Cross validation of the universe teachpack of Racket in OCaml

Chihiro Uehara, Kenichi Asai

Department of Information Science
Ochanomizu University
Tokyo, Japan

2 June 2015

Our study

- Cross validation of the universe framework in OCaml
  1. Creating the library that offers the same functionality as the universe teachpack of Racket (FFFK, 2009)
  2. User testing
     - To confirm that the library is (to certain extent) equally useful in OCaml as in Racket
  3. Analyzing how the universe framework fits in OCaml
- We call the library we implemented the universe library
- `http://pllab.is.ocha.ac.jp/~asai/Universe/`

Explanation of the universe library

- Demo
- Universe framework
- How to use the universe library

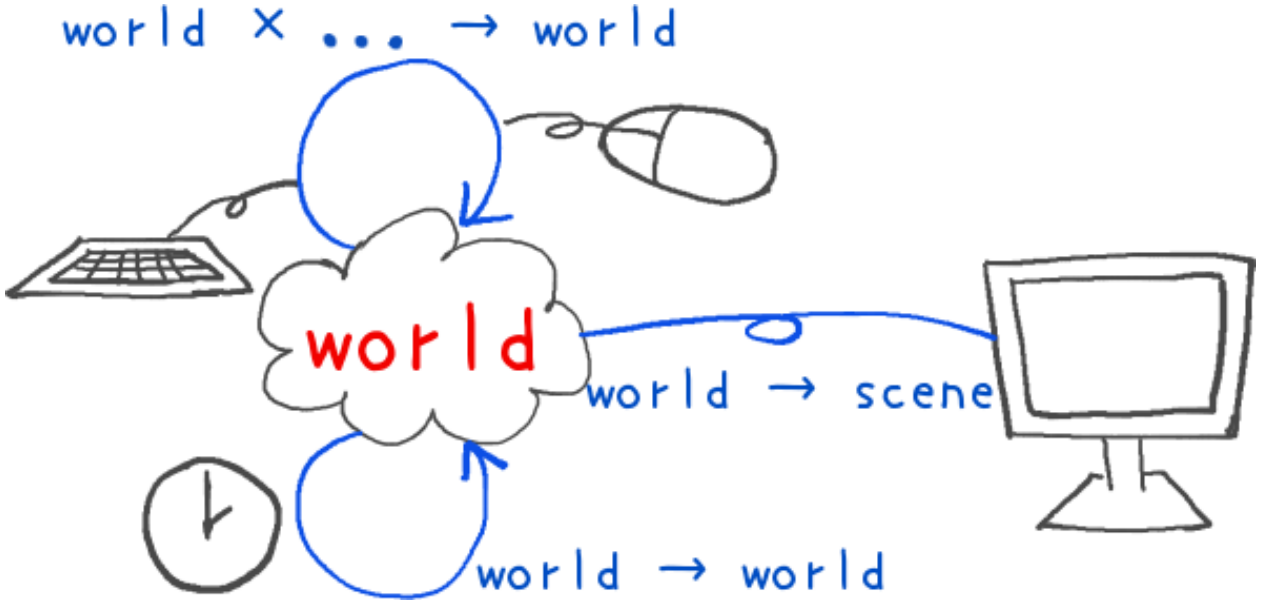
Demo

- Stand-alone game made with the universe library
  - 94 lines
  - We use this game in the explanations in the following slides

Universe framework

- Users identify states
- Interaction is regarded as transition from old states to new states
- The state is called *world* in stand-alone games
  - All the information to specify the state uniquely
  - World-passing style that takes *world* around
- e.g. universe teachpack

World-passing style

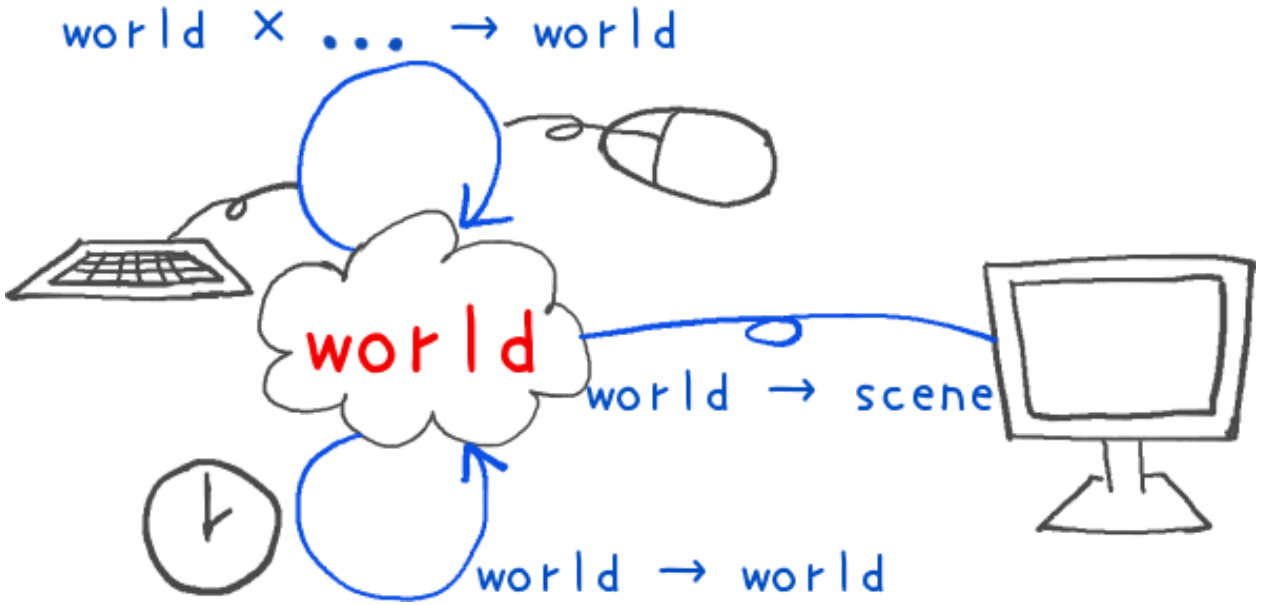


- A draw function that creates a game screen from the world
- Event handlers that return a new world according to the current world and events

World-passing style

- Calculating a new world from the old world does not involve mutation
  - The style goes well with the introductory courses
  - Students need to know only the function definition and basic algebra, which are the main initial focus of the introductory programming courses

A stand-alone (client) program



* Identify what constitutes the world
  * `Draw` that creates a game screen from the world
  * `Move_on_tick` called at every time interval
  * `Change_on_mouse` called on mouse click
* Registration of all definition to `big_bang`

A stand-alone (client) program

- Identify what constitutes the world of the game

```
 (* type of world *)
type world_t = ball_t list      (* a list of balls *)

 (* function called at every time interval *)
 (* move_on_tick : world_t -> (world_t, 'a) World.t *)
let move_on_tick world =
  let new_world = List.map move_ball_on_tick world in
  World new_world
```

- The type `('a, 'b) World.t` is defined in the library:

```
type ('a, 'b) t = World of 'a
                | Package of 'a * 'b
```

A stand-alone (client) program

- Identify what constitutes the world of the game

```
 (* type of world *)
type world_t = ball_t list      (* a list of balls *)

 (* function called at every time interval *)
 (* move_on_tick : world_t -> (world_t, 'a) World.t *)
let move_on_tick world =
  let new_world = List.map move_ball_on_tick world in
  World new_world
```

- The type `('a, 'b) World.t` is defined in the library:

```
type ('a, 'b) t = World of 'a
                | Package of 'a * 'b
```

A stand-alone (client) program

- Identify what constitutes the world of the game

```
 (* type of world *)
type world_t = ball_t list      (* a list of balls *)

 (* function called at every time interval *)
 (* move_on_tick : world_t -> (world_t, 'a) World.t *)
let move_on_tick world =
  let new_world = List.map move_ball_on_tick world in
  World new_world
```

- The type `('a, 'b) World.t` is defined in the library:

```
type ('a, 'b) t = World of 'a
                | Package of 'a * 'b
```

A stand-alone (client) program

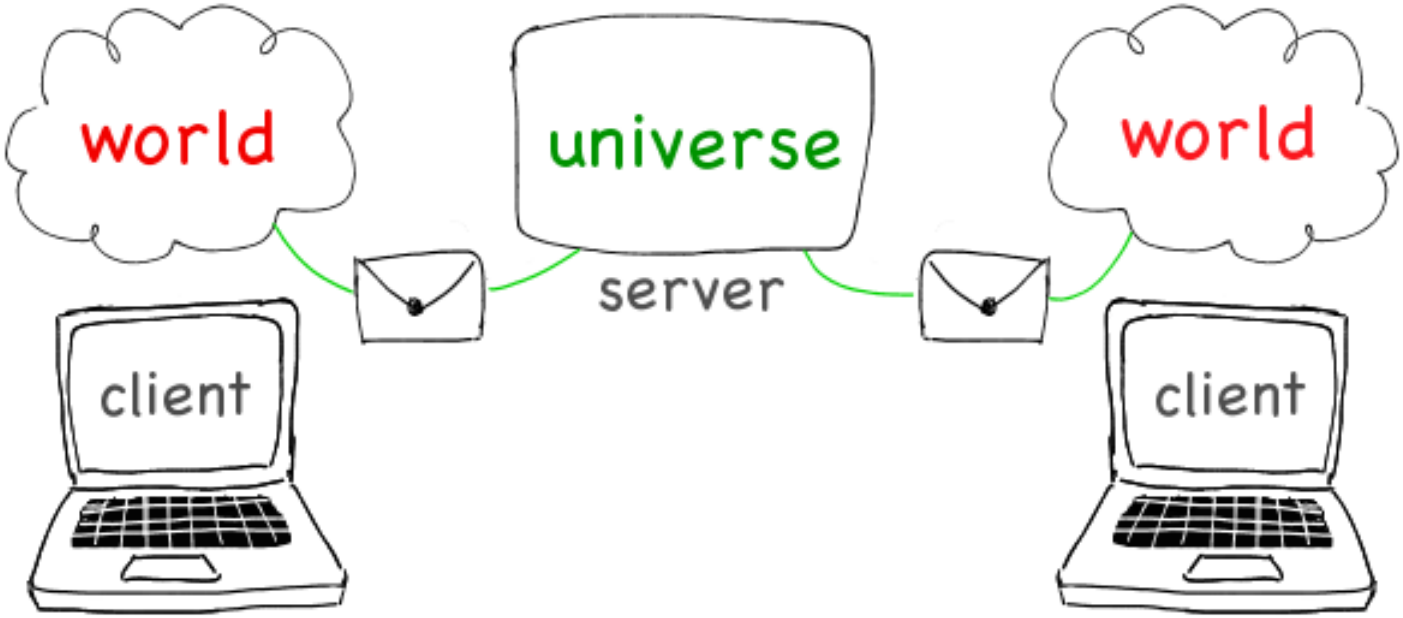
```
 (* register necessary definitions to big_bang, and start *)
let _ =
big_bang initial_world     (* initial value of world *)
         ~name:"BallGame" (* name of screen *)
         ~to_draw:draw     (* draws a game screen along world *)
         ~width:widt       (* width of the screen *)
         ~height:height    (* height of the screen *)
         ~on_mouse:handle_mouse
                           (* called on mouse click *)
         ~on_tick:move_on_tick
                           (* called at every time interval *)
         ~rate:0.1         (* time interval to call on_tick *)
         ~stop_when:game_over
                           (* checks whether the game is over *)
```

Demo

- Communicating two-person game made with the universe library
  - 166 lines
  - We use this game in the explanations in the following slides

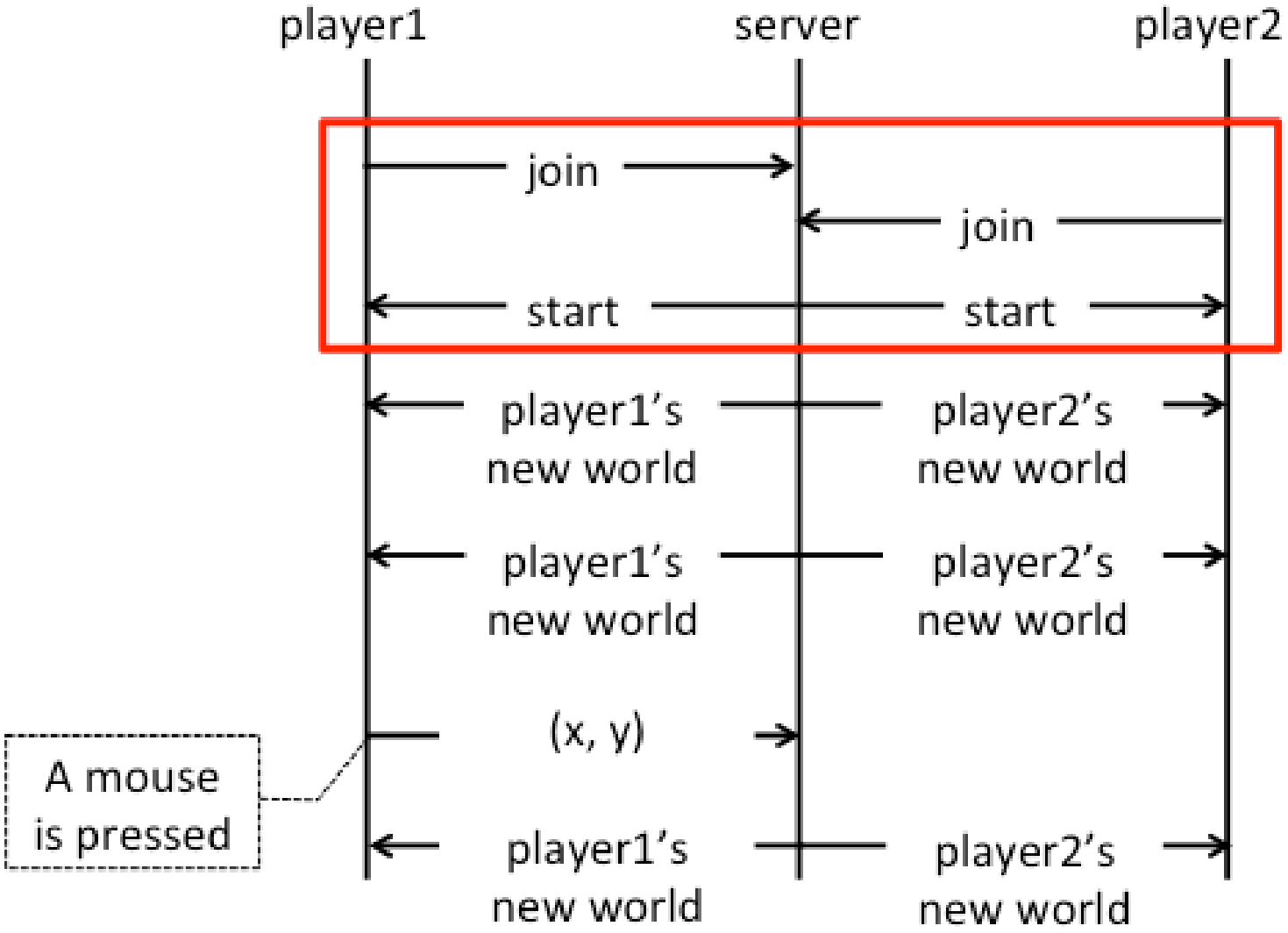
Communicating games

- A server and clients

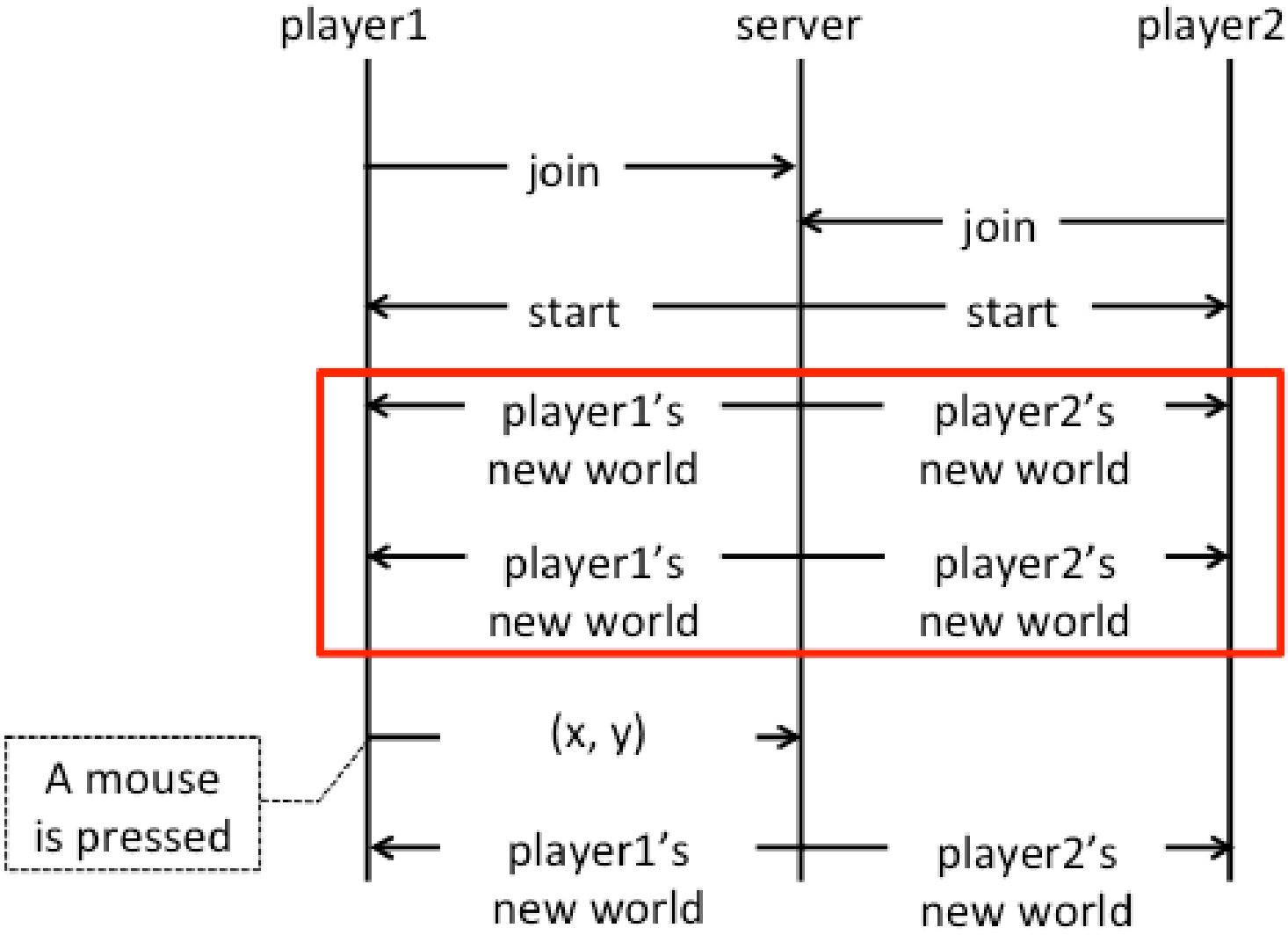


- It is recommended to write a communication diagram (Morazan, 2013)

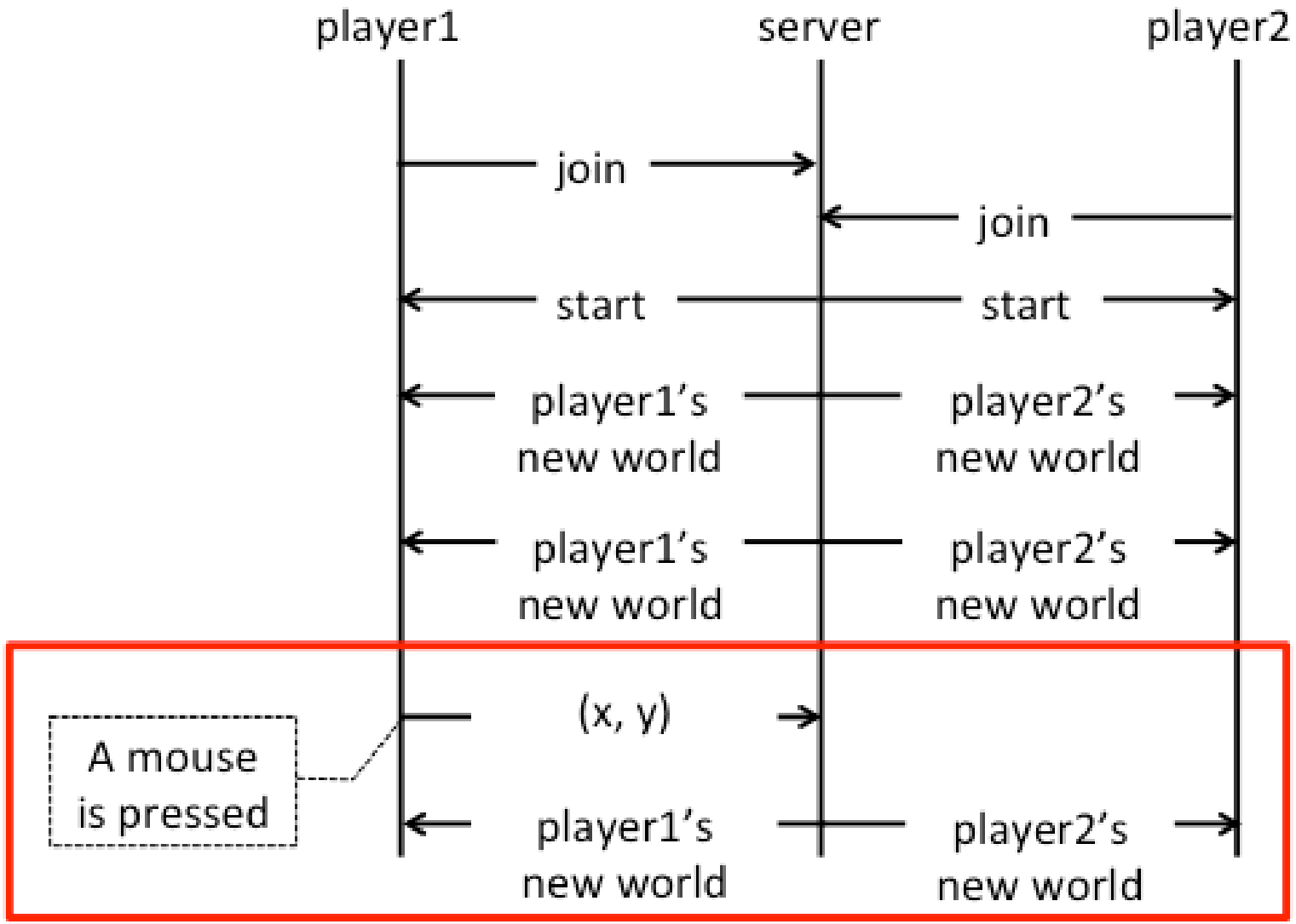
A game starts



Worlds change at every time interval



A mouse is pressed



Communicating client program

```
 (* type of the world *)
type world_t = ball_t list    (* a list of my balls *)
             * ball_t list    (* a list of rival's balls *)

 (* function called when a message is received *)
 (* receive : world_t -> world_t -> (world_t, 'a) World.t *)
let receive world message = World message

 (* function called on mouse click *)
 (* handle_mouse : world_t -> int -> int -> string
                   -> (world_t, int * int) World.t *)
let handle_mouse world x y event =
  if event = "button_down" then Package (world, (x, y))
  else World world
```

- Registration of all definition to big_bang

Communicating client program

```
 (* type of the world *)
type world_t = ball_t list    (* a list of my balls *)
             * ball_t list    (* a list of rival's balls *)

 (* function called when a message is received *)
 (* receive : world_t -> world_t -> (world_t, 'a) World.t *)
let receive world message = World message

 (* function called on mouse click *)
 (* handle_mouse : world_t -> int -> int -> string
                   -> (world_t, int * int) World.t *)
let handle_mouse world x y event =
  if event = "button_down" then Package (world, (x, y))
  else World world
```

- Registration of all definition to big_bang

Communicating client program

```
 (* type of the world *)
type world_t = ball_t list    (* a list of my balls *)
             * ball_t list    (* a list of rival's balls *)

 (* function called when a message is received *)
 (* receive : world_t -> world_t -> (world_t, 'a) World.t *)
let receive world message = World message

 (* function called on mouse click *)
 (* handle_mouse : world_t -> int -> int -> string
                   -> (world_t, int * int) World.t *)
let handle_mouse world x y event =
  if event = "button_down" then Package (world, (x, y))
  else World world
```

- Registration of all definition to big_bang

Communicating client program

```
 (* type of the world *)
type world_t = ball_t list    (* a list of my balls *)
             * ball_t list    (* a list of rival's balls *)

 (* function called when a message is received *)
 (* receive : world_t -> world_t -> (world_t, 'a) World.t *)
let receive world message = World message

 (* function called on mouse click *)
 (* handle_mouse : world_t -> int -> int -> string
                   -> (world_t, int * int) World.t *)
let handle_mouse world x y event =
  if event = "button_down" then Package (world, (x, y))
  else World world
```

- Registration of all definition to `big_bang`

A Server program

- Keeps track of the state of all the world

```
(* type of the universe *)
(* a list of a pair of the client and its world *)
type universe_t = (iworld_t * ball_t list) list

(* function called at every time interval *)
(* move_on_tick : universe_t ->
                  (universe_t, world_t) Universe.t *)
let move_on_tick universe =
  let new_universe =
    List.map
      (fun (world_id, lob) ->
        (world_id, List.map move_ball_on_tick lob))
      universe in
  send_messages new_universe
```

  - Registration of all definition to universe

A Server program

- Keeps track of the state of all the world

```
 (* type of the universe *)
 (* a list of a pair of the client and its world *)
type universe_t = (iworld_t * ball_t list) list

 (* function called at every time interval *)
 (* move_on_tick : universe_t ->
                    (universe_t, world_t) Universe.t *)
let move_on_tick universe =
  let new_universe =
    List.map
      (fun (world_id, lob) ->
        (world_id, List.map move_ball_on_tick lob))
      universe in
  send_messages new_universe
```

  - Registration of all definition to universe

A Server program

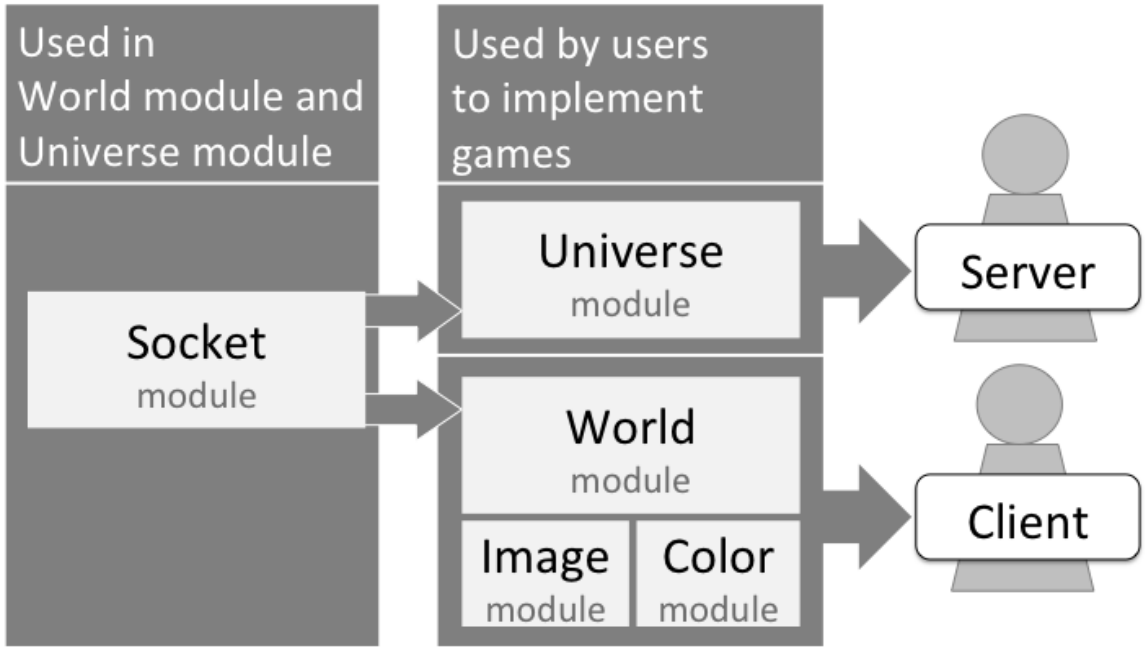
- Keeps track of the state of all the world

```
 (* type of the universe *)
 (* a list of a pair of the client and its world *)
type universe_t = (iworld_t * ball_t list) list

 (* function called at every time interval *)
 (* move_on_tick : universe_t ->
                    (universe_t, world_t) Universe.t *)
let move_on_tick universe =
  let new_universe =
    List.map
      (fun (world_id, lob) ->
        (world_id, List.map move_ball_on_tick lob))
      universe in
  send_messages new_universe
```

  - Registration of all definition to `universe`

Implementation of the universe library



- Images and handling of events are implemented using LablGtk2, the OCaml interface to GTK+
- The `Socket` module is implemented using the Unix module in the standard library

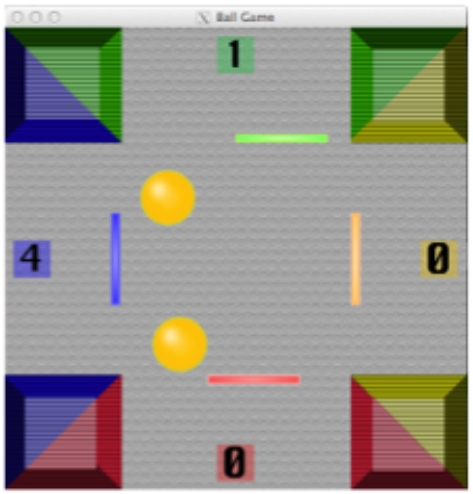
User testing

- A class to create communicating games using the library
- About 2 months once a week

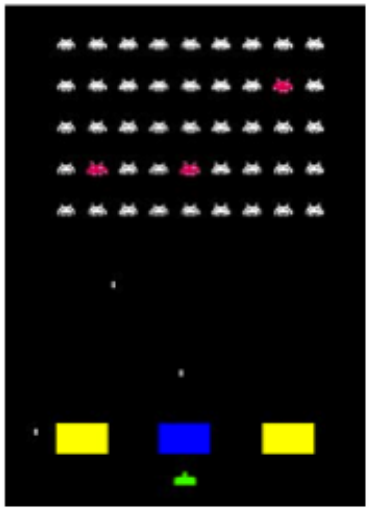
Year	Number	C	OCaml	Racket
1	2	○	×	△(1)
2	10	○	○	△(3)
3	3	○	○	○(3)

- They formed six teams each consisting of two or three students

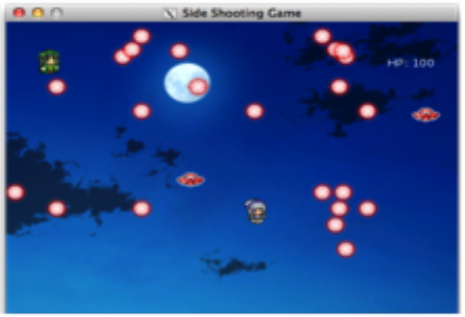
Games created by students



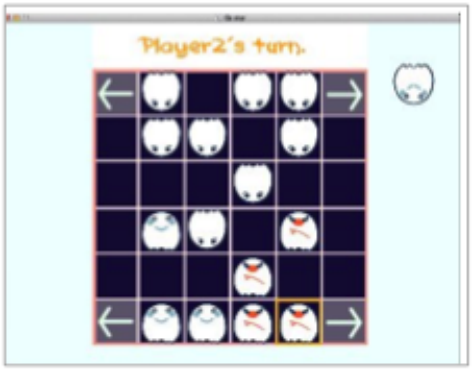
Three 3rd-year students



Three 2nd-year students



Two 2nd-year students



Two 2nd-year students

Three 2nd-year students

Two 1st-year students

Comments on the universe library

- After the class finished, we asked students:
  - Advantages of the universe library (advantages)
  - Shortcomings of the universe library (shortcomings)
  - How they compare the universe library with the universe teachpack of Racket (comparison)
  - Other comments (others)
- We consider
  1. What turned out to be good
  2. What students find difficult
  3. The universe library in education

  - We use some of the comments today
  - All comments are in the paper

1. What turned out to be good

- Ease of use due to the interface which is similar to the universe teahcpack (advantages, 15)
- It is attractive that I can see the result of what I program in the screen (others, 2)
- No support for sounds (shortcomings, 2)
  - This came from their desire to make better games
- One of big reasons for enthusiasm is the attractive environment provided by the library

2. What students find difficult

- To synchronize time and values (others, 1)
  - By struggling with this problem, the users must have deepen their understanding of synchronization
  - It is a good platform to encourage students to think about synchronization
- Some games maintained time events in each client independently and failed synchronization
- The universe framework does not necessarily lead to a uniform solution
  - We gave brief explanation, but should have explained more

3. The universe library in education

- 13 students had already finished the introductory OCaml course
  - They found programming in the library interesting
  - They took part in the course enthusiastically and with success
  - The universe library is suitable for the CS2 course

3. The universe library in education

- Two students programmed in OCaml for the first time
  - They created a working communicating game
  - The universe library is easy enough for beginners
  - We have not yet incorporated the universe library into our introductory OCaml course, but the library appears to be ready to be included in it

Analysis

- How the universe framework fits in OCaml
  1. Influence of static typing
  2. Debugging
  3. Testing
  4. Error messages

1. Influence of static typing

- When we change definition of the world or the universe, we need to make changes to all the places where they are mentioned
  - Racket provides a special mechanism to check the shape of the world at runtime
- With static typing, they are detected as type errors
  - Second-year and third-year students could detect them as type errors because the are used to OCaml
  - We didn't need to make a special mechanism as Racket

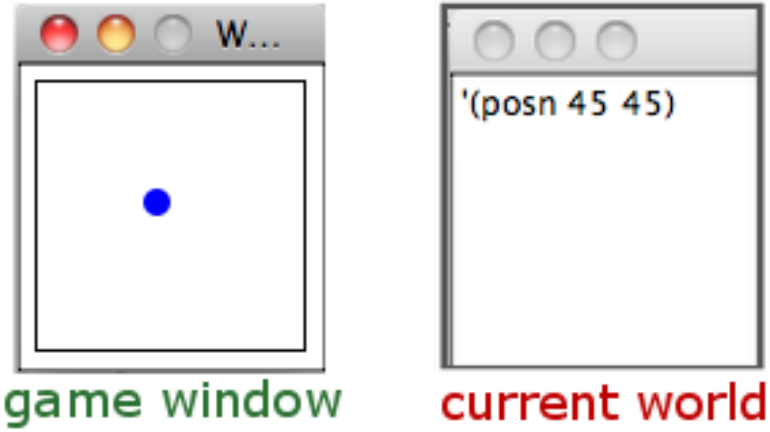
1. Influence of static typing

It is hard both to fix the type of world and universe at first and to change them later (comparison, 2)

- Regardless of the language, students struggle to make a suitable definition of world and universe
- Static typing does help searching for the necessary changes when the type of world and universe changes

2. Debugging

- Racket shows the current world in a separate window



- To show the contents of world in OCaml, one writes explicitly printing statements
  - Printing is typically covered only at the end of introductory functional language courses
  - It is desirable to support the functionality as in Racket in the universe library

3. Testing

- Testing is important (HtDP, 2001)
  - We have not come up with a test method to check whether interactive games are running as specified
- We can do unit testing (as in Racket)
  - We instruct students to write basic test cases:
    `let test = (program = result)`
- Racket integrated environment displays the test coverage by highlighting not executed parts
  - We have not tried a coverage tool for OCaml (Bisect) yet with the universe library

4. Error messages

Readable error messages are important to understand what is going on in a program, especially for beginners

- Type error
  - Standard type errors arise (as you expect)
  - The type debugger (IA, 2014) helps navigating the user to the source of the type error by asking questions
- Runtime error
  - We have no support for runtime errors
  - Error messages are hard to understand (shortcomings, 4)
  - considered in detail in next slides

Error messages for the runtime error

- A specified image file was not found
- The user launches a client before a server

- A server stops while clients are still running

- Exception in the call-back functions
- Type mismatch of sender and receiver

Error messages for the runtime error

- A specified image file was not found
- The user launches a client before a server
  - The user gets a readable error message
- A server stops while clients are still running

- Exception in the call-back functions
- Type mismatch of sender and receiver

Error messages for the runtime error

- A specified image file was not found
- The user launches a client before a server
  - The user gets a readable error message
- A server stops while clients are still running
  - Clients continue to run except that they no longer receive any messages from the server
  - When they send messages to the server, no error arises but the sent messages are simply ignored
  - The user doesn't get any error message
- Exception in the call-back functions
- Type mismatch of sender and receiver

Error messages for the runtime error

- A specified image file was not found
- The user launches a client before a server
  - The user gets a readable error message
- A server stops while clients are still running
  - Clients continue to run except that they no longer receive any messages from the server
  - When they send messages to the server, no error arises but the sent messages are simply ignored
  - The user doesn't get any error message
- Exception in the call-back functions
- Type mismatch of sender and receiver

Exception in the call-back functions

```
In callback for signal button_press_event,
  uncaught exception: Not_Found
```

- It does not show the name of the user-defined function that raised the exception
- It shows the signal name that is used internally in the underlying GTK+
- It is hard for beginners to understand the error message

```
In on_mouse function, uncaught exception: Not_Found
```

- To wrap the functions with a `try ... with`
  - When they are registered to show which handler raised an exception

Exception in the call-back functions

```
In callback for signal button_press_event,
  uncaught exception: Not_Found
```

- It does not show the name of the user-defined function that raised the exception
- It shows the signal name that is used internally in the underlying GTK+
- It is hard for beginners to understand the error message

```
In on_mouse function, uncaught exception: Not_Found
```

- To wrap the functions with a `try ... with`
  - When they are registered to show which handler raised an exception

Exception in the call-back functions

```
In callback for signal button_press_event,
  uncaught exception: Not_Found
```

- It does not show the name of the user-defined function that raised the exception
- It shows the signal name that is used internally in the underlying GTK+
- It is hard for beginners to understand the error message

```
In on_mouse function, uncaught exception: Not_Found
```

- To wrap the functions with a `try ... with`
  - When they are registered to show which handler raised an exception

Type mismatch of sender and receiver

- Messages are marshalled before sent
  - The type of the marshalled data is lost during communication
- If a program unmarshalls received data as a value of different type
  - The program crashes with `Segmentation fault`

```
(Marshal.from_channel chan : type)
```

Anything can happen at run-time if the object in the file does not belong to the given type (from OCaml manual)

Type mismatch of sender and receiver

- To check the consistency of the type of messages statically appears to be fundamentally difficult
- Telling students to check the type of messages whenever `Segmentation fault` occurs

Conclusion

- Cross validation of the universe framework in OCaml
- The universe library is a nice environment for students to write interactive games, as the universe teachpack
- There is room for improvement, especially in error messages
- Most of the students who took the course had already finished the CS1 course
  - We have not incorporated the universe library into the CS1 course yet (as was done in HtDP2e)