_input('Input a move (u,d,l,r,q): ')
        if len(move) == 1 and move in 'udlrq':
            if move == 'q':
                exit(0)
            return move   # for now
```

```
def print_board (board):
    The time sink to end all time sinks
```

Updating the board

2048 clone

```
def update_board (board,move):
```

Updating the board

2048 clone

```
def update_board (board,move):

    if move == ‘l’:
        board, changed = move_left (board)
    elif move == ‘r’:
        board, changed = move_right (board)
    elif move == ‘u’:
        board, changed = move_up (board)
    elif move == ‘d’:
        board, changed = move_down (board)

    if changed:
        board = add_random_free_cell(board)

    return board
```

Updating the board

```
def move_left (board):
    the only remotely interesting part of this exercise
```

2048 clone

Updating the board

2048 clone

```
def move_left (board):
    changed = False
    for j in range(1,GRID_SIZE+1):
        next_free = position(1,j)
        block_value = 0
        for i in range(1, GRID_SIZE+1):
            pos = position(i,j)
            if board[pos] > 0:
                if board[pos] == block_value:
                    board[pos] = 0
                    board[next_free+1] = block_value * 2
                    block_value = 0
                    changed = True
                else:
                    block_value = board[pos]
                    if next_free != pos:
                        changed = True
                        board[next_free] = board[pos]
                        board[pos] = 0
                    next_free += 1
    return board,changed
```

Updating the board

2048 clone

```
def move_left (board):
    changed = False
    for j in range(1,GRID_SIZE+1):
        next_free = position(1,j)
        block_value = 0
        for i in range(1, GRID_SI
            pos = position(i,j)
            if board[pos] > 0:
                if board[pos] == bloc
                    board[pos] = 0
                    board[next_free+1]
                    block_value = 0
                    changed = True
                else:
                    block_value = board[pos]
                    if next_free != pos:
                        changed = True
                        board[next_free] = board[pos]
                        board[pos] = 0
                    next_free += 1
    return board,changed
```

Convert into an index in the array representing the board

Updating the board

2048 clone

```
def move_left (board):
    changed = False
    for j in r          GRID_SIZE+1):
        next_fr              ion(1,j)
                                                     2
            else:
                block_value = board[pos]
                if next_free != pos:
                    changed = True
                    board[next_free] = board[pos]
                    board[pos] = 0
                next_free += 1
    return board,changed
```

move_right
move_up
move_down

All similar

(Yuck, can we reuse?)

Updating the board (v2)

2048 clone

```
MOVE_MAP= {  'u': move_up,
             'd': move_down,
             'l': move_left,
             'r': move_right  }

def read_player_input (board):
    while True:
        move = raw_input('Input a move (u,d,l,r,q): ')
        if len(move) == 1 and move in 'udlrq':
            if move == 'q':
                exit(0)
            return MOVE_MAP[move]

def update_board (board,move):
    board, changed = move(board)
    if changed:
        board = add_random_free_cell(board)
    return board
```

Finishing touches

2048 clone

```
def done (board):
```

```
def winning_board (board):
```

Finishing touches

2048 clone

```
def done (board):
    return (2048 in board or 0 not in board)



def winning_board (board):
```

Finishing touches

2048 clone

```
def done (board):
    return (2048 in board or 0 not in board)


def winning_board (board):
    return (2048 in board)
```