

Tetris: A Software Case Study

We must first know what is required.

How can we find out?

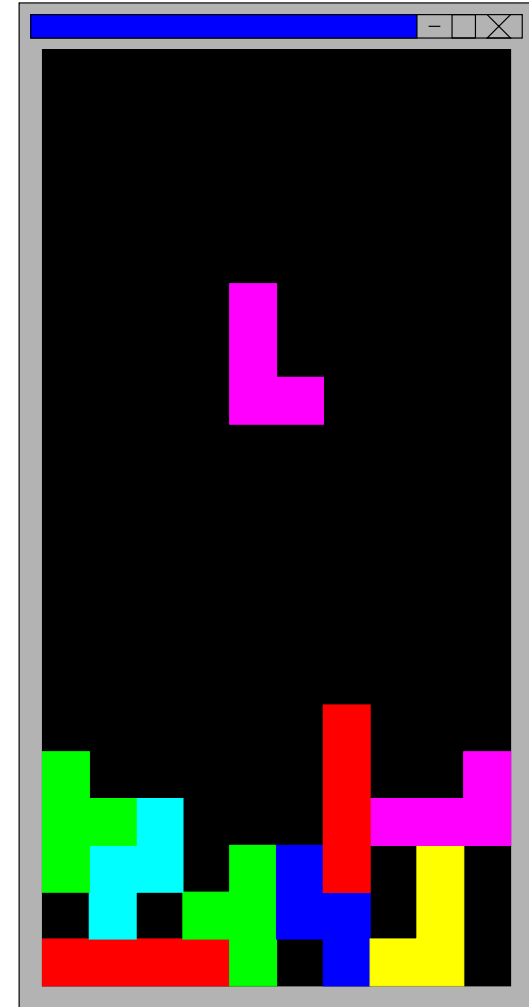
- Read a description.
- Reverse-engineer from an example.
- Discuss with client.
- ... perhaps build a prototype.

The rules of Tetris

- The *display* is rectangular, and

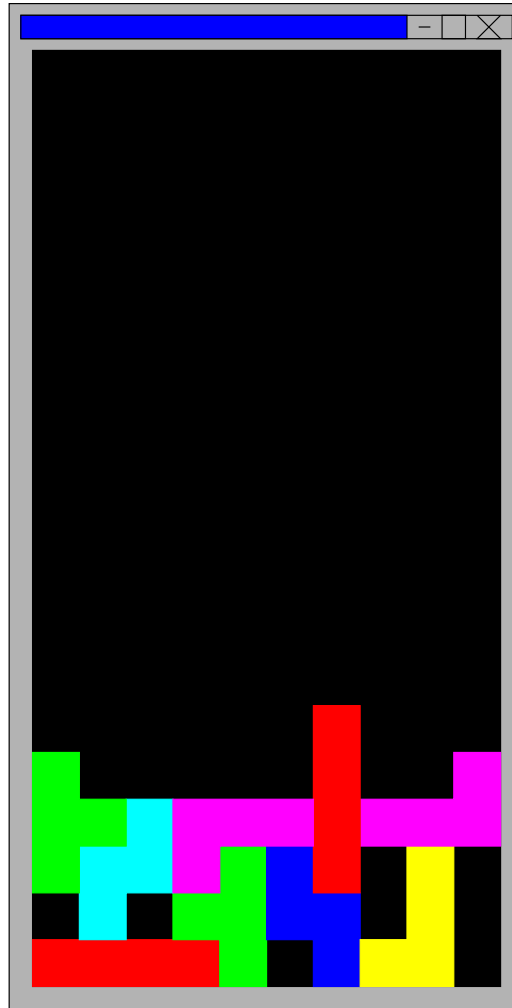
consists of x by y coloured squares.

- A *brick* is four contiguous squares of the same colour - several shapes.
- The game starts when a brick begins to 'fall' from the top of the display.
- A brick falls, pausing briefly on each row, until it 'lands'.

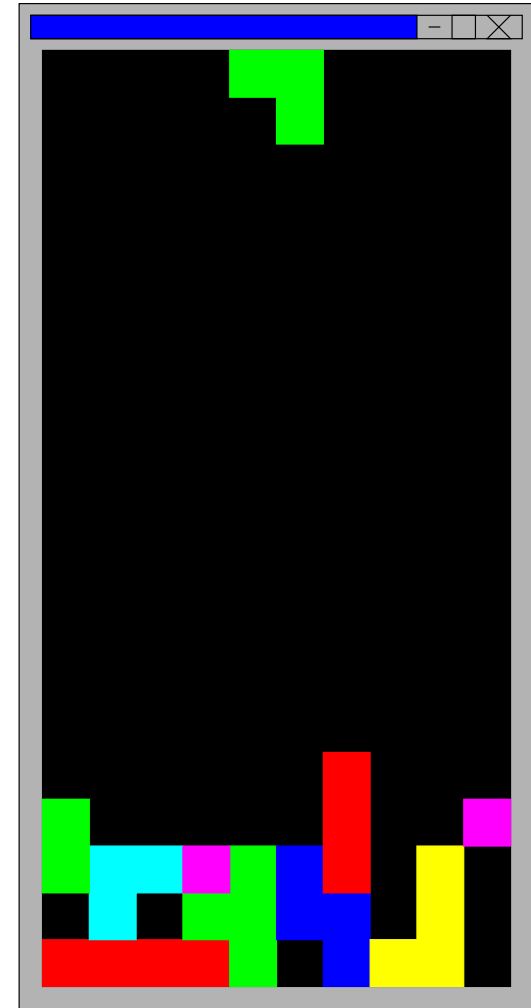


More Requirements

- After a brick lands, another will fall.
- Game ends when a brick occupies the top row.
- While a brick falls, the player can:
 - move it left/right
 - rotate it left/right
- Any full row is removed, and the rows above squash down.



A full row ...



... is removed.

Design: What Are The Classes?

Object-Oriented Design begins by considering the classes in the application.

What sort of objects have we discussed so far?

● *BRICK*

- rotate
left/right
- move
L/R/down
- landed?

● *PLAYER*

- request
rotation
- request
move

● *DISPLAY*

- field of
coloured
squares
- clear row
- check if
full to top

● *GAME*

- start
- end

Some of these — *BRICK*, *DISPLAY* — will feature in the implementation.

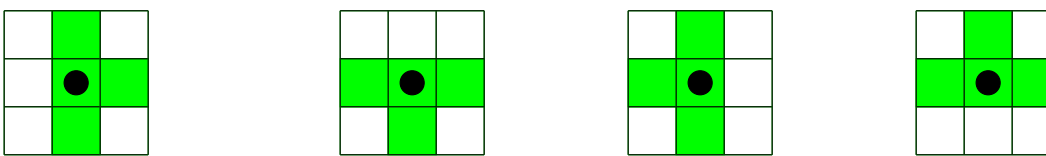
Others — *PLAYER* — will not.

Note: Although we cannot implement *PLAYER*, we will need to get requests from a *PLAYER* to a *BRICK* — we need an *INTERFACE*.

Features of a Brick

- each brick has a shape: 

- and a colour

- bricks can be rotated: 

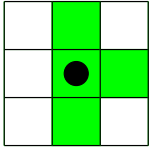
- bricks can be moved left, right, or down

- bricks land

- bricks appear on a display. No more than one brick can occupy any one square of the display.

There must be some coupling between a display and a brick.

Representing a Brick



The image could be:

- *image: ARRAY2 [COLOUR]*

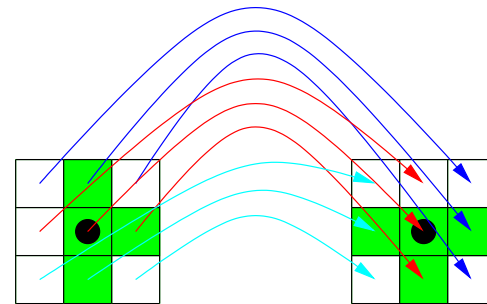
E.g., $\langle\langle\langle\langle\text{Void}, \text{Green}, \text{Void}\rangle\rangle, \langle\langle\text{Void}, \text{Green}, \text{Green}\rangle\rangle, \langle\langle\text{Void}, \text{Green}, \text{Void}\rangle\rangle\rangle\rangle$

Allows multi-coloured bricks.

- *colour: COLOUR; image: ARRAY2 [BOOLEAN]*

E.g., $\langle\langle\langle\langle\text{False}, \text{True}, \text{False}\rangle\rangle, \langle\langle\text{False}, \text{True}, \text{True}\rangle\rangle, \langle\langle\text{False}, \text{True}, \text{False}\rangle\rangle\rangle\rangle$

Enforces single coloured bricks



Rotation requires making a fresh copy of the array:

Alternative Representation – coordinate transformations

colour: COLOUR; *dx*, *dy*: ARRAY[INTEGER]

rotate (*direction*: INTEGER) **is**

local *i*, *tx*, *ty*: INTEGER

do from *i* := *dx.lower*

until *i* > *dx.upper*

loop

tx := *dx*@*i*; *ty* := *dy*@*i*

if *direction* = Left **then**

dx.put(-*ty*, *i*); *dy.put*(*tx*, *i*)

else

dx.put(*ty*, *i*); *dy.put*(-*tx*, *i*)

end

i := *i* + 1

end

end

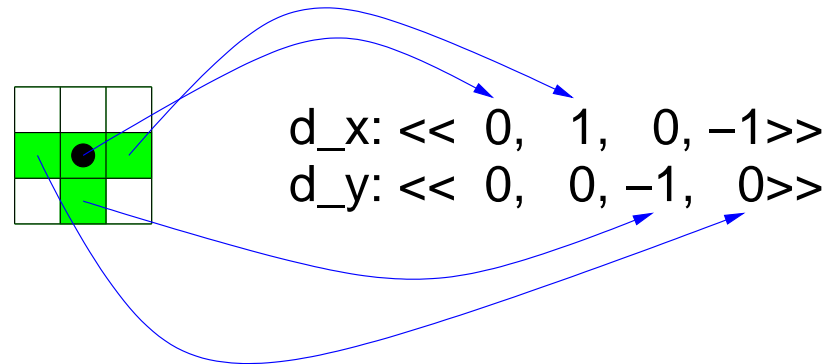
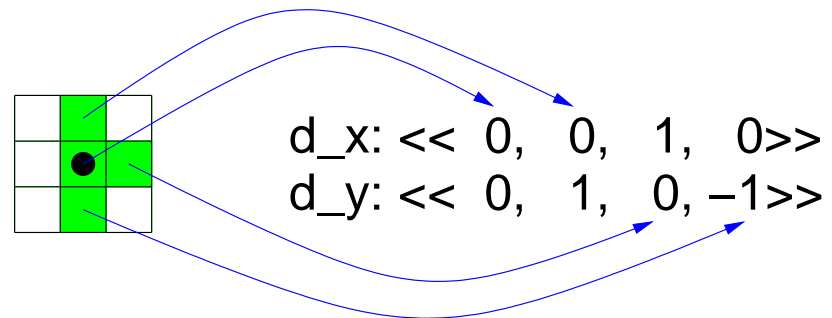


Image rotated without making a copy.

Features of the Display

- A rectangle of coloured squares.
- A command to draw a picture of the display on the screen.
- A query to check if the top row is occupied.
- A command to remove full rows.

The Brick/Display Coupling

Bricks interact closely with the display

- | | |
|---|---|
| ● the “live” brick needs to know | ● old bricks |
| ■ the edges of the display | ■ never move |
| ■ collisions with old bricks in the display | ■ may be squashed if a display row becomes full |

Which scenario to choose?

- the display knows about a brick, and can access its image and position as required.

- a brick knows about the display, and can access individual squares as required.

- The display has a brick
 - ✗ brick has public features to query the colour, position, and shape.
 - ✗ *this reveals the inner workings of the brick*
 - ✗ game needs to notify display of each new brick object created.

- The brick has a display
 - ✓ display has public features to set and query the colour of each square.
 - ✓ *the live brick can interact with squares left by old bricks*
 - ✓ brick keeps the same display for its entire lifetime.

A Reusable Class: *DISPLAY*

- queries about dimensions

- *lower1: INTEGER*

- *lower2: INTEGER*

- *valid_index1(x: INTEGER): BOOLEAN*

- *valid_index2(y: INTEGER): BOOLEAN*

- *valid_index(x, y: INTEGER): BOOLEAN*

- *upper1: INTEGER*

- *upper2: INTEGER*

- access to individual squares

- *put(c: COLOUR; x,y: INTEGER)*

- *item(x, y: INTEGER): COLOUR*

- refresh the picture on screen

- *refresh*

DISPLAY

*How should these classes
be related?*

TETRIS_DISPLAY

*make
full
pack*

A Specific Class: *TETRIS_DISPLAY*

● Option 1

```
class TETRIS_DISPLAY
creation make
feature
  full (d: DISPLAY):BOOLEAN
  pack (d: DISPLAY)
end — TETRIS_DISPLAY
```

● Option 2

```
class TETRIS_DISPLAY
creation make (d: DISPLAY)
feature
  full : BOOLEAN
  pack
end — TETRIS_DISPLAY
```

● Option 3

```
class TETRIS_DISPLAY
inherit DISPLAY
creation make
feature
  full : BOOLEAN
  pack
end — TETRIS_DISPLAY
```

✗ Option 1

There is no state, this is no class!

✗ Option 2

Other objects must have direct access to 'internal' display.

✓ Option 3

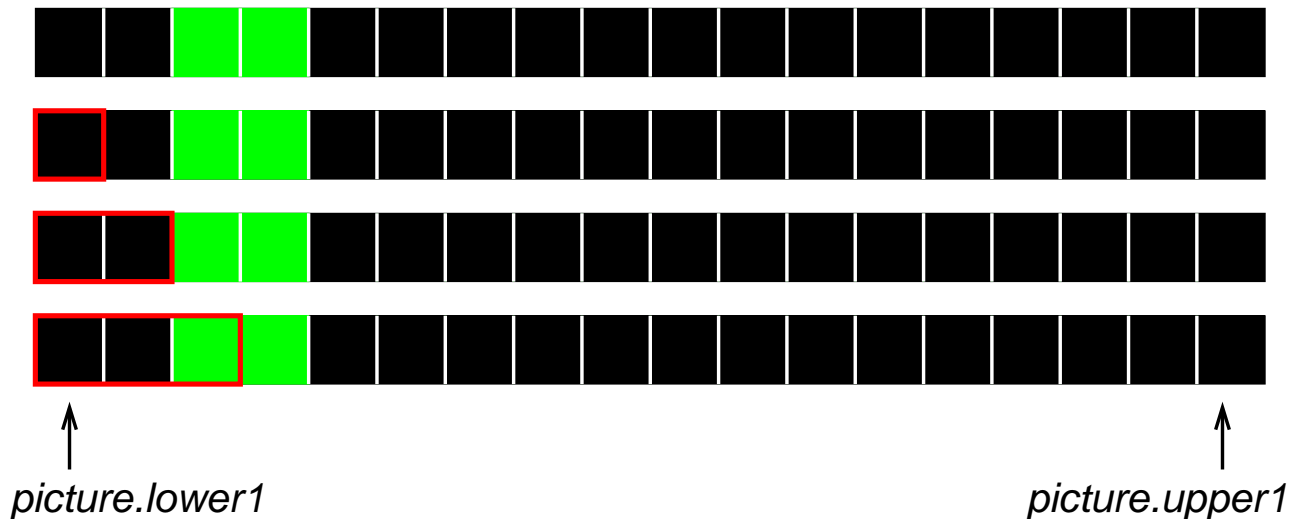
External communication is explicit.

Implementing *TETRIS_DISPLAY*: full

full : *BOOLEAN* is

— Is there a non-empty square on the top row.

Postcondition: $Result = (\exists i \in Z, picture.lower1 \leq i \leq picture.upper1 \wedge picture.item(i, picture.upper2) \neq background)$



Invariant: $Result = (\exists i \in Z, picture.lower1 \leq i \leq j \wedge picture.item(i, picture.upper2) \neq background)$

● Option 1

from

```
j := picture.lower1  
Result := picture.item  
(j, picture.upper2) /=  
background
```

● Option 2

✓ *simpler*

from

```
j := picture.lower1 - 1  
Result := False
```

● Option A

until *j = picture.upper1* **loop** *j := j + 1*

Result := Result **or**

picture.item(j, picture.upper2) /= background

end

● Option B

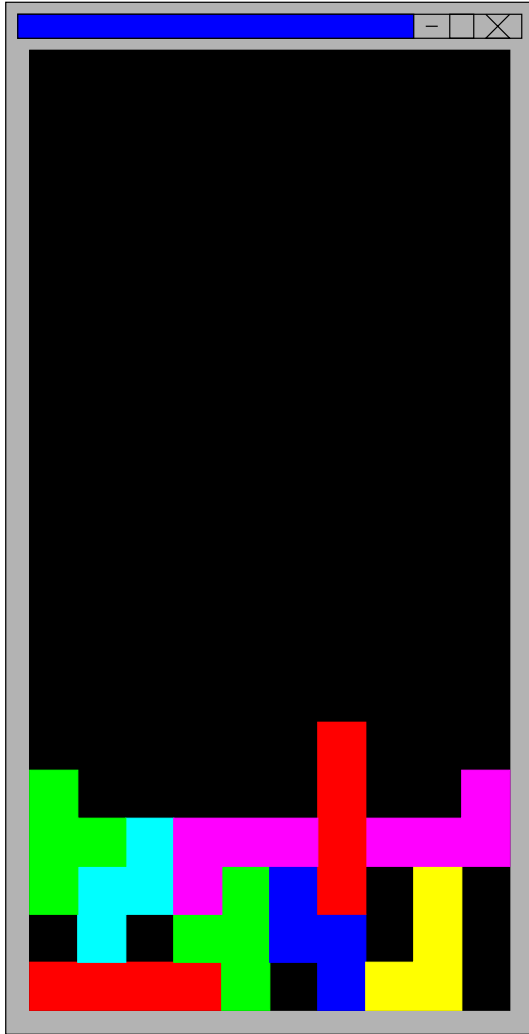
✓ *faster*

until *j = picture.upper1* **or** *Result* **loop** *j := j + 1*

Result := picture.item(j, picture.upper2) /= background

end

Implementing *TETRIS_DISPLAY*: *pack*



```
from i := picture.lower2
until i > picture.upper2
loop
  if full_row(i) then
    from j := i
    until j > picture.upper2
    loop
      copy_row(j+1, j)
      -- invalid row empty
      j := j + 1
    end
  else
    i := i + 1
  end
end
```

Assume **1.** $picture.lower2 = 1$, **2.** invalid rows are empty.

Definitions: $nonfull_index(a, i) \stackrel{\text{def}}{=} \text{Index of the } i\text{th nonfull row of } a.$

$equal_rows(a, i, b, j) \stackrel{\text{def}}{=} \text{Row } i \text{ of } a \text{ same as row } j \text{ of } b.$

Postcondition: $\forall i \in \mathbb{Z}, 1 \leq i \leq picture.upper2 \rightarrow$

$equal_rows(picture, i, old\ picture, nonfull_index(old\ picture, i))$

from

$i := 0$

$j := 1$

until $i = picture.upper2$

loop

if not $full_row(j)$ **then**

$i := i + 1$

$copy_row(j, i)$

end

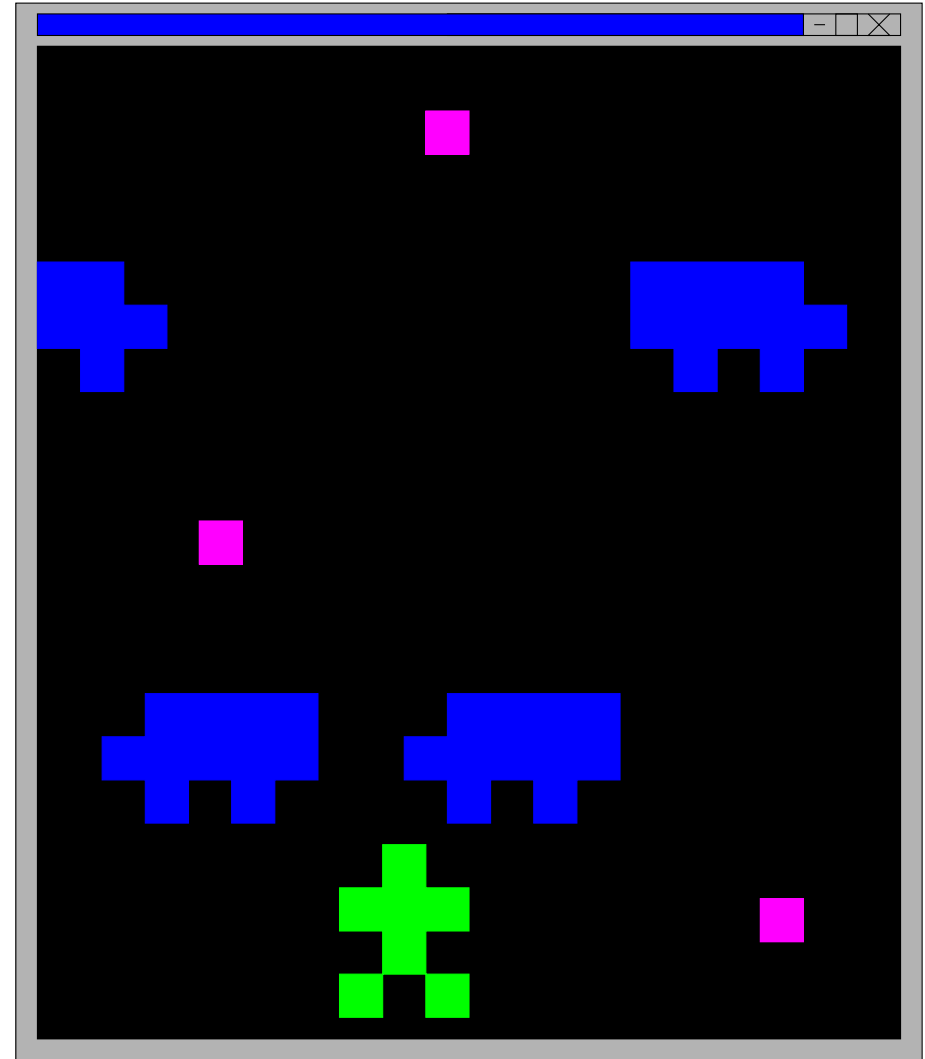
$j := j + 1$

end

Frogger: A Software Case Study

Requirements:

- A grid of coloured squares represents a road.
- There are two lanes on the road.
- Trucks move along each lane.
- There is a frog on the road.
- The player can cause the frog to turn or hop.
- Periodically, a fly will land on the road.



More Frogger Requirements

- When a truck and a fly collide, the fly is destroyed.
- When a truck and a frog collide, the frog is destroyed, and the game ends.
- When a frog and a fly collide, the fly is destroyed, and the player gains a point.

Frogger Design

Consider the possible classes:

● *ROAD*

● *LANE*

● *TRUCK*

■ *move*

● *FLY*

● *PLAYER*

● *FROG*

■ *turn_left*

■ *turn_right*

■ *hop*

■ *eat*

■ *squish*

■ *alive*

■ *score*

● The class *PLAYER* will not feature in the implementation, but perhaps *INTERFACE* will.

● The *score* feature may be more appropriate to *FROG* than it is to *PLAYER*.

● We will want to reuse the *DISPLAY* class from Tetris.

DISPLAY

- queries about dimensions

- *lower1: INTEGER*

- *lower2: INTEGER*

- *valid_index1(x: INTEGER): BOOLEAN*

- *valid_index2(y: INTEGER): BOOLEAN*

- *valid_index(x, y: INTEGER): BOOLEAN*

- *upper1: INTEGER*

- *upper2: INTEGER*

- access to individual squares

- *put(c: COLOUR; x, y: INTEGER)*

- *item(x, y: INTEGER): COLOUR*

- refresh the picture on screen

- *refresh*

A Design Challenge

We must be able to

- Detect collisions between objects on the screen. (frogs, flies, and trucks)
- Initiate appropriate actions.

If all objects share a common display, then objects can see when they hit another, but

- They don't know what they hit (except from the colour)
- They cannot send a message to the other object. E.g., a *Truck* squishes a *Frog*.

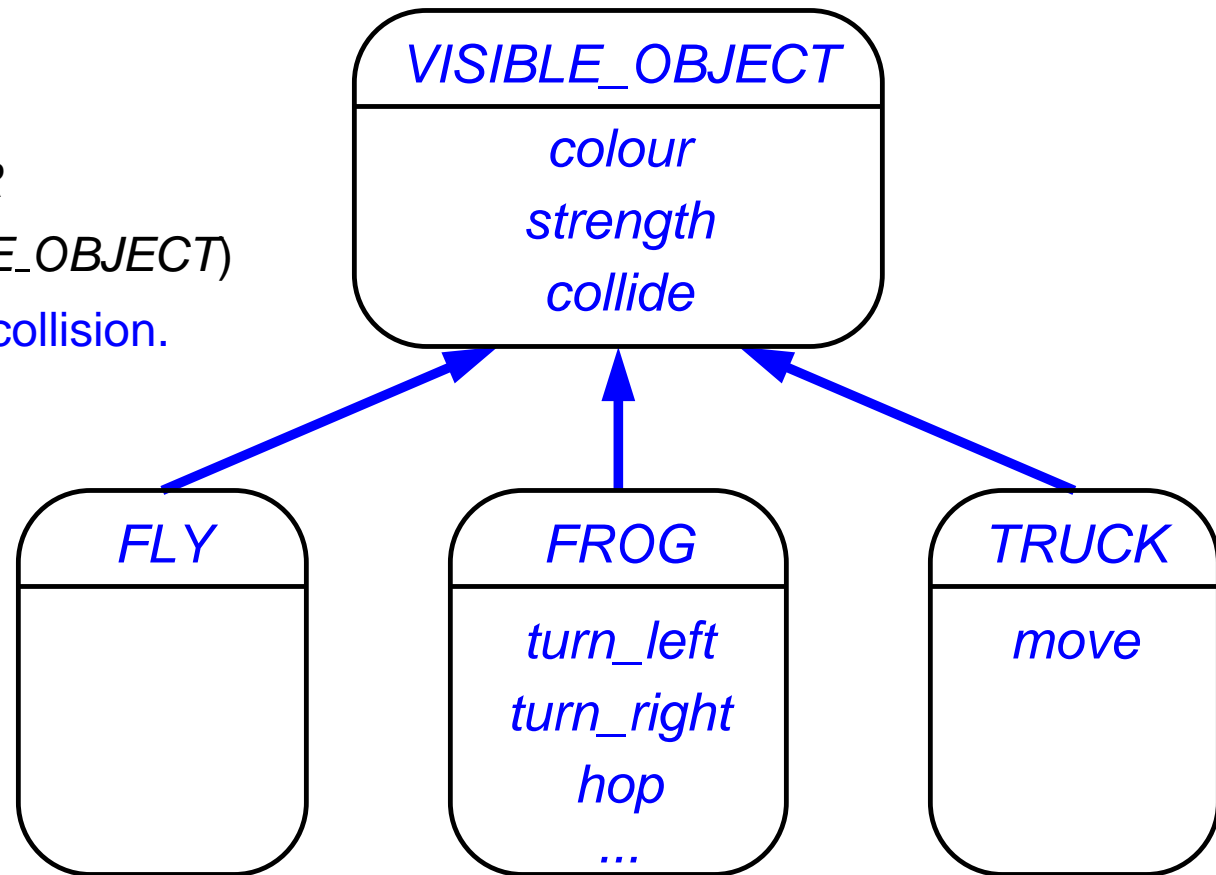
One solution involves

- A notion of a general collidable objects.
- A notion of a display based on objects rather than colours.

A Hierarchy of Visible Game Objects

- *VISIBLE_OBJECT*
 - *colour: COLOUR*
 - *strength: INTEGER*
 - *collide (o: VISIBLE_OBJECT)*

The stronger object 'wins' a collision.



The notion of *strength* and *collision* subsumes features like *eat* and *squish*.

OBJECT_DISPLAY Features

- queries about dimensions

- *lower1: INTEGER* | ■ *lower2: INTEGER*

- *valid_index1(x: INTEGER): BOOLEAN*

- ...

- access to individual squares

- *put(o: VISIBLE_OBJECT; x, y: INTEGER)*

- ◆ This square now shows *o*'s colour.

- ◆ If previously occupied by *p*, then *o. collide (p)* and *p. collide (o)*.

- *item(x, y: INTEGER): VISIBLE_OBJECT*

- refresh the picture on screen

- *refresh*

OBJECT_DISPLAY Implementation

An *OBJECT_DISPLAY* will need to use a *DISPLAY* to present itself.

✗ Option 1: *OBJECT_DISPLAY* is an heir of *DISPLAY*

Not possible as types are not in agreement:

■ *DISPLAY*

◆ *put(c: COLOUR; x, y: INTEGER)*

◆ *item(x, y: INTEGER): COLOUR*

■ *OBJECT_DISPLAY*

◆ *put(o: VISIBLE_OBJECT; x, y: INTEGER)*

◆ *item(x, y: INTEGER): VISIBLE_OBJECT*

We could proceed using different names, but this would leave the raw, colour, interface exposed.

✓ Option 2: *OBJECT_DISPLAY* has a *DISPLAY*