CSC 591
Homework 3

Aurora Tiffany-Davis
(attiffan)

TABLE OF CONTENTS

1. Part 1 - Time

1.1. Timelines

1.1.1. REAL Time

Clock time, expressed as milliseconds since epoch.

1.1.2. GAME Time

Anchored to REAL time, expressed as milliseconds since the game has started.

1.1.3. LOOP Time

Expressed as iterations of the game loop.

1.1.3.1. PLAY LOOP Time

Expressed as iterations of the game loop while the game is not paused. This timeline also keeps track of the time delta (game time elapsed during single iteration of the game loop). This information is used to keep Client- and Server-controlled game objects moving at the correct speeds even when Client and Server frame rates differ. This would enable a future design which has the Server and Client on different computers which are able to realize different frame rates. To develop / test this feature, Server and Client frame rates were explicitly changed using the Processing .frameRate() method.

1.1.3.2. REPLAY LOOP Time

Anchored to PLAY LOOP time. Expressed as iterations of the play loop during replay of a recorded game section. Replay speed is determined by this timeline's tick size. For example, if tick size is 2, then each game loop iteration during replay will replay 2 iterations of the recorded play loop time ("double time" replay). Replay is discussed in Part 3.

1.2. Pausing the Game

1.2.1. Explicit / Implicit Pause

The game may be paused explicitly by the user by pressing 'P'. Explicit pause was incorporated to assist in understanding timing during development. The game is paused implicitly during replay and replay recording (discussed in Part 3).

1.2.2. Global Pause

Every pause is global. That is, a Client cannot pause locally without pausing the entire game. This decision was made to avoid code complexity not warranted by the requirements of the assignment. Specifically, if a Client is paused locally, but the Server continues to move platforms around the game world in service of un-paused Clients, then what should happen when the paused Client is resumed? Game pause is communicated among the Server and all Clients by way of an event. Events are discussed in Part 2.

1.2.3. Restricted Resume

The Client whose actions paused the game is the only Client permitted to resume (un-pause) the game. If the Client whose actions paused the game leaves the game, then the Server resumes the game automatically. This automatic resumption was added in order to keep the graceful shutdown which was ironed out in Homework 2.

1.2.4. Meaning of a Pause

While the game is paused, neither the Server nor any Client "ticks" forward the play loop timeline. While the game is paused, the Server does not move any platforms as it normally would during each game loop iteration.

1.3. Timers

A timer feature is added to the game which permits a named timer to be started, restarted, stopped, read, or destroyed. Timers track elapsed time in 3 timelines: Real time, Game time, and Play Loop time. Timers were used during development to understand the runtime characteristics of various subsystems, and are used in the Performance Test discussed in Part 4. This feature will enable easier performance measurement in any future versions of the game.

2. Part 2 - Event Management System

2.1. Multiple Event Management Models

The game supports two different event management models. The default event management scheme is Distributed Event Management, which is discussed here. The alternative event management scheme is discussed in Part 4.

2.2. Event Structure

An event is composed of the following attributes.

2.2.1.1. nTimeReal_ms - Real time, as described in the previous section of this report

2.2.1.2. nTimeGame_ms - Game time, as described in the previous section of this report

2.2.1.3. nTimePlayLoop - Play loop time, as described in the previous section of this report

2.2.1.4. nEventType - Event type

2.2.1.5. bClient - True if the event originated on a Client, otherwise false

2.2.1.6. bHandledByOriginator - True if the event has been handled by the originator, otherwise false

2.2.1.7. oArgs - A hashmap, the keys of which are strings and the values of which are objects. This structure was chosen for flexibility, so that different event types could effectively have different arguments.

Furthermore, an event also has an attribute nPlayerID, inherited from HW03_ObjectCommunicable, which indicates the player ID of the originating Client (or -1 if originated on the Server).

2.3. Event Types

WILDCARD, ADMIN, USER_INPUT, COLLISION, SCORE_CHANGE, SPAWN, REPLAY, GAME_OBJECT_CHANGE, GAME_PAUSE

2.4. Event Registration

The event manager tracks registration by means of a hashmap whose keys are event types and whose values are a collection of interested objects. If a Client or Server object is interested in an event type, it calls an event manager method to register for notifications about that event type.

Optionally, it may also register its interest in that event type from other network partners (Server / Clients). In this case, a remote registration object is communicated out. When the network partner receives that remote registration object, it registers a network partner proxy as an object that is interested in the event type.

If a new Client joins the game, the Server is responsible for sending them any remote registration objects that have already been sent to existing Clients.

2.5. Event Raising

If a Client or Server object wishes to raise an event, it calls an event manager method which creates a timestamped event object and adds that object to a prioritized queue. Events are prioritized first by event type (WILDCARD < ADMIN < USER_INPUT < [remaining event types] < COLLISION) and then by timestamp (with recent events being higher priority).

2.6. Event Scheduling

A separate event manager thread repeatedly pulls an event from the prioritized queue, handles it, pulls another event, etc. Event prioritization ensures that the most important events are handled first.

2.7. Event Handling

When the event manager handles an event, it gets from the event registration hashmap a collection of all interested objects. It then calls the event handling method of each interested object, which is implemented via the HW03_EventObserver interface. It then does the same for all interested objects which are associated with the event type of WILDCARD.

Earlier, it was noted that during event registration, an object may register interest in an event type from other network partners. In this case, the event manager may find a network partner proxy object in the collection of objects that is interested in an event. To the event manager, this is just another interested object. The network partner proxy object will handle the event by communicating it to the appropriate network partner.

2.8. Changes to Design to Support Event Handling

2.8.1. Client-Server Communication

In Homework 2, all Client → Server communication was in the form of game objects, and all Server → Client communication was in the form of collections of game objects. In Homework 3, all communication is in the form of a single object, which may be a game object, an event object, or an event remote registration object.

Because the game now writes and reads single objects rather than collections, a further simplification was made. The Client and Server no longer use .wait() and .notify() to perform writes. Instead, the write thread simply uses .take(), a blocking dequeue operation, to get the next object to write out. This is the same strategy used to pull events from the event queue in the event manager and in the logger.

Both of these changes resulted in code that was more flexible and easier to understand, so this was a valuable exercise.

2.8.2. Collision Handling

In Homework 2, several game object types handled collision in part by backing away from the collided object. The method which supported backing away was inefficient and made assumptions about the state of game objects.

These drawbacks were not noticed when collision events were handled immediately. However, now that collision events are handled only after being queued (possibly with other more important events), backing away must be done efficiently, and without making so many assumptions. The game world may have moved on since the collision occurred.

The improvement of the code which supports backing away from a collision makes the game more flexible and easier to understand, so this was a valuable exercise.

3. Part 3 - Replays

3.1. What is Recorded and Why

Replays are handled entirely on the Client side. If they were handled by the Server, then it would be challenging to support different replays on multiple Clients at the same time, which is a feature that is not required by the homework assignment, but is a very reasonable feature to expect in a real system.

During replay recording and during replay itself, the entire game (the Server and all Clients) are considered to be in a paused state. This decision was made to avoid code complexity not warranted by the requirements of the assignment.

The Server is in charge of moving platforms around and handling events related to those platforms (e.g. collisions). We cannot expect collision handling help from the (paused) Server during replay.

The end result of the circumstances described above is that we record and act on only to those events which indicate the positions and deletions of game objects. This is not a problem, since a replay after all is only replaying game objects moving around on screen.

3.2. Replay Order of Operations

3.2.1. Start of Recording

Set attribute on all located objects which marks their location (x,y).

3.2.2. During Recording

Log all located object GAME_OBJECT_CHANGE events along with the object's current location. Refrain from deleting any character objects representing a disconnected client (hide them instead).

3.2.3. End of Recording

Pause the game globally.

3.2.4. Start of Replay

"Teleport" all located objects back to their original location according to the attributes set at the start of recording. Hide any character objects representing clients which were not yet connected at the start of recording. Show any hidden character objects representing clients which disconnected during recording.

3.2.5. During Replay

Read out the replay log and move located objects around on screen accordingly. Speed is determined by the tick size of the Replay Loop timeline, which is described in Part 1. Tick size / replay speed can be altered by user command during replay. Show or hide character objects at the appropriate time based on client connection / disconnection.

3.2.6. After Replay

Resume gameplay at normal speed. All located objects, having been moved throughout the replay, are back where they were at the end of replay recording. Delete any character objects representing clients which disconnected during recording.

3.3. Further Logging

As noted above, the Client logs GAME_OBJECT_CHANGE events in a replay log during replay recording. However, logging is also used more generally to support debugging. Both the Client and the Server also log (in separate files) all events of which they are made aware.

3.4. Tracking Replay State

Replay state is tracked so that it can be referenced by the additional on screen banners (described below), by the game loop, and by the keypress observer, to determine the correct action to take at any point during the game. Replay states are IDLE, RECORDING, WAITING_TO_REPLAY, and REPLAYING.

3.5. Challenges / Limitations

Properly handling the case where Client A is recording while Client B connects or disconnects from the game was a bit of a challenge. This may have been more of a challenge due to the specific replay implementation of teleporting existing game objects to show the replay.

It is assumed that the user will not wish to explicitly pause recording replay itself (this is not supported)

3.6. Changes to Design to Support Replays

3.6.1. Additional On Screen Banners

In Homework 2, there was one on screen banner, showing a scoreboard of all players' scores. Now there are two additional on screen banners to help the user understand the increasingly complex gameplay. First, Game Status ("*Have Fun!*", "*Replaying your recording…*", etc.). Second, Available Commands, depending on game status (e.g. "*'Q' quit, 'P' pause, 'R' start recording, '<-' go left, '->' go right, [SPACE] jump*").

3.6.2. Incorporation with Event Management

Few changes were needed to the event management model to support Replay. This is not to imply that implementation of event management was easy. However, after working through the Pause feature (described in Part 1), the communication of events (described in Part 2), and the delayed collision handling (described in Part 2), event management was pretty solid.

Obviously, a needed change was simply to incorporate replays into the event system. To keep the event-driven nature of the game engine clear and explicit, the Replay Manager handles USER_INPUT events in order to trigger REPLAY events. That is, user input does not directly trigger a REPLAY event, but rather it starts a chain that goes through USER_INPUT. The Replay Manager handles REPLAY events to track replay state, to save located object positions at the start of recording, to pause the game at the end of recording, to teleport objects at the start of replay, and to resume (un-pause) the game at the end of replay.

The Logger handles REPLAY events to kick off the replay log.

4. Part 4 - Performance Comparison

4.1. Server-Centric Event Management

The game supports two different event management schemes. The alternative event management scheme, Server-Centric Event Management, is discussed here. This section discusses only those aspects which differ from Distributed Event Management, which is discussed in part 2.

4.1.1. Event Registration

This functionality is the same as in Distributed Event Management. Even in a Server-Centric scheme, a Client still must have some means of tracking which Client objects are interested in which event types.

4.1.2. Event Raising

Instead of adding the new event object to a queue, a Client sends the event to the Server for management.

4.1.3. Event Scheduling

Because the Client has sent the event to the Server for management, it is the Server which performs event scheduling. If prioritization of events as described for Distributed Event Management results in two events with the same priority, the tie is broken in favor of the event that has not yet been handled by its originator (that is, the Client who has passed its event to the Server for management).

4.1.4. Event Handling

The Server handles events which have been sent to it for management both in the normal way (if there are any objects which have registered interest in the event), and by sending the event back to the originating Client for handling. When a Client receives such an "echo" event, it handles the event immediately rather than queueing it.

4.1.5. Limitations

As noted by the instructor, it is not required that the game be fully debugged in the alternative event management scheme.

4.2. Performance Testing

4.2.1. Metrics Collected

This test is performed on the Client side. The test kicks off a burst of fake collisions of the character object with the death zone, and measures the elapsed time until the character object is re-spawned once for each death zone collision. This test exercises the following:

4.2.1.1. Communication between client and server

4.2.1.2. The interaction of multiple events: COLLISION → SCORE_CHANGE → SPAWN

4.2.1.3. Event Registration (registering for SPAWN events)

4.2.1.4. Event Raising (raising COLLISION events)

4.2.1.5. Event Scheduling (queuing of the COLLISION, SCORE_CHANGE, and SPAWN events)

4.2.1.6. Event Handling (handling the SPAWN events)

4.2.2. Parameters Varied

4.2.2.1. Number of collision events in the "burst" (higher number means more events raised on Client)

4.2.2.2. Event Management Scheme (distributed or Server-centric)

4.2.2.3. Number of moving platforms in the game (higher number means more events raised on Server)

4.2.3. Results

4.2.3.1. Effect of Varying Number of Collision Events

As expected, the more collision events in the "burst" (i.e. the more events raised on Client), the longer the elapsed time to resolution of these events on the Client.

4.2.3.2. Effect of Varying Event Management Scheme

The results were striking. The Server-Centric model is significantly less efficient than the Distributed model. It is expected that events will be handled less efficiently in the Server-centric model, because any event raised by a Client must first be communicated to the Server, then be scheduled for handling, then be communicated back to the Client, and finally be handled. However, the drastic performance difference was surprising.

Through debug print statements (since removed from the code), it was confirmed that the delays are occurring between the Server placing the "echo" event on the object write output stream, and the Client receiving the "echo" event on the object read input stream.

This delay is thought to be due to the streams getting overwhelmed by a huge number of events flying back and forth. Events are generated quite often in the game. One example is that due to "gravity", character objects are always attempting to move down, and usually are stopped by colliding with platforms. In other words, there are collision events being raised on nearly every game loop iteration.

In the Distributed Event Management model, these collisions are quickly handled locally by the Client. In the Server-Centric Event Management model, these collisions (and any knock-on events after them) trigger network traffic, resulting in extreme inefficiencies.

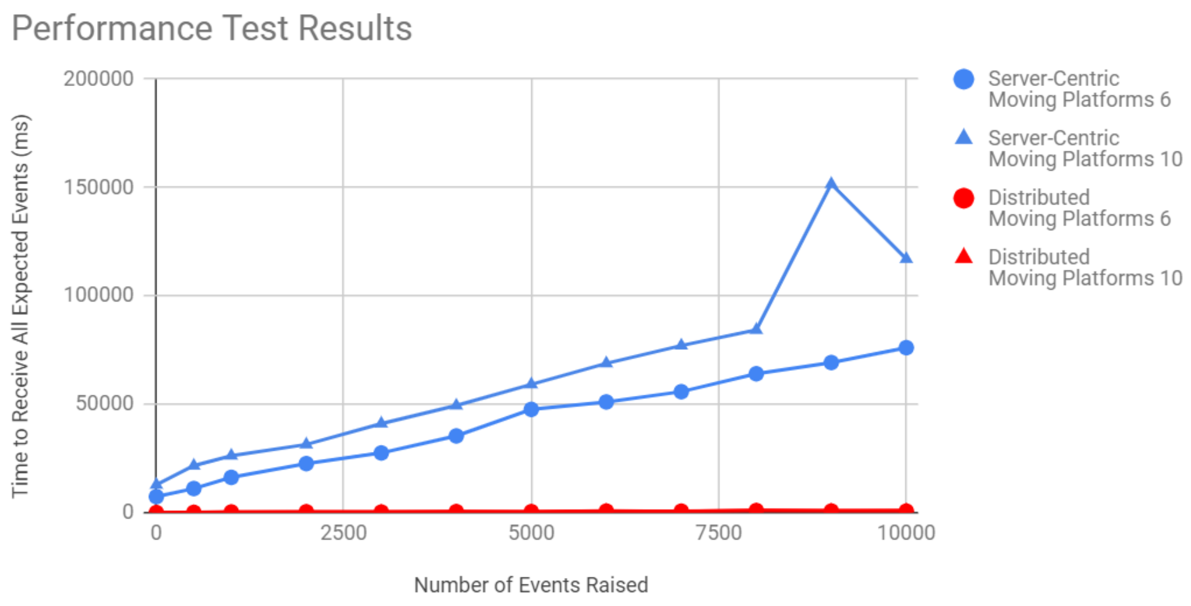
4.2.3.3. Effect of Varying Number of Moving Platforms

Increasing the number of moving platforms in the game exacerbated the delays in the Server-Centric model, by further increasing the number of events raised and therefore the network traffic. Increasing the number of moving platforms in the game had no noticeable effect in the Distributed model.

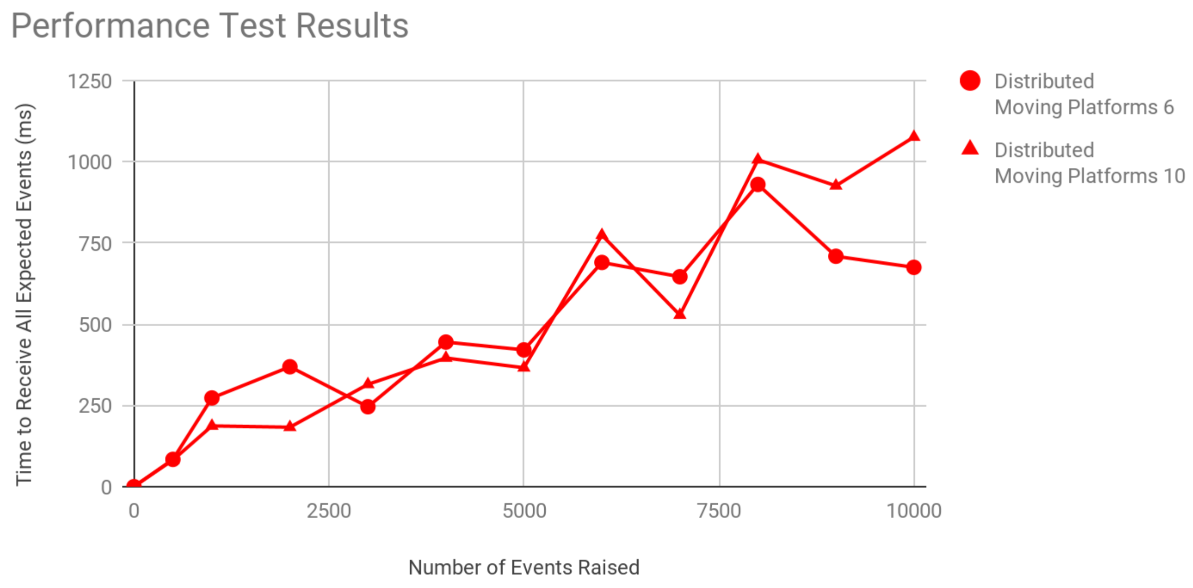
5. Appendix

5.1. Performance Test Results

5.1.1. Distributed vs. Server-Centric Event Management

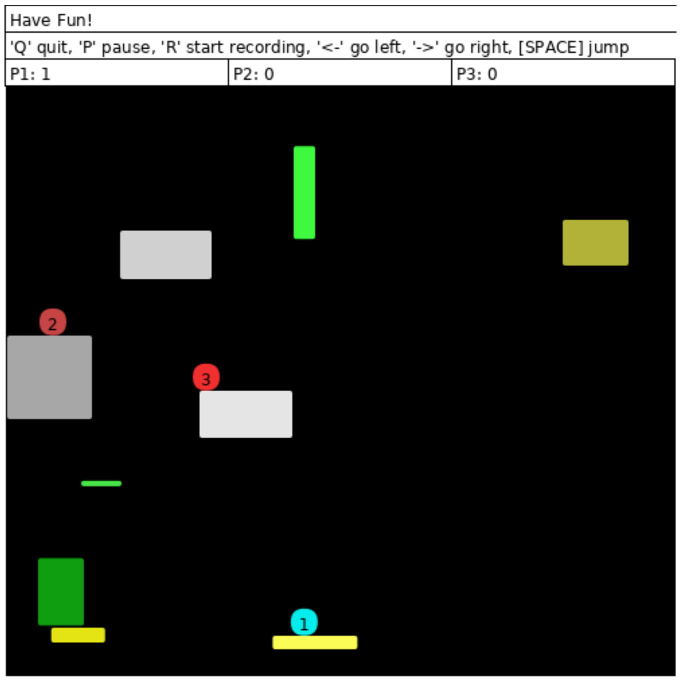


5.1.2. Distributed Event Management (detail)



5.2. Screenshots

5.2.1. Multi-Player Game (Player 1 View)

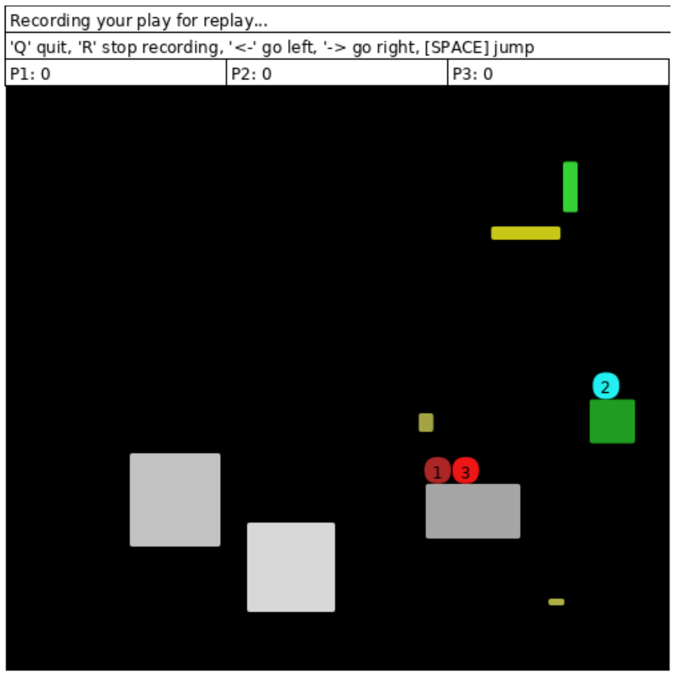


Have Fun!

'Q' quit, 'P' pause, 'R' start recording, '<-' go left, '->' go right, [SPACE] jump

P1: 1 P2: 0 P3: 0

5.2.2. Multi-Player Game, Recording for Replay (Player 2 View)

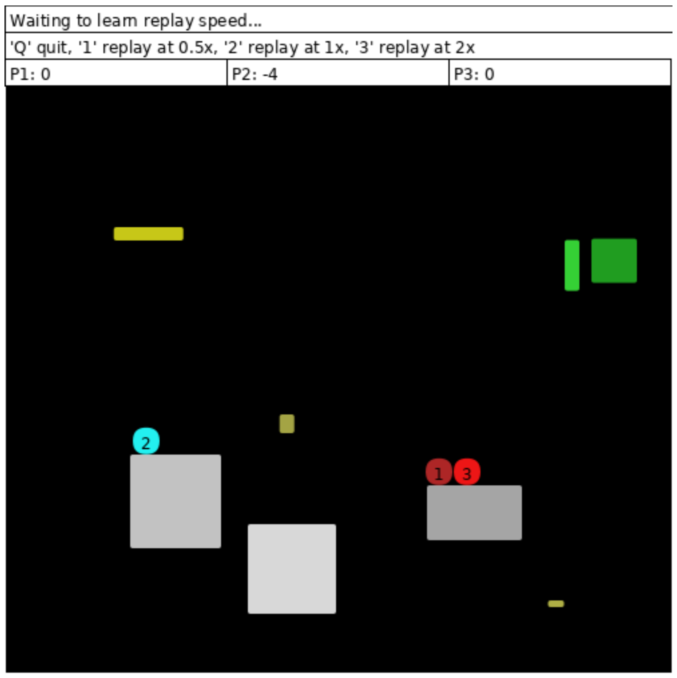


Recording your play for replay...

'Q' quit, 'R' stop recording, '<-' go left, '-> go right, [SPACE] jump

P1: 0 P2: 0 P3: 0

5.2.3. Multi-Player Game, Waiting to Learn Replay Speed (Player 2 View)

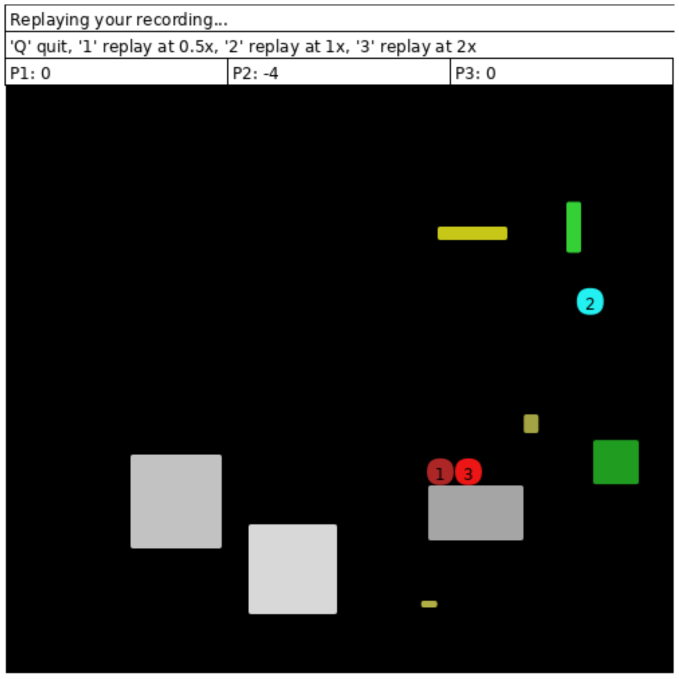


Waiting to learn replay speed...

'Q' quit, '1' replay at 0.5x, '2' replay at 1x, '3' replay at 2x

P1: 0 P2: -4 P3: 0

5.2.4. Multi-Player Game, Replaying (Player 2 View)

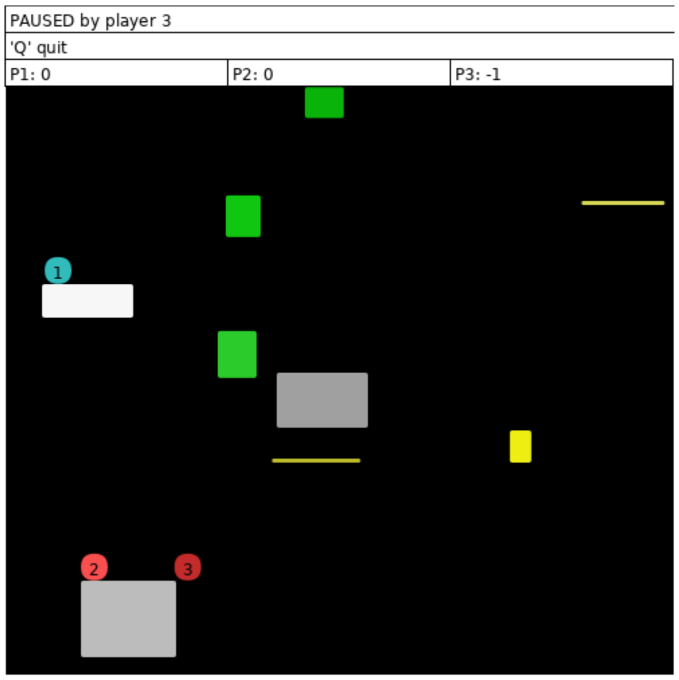


Replaying your recording...

'Q' quit, '1' replay at 0.5x, '2' replay at 1x, '3' replay at 2x

P1: 0 P2: -4 P3: 0

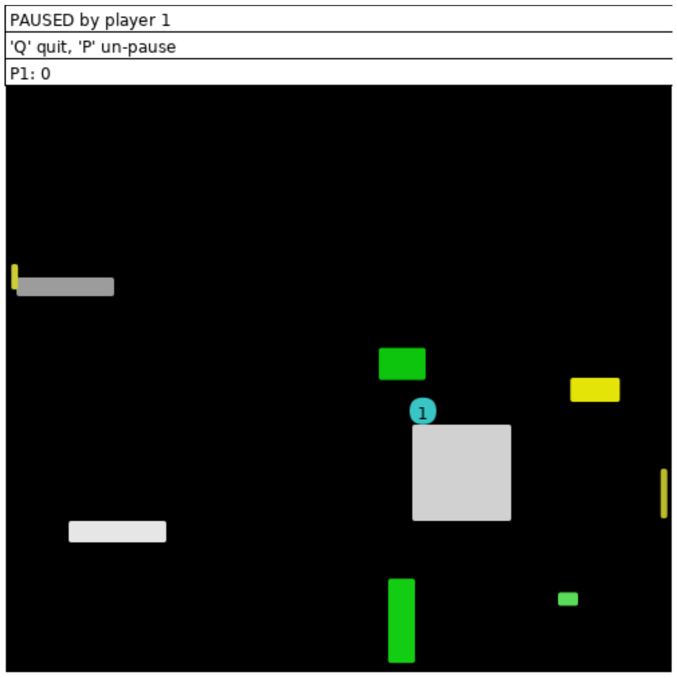
5.2.5. Multi-Player Game, Paused by Player 3 (Player 1 View)


PAUSED by player 3

'Q' quit

P1: 0 P2: 0 P3: -1

5.2.6. Single-Player Game, Paused


PAUSED by player 1

'Q' quit, 'P' un-pause

P1: 0