RetroCraft – A design language for retro platformers

Language Reference Manual

Fernando Luo **(fbl2108)**
Papoj Thamjaroenporn **(pt2277)**
Lucy He **(lh2574)**
Kevin Lin **(kl2495)**

Table of Contents

1. Introduction

This is the Language Reference Manual for the RetroCraft programming language. RetroCraft is a language that provides users with the building blocks to conveniently and creatively design their own game level for a platform game. RetroCraft defines an intuitive syntax that will allow the programmer to express the boundaries of a level, gameplay mechanics, and events. The language will execute user specified events including collisions, transitions, and movements. RetroCraft offers a default collision detection engine for appropriate element interactions, which users can choose to overload to customize their own events.

The default file extension for our language is **.rc**.

2. Lexical Convention

2.1 Comments

Double forward slashes `//` indicate the beginning of a single line comment. Multiple line comments will begin with `/*`and end with `*/`.

2.2 Tokens

The types of tokens in our language are: keywords, identifiers, constants, string literals, operators and separators.

2.2.1 Keywords

RetroCraft has a list of reserved words with fixed purposes.
Variable type declaration: `int, float, char, string, bool, function`
Control flow: `if, else, while, for, return`
Data object: `Array, Image, Map, PlayerObj, Object, EnvObj, ActObj, EventManager`
Truth values: `true, false`

2.2.2 Identifiers

Identifiers begin with a dollar sign (`$`) followed by a sequence of upper and/or lowercase characters, digits and underscores, starting with a non-numerical character. The keywords in 2.2.1 are not valid identifiers. Upper and lower case characters are unique, making identifiers case-sensitive.

2.2.3 Separators

`\t`	tab
`\n`	new line feed
`<space>`	space

2.2.4 Punctuators

;	end of line
,	separates arguments, object attributes, and array elements
{ }	code block
‘ … ’	single quotes for char
“ … ”	double quotes for string
( )	function calls or arithmetic operations
[ ]	array
.	referencing object’s attributes and functions

2.3 Operators

2.3.1 Arithmetic

Our arithmetic operators will be the standard operators present in most languages. The symbols and associated operations are as follows:

:	Assignment
+, -	Addition and Subtraction
+:, -:	Shorthanded Increment and Decrement
*, /	Multiplication and Division
%	Modular
^	Exponentiation

Arithmetic expressions will be made using infix notation, i.e. `operand1 operator operand2.` The standard order of operations specified by arithmetic will be honored, i.e. PEMDAS.

Arithmetic can be done with the types: `int, float, char`. The result type of the arithmetic operation will depend on the operands. For example, arithmetic with integers will return an integer. However, arithmetic with an integer and a float will result in a syntax error.

2.3.2 Comparison

`=`	Equal
`!=`	Not equal
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal

These operators compare variables and/or constants with each other and return a boolean constant (`true` or `false`). Incompatible types will result in a syntax error.

2.3.3 Logical Operators

&&	AND
||	OR
!	NOT

Logical operators can be used with expressions which evaluate to either `true` or `false`. The order of precedence is: NOT, AND, then OR. It is recommended that a parenthesis is used when an expression involves multiple logical operators, e.g., `($x = 3) || (($x = 4) && ($y = 1))` instead of `($x = 3) || ($x = 4) && ($y = 1)`

2.3.3 Member Operators

Member operators on objects will use a single dot (`.`) notation. For example, to access the height property of a Map object `gameMap`, the notation `gameMap.height` should be used.

Member operators on our zero based arrays will use a square bracket notation. For example, to access the 2nd index of an array `sampleArray`, the notation `sampleArray[1]` should be used.

3. Statements

3.1 `if, else if, else`

`if`, `else if` and `else` statements are used to control when their contained blocks of code will be executed. For example:

```
if (logical expression) {
     // code executed if above expression evaluated to true
} else if (logical expression) {
```

```
            // code executed if first logical expression was false and
            // the second was true
      } else {
            // code executed if both logical expressions were false
      }
```

Code conditional on an if statement must be surrounded by brackets.

3.2 `for`

`for` statements are used to control the number of times a block of code is executed. The for statement has three components:

```
      for (<variable initiation> ; <logical expression> ; <variable
increment/decrement>) {
            // code to execute
      }
```

The code will continue to be executed as long as the logical expression is true. The variable initiation and increment/decrement give a compact way to control the number of times the code is executed. For example, the following would iterate through the code 5 times:

```
      for (int $i : 0; $i < 5; $i++) {
            // code to execute
      }
```

3.3 `while`

A `while` loop evaluates the bracketed statements if the given logical expression remains true.

```
      while (logical expression) {
            // code to execute
      }
```

3.4 `break`

`break` keyword is only used in either `for` or `while` loops. When the statement '`break;`' is used, the program steps out of the loop regardless of the current state of the logical expression.

3.5 `continue`

`continue` keyword is only used in either `for` or `while` loop. When the statement '`continue;`' is used, the program skips to the end of the current iteration.

3.6 `return`

Functions terminate when they reach a return statement. If the function has a return type, `return` must be followed by a value of that type.

4. Declarations and Assignments

4.1 Variable Declaration

We can declare a new primitive variable using the following syntax:

```
    // Declaration and Assignment combined
    <primitive type> $<var_name> : <value>;
```

or,

```
    // Declaration and Assignment done separately
    <primitive type> $<var_name>;
    $<var_name> : <value> ;
```

For example:

```
    int $myInt : 5;
```

or,

```
    int $myInt;
    $myInt : 5;
```

Section 6.1 will discuss the declaration and construction of object types.

4.2 Array Declaration

To define an array, we use a square bracket to declare the array, while the elements of the array are specified by curly brackets. If the user wishes to specify only the size of the array during construction, the user can just specify the size after the object type (without square brackets). The syntaxes are shown below:

```
    Array $<name_of_array> : Array <object_type> <size>;
```

or,

```
    Array $<name_of_array> : Array <object_type> [
        <object_type> {
            <attribute_name> : <attribute_type> value
            ...
        },
        ...
    ];
```

For example:

```
    Array $arrayOfInts : Array int 3;
    $arrayOfInts[0] = 4;
```

```
        $arrayOfInts[1] = 1;
        $arrayOfInts[2] = 2;
```

or,

```
        Array $arrayOfInts : Array int [
                4,
                1,
                2
        ];
```

For objects:

```
        Array $arrayOfEnvs: Array EnvObj [
                EnvObj {
                        $x : 1.0,
                        $y : 1.0
                },
                EnvObj {
                        $x : 2.0,
                        $y : 2.0
                }
        ];
```

The way we can access an array element is the following:

```
        $arrayOfEnvs[1]
```

The index of any array starts from zero.

The size of elements in the array can also be accessed by the attribute `length`. For example:

```
        $arrayOfEnvs.length
```

4.3 Function Declaration

Function declarations begin with the keyword `function`. The header will also contain the return type and formal parameters. If there is no return type, `void` should be used instead.

```
        function func_name : <return type> (<parameters>) {
                // Implementation
        };
```

For example,

```
        function addStairs: void (Map $map, Image $stepImg, int $size,
        float $x, float $y) {
                for (int $i : $x; $i < $x+$size; $i++){
                        for (int $j : $y; $j < $y+$size-$i; $j++){
                                EnvObj $step : EnvObj {
```

```
                                $envImage : $stepImg,
                                $x : $i,
                                $y : $j
                        };
                        $map.addEnvObj($step);
                }
        }
        return;
};
```

We will inherit the same mechanism on parameter passes from OCaml: all parameters are implicitly passed by reference.

5. Primitive Data Types and Basic Data Types

5.1 Primitive Data Types

`bool`	`true, false, 0, 1`
`int`	`..., -1, 0, 1, ...`
`float`	floating-point numbers, such as `3.14127`
`string`	`“Hello World”`
`char`	`‘c’`

5.2 Basic Data Types

`Array` (See 4.2)	Stores a collection of data elements of the data type. Array elements are accessed with square brackets. *Attributes* `int $length` The length of the array
`Image`	Contains the string of the path to the input image. *Attributes* `string $src` The relative path to the image file
`Map`	The canvas for the game. It is the container for all the `PlayerObj`, `EnvObj` and `ActObj` objects in the game. It also contains attributes that affect its contained objects, including gravity values. *Variable and Object Attributes* `float $gx` Gravity vector. Determines how quickly (and in which

`float $gy`	direction) `Player` or `EnvObj` objects accelerate when unrestricted

`float $width`
`float $height`
	The width and height of the grid

`Image $background`	A background image
`PlayerObj $player`	Player character
`Array ActObj $actObjs`	An array of the `ActObj` 's in the map
`Array EnvObj $envObjs`	An array of the `EnvObj` 's in the map
`Array TextObj $textObj`	An array of the `TextObj` 's in the map

Function Attributes

`addPlayer (PlayerObj $player)`	Add the given player object to the map.
`addActObj (ActObj $actObj)`	Add `$actObj` to the array `$actObjs`
`addEnvObj (EnvObj $envObj)`	Add `$envObj` to the array `$envObjs`
`addTextObj (TextObj $textObj)`	Add `$textObj` to the array `$textObjs`

`Object`

The superclass of `PlayerObj`, `EnvObj`, `ActObj`, `TextObj`. Will be useful for collision detection and polymorphism.

Variable and Object Attributes

`float $px`
`float $py`
	x and y coordinate of the object

`float $width`
`float $height`
	width and height of the object

`bool $visible`	indication if the object can be seen or not

`PlayerObj`

This class extends `Object`.
The user controlled character which can be controlled to move through the map.

Variable and Object Attributes

`int $imageIndex`	Index that indicates what image in `$playerImgs` to be drawn on the screen
`Array Image $playerImgs`	Images of the character at different states

`float $vx`
`float $vy`
	Velocity of the player

	Function Attributes

`onKeyPressed(Map $map, char $c)`	Given the keyboard input, update the `PlayerObj`
`onUpdate(Map $map)`	For each time step, update the attributes of the player based on the given environment, such as gravity
`onCollision(Map $map, Object $input)`	Specifies action when the player collides with object input

	For every time step, `EventManager` will call the function attributes in the following order: `onKeyPressed`, `onUpdate`, and `onCollision`
`EnvObj`	*Environmental object.* This class extends `Object`. Environmental objects are arranged in the map grid to define the valid, navigable space for the `PlayerObj` and `ActObj`'s. All environmental objects are static and cannot affect the state of other objects. *Variable and Object Attributes*

`Image $envImage`	The image for the object

	Examples: unbreakable walls, static platforms, hills
`ActObj`	*Active object*. This class extends `Object`. Active objects are those that have more than one state (right now dictated by variable `visible`), or can change the state of other objects (e.g. make the player invisible, i.e., die). They are also arranged in the map grid, but can be mobile. *Variable and Object Attributes*

`int $imageIndex`	Index that indicates what image in `$objImgs` to be drawn on the screen
`Array Image $objImgs`	Images of the object at different states

`float $vx`
`float $vy`
	Velocity of the player

Function Attributes

`onKeyPressed(Map $map, char $c)`	Given the keyboard input, update the object
`onUpdate(Map $map)`	For each time step, update the attributes of the ActObj based on the given environment, such as gravity or existing velocity
`onCollision(Map $map, Object $input)`	Specifies action when this ActObj collides with object input

	Examples: script controlled characters ('enemies'), static objects that change the state of anything else, traps, spikes.
`TextObj`	*Text object.* This class extends `Object`. We can display text on the screen along with the scene in the game. *Variable and Object Attributes* `string $text` the message to be displayed *Examples*: score panel at the top of the screen, "Game Over" message.
`EventManager`	Iterates through all the objects at each time step and calls the `onUpdate`, `onKeyPressed`, and `onCollision` functions of the objects when appropriate. *Function Attributes* `setTimeStep(float $timestep)` Set the global time step; i.e., the frequency in which `EventManager` will be called. This is function call is mandatory to run the game. (Default value = 0.04 s) `start(Map $gameMap)` Given the map which contains all the game objects, start running the game. Perform collision detection/resolution and updates until `$gameMap.gameEnded` is true, then call `$gameMap.onGameEnded()`

6. Operations on Graphics Objects

Since RetroCraft is primarily graphics based, we require a specific set of attributes and methods in order to control the layout and flow of the game. The following sections describe them.

6.1 Object Construction

Object variables are declared and constructed similar to the syntax specified in the variable declaration section above (4.1):

```
    <object type> $<var_name>;
    $<var_name> : <attributes>;
```

or,

```
    <object type> $<var_name> : <attributes>;
```

Instead of a primitive type, the variable name is preceded by an object type, specified as a data object keyword in section 2.2.1. The value specified is a dynamically constructed object written in bracket notation. For example,

```
    Map $gameMap;
    $gameMap : Map {
        $width: 600.0,
        $height: 480.0,
        $background: Image {
            $src: “images/forestScene.jpg”
        },
        $player: Array Player [
            $mario //a previously defined Player object
        ],
        $actObjs: Array ActObj [],
        $envObjs: Array EnvObj [],
        $onUpdate: null
    };
or,
    Map $gameMap : Map {
        $width: 600.0,
        $height: 480.0,
        $background: Image {
            $src: “images/forestScene.jpg”
        },
        $player: Array Player [
            $mario //a previously defined Player object
        ],
        $actObjs: Array ActObj [],
        $envObjs: Array EnvObj [],
        $onUpdate: null
    };
```

6.2 Display and Movement

The game map is a grid of a user-determined height and width measured in pixels. Coordinates increment up and to the right, such that the bottom left space in the map has the coordinates (0,0). Game objects, such as players, enemy characters and walls, are rectangular shaped entities specified by height and width values and are placed on the game map grid at specified coordinates according to their `$px` and `$py` attributes. Upon rendering an object, the bottom left corner of the object is placed at the specified coordinate on the game map and the rest of the object spans the space above and to the right.

In order to simulate movement, we have provided an `EventManager` oracle which redraws the scene described by the game map and its objects at each timestep. The coordinate values of each object can be changed by any of the user defined functions assigned to its `onKeyPressed`, onUpdate, or onCollision attributes. For each frame, the EventManager oracle cycles through each of the objects on the map currently being run and calls the onKeyPressed function if a key is being pressed and updates each of the objects according to those functions. Then cycling through the objects a second time, the onUpdate function of each object is called to apply more changes. Finally, the EventManager cycles through all possible pairs of objects on the map to determine which pairs are at a point of collision, *a state we define as two objects whose bounding box perimeters are either in contact or overlapping*. Then for each of those objects found to be in a point of collision, their onCollision function is called with an input parameter of the object colliding with it in order to resolve those collisions.

Ideally, a user defined onKeyPressed function would be written to govern all changes to the object that user input would control, such as the increase of the velocity of the Player object when the user inputs a move forward key. Then the onUpdate function would be written to make changes to the object based on its current attributes and the passive rules of the environment, such as gravity and friction. Lastly, the onCollision function acts to apply the final checks to the system in the common case of object collision, such as making sure Player objects do not pass through the walls of the map.

For example, here is the definition of a player object on a map with wall objects on the south, east and west borders who starts on the accelerates to the right up to a certain speed as a user presses and holds down the 'D' key but gradually comes to a halt when the user lets go of the key. The reverse is also true if the user were to press and hold down the 'A' key. Additionally, when the player object runs into the wall object, it will come to an immediate halt.

```
PlayerObj $myPlayer: PlayerObj {
      $height: 20.0,
      $width: 10.0,
      $px: 10.0, //assuming the walls and floor are 10px thick
      $py: 10.0,
      $vx: 0.0,
      $vy: 0.0,
      $playerImgs: Array Image {
            Image {
                  $src: "images/playerImage.jpg"
            }
      },
      $visible: true,
      $onKeyPressed: void (Map $gameMap, char $keyPressed) {
            if ($keyPressed = 'd') {
                  if ($vx >= 0.0) { $vx +: 2.0; }
                  else { $vx : 2.0; }
                  if ($vx > 10.0) { $vx : 10.0; }
            }
            else if ($keyPressed = 'a') {
                  if ($vx <= 0.0) { $vx -: 2.0; }
```

```
                else { $vx : -2.0; }
                if ($vx < -10.0) { $vx : -10.0; }
            }
        },
        $onUpdate: void (Map $gameMap) {
            // $timestep is a global variable of the game
            $px : $px + $vx * $timestep - 0.5 * $gameMap.gx *
                $timeStep ^ 2.0;
        },
        $onCollision: void (Map $gameMap, Object $collidingObject) {
            if (typeOf($collidingObject, EnvObject)) {
                if ($px <= $collidingObject.px) {
                    $px : $collidingObject.px - $width;
                    $vx : 0;
                }
                else if ($px > $collidingObject.px) [
                    $px : $collidingObject.px + width;
                    $vx : 0;
                }
            }
        }
};
```

6.3 Modifying Objects

Attributes of various objects can be modified after object creation by referencing the object (`$<object name>`) and using the punctuator ‘`.`’ to call attributes:

```
main : int () {
        ActObj $newturtle : $createTurtle: void (50, 50,
            “./img/turtle1.png”);

        /* some code */

        $newturtle.height : 40;
        $newturtle.width : 40;
}
```

6.4 Advanced Attributes and Functions of `Object`’s

The object does not only provide basic attributes such as width and height of the object, but also some functionality that, after being defined by the user, can be used to control the behavior of the object and its interaction with other objects.

6.4.1 Dimensions

Each object’s dimension attributes, `$height` and `$width`, define the rectangular area of pixels allotted to it on the grid.

6.4.2 Coordinate Location
Each object's coordinate attributes, `$px` and `$py`. These coordinates could be changed over the course of a game with functions such as `onKeyPressed`, `onUpdate`, and `onCollision`.

6.4.3 Visibility
The visibility of objects can either be `true` or `false.` These values, again, could be altered during the game with functions: `onKeyPressed`, `onUpdate`, and `onCollision`.

6.4.4 OnKeyPressed
`onKeyPressed` method provides the user an ability to define the behavior of the object dictated by some input from the keyboard. `onKeyPressed` command will be constantly called as long as keyboard input is received.

6.4.5 OnUpdate
`onUpdate` method allows user to specify behavior of the object that is dependent on the timer. The EventManager, at each time step will call on `Player` and `ActObj`'s `onUpdate` function and execute any specified instructions.

6.4.6 OnCollision
`onCollision` method lets users to define the behavior between interactions with `ActObj's` and `Player` objects. The `EventManager` will iterate through all objects to check for collisions. If there is a collision, the `onCollision` methods in the affected objects will be invoked.

6.5 The EventManager
The `EventManager` is a built-in object that monitors and updates each object at each time step which by default is set to 0.04 seconds. Updates are carried out by the `EventManager` looping through each of the objects associated with a game map and calling each of their `onKeyPressed`, then each of their `onUpdate` functions, and finally each of their `onCollision` functions. After all game map objects have been updated and all collisions have been resolved, the `EventManager` then renders each of the objects onto the game map.

Within the main class, the game is started by calling the `start` function of the `EventManager` instance: `eventManger.start(Map $map)`. More about this function can be found in Section 7.3

7. Built-in & Required Functions

7.1 `main : int ()`
Every game created by RetroCraft requires a main function. All games will begin execution from this function.

The `main()` function is composed of two main sections. The first section includes the initialization of the `Map`, `EnvObj` 's, `ActObj` 's, and the `PlayerObj`. The next section is a call to the

EventManager, the engine of the game, using the start() function. Within this method (details in 7.3), the EventManager will manage collisions and events, until an end condition is reached. Then, the start function will proceed to call the maps onGameEnded()function. Post-game events would be coded here, for example the user can display a "Game Over" method.

7.2 Adding Objects

```
addPlayer : void (PlayerObj $player)
addEnvObj : void (EnvObj $envObj)
addActObj : void (ActObj $actObj)
addTextObj : void (TextObj $textObj)
```

These are functions of the Map.
The function adds the PlayerObj, EnvObj or ActObj object to the map at the coordinates ($px, $py) where $px and $py are attributes of the object, given to it when it's initiated. The function will error if there is already an object at ($px, $py).

7.3 start : void (Map $map)

This is a function of the EventManager.
In order for the EventManager to run, certain functions for the Player and ActObjs must be defined. More specifically, the onUpdate, onKeyPressed and onCollision functions must be defined before eventManager.start($gameMap)can be called.

The start function loops through code which goes through a series of steps:

1. Checks if a key has been pressed
    a. If a key has been pressed, pass that key to each ActObj's and Player's onKeyPressed()function
2. Calls the onUpdate of each ActObj and the Player
3. Checks for collisions between all objects
    a. If there is a collision, call the appropriate onCollision functions contained in each object.
    b. Correct the position of the objects if necessary to avoid overlapping.
4. Draw the Map and objects at their updated locations.
5. Continue looping back to step 1 as long as an $gameMap.gameEnded is false
6. Call $gameMap.onGameEnded()

7.4 onGameEnded : void ()

This is a function of the game Map. It is executed by the EventManager when gameplay has ended, with functionality differing based on the outcome of the game (i.e. win or lose).
The function will evaluate the objects in the map and display the appropriate end of game message. For example, the Map could determine that the player has lost if the PlayerObj is invisible.

```
onGameEnded : void () {
if ($player.visible = true) {
```

```
            $winMsg : TextObj {
                  $height: 30,
                  $width: 300,
                  $px: 10.0,
                  $py: 10.0,
                  $visible: true,
                  $text: “You win!”
            };
            addTextObj ($winMsg);
      } else if ($player.visible = false) {
            $loseMsg : TextObj {
                  $height: 30,
                  $width: 300,
                  $px: 10.0,
                  $py: 10.0,
                  $visible: true,
                  $text: “Game Over”
            };
            addTextObj ($loseMsg);
      }
      return;
}
```

After this function runs, the `start()` function of the `EventManager` terminates.

7.5 `typeOf : bool (Object $o, <type>)`

Given `Object $o`, and a type this function returns whether the object is of the given type. This can be useful for implementing unique behaviours between different types of objects. For example in collision resolution:

```
$onCollision: void (Map $gameMap, Object $collidingObject) {
      if (typeOf($collidingObject, EnvObj)) {
            // Some behaviour
      } else if (typeOf($collidingObject, ActObj)) {
            // Alternative behaviour
      }
}
```

7.6 `cast : <newType> (<variable type> $x, <newType>)`

Given a variable of data type (`int`, `float`, `char`, `string`, `bool`) and a new type to convert to, cast the variable to the specified new type. An example is shown below:

```
int $myInt : 10;
string $myFloat;
$myFloat : cast($myInt, float);  // $myFloat is now 10.0
```

7.7 `setTimeStep : void (float $timeStep)`

This is a function of the `EventManager`.

Set the global time step; i.e., the frequency in which `EventManager` will be called. This is function call is mandatory to run the game. The use of example:

```
setTimeStep();
```

8. Sample Code

PyramidTurtle.rc

```
/*
      Demonstrate a simple game with one player and one enemy. If the
      player reaches the flag, then he wins. The player loses the game
      if he hits the turtle.
*/

float $timeStep : 0.04;

/*
Create a declining stairs that has bottom-left corner at (0,0), with
its width, height, and step size determined by given parameters.
*/
function $createStairs: Array EnvObj (string $stepImg, float $height,
float $width, float $steps) {
      Array $envObjs : Array EnvObj $steps;
      float $stepHeight : ($height / $steps);
      float $stepWidth : ($width / $steps);

      float $x_coord : 0.0;
      float $obj_height : $height;
      while ($obj_height > 0.001) { // Prevent floating point errors
            EnvObj $step : EnvObj {
                  $envImage : Image { $src : $stepImg },
                  $px : $x_coord,
                  $py : 0.0,
                  $height : $obj_height,
                  $width : $stepWidth,
                  $visible: true
            }
            $envObjs.add($step);
            $x_coord +: $stepWidth;
            $obj_height -: stepHeight;
      }
```

```
        return $envObjs;
};

function  $createHorizontalGround:  Array  EnvObj  (string  $groundImg,
float  $x,  float  $y,  float  $obj_height,  float  $obj_width  int
$noOfTiles) {
        Array $envObjs : Array EnvObj $groundTiles;
        float $x_coord : $x;
        float $y_coord : $y;
        for (int $i = 0; $i < $noOfSteps; $i++) {
                EnvObj $tile : EnvObj {
                        $envImage : Image { $src : $groundImg },
                        $px : $x_coord,
                        $py : 0.0,
                        $height : $obj_height,
                        $width : $obj_width,
                        $visible: true
                }
                $envObjs.add($tile);
                $x_coord +: $obj_width;
        }
        return $envObjs;
};


function main : int () {
        ActObj $turtle : {
                $objImg : Image {
                        $src : “turtle.gif”
                },
                $px : 30.0,
                $py : 270.0,
                $width : 30.0,
                $height : 50.0,
                $vx : 10.0,
                $vy : 0.0,
                $visible : true,
                $onUpdate : void (Map $gameMap) {
                        $vy : $vy - $timestep * $gameMap.gy;
                        $px  :  $px  +  $vx  *  $timestep  -  0.5  *  $gameMap.gx  *
                                $timeStep ^ 2.0;
                        $py  :  $py  +  $vy  *  $timestep  -  0.5  *  $gameMap.gy  *
                                $timeStep ^ 2.0;
                },
                $onKeyPressed : void (Map $gameMap, char $c) {
                        // Do nothing. Turtle unaffected by key presses.
                },
          $onCollision : void (Map $gameMap, Object $o) {
               // If the turtle is on the ground, move the y
               // coordinate back to prevent penetration.
               if (typeOf($o, EnvObj) &&
                    ($py < $o.py + $o.height) &&
                    ($py + $height > $0.py)) {
                    $vy : 0;
                    $py : $py + $vy * $timestep + 0.5 * $gameMap.gy *
                         ($timeStep ^ 2.0);
               // If the turtle touches the player, the player
               // faints and the game is over.
               } else if (typeOf($o, PlayerObj)) {
                    $gameMap.player.visible : false;
                    $gameMap.gameEnded : true;
               }
          }
     };

     ActObj $goal : {
          $objImg : Image { $src : “flag.png” },
          $px : 0.0,
          $py : 300.0,
          $height : 30.0,
          $width : 30.0,
          $vx : 0.0,
          $vy : 0.0,
          $onUpdate : null,
          $onKeyPressed : null,
          $visible: true,
          $onCollision : void (Map $gameMap, Object $o) {
               // If turtle touches the flag, do nothing
               if (typeOf($o, ActObj)) {
                    return;
               }
               // If the player touches the flag, end the game
               if (typeOf($o, PlayerObj)) {
                    $gameMap.player.visible : true;
                    $gameMap.gameEnded : true;
               }
          }
     };


     Array EnvObj $stairs : createStairs(“stairs.png”, 300.0, 300.0,
30.0);
     Array   EnvObj   $ground   :   createHorizontalGround(“ground.png”,
     300.0,
    0.0, 30.0, 30.0, 10.0);

Map $gameMap : Map {
    $width : 600.0,
    $height: 480.0,
    $gy : -10.0,
    $gx : 0.0,
    $gameEnded: false,
    $onUpdate: null,
    $onGameEnded: void () {
        TextObj $message;
        if ($player.visible = false) {
            $message : TextObj {
                $height : 100.0,
                $width : 50.0,
                $px : 275.0,
                $py : 190.0,
                $text : “You lost!”
            };
        }
        else if ($player.visible = true) {
            $message : TextObj {
                $height : 100.0,
                $width : 50.0,
                $px : 275.0,
                $py : 190.0,
                $text : “You won!”
            };
        }
        addTextObj($message);
    }
};

Player $myPlayer: Player {
    bool $jumping : false, // Define custom variable $jumping
    float $jumpV : 15.0,   // Define initial jumping velocity
    $height: 20.0,
    $width: 10.0,
    $px: 500.0,
    $py: 30.0,
    $vx: 0.0,
    $vy: 0.0,
    $playerImg: Image {
        $src: “images/playerImage.jpg”
    },
    $visible: true,
    $onKeyPressed: void (Map $gameMap, char $keyPressed) {
         if ($keyPressed = ‘d’) {
               if ($vx >= 0.0 && $vx < 10.0) { $vx +: 2.0; }
               else { $vx : 10.0; }
         }
         else if ($keyPressed = ‘a’) {
               if ($vx <= 0.0) { $vx -: 2.0; }
               else { $vx : -2.0; }
               if ($vx < -10.0) { $vx : -10.0; }
         }
         else if ($keyPressed = ‘w’) {
               if ($jumping = false) {
                     $jumping = true;
                     $vy = $jumpV;
               }
         }
    },
    $onUpdate: void (Map $gameMap) {
         // $timestep is a global variable of the game
         $vy : $vy + $timestep * $gameMap.gy;
         $px  :  $px  +  $vx  *  $timestep  +  0.5  *  $gameMap.gx  *
               $timeStep ^ 2.0;
         $py  :  $py  +  $vy  *  $timestep  +  0.5  *  $gameMap.gy  *
               $timeStep ^ 2.0;
         },$onCollision: void (Map $gameMap, Object $o) {
         if (typeOf($o, EnvObject)) {
               // For horizontal collisions, move the x
               // coordinate back.
               if (($px < $o.px + $o.width) &&
                     ($px + $width > $o.px)) {
                     // If hitting EnvObj from right
                     $px = $o.px + $o.width;
               } else if (($o.px < $px + $width) &&
                     ($o.px + $o.width > $px)) {
                     // If hitting EnvObj from left
                     $px = $o.px - $width;
               }

               // For vertical collision, set $vy = 0
               else if (($o.py < $py + $height) &&
                          ($o.py + $o.height > $py)) {
                     // If hitting EnvObj from below
                     $vy = 0.0;
                     $py = $o.py - $height;
               } else if (($py < $o.py + $o.height) &&
                          ($py + $height > $o.py)) {
                     // If hitting EnvObj from above
                     $vy = 0.0;
                     $py = $o.py + $o.height;
                     $jumping = false;
```

```
                }
            }
};
    $gameMap.addPlayer($myPlayer);
    $gameMap.addActObj($turtle);
    $gameMap.addActObj($goal);
    for (int $i : 0; i < $stairs.length; $i++) {
        $gameMap.addEnvObj($stairs[$i]);
    }
    for (int $i : 0; i < $ground.length; $i++) {
        $gameMap.addEnvObj($ground[$i]);
    }

    EventManager.setTimeStep($timeStep);
    EventManager.start($gameMap);
}
```