RetroCraft – A design language for retro platformers

Project Proposal

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

1. Introduction

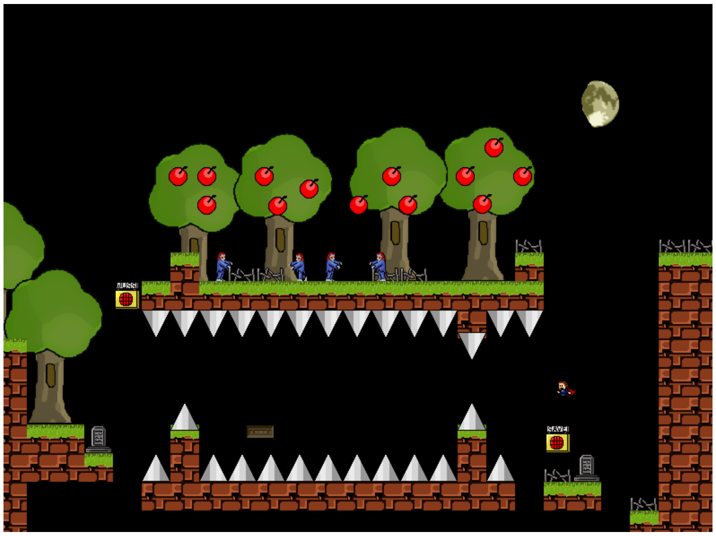
Since the creation of platform games in the 1980s, video gamers have witnessed the growth and evolution of 2D platformers. The genre persists today with various legacies of games such as Super Mario Bros and Donkey Kong. However, gamers and hobbyists rarely have the chance to design their own levels, let alone the intricate game mechanics. We are going to implement 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 and impose their own events. Finally, map transitions and basic movements can expressed easily with RetroCraft.

2. Objective

Making even a basic game from scratch requires a significant time investment from the programmer to set up the essential data structures and objects. Our language aims to significantly reduce this overhead by providing the programmer with basic tools to create maps, design characters, and define events within the application. After specifying these base elements, our tool will render the game in a window, following the user specified rules of behavior for those objects. While limited to the creation of platformer style games such as Super Mario Bros., the user can quickly generate playable content and focus more of his or her time on evaluating game concepts and playability.

Using the data structures we have defined, a programmer will first establish a gridded canvas or a map on which the level elements will be placed. The user will then be able to explicitly place `EnvObjects`, `Characters`, and `ActObjects` at specific coordinates on the grid. With the attributes and event triggers built into those objects, the programmer will be able to control how they interact and change each other's states, creating a unique gameplay experience.

The ideal output of our language would be something similar to the privately developed game, "I Want To Be the Guy" which can be found at http://kayin.pyoko.org/iwbtg/faq.php. The screenshot below encapsulates the essential nature we want games developed in our language to have.



In addition, our language would be very friendly to beginner programmers eager to break into the game design industry. The relative simplicity of our language would significantly reduce the programming learning curve for students trying to express their game ideas.

3. Program

As an example, our language allows users to generate an arbitrary game level of Super Mario. Our features include the capability of designing the terrain and environment of the scene, defining the responsive events when the player (i.e., Mario) *collides* with scene items or application-controlled characters (i.e., Koopa), as well as arbitrary responsive actions to various items in the game.

One simple level would involve the player traversing predefined platforms from top to bottom. Each section of the map will have obstacles including, but not limited to, enemy characters, spikes, and even moving objects. Programmable gravity and character speed will determine the game flow. Furthermore, user can expand the capability of speed and gravity to incorporate advanced game mechanics such as wind (horizontal force), ice terrain (high terminal velocity), and even mud (slower character speed).

Upon reaching a certain point in the map, the player will be presented with a map transition or a sequence of events that signals completion of a level. It is also interesting to note that the gameplay will be dictated by various events dependent on an internal timer.

4. Syntax

4.1 Primitive Data types

boolean	TRUE, FALSE, 0, 1
int	…, -1, 0, 1, …
float	floating-point numbers, such as 3.14127
string	“Hello World”
char	‘c’

4.2 Basic Data types

<table>
<tr><td>Point</td><td>Stores two attributes specifying x, y components of the point</td></tr>
<tr><td>Vector</td><td>Stores two attributes specifying x, y components of the vector</td></tr>
<tr><td>Image</td><td>Contains binary data of the input image</td></tr>
<tr><td>Map</td><td>A square grid that serves as the organizational base for the Player, EnvObjects and ActObjects.
<i>Attributes</i>
<table>
<tr><td>gravity: Vector</td><td>Determines how quickly (and in which direction) the player or objects accelerate when unrestricted</td></tr>
<tr><td>map: Array Array Object</td><td>A dimension x dimension grid of all the objects.</td></tr>
<tr><td>dimension: int</td><td>The height and width of the grid</td></tr>
<tr><td>background: Image</td><td>A background image</td></tr>
<tr><td>music: string</td><td>Path to music file to play in the background</td></tr>
</table></td></tr>
<tr><td>Object</td><td>The superclass of Player, EnvObject, ActObject. Will be useful for collision detection and polymorphism.</td></tr>
<tr><td>Player</td><td>The user controlled character which travels through the map.
<i>Attributes</i>
<table>
<tr><td>states: Array string</td><td>All possible states for the player (e.g. super speed or injured)</td></tr>
<tr><td>currPosition: Point</td><td>Player’s current position</td></tr>
<tr><td>playerImgs: Array Image</td><td>Images of the character at different states</td></tr>
<tr><td>currState: int</td><td>Player’s current state, as an index of the array of states</td></tr>
<tr><td>currVelocity: Vector</td><td>Player’s current velocity</td></tr>
<tr><td>termVelocity: Vector</td><td>The maximum velocity the character can achieve</td></tr>
<tr><td>livesLeft: int</td><td>Number of lives the player has left</td></tr>
</table>
<i>Functions</i></td></tr>
</table>

<table>
<tr>
<td></td>
<td>
<table>
<tr><td>onKeyPressed(char c)</td><td>Given the keyboard input, update the player’s position</td></tr>
<tr><td>onUpdate()</td><td>For each time step, update the attributes of the player based on the given environment, such as gravity</td></tr>
<tr><td>onCollision(Object input, string contact)</td><td>Specifies action when the player collides with object input, given the contact direction</td></tr>
</table>
</td>
</tr>
<tr>
<td>EnvObject</td>
<td>
Environmental object. Environmental objects are arranged in the map grid to define the valid, navigable space for the Character and ActObjects. All environmental objects are static, have only one state, and cannot affect the state of other objects.
Attributes
<table>
<tr><td>envImage: Image</td><td>The image for the object</td></tr>
</table>
Examples: unbreakable walls, static platforms, hills
</td>
</tr>
<tr>
<td>ActObject</td>
<td>
Active object. Active objects are those that has more than one state, or can change the state of other objects. They are also arranged in the map grid, but can be mobile.
Attributes
<table>
<tr><td>actObjStates: static Array string</td><td>All possible states for the object</td></tr>
<tr><td>currPosition: Point</td><td>Object’s current position</td></tr>
<tr><td>currState: int</td><td>Object’s current state, as an index of the array of states</td></tr>
<tr><td>currVelocity: Vector</td><td>Object’s current velocity</td></tr>
<tr><td>objImgs: Array Image</td><td>Images of the object at different states</td></tr>
</table>
Functions
<table>
<tr><td>onKeyPressed(char c)</td><td>Given the keyboard input, update the object’s position</td></tr>
<tr><td>onUpdate()</td><td>For each time step, update the attributes of the ActObject based on the given environment, such as gravity</td></tr>
<tr><td>onCollision(Object input, string contact)</td><td>Specifies action when this ActObject collides with object input, given the contact direction</td></tr>
</table>
Examples: script controlled characters (‘enemies’), static objects that change the state of anything else, traps, spikes.
</td>
</tr>
<tr>
<td>EventManager</td>
<td>Iterates through all the objects at each time step and calls the onUpdate, onKeyPressed, and onCollision functions of the objects when appropriate.
onUpdate and onKeyPressed will be called for every active object and the player, whereas onCollision will be called for all objects.</td>
</tr>
</table>

4.3 Operators

Arithmetic	`+, -, *, /, %`
Assignment	=
Boolean	`==, >=, <=, >, <`
Unary	`!, -, ++, --`

5. Code Examples

Our syntax takes on the following general format. Objects are initialized by specifying:

[variable name]: [object type] {
[attribute name]: [object type] ...
}

Attributes of an object can take on values of other objects, primitives or functions. The syntax of functions follow a java-like example. This may change as we develop our project concept further.

```
/*
Generate a simple mario game with one interactive enemy turtle, with
one very simple rule: if the player touches the turtle, the turtle
dies.
*/
playerImgs: Array Image [
      Image {
            src: string "Mario.jpg"
      }
]
turtleImgs: Array Image [
      Image {
            src: string "Turtle.jpg"
      }
]

/*
Define a turtle enemy
Returns an ActObject
*/
turtle1: ActObject
{
      name: string "Turtle"
      dimension: Object {
            width: int 10,
            height: int 10
      },
      currPosition: Point {
            x: int 5,
            y: int 500
      },
      currVelocity: Vector {
            x: int 2,
            y: int 0
```

```
      },
      visible: bool 0,
      image: Image turtleImgs[0],

      states: Array string [
            string “turtleAlive”,
            string “turtleDead”
      ],
      currState: int 0,

      /* functions calls by event manager */
            onUpdate: void function() {
                  currPosition.x += currVelocity.x;
                  currPosition.y += currVelocity.y;
      }

      onKeyPress: null, /* do nothing */
      onCollision: void function(Object input, string contact) {
            if (states[currState] == “turtleAlive” &&
               typeOf(input) == Player) {
                  currState = 0;
                  visible = 0;
            }
      }
}

/*
Define Map. The map contains not only the background, but also all the
objects (Player, EnvObject, and ActObject) in the scene. The inclusion
of scene objects is done in the main method.
*/
gameMap: Map {
      height: int 1000,
      width: int 1000,
      background: Image {
            src: string “Sky.jpg”
      }
}

/*
Create the ground platform composed of a simple rectangle block of size
1000 x 500. The block is placed at a location specified by
currentPosition, started from the bottom-left corner.
*/
groundPlatform: EnvObject {
      currentPosition: Object {
            x: int 0,
            y: int 0
      },
      height: int 500,
      width: int 1000,
      visible: int 1,
      image: Image {
```

```
        src: string “RockBlock.jpg”
    }
}



/*
Define playable character attributes
*/
player: Player {
    name: string “Mario”
    dimension: Object {
        width: int 10,
        height: int 10
    },
    currPosition: Point {
        x: int 200,
        y: int 500
    },
    currVelocity: Vector {
        x: int 2,
        y: int 0
    },
    visible: bool 1,

    states: Array string [
        string “marioAlive”,
        string “marioFainted”
    ],
    currState: int 0,

    jumpHeight: int 10,
    image: Image playerImgs[0],

    onUpdate: null, /* movement will be controlled by keypress */
    onKeyPress: void function (string keyinput) {
        /* Move the player in the direction specified by the
keyinput
          while increasing the velocity until the terminal velocity
        is
          reached */
        if (keyinput == “L”) {
            currPosition.x -= currVelocity.x;
            if (currVelocity.x > -10) {
                currVelocity.x--;
            }
        }
        else if (keyinput == “R”) {
            currPosition.x += currVelocity.x;
            if (currVelocity.x < 10) {
                currVelocity.x++;
            }
        }
```

```
        },

        /* In this example the player can move only left or right. So the
        player cannot jump. Furthermore, the collision with enemy turtle
        will have no effect on the player. Therefore, the method is
        defined as null.
        */
        onCollision: null
}



/*
main function
*/
main: int function ()
{
        insertObject(gameMap, groundPlatform)
        insertObject(gameMap, turtle1);
        insertObject(gameMap, player);

        /*
        Final call to play the game. The system runs in an infinite loop
        triggered by each tick of a timer (internally employed in OCaml).
        The loop contacts event manager to respond to keyboard input,
        detect collisions, and update the corresponding objects.
        */
        play(gameMap);
}
```