UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina Finale di

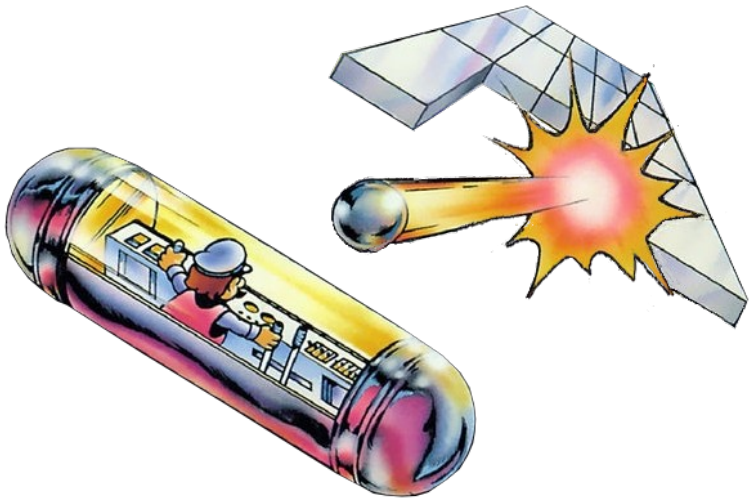
Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2018-2019

docente
Prof. Luca Grilli

JBrick

applicazione desktop JFC/Swing



293500 **Paolo Leopardi** paolo.leopardi@studenti.unipg.it

Data ultimo aggiornamento: 12 Febbraio 2020

Sommario

1. Descrizione del problema 3
    1.1 Il videogioco arcade Breakout 3
    1.2 L’applicazione JBrick 3

2. Specifica dei requisiti 4

3. Progetto 4
    3.1 Architettura del sistema software 4
    3.2 Logic 5
    3.3 View 6
    3.4 Problemi riscontrati 7

4. Conclusioni e sviluppi futuri 8

5. Bibliografia e sitografia 9

1. Descrizione del problema

L’obbiettivo di questo lavoro è lo sviluppo di un’applicazione desktop, denominata JBrick, che realizza una versione semplificata del videogioco arcade Breakout, realizzato da Atari a metà degli anni ‘70.

L’applicazione sarà implementata utilizzando la tecnologia JFC/Swing.

Di seguito vi è una breve descrizione del videogioco originale Breakout, in seguito si fornirà una descrizione della versione che si intende realizzare.

1.1 Il videogioco arcade Breakout

In Breakout lo scopo del giocatore è abbattere un muro di mattoni posto nella parte superiore dello schermo, mentre in quella inferiore c'è solamente una piccola barra che può essere mossa a destra e sinistra: con questa bisogna colpire una palla che rimbalza, in modo che distrugga tutti i mattoni che compongono il muro. I mattoni sono disposti su 8 file ed ogni coppia di file è disegnata con un colore diverso: dal basso in alto, giallo, verde, arancio e rosso. Ogni mattone colpito assegna un punteggio: 1 punto per i mattoni gialli; 3 punti per i mattoni verdi; 5 punti per i mattoni arancio e 7 punti per quelli rossi. La velocità della barra comandata dal giocatore aumenta dopo 4 colpi, poi dopo altri 12 colpi ed infine quando la pallina raggiunge le file di mattoni arancio e rossi. Quando poi questa sfonda l'ultima fila di mattoni rossi e colpisce il muro superiore, la barra dimezza la sua larghezza[1].

1.2 L’applicazione JBrick

Nell’applicazione grafica *JBrick* l’utente ha come obbiettivo quello di distruggere il muro di mattoni situato nella parte superiore dello schermo grazie a una sfera libera di muoversi nell’intera finestra di gioco. Il giocatore, attraverso i tasti direzionali, controlla una piattaforma che spostandosi orizzontalmente ha il compito di non far cadere la sfera al di sotto (causando la perdita di una vita) indirizzando quest’ultima verso le componenti del muro. Inizialmente si hanno a disposizione tre vite, l’esaurimento di queste causerà la terminazione del gioco; è possibile inoltre accumulare fino a un massimo di quattro vite con l’ausilio di *powerUp*. Esistono tre tipologie di *powerUp*: *extraLife*, incrementa di uno il numero delle vite a disposizione, *bigPaddle*, aumenta la dimensione della piattaforma, *bigBall*, aumenta la dimensione della sfera. I potenziamenti sono sparsi all’interno del muro e saranno disponibili solamente quando il relativo mattone verrà distrutto; la piattaforma per usufruirne dovrà intercettare la traiettoria. I mattoncini possono resistere a uno, due o tre colpi (la resistenza è individuabile dalle varie colorazioni) e la disposizione varia di livello in livello. Per raggiungere la vittoria è necessario completare tutti i livelli.

***Figura 1**: Schermata iniziale*

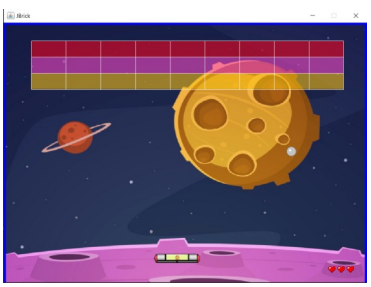


***Figura 2**: Schermata livello 1*

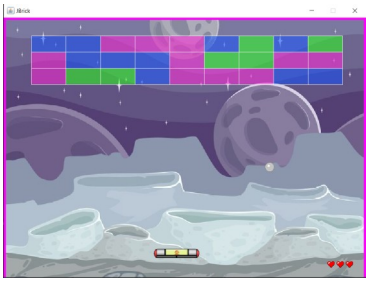


***Figura 3**: Schermata livello 2*

1 Breakout. *Wikipedia, L’enciclopedia libera.* https://it.wikipedia.org/wiki/Breakout_(videogioco)

2. Specifica dei requisiti

I requisiti possono essere distinti in due tipologie: requisiti grafici e requisiti interattivi.

Requisiti grafici

I requisiti grafici servono a rappresentare le varie situazioni a cui l'utente può andare incontro nell'avanzare del gioco.

- Schermata iniziale.
- Livelli con disposizioni differenti dei mattoncini.
- Mattoncini con colori differenti a seconda della resistenza ai colpi.
- Sfondo di gioco differente per ogni livello.
- Visualizzare le vite rimanenti.
- Presenza di *powerUp* nei livelli.
- Presenza di suoni.

Requisiti interattivi

I requisiti interattivi servono a gestire l'interazione fra l'utente e l'applicativo.

- Spostamento della piattaforma tramite tasti direzionali.
- Possibilità di mettere in pausa il gioco.
- Scorrimento da una schermata a un'altra deciso dall'utente (se consentito[2]).

3. Progetto

Viene ora descritta la struttura dell'applicazione realizzata, illustrandone prima l'architettura software per illustrare più dettagliatamente i blocchi funzionali che la compongono.

3.1 Architettura del sistema software

Per la realizzazione di JBrick è stato scelto il pattern di programmazione *Logic–View.* Il *Logic* si occupa principalmente del controllo delle collisioni, tuttavia ha anche il compito di verificare l'accadimento degli eventi, ad esempio la vittoria di un livello, che possono verificarsi durante il gioco; inoltre carica ed esegue i suoni. Il *View* si occupa dell'aggiornamento grafico della schermata basandosi sia sui parametri che gli vengono forniti dal *Logic,* ad esempio il continuo aggiornamento della posizione della sfera, che sugli input utente (movimento della piattaforma, scorrimento fra le schermate). Al fine di ridurre l'accoppiamento tra classi di moduli distinti sono state definite le interfacce *ILogic* e *IView*.

Figura 4*: Direzione dei flussi informativi*

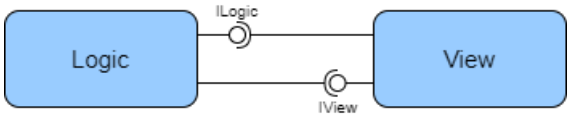


Figura 5*: interfacce esposte dai singoli moduli*

2 Lo scorrimento da una schermata alla successiva è scandito dall'utente nelle seguenti situazioni: avvio di una nuova partita, vittoria/perdita di un livello, perdita di una vita, pausa.

3.2 Logic

Le classi appartenenti al blocco *Logic* si trovano nel package *JBrick.logic*.

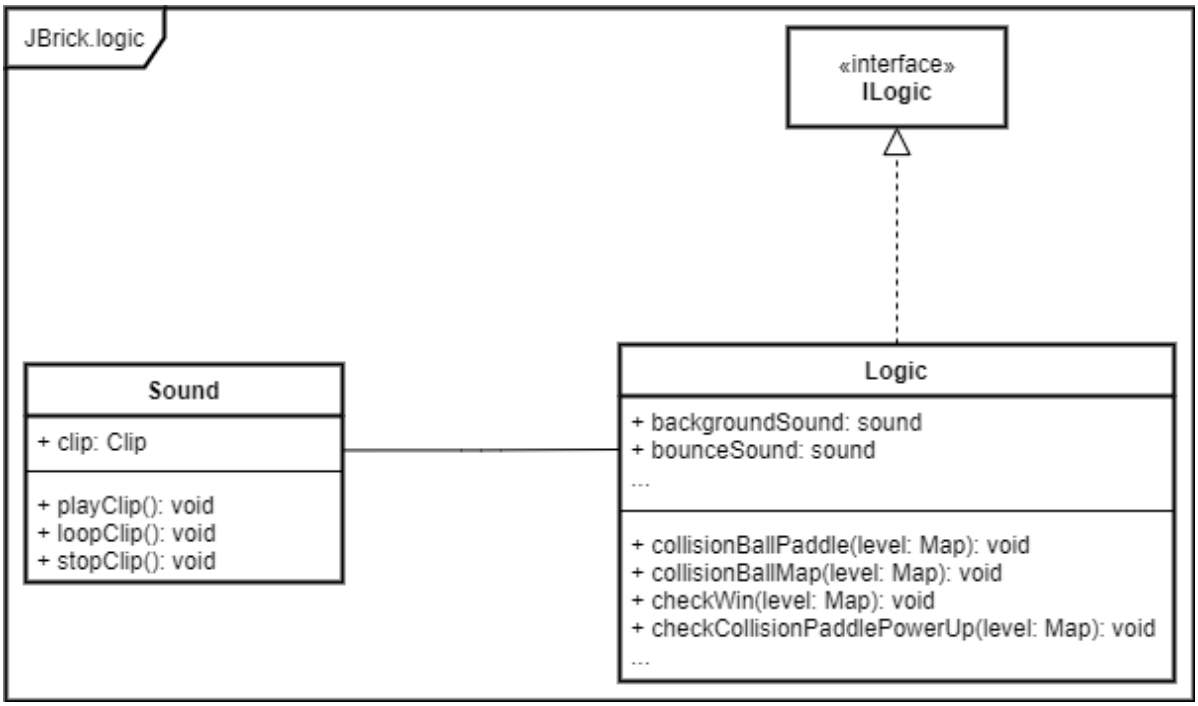


***Figura 6**: Diagramma UML delle classi di JBrick.logic*

In particolare:

- **Sound**: si occupa della gestione dei suoni. Consente di riprodurre (anche in loop) un file audio in formato .wav; inoltre permette anche di arrestare la riproduzione.

- **Logic:** La classe *Logic* che è la più importante gestisce tutta la parte logica dell'applicazione. I metodi che controllano le varie collisioni si basano tutte su una filosofia comune: viene creato un rettangolo in corrispondenza della sfera (che per comodità chiameremo *ballRect*) e un rettangolo (*elementRect*) per ogni elemento che è in grado di collidere con quest'ultima. Quando si verifica un'intersezione fra *ballRect* e *elementRect* viene rilevata una collisione che farà cambiare la direzione della sfera; se la collisione avviene alla sinistra o alla destra di *elementRect* sarà la componente x della direzione ad essere invertita, mentre invece se la collisione avviene superiormente o inferiormente rispetto a *elementRect* sarà la componente y della direzione ad essere invertita. La collisione fra *paddle* e *powerUp* si basa sempre sul principio di intersezione fra rettangoli. Un ulteriore gruppo di metodi serve a verificare gli eventi di vittoria/perdita di un livello e perdita di una vita. La riproduzione dei suoni viene sempre affidata a *Logic.*

È importante specificare che nel package *JBrick.logic* è presente la classe *Main* per avviare l'applicazione.

3.3 View

Le classi appartenenti al blocco *View* si trovano nel package *JBrick.view*.

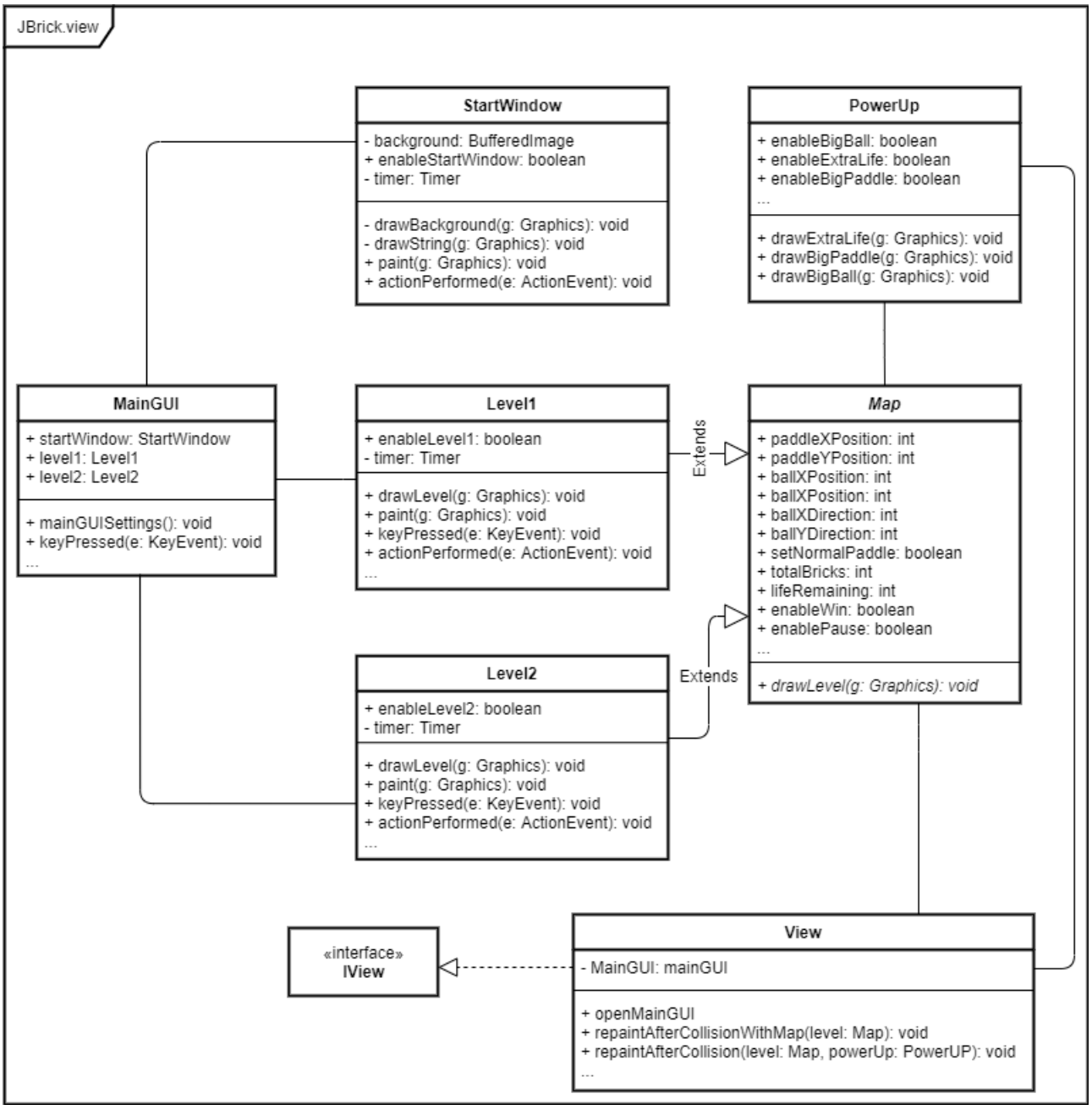


***Figura 7**: Diagramma UML delle classi JBrick.view*

In particolare:

- **MainGUI**: estende *JFrame* e si occupa di gestire lo scorrimento dei pannelli in base alla schermata raccogliendo l'input utente. Stabilisce le dimensioni e la posizione della finestra di gioco.
- **StartWindow**: estende *JPanel* ed è il pannello della schermata iniziale.
- **Map**: estende *JPanel*, è la classe astratta che definisce le variabili comuni a tutti i livelli.

- **Level**: si occupa dell'estetica dei livelli modificando a seconda del livello i vari elementi. Ogni livello implementa il metodo `drawLevel` in maniera differente così da ottenere una diversa disposizione dei mattoni. Riceve gli input utente per spostare il *paddle*, mettere in pausa il gioco e passare alla schermata successiva.

- **PowerUp**: si occupa della creazione dei potenziamenti.

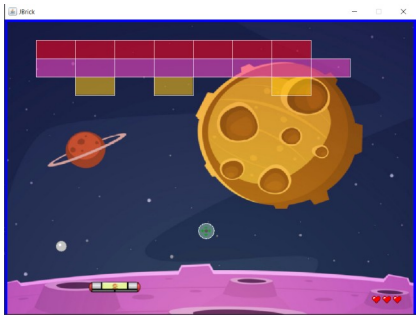


***Figura 8**: bigBall*

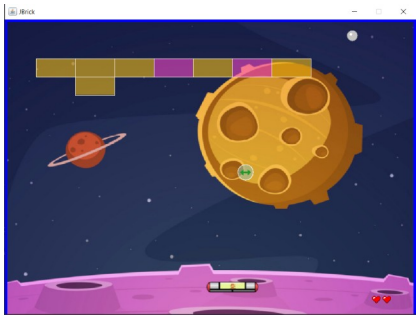


***Figura 9**: bigPaddle*

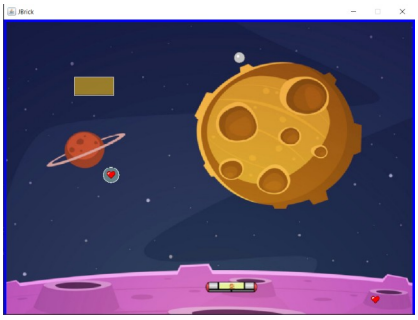


***Figura 10**: extraLife*

- **View**: La classe *View* si occupa di gestire continuamente il *repaint* dell'immagine. Grazie ai parametri passati dal *logic* riesce a rappresentare tutte le componenti grafiche che subiscono una qualsivoglia modifica durante una partita (movimento della sfera, movimento della piattaforma, rottura di un mattoncino, collisione con un *PowerUp*). Il metodo `repaintAfterCollision` si occupa anche di disegnare le schermate di vittoria/sconfitta, perdita di una vita.

3.4 Problemi riscontrati

I problemi riscontrati nello svolgimento del progetto sono i seguenti:

- Passaggio da una schermata all'altra.
- Individuazione delle collisioni.

La problematica del passaggio fra le varie schermate ha riguardato il dover analizzare le situazioni in cui l'utente può effettivamente passare alla schermata successiva; ad esempio non deve essere concesso all'utente di avanzare al livello successivo se non è stato completato quello attuale. Sono state utilizzate le variabili booleane per verificare la vittoria/sconfitta e per l'abilitazione delle varie schermate, così da determinare quando l'input utente può far procedere alla schermata successiva. Ogni volta che si ha un cambio fra schermate il pannello attuale viene rimosso e viene aggiunto il pannello successivo, facendo attenzione a modificare le variabili di controllo.

```
// from startWindow to level1
if(e.getKeyCode() == KeyEvent.VK_SPACE && startWindow.enableStartWindow){

    level1 = new Level1();
    addKeyListener(level1);
    level2 = new Level2();
    this.remove(startWindow);
    this.add(level1);
    startWindow.enableStartWindow = false;
    level1.enableLevel1 = true;
    this.repaint();
    this.revalidate();
}
```

***Figura 11**: Porzione di codice che consente il passaggio da startWindow a level1*

Per quanto riguarda l'individuazione delle collisioni si è deciso di adottare la strategia più semplice[3] poiché era la prima volta in cui ho dovuto affrontare una situazione del genere, inoltre il numero di collisioni da rilevare è molto elevato. La ***Figura 12*** mostra la gestione delle collisioni fra il *paddle* e la sfera, è interessante notare che il punto della piattaforma in cui avviene la collisione determinerà una diversa traiettoria assunta dalla pallina.

```
Rectangle normalPaddleRect = createNormalPaddleRectForCollision(level.paddleXPosition, level.paddleYPosition);
Rectangle ballRect = createBallRectForCollision(level.ballXPosition, level.ballYPosition, level.normalBallDimension);

if (ballRect.intersects(normalPaddleRect) && normalPaddleRect.y > ballRect.y) {

    if (ballRect.x <= normalPaddleRect.x + 30) {

        if (level.ballXDirection > -1)
            level.ballXDirection--;
        level.ballYDirection = -level.ballYDirection;

    } else if (ballRect.x > normalPaddleRect.x + 30 && ballRect.x <= normalPaddleRect.x + 70) {
        level.ballYDirection = -level.ballYDirection;

    } else if (ballRect.x > normalPaddleRect.x + 70) {
        if (level.ballXDirection < 1)
            level.ballXDirection++;
        level.ballYDirection = -level.ballYDirection;
    }

    bounceSound.playClip();


} else if (ballRect.intersects(normalPaddleRect))
    level.ballXDirection = -level.ballXDirection;
```

***Figura 12**: Porzione di codice che gestisce la collisione fra il paddle e la sfera*

4. Conclusioni e sviluppi futuri

Il progetto JBrick rappresenta un punto di partenza, poiché, pur essendo una versione essenziale ha la possibilità di aggiungere ulteriori funzionalità. Partendo dal progetto iniziale è possibile incrementare il numero dei livelli basandosi su una struttura di base comune; la gestione di ulteriori livelli non aumenterebbe la complessità nella gestione del codice, la logica dello scorrimento dei pannelli rimarrebbe pressoché invariata. Un ulteriore sviluppo potrebbe prevedere l'aggiunta di un punteggio assegnato con l'avanzare nel gioco e quindi anche l'implementazione di una classifica che tiene traccia dei migliori risultati ottenuti. La classe *powerUp* permette di aggiungere ulteriori potenziamenti, come ad esempio una piattaforma in grado di sparare e distruggere i mattoni. Si potrebbe pensare anche di aggiungere degli elementi che incrementino la difficoltà del gioco: una sfera con velocità crescente oppure dei mattoni con una resistenza maggiore ai colpi.

3 Vedi paragrafo 3.2 Logic

5. Bibliografia e sitografia

- Programmazione di interfacce grafiche e dispositivi mobili. *Unistudium*. https://www.unistudium.unipg.it/unistudium/course/view.php?id=13807

- Java™ Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/

- Breakout. *Wikipedia, L'enciclopedia libera.* https://it.wikipedia.org/wiki/Breakout_(videogioco)

- DrawIO. https://www.draw.io/