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, attacks, and special moves.

1.2 Key Features

- Engine
- GUI
- AI
- Characters
- Multiplayer

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: three ground attacks (up, down, left/right), five aerial attacks (one in each direction + neutral), and four special attacks (up, down, left/right, neutral). However, the specific moves of each character are unique, and this makes the characters feel different to play with. Players can shield to protect themselves from attacks, but the shield can only withstand a certain amount of damage. Also, each character has different speeds, weights, and strengths. A character's speed affects not only

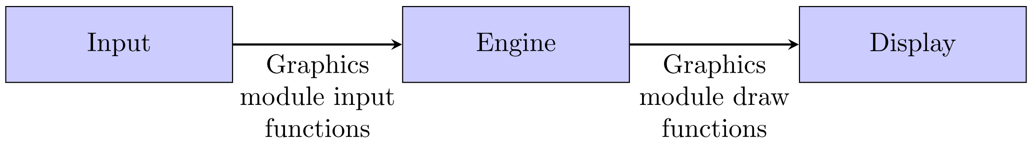
Input
Engine
Display
Graphics module input functions
Graphics module draw functions

Figure 1: Components and connectors diagram

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 another player or against an AI.

2 Architecture

Our system has three main components: input, engine, and display. First, the player(s) 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 list 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 list 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

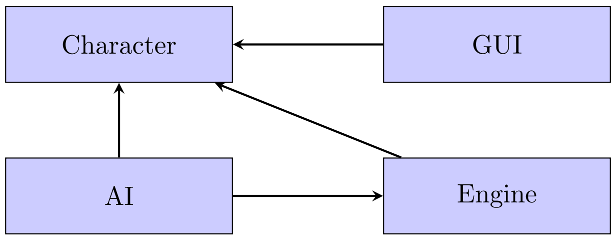

Figure 2: Module dependency diagram

the main function of the module by having the function take in the Character list. 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. These actions will be given to the engine by calling its exposed functions.

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 input function which will recursively call wait on event [key pressed] to process inputs from the user. Based on the key pressed, the input function will parse this into 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 t
type point
type rect
type guy

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

(** [attack x] simulates an attack by the character x.  *)
```

```
val 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

(** [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

```
(*List of Character.t*)
type observed

(*Draws every character and the stage and background*)
val draw : observed -> unit
```

4.3 engine.mli

```
type attack

type move

(** character is a list of characters in the current game. *)
val character : Character.t list

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

(** [process_attack a i] executes an attack from player i. *)
val process_attack : attack -> int -> unit

(** [process_move m i] executes a movement. *)
val process_move : move -> int -> 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 list -> int -> unit
```

5 Data

The data for all of our game will be maintained as a Character.t list. 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. All of the modules will be able to access this for their computations because we will declare it as an exposed value in our Engine module. 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

We will unit test as we go along. For example, when we code a function in the Character module, such as `change_velocity`, we will use extensive unit tests to ensure that the function works as intended. As a team, we will review each other's test cases and try to break each other's specifications using abstraction (black box testing). Finally, we will test by playing the game.