DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Laberintos musicales con OSU!

Trabajo práctico final

Computación Gráfica
Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Macarena Piaggio	212/18	piaggiomacarena@gmail.com
Tomás Agustín Chimenti	99/18	tach.365@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

Todo list

1. Introducción

El presente trabajo permite generar ”salvapantallas.[en] el estilo del salvapantallas de laberinto 3d incluído en Windows 95, pero donde el movimiento a través del laberinto se coordina con la música seleccionada utilizando un mapa del juego de ritmo OSU!. Se puede ver una demo de la aplicación en https://makobot-sh.github.io/music_labyrinths/, pero si se desea agregar nuevas canciones o texture packs se debe descargar la aplicación y correrla localmente. El modo de uso de la aplicación es el siguiente:

1. Descargar de https://osu.ppy.sh/beatmapsets el archivo `.osz` del mapa que se desee colocar en la aplicación.
2. Correr el progrmaa `osz_parser.py` del siguiente modo: `python3 osu_parser.py path/to/osz` y seleccionar las opciones que correspondan cuando dicho programa lo requiera.
3. Levantar un servidor local, recomendamos el que viene con python3: `python3 -m http.server`. Incluímos archivos `.sh` y `.cmd` para agilizar este paso.
4. Abrir la aplicación (se encontrará en localhost en el puerto seleccionado, por defecto en http://localhost:8080/). En la selección de mapas debería aparecer el mapa nuevamente añadido!

Para ver instrucciones sobre como añadir un nuevo texture pack, ver la sección 6

Nota sobre soporte: el soporte de parseo de mapas en Windows es experimental, especialmente la creación de laberintos aleatorios puede fallar. Si se desea agregar un mapa, recomendamos usar Linux o. Si quiere usar Windows y al correr `osz_parser.py` falla la creación del laberinto, correr el ejecutable de c `mazegen` por cuenta propia (generará un archivo .svg) y luego correr `python3 svg_parser.py path/to/svg` a mano. Alternativamente, se puede copiar alguno de los `maze.json` de los mapas de demo incluidos a la carpeta que se genera para el mapa parseado.

2. Generación de laberintos

Para la generación de los laberintos partimos del generador de laberintos aleatorios en formato `.svg` de `guilbep`[1]. Cada vez que se agrega un mapa de OSU! a la aplicación (sección 3) se genera un mapa de dimensiones 30x30 (30 ”paredes”de alto y ancho). La ventaja del formato de imagen `.svg` es que, al ser un formato de gráficos vectoriales, su información nos es sencilla de leer y parsear para generar una estructura propia que represente dicho laberinto. Tomamos entonces este svg y, utilizando nuestro programa `svg_parser.py`, generamos un `.json` filtrando únicamente la información que nos interesa, que es las posiciones de las paredes y la dimensión total del mapa (que incluye 30 unidades de borde blanco en todas las direcciones). El formato .json tiene la ventaja de ser muy fácilmente utilizable desde aplicaiones de javascript, motivo por el cual lo utilizamos a lo largo del trabajo.

Una vez generado `data_lines.json`, querremos utilizar este archivo dentro de nuestra aplicación para generar el laberinto 3d. Generar las paredes 3d es sencillo, dado que ya hicimos el parseo previo del .svg. Tomamos el mismo tamaño de pared que utiliza `mazegenerator` para el .svg: por cada pared, colocamos un plano de tamaño 30x30.

Para poder generar los movimientos a través del mapa más adelante, necesitaremos una estructura que, para cada posición en la que pueda uno estar parado, indique cuales son las paredes inmediatamente adyacentes. Decidimos pensar en el laberinto como una matriz, donde cada espacio representa un punto del laberinto que puede estar rodeado de cuatro paredes (en la figura 1 se puede ver como un punto rojo cada una de las posiciones del laberinto). Los movimientos de la cámara se realizarán siempre de

alguno de estos puntos a otro adyacente que no esté separado por una pared (de la duración de dichos movimientos hablaremos en la sección 4.

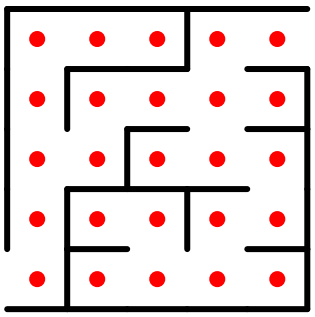

Figura 1: Laberinto: los puntos rojos indican los puntos en los que frenaremos a decidir cual es el siguiente movimiento.

En cada posición de la matriz que representa al laberinto guardaremos un entero de 4 bits, con el formato `LEFT-RIGHT-DOWN-UP`. Es decir, el bit menos significativo prendido (0b0001, o 1 en decimal) representará que hay una pared arriba del punto rojo; el bit más significativo prendido (0b1000, 8 en decimal) representará que hay una pared a la izquierda del punto rojo, y así. El completado de esta matriz lo hacemos a medida que procesamos el .json y generamos el laberinto 3d: cada vez que colocamos una pared, prendemos los bits correspondientes en las posiciones adyacentes a dicha pared.

3. Parseo de archivos `.osz`

OSU![2] es un juego de ritmo gratuito publicado en 2007[3] para Windows (y luego adaptado a macOS, Linux, Android y iOS), inspirado en otros juegos del género como *Taiko no Tatsujin* y *Elite Beat Agents*. El juego tiene un fuerte enfoque en la comunidad: todos los mapas de ritmos fueron creados por la comunidad en el editor del juego. En la wiki del juego[4] se puede encontrar documentación completa para el formato de archivo de los mapas de juego[1] (`.osu`) y para otros tipos de archivos, como *skins* para el juego.

Por estos motivos es que decidimos soportar beatmaps de OSU! (archivos `.osz`, específicamente) como entrada para nuestro programa, permitiendo generar laberintos musicales para cualquier canción que tenga un beatmap en el juego (específicamente mapas para OSU! clásico, no OSU!mania).

3.1. Formato `.osz`

En `https://osu.ppy.sh/beatmapsets` se encuentran disponibles para descargar todos los beatmaps creados por la comunidad del juego, en formato `.osz`. Este formato es un renombre de `.zip`, por lo que es sencillo descomprimirlos desde la aplicación de python que acompaña a la aplicación WebGL en sí.

Los ficheros contienen:

- Tantos archivos `.osu` como versiones del mapa se incluyan (en general, distintas versiones representan distintas dificultades del mapa).
- Uno o más audios, que se corresponderán cada uno a al menos uno de los mapas `.osu` incluídos. Actualmente sólo se soportan mapas con audio en formato `.mp3`.

[1] `https://osu.ppy.sh/wiki/en/osu%21_File_Formats`

- Otros archivos, como imagenes para el fondo del mapa dentro del juego y sonidos extra que se puedan requerir para ciertas acciones dentro del juego.

Nosotros utilizamos los archivos `.osu` para obtener la información rítmica de la música del mapa, y al cargar la aplicación utilizamos el archivo de audio incluido como música.

3.2. Parseo del formato .osu

Los archivos `.osu` contienen una gran cantidad de información sobre el mapa en cuestión. Aquí se mencionará sólo la información que utilizamos para el funcionamiento de la aplicación, pero una descripción completa puede encontrarse en la wiki de OSU![4].

Los archivos `.osu` están divididos en secciones, donde cada seccion comienza con una linea con formato `[Nombre de sección]`. Las secciones relevantes al trabajo son:

- General
  - De aquí se obtiene el nombre del archivo de audio que corresponde al `.osu` actual.
- Metadata
  - Entre otra información, contiene el título (romanizado) del mapa en cuestión, el artista, y el nombre de la versión.
- HitObjects
  - Esta sección contiene la información de los objetos que se clickean durante el transcurso del mapa del juego: su duración, momento en que aparecen, entre otros. La generación de movimientos para el laberinto se puede realizar a partir de esta información, cambiando `BPM movement` de true a false en las `Movement configs` de `config.json`. De esta sección se obtiene la información para crear el archivo `Título del mapa - versión_times.json`.
- TimingPoints
  - Sin entrar en tecnicismos del juego, esta sección nos da información de los cambios de tempo (bpm) de la canción/mapa. Por defecto, nuestra aplicación utiliza esta información para generar los movimientos del laberinto. De esta sección se obtiene la información para crear el archivo `Título del mapa - versión_bpms.json`.

3.3. Generación de archivos .json utilizados por la aplicación

La aplicación `osz_parser.py` toma como único parámetro un archivo `.osz` y a partir de él generará los siguientes archivos:

- Una carpeta con nombre `Título del mapa - versión` que contendrá:
- `Título del mapa - versión_audio.mp3` (u otro formato de audio, aunque actualmente .mp3 es el único soportado por la aplicación gráfica) que es una copia del archivo de audio contenido en el `.osz` correspondiente a la versión del mapa seleccionada.
- `Título del mapa - versión_bpms.json` que contiene una lista de pares `(start, beatlen)` donde start indica cuando comienza una sección con cierto tempo, y beatlen indica la duración de cada unidad de tiempo de la sección (*beat*) en milisegundos. Si beatlen es negativo, el valor absoluto del número se usa como multiplicador relativo al tempo de la sección anterior (actualmente se ignoran valores relativos en la aplicación).
- `Título del mapa - versión_times.mp3` que contiene una lista del momento en que debe ser apretado cada objeto del mapa de osu. Este archivo sólo se utiliza si `BPM movement` está deshabilitado en `config.json`.

3.4. Selección de mapa

Como mencionamos, la aplicación soporta archivos `.osz` como aquellos que se pueden descargar en https://osu.ppy.sh/beatmapsets. Se incluyen algunos mapas de prueba, pero si se deseara agregar uno nuevo se puede hacer corriendo el programa `osz_parser.py` con el *path* al archivo .osz del siguiente modo:

```
python3 osz_parser.py path_to_osz
```

Este programa nos dejará seleccionar cuál versión del mapa usar, y generará una carpeta `beatmaps/Título del mapa - versión` con toda la información que requerirá la aplicación para disponibilizar ese mapa. Además, la aplicación actualiza `maps_index.json`, haciendo que el mapa aparezca como opción en el menú de inicio de la aplicación.

4. Generación de movimientos y coordinación

Hay dos formas de generar el movimiento: a partir de los tempos del mapa y a partir de los HitObjects del mapa. En nuestras pruebas determinamos que la primera generaba movimientos más coherentes que la segunda, por lo que dejamos esta modalidad en el trabajo. Si se quiere, se puede cambiar a la segunda modalidad cambiando `BPM movement` de true a false en las `Movement configs` de `config.json`, pero notar que fue ”deprecada”(se mantiene como una opción viable únicamente como curiosidad).

Para generar los movimientos, partimos del archivo `Título del mapa - versión_bpms.json`. Recordemos que en este archivo tenemos el tempo de las distintas secciones de la canción, por lo tanto lo primero que querremos hacer es convertir esto a ”pulsos”, que serán los momentos de la canción en los que se podrá **comenzar** un movimiento. Como contamos con la longitud de un *beat* en milisegundos, basta con decir que está permitido comenzar un movimiento cada vez que haya pasado dicha cantidad de milisegundos (en verdad, dado que los mapas de OSU suelen ser muy rápidos, seleccionamos a través de pruebas una mínima longitud de *beat*. Si el medida actual es menor a esta, se multiplica por dos hasta que no lo sea).

Por lo tanto, para cada sección de tiempo agregamos al array `times` el momento en que comienza cada pulso de esa sección (medido en milisegundos desde 0, el comienzo de la canción). Cada movimiento a través del laberinto comenzará en alguno de estos tiempos seleccionados, y durará exactamente lo necesario para que el siguiente movimiento comience también en algún número posterior del arreglo.

En el contexto del proyecto, cuando hablamos de *un movimiento* podemos estar refiriendonos a:

- La acción de moverse hacia adelante üna vez”(avanzar hacia donde apunta la cámara cierta medida fija, que coincide con el largo de cada pared individual del laberinto).
- La acción de girar una o dos veces (girar la cámara 90 o 180 grados en una coordenada), y luego avanzar hacia adelante una vez.

Sabemos cuánto queremos que dure el movimiento total, por lo que en el primer caso basta con mover la cámara a una velocidad tal que pase de un punto A al punto B en ese tiempo. Pero en el segundo caso, debemos definir cuánto tiempo toma hacer una rotación. ¿Será proporcional a la duración total del movimiento? ¿Un número fijo? Nosotros consideramos que la rotación es uno de los movimientos más importantes del laberinto, ya que es el que marca el ritmo de la canción más fuertemente, por lo que decidimos que el tiempo que dura cada rotación fuera siempre múltiplo de la logitud del *beat* de la sección en curso. Para poder lograr esto, el arreglo `times` está acompañado por otro arreglo, `rotspeeds`, tal que para cualquier `t = times[i]`, `rotspeeds[i] = beatLength * multiplier`. El valor de `multiplier` es fijo y se puede cambiar en `config.js`.

Si el movimiento se genera a partir de los HitObjects, se reemplaza el arreglo `times` recién generado por el arreglo guardado en `Título del mapa - versión_times.json`, que indica cuando comienza cada

HitObject del mapa. Para la velocidad de rotación se mantiene fija la primer velocidad de rotación del arreglo `rotSpeeds` (para que la modalidad de movimiento por HitObjects utilice las rotation speeds correctas habría que adaptar el arreglo. Dado que "deprecamos.[es]ta modalidad de movimiento, decidimos no hacer la adaptación y utilizar esta solución rápida en su lugar). Una vez reemplazado el arreglo `times`, se procede de la misma forma para ambas modalidades de movimiento.

Ya decididos cuales son los momentos de la canción en la que podemos movernos, queda elegir en cuales vamos a querer efectivamente movernos, y qué tipo de movimiento seleccionar en cada uno de estos momentos.

En primer lugar, mantendremos en todo momento un *timer* que contendra el valor del último momento en que se inició un movimiento. Comenzará en el valor del primer elemento del arreglo `times`. Iteraremos sobre el arreglo `times`, y en cada iteración haremos lo siguiente:

1. Sea `t = times[i]`, definimos `movDuration = t - timer` la duración del último movimiento realizado, si comenzamos un nuevo movimiento en el i-ésimo tiempo.
2. Si `movDuration` es menor a la mínima longitud de movimiento seleccionada en `config.json`, no se iniciará un movimiento en `t`. Si este no es el caso, se continua:
3. Si `movDuration` es mayor o igual a `minMovLen + rotSpeed` (donde `rotSpeed` es la velocidad de rotación en la sección musical actual) y no hay paredes a la izquierda y/o derecha, se agregan la posibilidad de rotar a la izquierda y/o de rotar a la derecha al conjunto de movimientos posibles para el t actual.
4. Si no hay una pared en la dirección a la que mira la cámara, se agrega la posibilidad de avanzar al conjunto de movimientos posibles en el t actual.
5. Si la única dirección a la que se puede moverse es hacia atrás (se llegó al final de un camino sin salida) y hay suficiente tiempo para girar y moverse (`movDuration` es mayor o igual a `minMovLen + rotSpeed*2`), se agrega al conjunto de movimientos posibles para el t actual el movimiento para atrás (será el único movimiento en el conjunto). Además, se marca la posición en el laberinto para no volver a visitarla (se coloca una pared en la matriz que representa al laberinto) y se pone en verdadero la flag `isDeadEndPath`. Esta flag permanecerá prendida hasta que se llegue a una bifurcación en el camino, y por cada posición del laberinto que se recorre con esta flag prendida, se coloca una nueva pared detrás de donde apunta la cámara en la matriz que representa al laberinto (no se colocan paredes en el mapa 3d).
6. Entonces, si hay al menos un movimiento en el conjunto de movimientos posibles para el t actual, elegimos alguno al azar y lo agregamos a la lista de movimientos a realizar.
7. Además, actualizamos `timer = timer + movDuration` y actualizamos la dirección hacia la que apunta la cámara.

Una vez que terminamos de iterar para todos los elementos de `times`, habremos generado los movimientos para toda la canción. Este arreglo de movimientos se anima utilizando `Tween.js` (en la función `animateSeries`). Por la forma en la que construimos los movimientos, todos se realizarán de forma coordinada con la música!

5. Animación

Para todas las animaciones del proyecto (movimientos de la cámara) se utiliza la librería `tween.js`. Esta librería nos permite elegir el valor que queremos tenga la propiedad de un objeto (puede ser relativo al valor actual o absoluto) y elegir cuanto tiempo queremos que dure la transición, y se ocupará de hacer la interpolación del valor en el tiempo. De esta forma animamos los movimientos de la cámara, utilizando las duraciones comos e calcularon arriba y modificando las propiedades de posición (x, y, z) y rotación (x, y) de la cámara.

6. Apariencia

Incorporamos un sistema de *texture packs* que permiten cambiar la apariencia del laberinto durante la pantalla de inicio de la aplicación. Cada texture pack permite especificar distintas imágenes para cada una de las superficies del laberinto (suelo, techo, paredes, entradas) y cuanto se repiten horizontal y verticalmente a lo largo de dichas superficies las texturas. Agregar un nuevo texture pack es muy sencillo, se puede realizar copiando alguno de los .json existentes en la carpeta `config/resorce_packs` y modificando las imagenes a las que se apunta. El nombre del resource pack (sin .json) debe ser incluido en `texture_index.json` para que aparezca como opción en el menú de inicio de la aplicación.

7. Trabajo futuro

Algunas ideas que fueron descartadas por cuestiones de escala y tiempo pero que nos gustaría revisitar en el futuro son:

- Mejorar la UI de la aplicación.
- Incorporar las imagenes que vienen en el `.osz` (carátula de la canción y otras si las hay) como ”pósteres”que pueden aparecer aleatoriamente en algunas paredes del laberinto.
- Utilizar la paleta de colores del mapa (viene incluída en el .osu) para influenciar el color de la luz o de las texturas del mapa.
- Incluir alguna animación extra que marque las unidades de tiempo, como pasos (movimiento vertical de la cámara) o golpes de luz.
    - Hicimos varias pruebas en este respecto, pero ninguno de los efectos que logramos nos parecieron efectivos, y de hecho consideramos que mareaban más, por lo que vienen desactivados por defecto. El golpe de luz no llegó a la versión final del trabajo, pero el salto se puede activar colocando la opción ”Jump ON”de ”Movements configs.[en] `true` en `config.js`.
- Botón de pausa para la ejecución.

Referencias

[1] guilbep. Maze generator, 2016. https://github.com/guilbep/mazegenerator.

[2] Dean Herbert. Osu! https://osu.ppy.sh/home/download.

[3] Wikipedia la enciclopedia libre. Osu! https://es.wikipedia.org/wiki/Osu!

[4] Osu! community. Osu! wiki. https://osu.ppy.sh/wiki/en/osu.