Michael Fagan - mef2224 (Manager)

Elliott Morelli - gnm2123 (Language Guru)

Cherry Chu - ccc2207 (System Architect)

Robert Becker - rb3307 (Tester)

Lucifer: An Object-Oriented Language for Game Development

1) Describe the language that you plan to implement.

Our language proposal is to design and implement a language that will make it easier to develop 2D video games efficiently. We plan to have our implemented language syntactically similar to Java, in addition to being strongly and statically typed, have static scoping, and pass objects by “value” (call-by reference) like Java. Lucifer will also feature direct integration with the SDL2 library. SDL is a software development library that provides low level access to audio, keyboard, mouse, joystick and graphical hardware. This library is commonly used in the design of indie video games such as Cave Story, Angry Birds, and Dwarf Fortress, and its support for OCaml allows us to use it to mediate hardware interactions. We plan to include functionality from SDL directly into our language in order to create a more streamlined language that is hardware independent.

2) Explain what sorts of programs are meant to be written in your language

Our language will be built with the purpose of efficiently creating video games. In order to keep the scope of the project within a reasonable margin, our language will be developed with a focus on 2D games in mind. The examples we are hoping to implement are simpler games such as Pong, Space Invaders, and Pac-Man. That being said, 2D games of any type will be able to be implemented. As our language will also be able to produce console output and perform arithmetic, programs performing arithmetic operations will also be possible (although these would usually be used to move an entity around in a game).

3) Explain the parts of your language and what they do

The syntax of our language is nearly identical to the syntax of Java, with minor differences. These differences include that our print statements will just be formatted as ‘print();’ , that we are including a keyword ‘fun’ to place before functions to designate them as functions, and that we are defining a “try/detect” block in a similar format to Java’s “try/catch” to detect and handle in-game events, where events function similarly to Java exceptions.

The data types that we plan to include in our language include standard types and predefined objects designed to streamline the process of game design. The standard types include integers, floats, doubles, strings(these will work like Java strings), booleans, and characters.

The predefined object types include “Screen Objects”, “Player Objects”, “Entity Objects”, and “Event Objects”.

Screen Objects will serve as graphical bases that other objects can be positioned upon. They will call SDL’s initialization functions to simplify the process of rendering a screen. The Screen objects will also contain a listen(player,player,....) function that will take in Players and continuously check for and instantiate KeyPressEvents. When a key is pressed while the game is running, listen() will see if the key is in any of the player’s lists of attached keys, and if so it will instantiate a KeyPressEvent.

Player Objects will hold initial positions, a list of keys that are attached to the player, hitboxes, textures, some predefined movement functions and keyboard input for any “playable” element that can be rendered on top of the screen. Player objects will have the methods addHitBox() and addControls().
Entity Objects will contain positions, hitboxes, textures, and some predefined movement functions for non-playable elements of games (walls, obstacles, balls). The Player is a child of Entity, as its purpose is mainly to add keyboard input functionality and the two classes have many of the same features.

Lastly, we will have an Event Object, which works like an exception in Java. Event objects are designed to be used within “try/detect” blocks, while the “game” is running. Similar to exceptions in Java, events occur during the execution of a program and change the normal behavior of the program. There are two different built in types of events in our language: KeyPressedEvents, and CollisionEvents. A KeyPressedEvent is raised when the program detects hardware input with SDL.

A CollisionEvent is instantiated and raised when the program detects a collision between objects in the game. There will also be a preset collide() function in the CollisionEvent type that will default to a bounce but can be overridden in child Event classes. Our language uses the try-and-detect block, which mimics the try-catch in Java, to detect events raised by the program, and process the event by using the Execute statement. The Execute statement matches the case in the Event object and runs the case logic dictated in the object that matches the corresponding case (for example, a KeyPressedEvent instantiated with a keycode taken from SDL is processed into movement in a Player’s definition).

The Input/Output of our language is based on SDL I/O-- that is, we will be handling low-level keyboard and mouse input using SDL’s library and predefined keycodes. Our language should map our code to existing SDL code in order to manage library overhead, so it would first compile down to SDL C++ code, and then compile to LLVM. The language should have two types of output: console output and graphical output. The console output will display strings taken as arguments of the print() statement. The graphical output works using SDL’s connection to a graphics server, such as an X server. The graphics are then shown using OpenGL in a separate window. SDL handles the client connection to the graphics server, and the setup for this output connection will be handled in Screen Object types.

We will also be including preset drawing functions on top of current SDL drawing capabilities to make the rendering of shapes on the screen easier for the programmer.

Lastly, as mentioned in the description of Entity Types and Player Types, we will be including preset physics-based movement functions to facilitate the coding of interactions between entities. These functions will exist in the Entity Types (and by inheritance, the Player Types). These functions will be based on vector movement, and the move() function will take direction, length, speed and frame rate over time as arguments and run a loop to move the entity across the screen.

Notes about code: The "main" method serves as our primary submission for source code. The class definitions show how the default classes in our languages could be overridden.

```
main() {

        Screen app = new Screen();
        app.init();


        Player player1 = new Player(x,y);
        player1.addControls(up,down,right,left);
        player1.addHitBox(x1,x2,y1,y2);


        Player player2 = new Player(x,y);
        player2.addControls(w,s,a,d);
        player2.addHitBox(x1,x2,y1,y2);

        Ball ball = new Ball(x,y);
        ball.addHitBox(x1,x2,y1,y2);

        while(1){
                // listen for key pressed

                try {

        app.listen(player1,player2);
                        ball.move(env, env.p1,
        env.p2);

                } detect (KeyPressedEvent key) {
                        key.press();
                } detect (CollisionEvent coll) {
                        coll.collide();
                }
        }

}
```

```
public class Ball extends Entity{
        int round;
        int x1,x2,y1,y2;

        public Ball(x,y,vel_x, vel_y, round){
                super(x,y,vel_x,vel_y);
                this.round = round;

        }

        Fun addHitBox(x1, x2, y1, y2) {
                this.x1 = x1;
                this.x2 = x2;
                this.y1 = y1;
                this.y2 = y2;
        }

        Fun move(Screen screen, Player p1, Player
p2) {

                int x_prime = x + vel_x;
                int y_prime = y + vel_y;
                int x1_prime = x1 + vel_x;
                int x2_prime = x2 + vel_x;
                int y1_prime = y1 + vel_x;
                int y2_prime = y2 + vel_x;

                if (y2_prime  <= 0) {
                        return new
ScoreCollisionEvent(this,p2,screen);
                }

                if (y1_prime  >= screen.height) {
                        return new
ScoreCollisionEvent(this,p1, screen);
                }
                if (x1_prime  <= 0) {
                    return new
ScreenBounceCollisionEvent(this,0);
                }
                if (x2_prime  >= screen.width) {

                    return new
ScreenBounceCollisionEvent(this,screen.width);
                }
                if (y1_prime  <= p1.y2 &&
(x2_prime >= p1.x1 && x1_prime <= p1.x2) {
                    return new
PlayerHitCollisionEvent(this,p1);
                }
                if (y2_prime  >= p2.y1 &&
(x2_prime >= p2.x1 && x1_prime <= p2.x2) {
                    return new
PlayerHitCollisionEvent(this,p2);
                }
                x = x_prime;
                y = y_prime;
                x1 = x1_prime;
                x2 = x2_prime;
                y1 = y1_prime;
                y2 = y2_prime;
        }
}

Public class KeyPressedEvent extends Event {
        Player player;
        Keycode keycode;

        KeyPressedEvent(Player p, Keycode k){
                this.player = p;
                this.keycode = k;
        }

        press(){
                switch(keycode){
                    Case 'left': player.x++;
                                 player.x1++;
                                 player.x2++;
                                Break;
                    Case 'right': player.x--;
                                  player.x1--;
                                  player.x2--;
                                Break;

                }
        }
}

Public class ScreenBounceCollisionEvent extends
CollisionEvent{
        int wall_location;

        public ScreenBounceCollisionEvent(Entity e,
int loc){
                super();
                wall_location = loc;
        }


        collide() {
                        int dx;
                        if (wall_location == 0) {
                                dx = e.x1 *
        e.vel.x;
                        } else {
                                dx = e.x2 * e.vel.x
        - wall_location;
                        }
                        e.x = e.x * e.vel.x - 2*dx
                        e.x1 = e.x1 * e.vel.x - 2*dx
                        e.x2 = e.x2 * e.vel.x - 2*dx
                        e.y = e.y * e.vel.y;
                        e.y1 = e.y1 * e.vel.y;
                        e.y2 = e.y2 * e.vel.y;
                        e.vel.x = -e.vel.x;
                }

}

Public class ScoreCollisionEvent extends
CollisionEvent{
        Player p;
        Screen s;

        ScoreCollisionEvent(Entity e, Player
p,Screen s){
                super(e);
                this.p = p;
                this.s = s;
        }

        collide(){
                p.score++;
                if (e.round > 0) {
                        e.round = e.round - 1;
                        e.x = s.width / 2;
                        e.y = s.height / 2;
                 }
        }

}

Public class PlayerHitCollisionEvent extends
CollisionEvent{
```

```
Player p;

PlayerHitCollisionEvent(Entity e, Player p){
        super(e);
        this.p = p;
}


collide(){
        int dy;
        if (p.y1 == 0) {
                dy = e.y1 * e.vel.y;
        } else {
                dy = e.x2 * e.vel.y - p.y1
        }
        e.y = e.y * e.vel.y - 2*dy;
        e.y1 = e.y1 * e.vel.y - 2 * dy;
        e.y2 = e.y2 * e.vel.y - 2 * dy;
        e.vel.x = (e.x - p.x) / (p.x - p.x1 +
e.x2 - e.x) e;          //normalized in x-direction
        e.x = e.x * e.vel.x;
        e.x1 = e.x1 * e.vel.x;
        e.x2 = e.x2 * e.vel.x;
        e.vel.y = -e.vel.y;


    }


}
```