SIMON'S TRASH

Progetto d'esame

Laboratorio di Game Programming

Corso di Laurea Scienze e Tecnologie Multimediali

Apicella Andrea 145827 apicella.andrea@spes.uniud.it
Carrabino Francesco 145824 carrabino.francesco@spes.uniud.it
Passamonti Daniele 138382 passamonti.daniele@spes.uniud.it

Indice

1. Introduzione

- *a. Target*
- *b. Come si gioca*
- *c. Progressione*

2. Struttura del codice

- *a. Scene*
- *b. Composer*

3. game.lua

- *a. startGame( )*
- *b. visualizeObj( )*
- *c. timerInput( )*
- *d. userInput( )*
- *e. winCycle( )*
- *f. lostCycle( )*

4. tutorial.lua

- *a. Transizioni e timer*
- *b. userInput( )*

5. Garbage Collection

- *a. removeScene( )*
- *b. removeTimers( )*
- *c. removeTransitions( )*
- *d. audio.dispose( )*

6. Conclusioni

Introduzione

Il gioco si basa sul famoso videogioco "Simon Says", il cui funzionamento prevede che il giocatore ripeta in ordine una combinazione di colori apparsi a schermo.
Visto il tema "Ecologia e Ambiente", abbiamo rivisitato il meccanismo di base, trasformando la sequenza di colori in una sequenza di rifiuti.

- ***Target***
  Dallo stile grafico e dal meccanismo di gioco, si intuisce che il target sia un pubblico di giocatori da giovanissimi a molto giovani. La nostra intenzione è quindi quella di educare all'immediata individuazione della categoria di smaltimento per i rifiuti con cui si ha a che fare nella quotidianità.

La nostra mascotte è Simon, una simpatica ape. La scelta di questo animale è dovuta al fatto che le api sono tra le specie più a rischio di estinzione e al contempo tra le più importanti per l'equilibrio dell'ecosistema.

- ***Come si gioca***

All'inizio del gioco, la sequenza sarà composta di quattro rifiuti scelti casualmente tra quattro disponibili, che sono: un aeroplanino di carta, un flacone di plastica, una bottiglia di vetro e una mela morsicata.

L'utente, una volta apparsa tale sequenza, dovrà, nei dieci secondi a disposizione, ripetere la sequenza premendo i secchioni corrispondenti.

Ad ogni serie corretta, il gioco avanzerà di un livello facendo sì che un nuovo elemento venga aggiunto alla sequenza precedente. L'utente dovrà ripetere la combinazione nella sua interezza.

Allo scadere del timer o nel caso di un input sbagliato, si passerà alla schermata di Game Over.

L'obiettivo del gioco è superare ogni volta il proprio record.

- ***Progressione***
  Per introdurre degli elementi di varietà, ogni tre livelli verrà inserito, tra quelli disponibili, un nuovo rifiuto: al livello tre, verrà aggiunta una scatola di cartone; al livello sei, uno spruzzino di plastica; al livello nove, un barattolo di vetro; al livello dodici, un cheeseburger morsicato.

La visualizzazione della sequenza di rifiuti avverrà sempre più velocemente all'aumentare dei livelli. La difficoltà sarà proporzionale al numero di rifiuti apparsi e al tempo a disposizione dell'utente per memorizzarli.

Struttura del codice

Il progetto è composto da otto scene:

- *globals.lua:* contiene una table che include i file audio che non sono specifici di alcuna scena, ma di uso generico.
- *main.lua:* richiede l'uso del composer e accede alla scena intro.lua
- *intro.lua:* prima schermata visualizzata contenente i loghi dell'Università e del corso.
- *menu.lua:* contiene il titolo e i collegamenti alle scene info.lua, game.lua e tutorial.lua. Permette inoltre di silenziare o abilitare l'audio di gioco.
- *info.lua:* mostra al giocatore informazioni generali sul gioco, tra cui: quattro sprite che indicano dove gettare ciascun rifiuto, i rifiuti sbloccabili col progredire dei livelli e i credits relativi agli sviluppatori.
- *tutorial.lua:* contiene un tutorial interattivo a cui si accede al primo avvio del gioco (o finchè non si supera il primo livello) o tramite l'apposita icona nel menu.
- *game.lua:* è la scena di gioco in cui, dopo un countdown iniziale, viene visualizzata la sequenza e si abilita l'input dell'utente.
- *gameover.lua:* scena a cui si accede qualora l'input dell'utente sia errato. Consente di tornare al menu o di rigiocare.

Tramite il composer, passiamo da una scena all'altra durante l'esecuzione a seconda delle scelte dell'utente e dell'andamento del gioco.
Ogni volta che avviene un cambio di scena, tramite la funzione composer.removeScene( ) accediamo al metodo scene:destroy( ) del composer, in cui effettuiamo la garbage collection (capitolo 5).

game.lua

La scena game.lua è il vero e proprio core del gioco. Per come è strutturato il Runtime di Corona, ci siamo avvalsi di una concatenazione di timer e transizioni per scandire la progressione temporale del gioco. Nel nostro caso, infatti, non è presente una suddivisione in livelli: il gioco è potenzialmente un ciclo infinito che si interrompe solo nel momento in cui l'utente commette un errore (input sbagliato o tempo scaduto).

- **startGame( )**

```
local function startGame()
    livelloText.isVisible = true
    for i = 0, 3, 1 do
        local randomObj = math.random(table.getn(oggettiSbloccati))
        arraySimon[i] = oggettiSbloccati[randomObj]
    end
    timerStash[#timerStash + 1] = timer.performWithDelay(1000, startAn
imation, count + 1)
end
```

La funzione startGame( ) inizializza l'array contenente la sequenza di rifiuti inziale. I rifiuti vengono selezionati randomicamente all'interno dell'array oggettiSbloccati.

Tale array contiene inizialmente solo i quattro rifiuti standard e viene ampliato al livello 3, 6, 9 e 12 con quelli sbloccabili.

Riempito arraySimon, contenente la sequenza da memorizzare, fa partire un timer, tramite la funzione performWithDelay(delay, callback, numVolte) che esegue la funzione di callback per numVolte dopo il delay specificato.

Nel nostro caso, la callback eseguita è startAnimation( ) – il countdown "3…2…1…VIA!" – al termine della quale, tramite un altro timer.performWithDelay( ), si passa alla visualizzazione degli oggetti contenuti in arraySimon.

```
 timerStash[#timerStash + 1] = timer.performWithDelay(timeOS, visualizeO
bj, table.getn(arraySimon) + 1)
```

Per rendere possibile l'interruzione della scena game in qualsiasi momento (tramite il tasto menu in alto a destra), abbiamo deciso di inserire ciascun nuovo timer nella table "timerStash". La scelta di questa convezione si è rivelata di grande utilità anche per rendere più agevole la garbage collection.

- **visualizeObj( )**

```
local function visualizeObj()
    via.isVisible = false
    objOnScreen = arraySimon[lengthVis]

    transitionStash[#transitionStash + 1] =
        transition.to(
        objOnScreen,
        {
            time = timeOS / 2,
            alpha = 1,
            onComplete = disappear,
            onStart = audio.play(objectRevealSound)
        }
    )
    newSetText.isVisible = false

    lengthVis = lengthVis + 1

    if lengthVis == table.getn(arraySimon) + 1 then
        timerStash[#timerStash + 1] = timer.performWithDelay(timeOS, s
equenzaCompletata, 1)
    end
end
```

La visualizzazione degli oggetti avviene, come detto, grazie ad un performWithDelay il cui delay è timeOS.

```
if timeOS > 400 then
      timeOS = 1000 - (livello * 20)
  else
      timeOS = 400
  end
```

timeOS, inizialmente impostato a 1000 ms, viene aggiornato nella funzione winCycle( ) ogni volta che si passa al livello successivo.

L'aggiornamento di questa variabile rende la visualizzazione dei rifiuti ogni volta più rapida, contribuendo ad incrementare la difficoltà generale del gioco.

La visualizzazione degli oggetti avviene tramite la concatenazione di due transizioni, attraverso l'onComplete di transition.to, che richiama la funzione disappear( ): nella prima transizione l'alpha dell'oggetto da visualizzare viene portato a 1 in un tempo

che equivale a metà timeOS e nella seconda transizione, contenuta in disappear( ), viene riportato a 0 nello stesso tempo.

```
local function disappear()
    transitionStash[#transitionStash + 1] = transition.to(objOnScreen, {
time = timeOS / 2, alpha = 0})
end
```

L'oggetto da visualizzare viene selezionato tramite la variabile lengthVis, che incrementa ad ogni ripetizione di visualizeObj, ovvero tante volte quanto il numero di rifiuti all'interno di arraySimon.

Quando tutti gli oggetti sono stati visualizzati, attraverso un altro performWithDelay si richiama la funzione sequenzaCompletata, che aggiunge gli EventListener ai secchioni della spazzatura e fa partire i 10 secondi a disposizione del giocatore.

Anche per quanto riguarda le transition.to, ci siamo avvalsi di una table transitionStash che contiene tutte le transizioni e che, allo stesso modo di timerStash, rende più agevole la garbage collection.

- **timerInput( )**

```
local function timerInput()
  timerText.isVisible = true
  if timerLeft == 0 then
    removeUserInteraction()
    audio.play(wrongSound)
    transitionStash[#transitionStash + 1] = transition.to(gameOver, {time = 1400, alpha = 1,
onComplete = lostCycle})
    timerText.isVisible = false
  end

  timerText.text = timerLeft
  timerLeft = timerLeft - 1
end
```

La funzione timerInput( ) stampa a schermo il tempo a disposizione del giocatore e nel momento in cui questo termina, elimina gli EventListener dei secchioni, mostra la scritta Game Over e all'onComplete di tale transizione richiama la funzione lostCycle( ).

- **userInput( )**

```
function userInput(event)
   local rifiuto = event.target
   rifiuto:play()
   if rifiuto.name == arraySimon[countInputs].name then
      audio.play(correctSound)
      table.insert(arrayUtente, countInputs, arraySimon[countInputs])
      if countInputs == table.getn(arraySimon) then
         removeUserInteraction()
         removeTimers()
         timerText.isVisible = false
         transitionStash[#transitionStash + 1] = transition.to(nice, {time = 1000, alpha = 1,
onComplete = winCycle})
         audio.play(niceSound)
      end
      countInputs = countInputs + 1
   else
      removeUserInteraction()
      audio.play(wrongSound)
      removeTimers()
      timerText.isVisible = false
      transitionStash[#transitionStash + 1] =
         transition.to(gameOver, {time = 1400, alpha = 1, onComplete = lostCycle})
   end
end
```

La funzione userInput(event) viene richiamata ogni volta che il giocatore tocca uno dei secchioni.

Controlla inizialmente qual è il target dell'event, individuando quale dei secchioni sia stato premuto. Successivamente controlla che il nome del rifiuto coincida con il nome del secchione toccato – abbiamo usato un sistema di attributi ".name" che ci permette di confrontare le tipologie di rifiuti con i secchi.

Nel caso in cui questi coincidano, il rifiuto viene inserito nell'arrayUtente e, quando tale array raggiunge la stessa lunghezza dell'arraySimon, si avvia la transition.to che all'onComplete richiama la funzione winCycle( ).

Nel caso in cui, invece, essi non coincidano, si avvia la transition.to che all'onComplete richiama la funzione lostCycle( ).

- **winCycle( )**

```
local function winCycle(event)
```

```
    removeUserInteraction()
    transitionStash[#transitionStash + 1] = transition.to(event, {time = 50, alpha = 0})
    livello = livello + 1

    if livello == 3 then
        table.insert(oggettiSbloccati, oggettiTotali[5])
        newSetText.isVisible = true
    elseif livello == 6 then
        table.insert(oggettiSbloccati, oggettiTotali[6])
        newSetText.isVisible = true
    elseif livello == 9 then
        table.insert(oggettiSbloccati, oggettiTotali[7])
        newSetText.isVisible = true
    elseif livello == 12 then
        table.insert(oggettiSbloccati, oggettiTotali[8])
        newSetText.isVisible = true
    end

    if livello > highScore then
        highScore = livello
        updateHighScore()
    end

    livelloText.text = "Livello: " .. livello
    arrayUtente = {}
    if timeOS > 400 then
        timeOS = 1000 - (livello * 20)
    else
        timeOS = 400
    end
    lengthVis = 0
    countInputs = 0
    timerLeft = 10
    table.insert(arraySimon, oggettiSbloccati[math.random(table.getn(oggettiSbloccati))])

    timerStash[#timerStash + 1] = timer.performWithDelay(timeOS, visualizeObj,
table.getn(arraySimon) + 1)
end
```

La funzione winCycle( ) viene richiamata quando il livello è stato completato, quindi ha il compito di reimpostare le variabili di gioco per consentire la progressione al livello

successivo. Controlla inoltre il livello attuale per sbloccare i nuovi oggetti ed inserirli nell'array oggettiSbloccati.

Aggiorna anche l'highScore se il giocatore ha superato il suo precedente record e inserisce un altro elemento randomico nell'arraySimon.

Infine, fa ripartire la visualizzazione degli oggetti, e quindi il ciclo di gioco, tramite un timer.performWithDelay che richiama visualizeObj( ).

- **lostCycle( )**

```
local function lostCycle()
   transitionStash[#transitionStash + 1] = transition.to(gameOver, {time = 300, alpha = 0})
   timerText.isVisible = false
   composer.removeScene("game")
   composer.gotoScene("gameover", {effect = "fade"})
end
```

La funzione lostCycle( ) sfrutta il composer per rimuovere la scena game.lua e passare alla scena gameover.lua.

tutorial.lua

- **Transizioni e Timer**

Volendo realizzare un tutorial interattivo, ci siamo avvalsi di alcuni dialoghi che sfruttano l'evento "tap" e una serie di transizioni e timer per guidare il giocatore passo passo.

In particolare, il giocatore, toccando su ogni dialogo passerà alla seguente fase del tutorial: innanzitutto gli verrà mostrato il countdown, presente all'inizio di ogni partita; dopo di che, lo avviserà di memorizzare la sequenza di rifiuti che gli sarà mostrata; infine farà partire il timer di dieci secondi entro cui il giocatore dovrà ripetere la sequenza.

Transition.to viene quindi usata per far apparire i box di dialogo e per rimuoverli dopo il tap.

I timer, come nella scena game, dettano il ritmo dell'esecuzione, stavolta subordinata al tocco del giocatore.

- **userInput( )**

```
function userInput(event)
   local rifiuto = event.target
   rifiuto:play()

   if rifiuto.name == arraySimon[countInputs].name then
      audio.play(correctSound)
      table.insert(arrayUtente, countInputs, arraySimon[countInputs])
      if countInputs == table.getn(arraySimon) then
         removeUserInteraction()
         removeTimers()
         timerText.isVisible = false
         audio.play(dialogoSound)
         transitionStash[#transitionStash + 1] = transition.to(ape, {time = 350, x = 5, transition =
easing.outCirc})
         transitionStash[#transitionStash + 1] = transition.to(dialogo4, {time = 400, alpha = 1})
         dialogo4:addEventListener("tap", fineQuartoDialogo)
      end
      countInputs = countInputs + 1
   else
      audio.play(wrongSound)
      removeUserInteraction()
      timerText.isVisible = false
      removeTimers()
      timerLeft = 10
```

```
    countInputs = 0
    arrayUtente = {}
    sequenzaCompletata()
  end
end
```

La funzione userInput( ) riprende la sua omonima nella scena game.lua, ma con la differenza che non si passa alla scena gameover.lua nel caso in cui il giocatore sbagli, bensì si ritorna al dialogo precedente, dando la possibilità di terminare correttamente il tutorial.

Quando l'utente riproduce correttamente la sequenza, verrà mostrato l'ultimo dialogo per poi passare alla scena game.lua ed avviare la vera e propria partita.

Nel caso in cui l'utente non riesca a terminare il tutorial può tornare in qualsiasi momento al menu con l'apposita icona (in alto a destra).

Garbage Collection

- **removeScene( )**

In ognuna delle scene da noi create, abbiamo utilizzato il composer di Corona, la cui suddivisione avviene attraverso fasi distinte.

In particolare, posizioniamo le grafiche nella fase scene:create( ), che è eseguita quando la scena viene creata ma non è ancora apparsa a schermo, e inizializziamo tutte le variabili locali ad una data scena. Nella fase scene:show( ), quando la scena è ancora fuori dallo schermo, posizioniamo tutti gli elementi grafici, mentre quando la scena è totalmente a schermo, richiamiamo la prima funzione necessaria al progredire del gioco ed inseriamo gli EventListener.

Tutti gli elementi grafici sono inseriti in dei GroupObject in modo da semplificare la loro rimozione.

Tale rimozione avviene nella fase scene:destroy( ), che viene richiamata dal metodo removeScene del composer.

- **removeTimers( )**

```
local function removeTimers()
   for i = 1, #timerStash do
      if timerStash[i] ~= nil then
         timer.cancel(timerStash[i])
      end
   end
end
```

Come già detto, ci siamo avvalsi di una table per contenere tutti i timer utilizzati durante la progressione del gioco. Questo meccanismo ci ha consentito di interrompere l'esecuzione in qualsiasi momento grazie alla funzione removeTimers( ). Ciò che essa fa è scorrere tutta la table, controllare che il timer all'indice i-esimo non sia nullo e utilizzare la funzione timer.cancel per rimuoverlo.

- **removeTransitions( )**

```
local function removeTransitions()
   for i = 1, #transitionStash do
      if transitionStash[i] ~= nil then
         transition.cancel(transitionStash[i])
      end
   end
end
```

La funzione removeTransitions( ) opera in modo del tutto analogo a removeTimers( ): controlla ogni transizione presente nell'apposita table e la rimuove nel caso questa non sia già nulla.

- **audio.dispose( )**

Per rimuovere i suoni locali di ogni scena, abbiamo usato il metodo audio.dispose( ). I suoni condivisi da più scene invece, come la soundTrack, sono solamente interrotti con audio.stop(numeroCanale) e ripristinati quando necessario.

Conclusioni

Il concept del gioco non è troppo complesso. Questa scelta è dovuta alla volontà di realizzare, tenendo presente che si tratta di un primo approccio a Corona SDK, un gioco che sia il più fluido, efficiente e credibile possibile.
Nonostante ciò, la resa in codice di questo concept ha rivelato delle insidie, soprattutto nel momento in cui abbiamo voluto rendere flessibile l'esecuzione, che, come detto, può essere interrotta in qualsiasi momento.
Una efficace gestione della memoria, sebbene il gioco in sé per sé non sia tassativo in termini di performance, ha richiesto una comprensione approfondita del funzionamento dei timer e delle transizioni, nonché del composer e di come una scena viene rimossa.
Tutti gli asset grafici sono realizzati da noi, mentre i file audio sono stati presi da fonti free copyright.
Il gioco si presta a futuri upgrade, come l'aggiunta di powerUp, di nuovi rifiuti e di variazioni del playstyle di base.
Allo stato attuale, il gioco ha una difficoltà piuttosto elevata, raggiunto un certo livello, ma l'inserimento di una "easy mode" o di una "hard mode" è facilmente implementabile, variando il tempo a disposizione dell'utente.