MakerGame

Cindy Wang (xw2368),
Steven Shao (ys2833),
Yuncheng Jiang (yj2433)

Description

Our language is inspired by the programming models of popular game engines, like Game Maker and Unity. Game Maker games are made up of many types of resources ranging from Sprites to Fonts to Shaders to Objects. Our goal is to define a stripped down version of this model, focused on expressing entities with individual encapsulated behaviour, but also including capabilities to respond to input, render to a window, and produce audio. This makes possible various kinds of graphical programs, from creative animations to physical simulations and to, of course, video games.

Like in Ballr, the key feature in our language is the ability to specify game objects, which roughly correspond to entities—like players, enemies, collectibles—in a video game. Game objects are allowed to have primitive member variables, and also have a few built-in functions that characterize behaviour over a lifetime. Game programmers are removed from the burden of game loop setup and instead can focus solely on individual object behaviour.

Language Features

Generic features

1. Standard primitives: `int, float, bool, string`
2. Fixed-size arrays: `int[5], int[3][4]`
3. Operators:
    a. Arithmetic `(+, -, *, /, ^, %, !)`
    b. Assignment `(=)`
    c. Comparison `(<=, >=, < , >, ==, !=)`
    d. Boolean operators (short-circuit `&&`, `||`)
4. Built-in game types: `sprite, sound`
5. Comments: `/* block */`, `// line`
6. General control flow
    a. Conditions (`if, else`)
    b. Generic loops (`while, for`)
    c. Object-based foreach loop (`foreach (Enemy e) { … }`)

Functions & Variables

Global variables can be defined and modified in the top level, as can statically typed global functions like in C. Expressions can reference previously defined variables and functions. Functions can also refer to object types, to be described below.

```
int factor = 3;
int triple(x) { return factor*x; }
```

Objects

Our key feature is the ability to define a game object: a structure with public member variables and functions upon object activation, destruction, rendering, and time-step. The tentative syntax is as follows:

```
ExampleObject {
      // member variable declaration
      // might disallow refs to other objects; depends on time
      int x; int y; int health; Sprite sprite;

      // game loop function definitions, with arbitrary behaviour
      // all definitions are optional, with defaults empty

      // called when an instance is constructed
      CREATE { x = 100; y = 100; health = 100; sprite = heroSprite; }
      // called when an instance is destroyed
      DESTROY { play_sound(screamSound); }
      // called every frame the instance is active in the game
      STEP {
            // 'this' refers to calling game object
            if (y > 1000) destroy(this);
      }
      // called every frame after STEPs for rendering
      DRAW { draw_sprite(x, y, sprite); }
}
```

We are also considering providing a GameObject “preset” that has built-in x, y, and sprite variables, and a default DRAW function that renders the sprite at the location. This covers a lot of classes of game objects that we might need so it should avoid a lot of setup repetition (GameMaker uses this kind of preset as the default). An equivalent object using this preset might look like:

```
ExampleObject : GameObject {
      int health;
      CREATE { x = 100; y = 100; health = 100; sprite = heroSprite; }
      DESTROY { play_sound(screamSound); }
      STEP { if (y > 1000) destroy(this); }
}
```

Built-in functions

1. Printing (print(string), print(int), print(bool), print(object))
2. Math (sin, sqrt, other things in cmath as deemed necessary)
3. Input (key_down(key), mouse_down(), mouse_x(), mouse_y())
4. Window (set_window_size(w, h), set_window_fps(fps), exit())
5. Resources
    a. load_sprite(filename), load_sound(filename)
    b. draw_sprite(x, y, sprite) - render an image to a location on the screen
    c. play_sound(sound, repeat) - play the sound, possibly repeating
6. Objects
    a. create(Object) - create and get a handle to a new instance of Object
    b. destroy(object) - remove the object referenced by this handle from the game
    c. is_valid(object) - check if the handle references an object still in the game
7. Geometry and collision - maybe we'll have helper functions for distances, hitboxes, etc., along with an associated type to express hitboxes

Sample Program

This is an example of a common "first game", where objects periodically spawn and fall from the sky, and players have to move a basket on the ground to catch them.

```
// Declarations of just object types and member variables
Egg { int x; int y; }
Player { int x; int y; }
Spawner { int timer; }
RmMenu { }
RmGame { }

// Initializing resources & global variables
Sprite playerSprite = load_sprite("player.png");
Sprite eggSprite = load_sprite("egg.png");
Sound boinkSound = load_sound("boink.ogg");
int score = 0;
```

```
// Definitions of global functions
// notice how these functions can operate on game object variables
bool hit_ground(Egg e) { return (e.y > 600); }
bool egg_touching_player(Egg e, Player p) {
     return (e.x < p.x + 50 && e.x > p.x - 50
          && e.y < p.y + 10 && e.y > p.y - 10);
}

// Definitions of object types
Egg {
     int x; int y;
     // sound effect upon spawning
     CREATE { play_sound(boinkSound); }
     // fall 5px every frame, and game over upon hitting the ground
     STEP { y += 5; if (hit_ground(this)) exit(); }
     // render the location of the egg every frame
     DRAW { draw_sprite(x, y, eggSprite); }
}

Player {
     int x; int y;

     STEP {
          // respond to keyboard input
          if (key_pressed(left)) x -= 5;
          if (key_pressed(right)) x += 5;

          // continuously check if we caught an egg
          foreach (Egg egg) {
               if (check_touching(obj, this)) {
                    // "catch" the egg and increase score
                    destroy(egg);
                    score += 5;
                    print("SCORE: " + score);
               }
          }
     }

     // every frame, rerender the player at the right place
     DRAW { draw_sprite(x, y, playerSprite); }
}

Spawner {
```

```
	int timer = 50;
	int spawn_positions[4] = {100, 200, 300, 400};
	STEP {
		// each frame, decrement timer.
		// when it reaches 0, create an egg somewhere.
		--timer;
		if (timer == 0) {
			timer = 50;
			Egg egg = create(Egg);
			Egg.x = spawn_positions[random(4)];
			Egg.y = 100;
		}
	}
}

RmGame {
	CREATE {
		// at start of game:
		//  - put objects into scene/room
		//  - set window paramters
		set_window_size(600, 600);
		Player p = create(Player);
		p.x = 300; p.y = 500;
		create(Spawner);
	}
}

// entry point: start the game!
create(RmGame);
```