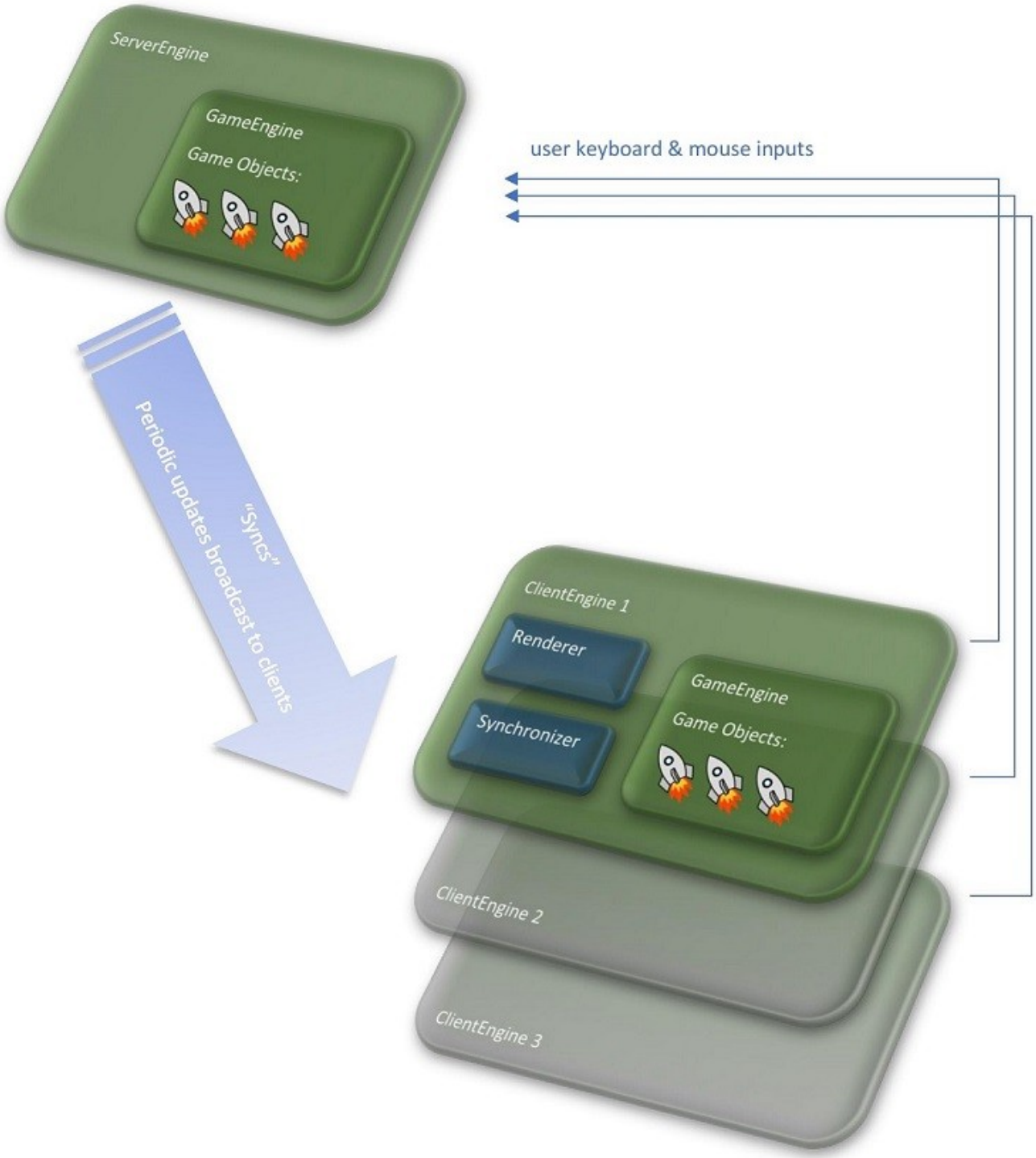
How the Lance Game Engine Works

Brooke Hedrick
brooke.t.hedrick@gmail.com
9.17.2018

High Level Interaction Diagram



http://docs.lance.gg/r3.0.0/tutorial-overview_architecture.html

Server Flow

The server logic is implemented by the server engine, which must do the following: (1) it must initialize the game, (2) it must accept connections from players over a socket, (3) it must execute the game loop, by calling the game engine's step() method at a fixed interval, and (4) it must broadcast regular updates to all the clients at a fixed interval.

The server engine schedules a step function to be called at a regular interval. The flow is:

- **ServerEngine - start of a single server step**
  - **GameEngine - read and process any inputs that arrived from clients since the previous step**
  - **GameEngine - start of a single game step**
    - **PhysicsEngine - handle physics step**
  - **If it is time to broadcast a new sync**
    - **for each player: transmit a "world update"**

http://docs.lance.gg/r3.0.0/tutorial-overview_architecture.html

Client Flow

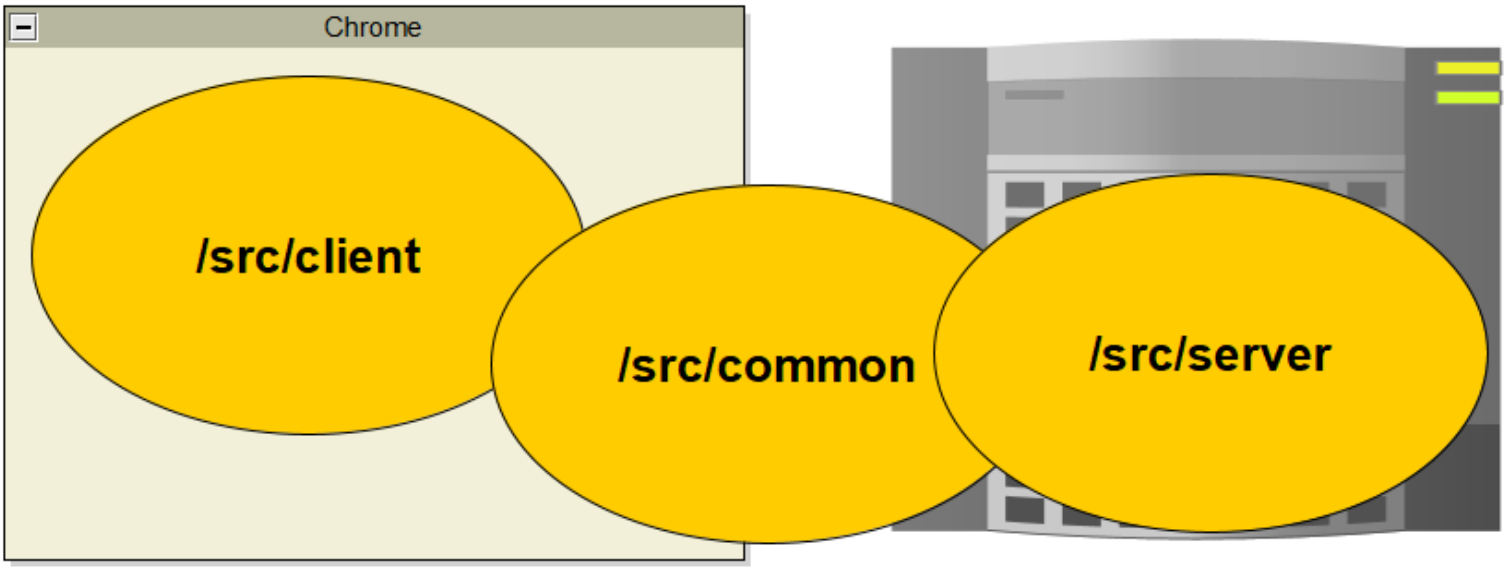
The client flow is more complicated than the server, for two reasons. First it must listen to syncs which have arrived from the server, and reconcile the data with its own game state. Second, it must invoke the renderer to draw the game state.

- **ClientEngine - start of a single client step**
  - **check inbound messages / syncs**
  - **capture user inputs that have occurred since previous step**
  - **transmit user inputs to server**
  - **apply user inputs locally**

- **ClientEngine - start of a single render step**
  - **Renderer - draw event**
    - **GameEngine - start of a single game step - may need to be executed zero or more times, depending on the number of steps which should have taken place since the last render draw event**
      - **PhysicsEngine - handle physics step**

http://docs.lance.gg/r3.0.0/tutorial-overview_architecture.html

Software Folders



/src/server

1. MyServerEngine.js

a. Initialize player variables

* start()

b. Tracks players connecting and disconnecting

* onPlayerConnected()
* onPlayerDisconnected()

/src/client

1. MyClientEngine.js

a. Set up which keypreses to listen to
    * constructor(gameEngine, options)

b. Specify the renderer

2. MyRenderer.js

a. Initialize the sprite list
    * constructor(gameEngine, clientEngine)

b. Update the player and ball positions
    * draw(t, dt)

b. Add items to the sprites list
    * addSprite(obj, objName)

3. clientEntryPoint.js

a. Initialize default client options
    * Can also be read from URI?!

b. Create a gameEngine and clientEngine

/src/common

1. Ball.js

a. Initialize a ball object. Set some defaults

* constructor(gameEngine, options, props)

b. Add a ball into the game via event & sprites

* onAddToWorld(gameEngine)

2. Paddle.js

a. Initialize a paddle object. Set some defaults

* constructor(gameEngine, options, props)

b. Add a paddle into the game via event & sprites

* onAddToWorld(gameEngine)

3. PlayerAvatar.js – UNUSED.

Continued...

/src/common

4. MyGameEngine.js

a. Allow serialization – data about paddle and ball move between client and server
   * registerClasses(serializer)

b. Add callback to handle postStep and objectAdded
   * start()

c. Handle player input (up/down keys)
   * processInput(inputData, playerId)

d. Add two paddle objects and a ball into the game
   * initGame()

e. Handle ball collision with borders or paddles
   * postStepHandleBall()