OCaml Smash Bros—Design

Kevin Greer (keg84)
Sahil Kanjiyani (ssk225)
Kevin Shen (ks864)
Yungton Yang (yty4)

12 November 2015

1 System Description

1.1 Proposal

We intend to build a game system that resembles Nintendo's Super Smash Bros. Our project will be a fighting game with multiple playable characters each with different attributes.

1.2 Key Features

- Engine
- GUI
- AI
- Characters

1.3 Narrative

Super Smash Bros. is a fighting game in which characters seek to launch each other off of the stage and out of the map. Instead of having a health meter, players have a percentage meter which rises as they take damage. A player's percentage indicates how much further they will be launched by an attack relative to an arbitrary amount said to be "0%".

To knock out an opponent, the player must launch them outside the arena's boundary in any direction. However, when a character is hit off of the stage, they can attempt to recover by jumping and using abilities to get back onto the stage.

Each character has a similar set of abilities: four attacks (up, right, down, left). Also, each character has different speeds, weights, and strengths. A character's speed affects not only his running speed but also his attack speed, a character's weight affects how far he is launched (i.e. heavy characters get launched less far), and a character's strength affects the amount of damage his attacks do to opponents as well as their launching power.

The game could either be played against an AI.

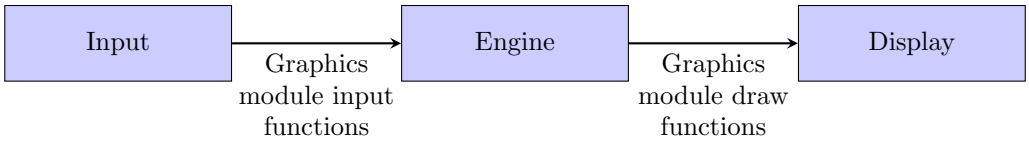

Figure 1: Components and connectors diagram

2 Architecture

Our system has three main components: input, engine, and display. First, the player will enter input through the keyboard. This input will be gathered by functions in the Graphics module. Once the engine receives the player input, it will process the input, determine an appropriate response and execute that response. Meanwhile, the engine will calculate all changes that occur in the game during this particular frame (i.e. affects of gravity, attacks, AI responses, etc.). Once the engine has processed the new state of the game, it will draw a representation of this state to the screen using drawing functions defined in the Graphics module. Figure 1 concisely summarizes this architecture in a components and connectors diagram.

3 System Design

The important modules that we will implement are Character, GUI, AI, and Engine.

The Character module contains the information modeling each character. It will contain a record for each character with mutable fields including the character's weights, speed, attacks, position, etc. Functions in the Character module (attack, run, change velocity) will change the fields of the character. In a sense, all of these functions will be update functions. This design choice was made because we thought it made sense that only the Character module should be updating fields of characters in the game. This would make the Character module encompass all of handling of Character data, and that would in turn make the module more strongly coupled.

The GUI module will use the Graphics module of OCaml to open a graphics window and draw the stage, background, and the characters. It will redraw the updated characters on every tick of the game clock. The GUI module will have a pair of Character records to draw each of their locations and animations. It will also have a draw function that will use the Graphics module functions to draw to the screen. More specifically there will be a pair of objects to draw at every tick, and the draw function will iterate through all of them to draw everything. Most of the drawing and animations will be done through drawing custom images that we will have to create for the characters, stage, and events. Relative positions of where to draw these things will come from the Character record.

The AI module will implement the behavior of the Artificial Intelligence. Given the state of the game, the AI module will decide what the character should do. The state of the game will be given to the main function of the module by having the function take in the Character pair. The information from this will then be used to calculate, in a sense, fake inputs that the CPU should do. These fake inputs will be design decisions by us when we implement the game. After these fake inputs are generated, they will be converted into actions the physics engine should do for the CPU characters.

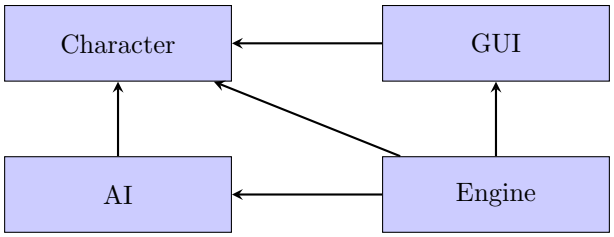

Figure 2: Module dependency diagram

The Engine module will handle most of the processing of the data. It will process inputs, execute inputs, handle collision detection, and take care of the physics of the game. It will handle gravity, calculate launch velocities, determine if characters can move or not based on hit stun, and calculate anything to do with the characters and change their positions. The main function of the Engine module is the tickprocessor function which will be run concurrently with a recursive input function that call waits on event [key pressed] to process inputs from the user. The tickprocessor delays its thread so that it runs at 60 frames a second. Based on the keys pressed, the tickprocessor function will parse an action in the game, then call the appropriate function to execute the action. This will be split into two main functions which process movement and attacks. Calculation will be done here to see if this is a valid input. For example, a person can't attack if they are stunned. Also, character velocity needs to be changed if a jump is processed. When the appropriate game response is calculated, the Engine will call functions in the Character module to update its information. Furthermore, on every tick, an AI response will also be generated. To make our game responsive, the Input processing and physics calculations will be done on separate threads from the drawing.

The interaction between the modules is illustrated by Figure 2.

4 Module Design

4.1 character.mli

```
(** A character in OCaml Smash that is of type t.  *)

type point = {
   x:  int;
   y:  int
}
type rect = point * point
type attack = Up | Down | Left | Right

type t = {
   mutable hitbox:  rect;
   mutable percent:  int;
   mutable stun:  int;
   mutable air:  bool;
   mutable velocity:  point;
   mutable jumps:  int;
   mutable lives:  int;
   mutable current_attack:  attack option;
   range:  int;
```

```
   speed:  int;
   weight:  int;
}

type guy = Light | Medium | Heavy

val attack_length :  int

(** * [get_width g] returns the width of character g *)

val get_width :  t -> int

(** * [get_height g] returns the width of character g *)

val get_height :  t -> int

(** * [create g] creates a new character that is of type g, which is of type guy.  *)
val create :  guy -> point -> t

(** * [set_position x p] moves the character x to point p *)
val set_position :  t -> point -> unit

(** * [start_attack c a] simulates sets the current_attack of c to a.  *)
val start_attack :  t -> attack -> unit

(** * [start_attack c a] simulates sets the current_attack of c to None.  *)
val stop_attack :  t -> unit

(** * [stun x len] stuns updates the stun field for character x to len.
   * This means the character x becomes stunned for len seconds.  *)
val stun :  t -> int -> unit

(** * [set_jumps c i] sets the number jumps c has remaining to be i *)
val set_jumps :  t -> int -> unit

(** * [get_hit x dmg] adds dmg to the damage field of the character x.  *)
val get_hit :  t -> int -> unit

(** * [change_velocity x vel] updates the velocity of the character x.  *)
val change_velocity :  t -> point -> unit

(** * [reset x] takes a life off character x.  *)
val reset :  t -> unit
```

4.2 gui.mli

```
(** [setup_window() opens the game window, appropriately titles it, and draws
   * the background and the stage.  *)
val setup_window :  unit -> unit

(** [draw (c1,c2)] draws all the needed elements to the screen:  the supplied
   * characters, the background, the stage, any animations, and status boxes.  * *)
val draw :  Character.t * Character.t -> unit

(**
   * [start_blast x y vert up player] begins a blast animation with a base centered
   * at the point (x,y).  The blast if drawn vertically if [vert] is true (i.e.  the
   * player dies on the top of the screen or the bottom) and is drawn horizontally
   * otherwise (i.e.  the player dies on the left or right side on the screen).  The
```

```
   * [player] parameter specifies the player who triggered the blast (for coloring)
   * purposes:  1 for player one and 2 for player two.  If [up] is true, the blast
   * will be drawn either upwards or rightwards appropriately based on [vert].
   *
   * Example usuage:
   * [start_blast 100 0 true true 1] -- player 1 has died on the bottom of the
   * screen 100 pixels from the left.  So we want a vertical blast facing up.
   * [start_blast 0 235 false true 2] -- player 2 has died on the left of the
   * screen 235 pixels from the bottom.  So we want a horizontal blast facing
   * right.
   * [start_blast stage_height 400 true false 2] -- player 2 has died on the top
   * of the screen 400 pixels from the left.  So we want a vertical blast
   * facing down.
   * [start_blast stage_width 500 false false 1] -- player 1 has died on the right
   * of the 500 pixels from the bottom.  So we want a horizontal blast facing
   * left.
   *
   * Note:  the engine only has to call this once for each blast.  The animation
   * stops when the blast ends and the blast erases itself.
*)
val start_blast :  int -> int -> bool -> bool -> int -> unit

(** [draw_end winner] draws the end of game message says that player number
   * [winner] has won the game with instructions for playing again or exiting.
*)
val draw_end :  int -> unit

(** [start_countdown sec] starts a countdown in the middle of the screen that
   * that decreases once per second in the format \sec, sec-1, ..., 1, GO!"
*)
val start_countdown :  int -> unit
```

4.3 engine.mli

```
type move = MLeft | MRight | MDown | MUp

(** [input_loop ()] is a recursive loop that processes inputs continuously.  *)
val input_loop :  unit -> unit

(** [start_engine ()] starts the engine.  *)
val start_engine :  unit -> unit
```

4.4 ai.mli

```
(** [execute_response_to_state c i] is the AI's response to character i's state.  *)
val execute_response_to_state :  Character.t -> Character.t -> string
```

5 Data

The data for all of our game will be maintained as a Character.t pair. Each element will represent all of the necessary information for one character. A Character.t is just a record containing many pertinent fields such as position, velocity, stun, strength, speed, jumps left, damage taken, and attacks. This record will be mutable so changes can be made to it as things get updated. The other modules will access these characters by the engine passing

them as arguments. Because, we have such a small amount of objects on the screen, we won't need any data structures to increase our efficiency. The biggest efficiency problem we face is collision detection between characters, but a rudimentary bounding box solution will give us a fast implementation of this feature.

6 External Dependencies

We will be using the Graphics module to draw and handle input. The Graphics module has functions to process mouse/keyboard inputs.

7 Testing Plan

For our project, we had a very extensive testing plan.

We tested engine by using environments that isolated the component we wanted to test. For example, to test `process_move`, we created an environment where we only had one character, and we would ensure that left inputs would result in a left movement (the signature of this function was of type unit, so this was the best way to test movement). To test `process_attack`, we had a debug mode where characters would change color if they were stunned (characters who attack and characters who are hit change color). We had an environment where the player could move and attack, with another stationary character that we could test attacks on.

We tested our character module with careful unit tests. We intensively unit tested each function in character, such as `change_velocity` and `hitbox_at_point`. These tests basically ensured that our math was correct.

We tested our AI module primarily with simulations. Our AI read the state of the game and reacted accordingly, so we simulated environments where the AI would have to react in different ways (player is off stage, player is on stage, player is in the air, player is stunned, etc).

We tested our GUI module by simply looking at the graphics. The numbers in our GUI module were tweaked by how it looked on the screen. We think this was the best way of testing it because there was less math involved and more of just seeing what was drawn where relatively to other things (ex. we modified the numbers of the stage so it showed up in the center, then modified the numbers of the characters until they were in appropriate places on the stage).

8 Division of Labor

8.1 Kevin Greer

Kevin implemented the Character module and the GUI module. In the Character module he created functions that can be easily called by other modules to edit the state of a character. In the GUI he designed and created functions to draw the background, stage, characters, and everything that can be seen by the user. He implemented animations for character attacks, as well as animations for character deaths. To achieve this, Kevin worked very closely with the graphics module. He worked for about 13 hours on this project.

8.2 Sahil Kanjiyani

Sahil implemented a lot of the engine. He created the architecture for reading inputs, reading AI inputs, and processing game events at 60 frames per second. All of this was done concurrently to reduce latency. Furthermore, he implemented the functions which allow the characters to move, collide with the stage, update their positions due to gravity and momentum, and check whether they are dead. He also created the replay system. In total, he spent about 15 hours writing code for the project.

8.3 Kevin Shen

Kevin implemented a collision algorithm and process_attack. He calculated the hitboxes for all the attacks, along with the appropriate interactions. Also, Kevin refactored a lot of the code in engine when we decided to change certain aspects of our design. He worked an estimated 10 hours on the project.

8.4 Yungton Yang

Yungton's primary contribution to the project was his work with the artificial intelligence and his in-depth knowledge of the fundamentals of "Super Smash Brothers," the game after which our project was based. He calculated the position of the Player in relation to the AI and made the AI react accordingly. For example, if the Player is to the left of the AI but not close enough to reach with an attack, the AI will move left. But if the AI is close enough, then the AI will perform a left attack. He spent about 6-7 hours coding with the artificial intelligence.