

Super 6.009 Adventure (Part 1)

Submission to website (part 1): Monday, May 8, 10PM

Checkoff by LA/TA (part 1): Thursday, May 11, 10PM

This lab assumes you have Python 3.5 or later installed on your machine. Please use the Chrome web browser.

The usual collaboration policy applies. Showing another student the web UI running your code is OK; showing them your code is not OK.

Concrete tasks to complete are marked with ➡. There are 10 coding points for this lab awarded for good organization and class hierarchy, minimizing the amount of repeated code, useful comments.

Schedule

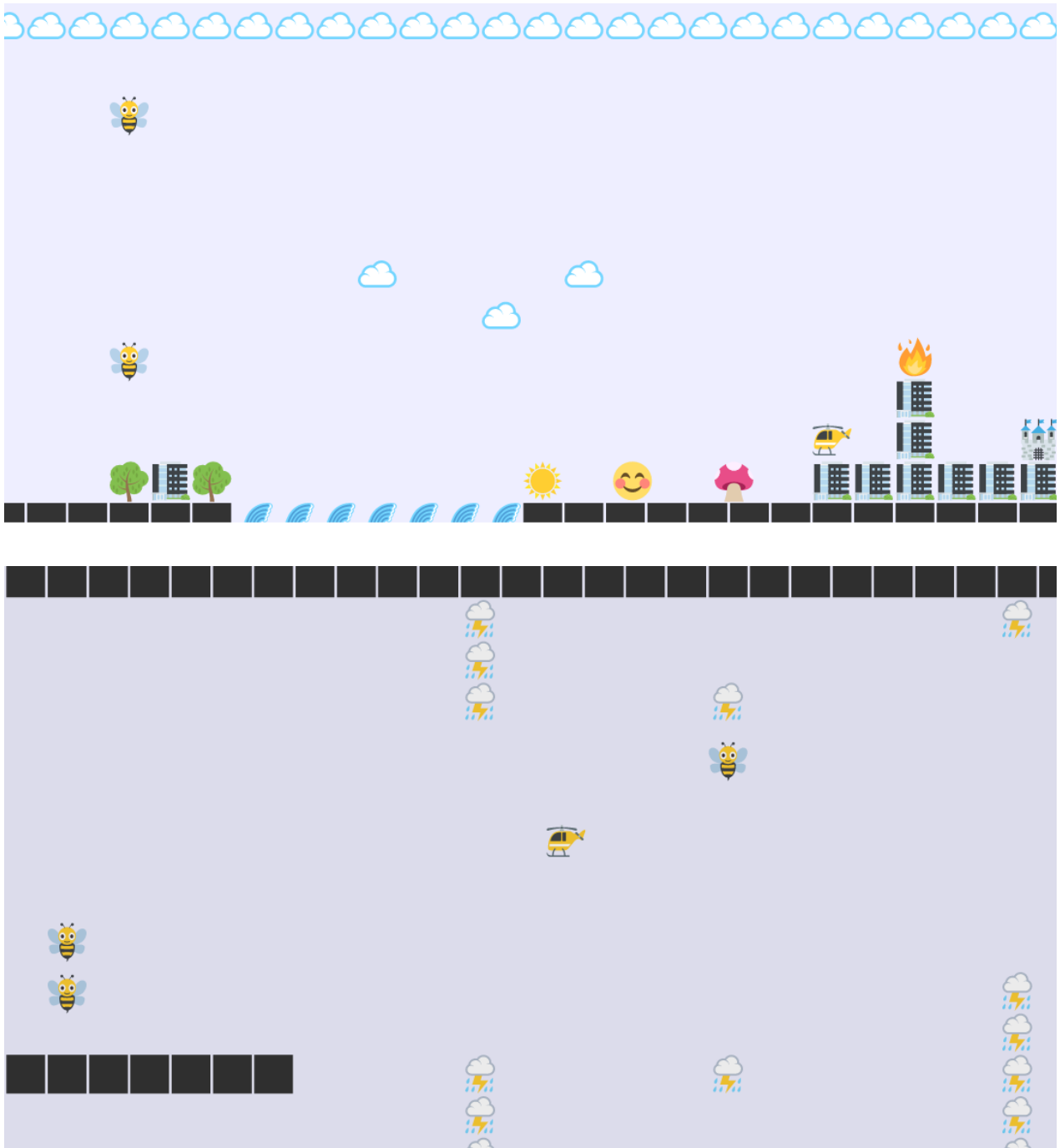
This lab is about twice as long as the first seven labs and about the same length as Lab 8; accordingly, it is distributed over two weeks:

- In week 1 (this lab), we'll implement basic collision detection and collision resolution, as well as a few simple items and basic motion.
- In week 2, we'll add faster collision detection, more blob types, nice-looking jumps, items, and foes.

This zip contains tests for the first week (tests 1 -- 22); tests for the second week will be released in the following lab.

Overview

We're building a [platformer](#)! Even better: it's a meta-platformer: you can encode a bunch of games in it, like Super Mario, Chopper, Flappy Bird, Donkey Kong, Pong, etc. Here are a few pictures of what the complete game will look like:



All these games are built out of relatively simple blocks. The core element is a *blob* (often called a *sprite*): it represents any object of the game, with a position, a size (fixed to 128x128 pixels in this lab) and a texture (determining how it looks). Here are some of the textures that we will use:



The first one is an evil mushroom; the second one is the player; the third one is a building.

A Game is just a large collection of blobs, with a global clock; every time the clock ticks, the state of the game is updated by removing, adding, updating, and moving blobs as appropriate. This makes it very simple to describe a game as a class with three methods:

- `__init__(self, levelmap)` initializes the game by parsing a game map.
- `timestep(self, keys)` advances the game state by one time step, based on user input; in other words, `timestep` is responsible for simulating the evolution of the world across one unit interval of time.
- `render(self, w, h)` renders part of the game world, returning a game status (`"ongoing"` , `"victory"` , or `"defeat"`) and a list of dictionaries describing blobs.




Of course, the actual implementation has many more functions and classes (our implementation has about 25 classes and 75 functions); but these 3 are all the UI needs to know about! We expect to see many very different solutions, depending on your personal taste, providing a concrete example of the flexibility that data abstraction provides in programming significant systems.

Our web UI should prove a powerful debugging ally; use it early and often! When a specification seems ambiguous, try to use your best guess and feel free to ask on Piazza (fully describing a game is tricky, but we have tried to restrict the tests to exactly the things detailed in the readme).

Technical overview

The world that we play the game in is an infinite grid of pixels, with `x` s increasing towards the right and `y` s increasing towards the top. Each blob is a square of 128x128 pixels, located by the coordinates of its bottom-left corner. Initial blob positions (positions of blobs when the game starts) are all multiples of the blob size.

There are two types of blobs: *soft blobs* and *hard blobs*. Hard blobs never move, are not affected by gravity, and do not interact with one another. Soft blobs may move, may be affected by gravity, can interact with each other, and may bounce off *hard* blobs. The player is a soft blob; the floor and walls are hard blobs. This distinction makes it easier to detect and handle collisions.

The physics of our world are pretty standard: each blob has a position and a speed, and may accelerate; after each time step, the blob's acceleration is added to its current speed, and the blob's coordinates are incremented by the resulting velocity. Gravity is one cause of (downwards) acceleration (it affects soft blobs but not hard blobs). User input, e.g. pressing the  key, is another cause of (horizontal) acceleration. Drag (a horizontal acceleration in the direction opposite to movement) is yet another one (it causes the player to slow down when the  and  keys are released, for example). At the end of each time step, accelerations are reset to 0 (speeds and positions persist across time steps, but accelerations do not).

Using the Web UI and debugging

The Web UI for this lab comes with a debugger. After you open a test map (`w1-tests-*` and `w2-tests-*`), you can play the `ghost mode` button before pressing `step` or `run` . This will display a ghost of the expected rendering on top of the rendering produced by your own code. This is useful to quickly spot small mistakes.

Week 1

Serializing and deserializing blobs [tests 1 -- 3]

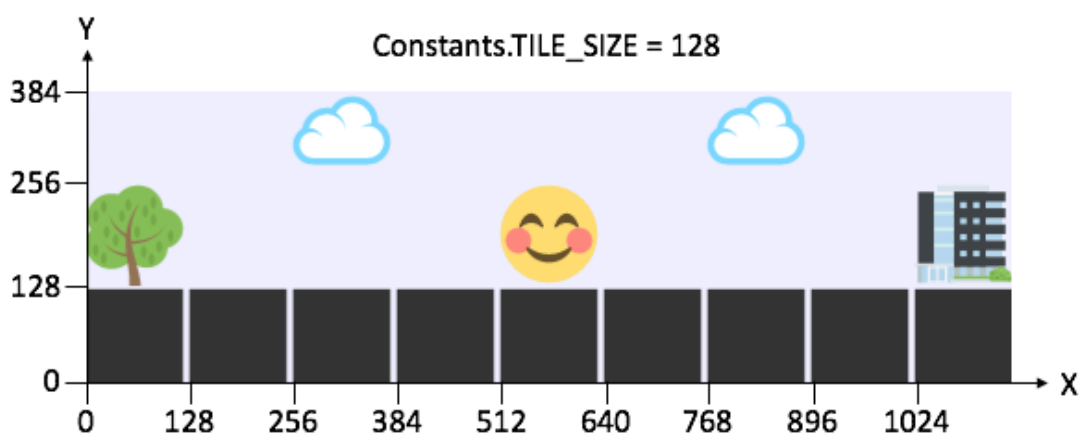
To communicate with the UI, we need to be able to read in a game map (this is called *deserializing*) and to export a list of visible blobs (this is called *serializing*).

Deserialization

Each new game is initialized from a *level map*, represented as a 2D array of characters, stored as nested lists of strings. Here's a simple example of a small map:

```
  c   c
t    p  B
=====
```

And here's what it looks like (axes and labels added to image to show coordinates -- these aren't included in the user's display):



⇒ **Implement the function** `Game.__init__` **according to its docstring.** For now, you only need to recognize two types of blobs: `f` (floor) and `p` (player) (you probably want to make sure that your code ignores unknown blob types, so that it doesn't crash when given other types of blobs). There will always be at most one player. The full list of blob types is given at the

end of this document.

To compute the absolute coordinates of each tile, use the convention that the first character of the last line in the level map is at position `(0, 0)`, and remember that each blob is `Constants.TILE_SIZE x Constants.TILE_SIZE` pixels in size, so in the image above the player is at coordinates `(512,128)` since `Constants.TILE_SIZE` is 128.

Serialization

To render a scene, the UI needs to know which blobs are in it. It expects `Game.render(w, h)` to return a tuple of two values:

1. A string indicating whether the game is ongoing (`"ongoing"`, `"victory"`, or `"defeat"`). Here in Week 1 the status is always `"ongoing"` until we reach test 20.
2. A list of blobs, stored as *blob dictionaries*.

Here's an example blob dictionary:

```
{'identifier': 22,    # Unique blob ID
 'texture': '1f60a', # Texture (one of the ones listed in the Textures class)
 'pos': [512, 128],  # Pos (x, y) in absolute coords
 'player': True      # boolean indicating if this the Player blob
}
```

Dictionaries should be provided for each blob that is visible in a window of size `(w, h)` with the player positioned in the center horizontally. A blob will be visible if any portion of the blob falls in the window, so if the player's X-coordinate is `px`, then a blob at coords `(bx,by)` would be included in the rendering if

- `px - w//2 - Constants.TILE_SIZE < bx < px + w//2`
- `0 <= by < h`

Notice the identifier field: this should be a number that uniquely identifies this blob, which allows the UI to track blobs across frames of an animation. `texture` also deserves more explanation: each blob has a character-based representation used for maps, and, in addition, it has a unicode character code used for rendering the appropriate emoticon character. Your code must include the unicode character code in the dictionaries produced by render. You will find the `Constants.TEXTURE_MAP` dictionary useful.

⇒ **Implement the function** `Game.render`. The view should follow the player horizontally, centered around the player's position. This means that if the UI requests a window of size `(w, h)`, and the player is at position `(px, py)`, then you should return dictionaries for all blobs (possibly partially) contained in a rectangular `(w, h)` window whose bottom-left corner is at

`(px - w // 2, 0)` (remember that `y = 0` corresponds to the bottom of the last line of the level map). In other words, you should return dictionaries for all blobs that strictly overlap with the window.

Note that our test suite has no idea of how you keep your game state. Because of this, we can't test your functions individually. Instead, we test `__init__` and `render` together. If you have trouble with the tests, remember that you need to return absolute coordinates, and that you should not include blobs that fall outside the window. Be careful to include partially visible blobs too (otherwise, you'll run into surprising test failures later on).

Errors are reported when unexpected values are found in the dictionaries returned by `render`. For example:

```
## Found in your rendering, but unexpected:
[{'identifier': 1, 'texture': '2b1b', 'pos': [0,0], 'player': False }]
```

might indicate one of two errors: the blob's position was incorrect, or that the blob should not have been included in the rendering since it falls outside the window (or, in Week 2, the blob has been removed from the game). The tests will also report blobs that should have been rendered, but weren't.

NOTE: start by debugging the player's position if that's reported as incorrect. Since `render` chooses which blobs to include based on the player's position, if the player's position is wrong, some blobs may have been inappropriately included/excluded from the list of blob dictionaries and you'll see additional error reports for those blobs. Once the player's position is correct, then errors reported for other blobs can be dealt with.

Once you pass the `w1-tests-*-init-render` tests, you can start using the UI! Many items will be missing, however, since you only deserialize floor and player blobs. If you're curious to see our more advanced levels, you can add the missing blob types right now (see the list at the end of the document).

Free fall [tests 4 -- 5]

Gravity is an important aspect of our basic physics simulation: when a soft blob affected by gravity isn't resting against a hard blob, it falls, pulled by gravity. Concretely, this means that on each time step, the blob's vertical velocity is decreased by the value of `Constants.GRAVITY`, and the position is then adjusted by adding the velocity to it (you may recognize this process as a simplified take on Euler's method). Air drag prevents downward speeds from rising (excuse the pun) above `Constants.MAX_DOWNWARDS_SPEED`.

Here's an example:

```
Constants.GRAVITY = -9
Constants.MAX_DOWNWARDS_SPEED = 48
Initial state: y_speed = 0; y = 500
After 1 step: y_speed = -9; y = 491
After 2 steps: y_speed = -18; y = 473
After 3 steps: y_speed = -27; y = 446
After 4 steps: y_speed = -36; y = 410
After 5 steps: y_speed = -45; y = 365
After 6 steps: y_speed = -48; y = 317
After 7 steps: y_speed = -48; y = 269
After 8 steps: y_speed = -48; y = 221
```

⇒ **Implement gravity as part of the `timestep` function.** The blob table at the end of the document describes which blobs are subject to gravity. To pass the first batch of tests, you only need to support floors (they don't feel gravity) and the player (it does feel gravity).

This is a great time to start pressing the "run" button in the UI! With a bit of luck, things will start falling around. Of course, it will look better if you add support for more blob types than just the player and the floor!

A few more blobs

⇒ **Add serialization, deserialization, and gravity support for clouds, buildings, and trees.** (All three are hard blobs that do not feel gravity.)

Events [tests 6 -- 9]

As it stands, the game isn't very exciting yet. The two missing ingredients are user input and collisions. Let's start with user input.

On each time step, your `timestep` function receives a list of currently pressed keys. These keys are strings: ASCII letters, or one of "up", "left", "down", "right". This list of keys determines whether the main character is performing a specific action, moving, etc.

Here are the basic keys and what they do (this lab is open-ended: you can add your own! We'll discuss this more in week 2):

- `up` : Jump; set vertical speed to `Constants.PLAYER_JUMP_SPEED` (we'll refine the behavior of `up` in a later phase).
- `left` : Move left: set horizontal acceleration to `-Constants.PLAYER_HORIZONTAL_ACCELERATION` (note the minus sign!).
- `right` : Move right: set horizontal acceleration to `Constants.PLAYER_HORIZONTAL_ACCELERATION`.

Remember how gravity worked? The left and right keys work the same, with two twists.

First, in addition to acceleration due to events, and even if there are no events, the player is subject to drag acceleration. Drag ensures that the player stops shortly after direction keys are released. It is equal to `+Constants.PLAYER_DRAG` if the player is going left, `-Constants.PLAYER_DRAG` if the player is going right, and 0 otherwise. If the drag is greater (in magnitude) than the player's speed, the drag should be reduced to match the player's speed. One more tricky aspect of drag is that it is computed *after* applying speed changes due to direction keys. Concretely:

```
Assume PLAYER_DRAG = 6
If x speed is...   Then drag is...
25                 -6
-32                6
6                  -6
4                  -4
-2                 2
0                  0
```

Here are two more examples:

- Assume player's `x` speed is `4`, `PLAYER_HORIZONTAL_ACCELERATION = 16`, and `PLAYER_DRAG = 6`. Then if the right key is pressed, the player's `x` speed becomes (ignoring drag) 20; the drag force is then computed based on this new speed, yielding -6, and applied; the final speed is thus 14. Note how this differs from computing the drag value from the original speed (hence the correct value here is indeed 14, and not 16).
- Similarly, assume player's `x` speed is `36`, `PLAYER_HORIZONTAL_ACCELERATION = 16`, and `PLAYER_DRAG = 6`. Then if the right key is pressed, the player's `x` speed becomes (ignoring drag) 52; the drag force is then computed based on this new speed, yielding -6, and applied; the final speed is thus 46.

Second, horizontal player speed is capped (in magnitude) to

`Constants.PLAYER_MAX_HORIZONTAL_SPEED`. This capping happens last, after applying acceleration due to events if any, and after computing and applying drag.

⇒ **Implement event handling.**

With this feature implemented, you should be able to start controlling your fall! And since we haven't implemented checks to make sure that we can only jump when touching the floor (yet!), you should be able to simply fly around. :)

Collisions [tests 10 -- 18]

Collision detection and resolution are the last significant bit that we need to implement. Together, these two pieces ensure that blobs don't simply fall through each other. This is a well-studied and active research area, with countless algorithms and strategies ([here's a surprisingly comprehensive list](#)). We will implement a very simple version, based on minimum distance resolution. We'll proceed in two steps, and we'll implement something a bit more general than what we actually need for the game. The lab contains a `Rectangle` class; that's the one we'll be modifying now.

Collision detection

⇒ **Implement the function** `Rectangle.intersects(self, other)` . Remember that all coordinates are integer; thus, you shouldn't need floating point numbers.

⇒ **Use this implementation to simplify the part of** `Game.render` **that checks whether a blob is visible.**

Collision resolution

Our collision resolution strategy is called *a posteriori*: on each time step, we ignore collisions while we move all blobs, then detect and "resolve" the resulting collisions (collision resolution is the process of moving blobs so that they stop overlapping). Remember that we only resolve collisions between soft and hard blobs; other overlaps are permitted (that is, it's ok for the player to overlap with a soft blob, or for two enemies to overlap).

You can think of the whole process as a two-phase simulation: move everything, then move soft blobs so that they stop intersecting with hard blobs.

The concrete strategy that we use for resolving collision is "minimum L1 distance." That is, if blob `soft` intersects blob `hard` , then we want to move blob `soft` so as to minimize the sum of the displacement along each axis (horizontal and vertical). This is facilitated by the fact that our blobs are always rectangles whose sides are parallel to the axes.

⇒ **Implement the static method** `Rectangle.translationvector(r1, r2)` **according to its docstring.**

Putting it all together

The rectangle class is all we need to implement collision detection and resolution. The basic algorithm is shown below:

```
for each soft blob b1 and each hard blob b2:
    if b1 and b2 intersect:
        move b1 along the minimal translation vector to stop intersecting b2
```

⇒ Implement collision detection and resolution between soft and hard blobs in your game.

At this point, you should pass test `w1-tests-12-collisions-static` , and `w1-tests-15-collisions-fall` but not `w1-tests-13-collisions-slide-right` and `w1-tests-14-collisions-slide-left` . This is because our collision resolution algorithm works well... except for one thing (can you guess what the issue is? If not, try sliding left and right on map `w1-tests-13-collisions-slide-right` ; does something look wrong?).

To solve this problem, we change to the following algorithm:

```
# Resolve vertical collisions first
for each soft blob b1 and each hard blob b2:
    if b1 and b2 intersect:
        v = minimal translation vector for b1 to stop intersecting b2
        if v is vertical:
            move b1 along v

# Resolve horizontal collisions second
for each soft blob b1 and each hard blob b2:
    if b1 and b2 intersect:
        move b1 along the minimal translation vector to stop intersecting b2
```

⇒ Implement this improved collision resolution mechanism (note that it still isn't perfect; see [this discussion](#) about potential issues, if you're curious)

Try it in the UI! Doesn't sliding across a flat surface look much better? So why is the test suite complaining about `w1-tests-16-collisions-stairs-right` and `w1-tests-17-collisions-stairs-left` ? (Try the first one in the UI and see if something feels wrong when going down the stairs; then read on :)

Well, there's still a small problem. Hitting a hard blob should slow soft blobs down. More precisely, if the soft blob's horizontal speed is `vx` , and it hits a hard blob in such a way that the minimal translation vector is `dx, dy` with `dx != 0` , then its horizontal speed should become 0 if `vx` and `dx` have different signs (if their signs are the same, the speed should remain unchanged). The symmetrical behavior applies to the vertical dimension. Here's a handy chart:

If vx is... and vy is... and dx is... and dy is... then vx becomes... and vy becomes...					
5	7	3	0	5	7
5	7	-3	0	0	7
5	7	0	3	5	7
5	7	0	-3	5	0

⇒ Implement speed correction after collisions

Try it in the UI! Doesn't it look better?

Interlude: a change in texture [test 19]

If no user input is received for `Constants.PLAYER_BORED_THRESHOLD` steps in a row, then the texture of the player should change to `Textures.PlayerBored`, until the next keyboard event.

⇒ **Make sure that the player's texture changes appropriately when no keys are pressed for a while.**

Victory and defeat [tests 20 -- 22]

We're almost done with week 1! Each level has a castle, which indicates where the level ends.

⇒ **Implement support for the `Castle` blob (`c`); when the player collides with the castle, the game should complete.** Indicate game completion by returning `"victory"` as the first element of the tuple returned by `render`.

Falling off the board terminates the game.

⇒ **Interrupt the game if the player's `y` coordinate is `< -Constants.TILE_SIZE`.** In that case, `render` should indicate `"defeat"`, as this means the player has fallen out of the board.

If the player reaches the castle or is defeated during a given time step, that time step terminates normally, but future calls to `timestep` must not do anything. Additionally, the player's texture should change based on the game's completion: if it's ongoing, the texture is `Textures.Player`; if it's a victory, the texture is `Textures.PlayerWon`; if it's a defeat, the texture is `Textures.PlayerLost`.

⇒ **Make sure that the player's textures adjusts based on game status.**

Conclusion

This is it for week 1! Don't forget to submit your code and get a check-off on your first week's work.

Appendix: blob list

The following table lists all blob types that we ask you will implement in this lab:

Blob	Hard?	Feels gravity?	Character in maps	Textures
Bee			e	e
Building	✓		B	B
Castle	✓		C	C
Cloud	✓		c	c
Fire		✓	f	f
Fireball				F
Floor	✓		=	=
Helicopter		✓	h	h
Mushroom		✓	m	m
Player		✓	p	p (normal), b (boat), h (flying), bored , defeat , victory
Storm	✓		s	s (storm with thunder), r (rain)
Sun		✓	o	o
Tree	✓		t	t
Water	✓		w	w