Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

# Modelación y Computación Gráfica para Ingenieros

# Tarea 2

Super HexaGon

Integrantes: Gonzalo Uribe
Profesor: Nancy Hitschfeld K.
Auxiliares: Pablo Pizarro R.
Pablo Polanco
Ayudantes: Joaquín T. Paris
Roberto Ignacio Valdebenito
Rodrigo E. Ramos T.

Fecha de entrega: Domingo 14 de Mayo
Santiago, Chile

# Índice de Contenidos



# Lista de Figuras



# Lista de Tablas

# 1. Problema

Se debe programar un juego basado en *Super Hexagon*, donde se controla un puntero que se mueve al rededor de una figura central esquivando obstáculos al ritmo de la musica. El tamaño de los obstáculos depende de la cantidad de lados que tenga la figura central, además la cámara debe rotar al rededor de la figura central. El jugador gana el nivel al resistir una cantidad determinada de tiempo.

# 2. Solución

## 2.1. Modelo MVC y estructura de juego

Para el juego se uso la estructura MVC tratando de automatizar los procesos mas repetitivos. Para hacer esto se establecen las siguientes reglas:

- Cada etapa del juego corresponde a una escena (menú principal, pantalla de derrota, el juego en si, etc).
- En todo momento debe haber una escena activa.
- Todos los controladores deben heredar de la clase *GameObject* y ser agregados a la escena.
- Todos los objetos que se quieren dibujar deben heredar de la clase *Drawable* (que hereda de GameObject) y ser agregados a la escena.

De esta manera se puede automatizar el proceso de dibujado y es mas sencillo controlar las posiciones de los objetos, sus rotaciones y su escala.

La clase GameObject posee la función *update()* pensada para ser sobre escrita por las clases que hereden de GameObject y ejecutar en ella la lógica del objeto. La clase Drawable posee la función *draw on screen()* que dibuja usando pygame los vertices de la figura en la pantalla.

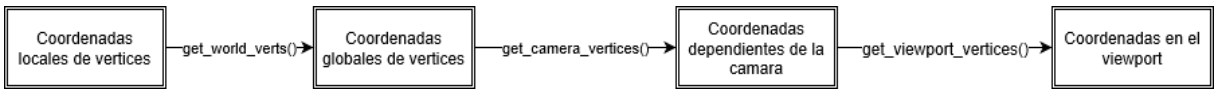


Figura 2.1: Pipeline de transformaciones realizadas

La secuencia de transformaciones realizadas para dibujar un objeto se describe en la figura 2.1, el resultado final considera las coordenadas x e y del objeto, su rotación y su escala, la posición, rotación y zoom de la cámara y el tamaño de la ventana de visualización.

El archivo *GameStructure.py* incluye la implementación de la clase *Scene*, *GameObject*, *Drawable*, *Camera* y otras variaciones de *Drawable* como *GUIFigure* y *GUIText* las cuales no toman en consideración la posición de la cámara.

## 2.2. Colisiones

Para un juego de este genero se necesita un sistema de detección de colisiones preciso y rápido, por se implementó un sistema propio de colisiones que aproveche las características únicas del juego.

Detectar colisiones es muy sencillo cuando se dispone solo de figuras ortogonales, por esto varios motores de juego incluyen funciones para detectar colisiones entre rectángulos, para poder utilizar los métodos de detección de colisiones ortogonales se utilizan coordenadas cilíndricas (este es uno de los motivos por los cuales las coordenadas de los objetos se escriben de forma cilíndrica). Para transformar entre coordenadas cilíndricas y cartesianas se utilizan funciones auxiliares de la clase *GameMath*.

Para detectar si un punto esta dentro de un obstáculo primero se revisa si el ángulo esta entre los ángulos de los bordes de este, luego se revisa si el componente radial del punto esta entre los componentes radiales del obstáculo. Cabe mencionar que a pesar de que el valor del componente radial cambie según el ángulo, se tomo en consideración este problema y se obtiene un valor calculado en función del ángulo.

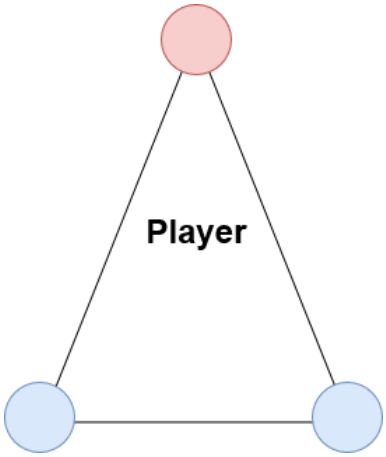


Figura 2.2: Puntos de colisión del jugador

De esta forma se puede obtener un resultado suficiente revisando las colisiones en la punta del puntero, pero se implementó un sistema que revisa tres puntos diferentes como lo indica la figura 2.2 para que si el jugador colisiona en uno de los puntos azules choque contra el obstáculo sin perder y solo si choca en el punto rojo el nivel termine, obteniendo así un comportamiento similar al del juego original.

## 2.3. Generación de bloques

Para agregar un bloque al juego se debe ingresar su posición inicial, su velocidad y su grosor, la posición inicial es un numero del 0 al 5 que representa lado de la figura central al que se acerca. Si la figura central tiene un numero de lados menor al índice de la posición la figura no se mostrara en pantalla.

La clase *Obstacle* utiliza las funciones de *GameMath* para calcular la posición de sus vértices y la variable estática *GameManager.levelCurrentSides* para calcular su largo, de esta forma los obstáculos funcionan adecuadamente para cualquier numero de caras que tenga la figura central.

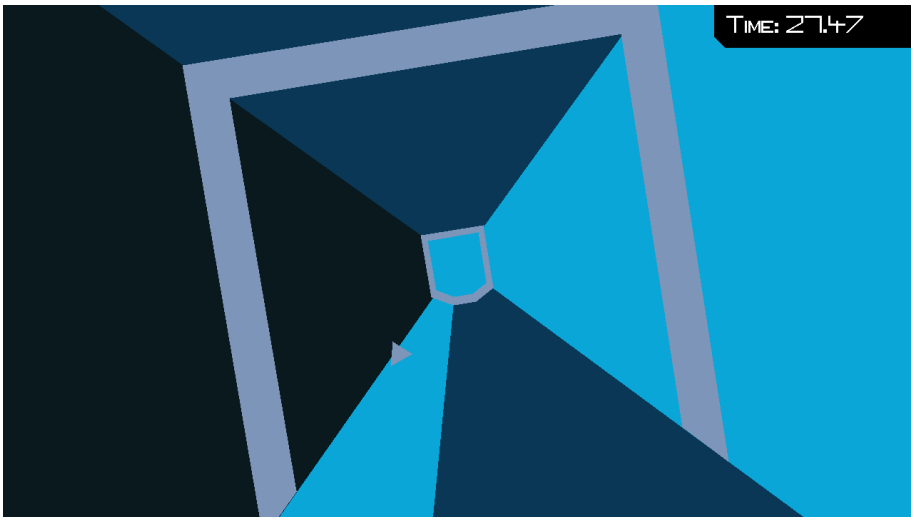


Figura 2.3: Visualización de obstáculos mientras cambia el numero de lados de la figura central

Como se observa en la figura 2.3, los bloques incluso se dibujan de manera adecuada mientras cambia el numero de lados de la figura central. Se aprovecho esto para incluir cambios de figura a entre medio de los niveles.

# 3. Contenido Extra

## 3.1. Sincronización con Musica

Una de las características principales de Super Hexagon es que todo lo que ocurre esta sincronizado con la musica, por lo tanto se creo el archivo *MusicManager.py* para sincronizar la musica con el juego. Esto se logra ingresando los BPM de la canción actual y calculando los milisegundos a los que corresponde cada tiempo. No ocurren mayores problemas de desfase si solo se usan sumas y restas para controlar los tiempos.[1]

Las clases que contiene el archivo *JuiceManager.py* están orientadas a agregar pequeños detalles al juego, estas generalmente utilizan la información del MusicManager para verse mejor. Por ejemplo la clase *CameraManager* llama a la función *on beat()* para aumentar el zoom en cada tiempo de la canción Se puede ver también en su función *update()* que la variable *last beat* nunca es asignada directamente si no que solo mediante sumas.

## 3.2. Pauta de Niveles

Los niveles además de incluir un componente de aleatoriedad incluyen una parte estructurada, a cada nivel le corresponde una instancia de *Pattern* (en el archivo *Levels.py*) que incluye:

- Tiempos en los que cambia la velocidad de la cámara.

[1]Esto se explica en mas detalle en http://ludumdare.com/compo/2014/09/09/an-ld48-rhythm-game-part-2/, en base a este post se implementó el sistema de musica.

- Momentos en los que cambia la figura central.
- Información de los obstáculos que se agregan al juego
- Paletas de colores que utiliza el nivel
- Rotacion y numero de lados de la figura al inicio del nivel.

Los diferentes *managers* del archivo *JuiceManager.py* se encargan de utilizar toda la información de los niveles.

A pesar de que los bloques se instancian según una pauta, se agregan rotaciones aleatorias para que los niveles no sean exactamente iguales pero aun así mantengan una cierta estructura.

# 4. Conclusiones

Se realizo el bonus de agregar musica al juego y se tiene un cambio constante de paleta de colores. Se comprendió la importancia del modelo MVC para organizar el proyecto y se optimizo el sistema de colisiones para que cada iteracion del bucle de juego no tarde mas de 1/60 segundos.